UNIVERSITÉ PARIS CITÉ

École doctorale 386 - Sciences Mathématiques de Paris Centre

Inria

LRGrep: Selecting Error Messages for LR Parsers

FRÉDÉRIC BOUR

Thèse de doctorat d'INFORMATIQUE

Dirigée par FRANÇOIS POTTIER

Présentée et soutenue publiquement le 18 décembre 2024

Devant un jury composé de :

LAURENCE TRATT	Professor, King's College London	(rapporteur)
JEREMY YALLOP	Associate Professor, University of Cambridge	(rapporteur)
FELIENNE HERMANS	Professor, Vrije Universiteit Amsterdam	(examinatrice)
SYLVAIN SCHMITZ	Professeur HDR, Université Paris Cité	(examinateur)
NINGNING XIE	Assistant Professor, University of Toronto	(examinatrice)
FRANÇOIS POTTIER	Directeur de Recherche, Inria Paris	(directeur)
THOMAS GAZAGNAIRE	Docteur, Entreprise Tarides	(membre invité)

Abstract

LR parsers are highly suitable for the syntactic analysis of programming languages as they guarantee linear-time recognition and can ensure the absence of grammatical ambiguity. When processing an incorrect input, an LR parser recognises the correct prefix (common with some valid sentences) but does not inherently provide additional context. While this allows an LR parser to pinpoint the position of a syntax error, it does little to explain the issue to users. Consequently, LR parsers have developed a reputation for producing brief and unhelpful error reports, and an alternative approach is often preferred when more informative feedback is necessary.

This thesis challenges that perception. In an LR parser, the grammar, through the automaton, is available as an object which can be manipulated, for example, to study incorrect sentences and refine the parser's behaviour in case of errors. We propose a declarative language for expressing error specifications that complement grammars. We develop tools to recognise these specifications and facilitate their design. Finally, we validate this approach by reproducing or improving syntax error messages for three programming languages: OCaml, Catala, and Elm.

Keywords Compilation, Parsing, LR parser, Finite automaton, Regular language, Syntax error, Error diagnosis, Reachability

Résumé

Les analyseurs LR sont une technologie de choix pour l'analyse syntaxique des langages de programmation ; notamment, ils garantissent une reconnaissance en temps linéaire et l'absence d'ambiguïté grammaticale. Lorsqu'ils traitent une phrase incorrecte, ces analyseurs reconnaissent le préfixe correct (partagé avec des phrases du langage) mais ne fournissent a priori pas d'autre information. Cela permet de pointer la position d'une erreur syntaxique mais pas d'expliquer le problème à un utilisateur. Les analyseurs LR ont donc acquis la réputation de produire des rapports d'erreur sommaires ; quand cet aspect est important, une autre approche que LR sera souvent préférée.

Cette thèse remet en question cette idée. Dans un analyseur LR, la grammaire, via l'automate, est un objet qui peut être manipulé à part entière, par exemple pour étudier les phrases incorrectes et raffiner le comportement de l'analyseur en cas d'erreur. Pour réaliser cette vision, nous proposons un langage déclaratif de spécification des messages d'erreur, complémentaire à la grammaire. Nous développons des outils pour reconnaître ces spécifications et faciliter leur conception, et nous testons la validité de cette approche en reproduisant ou en améliorant les messages d'erreur syntaxiques pour trois langages de programmation : OCaml, Catala et Elm.

Mots clés Compilation, Analyse syntaxique, Analyseur LR, Automate fini, Langage rationnel, Message d'erreur, Diagnostique, Analyse d'accessibilité

Résumé substantiel

La thèse étant rédigée en anglais, ce résumé présente l'approche et les principaux résultats en français. Il reprend les premiers paragraphes de l'introduction, puis se concentre sur les idées les plus importantes et liste les contributions.

Les analyseurs LR (Grune et Jacobs, 2008 ; Knuth, 1965) sont une solution efficace pour l'analyse syntaxique des langages de programmation. Le formalisme LR permet de décider statiquement l'absence d'ambiguïté et garantit une analyse en temps linéaire. Cela a conduit à la création de générateurs d'analyseurs performants tels que Bison (Donnelly et Stallman, 2021), Menhir (Pottier et Régis-Gianas, 2021) et LangCC (Zimmerman, 2022).

Même si LR ne peut gérer qu'un sous-ensemble des grammaires hors-contexte, il est en pratique suffisamment expressif pour gérer la plupart des langages de programmation, comme le démontrent par exemple Morbig (Régis-Gianas et al., 2020) ou DiffAST (Hashimoto, 2021). Si nécessaire, GLR (Grune et Jacobs, 2008 ; Lang, 1974 ; Tomita, 1991) étend LR à toutes les grammaires hors-contexte mais ne garantit plus ni l'absence d'ambiguïté ni la reconnaissance en temps linéaire.

Cependant, une limitation majeure des analyseurs LR a été la prise en charge des entrées syntaxiquement incorrectes. LR garantit la propriété de préfixe viable : une erreur est détectée le plus tôt possible. L'entrée est décomposée en un préfixe valide qui peut être complété pour obtenir une phrase correcte et un premier jeton invalide. Par exemple, un analyseur LR pour une grammaire d'expressions arithmétiques décomposerait l'entrée erronée « 1 + (∗2 » en « 1 + (» et « ∗ ». Le préfixe peut être complété par exemple en « 1 + (2) », qui est valide, alors qu'il n'existe aucun moyen de compléter « 1 + (∗ ». Cela suffit à signaler l'emplacement de l'erreur à l'utilisateur mais ne permet pas d'expliquer la cause de cette erreur. Les générateurs d'analyseurs courants n'offrent donc qu'une prise en charge limitée du traitement des erreurs, et les analyseurs LR ont tendance à produire des rapports d'erreur médiocres.

On propose de résoudre ce problème en inspectant l'état d'un analyseur sur le point d'échouer et en utilisant ces informations pour créer un message d'erreur. Ce travail s'appuie sur les idées introduites par Merr (Jeffery, 2002) et l'infrastructure des messages d'erreur de Menhir (Pottier, 2016). Reprenons l'exemple précédent : avant de rejeter l'entrée, l'analyseur a déjà donné une interprétation partielle à « 1+(» dans la *pile* de son automate. Il est possible de produire un message d'erreur à partir de cette pile (incluant l'état actuel de l'automate) et du jeton rejeté, une paire que l'on appelle une *configuration d'échec*. LRGrep est un langage dédié (un « DSL »)

qui permet à un programmeur de classer ces configurations et de déclencher des actions spécifiques lorsque certaines configurations sont identifiées dans le but de construire une explication pertinente de l'erreur.

Ma thèse est que les *analyseurs LR se prêtent particulièrement bien à la production de bons messages d'erreur*. Pendant la conception, l'automate LR permet de raisonner statiquement sur la grammaire et peut être utilisé pour guider l'auteur des messages d'erreur. Pendant l'exécution, la configuration de l'automate offre une représentation concrète et succincte de la situation grammaticale d'un texte en cours d'analyse, que l'on peut exploiter pour produire une explication en cas d'erreur. Cette perspective remet donc en cause l'idée reçue selon laquelle les analyseurs LR ne sont pas adaptés à une gestion fine des rapports d'erreurs. Au contraire, les grammaires LR offrent une expressivité suffisante pour reconnaître des langages de programmation pratiques et conservent de nombreuses propriétés décidables. Cela permet de spécifier déclarativement des analyseurs efficaces et facilite l'étude systématique de la structure des grammaires LR, par exemple pour raisonner sur les erreurs de syntaxe.

Classifier des situations grammaticales

Le premier outil que l'on introduit pour identifier une situation grammaticale sont les motifs *reduce-filter*, notés $[\alpha / A \to \beta \cdot \gamma]$. Ici α représente une construction grammaticale qu'il doit être possible de *réduire* dans le contexte courant ; c'est-à-dire qu'un suffixe du texte qui vient d'être analysé doit être une instance grammaticalement correcte de la construction α. Le second composant $A \to \beta \cdot \gamma$ identifie une position spécifique dans une règle grammaticale, ce que l'on appelle aussi un « item LR(0) ». Il vient *filtrer* les situations restantes à celles dans lesquelles cette règle grammaticale a été partiellement reconnue jusqu'à la position indiquée. Ainsi, le préfixe d'un texte satisfera un motif *reduce-filter* si, après analyse de ce préfixe, α est réductible et si, après réduction, le préfixe β de la règle $A \to \beta \cdot \gamma$ vient d'être reconnu. α peut être laissé vide si l'on souhaite identifier une règle sans réduire.

Illustrons ces concepts sur une grammaire des expressions arithmétiques réduite aux règles suivantes :

- $E \to T$
- $E \to T + E$
- $T \to n$
- $T \to (E)$

Une expression E est soit un terme, noté T, soit un terme plus une autre expression. Un terme est soit un entier n soit une expression entre parenthèses. Cette grammaire minimaliste traite donc seulement des séquences d'additions, éventuellement parenthésées. Voici quelques *items* de cette grammaire :

- $E \to T + \cdot E$ représente une situation où l'on vient juste de reconnaître le signe + d'une addition ; elle devra être suivie d'une expression pour que la règle s'applique.
- Dans $T \to \cdot n$, on attend un entier, dans le but de reconnaître un terme.
- Enfin, $T \to (E\cdot)$ attend une parenthèse fermante pour terminer la reconnaissance d'une expression parenthésée.

Pour spécifier les erreurs de la grammaire, on peut par exemple associer le motif $[/\, T \to \cdot n]$ au message « Ce terme est incomplet ; un entier est attendu » ou bien $[E/\, T \to (E\cdot)]$ au message « Une parenthèse fermante est manquante après cette expression ».

Dans le second motif, on a demandé explicitement la réduction de E. Pour comprendre pourquoi, considérons la phrase incomplète « (1 + 2 », pour laquelle on souhaiterait rapporter qu'une parenthèse fermante manque. L'interprétation partielle donnée à cette phrase est la pile « $(T + n$ » : 1 a été reconnu comme un terme à part entière mais l'interprétation des autres symboles n'a pas encore été fixée. Si l'on considère la pile telle quelle, le filtre $/\, T \to (E\cdot)$ échoue. Mais en demandant la réduction de E, on force à interpréter « 1 + 2 » comme une expression, pour obtenir la pile « $(E$ » qui permet au filtre de réussir.

Cette mécanique de réduction fait appel au fonctionnement interne des analyseurs LR et il n'est en fait pas nécessaire de l'exposer aux utilisateurs. Dans $/\, T \to (E\cdot)$, on peut voir qu'il sera nécessaire de réduire une expression car E est immédiatement à gauche du point. Notre outil peut donc inférer cette information ; cependant, la notation explicite que nous avons choisie décompose clairement les deux étapes de la reconnaissance d'un motif : réduire pour s'abstraire du contexte immédiat, puis filtre par *item* pour identifier des situations précises.

Appliqués à un analyseur LR, les motifs s'interprètent comme des ensembles réguliers de piles : un motif reconnaît une certaine configuration si la pile appartient à l'interprétation du motif. Pour augmenter le pouvoir de classification des motifs *reduce-filter*, on peut les généraliser en un dialecte des expressions rationnelles dont les atomes sont des symboles grammaticaux. En plus des opérateurs usuels, disjonction, concaténation et étoile de Kleene, on ajoute deux opérateurs notés $[e]$ pour *reduce*, qui reconnaît les piles dont un suffixe peut se réduire vers une séquence reconnue par e, et $/\, A \to \alpha \cdot \beta$ qui filtre les piles dans lesquelles l'item $A \to \alpha \cdot \beta$ est actif. Les motifs *reduce-filter* sont donc un cas particulier des expressions de ce dialecte. On ajoute également un opérateur $v = e$ pour lier des variables, afin de référencer des constructions grammaticales déjà reconnues. Dans l'exemple de la parenthèse manquante, cela permet par exemple de rappeler la position de la parenthèse ouvrante dans l'explication de l'erreur.

Finalement, une spécification d'erreur est composée d'une séquence de motifs et d'actions. En cas d'ambiguité, c'est-à-dire quand une pile est reconnue par plusieurs motifs, la priorité est donnée au premier motif dans l'ordre de la séquence. L'action associée au premier motif reconnu est donc utilisée pour décrire une pile. Cette approche est inspirée des analyseurs lexicaux (Paxson et al., 2007 ; Trofimovich, 2019),

dont LRGrep est conceptuellement très proche, à ceci près que le texte est remplacé par la pile d'un analyseur LR.

Contributions

Mes premières contributions concernent la classification des configurations d'un analyseur LR :

- La notion de motif *reduce-filter* pour classifier des situations grammaticales.
- Un algorithme s'appuyant sur les automates finis pour reconnaître efficacement ces motifs.
- Un dialecte d'expression rationnelle pour décrire des « spécifications d'erreurs », en généralisant ces motifs pour reconnaître et expliquer les configurations d'un analyseur LR.
- Un schéma de compilation efficace pour ce dialecte.

Je me suis ensuite intéressé à l'analyse statique des échecs d'un automate LR. Cette tâche est rendue significativement plus complexe par la notion de résolution de conflits. Proposée par l'analyseur Yacc, et toujours utilisée de nos jours, elle permet de combler certaines limites de l'analyse LR mais introduit des irrégularités qui font grossir l'espace des configurations possibles. Mes contributions sur cet aspect sont :

- Un algorithme qui permet de factoriser cet espace, accélérant de plusieurs ordres de grandeur l'analyse d'accessibilité introduite dans Pottier (2016).
- Un automate permettant d'énumérer les *configurations d'échecs*.
- Pour la conception des spécifications d'erreurs :
  - Une analyse permettant pour chaque motif *reduce-filter* de lister les *configurations d'échecs* qu'ils peuvent reconnaître, en illustrant chaque configuration par une phrase incorrecte échouant dans cette configuration.
  - Une analyse de couverture pour lister les configurations qui n'ont pas de message d'erreurs, ou bien pour garantir que toute configuration d'échec est expliquée par au moins un message.

Autrement dit, ces outils aident l'auteur d'une grammaire à gérer les erreurs en donnant des exemples de phrases incorrectes à gérer, en indiquant quels motifs pourraient gérer ces phrases et en vérifiant qu'il n'a pas oublié de cas.

Enfin, la dernière contribution est pratique : tout le travail décrit jusqu'ici a été implanté dans l'outil LRGrep qui permet de complémenter une grammaire Menhir avec une spécification d'erreur et peut assister l'auteur de la grammaire dans la conception de cette spécification. Pour valider et raffiner cette méthode, je l'ai appliquée à trois langages de programmation : OCaml (Leroy et al., 2019), Catala (Merigoux et al., 2021) et Elm (Czaplicki et Chong, 2013).

Chaque langage a demandé une approche différente :

- Peu d'attention a été donnée aux messages d'erreur de syntaxe de l'implantation de référence d'OCaml. L'objectif était d'améliorer cet état de fait. De plus, la grammaire d'OCaml utilise peu de délimiteurs : cela permet des notations très denses mais peut occasionner des erreurs subtiles, difficiles à expliquer. Les fonctionnalités avancées de LRGrep ont été conçues pour prendre en charge ces situations.
- Catala utilise l'infrastructure de messages d'erreur de Menhir. Mon travail ici a consisté à vérifier que LRGrep permettait de reproduire fidèlement son comportement, voire de l'améliorer (avec une spécification plus succincte et lisible).
- Enfin, Elm s'appuie sur un analyseur syntaxique écrit à la main, utilisant des combinateurs spécialement conçus pour permettre la gestion fine des erreurs de syntaxe. Elm est reconnu parmi ses utilisateurs pour le soin porté aux messages d'erreur. Il a fallu écrire une grammaire LR pour un sous-ensemble réaliste de Elm et une spécification d'erreurs reproduisant les explications du compilateur de référence.

Dans le cas de Elm, l'écriture de la spécification a été en grande partie automatisée en utilisant la grammaire LR comme guide pour la génération de phrases incorrectes et en utilisant le compilateur Elm de référence pour les expliquer. Cela a démontré que les motifs *reduce-filter* sont des bons classificateurs d'erreur et qu'une spécification déclarative pouvait reproduire finement le comportement d'un analyseur écrit à la main avec des messages d'erreur à l'état de l'art.

À la mémoire de mon père,
à son humour, sa gentillesse et son esprit passionné.

Remerciements

Ce travail n'aurait pas pu voir le jour sans l'immense soutien et les encouragements de nombreuses personnes.

Je tiens d'abord à exprimer ma profonde gratitude envers ma famille. Kiko, mon épouse, a été un pilier constant pendant ces années, me soutenant avec amour et patience face aux défis du doctorat. Mes parents, Marie-Noëlle et Roger, m'ont transmis la passion pour l'étude et la recherche. Mon père nous a malheureusement quittés en 2020, juste au début de cette aventure. Sa présence, son esprit curieux et la confiance qu'il m'accordait me guident encore aujourd'hui.

Mes amis ont également joué un rôle essentiel dans ce parcours. Un merci particulier à Thomas Refis, Simon Castellan, Ulysse Gérard et Gabriel Scherer, qui m'ont encouragé à poursuivre le chemin du doctorat et à avoir confiance dans mon travail. Arthur Saiz m'a appris à penser différemment, en dehors des conventions, et à affirmer nos propres convictions. Vincent Bernardoff et Diego Pons m'ont montré la richesse qu'il y a à connecter des domaines variés, me poussant à persévérer malgré les difficultés.

Marie-Hélène et Olivier Chiarisoli m'ont offert un soutien précieux en dehors du monde académique, notamment dans les moments difficiles. Leur amitié a été une bouffée d'air frais et de bienveillance.

Je tiens également à remercier toutes les personnes qui ont contribué à forger mon bagage technique et théorique. Akim Demaille, en particulier pour son introduction à la théorie des langages, Yann Régis-Gianas, pour m'avoir fait découvrir le fascinant monde de la recherche en informatique. Enfin, mon directeur de thèse, François Pottier, pour sa patience, son écoute, et pour n'avoir jamais douté de mon travail, même lorsque je me perdais.

Bien sûr, tout cela n'aurait pas été possible sans les pionniers du langage OCaml, qui ont démontré qu'une approche différente de la programmation était possible : à l'Équipe Cambium/Gallium à Paris (Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Remy, ...) et au Laboratoire OCaml Labs de l'Université de Cambridge.

En dernier lieu, je ne peux oublier Thomas Gazagnaire, Jan Midtgaard et, plus largement, les équipes de Tarides et Jane Street, qui ont soutenu ce projet techniquement et financièrement, ont cru en mes idées et m'ont laissé explorer des chemins inhabituels.

Contents

Introduction

LR parsing (Grune and Jacobs, 2008; Knuth, 1965) is an efficient technology for implementing programming language parsers. The LR formalism enables static testing of non-ambiguity and guarantees linear-time analysis. This led to the creation of parser generators producing high-performance parsers, such as Bison (Donnelly and Stallman, 2021), Menhir (Pottier and Régis-Gianas, 2021), and LangCC (Zimmerman, 2022).

A significant challenge of LR parsers has been the generally poor support for dealing with invalid inputs. The LR formalism guarantees the viable-prefix property: an error is detected as early as possible. The input is decomposed into a valid prefix that can be completed to get a correct sentence and the first invalid token. For example, an LR parser for a simple grammar of arithmetic expressions would break the erroneous input "`1+(*2`" into "`1+(`" and "`*`". The former can be completed to, e.g., "`1+(2)`" while there is no way to complete "`1+(*`". This is sufficient to report the location of the error to the user but not to explain the cause of this error. Most mainstream LR parser generators provide limited support for dealing with errors, and as a result, typical implementations of LR parsers tend to produce poor error reports.

Our research attempts to address this problem by providing a framework for analysing the state of a failing parser and using this information to craft an error message. We build upon the work of Merr (Jeffery, 2002) and the error-message infrastructure of Menhir (Pottier, 2016). Continuing with our example, before rejecting the input, the parser has already given a partial interpretation to "`1+(`" in the form of the *parsing stack*. We propose to generate an error message from these pieces of information, the stack (including the current state) and the rejected token, which we call a *failing configuration*. We design a domain-specific language (DSL) that allows the programmer to classify these configurations and take action when specific ones are identified.

In short, our thesis is that *LR parsers offer a unique opportunity to improve error message quality by exploiting their ability to track grammatical state and context.* In the continuation of Pottier (2016), this perspective further challenges the common belief that LR parsers are unsuitable for high-quality error handling. LR grammars occupy a distinctive niche: they possess sufficient expressiveness to accommodate practical programming languages while retaining numerous decidable properties. This duality enables the development of efficient parsing algorithms and facilitates the analysis of the structure of LR grammars. As we demonstrate, it becomes feasible to reason about syntax errors in a manner independent of a specific implementation.

$$
\begin{array}{lcl}
E & \to & T \\
E & \to & T \texttt{ + } E \\
T & \to & F \\
T & \to & F \texttt{ * } T \\
F & \to & i \\
F & \to & \texttt{(} E \texttt{)}
\end{array}
$$

Figure 1: Rules for a simplified arithmetic language

A minimalist example Consider the grammar for a subset of arithmetic expressions with the rules given in Figure 1. The terminals are `+`, `*`, `(`, `)` and i, representing an integer literal. The nonterminals are E, T and F (for expression, term, and factor). The starting symbol is E. The nonterminals represent the different priority levels of an arithmetic expression, interpreting + and $*$ as right-associative.

We provide the grammar with an *error specification* such as:

```
/F → ·i          { "Expecting an integer" }
[E/F → (E·)]     { "Expecting a closing parenthesis" }
```

This specification is composed of two clauses, made up of a pattern and a semantic action, and gives an interpretation to some parsing failures. If the LR parser rejects an input, the patterns are compared against the parser's stack to determine if a clause applies. If several clauses apply, the first one, in the order of the text, is retained, and its semantic action is evaluated. The pattern language is a dialect of regular expressions extended with two operators, *filter* and *reduce*. To understand these examples, let us put aside the regular expression part and focus on the new operators.

The first clause handles failures in a situation where it would have been possible to read an integer. $/A \to \alpha \cdot \beta$ is the *filter* operator and is parameterised by an *LR(0) item*. An LR(0) item represents a particular position within a grammatical rule by using a dot to separate what has already been recognised from a potential continuation. The operator recognises the stacks where the item appears at the top. Thus $/F \to \cdot i$ succeeds when the parser is about to recognise an i as part of the rule $F \to i$. The filter operator is used to categorise situations according to their direct context.

The second clause uses a similar logic to filter situations in which a closing parenthesis is expected, but only after *reducing* the top of the stack to E. If a canonical LR parser for our arithmetic expressions tries to recognise "`(1+2`", it will fail with a stack containing "`(T+i`". The number 1 is *reduced* to a T, as part of the $E \to T$+E rule, but 2 is still represented by an i. The parser needs to know what comes next before reducing it: if it is followed by $*$, it will be reduced to an F; if it is a +, it will be further reduced to a T. More generally, when an LR parser encounters an error, the details necessary for explaining the situation may be buried deep within the stack. Directly matching the stack can be impractical, but reducing a few selected rules often proves sufficient to surface the information needed for producing an appropriate error message. The second operator of our pattern language is $[p]$, or *"p modulo reduction"*. It matches if there is a sequence of reductions applicable on the current stack that

produces a suffix matching p. Back to our example, the sub-pattern $E\ /F \to (E\cdot)$ matches a suffix made up of an expression E that satisfies the filter operator. The complete pattern $[E\ /F \to (E\cdot)]$ can be read as "*match stacks which suffix can be reduced to an expression appearing between parentheses*"[1].

These two patterns belong to a subset of our DSL which we call the *reduce-filter* patterns. They have the form $[\alpha\,/\,A_1 \to \beta_1\cdot\gamma_1 \ldots /\,A_i \to \beta_i\cdot\gamma_i]$ and they recognise situations in which a suffix of the stack can be reduced to the sentential form α, leading to a new stack in which the items $A_1 \to \beta_1\cdot\gamma_1$ to $A_i \to \beta_i\cdot\gamma_i$ appear. When no reductions are needed, as in the first example, one can set α to ϵ, and even omit the brackets and directly write $/\,A_1 \to \beta_1\cdot\gamma_1 \ldots /\,A_i \to \beta_i\cdot\gamma_i$. This subset plays a central role in the DSL and is the main topic of Chapter 2.

Let us go back to the sample error messages. In our example they are simple strings but in a more realistic setting, they will be actual code that refers to semantic values or to source code locations. For instance, the second clause could point out the location of the unclosed left parenthesis in the error message. As for the wording of the messages, one could object that they are only half-truths: the explanations are valid but focus on a single possible continuation. When an integer is expected, it is also valid to input a parenthesised expression. We focused on this case as we felt that this was the most common way to phrase error messages: suggesting the most likely explanation, according to the author of the error messages, and not flooding the user with an exhaustive report. The formalism extends naturally to offering multiple explanations—by collecting all matching messages instead of stopping at the first one.

However, some situations can deserve more specific error messages. Hence our choice of using the text order for ranking clauses: when multiple clauses apply, the first one is chosen. Error specification for realistic grammars tend to start by catching highly specific errors using rich patterns, and progressively fall back to generic ones; possibly ending at the uninformative "Syntax error" in the worst case.

Here is an excerpt of the error specification for the OCaml language from our reference implementation:

```
| item=structure_item; [let_bindings(ext)] @ IN
  {
   "A 'let ... in' expression is not allowed immediately after a definition.
    This can be solved by inserting ';;' or 'let () = ' after " ^
    print $endloc(item)
  }
```

This clause catches an incorrect use of `let`-bindings. The `let` keyword introduces two different constructions in the OCaml language:

- **let** *pattern* = *expression* for a module-level definition,

[1] The pattern can appear redundant because the prefix E is specified twice; first as part of the suffix to reduce to, then for the filter. We choose to stick with this notation to simplify the interpretation: both parts can be given an independent interpretation that compose as expected. Our implementation offers syntactic sugars to hide those redundancies when desired.

- **let** *pattern* = *expression* **in** *expression* for a local definition.

Things get confusing because a local definition is allowed in a module, but only if it comes first, or if it is separated from the previous definition by `;;`. Otherwise, the parser fails when reaching the **in** keyword. The pattern detects this situation by:

1. looking for a `structure_item`, the nonterminal for module-level definitions,
2. followed by a construction that reduces to a `let_bindings(ext)`, the nonterminal recognising **let** *pattern* = *expression*,
3. and fails when looking ahead at `IN`, the terminal for the **in** keyword.

The concepts used in this pattern are formally presented in Chapter 3, while Chapter 7 introduces the reference implementation and the concrete syntax.

To conclude, we would like to point out a symmetry in the three examples so far: the pattern and the message mention the same elements, once using formal constructions and once in text. The integer appears as i in $F \to \cdot\, i$, the parentheses in $F \to (\, E \cdot\,)$, and the OCaml example clearly delimits the *definition*, the *let binding* and the **in**. A good pattern naturally tends to mirror the explanations given for an error.

Contributions This work introduces a domain-specific language (DSL) for identifying and describing failing configurations of LR parsers. The DSL empowers developers to clearly articulate the expected behaviour in the face of invalid input using a declarative specification.

A standalone program implements this DSL, compiling specifications into code and integrating with Menhir, the leading LR parser generator for OCaml. To facilitate the authoring of error messages, a suite of tools is provided. An instrumented interpreter for LR parsers produces annotated backtraces to understand failing configurations. A static analyser identifies uncovered cases (invalid sentences lacking an error message) and can generate comprehensive test suites to rigorously test the failure paths of LR parsers. To make this practical, we developed a new reachability analysis for LR parsers that increases performance by two orders of magnitude over the current state of the art.

This dissertation introduces the DSL, its implementation, the new approach to reachability and the algorithmic challenges encountered, and presents empirical findings obtained by applying this methodology to the parsers of three full-featured programming languages.

Limitations This study explores how LR parsers handle invalid input with the goal of enabling users of LR parser generators to produce effective error messages. Our research is experimental, focusing on the development and evaluation of reduce-filter patterns as a novel approach to improving error handling in compilers and related tools.

While we have successfully implemented this technique and demonstrated its practical utility through case studies involving realistic grammars, our work has several limitations. Notably, we do not provide formal proofs for our claims, which should be considered conjectures rather than theorems. Furthermore, our investigation is narrowly focused on extracting relevant information from LR parsers, not looking at the broader question of what constitutes a good error message nor how it should be communicated. This topic has been extensively studied in industry with notable examples including the C/C++ frontend of Clang and GCC or the Elm compiler, and in academia, where decades of experience have been collected and summarised in the major survey of Becker et al. (2019). For further insights, interested readers may refer to Barik et al. (2018), which provides hints on the argumentative structure of error messages, and Wrenn and Krishnamurthi (2017), for evaluating the quality of error messages.

Our work complements these efforts by providing a technical and systematic solution to syntax error handling for LR parsers. We validate the approach by showing that (1) we can reproduce the high-quality error messages of handwritten parsers (Elm), and (2) our technique extends to intricate errors that require relating information wide apart (with OCaml examples).

Organisation The rest of the document is structured as follows:

- Chapter 1 briefly introduces the main concepts used throughout the thesis, namely context-free grammars, LR parsers and regular automata, and the approaches to error-handling pioneered by Merr and Menhir.
- Chapter 2 focuses on the reduce-filter patterns. We introduce and formalise them on context-free grammars and we show how to recognise them efficiently on LR automata.
- Chapter 3 generalises these patterns into a richer calculus with stronger classification power. We give a formal semantics for this calculus and define a derivative operator, showing that the calculus defines a class of regular languages.
- In Chapter 4, we compile this calculus to a variant of deterministic finite automata suitable for a performant implementation.
- Chapter 5 studies the reachability problem for LR parsers, introducing a novel and efficient solution.
- Chapter 6 uses reachability to develop two static analyses: an enumerator of failing configurations capable of (finitely) covering all reduce-filter patterns and suitable for fuzzing existing parsers, and an exhaustiveness checker to find failing configurations not covered by a specification.
- Chapter 7 reports our experience with the LRGrep tool, an implementation of the calculus. It also covers the features and implementation details that do not affect the theory but which we found important in practice.

- Chapter 8 applies LRGrep to OCaml, Catala, and Elm programming languages.

Chapter 5 was published separately in Bour and Pottier (2021). The approach described has been implemented and upstreamed into Menhir, improving over the previous implementation by two orders of magnitude both in time and space. All the other techniques we present have been implemented and tested in the LRGrep tool[2].

[2] https://github.com/let-def/lrgrep

Chapter 1

Background on grammatical analysis and LR parsing

This chapter introduces the theoretical foundations on which LRGrep is built: in particular, regular and context-free languages, and the tools for recognising and manipulating them—derivations, finite automata, Shift-Reduce parsers and LR automata. When appropriate, the definitions are adapted to better fit the needs of LRGrep.

A second part of the chapter reviews the existing approaches for handling error messages with an LR parser.

1.1 Notation for sequences

We write:

- ϵ for the empty sequence (or word),
- X^* for the set of sequences of elements of some set X,
- $|u|$ for the length of a sequence,
- u_i, with $0 \leq i < |u|$, for the i-th element of the sequence, starting from 0,
- $u \cdot v$, or uv if it is not ambiguous, for the concatenation of u and v, e.g. $(uv)_i = u_i$ if $i < |u|$, $v_{i-|u|}$ otherwise,
- $||u||$ for the set of elements of u, e.g., $||\epsilon|| = \emptyset$ and $||x \cdot u|| = \{x\} \cup ||u||$.

When no other notation has been established for a set X, we write $\vec{x}$ for an arbitrary element of X^*. We assume the following functions on sequences:

- head(u) is the leftmost element of u, e.g. head(abc) = a,
- tail(u) is the sequence u without its leftmost element, e.g. tail(abc) = bc,
- top(u) is the rightmost element of u, e.g. top(abc) = c,
- pop(u) is the sequence u without its rightmost element, e.g. pop(abc) = ab.

1.2 Regular languages and finite automata

Given an alphabet Σ, a regular language is a subset of words $W \subset \Sigma^*$ that can be recognised by a finite automaton. W is the language of the automaton.

1.2.1 Deterministic finite automaton

A deterministic finite automaton (DFA) is given by a tuple $M = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states,
- Σ is the alphabet, a finite set of symbols,
- $\delta : Q \times \Sigma \rightharpoonup Q$ is the partial transition function,
- $q_0 \in Q$ is the initial state,
- $F \subset Q$ is the set of final or accepting states.

Finite automata are used to recognise sets of strings. To check if a string w is recognised, one starts from the initial state q_0 and follows the transitions labelled by the letters of w, going through a sequence of states $q_i = \delta(q_{i-1}, w_i)$: $q_1 = \delta(q_0, w_1)$, $q_2 = \delta(q_1, w_2)$, etc. The string is recognised if, when reaching the end of the word, the state is final: $q_{|w|} \in F$. This simple procedure is amenable to efficient implementations, making finite automata popular devices for recognising regular languages.

For convenience, we lift δ to operate on words: $\delta(q, a \cdot w) = \delta(\delta(q, a), w)$ and $\delta(q, \epsilon) = q$. A word w is recognised if and only if $\delta(q_0, w) \in F$. The language recognised by the automaton is $\mathscr{L}(M) = \left\{ w \in \Sigma^* \middle| \delta(q_0, w) \in F \right\}$.

We allow δ to be a partial function—$\delta(q, a)$ can be undefined when no continuation of the prefix would be recognised by the automaton. This is equivalent to having a distinguished *sink* state $q_\perp \notin F$ looping on itself: $\forall b \in \Sigma, \delta(q_\perp, b) = q_\perp$, and to which we throw undefined transitions like $\delta(q, a) = q_\perp$. Operationally, an undefined transition can be seen as an early-exit.

1.2.2 Non-deterministic finite automaton

Non-deterministic finite automata (NFA) generalise DFA by allowing transitions to target multiple states. In this work, we use a flavour of ϵ-NFAs defined by tuples $M = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is the finite set of states,
- Σ is the alphabet, a finite set of symbols,
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathscr{P}(Q)$ is the transition function,
- $q_0 \in Q$ is the initial state,

- $F \subset Q$ is the set of final or accepting states.

Compared to the DFA introduced above, the transition function can target zero, one, or more states, and there are transitions labelled ϵ which represent states reachable without consuming an input symbol.

An NFA also describes a set of strings although the operational interpretation is more involved than in the DFA case since we have to keep track of a set of active states rather than a single state. In return, they often allow for a more compact representation of sets of strings—in the worst case, a DFA can be exponentially larger than an equivalent NFA.

A string w is recognised by an NFA if and only if $\delta^*(I, w) \cap F \neq \emptyset$ where:

$$\begin{aligned} \mathit{closure}(Q') &= Q' \cup \bigcup_{q \in Q'} \mathit{closure}(\delta(q, \epsilon)) \\ \delta^*(Q', a) &= \bigcup_{q \in \mathit{closure}(Q')} \delta(q, a) \\ \delta^*(Q', \epsilon) &= \mathit{closure}(Q') \\ \delta^*(Q', a \cdot w) &= \delta^*(\delta^*(q, a), w) \end{aligned}$$

In three steps:

1. *closure* is the ϵ-closure, completing a set of states with all the states reachable by following ϵ-transitions,
2. $\delta^* : \mathscr{P}(Q) \times \Sigma \to \mathscr{P}(Q)$ extends δ to operate on the closure of a set of states,
3. $\delta^* : \mathscr{P}(Q) \times \Sigma^* \to \mathscr{P}(Q)$ lifts the previous definition to operate on a word rather than on a single symbol.

In this work, we use NFAs as intermediate tools for reasoning about properties of sets of strings, eventually producing a DFA to recognise specific strings. The *subset construction* (Rabin and Scott, 1959) is the standard algorithm for transforming an NFA into a DFA.

1.2.3 Notations for automata, graphs and transitions

When it is not ambiguous, we identify an automaton and the set of its states. E.g., we call Q the automaton $(Q, \Sigma, \delta, q_0, F)$. We use the notation $q \xrightarrow{s} q'$ for a transition from q to q' labelled by s: $\delta(q, s) = q'$.

We extend this notation to graphs and edges, identifying a graph (V, E) with the set of its vertices V (we say *"a path in V such that…"*) and write $v \to v'$ for an unlabelled edge or $v \xrightarrow{l} v'$ for a labelled edge.

When multiple automata or graphs are being considered, we write their name as a subscript of the arrow to avoid ambiguities. E.g., *"if $q \xrightarrow{s}_Q q'$ and $v \xrightarrow{s}_V v'$, then…"*

1.3 Regular expressions and their derivatives

Regular expressions $e, e_1, e_2 \in E$ denote sets of strings over an alphabet Σ. They are built using the following operators:

- $\emptyset$, which denotes the empty set,
- ϵ, which denotes the empty string,
- a literal character $a \in \Sigma$, which denotes the string made of this single character,
- the concatenation $e_1 \cdot e_2$, which denotes strings that are a concatenation of a string denoted by e_1 with a string denoted by e_2,
- the disjunction $e_1|e_2$, which denotes strings denoted by e_1 or by e_2,
- the Kleene star e^*, which denotes the repetition of zero, one or more instances of strings denoted by e.

The denotation or language $\mathscr{L}(e)$ of an expression e is the set of strings that is the least solution to these equations:

$$\begin{aligned}
\mathscr{L}(\emptyset) &= \emptyset \\
\mathscr{L}(\epsilon) &= \{\epsilon\} \\
\mathscr{L}(a) &= \{a\} \\
\mathscr{L}(e_1 \cdot e_2) &= \{u \cdot v \mid u \in \mathscr{L}(e_1), v \in \mathscr{L}(e_2)\} \\
\mathscr{L}(e_1|e_2) &= \mathscr{L}(e_1) \cup \mathscr{L}(e_2) \\
\mathscr{L}(e^*) &= \mathscr{L}(\epsilon|e \cdot e^*)
\end{aligned}$$

1.3.1 Connection to finite automata

To efficiently decide whether an input is recognised by a regular expression, it is common to translate the expression into a finite automaton. The standard approach among lexer generators is to use a variant of Thompson (1968) or Glushkov (1961) constructions to generate a non-deterministic automaton from a regular expression; also treated in Berry and Sethi (1986). The automaton is then determinised and sometimes minimised before further use. These techniques have continued to receive a lot of attention in modern lexer generators or regular expression recognisers, such as the traditional flex (Paxson et al., 2007) and the modern alternatives RE2C (Trofimovich, 2019), rust-regex[1], or re2[2].

Our use case is very similar to that of a lexer generator where the text is replaced by the stack of an LR parser (presented in 1.6). However, the standard techniques are difficult to apply directly to LRGrep patterns, mainly because of the need to reason on reductions.

[1] https://blog.burntsushi.net/regex-internals/
[2] https://github.com/google/re2

Instead, we opted for an approach based on regular expression derivation. Derivation gives a direct definition of the transition function, without explicitly constructing the automaton. From a regular expression e and a character a, the derived expression $d(e,a)$ recognises the suffixes of words in $\mathscr{L}(e)$ beginning with a; in other words, what remains to match after seeing a. Matching a single word can be done by deriving with respect to each character in sequence. An automaton can be constructed by closing over the derivatives—deriving each expression with respect to each character, until no new expressions are discovered. Each expression is a state of the automaton, and the derivative is the transition function. This formalism is easy to extend to other operators, as studied by Thiemann (2016). This ease of extension was confirmed by our experience in adding two custom operators, reduce and filter.

Though less popular than Thompson-like approaches, derivation has been successfully used for the translation of regular expressions into automata (Moseley et al., 2023; Owens et al., 2009). Derivation also shines when formally reasoning about properties of regular languages (Coquand and Siles, 2011; Midtgaard et al., 2016).

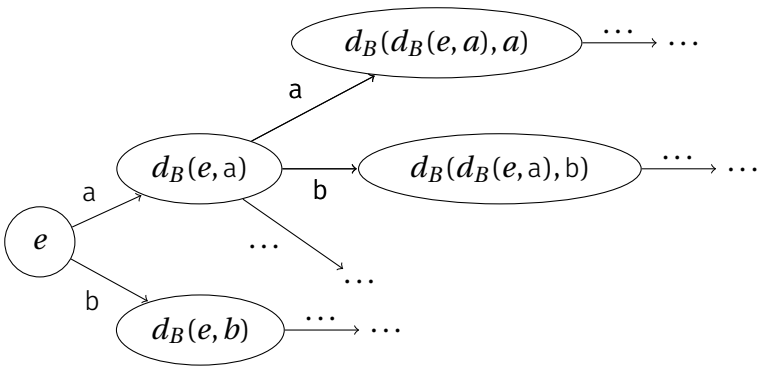
1.3.2 Regular expressions derivatives

The notion of derivation for languages and regular expressions was introduced by Brzozowski (1964) and has been adapted to different contexts since. Our work builds upon the variant introduced in Antimirov (1996). In this section, we recall their definitions and point out the benefits of Antimirov's variant in our setting.

Brzozowski's derivative The Brzozowski derivative $d_B(S,a)$ of a set of strings S with respect to a character a is the set $\{u | a \cdot u \in S\}$: the remainders of strings in S that begin with a. For instance, $d_B(\{\texttt{apricot},\texttt{banana},\texttt{berry}\},\texttt{b}) = \{\texttt{anana},\texttt{erry}\}$.

Brzozowski generalised the derivative to regular expressions, such that the derivative of the language is the language of the derivative: $d_B(\mathscr{L}(e),a) = \mathscr{L}(d_B(e,a))$. Furthermore, Brzozowski proved that, if enough equivalent expressions are identified, the derivatives of a regular expression are finite.

This gives an incremental way of constructing a DFA recognising the language of an expression e: the states are the regular expression e and its derivatives, the transition function is d_B.



The base definition of derivatives can be conveniently extended to new operations; for instance, the intersection and complementation of regular expressions have a natural definition using derivatives.

On regular expressions, d_B is the least function satisfying:

$$\begin{array}{rcl}
d_B(\emptyset, a) & = & \emptyset \\
d_B(\epsilon, a) & = & \emptyset \\
d_B(a, a) & = & \epsilon \\
d_B(b, a) & = & \emptyset, \text{ if } \quad b \neq a \\
d_B(e_1.e_2) & = & \begin{cases} d_B(e_1, a).e_2 \mid d_B(e_2, a), & \text{if } c(e_1) \\ d_B(e_1, a).e_2, & \text{otherwise} \end{cases} \\
d_B(e_1|e_2, a) & = & d_B(e_1, a) \mid d_B(e_2, a) \\
d_B(e^*, a) & = & d_B(\epsilon|e.e^*, a)
\end{array}$$

c is an auxiliary definition to determine whether ϵ is in the denotation of a regular expression. If it is the case, the expression is said to be *nullable*. It is defined as:

$$\begin{array}{rcl}
c(\emptyset) & = & \bot \\
c(\epsilon) & = & \top \\
c(a) & = & \bot \\
c(e_1.e_2) & = & c(e_1) \wedge c(e_2) \\
c(e_1|e_2) & = & c(e_1) \vee c(e_2) \\
c(e^*) & = & \top
\end{array}$$

A significant complication is the need to identify some semantically equivalent regular expressions, which is mandatory for regular expressions to have finite derivatives. For instance, the following identities of operators can be considered:

- associativity, commutativity and idempotence (ACI) of the disjunction,
- associativity of the concatenation,
- idempotence of Kleene-star, ...

While the first identities are necessary for the derivatives to be finite, the stronger the identification, the closer the DFA will be to a minimal one. Owens et al. (2009) presents an elegant functional implementation enforcing many identifications using smart constructors, though we could not get it to perform efficiently. The identities considered do not appear to be sufficient to efficiently handle certain patterns[3] and that made it difficult to extend the derivative to custom operators.

Antimirov's derivative Antimirov (1996) introduces the non-deterministic counterpart to Brzozowski's derivative, which we note $d_A(e, a)$. It is such that $d_B(\mathscr{L}(e), a) =$

[3]We observed combinatorial explosions in certain situations. In private communications with Andy Chu, we found that this is also the case with known problematic expressions such as $\texttt{a?}^n\texttt{a}^n$ from https://swtch.com/~rsc/regexp/regexp1.html.

$\bigcup \{ \mathscr{L}(e') \mid e' \in d_A(e,a) \}$. Where Brzozowski's derivative produces a single expression, Antimirov's derivative produces a set of expressions. A single expression can be recovered by taking the disjunction of all of those; thus, this is equivalent to identifying expressions modulo ACI of top-level disjunctions.

Just like d_B can be seen as the transition function of a DFA, Antimirov's derivative d_A is the transition function of an ϵ-free NFA. An advantage of Antimirov's derivative is that it is no longer needed to identify regular expressions modulo identities. Instead, they can be considered syntactically which simplifies the implementation.

It is defined as the least solution to:

$$
\begin{array}{rcll}
d_A(\emptyset, a) & = & \emptyset & \\
d_A(\epsilon, a) & = & \emptyset & \\
d_A(a, a) & = & \{\epsilon\} & \\
d_A(b, a) & = & \emptyset, \qquad b \neq a & \\
d_A(e_1|e_2, a) & = & d_A(e_1, a) \cup d_A(e_2, a) & \\
d_A(e_1.e_2, a) & = & \begin{cases} \{e_1'.e_2 \mid e_1' \in d_A(e_1, a)\} \cup d_A(e_2) & \text{if } c(e_1) \\ \{e_1'.e_2 \mid e_1' \in d_A(e_1, a)\} & \text{otherwise} \end{cases} & \\
d_A(e^*, a) & = & d_A(\epsilon|e.e^*, a) &
\end{array}
$$

We base our implementation on Antimirov's derivative. Since it works directly on the syntax, it is easier to extend to new operators and, in our application, there are no benefits in immediately getting a DFA over an NFA.

1.4 Context-free grammars

A context-free grammar (Chomsky, 1956; Grune and Jacobs, 2008) is a tuple $G = (N, T, R, S)$. N is the set of nonterminals and T is the set of terminals. $R \subset N \times (N \cup T)^*$ is the set of rules. N, T and R are finite. Members of $N \cup T$ are called symbols. $S \in N$ is a distinguished nonterminal called the *start symbol*. By convention, we use $A, B \in N$ for arbitrary nonterminals, $a, b \in T$ for arbitrary terminals, $x \in N \cup T$ for a symbol, $\alpha, \beta \in (N \cup T)^*$ for sequences of symbols and $w \in T^*$ for a sequence of terminals, and we write $A \to \alpha$ for $(A, \alpha) \in R$. A sequence of symbols is called a *sentential form*, and a sequence of terminals a *sentence*.

Example (arithmetic grammar) The arithmetic grammar from the introduction (Figure 1) is formally defined by $G_A = (N_A, T_A, R_A, E)$, where:

- $N_A = \{E, T, F\}$
- $T_A = \{+, \star, (,), i\}$
- $R_A = \{(E, T), (E, T+E), (T, F), (T, F\star T), (F, i), (F, (E))\}$

1.4.1 Generating sentences

The language *generated* by a grammar G is written $\mathscr{L}(G)$, and is constructed using the derivation relation $\Rightarrow$. We write $\Rightarrow^*$ for the reflexive and transitive closure of $\Rightarrow$.

Definition 1 (Derivation relation) *The derivation relation $\Rightarrow$ is the least relation such that $\alpha A\gamma \Rightarrow \alpha\beta\gamma$, for all sequences of symbols α and γ and rule $A \to \beta$.*

A sentential form or a sentence derived from the start symbol is said to be *valid*. Conversely, a sentential form or a sentence is *invalid* if it cannot be derived from the start symbol[4].

Definition 2 (Valid sentential forms and sentences) *A sentential form α is valid iff $S \Rightarrow^* \alpha$. A sentence w is a valid iff $S \Rightarrow^* w$.*

Definition 3 (Language generated by a grammar) *The language of a grammar is the set of all valid sentences:*

$$\mathscr{L}(G) = \left\{ w \in T^* \mid S \Rightarrow^* w \right\}$$

Example (Arithmetic derivations) Deriving some valid sentences of G_A:

- $E \Rightarrow T \Rightarrow F \Rightarrow n$
- $E \Rightarrow T + E \Rightarrow T + T \Rightarrow F + T \Rightarrow F + F \Rightarrow i + F \Rightarrow i + i$
- $E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (T) \Rightarrow (F) \Rightarrow (i)$

Observe that a sequence of derivations can be read in both directions: from left to right as a device for generating valid sentences, and from right to left as a proof that a sentence is valid.

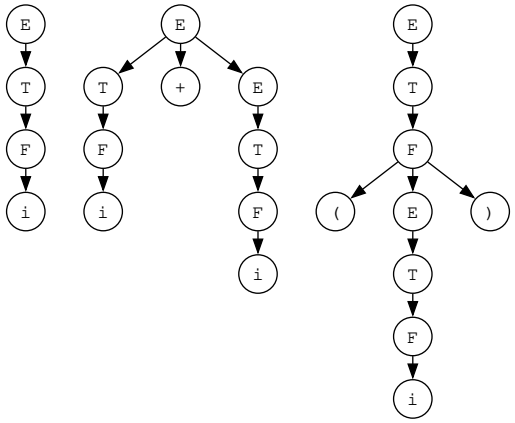
1.4.2 Giving structure to sentences

The derivation relation substitutes a nonterminal A with a sequence α, for some rule $A \to \alpha$. In a derivation sequence, nonterminals are expanded until eventually reaching a sentence. The nesting of the successive substitutions is not explicitly represented in the derivation sequence. However, the nesting structure usually reflects semantic properties of the language. This permits giving compositional semantics to the syntax: each rule is interpreted independently, and those interpretations are composed. In practice, it is important for a parsing algorithm to do more than just decide that a sentence belongs to the language; it must also recover this structure. This is done with a *derivation tree*.

Definition 4 (Derivation tree) *The derivation tree is a tree with nodes labelled by symbols. The root of the tree is labelled by the start symbol. A node labelled by a nonterminal A has n children labelled $x_1 \ldots x_n$ iff there exists a rule $A \to x_1 \ldots x_n$. A node labelled by a terminal is a leaf.*

[4]Some works reserve the wording "sentential form" and "sentence" for valid inputs, using "sequence of symbols" for uncertain or invalid ones. Since we are concerned with the detection and classification of invalid ones, it is more convenient for us to use sentential forms and sentences from the start, and to qualify them after determining their validity.

The sentences above correspond to the following derivation trees:



A grammar is unambiguous if there is a single derivation tree per sentence.

1.4.3 Cyclic grammars

A grammar is cyclic if there is a nonterminal A such that $A \Rightarrow^+ A$.

For instance, if a rule $T \rightarrow E$ was added to grammar G_A, it would be possible to derive $E \Rightarrow T \Rightarrow E$. Such a sequence can be repeated an arbitrary number of times, leading to sentences admitting an infinite number of derivations. Rules with an empty right-hand side, called ϵ-rules, can make the matter more complex by hiding the cycle. Adding rules $E \rightarrow PE$ and $P \rightarrow \epsilon$ allows sequences like $E \overset{\text{rm}}{\Longrightarrow} PE \overset{\text{rm}}{\Longrightarrow} PPE \overset{\text{rm}}{\Longrightarrow} PPPE \overset{\text{rm}}{\Longrightarrow} \ldots$, which do not immediately appear to be cyclic. See Nederhof and Sarbo (1996) for a detailed treatment of *hidden left recursion*.

We assume that the grammars and the LR automata processed by LRGrep contain no such cycles. In particular, the construction of the reduction automaton in Chapter 2 diverges in the presence of cycles.

This is not a problem in practice since cycles can be detected and removed. Furthermore, cyclic rules do not present any practical interest and are likely the result of a mistake in the specification of a grammar (they do not change the language recognised, they only add new derivations). In Grune and Jacobs (2008), section 2.9 *Hygiene in Context-Free Grammars* gives algorithms for detecting and cleaning up problematic grammars. In the Menhir parser generator, the "LoopDetection" module is designed to reject these grammars while explaining the problematic constructions to grammar authors.

1.5 Shift-Reduce parsers

Parsing algorithms construct a derivation sequence from a given input sentence. LRGrep is specifically designed for LR parsers, a subclass of Shift-Reduce parsers, but the underlying techniques are applicable to Shift-Reduce parsing methodologies in general. In Chapters 2 and 3, we follow this dual perspective when introducing

and formalising the DSL: first with a high-level view that manipulates grammatical constructions and is applicable to all Shift-Reduce parsers, and then with a concrete presentation that leverages the LR automaton and is suitable for an efficient implementation.

Shift-Reduce parsers operate on a restricted notion of derivation called *rightmost derivation*. The derivation relation $\Rightarrow$ is too loose for strict parsing concerns: a given sentence generally admits multiple derivation sequences, even if the grammar is unambiguous. Here are two possible derivations of $i + i$:

$$\begin{array}{llllllllllll} E & \Rightarrow & T+E & \Rightarrow & T+T & \Rightarrow & F+T & \Rightarrow & i+T & \Rightarrow & i+F & \Rightarrow & i+i \\ E & \Rightarrow & T+E & \Rightarrow & F+E & \Rightarrow & i+E & \Rightarrow & i+T & \Rightarrow & i+F & \Rightarrow & i+i \end{array}$$

The choice lies in the nonterminal expanded at each step. The *rightmost derivation* restricts this nonterminal to be the rightmost one.

Definition 5 (Rightmost derivation) *The rightmost derivation* $\overset{\text{rm}}{\Longrightarrow}$ *is the least relation such that for all* $\alpha \in (N \cup T)^*$, $w \in T^*$ *and rule* $A \to \beta$*:*

$$\alpha A w \overset{\text{rm}}{\Longrightarrow} \alpha \beta w$$

Definition 6 (Right sentential form) *A sentential form* α *is a (valid) right sentential form if it is derivable from the start symbol using rightmost derivations:*

$$S \overset{\text{rm}}{\Longrightarrow}^* \alpha$$

Rightmost derivations generate the same sentences as the unrestricted derivation but a much smaller set of sentential forms. In other words:

$$\begin{array}{rcl} \left\{ w \in T^* \mid S \overset{\text{rm}}{\Longrightarrow}^* w \right\} & = & \left\{ w \in T^* \mid S \Rightarrow^* w \right\} \\ \left\{ \alpha \mid S \overset{\text{rm}}{\Longrightarrow}^* \alpha \right\} & \subset & \left\{ \alpha \in T^* \mid S \Rightarrow^* \alpha \right\} \end{array}$$

A Shift-Reduce parser processes the input from left to right and attempts to reconstruct a rightmost derivation using a stack. At each step, two actions are possible:

- `Shift` moves a symbol from the beginning of the input to the top of the stack.
- `Reduce` which, given a rule $A \to \alpha$, removes the sequence α from the top of the stack and replaces it with A.

Parsing $i + i$ with the Shift-Reduce framework leads to the following sequence of actions:

Stack	Input Suffix	Action
ϵ	$i+i$	`Shift`
i	$+\,i$	`Reduce` $F \to i$
F	$+\,i$	`Reduce` $T \to F$
T	$+\,i$	`Shift`
$T+$	i	`Shift`
$T+i$	ϵ	`Reduce` $F \to i$
$T+F$	ϵ	`Reduce` $T \to F$
$T+T$	ϵ	`Reduce` $E \to T$
$T+E$	ϵ	`Reduce` $E \to T+E$
E	ϵ	

The parse succeeds when the stack contains only the start symbol and the input suffix is empty. This is because both `Shift` and `Reduce` maintain the invariant that the stack can be expanded to the prefix of the input that has been consumed. At any step, if we name the stack α, the source input u, the consumed prefix v and the remaining suffix w such that $u = vw$, then $\alpha \overset{\text{rm}}{\Longrightarrow}^{*} v$. If $\alpha = S$ and $w = \epsilon$, then $u = v$ and $S \overset{\text{rm}}{\Longrightarrow}^{*} u$: the input has been derived from the start symbol. Conversely, if no action sequence permits reducing the input to the start symbol, it is not a valid sentence.

From a successful parse, we can recover the rightmost derivation by reading the successive stacks from bottom to top. At each step we have that $S \overset{\text{rm}}{\Longrightarrow}^{*} \alpha v$: αv is a valid right sentential form and α is said to be a *viable prefix of a right sentential form.*

A derivation tree can be constructed by interpreting the actions:

- a `Reduce` $A \to \alpha$ action translates to a node labelled A with children labelled α
- a `Shift` step translates to a leaf

Formalisation Shift-Reduce parsing can be presented by relating states of the form (α, u) with a non-deterministic transition relation (α, u) R (β, v) such that:

- (α, au) R $(\alpha a, u)$
- $(\alpha\beta, u)$ R $(\alpha A, u)$ when $A \to \beta$

Both rules maintain the invariant that if (α, u) R (β, v) then $\beta v \overset{\text{rm}}{\Longrightarrow}^{*} \alpha u$. Thus a sentence u is valid when (ϵ, u) R* (S, ϵ) since it implies that $S \overset{\text{rm}}{\Longrightarrow}^{*} u$.

Definition 7 (Viable prefix of a rightmost sentential form) *α is a viable prefix of a rightmost sentential form if there exists w such that* (α, w) R* (S, ϵ).

In this definition, "R* (S, ϵ)" means that there is a path to success: a viable prefix is one in which the parser is not stuck; there exists at least one suffix w to complete the recognition. The viable prefixes play a significant role in the rest of the dissertation, but the formalisation of Shift-Reduce parsing was given for reference and is not used after.

1.6 LR parsers

Shift-Reduce is a general framework for parsing using rightmost derivations, but the transition relation is non-deterministic and there is nothing to tell when to `Shift` and when to `Reduce` (and, if reducing, which rule to reduce). The only guarantee is that, if the input is entirely consumed and the start symbol is reached, then we have built a rightmost derivation witnessing that the input is a valid sentence.

The problem can actually be refined by asking two questions:

1. Which actions are viable, in the sense that the stack remains a viable prefix when they are applied?

2. Among the viable actions, which one is the right one for analysing the input sentence?

LR parsers were proposed by Knuth (1965) to decide, at each step, which actions are applicable. They offer a definitive answer to the first question and a partial one to the second question. His observation, known as the fundamental property of LR parsing, was that the viable prefixes of right sentential forms form a regular language and, as such, can be enumerated by a finite automaton.

This automaton is the LR(0) automaton and is at the core of all LR parsers, but it is not sufficient in itself. For many practical grammars, the automaton reveals that multiple actions are often possible and so the second question remains.

The usual solution is to *look ahead* at a prefix of the remaining input, which is often sufficient to determine that some of the actions are impossible. This is the basis of LR(k) parsing, which use the k next symbols to select an action. The original representatives are the canonical LR(k) parsers, which are unfortunately impractical due to their excessive size. Many practical variants have since been proposed, notably optimised LR(1) parsers which compact the automaton (Denny and Malloy, 2010; Pager, 1977), or less expressive but simpler variants such as LALR(1) (DeRemer, 1969).

If the look-ahead information is not sufficient for determining that a single action is possible, the automaton is said to have conflicts. LRGrep does not rely on a specific technique for constructing the automaton and works with all automata of the LR(1) family, including those with conflicts.

1.6.1 The LR(0) automaton

To construct the automaton, we need to adjust the grammar to explicitly handle the end of the input. We add a new terminal \$ to denote the end of the input and replace the start symbol S with a new nonterminal S' with the rule $S' \to S\$$. The purpose of this change is to distinguish a complete sentence, such as $1+2\$$, from a prefix $1+2$ that can form a valid sentence but can also be continued ($1+2+3$, $1+2*\ldots$, etc).

Although we refer to *the LR(0) automaton*, three automata are actually involved: the LR(0) NFA, DFA and PDA (Pushdown Automaton); see Gallier and Quaintance (2021). We start by presenting the simplest one, the LR(0) NFA. It is mainly useful as

an educational step. Determinising it leads to the LR(0) DFA, which is usually directly constructed by LR parser generators. The DFA can be given a second interpretation as a PDA. The PDA implements a Shift-Reduce parser, using the DFA to determine the valid actions, ensuring that the stack always represents a path in the DFA. Unless otherwise specified, the LR(0) automaton refers to the LR(0) PDA. This dual view is important:

- as a finite automaton, it characterises the language of viable prefixes of rightmost sentential forms,
- as a pushdown automaton, it provides a (non-deterministic) strategy for establishing rightmost derivations.

A large part of this work builds on switching between these two views: using the finite automaton to reason about the possible prefixes or suffixes of the configurations reached by the pushdown automaton.

Definition 8 (The LR(0) NFA) *The LR(0) NFA is constructed by going through all items of the grammar: starting at the left of the initial symbol and moving to the right symbol by symbol, connecting a nonterminal immediately after the dot to the initial items of the rules that define this nonterminal. Concretely:*

- *the states are the LR(0) items, i.e., the set of all $A \to \alpha \cdot \beta$ for $A \to \alpha\beta$ ranging over all rules*
- *the initial state is $S' \to \cdot\, S\$$*
- *all states are accepting*
- *for each state $A \to \alpha \cdot x\gamma$ there is a transition labelled x to the state $A \to \alpha x \cdot \gamma$*
- *for each state $A \to \alpha \cdot B\gamma$ and rule $B \to \beta$ there is an ϵ-transition to the state $B \to \cdot\, \beta$*

Figure 1.1 shows the LR(0) NFA of G_A. Starting from the initial state $S \to \cdot\, E\,\$$, the transitions describe the sequences that can be found on the stack. An ϵ-transition simply means that more actions are possible in this situation, as more states are reachable from the current one without consuming any input. Following a path in the NFA tells us which actions are admissible for a Shift-Reduce parser. When reaching a state with an outgoing transition labelled a, it is valid to `Shift` when looking ahead at a. When reaching a state $A \to \alpha \cdot$ it is possible to reduce the rule $A \to \alpha$.

The next step is to determinise the NFA and obtain *the LR(0) DFA*. Figure 1.2 shows the LR(0) DFA of G_A. Each state is annotated by the set of items active when in this state. The sets are split into two parts (materialised by a horizontal bar) to distinguish items reached by following an ϵ-transition from other items. For instance, the initial state starts with item $S \to \cdot\, E\$$, but because the dot is on the left side of nonterminal E, the ϵ-closure reveals that items $E \to \cdot\, T$, $E \to \cdot\, T{+}E$, etc., are also active.

Definition 9 (Kernel of a state) *The set of items that are simultaneously active in a state of the LR(0) DFA without following the ϵ-transitions is called the kernel of the state. A state is uniquely identified by its kernel.*

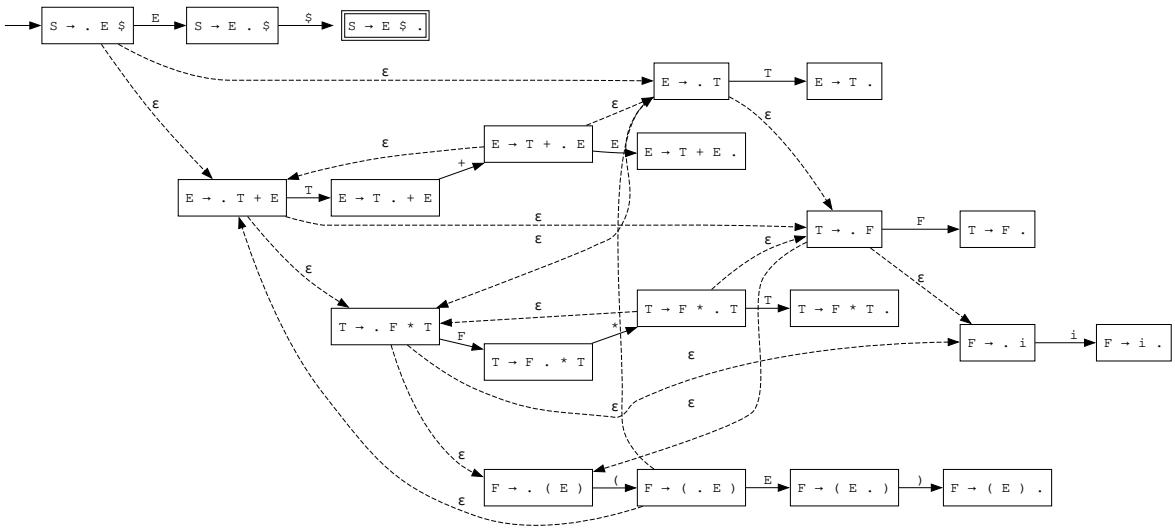


Figure 1.1: LR(0) NFA of the arithmetic grammar G_A.

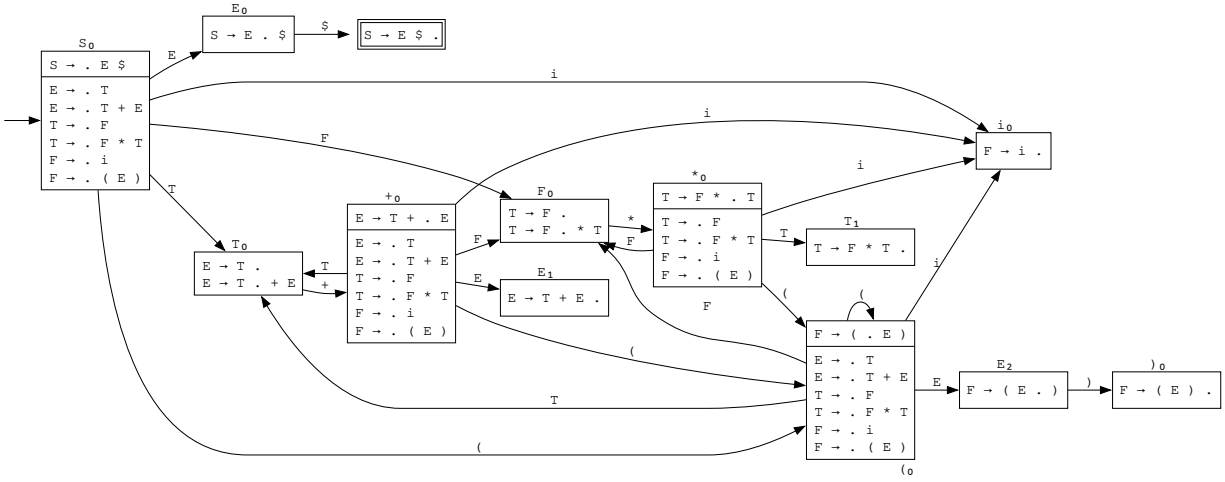


Figure 1.2: LR(0) DFA of the arithmetic grammar G_A.

The set $\{S \rightarrow \ . \ E \ \$\}$ is the kernel of the initial state. Adding all items reachable through ϵ-transitions is the closure of the item set; this is the set used by the filter operator to select states (as in the example from the Introduction).

Definition 10 (Incoming symbol) *By construction, all incoming transitions of a state have the same label. We define a function* ***incoming***(s) *to get the incoming symbol of state s. The initial state has no incoming transitions, therefore the* ***incoming*** *function is partial.*

The uniqueness of the incoming symbol lets us define a short notation for naming the states: x_i for the i-th state with incoming symbol x, and we reserve the name S_0 for the initial state that has no incoming symbol. For instance, in G_A, we have $\textbf{incoming}(E_0) = E$.

We lift it to sequences, allowing us to write $\textbf{incoming}(\vec{s})$ for $\vec{s} \in S^*$ such that:

$$\begin{array}{lcl} \textbf{incoming}(S_0) & = & \epsilon \\ \textbf{incoming}(\epsilon) & = & \epsilon \\ \textbf{incoming}(\vec{s} \cdot s) & = & \textbf{incoming}(\vec{s}) \cdot \textbf{incoming}(s) \end{array}$$

Stack of the LR(0) automaton The DFA determines which actions can be applied to a Shift-Reduce parser. The stack of the Shift-Reduce parser that we introduced earlier is made up of grammar symbols. With a naive interpretation, the DFA is traversed for each stack to find which state is active at the end of the stack. This style is called "Direct LR parsing" and is not efficient in an implementation. Instead, practical LR implementations record the states of the DFA on the stack such that the current state is always at the top of the stack. Parsing starts with only the initial state on the stack. When following a transition, the target state is pushed onto the stack. To reduce a rule $A \rightarrow \alpha$, $|\alpha|$ states are removed from the stack and the transition labelled A from the new top of the stack is followed.

It is possible to recover the stack of a Shift-Reduce parser from the stack of an LR parser by mapping the **incoming** function over the stack. The initial state has no incoming symbol and is removed. Other states can be seen as tagged versions of symbols, and the **incoming** function drops the tags. Hence our notation x_i, where x is the incoming symbol and i is a tag remembering the context in which this symbol occurred, allowing quick decisions about valid actions.

Let's illustrate with the G_A grammar. After processing the input $1+2*3$, the stack of a Shift-Reduce parser contains symbols $T + F * i$, while the stack of an LR parser has symbols $S_0\,T_0\,+_0\,F_0\,*_0\,i_0$. With the LR stack, it is immediate to see that the current state is i_0. To reduce $T \rightarrow F*i$, one needs to look only three states deeper in the stack, consuming $F_0 *_0 i_0$ and reaching $+_0$, and find that the goto transition to follow is $+_0 \xrightarrow{F} F_0$, ending with stack $S_0\,T_0\,+_0\,F_0$.

1.6.2 LR conflicts

The LR(0) automaton refines the Shift-Reduce parser by providing an efficient way to decide which actions are applicable. If no action is applicable, we can reject the input

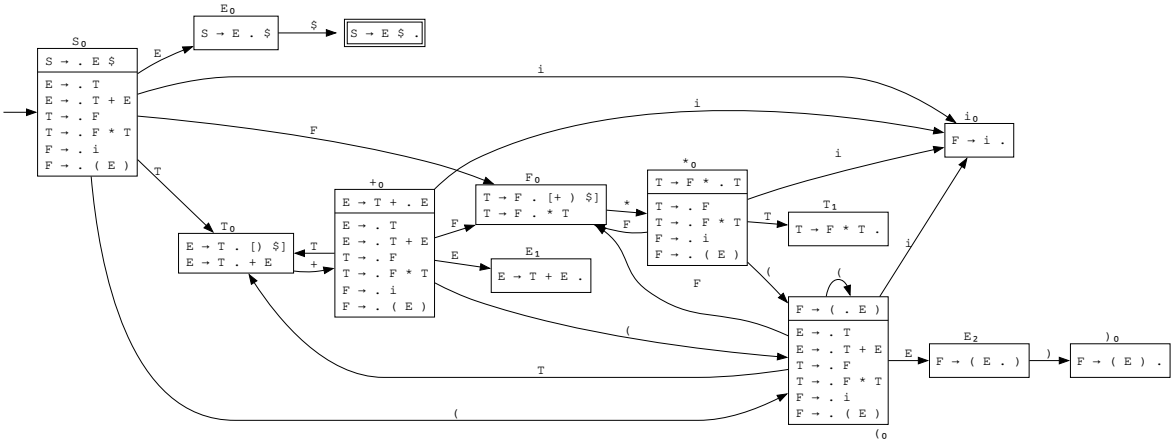


Figure 1.3: An LALR automaton for G_A

as early as possible. If one action is applicable, we can proceed deterministically. But if multiple actions are possible, we have a problem.

In Figure 1.2, observe that in state T_0 two actions are possible: shifting + to reach the state $+_0$ or reducing $E \to T$. LR(0) does not provide enough information to deterministically decide which action should be taken. This situation is called an LR conflict.

A conflict can be due to a real ambiguity in the grammar, when there really are more than one way to parse some inputs, but it is often due to the limited power of the automaton. Many conflicts can be solved just by looking at the next input symbol. For instance, in G_A we can see by looking at the predecessors of T_0 that the reduction can bring us to three states:

- E_0 when preceded by S_0,
- E_1 when preceded by $+_0$,
- E_2 when preceded by $(_0$.

When in E_1, the only possible action is to reduce, so we repeat the process, until ending either in E_0 or E_2. Neither of them have a transition on +. This solves the conflict: in T_0, if we are looking ahead at +, we should `Shift`, otherwise we can `Reduce`$(E \to T)$. All conflicts of G_A can be solved this way, as shown in Figure 1.3. It represents the automaton refined by annotating the conflicting reductions with the possible lookahead tokens between square brackets.

This technique is known as Simple LR(1), or SLR(1). It is slightly more expressive than LR(0) and is sufficient for analysing G_A. Unfortunately, conflicts are often harder to resolve. Switching to more expressive parsing strategies takes care of some of them (SLR(1) $\subset$ LALR(1) $\subset$ LR(1)). GLR takes another stance and explores all possible actions in parallel, with clever algorithms to avoid as much redundant work as possible. Sometimes it is also possible to refactor the grammar to delay a decision enough to resolve a conflict.

$$
\begin{array}{lcl}
E & \rightarrow & E{+}E \\
E & \rightarrow & E{*}E \\
E & \rightarrow & i \\
E & \rightarrow & (E) \\
\%\text{right} & * & \\
\%\text{right} & + &
\end{array}
$$

Figure 1.4: Arithmetic parsing using Yacc directives

Intentional conflicts and truncation of the automaton Conflicts can also be introduced on purpose to give a more concise specification of a language. Yacc (Johnson, 1975), an influential parser generator, offers directives that complement a grammar to resolve conflicts by forcing it to shift or reduce depending on the symbols involved. The other actions involved in the conflicts are simply removed from the automaton.

Figure 1.4 shows a grammar with Yacc directives that is equivalent to our grammar G_A. Interpreted as a Context-Free Grammar, it is ambiguous and leads to a conflicting automaton. However, the directives force + and $*$ to be right-associative, and the order of the declarations gives priority to $*$ over +. This is enough to solve all conflicts.

We see the benefits of using directives in this case: the grammar is terser, and some would say more readable because of the explicit prioritisation of the operators. In general, directives should be used with caution. Ideally, conflict resolution should only change the admissible derivations without changing the language, but this is difficult to reason about. In worse cases, conflict directives silence real ambiguities and change the language being recognised in unpredictable ways.

Because this feature is widely used in practice, LRGrep supports automata with removed actions. Conflict resolution affects LRGrep because it restrains the stacks reachable by an LR parser. In the static analyses, over-approximation can lead to false reports and we have to use a precise, but more complex, enumeration of stacks that accounts for truncated actions. For compiling LRGrep patterns to a matching automaton, it is fine to over-approximate as, in practice, it only leads to marginally larger automata.

1.6.3 Formal definition of the LR(1) automaton

We write:

- $a, b, c, z \in T$ for terminal symbols,
- $A \in N$ for nonterminal symbols,
- $x \in X$, where X is $T \cup N$, for arbitrary symbols,
- $Z \in \mathscr{P}(T)$ for sets of terminal symbols,
- $w \in \vec{T}$ for sentences (sequences of terminal symbols),
- $\alpha, \beta \in \vec{X}$ for sentential forms (sequences of symbols),

- $s \in S$ for automaton states, and $\bar{s}$ for the initial state.

A transition in the automaton is a directed edge from a source state s to a target state s'. Every transition is labelled with a symbol x. The "shift" transitions are the transitions whose label is a terminal symbol; the "goto" transitions are those whose label is a nonterminal symbol. We write Tr for the set of all transitions.

Because the automaton is deterministic, the target state s' of a transition is determined by s and x. Thus, a transition can be identified by the pair (s, x), and Tr can be viewed as a subset of $S \times X$. The target state of a transition is given by the function target : $Tr \to S$, which can also be viewed as a partial function target : $S \times X \rightharpoonup S$.

We extend the target function to operate on sequences of symbols:

$$\begin{aligned} \text{target}(s, \epsilon) &= s \\ \text{target}(s, x \cdot \alpha) &= \text{target}(\text{target}(s, x), \alpha) \end{aligned}$$

Incoming symbol All incoming transitions of a state are labelled with the same symbol, given by the partial function **incoming** : $S \rightharpoonup X$. The function is not defined on the initial state as it has no incoming transition.

Reductions We assume that the reduction actions of the automaton are given by a function *reduce*, which maps a pair of a state s and a production $A \to \alpha$ to a set of terminal symbols. When the automaton is in state s, if the next input symbol—the "lookahead symbol"—is a member of reduce$(s, A \to \alpha)$, then the next action of the automaton is to reduce the rule $A \to \alpha$.

Predecessors From the set of transitions we derive the function **pred** that associates a state with its predecessors:

$$\mathbf{pred}(s) = \left\{ s' \in S \mid \exists x, \text{target}(s', x) = s \right\}$$

We write $\mathbf{pred}^i$ for iterated applications, defined by:

$$\begin{aligned} \mathbf{pred}^0(s) &= \{s\} \\ \mathbf{pred}^{i+1}(s) &= \bigcup_{s' \in \mathbf{pred}(s)} \mathbf{pred}^i(s') \end{aligned}$$

Well-formedness of reductions If reduce$(s, A \to x_1 \dots x_n) \neq \emptyset$:

- the predecessors of s should have incoming symbols that coincide with the producers $x_1 \dots x_n$ of the rule:

$$\forall i < n, s' \in \mathbf{pred}^i(s), \mathbf{incoming}(s') = x_{n-i}$$

- the n-th predecessors should have a goto transition on A:

$$s' \in \mathbf{pred}^n(s) \implies (s', A) \in Tr$$

Conflict-free automaton (actions are deterministic) A state s should have at most one action per lookahead terminal a: a transition if $(s, a) \in Tr$ or a unique reduction if $\exists A \to \alpha, a \in \text{reduce}(s, A \to \alpha)$.

Items after conflict resolution When constructing the LR(0) automaton, we introduced the notion of items representing a position in the grammar, written $A \to \beta \cdot \gamma$ provided that $A \to \beta\gamma$ is a rule. Since each state of the LR(1) automaton refines an LR(0) state, we can in principle define the LR(0) items of an LR(1) state as the items of the LR(0) state it refines.

However, conflict resolution can make some of the LR(0) items ineffective. Consider the Yacc grammar from Figure 1.4: a state of the generated automaton might have $E \to E \cdot + E$ in its kernel and have the transition on + removed because it was conflicting.

Definition 11 (Effective Items) *We say that an LR(0) item $A \to \beta \cdot \gamma$ is effective in a state s if there is a path leaving this state that leads to reducing $A \to \beta\gamma$. The effective items are given by* ***items***(s)*:*

$$\textbf{\textit{items}}(s) = \left\{ A \to \beta \cdot \gamma \;\middle|\; \textit{reduce}(\textit{target}(s, \gamma), A \to \beta\gamma) \neq \emptyset \right\}$$

A state effectively has item $A \to \beta \cdot \gamma$ if, after following a path labelled γ, the state reached can reduce the rule $A \to \beta\gamma$ for at least one lookahead terminal. An item can be made ineffective in two ways: some transitions have been removed which prevent following a path labelled γ (target(s, γ) is not defined), or the reduce action has been removed (reduce(target$(s, \gamma), A \to \beta\gamma$) is empty).

For some degenerate automaton with heavily truncated actions, this definition is still an over-approximation of the reductions that can actually be performed on an input. If γ contains a nonterminal A, it is possible that the path is still present in the automaton (target(s, γ) is defined) and yet no input permits following the goto-transition labelled A. Either because reductions of rules $A \to \ldots$ have been removed during conflict resolution, or because the paths leading to reduce these rules themselves have some unreachable goto-transitions. This problem of reachability is the topic of Chapter 5: an exact solution can be computed but is non-trivial. In practice, we found that the over-approximation above is sufficient and close enough to the mental model the grammar author might have of the language recognised.

1.7 Dealing with Invalid Inputs

An input is invalid when, after consuming a certain prefix, the automaton enters a configuration where there is no action compatible with the lookahead symbol. When this occurs, the LR parser rejects the input, signalling to the rest of the program that the input could not be analysed.

Unfortunately, most parsers discard the automaton's configuration upon rejecting the input. Most generated parsers are presented as black boxes that keep their

configuration hidden. Furthermore, optimisations employed by practical implementations, such as the *default reductions of Yacc*, delay the detection of failures rather than immediately reporting the error. When this is the case, the parsing stack is in an unpredictable state for the developer. The configuration when the failure is reported cannot be relied upon in general.

On a positive note, it is easy to identify the location where the input was rejected. A parser can always produce a message such as "`Syntax error on line` x`, column` y". This is a bare minimum, but is quite unsatisfying from a production compiler. There is no clear path beyond this point, and different solutions have been proposed.

1.7.1 Failing Productions

As a first resort, it can be tempting to extend the grammar with rules to handle common mistakes. Instead of letting the parser fail because no action is applicable, we introduce new actions to deal with anticipated invalid inputs.

For instance, the error message for a missing number used in the introduction can be produced by a rule like:

$$E \to \texttt{)} \quad \{\ \texttt{error("Expecting an integer")}\ \}$$

An input such as `(1+)` is recognised as a valid prefix by the parser, but the error is reported by the semantic action when reducing $E \to$ `)`.

A variant of this approach involves writing productions that recognise a language slightly larger than the correct one and filtering extraneous cases in the semantic actions. This can be useful for enforcing fine syntactic constraints with no structural effects on the rest of the grammar. For instance, consider a language that restricts the use of lowercase and uppercase identifiers to specific contexts. Instead of representing each case with a different terminal, one can use a single terminal and distinguish between lowercase and uppercase identifiers in the semantic action.

Pros and Cons This approach is simple and pragmatic. No extra facilities are required from the parser, as long as it allows aborting the parsing process within an action and ensures that actions are evaluated at predictable times. Accessing semantic values in the action is straightforward (it is just like any other rule) and can be useful for crafting detailed error messages.

However, this approach is also very limited and does not scale well. The example above has two major flaws:

- It handles only a single case of a specific error message: expecting a number when looking ahead at a `)`, whereas the same message applies to many other cases; e.g., `(1++)`, `1+*`, etc.
- The semantic action triggers only if the parser reduces $E \to$ `)`. If the rest of the input is incorrect, there is no guarantee that this reduction will occur, and the parser can reject immediately after shifting `)`.

There is nothing we can do about the second problem as it is an implementation-defined behaviour. Different parser generators execute semantic actions at different times[5]. Handling errors in semantic actions makes the grammar less declarative.

The first problem is difficult to address without additional facilities from the parser generator. One can try to enumerate possible cases, but this introduces risks of new conflicts and possibly real ambiguities. Whether a given lookahead token is valid depends on the context in which a nonterminal is used, and thus this approach does not generalise well.

When adding rules for rejecting specific inputs, the grammar no longer describes the language one aims to recognise. This can be problematic when using the grammar for purposes other than strict parsing. For instance, here are a few interesting applications of an LR grammar:

- Extracting a readable reference grammar from an implementation, as done by Catala for documentation purposes (Merigoux et al., 2021).
- Suggesting syntax completion to the user (Sasano and Choi, 2021).
- Recovering from incorrect inputs (Bour et al., 2018; Diekmann and Tratt, 2020).
- Grammar-guided fuzzing.

In all these use cases, the grammar is used as a generation device where it is desirable to distinguish failing productions from regular ones to avoid generating failures.

1.7.2 The *error* Symbol (Yacc)

Yacc (Johnson, 1975) is a widely used LALR(1) parser generator that significantly impacted the field. We have already seen that it introduced directives for resolving conflicts. It also has an interesting facility for dealing with errors, via a a special terminal called *error*. This terminal cannot appear in the input but can be used in grammatical rules like any other terminal. When the parser enters a rejecting configuration, it replaces the invalid lookahead terminal with *error* and attempts to resume analysis.

Using *error* can address some limitations of the previous approach. The rule:

$$E \to \mathit{error} \quad \texttt{\{ error("Expecting an integer") \}}$$

does not suffer from the flaws of our earlier attempt. Since the *error* symbol is introduced only after the parser gets stuck, there is no risk of accidentally conflicting with the language and it represents precisely the terminals that are invalid in this situation.

However, *error*-rules are still susceptible to ambiguities. When attempting to exhaustively handle errors with *error*-rules, one quickly encounters situations where multiple error messages apply, which manifests as LR-conflicts on the *error* symbol.

[5]Common strategies include eagerly reducing when there is no need for a lookahead to disambiguate actions or, instead, always delaying by a lookahead token. In GLR parsing, it is also common to suspend evaluation of semantic actions in non-deterministic settings.

```
$ more foo.ml
(2 + )
$ ocamlc -c foo.ml
File "foo.ml", line 1, characters 5-6:
1 | (2 + )
         ^
Error: Syntax error: ')' expected
File "foo.ml", line 1, characters 0-1:
1 | (2 + )
    ^
  This '(' might be unmatched
```

Figure 1.5: A very confusing message due to Yacc's error semantics

This is to be expected: there are many ways to explain a given error, and the LR formalism is not well equipped for ranking these explanations.

For static analyses, the story is better than with regular failing productions: *error*-rules are easy to identify syntactically so other consumers of a grammar can take care of properly handling (or ignoring) them.

Panic mode One last thing worth mentioning is the behaviour of a parser if it gets stuck again when the lookahead has already been replaced by *error*. The simplest solution is to simply reject the input, but Yacc offers another strategy. States from the stack are popped until finding one that can handle the *error* symbol, or rejecting the input if reaching an empty stack.

The net effect is throwing away the most recently parsed constructions, until finding a place from which to resume the analysis. This is useful for providing coarse-grained recovery. For a compiler, it can mean throwing away the current definition and resuming on what looks like the next valid definition. For an interactive interpreter, it can mean resuming from a fresh prompt.

While convenient for recovery, this facility can be detrimental to error messages. The effect of throwing away contents from the stack can be counter-intuitive without a good understanding of the dynamics of an LR parser. When finding an *error* rule after throwing contents, the semantic action has no way to know what was thrown away and will likely report an error that would be applicable if this content was not part of the input to begin with.

The problem is well illustrated by this issue of the OCaml compiler discovered and fixed by François Pottier[6]. When fed with `(2 + )`, ancient versions of the compiler would report the very confusing message shown in Figure 1.5. This is due to a

[6] https://github.com/ocaml/ocaml/pull/10095

rule equivalent to $E \rightarrow$ (E *error*. Here is what happens:

- After consuming +, the stack contains (E+
- Looking ahead at), the parser gets stuck: in this state, it has actions neither for) nor for *error*
- The + is thrown away, and the parser looks again for an action on *error* from stack (E.
- This time it succeeds: the stack is (E, and the error rule matches
- It shifts *error* and reduces the rule, outputting the error message.

When a rule like $E \rightarrow$ (E *error* succeeds, it seems natural to complain about a missing parenthesis in the semantic action. However, because of the strategy of "popping" stuck states from the stack, the informal meaning of the rule is that an error happened *sometimes* after reading an opening parenthesis and an expression, not *immediately* after.

The problem is solved by disabling this strategy to restore the intuitive meaning of *error*. Therefore, by default, this Yacc facility is suitable for recovering from errors but not for explaining them. Even when *panic* is disabled, *error* is not sufficient for an exhaustive handling of failures and is very sensitive to ambiguities.

1.7.3 Merr: indexing error messages by LR(1) state

Merr (Jeffery, 2002) is the first solution that offered to separate the specification of the grammar from the specification of error messages. In this approach, the grammar and the parser are left unchanged. The error specification is a list of examples that pair an invalid sentence with an appropriate error message. At runtime, if the parser fails in a similar situation, the associated message is reported.

A possible Merr specification for our arithmetic example is:

```
1+ { "Expecting an integer" }
(1 { "Expecting a closing parenthesis" }
```

To determine the similarity, Merr uses the state of the LR(1) automaton reached at the point of failure. This is a simple solution, easy to integrate with an existing parser. It also offers an interesting workflow for refining error messages. When encountering a situation that deserves a good error message, one simply adds the problematic input to the specification and pairs it with an appropriate explanation.

However, directly indexing on LR(1) states is not a good way to identify erroneous situations. In particular, there is no guarantee that a state captures the features relevant for explaining the error.

Without any reduction, an LR(0) state only captures local syntactic properties. In our arithmetic example, it was necessary to reduce a complete expression to see that a closing parenthesis was missing. Canonical LR(1) automaton mitigates this by refining LR(0) states to account for all the lookahead tokens inherited from context.

In this situation, the LR(1) state alone is sufficient to confidently report the missing parenthesis.

But this is not without problems either. For some errors, such as the missing number, the LR(0) state is sufficient. In these situations indexing on LR(1) state is too fine-grained: a message that could apply to many situations ends up restricted to the specific case reached by the example. Reasoning about the inherited tokens is also difficult without assistance from specialised tools, so these situations are hard to predict for the grammar author.

When the automaton is non-canonical, indexing on states can instead be too coarse-grained. To keep the size of the automata manageable, parser generators resort to different optimisations that lead them to reduce more aggressively than a canonical automaton. When this happens, a different error message might be selected because the reductions have changed the current state.

This is again hard to predict for a grammar author and it is also fragile. The optimisations performed by the parser generator can be affected by seemingly unrelated changes in the grammar or changes to the generator itself. Thus, the error specification obtained by this approach is inherently unstable.

Finally, this approach still offers no way to guarantee complete coverage of syntax errors.

1.7.4 Menhir: controlling reductions and coverage

Menhir (Pottier, 2016) also uses a list of examples for specifying error messages and addresses the main flaws of Merr. It provides appropriate tooling to guide the grammar author and it supports directives to control the reductions that happen before looking for an error message.

Error coverage Menhir ensures comprehensive error handling by mandating that each state susceptible to failure has a corresponding error message associated with it. This feature automatically generates sample sentences triggering each failing state, proving invaluable for maintenance, testing, and the writing of initial error messages for a grammar. However, this comprehensiveness presents a significant drawback in practice: realistic parsers often possess over a thousand potential failure states. Maintaining such a vast repository of error messages incurs substantial overhead especially as the grammar evolves (even minor changes can necessitate widespread updates).

Each invalid sentence is annotated with the LR(1) items of the state in which the error is detected. A grammar author can use this to precisely understand the grammatical features characterised by the LR(1) state and to word a suitable error message.

Computing the set of states that can fail and generating sentences to reach them is surprisingly difficult for a non-canonical automaton that has gone through conflict resolution. This requires solving the reachability problem for LR automata, which

can be computationally expensive. In Chapter 5, we present a novel and more efficient solution to this problem.

Controlling reductions The second feature, forcing some reductions on error, permits generalising a situation before picking an error message. Let's compare the LRGrep pattern and the Merr example for the closing parenthesis:

```
[E /F → (E·)]    { "Expecting a closing parenthesis" }
(1               { "Expecting a closing parenthesis" }
```

The LRGrep pattern specifies that the expression E should be reduced before matching, while the Merr example left this unspecified. In Menhir, this is specified in the grammar with the following annotation:

```
%on_error_reduce E T F
```

It says that when an error is detected, rules of the form $E \to \ldots$, $T \to \ldots$, or $F \to \ldots$, should be reduced while this is permitted by the current state. This directive can be specified for multiple nonterminals, and the order of the declarations is used to choose which nonterminal to target when multiple ones are possible.

Comparison with LRGrep The effect achieved by an `%on_error_reduce` directive is close to specifying a reduction in an LRGrep pattern, but comes with a few significant differences. The error specification in Menhir is spread across the list of examples and the grammar directives, whereas LRGrep patterns unify reduction and selection.

By specifying and prioritising the reductions in the grammar, the strategy is fixed once and for all before looking for an error message. Menhir compiles the directives by changing the LR automaton, adding fallback actions that trigger the reductions rather than immediately rejecting. Specifying a reduction at the pattern-level offers more flexibility. For instance, by using multiple patterns, one can test different reductions before choosing a message.

For the grammar author, we think that ordering the patterns is more intuitive than ordering the directives: thinking about which error message should be selected first is more natural than thinking about which nonterminal should be reduced first. It also leads to more composable specifications: changing a reduction at the grammar-level has a global effect on error messages, while the impact of a pattern change is much more localised.

Finally, we can see that the nonterminals to reduce are interpreted differently. Menhir has a very immediate and operational interpretation of the directive: if a rule with E on its left-hand side is reducible in the current state, it should be reduced before proceeding. LRGrep interprets it as a target: it looks for a sequence of one or more reductions that end up producing an E. This is why in Menhir's case, we also had to specify T and F, even though they are intermediate steps, while $[E]$ is enough for LRGrep to find the sequence $i \to F \to T \to E$.

We feel the effect is easier to predict for the grammar author: a pattern reducing E matches if and only if there is at least one lookahead token that permits to reduce

the suffix at the point of failure to E. Using Menhir, the author has to think about the other nonterminals that might have to be reduced before E.

Conclusion

This chapter has provided a comprehensive overview of the theoretical foundations underpinning LRGrep, focusing on regular and context-free languages, finite automata, and parsing techniques. We have explored various approaches to handling error messages in LR parsers, from the *error* symbol in Yacc to more advanced solutions like Merr and Menhir. Each method has its strengths and weaknesses, but they all fall short in providing a seamless, comprehensive, and intuitive way to handle errors in parsing. By leveraging the inner workings of LR parsers and building upon the Merr and Menhir approaches, LRGrep aims to address these limitations and provide a robust solution for error handling in LR parsing.

Chapter 2

Reduce-filter patterns

This chapter explores the class of *reduce-filter patterns*, presented in the introduction. These patterns can be viewed as a generalization of the approaches used by Merr and Menhir, and offer a simple and efficient implementation. While they are less expressive than the full LRGrep framework, they are representative of the unique challenges involved in reasoning about reductions in LR parsers.

2.1 Reduce-filter patterns

We follow Merr and Menhir by using a specification separate from the grammar. Unlike them, we use a custom pattern language rather than fixed sentences to characterise error situations, and we associate a user-defined semantic action rather than an error message. A single pattern can abstract over many failing configurations, giving more control to capture the relevant features of an erroneous situation while being less verbose than an enumeration of invalid samples.

The primitive patterns are made from symbols and LR(0) items. Since LR(0) items are coarser-grained than LR(1) items, strictly speaking we lose some of the classification power of Merr's approach. However, we argue that this is beneficial. If needed, this power can be recovered in the full calculus (in Chapter 3) by constraining the stack prefix rather than relying on the inherited lookaheads. Depending on LR(1) items would leak implementation details of the automaton. Instead, a pattern can be interpreted independently of a specific automaton: we give a formal semantics to our calculus using only context-free grammars and rightmost derivations.

Like Menhir, we provide tools for working with error specifications:

- An interpreter that annotates an invalid sentence with the applicable reductions and the reachable LR(0)-items, allowing the author to identify which parts are specific to this input and which parts should be matched. This enables a workflow similar to Merr's one, where an error specification can be incrementally refined as new problematic cases are discovered.
- An enumeration of sentences covering all reduce-filter patterns for a given grammar, suitable for fuzzing, testing, and getting started.

- A coverage analysis that handles the full calculus and can provide sample invalid sentences which are not covered by any error message yet.

The full calculus is defined in Chapter 3, the analyses are treated in Chapter 6, and the actual tools are presented in Chapter 7. The rest of this chapter focuses on interpreting the reduce-filter patterns, first at the grammar level as a set of sentential forms, then on an LR automaton with an efficient decision procedure.

Definition 12 (Reduce-filter pattern) *A reduce-filter is written:*

$$[\,\alpha \,/\, A_1 \to \beta_1 \cdot \gamma_1 \ldots \,/\, A_n \to \beta_n \cdot \gamma_n\,]$$

It consists of a sequence of symbols α and a set of items $A_i \to \alpha_i \cdot \beta_i$ for $1 \le i \le n$. Given a set of items I, we also write $[\alpha \,/\, I]$ as an abbreviation.

Informally, a reduce-filter pattern $[\,\alpha \,/\, A_1 \to \beta_1 \cdot \gamma_1 \ldots \,/\, A_n \to \beta_n \cdot \gamma_n\,]$ denotes the set of stacks on which it is possible to apply a sequence of reductions leading to a new stack with suffix α and for which items $A_i \to \beta_i \cdot \gamma_i$ are *active*. We say that an item $A \to \alpha \cdot \beta$ is active for an LR stack when it is part of the closure of the item set of the state at the top of the stack. For a sentential form, this means that it has the form $\gamma\alpha$ and that both $\gamma\alpha$ and γA are viable prefixes.

We say that α is the reduction target. When it is empty, matching happens without reducing. In this case, the pattern denotes directly the stacks whose top state satisfies the item constraints.

More often, α is a single nonterminal N and matches the stacks on which it is possible to apply a sequence of reductions ending with a goto transition labelled N. Occasionally, it can also be a sequence of nonterminals. When a grammar has ϵ-rules, it is possible to follow consecutive goto transitions without shifting any input symbol. Consider a grammar with rules $A \to \beta_1 BC\beta_2$ and $C \to \epsilon$. Using BC as a reduction target identifies stacks from which it is possible to reduce B then X (trivially via the ϵ-rule).

Thus, although any sequence of symbols α is allowed, only certain sequences of nonterminals have a non-empty interpretation. We call them the *viable reduction targets*. For non-looping grammars, the set of viable reduction targets is finite.

2.2 Reduce-filter patterns on sentential forms

To interpret the reduce-filter pattern $[\,\alpha \,/\, A_1 \to \beta_1 \cdot \gamma_1 \ldots \,/\, A_n \to \beta_n \cdot \gamma_n\,]$, we proceed in two steps:

- Identify the set of sentential forms to which a sequence of reductions applies that produces the suffix α.
- Retain only the sentential forms where the items $A_i \to \alpha_i \cdot \beta_i$ remain active after the reductions.

The difficult part of the interpretation is to classify sentential forms according to the applicable reductions. But not all sentential forms need to be considered: reductions are sensible only on the forms that can appear on the stack of a shift-reduce parser, the *viable prefixes of right sentential forms*. We start by precisely characterising the viable prefixes. From there, we define a *reduction relation* to relate viable prefixes and reduction targets. Finally, we give a denotational semantics by interpreting reduce-filter patterns as subsets of viable prefixes.

2.2.1 Viable prefixes of right sentential forms

The viable prefixes of right sentential forms are the set of sentential forms that can appear on the stack of a shift-reduce parser. Given a grammar $G = (N, T, R, S)$, we consider the derived grammar $G' = (N', N \cup T, R', S')$ such that:

- The nonterminals of G' are a "$'$" tagged copy of N, $N' = \{A' | A \in N\}$,
- The terminals of G' are the symbols of G (the terminals and the nonterminals),
- $R' = \{A' \to \alpha \mid A \to \alpha\beta \in R\} \cup \{A' \to \alpha B' \mid A \to \alpha B\beta \in R\}$, a rule of G' is either a prefix of a rule of G, with nonterminals now considered as terminals, or a prefix up to a nonterminal of G, which is replaced by its tagged nonterminal counterpart in G'.

The language $\mathrm{VP} = \mathscr{L}(G')$ is the set of the viable prefixes of the right sentential forms of G, or simply the *viable prefixes*. This set is regular (it is generated by a right-regular grammar), and closed under prefixes ($\alpha\beta \in \mathrm{VP} \implies \alpha \in \mathrm{VP}$).

Dealing with ϵ and ϵ-rules The set of viable prefixes admits a simpler definition by means of an auxiliary notion of derivation $\overset{r}{\Rightarrow}$, the least relation such that:

$$\alpha A \overset{r}{\Rightarrow} \alpha\beta \quad \text{if} \quad A \to \beta\gamma$$

Only the last nonterminal can be expanded, and it can be expanded to any prefix of a rule that defines it. The viable prefixes are the sentential forms derivable from the start symbol via the reflexive and transitive closure of this relation:

$$\mathrm{VP} = \left\{ \alpha \,\middle|\, S \overset{r}{\Rightarrow}^{*} \alpha \right\}$$

While this definition generates the expected set of sentential forms, some of the intermediate steps of the derivation only accidentally coincide with the right definition. Since we can take any prefix of a rule when expanding, we can in particular take the empty prefix. If the sentential form before substituting ended with two nonterminals, say AB, after substituting B by ϵ, the form ends with A from which we can derive more strings.

Let us look at an example with grammar $S \to AB$, $A \to abc$, $B \to xyz$. To show that $ab \in \text{VP}$, many derivations are possible, such as:

1.
$$\begin{aligned} S &\overset{r}{\Rightarrow} A && \text{; using rule } S \to AB \text{ and prefix } A \\ &\overset{r}{\Rightarrow} ab && \text{; using rule } A \to abc \text{ and prefix } ab \end{aligned}$$

2.
$$\begin{aligned} S &\overset{r}{\Rightarrow} AB && \text{; using rule } S \to AB \text{ and prefix } AB \\ &\overset{r}{\Rightarrow} A && \text{; using rule } B \to xyz \text{ and prefix } \epsilon \\ &\overset{r}{\Rightarrow} ab && \text{; using rule } A \to abc \text{ and prefix } ab \end{aligned}$$

Intuitively, only the first one is well-behaved though. If we think of the derivation tree being built, the second derivation starts by introducing a node with two children A and B, makes the second child empty, and continues the expansion in the first node. The first derivation avoids these extra steps and directly explores the A branch.

Forbidding expansion with an empty prefix (requiring that $\beta \neq \epsilon$ in the definition of $\overset{r}{\Rightarrow}$) almost sidesteps the issue, but this does not feel right: ϵ is no longer derivable, and ϵ-rules cannot be substituted anymore. This phenomenon, having a simple and almost correct definition when ignoring cases involving ϵ, and in particular ϵ-rules, is recurrent in this work. While this is benign when looking only at viable prefixes, as soon as we look at the structure of derivations it becomes a real problem. For viable prefixes, G' solves the issue by distinguishing "active" nonterminals N' which can be substituted and can only appear as the last symbol, from "inactive" ones N.

When introducing LR grammars Knuth (1965) likely faced a similar problem. The first definition of viable prefixes, called "characteristic strings", concentrated on grammars without ϵ-rules. Only after introducing the main concepts would he extend it to all context-free grammars, handling the corner cases that arise when substituting by ϵ.

In this work, we introduce the general case first and sometimes comment on the simplifications permitted when there are no ϵ-rules.

2.2.2 Reduction relation

We introduce the *reduction relation*, noted $(\alpha)\beta \downarrow \gamma$, which holds when a viable prefix $\alpha\beta$ can be reduced to $\alpha\gamma$ by applying a sequence of zero or more reductions. The simple case of reducing a single rule $A \to \beta$ to get from $\alpha\beta$ to αA is witnessed by $(\alpha)\beta \downarrow A$.

The three components α, β, and γ capture the effect of the reductions on the sentential form:

- α is the prefix not affected by the reductions.
- β is the suffix consumed by the reductions.
- γ is the suffix produced, a sequence of nonterminals.

$(\alpha)\beta \downarrow \gamma$ is the least relation defined by these three inference rules:

$$\frac{\alpha \in \mathrm{VP}}{(\alpha)\epsilon \downarrow \epsilon}\ \text{(red-base)}$$

$$\frac{(\alpha\beta)\gamma \downarrow \vec{C} \quad \beta \neq \epsilon \quad A \to \beta\vec{C} \quad \alpha A \in \mathrm{VP}}{(\alpha)\beta\gamma \downarrow A}\ \text{(red-outer)}$$

$$\frac{(\alpha)\beta \downarrow \vec{A}\vec{C} \quad B \to \vec{C} \quad \alpha\vec{A}B \in \mathrm{VP}}{(\alpha)\beta \downarrow \vec{A}B}\ \text{(red-inner)}$$

In the base case *(red-base)*, no reduction is applied: the sentential form is just α, it has to be a viable prefix but nothing is substituted (ϵ is replaced by ϵ). The other rules capture the application of a single reduction, although we have to distinguish two cases.

(red-outer) applies when a reduction consumes symbols from the stack. If the stack $\alpha\beta\gamma$ reduces to $\alpha\beta\vec{C}$, and if it is possible to further reduce the production $A \to \beta\vec{C}$, then the rule concludes that $\alpha\beta\gamma$ reduces to αA, the prefix α is still unaffected and $\beta\gamma$ is consumed to produce A. The constraint $\alpha A \in \mathrm{VP}$ guarantees that the reduction is applicable on this stack, and the constraint $\beta \neq \epsilon$ prevents the rule from overlapping with the third rule in certain situations.

(red-inner) applies when a reduction does not consume any new symbol from the stack. This happens when reducing a production with a single nonterminal on the right-hand side, e.g., if $(\alpha)\beta \downarrow B$ and $A \to B$ is reducible, then $(\alpha)\beta \downarrow A$. The B produced by a previous reduction is replaced by A and the frontier between α and β is unchanged. It also happens when ϵ-rules are involved in the derivation. For instance if $C \to \epsilon$ and $\alpha C \in \mathrm{VP}$, then $(\alpha)\epsilon \downarrow C$ by *(red-inner)*. ϵ-rules can also cause multiple nonterminals to be produced. If $(\alpha)\beta \downarrow B$ and $\alpha BC \in \mathrm{VP}$, then $(\alpha)\beta \downarrow BC$.

Interpreting reduce-only patterns The reduction relation lets us interpret a pattern of the form $[\gamma]$: it denotes exactly the sentential forms $\alpha\beta$ such that $(\alpha)\beta \downarrow \gamma$. A corollary is that the γ such that $\exists\alpha\beta, (\alpha)\beta \downarrow \gamma$ are exactly the *viable reduction targets*. In other words, if $\nexists\alpha\beta, (\alpha)\beta \downarrow \gamma$, then a pattern $[\gamma \,/\, A_1 \to \alpha_1 \cdot \beta_1 \ldots / A_n \to \alpha_n \cdot \beta_n]$ always denotes the empty set.

Untouched prefix and ϵ-rules In the definition of $(\alpha)\beta \downarrow \gamma$, a special treatment is needed to distinguish the untouched prefix α from the suffixes.

It is tempting to use a simpler relation that considers the viable prefixes reachable by a single reduction step, say $\downarrow^r$, such that $\alpha\beta \downarrow^r \alpha A$ when $\alpha A \in \mathrm{VP}$ and $A \to \beta$. The reflexive and transitive closure is then sufficient to model the effect of sequence of reductions. However we cannot recover the semantics of $[\gamma]$ from this relation. The likely candidate $\Gamma = \{\alpha\beta \mid \alpha\beta \downarrow^{r*} \alpha\gamma\}$ is too big. As expected, it contains the $\alpha\beta$ such that $(\alpha)\beta \downarrow \gamma$, but it can also contain elements with accidentally overlapping prefixes or suffixes: the $\alpha_1\alpha_2\beta$ such that $(\alpha_1)\alpha_2\beta \downarrow \alpha_2\gamma$ and the $\alpha\gamma_1\beta$ such that $(\alpha)\gamma_1\beta \downarrow \gamma_1\gamma_2$, for $\alpha = \alpha_1\alpha_2$ and $\gamma = \gamma_1\gamma_2$.

In the absence of ϵ-rules, sequences of reductions produce at most one symbol in the end. The viable reduction targets are simply the $\gamma \in \{\epsilon\} \cup N$. The relation $\downarrow^r$ suffices to interpret $[\gamma]$: $[\epsilon]$ denotes VP and $[A]$ denotes $\{\alpha\beta \mid \alpha\beta \downarrow^{r+} \alpha A\}$, where $\downarrow^{r+}$ is the transitive and irreflexive closure of $\downarrow^r$.

Example (matching reduction on G_A) We can see that $[E]$ matches $(F + i$ with the following derivation:

$$\begin{array}{rl} ((F+i)\epsilon \downarrow \epsilon & \text{by (red-base)} \\ ((F+)i \downarrow F & \text{by (red-outer) reducing } F \to i \\ ((F+)i \downarrow E & \text{by (red-inner) reducing } E \to F \\ (()F+i \downarrow E & \text{by (red-outer) reducing } E \to F+E \end{array}$$

2.2.3 Interpretation

Now we have all the tools to interpret a reduce-filter pattern to a set of sentential forms.

A pattern $[\gamma \,/\, A_1 \to \alpha_1 \cdot \beta_1 \ldots / A_n \to \alpha_n \cdot \beta_n]$ denotes the set of the $\alpha\beta$ such that:

1. $\alpha\beta$ is reducible to $\alpha\gamma$. That is, $(\alpha)\beta \downarrow \gamma$.
2. The items $A_i \to \alpha_i \cdot \beta_i$ are active in $\alpha\gamma$: for all $1 \le i \le n$, there exists α' such that $\alpha'\alpha_i = \alpha\gamma$ and $\alpha' A_i \in \text{VP}$.

2.3 Reduce-filter patterns on LR stacks

In the rest of this chapter, we interpret the reduce-filter patterns on the stack of an LR automaton. This semantics is more operational and amenable to an efficient implementation using a finite automaton.

The concepts we built for sentential forms transpose to the stacks of LR automata:

- The viable prefixes map to the *viable stacks*, which are efficiently enumerable.
- The reduction relation maps to the (viable) *reduction automaton*. It consumes a suffix of a stack from right to left and can be used to enumerate the applicable reductions and their targets.
- The viable reduction targets become distinguished subsets of states of the reduction automaton.
- A reduce-filter pattern is representable as a subset of these target states. To recognise a set of reduce-filter patterns, one can simply extract the subset of the automaton reaching the target states of these patterns.

2.3.1 Viable stacks

Enumerating the stacks of an LR automaton is quite straightforward. The LR automaton already enumerates the sentential forms, but the stack of a practical LR parser stores the states of the automaton, not the symbols. A viable stack is simply a path in the automaton, a sequence of states obtained by starting from the initial state and following the outgoing transitions—ignoring the labels.

For matching a reduce-filter pattern, it is more efficient to analyse a stack starting from the right end (the current state of an LR automaton). So we enumerate the stacks in reverse, growing a suffix until reaching the initial state:

- The procedure starts with a singleton suffix s, for any $s \in S$
- A suffix $\vec{s}$ grows to all the suffixes $s' \cdot \vec{s}$, for $s' \in \text{pred}(\text{head}(\vec{s}))$
- Suffixes are grown until reaching the initial state (which has no predecessor), at which point they are viable stacks.

For this, we assume that all states are reachable from the initial state. The set of viable stacks is generated by a non-deterministic automaton and is therefore regular.

Conflict resolution and stack reachability If some actions have been removed from the automaton during conflict resolution, the stacks obtained by this procedure can be an over-approximation of the reachable ones. An LR automaton configured with a viable stack is well-behaved, but it is possible that no input can actually put the automaton in this configuration when starting from the initial configuration. The obvious case is when that would require taking a shift transition that was removed. It can also happen that some goto transitions are impossible to take because the reduce actions that would allow reaching them were removed.

We distinguish *viable* stacks and *reachable* stacks. The formers over-approximate the latters and are sufficient for recognition purposes. It is fine to recognise a few more stacks, even if these will never be exercised in practice. The *reachable* stacks are an exact enumeration of the stacks an LR automaton can reach when starting from the initial configuration, but are harder to determine. Chapter 5 studies the problem of reachability. Chapter 6 computes the set of reachable stacks and uses it to answer coverage questions that require an exact enumeration to avoid false reports.

2.3.2 Reduction automaton

The (viable) reduction automaton is an ϵ-NFA $(\mathbf{RA}, S, \delta_{\mathbf{RA}}, \texttt{Initial}, \{s_0\})$ that plays a role on viable stacks similar to the reduction relation $(\alpha)\beta \downarrow \gamma$ on the viable prefixes. The reduction relation is built by exploring the reductions applicable on the viable prefixes of a grammar; the reduction automaton is built by exploring the reductions applicable on the viable stacks of an LR(1) automaton.

The purpose of this automaton is to recognise the suffixes of a stack that can be reduced, in such a way that it is easy to determine the applicable reductions, the

targets reached by these reductions, and the intermediate stacks visited by the sequence of reductions. It does so by exploring suffixes of viable stacks and simulating the applicable reductions.

Besides an initial state `Initial`, the states of the automaton are of two forms:

- `Suffix`$(s, \vec{s}', T')$ represents a configuration in which the stack is known to have suffix $s \cdot \vec{s}'$ and the lookahead symbol belongs to $T' \in \mathscr{P}(T)/\{\emptyset\}$. s is the last state consumed from the input stack and $\vec{s}'$ is a simulated suffix made of the targets of goto transitions.

- `Reduce`$(s, \vec{s}', T', A, \alpha)$ represents an intermediate configuration entered when reducing a rule of the form $A \to \alpha\beta$. The stack has suffix $s \cdot \vec{s}'$, the lookahead symbol belongs to T', and A is the nonterminal labelling the goto transition to follow after consuming the remaining producers α from the top of the stack.

In both cases, s represents the last state consumed from the stack being analysed, $\vec{s}'$ represents the suffix appended to the stack by the goto transitions followed after reducing, and T' the lookahead symbols that permit to reach this configuration.

For each $s \in S$, there is a transition `Initial` $\xrightarrow{s}$ `Suffix`(s, ϵ, T). The initial transitions are there to determine the state at the top of the LR stack—the current state of the LR parser being analysed. Once the current state is known, the rest of the automaton explores the applicable reductions. For this, the other transitions simulate the reduction actions performed by an LR parser:

- `Suffix`$(s, \vec{s}', T') \xrightarrow{\epsilon}$ `Reduce`$(s, \vec{s}', T'', A, \alpha)$ when $T'' = T' \cap \text{reduce}(\text{top}(s \cdot \vec{s}'), A \to \alpha)$, $T'' \neq \emptyset$ to simulate an LR parser starting to reduce $A \to \alpha$ (permitted since it is in a state where the rule is reducible for the lookaheads in $T'' \subset T'$).

- `Reduce`$(s, \vec{s}', T', A, \epsilon) \xrightarrow{\epsilon}$ `Suffix`$(s, \vec{s}' \cdot \text{target}(\text{top}(s \cdot \vec{s}'), A), T')$ to simulate the goto transition followed when reducing $A \to \alpha$ after $|\alpha|$ symbols have been consumed from the stack.

- Transitions of state `Reduce`$(s, \vec{s}', T', A, \alpha)$ when $\alpha \neq \epsilon$ simulate the consumption of one symbol from the stack, depending on $\vec{s}$:

  - A single transition `Reduce`$(s, \vec{s}', T', A, \alpha) \xrightarrow{\epsilon}$ `Reduce`$(s, \text{pop}(\vec{s}'), T', A, \text{pop}(\alpha))$ when $\vec{s} \neq \epsilon$. In this case, the symbol is consumed from the simulated suffix $\vec{s}'$ and not from the input.

  - A transition `Reduce`$(s, \epsilon, T', A, \alpha) \xrightarrow{s'}$ `Reduce`$(s', \epsilon, T', A, \text{pop}(\alpha))$, for each $s' \in \text{pred}(s)$. The simulated suffix is empty; we have to consume one state from the input stack to proceed. The current state is known to be s, and since the input is a viable stack, the predecessor has to be one of $\text{pred}(s)$; we add a labelled transition for each candidate.

This first transition takes us to a state for the form `Suffix`(s, ϵ, T), representing the facts that the state at the top of the stack is s, no reductions have been performed

yet (no goto transition has been followed), and no assumption has been made about the lookahead symbol. As outgoing transitions are followed, the set of lookaheads decreases, accumulating the conditions made on the lookahead symbols by the reductions applied.

2.3.3 Reduction targets

In a standard non-deterministic automaton, reaching a final state establishes the membership of the input sequence to the language recognised by the automaton. In the reduction automaton, reaching a state $\texttt{Suffix}(s, \vec{s}', T')$ establishes that the reduction relation $(\alpha)\beta \downarrow \gamma$ holds for the stack being analysed for an appropriate choice of α, β and γ.

More precisely, let $\vec{u} = \vec{v}s\vec{w}$ be a viable stack such that the state $\texttt{Suffix}(s, \vec{s}', a)$ is reached after consuming the suffix $s\vec{w}$. Then we have $(\alpha)\beta \downarrow \gamma$ for:

$$\begin{aligned} \alpha &= \textbf{incoming}(\vec{v}s) \\ \beta &= \textbf{incoming}(\vec{w}) \\ \gamma &= \textbf{incoming}(\vec{s}') \end{aligned}$$

The lookaheads T' matter for the LR automaton and do not affect the relation. The **incoming** function recovers a sentential form from a sequence of LR states. The components are related as follows:

- In the automaton, s is the first state of the suffix that has not been consumed by a reduction. In the relation, α is the prefix of the viable prefix that has not been consumed by a reduction.
- The suffix $\vec{w}$ consumed by the automaton corresponds to β, the suffix of the sentential form consumed by the reductions.
- $\vec{s}'$ is the sequence of states followed by the goto transitions, and it directly coincides with γ, the sequence of nonterminals appended to the viable prefix by the reductions.

2.4 Interpretation

When studying the reduction relation, we observed that in a reduce-filter pattern $[\gamma \,/\, A_1 \to \alpha_1 \cdot \beta_1 \ldots / A_i \to \alpha_i \cdot \beta_i]$, the viable targets γ are the ones appearing in the relation $(\alpha)\beta \downarrow \gamma$. In the reduction automaton, γ translates to the set of $\vec{s}'$ such that $\textbf{incoming}(\vec{s}') = \gamma$, for the $\vec{s}'$ appearing in states of the form $\texttt{Suffix}(s, \vec{s}', T')$.

This gives an efficient strategy for matching those patterns: a pattern $[\gamma]$ recognises the stacks reaching any state $\texttt{Suffix}(s, \vec{s}', T')$ such that $\textbf{incoming}(\vec{s}') = \gamma$. An implementation can traverse the automaton while consuming the stack from right-to-left and accept the input as soon as such a state is reached.

To handle the filter part on a sentential form $\alpha\beta$ such that $(\alpha)\beta \downarrow \gamma$, we had to further constrain the reduced form $\alpha\gamma$: an item $A \to \alpha_i \cdot \beta_i$ is active if α_i is a suffix of

$\alpha\gamma$ and if replacing α_i by A resulted in a viable prefix. With an LR state, we can tell immediately whether an item is active or not by looking at its items. The LR state at the top of the simulated stack when reaching a state $\texttt{Suffix}(s, \vec{s}', T')$ is $\text{top}(s\vec{s}')$: the target of the last goto transition, or simply s if no goto transition has been followed ($\vec{s}' = \epsilon$).

This gives an efficient decision procedure for reduce-filter patterns, where all the expensive computations can be done in a preprocessing step:

1. Compute the reduction automaton from the LR(1) automaton.
2. Determine the set of states of the form $\texttt{Suffix}(s, \vec{s}', T')$ where $\textbf{incoming}(\vec{s}') = \gamma$ and $A_i \to \alpha_i \cdot \beta_i \in \text{items}(\text{top}(s\vec{s}'))$.
3. During execution, when an LR parser fails, look for a suffix of its stack reaching one of those states.

This strategy generalises naturally to matching multiple patterns at once: each pattern translates to a target subset of states of the reduction automaton. A single scan of the stack with the reduction automaton looking for these subsets suffices to list all the matches.

2.4.1 Implementation details

An efficient implementation will benefit from some generic optimisations:

- Determinising the automaton with the subset construction guarantees worst-case linear time recognition (Rabin and Scott, 1959).
- If the patterns are known ahead of time, the automaton can be pruned by removing the states from which no target is reachable.
- Minimising it can bring significant reductions in memory consumption (Hopcroft, 1971; Jacobs and Wissmann, 2022; Valmari, 2012).

But significant gains can be achieved by applying domain-specific knowledge.

Wildcard transitions When reducing a rule $A \to x_1 \dots x_n$ from a stack with suffix $s_0 \dots s_n$, the states $s_1 \dots s_n$ are entirely determined by s_0 and $x_1 \dots x_n$. Using the target function:

- $s_1 = \text{target}(s_0, x_1)$
- $s_2 = \text{target}(s_1, x_2)$
- $\dots s_i = \text{target}(s_{i-1}, x_i)$

This means that the reduction automaton, when simulating a reduction, does not really need to look at the intermediate states $s_1 ... s_n$; the final state s_0 is sufficient to determine which goto transition to follow.

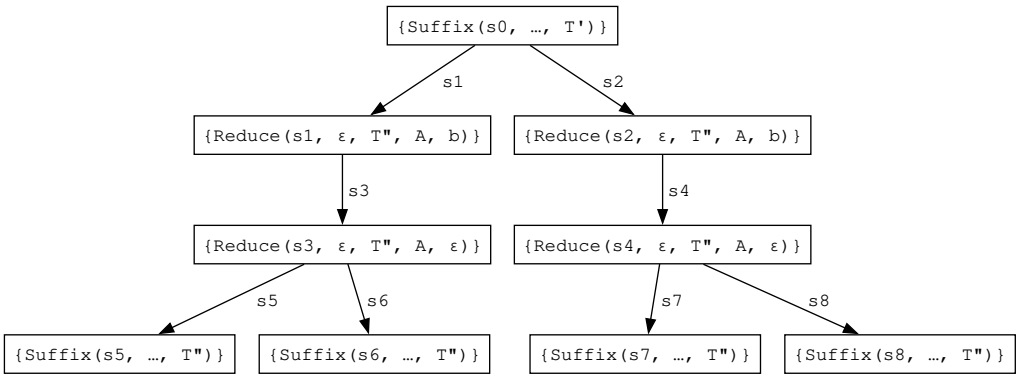
For an efficient implementation, it is interesting to consider a *wildcard* variant of the reduction automaton that exploits this fact by adding an extra _ symbol to the input alphabet, interpreted as any LR(1) state. We can then refine the transitions that simulate the consumption of an input symbol from the stack when reducing. Transitions leaving $\texttt{Reduce}(s, \epsilon, T', A, \alpha)$ when $|\alpha| > 1$ are now labelled "_":

$$\texttt{Reduce}(s, \epsilon, T', A, \alpha) \rightharpoonup \texttt{Reduce}(s', \epsilon, T', A, \text{pop}(\alpha)) \text{ for each } s' \in \text{pred}(s), \text{ when } |\alpha| > 1$$

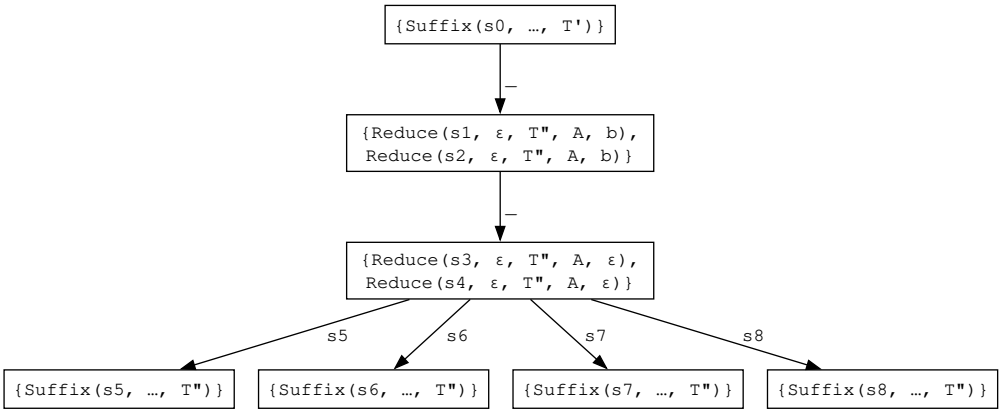
The other cases are unchanged. When $|\alpha| = 1$, the transitions are labelled s' rather than _. When $|\alpha| = 0$, this is the case where the end of the reduction has been reached and we have an ϵ-transition simulating a goto transition.

Delaying branching using wildcards can significantly decrease the size of the reduction automaton after determinisation. Informally, introducing wildcard transitions change the shape of paths simulating a reduction state from being tree-shaped to being fan-shaped. Compare both approaches on an artificial example:

- Tree-shaped:



- Fan-shaped:



The wildcards permitted merging more states in the subset construction. The former grows quadratically as the productions get longer, while the latter grows linearly. On realistic applications, the impact is really significant, and we think that this optimisation is mandatory for an efficient implementation.

2.4.2 Annotating LR Stacks

An immediate application of the reduction automaton is to annotate LR stacks with the possible reductions. Here is an example of the output from LRGrep's interpreter annotating the stack of the OCaml parser processing the source code `"1"`:

```
Parser stack (most recent state first):
- line 1:0-1  INT
              / constant
              / simple_expr
              / expr
                / list(structure_element)
               / list(post_item_attribute)
              / seq_expr
              / structure
- entrypoint  implementation
```

It is not necessary to know the definitions of the symbols involved to get an idea of what is happening. The output can be read as follows:

- The stack has two states: the integer 1 represented by the terminal `INT`, and the entrypoint `implementation` (we are trying to parse an OCaml implementation file).
- The integer can be reduced to a `constant`, then to a `simple_expr`, to an `expr`, and to a `seq_expr`; at this point the stack has the shape `implementation`, `seq_expr`.
- After the `seq_expr`, it is possible to reduce an (empty) list of attributes, leading to a stack with three entries:

  `implementation`, `seq_expr`, `list(post_item_attribute)`

- After the attributes, it is possible to reduce an (empty) list of other structure elements, leading to a stack with four entries:

  `implementation`, `seq_expr`, `list(post_item_attribute)`, `list(structure_element)`

  That list contains the definitions in the continuation of the file, but there are none in this example.

- The `seq_expr`, `list(post_item_attribute)`, `list(structure_element)` can be reduced to `structure`, a sequence of definitions that forms a syntactically valid OCaml module.

The annotations are a compact notation for the intermediate stacks that the LR parser can go through when reducing the input. Annotations on the same column represent points where the stacks diverge, while annotations on a more indented column represent an extension of the stack below.

For instance, `/ constant` and `/ simple_expr` are on the same column and followed by the entrypoint. They represent the two stacks:

- `implementation: constant`
- `implementation: simple_expr`

On the other hand, `/ list(post_item_attribute)` appears above `/ seq_expr` and is more indented. This represents:

- The base stack `implementation: seq_expr`
- The sub-stack `implementation: seq_expr; list(post_item_attribute)`

For reference, here are all the stacks reached while reducing, omitting the entrypoint:

- `INT` (the starting stack, before any reduction)
- `constant`
- `simple_expr`
- `expr`
- `seq_expr`
- `seq_expr; list(post_item_attribute)`
- `seq_expr; list(post_item_attribute); list(structure_element)`
- `structure`

Conclusion

Existing solutions specifically designed to provide error messages for LR parsers, Merr (Jeffery, 2002) and Menhir (Pottier, 2016), work by associating messages with LR states, provided as a list pairing invalid inputs with error messages. Using the current LR state alone quickly proves to be too limiting, as a single state captures only local properties of the syntactic context. Menhir solves this problem by letting the author of the grammar specify some nonterminals that should always be reduced before looking for an error message, abstracting some syntactic details away.

We introduce the *reduce-filter* patterns that offer an alternative solution. A pattern is designed to provide a concise way to identify a syntactic situation by specifying simultaneously the reductions and the states to look for. The nonterminals to reduce are interpreted as a search problem local to the pattern rather than as a global sequence of instructions to prepare the parser before looking for an error message. This offers a more declarative approach—the interpretation of a pattern is independent from the rest of the specification, and rather than using samples of invalid inputs,

we use grammatical items to characterise LR states, which is more general and less sensitive to implementation details of the LR automaton.

Given a context-free grammar, we interpret the patterns as a subset of right sentential forms, using the *reduction relation* $(\alpha)\beta \downarrow \gamma$. For practical applications, an efficient implementation strategy is to interpret the patterns as a regular subset of the stacks of an LR parser. To do so, we show how the *viable stacks* can be enumerated from right-to-left, and we define the *reduction automaton* **RA**, a finite automaton that enumerates the sequence of reductions applicable to a stack. A reduce-filter pattern translates directly to a subset of this automaton.

Chapter 3

A calculus for error specifications

In this chapter, we introduce a dialect of regular expressions for analysing the stacks of LR parsers, generalising the reduce-filter patterns of the previous chapter. After an informal explanation of this dialect and the rationale behind it, we give two semantics:

- A big-step semantics that relates terms to the sentential forms they recognise.
- A small-step operational semantics for matching stacks of LR parsers, using an appropriate extension of regular expression derivation.

The regular expressions are defined in terms of a grammar G—the atomic expressions consist of symbols and items from the grammar. The big-step semantics are used to give a precise meaning to the language constructions independently of a specific implementation—their meaning depends only on the grammar.

Given an LR automaton that recognises this grammar, the small-step semantics pave the way for an efficient implementation using finite automata, which is the topic of Chapter 4.

3.1 A regular-expression dialect for matching LR stacks

Our calculus extends regular expressions with the two operators presented earlier: item filters and reductions. We also support variable bindings[1] for naming certain parts of the input. In the formal presentation, the variables store positions of the input; in the actual implementation, they are used to refer to the semantic values saved on the stack.

[1] Also known as (named) captures or capture groups in the context of regular expressions.

The complete calculus has the following syntax:

$$
\begin{array}{rcl}
\text{Exp} \ni e & ::= & 0 \mid \epsilon \mid x \\
& \mid & e_1 \cdot e_2 \\
& \mid & e_1 \mid e_2 \\
& \mid & e^* \\
& \mid & / A \to \alpha \cdot \beta \\
& \mid & [e] \\
& \mid & n = e
\end{array}
\qquad
\begin{array}{rcl}
\text{Name} \ni n & & \\
\text{Var} \ni v & ::= & n_\mathsf{s} \mid n_\mathsf{e}
\end{array}
$$

The base cases are 0, which recognises nothing, ϵ, which recognises the empty word, and x, for $x \in T \cup N$, which recognises a single symbol x. Then come the usual regular expression operators:

- The concatenation $e_1 \cdot e_2$ recognises the words uv such that u is recognised by e_1 and v is recognised by e_2. We omit the $\cdot$ when it is not ambiguous: $a \cdot b \cdot c$ can be written abc and $e \cdot / A \to \alpha \cdot \beta$ can be written $e \ / A \to \alpha \cdot \beta$.
- The disjunction $e_1 | e_2$ recognises a word u if it is recognised by e_1 or by e_2.
- The Kleene star e^* recognises zero or more copies of e; it is equivalent to the expansion $e^* = \epsilon | e \cdot e^*$.

Variables Variables are introduced by a binding operator $n = e$ that gives a name to the subsequence of the input consumed by the sub-expression e. The name n is drawn from an arbitrary set Name of variable names. When matching, the binding defines two variables noted n_s and n_e, which refer to the starting and ending positions of the subsequence that matched.

Filters and reductions The reduce operator $[e]$ and the filter operator $/ A \to \alpha \cdot \beta$ generalise the reduce-filter patterns of the previous chapter. Unlike standard regular operators, filters and reductions do not just denote a set of words; they also constrain their prefix[2].

A filter is noted $/ A \to \alpha \cdot \beta$ and recognises situations where the item $A \to \alpha \cdot \beta$ is active. The filter does not consume input; it only restricts the possible matches. For instance, the terms E and $E / F \to (\ E \cdot\ )$ recognise a single arithmetic expression: the former recognises any expression, while the latter only recognises those occurring after an opening parenthesis. The expression is consumed by E, and the matching situations are further filtered by $/ F \to (\ E \cdot\ )$.

A reduction is introduced by square brackets $[e]$ and recognises a stack suffix $\vec{s}$ if $\vec{s}$ can reduce to a stack suffix that is recognised by e. For instance, the term $[E]$ recognises $T + i$, $F * i$, etc., since they all can reduce to E.

[2]Like the *lookbehind* operator found in certain dialects of regular expressions. Symmetrically, the *lookahead* operator constrains the suffix without consuming it (Mamouras and Chattopadhyay, 2024).

To simplify the implementation, reductions cannot be nested, e.g. $[[e]]$ is not allowed. A single use of the reduction operator is sufficient to explore all sequences of reductions that are consistent with a lookahead terminal. In principle, nesting two reductions would allow impossible sequences of reductions that result from starting with reductions allowed by a lookahead a and followed by reductions allowed by a different lookahead b. This use is forbidden.

Variable bindings are also not allowed inside reductions. The expression between brackets represents a suffix that the stack could potentially reduce to but which is never actually materialised in the LR parser, so we cannot extract information from it.

Ambiguous matching With bindings, the interpretation of an expression can become ambiguous: when there are multiple ways to match the input, which subsequences should be bound to the variables? We address this question by fixing disambiguation rules for the operators.

A disjunction $e_1|e_2$ prioritises its left-hand side: if both sides recognise u, it will be recognised by keeping the variable bindings from e_1. For instance, $(v = i)|i$ binds v if it succeeds, while $i|(v = i)$ never binds v.

A concatenation $e_1 \cdot e_2$ favours its right-hand side, and this disambiguation has priority over operators in the sub-expressions. For instance, matching the sentence abc against $(v = a|ab)(v = c|bc)$ binds v to c.

The Kleene-star e^* is disambiguated by keeping the shortest match, which coincides with the disambiguation given to the expansion $\epsilon|e \cdot e^*$.

A reduction $[e]$ is also subject to ambiguities when multiple sequences of reductions are applicable. For instance, matching the incomplete arithmetic expression $(1+2+3$ with $[E]$ is ambiguous. A parser analysing it would get stuck with the stack $(T+T+i$, which admits the following sequence of reductions: $(T+T+i \Leftarrow (T+T+F \Leftarrow (T+T+T \Leftarrow (T+T+E \Leftarrow (T+E \Leftarrow (E$. Since the pattern $[E]$ looks for reductions ending in E, it can succeed at the steps $(T+T+E$, $(T+E$ and $(E$. We choose to keep the candidate that consumes the fewest stack symbols—in this case, $T+T+E$. In the implementation, this is configurable: the author of the error specification can choose to keep the shortest or the longest match (explained in Section 7.1.4).

3.1.1 Right-to-left matching

When assessing an LR failure, the most recent information is found at the top of the LR stack. For instance, parsing $(1*2+\$$ with the LALR automaton for G_A (Figure 1.3) fails in a configuration with stack $S_0(_0T_0+_0$. Looking at state $+_0$ is sufficient to see that + is missing an operand: if a + has just been shifted, we know that the prefix $T+$ of the rule $T+E$ was correctly recognised. Looking deeper in the stack can reveal that a closing parenthesis will still be missing after adding an operand, but this information is of little value in this case.

The common case is therefore to extract information from some suffix of the stack and only occasionally look deeper. This is reflected in the matching language

by assuming that only the right end of the expression is *anchored*; we are looking for a matching suffix. The end of the stack should coincide with the end of the expression, but it is fine if some prefix of the stack has not been consumed. This is equivalent to interpreting an expression as if a wildcard _* had been prepended, matching any prefix.

Matching from right-to-left simplifies the derivation of item filters and reductions and leads to smaller matching automata where decisions are taken earlier. Looking for a suffix optimises for the common case. In the rare event where one would want to match a specific prefix, it is still possible to opt out of this behaviour by filtering with items matching the entry point of the grammar.

3.2 Formal Semantics

We give formal semantics to the calculus by relating an expression to the sentential forms it recognises. To account for the operators that restrict the matching situations without consuming input, we use the notation $(\alpha)\beta$ to represent the decomposition of the input $\alpha\beta$ into a prefix α which is only constrained and a suffix β which is consumed.

The *matching relation* noted $(\alpha)\beta \in e$ establishes that under prefix α, the suffix β is consumed and recognised by expression e. This relation does not handle variable bindings.

The *capturing relation* deals with the whole language. It is written $(\alpha)\beta \mathbin{@} \rho \in e$ where $\rho \in \mathrm{Var} \rightharpoonup \mathbb{N}$ is a *valuation*, a partial function that maps the variables bound in e to the offsets matched in $\alpha\beta$. $(\alpha)\beta \mathbin{@} \rho \in e$ holds when e recognises input β under prefix α while binding values ρ.

3.2.1 The matching relation

The matching relation is the least relation satisfying these rules:

$$\frac{}{(\alpha)\epsilon \in \epsilon}\ \text{(m-}\epsilon\text{)} \qquad \frac{}{(\alpha)x \in x}\ \text{(m-sym)} \qquad \frac{(\alpha)\beta \in e_1 \quad (\alpha\beta)\gamma \in e_2}{(\alpha)\beta\gamma \in e_1 e_2}\ \text{(m-seq)}$$

$$\frac{(\alpha)\beta \in e_1}{(\alpha)\beta \in e_1|e_2}\ \text{(m-disj-l)} \qquad \frac{(\alpha)\beta \in e_2}{(\alpha)\beta \in e_1|e_2}\ \text{(m-disj-r)} \qquad \frac{(\alpha)\beta \in \epsilon|ee^*}{(\alpha)\beta \in e^*}\ \text{(m-star)}$$

$$\frac{A \to \beta\gamma \in R \quad \alpha A \in VP}{(\alpha\beta)\epsilon \in / A \to \beta.\gamma}\ \text{(m-item)} \qquad \frac{(\alpha)\gamma \in e \quad (\alpha)\beta \downarrow \gamma}{(\alpha)\beta \in [e]}\ \text{(m-red)}$$

The first rules give the semantics of the standard regular expression operators, lifted to decompose the input into a constrained prefix α and a consumed suffix β:

- (m-ϵ) states that the empty expression recognises the empty sentential form under any prefix.
- (m-sym) states that expression x recognises the symbol x under any prefix.

- (m-seq) composes two expressions in sequence: if expression e_1 recognises β under prefix α and expression e_2 recognises γ under prefix $\alpha\beta$, then $e_1 e_2$ recognises $\beta\gamma$ under prefix α. The prefix of the second expression should coincide with the prefix and the input of the first one.

- (m-disj-l) handles disjunction when the left side matches; (m-disj-r) handles the symmetric case. If e_1 recognises β under prefix α, then so do $e_1|e_2$ and $e_2|e_1$.

- (m-star) handles the Kleene star by expanding e^* to $\epsilon|ee^*$. If $\epsilon|ee^*$ recognises β under prefix α, so does e^*.

The (m-item) rule handles item filters. It does not consume any input but constrains the prefix to be a viable prefix in which item $A \to \beta \cdot \gamma$ is active: the prefix should end with β, and substituting β with A should yield a viable prefix.

The (m-red) rule handles reductions by means of the reduction relation (introduced in 2.2.2). $[e]$ recognises β under prefix α if e recognises γ under prefix α and β reduces to γ under the same prefix.

3.2.2 The capturing relation

The capturing relation refines the matching relation, extracting more information from the input—in other words, $\exists \rho, (\alpha)\beta \mathbin{@} \rho \in e \iff (\alpha)\beta \in e$. Before giving the rules, let us look at ρ.

Valuations The ρ component of the relation is a valuation, a partial function that gives a value to the variables bound in e. This is used to interpret the binding construction $n = e$, which gives a name to the subsequence of the input that matched e. This is reflected in ρ by binding the two variables n_s and n_e to the positions where the input consumed by e started and ended.

For instance, extracting the position of a `b` surrounded by sequences of `a` can be done with the expression $\mathtt{a}^*(v = \mathtt{b})\mathtt{a}^*$. Applied to input "`aaaba`", we get $(\epsilon)\mathtt{aaaba} \mathbin{@} \rho \in \mathtt{a}^*(v = \mathtt{b})\mathtt{a}^*$ where $\rho(v_\mathrm{s}) = 3$ and $\rho(v_\mathrm{e}) = 4$. Positions of the input are numbered starting from 0: ${}_0\mathtt{a}_1\mathtt{a}_2\mathtt{a}_3\mathtt{b}_4\mathtt{a}_5$.

Valuations are introduced with three constructions:

- $\emptyset$ is the valuation with an empty domain.
- $v \mapsto i$ is a singleton valuation that only maps $v \in \mathrm{Var}$ to $i \in \mathbb{N}$.
- $\rho_3 = \rho_1 \uplus \rho_2$ is the left-leaning union: $\rho_3(v) = \rho_1(v)$ if $v \in \mathrm{dom}(\rho_1)$ else $\rho_3(v) = \rho_2(v)$ if $v \in \mathrm{dom}(\rho_2)$; we have that $\mathrm{dom}(\rho_3) = \mathrm{dom}(\rho_1) \cup \mathrm{dom}(\rho_2)$.

Rules The capturing relation $(\alpha)\beta \mathbin{@} \rho \in e$ is the least relation satisfying the rules:

$$\frac{}{(\alpha)\epsilon \mathbin{@} \emptyset \in \epsilon}\ \text{(c-}\epsilon\text{)} \qquad \frac{}{(\alpha)x \mathbin{@} \emptyset \in x}\ \text{(c-sym)}$$

$$\frac{(\alpha)\beta \mathbin{@} \rho \in e_1}{(\alpha)\beta \mathbin{@} \rho \in e_1|e_2}\ \text{(c-disj-l)} \qquad \frac{(\alpha)\beta \mathbin{@} \rho \in e_2}{(\alpha)\beta \mathbin{@} \rho \in e_1|e_2}\ \text{(c-disj-r)}$$

$$\frac{(\alpha)\beta \mathbin{@} \rho_1 \in e_1 \quad (\alpha\beta)\gamma \mathbin{@} \rho_2 \in e_2}{(\alpha)\beta\gamma \mathbin{@} \rho_1 \uplus \rho_2 \in e_1 e_2}\ \text{(c-seq)} \qquad \frac{(\alpha)\beta \mathbin{@} \rho \in \epsilon|ee^*}{(\alpha)\beta \mathbin{@} \rho \in e^*}\ \text{(c-star)}$$

$$\frac{A \to \beta\gamma \in R \quad \alpha A \in VP}{(\alpha\beta)\epsilon \mathbin{@} \emptyset \in \,/\, A \to \beta.\gamma}\ \text{(c-item)} \qquad \frac{(\alpha)\gamma \in e \quad (\alpha)\beta \downarrow \gamma}{(\alpha)\beta \mathbin{@} \emptyset \in [e]}\ \text{(c-red)}$$

$$\frac{(\alpha)\beta \mathbin{@} \rho \in e}{(\alpha)\beta \mathbin{@} (n_\mathrm{s} \mapsto |\alpha| \uplus \rho \uplus n_\mathrm{e} \mapsto |\alpha\beta|) \in n = e}\ \text{(c-var)}$$

Most rules are direct extensions of the corresponding rules from the matching relation, additionally defining a valuation:

- (c-ϵ), (c-sym), and (c-item) yield an empty valuation.
- (c-red) also yield an empty valuation; variables cannot be bound inside reductions. In $[e]$, the inner expression e is not matched against the input, so there would be no positions to store, and we can simply use the matching relation to relate an expression to a suffix it can reduce to.
- (c-disj-l) and (c-disj-r) leave the valuation unchanged.
- (c-seq) merges the valuations of the sub-expressions; the left-leaning union keeps the leftmost binding in case of redefinition. Since matching is done right-to-left, the leftmost bindings are the deeper in the stack and are the most "recently" seen ones.
- (c-star) is still defined by expansion to $\epsilon|ee^*$ and threads the valuation unchanged.
- (c-var) records the starting and ending position of the sub-expression in the valuation.

Disambiguating derivations When introducing the calculus, we observed that there are often different ways to match an expression. Formally, this translates to having multiple ways to relate a given input γ to an expression e: a family of sentential forms α_i, β_i and valuations ρ_i such that $(\alpha_i)\beta_i \mathbin{@} \rho_i \in e$ such that $\alpha_i\beta_i = \gamma$. The rules we gave to prioritise certain interpretations translate to an ordering on the family of proof derivations establishing that $(\alpha_i)\beta_i \mathbin{@} \rho_i \in e$, such that the smallest element is the disambiguated interpretation.

We define the ordering relation on two derivations:

$$\frac{\cdots}{(\alpha_1)\beta_1 \mathbin{@} \rho_1 \in e} < \frac{\cdots}{(\alpha_2)\beta_2 \mathbin{@} \rho_2 \in e}$$

where $\alpha_1\beta_1 = \alpha_2\beta_2$ (they recognise the same input but they can consume a different suffix). The application of most rules is directed by the syntax of expressions and deterministic; we discuss those later, but generally, the strategy is to do a lexicographic

ordering on the premises, from right-to-left. Cases that require more attention are the disjunction and reduction rules.

When relating two applications of (c-disj-l) and (c-disj-r), we favour the left side:

$$\frac{(\alpha_1)\beta_1 \mathbin{@} \rho_1 \in e_1}{(\alpha_1)\beta_1 \mathbin{@} \rho_1 \in e_1|e_2}\ \text{(c-disj-l)} \;<\; \frac{(\alpha_2)\beta_2 \mathbin{@} \rho_2 \in e_2}{(\alpha_2)\beta_2 \mathbin{@} \rho_2 \in e_1|e_2}\ \text{(c-disj-r)}$$

For reductions, we favour the reduction that consumed the fewest symbols:

$$\frac{(\alpha_1)\gamma_1 \in e \quad (\alpha_1)\beta_1 \downarrow \gamma_1}{(\alpha_1)\beta_1 \mathbin{@} \emptyset \in [e]}\ \text{(c-red)} \;<\; \frac{(\alpha_2)\gamma_2 \in e \quad (\alpha_2)\beta_2 \downarrow \gamma_2}{(\alpha_2)\beta_2 \mathbin{@} \emptyset \in [e]}\ \text{(c-red)} \iff |\beta_1| < |\beta_2|$$

Other cases are directed by the syntax of the expression, so we only need to relate different applications of the same rule:

- The base cases (c-ϵ), (c-sym) and (c-item) are deterministic and don't need to be disambiguated.
- (c-var) and (c-star) are ordered by their premises, e.g.:

$$\begin{array}{ccc} \dfrac{\dfrac{\cdots}{(\alpha_1)\beta_1 \mathbin{@} \rho_1' \in e}}{(\alpha_1)\beta_1 \mathbin{@} \rho_1 \in n = e} & < & \dfrac{\dfrac{\cdots}{(\alpha_2)\beta_2 \mathbin{@} \rho_2' \in e}}{(\alpha_2)\beta_2 \mathbin{@} \rho_2 \in n = e} \\ \Updownarrow & & \\ \dfrac{\cdots}{(\alpha_1)\beta_1 @ \rho_1' \in e} & < & \dfrac{\cdots}{(\alpha_2)\beta_2 @ \rho_2' \in e} \end{array}$$

- (c-seq) favours the right-hand side:

$$\begin{array}{ccc} \dfrac{\cdots}{(\alpha_1)\beta_1 \mathbin{@} \rho_{11} \in e_1} \dfrac{\cdots}{(\alpha_1)\beta_1 \mathbin{@} \gamma_1 \in \rho_{12} e_2} & < & \dfrac{\cdots}{(\alpha_2)\beta_2 \mathbin{@} \rho_{21} \in e_1} \dfrac{\cdots}{(\alpha_2\beta_2)\gamma_2 \mathbin{@} \rho_{22} \in e_2} \\ \Updownarrow & & \\ \dfrac{\cdots}{(\alpha_1\beta_1)\gamma_1 \mathbin{@} \rho_{12} \in e_2} < \dfrac{\cdots}{(\alpha_2\beta_2)\gamma_2 \mathbin{@} \rho_{22} \in e_2} & \vee & \dfrac{\cdots}{(\alpha_1\beta_1)\gamma_1 \mathbin{@} \rho_{12} \in e_2} = \dfrac{\cdots}{(\alpha_2\beta_2)\gamma_2 \mathbin{@} \rho_{22} \in e_2} \\ & & \wedge \\ & & \dfrac{\cdots}{(\alpha_1)\beta_1 \mathbin{@} \rho_{11} \in e_1} < \dfrac{\cdots}{(\alpha_2)\beta_2 \mathbin{@} \rho_{21} \in e_1} \end{array}$$

3.3 A Derivative for the LRGrep Calculus

With the big-step semantics in place, we now turn our attention to providing an equivalent small-step semantics operating directly on the stack of an LR parser.

Just like we introduced a matching relation first which we used to construct the richer capturing relation, we start with the matching derivative $d_M(k, s) \in \mathscr{P}(K)$ and follow with the capturing derivative $d_C(k, s) \in (\mathscr{P}(\text{Var}) \times K)^*$. They operate on a dialect of continuations $k \in K$ that generalises expressions and are derived with respect to states $s \in S$ of the LR automaton.

Both derivatives build upon Antimirov's derivative (Antimirov, 1996); see Section 1.3.2 for a short introduction. Before going into the details, let us review the main changes and their motivations.

Working with LR(1) States Derivation is performed with respect to a state of the LR(1) automaton rather than to a grammatical symbol. This is done because in practice we are matching against the stack of an LR(1) parser rather than a sentential form. Item filters and reductions are also easier to interpret on LR(1) states. This changes the alphabet the derivatives are operating on to S. The expressions, however, are still defined on symbols and we directly interpret a symbol as the set of LR(1) states with this incoming symbol and an item as the set of LR(1) states in which this item is *effective* (as defined by **items**, page 37).

Explicit Continuations The set of standard regular expressions is closed by derivation, but this is not the case for the reduction operator. The intermediate state of a reduction being recognised is generally not representable as a regular expression, so we define an explicit notion of continuation, "what remains to be matched". This also enables a more uniform treatment of variables and identification of successful matches (acceptance).

Right-to-left Matching For clarity and because this is central to our approach for recognising LR(1) stacks, we explicitly define *right* derivatives. The main change is to mirror the treatment of sequences $e_1 e_2$: e_2 is considered first, followed by e_1.

Ordered Continuations The disambiguation rules order the different ways a suffix can be matched. This translates to the image of the d_C derivative being a sequence rather than a set, to preserve the order between the different continuations. Given $\vec{k} = d_C(k, s)$ and $i < j < |\vec{k}|$, continuation $\vec{k}_i$ has a higher priority than $\vec{k}_j$. If both continuations succeed, the valuation associated with $\vec{k}_i$'s branch should be retained.

Variables to Update The d_C derivative yields the set of variables to update along each continuation. In the big-step semantics, we had to keep track of valuations mapping variables to input positions. The small-step semantics has no knowledge of the input position nor of the bound variables; it is solely concerned with the effects of a single transition. The rest of the interpretation is left to a *meta* level that feeds the input symbol-by-symbol to d_C, keeping track of the input position, of the active continuations and of the valuation associated with each continuation. When $d_C(k, s)_i = (V, k')$, k' should inherit the valuation associated to k with all variables $v \in V$ updated to the current position.

Finite Derivatives We say that the derivatives are finite if, starting from an arbitrary continuation, iterating the derivative function reaches only a finite number of continuations.

A technicality in the definition of Brzozowski's derivative is that regular expressions had to be considered at least modulo associativity, commutativity and idempotence (ACI) of disjunction for the derivatives to be finite. Antimirov's derivative relaxed this restriction: its image is a set of regular expressions that can be seen as a

form of normalisation modulo ACI of top-level disjunction, but the regular expressions themselves can be considered syntactically.

If we are solely interested in the semantics alone, it is not important that the derivatives are finite. For instance, to see if continuation k matches $\vec{s}$ with d_M, it suffices to iterate the function until reaching the end of the vector. Let us consider the function d^*_M defined by $d^*_M(\kappa, \epsilon) = \kappa$ and $d^*_M(\kappa, \vec{s}' \cdot s) = d^*_M(\bigcup_{k \in \kappa} d_M(k, s), \vec{s}')$. Then the interpretation of k on stack $\vec{s}$ is given by the set $d^*_M(\{k\}, \vec{s})$.

Finite derivatives are relevant for generating an automaton. If we know that, when starting from an initial continuation, iterated application reaches only a finite set of derivatives, we can use this set (together with the initial continuation) as the states of the automaton and the derivative function as the transition function.

Since we want, ultimately, to compile our expressions to an automaton, we need d_C to have finite derivatives. Like Antimirov's derivative, we consider continuations syntactically. However, we do not have a set but a sequence of pairs of a set of variables and a continuation. For the derivatives to be finite, we restrict the image to sequences of unique continuations. The variables are not considered in this process—equivalence is determined solely by the syntactic equality of continuations. This is done explicitly in the definition of the derivative by the use of a normalising concatenation operator, noted $\odot$, which keeps only the first occurrence, in the order of indices, of a continuation.

It is fine to keep only the first occurrence of a continuation, even if the other occurrences appear bound with different variables, because the sequence represents the continuations that should be matched against the remainder of the input in order of priority. Bound variables do not affect the language recognised by a continuation; they are only observed when matching succeeds. A continuation that appears twice necessarily recognises the remainder in the same way, accepting it or rejecting it twice. If it is accepted, the first occurrence has the priority and only its valuations are used.

Acceptance condition When using Brzozowski's or Antimirov's derivatives to build an automaton, accepting states are determined by checking if the empty string ϵ is in their denotation. This can be done syntactically using the c function. In our dialect, the filter operator introduces additional complexity, making it insufficient to simply check for ϵ in the denotation of a state. Specifically, given an expression $/\, A \to \alpha \cdot \beta$, we might have $(\gamma)\epsilon \in /\, A \to \alpha \cdot \beta$ for some γ, but accepting immediately does not check that the prefix γ passes the filter.

To address the challenges posed by the filter operator, we propose an alternative approach that involves following an extra transition to determine acceptance. Specifically, instead of relying on a syntactic check, we represent acceptance using a distinguished continuation noted $\blacksquare$. Input is accepted when following a transition targeting this continuation. The conditions imposed by an eventual item are moved to the label of a transition; if there are no conditions, there will simply be a transition for each label.

As a result, acceptance is always delayed by one transition, which may seem

$$\begin{array}{lll} k \in K & ::= & k \bullet e \mid k \bullet R(Q, Q_T) \mid k \bullet !v \mid \square \mid \blacksquare \\ Q, Q_T & \in & \mathscr{P}(\mathbf{RA}) \setminus \{\emptyset\} \end{array}$$

Figure 3.1: Syntax of continuations

counterintuitive at first but is actually necessary to accurately capture the behaviour of filtered expressions and does not affect correctness for other operators. The only case that could be problematic is when reaching the end of the input: what if there is no room for delaying by an extra symbol? Fortunately, the first state of the LR stack is the initial state, making the stack one symbol longer than the corresponding sentential form. So it is always fine to delay by one symbol.

3.3.1 Continuations

The syntax of continuations is given in Figure 3.1. The continuations are linear and managed like a stack: they are used to stash the pending work discovered during derivation, but they do not introduce new control structures.

The continuation $k \bullet e$ means that e should match first (against the top of the stack), followed by k. The expression appears on the right and the continuation on the left to reflect the right-to-left matching.

The binding continuation $k \bullet !v$ means that the current position of the input is to be saved in variable v, after which matching continues with k.

$\square$ is used to mark the bottom of the continuation stack and $\blacksquare$ represents acceptance. $\square$ derives to $\blacksquare$: once reaching the bottom of the stack, all pending work has been done and the acceptance condition is satisfied. To match an expression e, it is lifted to $\square \bullet e$. For instance, let us look at $\square \bullet \, / A \to \alpha \cdot \beta$. Deriving with respect to s succeeds if $A \to \alpha \cdot \beta \in \mathbf{items}(s)$. In this case, $/ A \to \alpha \cdot \beta$ is removed from the stack and derivation reaches $\square$, producing the new continuation $\blacksquare$: the expression has been recognised.

A reduction continuation $k \bullet R(Q, Q_T)$ represents an intermediate step in the process of recognising the suffixes reached by a sequence of reductions. Q and Q_T are non-empty sets of states of the reduction automaton. Q is the set of active states, representing the current positions in a non-deterministic traversal of the reduction automaton, and Q_T is the set of target states. In other words, $k \bullet R(Q, Q_T)$ recognises sequence of states leaving Q for which there is a path to Q_T. If a state in Q_T is reached, a match has been identified and derivation proceeds with k. The equations governing the derivation of reductions are treated separately in Section 3.3.4.

3.3.2 The matching derivative

The matching derivative is the least fixed point of the equations given in Figure 3.2. Deriving a continuation $k \bullet e$ is done by analysing e:

- If e is empty, the derivative is the derivative of the continuation k.

$$
\begin{array}{rcl}
d_M(k \bullet \epsilon, s) & = & d_M(k, s) \\
d_M(k \bullet x, s) & = & \begin{cases} \{k\} & \text{if } x = \textbf{incoming}(s) \\ \emptyset & \text{otherwise} \end{cases} \\
d_M(k \bullet e_1 \cdot e_2, s) & = & d_M(k \bullet e_1 \bullet e_2, s) \\
d_M(k \bullet e_1 | e_2, s) & = & d_M(k \bullet e_1, s) \cup d_M(k \bullet e_2, s) \\
d_M(k \bullet e^*, s) & = & d_M(k \bullet (\epsilon \mid e^* \cdot e, s)) \\
d_M(k \bullet / A \to \alpha.\beta, s) & = & \begin{cases} d_M(k, s) & \text{if } A \to \alpha.\beta \in \textbf{items}(s) \\ \emptyset & \text{otherwise} \end{cases} \\
d_M(\square, s) & = & \{\blacksquare\} \\
d_M(\blacksquare, s) & = & \emptyset
\end{array}
$$

Figure 3.2: The matching derivative

- The derivative of a symbol x with respect to state s is the singleton continuation k if x is the incoming symbol of the state s and is empty otherwise.
- A concatenation $k \bullet e_1 \cdot e_2$ is flattened to a continuation $k \bullet e_1 \bullet e_2$, and derivation proceeds on the flattened continuation.
- For a disjunction $e_1 | e_2$, each side is derived independently and the union of both sets of continuations is returned.
- For the Kleene star e^*, the derivation proceeds on the expansion $\epsilon \mid e^* \cdot e$. Some care needs to be taken when implementing it as the recursive call is non-decreasing. Consider the following term with a cyclic expansion:

$$
\begin{array}{rcl}
\underline{d_M(k \bullet \epsilon^*, s)} & = & d_M(k \bullet (\epsilon \mid \epsilon^* \cdot \epsilon, s) \\
& = & d_M(k \bullet \epsilon, s) \cup d_M(k \bullet \epsilon^* \cdot \epsilon, s) \\
& = & d_M(k \bullet \epsilon, s) \cup d_M(k \bullet \epsilon^* \bullet \epsilon, s) \\
& = & d_M(k \bullet \epsilon, s) \cup \underline{d_M(k \bullet \epsilon^*, s)}
\end{array}
$$

  In practice, visits of a Kleene star are memoized before expansion, and a second encounter during the traversal of the same expression is just skipped.
- The item filter $/ A \to \alpha \cdot \beta$ is treated by looking at $\textbf{items}(s)$, the effective items of s (Definition 11, page 37). If $A \to \alpha \cdot \beta$ is in $\textbf{items}(s)$, the derivation proceeds on the continuation. Otherwise, the derivative is empty.
- Finally, the two base cases: deriving $\square$ results in $\blacksquare$. Deriving $\blacksquare$ has no continuation.

Deriving with respect to sequences We lift d_M to work on sequences:

$$
\begin{array}{rcl}
d_M(k, \epsilon) & = & \{k\} \\
d_M(k, \vec{s} \cdot s) & = & \bigcup \{\, d_M(k', \vec{s}) \mid k' \in d_M(k, s) \,\}
\end{array}
$$

$$
\begin{array}{rcl}
d_C(k \bullet \epsilon, s) & = & d_C(k, s) \\
d_C(k \bullet x, s) & = & \begin{cases} (\emptyset, k) & \text{if } x = \textbf{incoming}(s) \\ \epsilon & \text{otherwise} \end{cases} \\
d_C(k \bullet e_1 \cdot e_2, s) & = & d_C(k \bullet e_1 \bullet e_2, s) \\
d_C(k \bullet e_1 | e_2, s) & = & d_C(k \bullet e_1, s) \odot d_C(k \bullet e_2, s) \\
d_C(k \bullet e^*, s) & = & d_C(k \bullet (\epsilon \mid e^* \cdot e, s)) \\
d_C(k \bullet / A \to \alpha \cdot \beta, s) & = & \begin{cases} d_C(k, s) & \text{if } A \to \alpha \cdot \beta \in \textbf{items}(s) \\ \epsilon & \text{otherwise} \end{cases} \\
d_C(k \bullet n = e, s) & = & d_C(k \bullet !n_{\mathrm{s}} \bullet e \bullet !n_{\mathrm{e}}, s) \\
d_C(k \bullet !v, s) & = & \mathit{bind}(d_C(k, s), v) \\
d_C(\square, s) & = & (\emptyset, \blacksquare) \\
d_C(\blacksquare, s) & = & \epsilon
\end{array}
$$

Figure 3.3: The capturing derivative

3.3.3 The capturing derivative

The capturing derivative is defined as the least solution to a similar set of equations but handles the whole language. We write $\kappa, \kappa' \in (\mathscr{P}(\mathrm{Var}) \times K)^*$ for elements of the image and $V \in \mathscr{P}(\mathrm{Var})$ for sets of variables. The images are sequences of pairs of a set of variables and a continuation, where the variables are the ones to update when reaching the continuation. The order of elements in the sequence implement the disambiguation rules, such that earlier continuations have higher priority.

Figure 3.3 gives the equations for all cases except the reduction operator, which is left to Section 3.3.4. Two helpers are used define the derivative:

- To ensure that the derivatives are finite (see paragraph 3.3 page 66), $\kappa \odot \kappa'$ concatenates and normalises sequences. When reading the equations, just think of it as plain concatenation, the precise definition is given below.
- Variable updates are introduced with $\mathit{bind}(\kappa, v)$ which adds v to the set of variables of all elements of κ:

$$\mathit{bind}(\kappa, v)_i = (\{v\} \cup V, k) \text{ where } (V, k) = \kappa_i$$

Compared to the matching derivative, the differences are:

- In all cases, order matters; the image is a sequence, not a set.
- In the image, each continuation is paired with a set of variables to update. The leaf cases return an empty set of captures, the variable case augments the variable sets with a new variable.
- In the disjunction case, the two sub-sequences are concatenated, such that the order of continuations reflects the disambiguation rule (continuations of e_1 are prioritised over continuations of e_2).

- As in d_M, an implementation should treat the Kleene star equations carefully.
- The derivation of $k \bullet n = e$ wraps e between two variable binding continuations for the starting and ending positions of e.
- $k \bullet !v$ is handled by deriving k and using *bind* to update variable v.

Normalised sequences

The normalising concatenation $\odot$ strips the continuations that are already in the left-hand side from the right-hand side, or simply skips the right-hand side if the left one already has a success continuation $\blacksquare$. It can be defined as follows:

$$\kappa \odot \kappa' \quad = \quad \kappa \cdot \text{prune}(\kappa')$$

where:

$$\begin{aligned} \text{prune}((V,k)\cdot\kappa') &= \begin{cases} \epsilon & \text{when } \exists i, V',\ \kappa_i = (V', \blacksquare) \\ \text{prune}(\kappa') & \text{when } \exists i, V',\ \kappa_i = (V', k) \\ (V,k)\cdot\text{prune}(\kappa') & \text{otherwise} \end{cases} \\ \text{prune}(\epsilon) &= \epsilon \end{aligned}$$

In a normalised sequence, a continuation occurs at most once and the success continuation $\blacksquare$ can appear only at the last position. ϵ and all singletons are already normalised, and $\odot$ produces a normalised sequence when applied to two normalised sequences.

3.3.4 Deriving reductions

We still have to define the derivation of continuations $k \bullet [\, e\,]$ and $k \bullet R(Q, Q_T)$. e is first translated to a set $Q_T \subset \mathbf{RA}$ of states of the reduction automaton (see Section 2.3.2) whose suffixes match e. This set offers a low-level, semantic view of the sequence of the reductions described by e. Derivation then follows paths of **RA** reaching a state in Q_T.

Arithmetic example To derive $[\, E \,/\, F \to E \cdot\,]$ we start by deriving $E \,/\, F \to E \cdot$ with the matching derivative. Looking at the LALR automaton for G_A (Figure 1.3, page 34), we see that the only state for which the matching derivative is non-empty is E_2:

$$\begin{aligned} d_M(\square \bullet E \,/\, F \to E \cdot, E_2) &= d_M(\square \bullet E \bullet \,/\, F \to E \cdot, E_2) \\ &= d_M(\square \bullet E, E_2) \\ &= \{\square\} \\ d_M(\square \bullet E \,/\, E \to E \cdot, s) &= \emptyset \text{ when } s \neq E_2 \end{aligned}$$

The only predecessor of E_2 in the LR(1) automaton is $(_0$ and deriving again with respect to $(_0$ succeeds: $d_M(\square, (_0) = \{\blacksquare\}$. We conclude that the target states are:

$$Q_T = \left\{\, s \in \mathbf{RA} \;\middle|\; \exists T', s = \texttt{Suffix}((_0, E_2, T') \,\right\}$$

Derivation should follow paths of the reduction automaton starting from the initial state and reaching the states in Q_T.

Defining the derivative on reductions

$$
\begin{array}{rcl}
d_C(k \bullet [e], s) & = & \begin{cases} d_C(k \bullet R(\{\texttt{Initial}\}, Q_T), s) & \text{when } Q_T \neq \emptyset \\ \epsilon & \text{otherwise} \end{cases} \\
 & & \text{where} \quad Q_T \quad = \quad \left\{ \texttt{Suffix}(s, \vec{s}', T') \in \mathbf{RA} \;\middle|\; \blacksquare \in d_M(\square \bullet e, s\vec{s}') \right\} \\
 & & \\
d_C(k \bullet R(Q, Q_T), s) & = & \begin{cases} d_C(k, s) \odot k' & \text{when } Q \cap Q_T \neq \emptyset \\ k' & \text{otherwise} \end{cases} \\
 & & \text{where} \quad k' \quad = \quad \begin{cases} k \bullet R(Q', Q_T) & \text{when } Q' \neq \emptyset \\ \epsilon & \text{otherwise} \end{cases} \\
 & & \qquad\quad Q' \quad = \quad \text{progress}(Q, s, Q_T) \\
\text{progress}(Q, s, Q_T) & = & \bigcup_{q \in \text{closure}(Q)} \left\{ q' \in \mathbf{RA} \;\middle|\; \exists q_T \in Q_T, q \xrightarrow{s} q' \rightarrow^* q_T \right\}
\end{array}
$$

Figure 3.4: Deriving reductions

Figure 3.4 gives the equations characterising derivation of reductions.

The derivation of $k \bullet [e]$ is initiated by translating e to the states Q_T recognising the reductions matched by e and proceeds to deriving $k \bullet R(\{\texttt{Initial}\}, Q_T)$, if Q_T is not empty.

To derive a continuation $k \bullet R(Q, Q_T)$, we start by checking if a target state was reached ($Q \cap Q_T \neq \emptyset$). If this is the case, then some paths of the reductions being simulated succeeded and we continue by deriving k.

k' represents the configurations reached by progressing in the reduction automaton. $Q' = \text{progress}(Q, s, Q_T)$ list the states that are one step further towards the targets. If Q' is empty, the reduction failed to reach a target state and we abort the continuation. $\text{progress}(Q, s, Q_T)$ walks the reduction automaton, starting from states in Q and following the transitions labelled s on a path to states in Q_T. Since **RA** is an ϵ-NFA, the closure of Q is taken before looking for a transition. If the wildcard transition optimisation is used (see Section 2.4.1), transitions labelled _ should also be followed.

3.4 Relating both semantics

We can relate the matching relations with the derivatives, focusing on the d_M function and $(\alpha)\beta \in e$ relation to simplify the discussion.

Claim 1 *Let $\vec{s} \cdot s \cdot \vec{s}'$ be the stack of an LR parser. Then, we have:*

$$\blacksquare \in d_M^*(\square \bullet e, s \cdot \vec{s}') \Longleftrightarrow (\alpha)\beta \in e$$

where:

$$\begin{array}{rcl} \alpha & = & \textbf{\textit{incoming}}(\vec{s} \cdot s) \\ \beta & = & \textbf{\textit{incoming}}(\vec{s}') \end{array}$$

That is, the d_M function can be used to find the stack suffixes matched by an expression e, decomposing the stack into a prefix ($\vec{s} \cdot s$) and a suffix ($\vec{s}'$) that coincide with the prefix (**incoming**($\vec{s} \cdot s$)) and suffix (**incoming**($\vec{s}'$)) recognized by the matching relation $(\alpha)\beta \in e$.

If the LR automaton has undergone conflict resolution (some actions have been removed), the claim weakens to:

$$\blacksquare \in d_M^*(\square \bullet e, s \cdot \vec{s}') \implies (\alpha)\beta \in e$$

Missing reductions can prevent some sentential forms from matching.

Conclusion

In this chapter we introduced a calculus designed to classify the stacks of failing LR parsers by extending the reduce-filter patterns with the regular expression operators and variable bindings.

We gave a big-step semantics relating expressions to sentential forms which serves as a reference specification of the calculus. But the main take away is the derivative d_C which implements a small-step semantics directly on the stack of an LR parser. It is the basis of the compilation strategy of the next chapter.

Chapter 4

Compilation to matching automata

In the previous chapter, we established a dialect of regular expressions that is suitable for analysing the stack of LR parsers and we defined an appropriate notion of derivation. In this chapter, we use derivation to construct and optimise automata that recognise multiple expressions at once.

Let us recall the notions used in this chapter:

- We are given an LR(1) automaton with states $s \in S$, an initial state noted s_0, and a function **pred** : $S \to \mathscr{P}(S)$ that maps a state to the set of its predecessors. $s' \in$ **pred**(s) means that there exists a transition from s' to s. We also have a partial function **incoming** : $S \rightharpoonup N \cup T$ that gives the symbol labelling the incoming transitions of a state. It is not defined on the initial state; this state has no incoming transition.

- For our dialect of regular expressions, we have defined a notion of continuations K equipped with a derivation function $d_C : K \times S \to (\mathscr{P}(\text{Var}) \times K)^*$. A regular expression e is lifted to a continuation $\square \bullet e$, which can then be derived. Matching succeeds when derivation reaches a distinguished continuation written $\blacksquare$.

$d_C(k, s)$ is an ordered sequence of pairs (V, k'), where k' is a continuation and V is a set of variables. Each continuation can be seen as a state of a non-deterministic automaton with d_C as the transition function. $\square \bullet e$ is the initial state and $\blacksquare$ is the only accepting state. Each pair (V, k') of $d_C(k, s)$ is a transition leaving state k with two labels:

- s is the input symbol allowing to take this transition.
- V are the variables in which to store the current position of the input.

An input $\vec{s}$ is recognised if there is a path from the initial state to $\blacksquare$. From this path, we can construct a mapping from variables to input positions by remembering the last position where a variable $v \in \text{Var}$ was encountered in the sets V annotating the transitions of the path.

Contrary to a standard NFA, the outgoing transitions of a state are ordered to disambiguate matches. If there are multiple paths labelled $\vec{s}$ from the initial state to ■, the matching is ambiguous as each path potentially gives a different variable mapping. To disambiguate between two paths, we favour the one that picked the earliest transitions. That is, the two paths have a shared prefix starting from the initial state after which they diverge. Assume that the shared prefix ends at some state k where the two paths picked a different successor in $d_C(k, s)$, say at indices i and j. The less ambiguous path is the one which picked the successor at index $\min(i, j)$.

Our goal is to obtain an efficient deterministic matching automaton that computes the same result in a single pass. The main steps are:

1. *Determinisation.* We devise a notion of determinisation that preserves the disambiguation semantics and recognises multiple expressions at the same time. In essence, we augment the standard subset construction to a "sub-sequence construction" that preserves the relative order of continuations along transitions.

2. *Exploration.* The standard subset construction for an automaton with states Q comes with two variants:

   - A systematic one, defining a new automaton with states $\mathscr{P}(Q)$, resulting in an automaton that is always exponentially larger.
   - For practical purposes, a "trimmed" one, where only the states $Q' \subset \mathscr{P}(Q)$ that are reachable from the initial state are considered.

   Trimming is relevant in our case, but we can go further. Since the automaton is used to classify the stacks of a failing parser, it only needs to handle viable stacks, not arbitrary sentences in S^*. We condition the construction to only explore the states reachable from the initial state via viable stacks.

3. *Register allocation and dead code elimination.* A state of the determinised automaton is a sequence of continuations and a naive interpreter reaching this state has to maintain a separate variable mapping for each continuation. We can show that some continuations never succeed, while for those that succeed, the different variables are often bound to the same value and do not need to be stored separately. Using data flow analyses, we drop variables belonging to dead continuations, group variables guaranteed to be equal, and compute the minimal storage needed by an optimised interpreter.

4. *Minimisation.* The deterministic automaton that we get after all these steps often contains redundant parts—distinct states that always perform the same sequence of observable actions; those can be merged to obtain a more compact automaton.

5. *Code generation.* The final step is to turn the automaton into a small program. Transitions are stored in a sparse table and the actions are described in a simple imperative language.

With the exception of the *exploration* step, this is a standard pipeline for high-performance lexer generators. Borsotti and Trafimovich (2022) explains the TDFA construction, a flavour of automaton which also features variable bindings and disambiguation, that powers the RE2C lexer generator. Their starting point is a non-deterministic automaton called "TNFA" built using Thompson-like constructions. It is comparable to our interpretation of d_C as the transition function of an NFA. They determinise it and go into the details involved in optimising the automaton and generating high-performance code.

Our focus is not the performance of the generated code but its size. The intermediate automata produced from non-trivial LRGrep expressions tend to be bigger than those produced from lexer specifications, so it was important for us to manage the size by tweaking some optimisations. For instance, on our error specification for the OCaml grammar, the data flow analyses made the automaton more compressible by a factor of 2 to 3. The code generation part is also not designed to produce the highest-performing code, but to represent the automaton using a simple and compact bytecode easy to integrate with an OCaml program.

While our goals differ, the pipelines are very similar with slightly different choices made in the different steps.

4.1 Determinisation

Before introducing our solution, let us look at how the subset construction works and the new challenges brought by disambiguation and variables.

From subsets to subsequences

In traditional finite automata theory, determinisation is achieved through the subset construction. This construction tracks the states that are simultaneously reachable when following transitions with the same label. Our extension of finite automata affects the construction in two ways:

- Disambiguation rules impose an order over the different paths and sometimes cause paths to abort.
- Variable bindings induce a finer relation between the states than mere reachability: we do not just need to know that two states are simultaneously reachable, but also the different variables updated on the transitions reaching them.

The standard subset construction To simulate an NFA with states Q, for instance to recognise a word `abc`, one can keep track of the set of simultaneously reachable states and update them in lockstep, symbol by symbol. Starting from the singleton made up of the initial state, all transitions labelled `a` are followed, reaching a new set of states. The process repeats for the subsequent labels, `b` then `c`, following the

labelled transitions leaving each state reached in the previous step. Recognition succeeds if at least one active state is accepting at the end of the word[1].

At each step, an active configuration is an element of $\mathscr{P}(Q)$. The essence of the subset construction is that we can construct a deterministic automaton by repeating this process for all transitions of all states, which is guaranteed to terminate since $\mathscr{P}(Q)$ is finite.

Disambiguation using non-repeating sequences In our extension, the transitions are ordered and we have to track the relative order of states, not just their reachability. A configuration is now a sequence of NFA states without repetition. A deterministic transition follows the transitions of each non-deterministic state while preserving this order: the first element expands into a new sequence of states, obtained by following the transitions in order; the second element to a second sequence; and so on. We collapse this sequence of sequences and remove the duplicates, keeping only the first occurrence, to obtain a new sequence of unique NFA states.

We replaced the powerset with sequences without repetition, which is still a finite set. As with the standard subset construction, we can close over all transitions to produce a new DFA that now also preserves the relative order of NFA states.

Simulating NFAs with disambiguation and variables To implement variables, on top of reachability and relative order, we want to keep track of a valuation ρ for each active state k. ρ maps variables to input positions and is updated each time a transition is followed.

A configuration now tracks the active states as a sequence $\vec{k}$, the current position as an integer, and a valuation ρ for each active state, represented as a second sequence $\vec{\rho}$ such that $|\vec{\rho}| = |\vec{k}|$. When following a transition labelled s, we update the sequence of states as before, increment the current position, and take care of the valuations as follows. A state k' is in the sequence of states expanded from state k because there is a non-deterministic transition (V, k') labelled s leaving k. k' receives a valuation ρ' obtained by updating ρ with variables V to the current position. This process expands the sequence of valuations into a sequence of sequences, mirroring the treatment of active states. To keep it synchronised with the sequence of states, we collapse it and remove the elements associated with the duplicate states that were removed from the sequence of states.

Determinising NFAs with disambiguation and variables To determinise this NFA, we proceed as before using sequences of states, but we also need to keep enough information to maintain the sequence of valuations during interpretation. A transition labelled s of the determinised automaton relates two sequences of NFA states. Each state k' in the target sequence comes from a non-deterministic transition (V, k') of a certain state k in the source sequence, and its valuation should be the valuation of

[1] This approach is at the core of an efficient non-deterministic recogniser by Rob Pike and reported by Russ Cox as the "PikeVM" on https://swtch.com/~rsc/regexp/regexp2.html.

k with variables V updated. To update the sequence of valuations, the deterministic transition also has to provide the index of k, so that its valuation can be reused, and the variables V to update.

We can finally define the DFA handling disambiguation and variable bindings: its states are sequences of unique NFA states and its transitions relate a state and a label to a target state and a sequence of indices and variables. This sequence has the same length as the target state, and the indices are valid indices of the source state.

Formalising the solution

Provided a non-deterministic automaton on alphabet Σ with states Q and a transition function $d : Q \times \Sigma \to \mathscr{P}(Q)$, determinising it with the subset construction can be seen as lifting d to $d' : \mathscr{P}(Q) \times \Sigma \to \mathscr{P}(Q)$, the transition function of a deterministic automaton with states $\mathscr{P}(Q)$.

$d'(Q', s)$ is obtained by applying $d(-, s)$ on each element $q' \in Q'$, yielding a set in $\mathscr{P}(\mathscr{P}(Q))$ which is then flattened:

$$d'(Q', s) = \bigcup_{q' \in Q'} d(q', s)$$

This amounts to tracking all the NFA states that are simultaneously reachable.

In LRGrep's setting, the procedure is essentially the same—lifting to a richer domain by deriving pointwise, then flattening to recover an image in the same domain. But directly flattening would lose too much information.

The non-deterministic transitions are given by:

$$d_C : K \times S \to (\mathscr{P}(\mathrm{Var}) \times K)^*$$

Rather than lifting to a set, we lift to a sequence of sequences, still paired with variable updates, which we flatten and de-duplicate while remembering the index of source continuations:

$$\begin{array}{rl} K \times S \to (\mathscr{P}(\mathrm{Var}) \times K)^* & \\ \Downarrow & \text{– lift to a sequence of sequences} \\ K^* \times S \to (\mathscr{P}(\mathrm{Var}) \times K)^{**} & \\ \Downarrow & \text{– flatten and index} \\ K^* \times S \to (\mathbb{N} \times \mathscr{P}(\mathrm{Var}) \times K)^* & \end{array}$$

Definition 13 (Determinised derivative) *The d_C derivative is made deterministic by lifting it to sequences and flattening while preserving source indices:*

$$\begin{array}{rcl} d'_C : K^* \times S & \to & (\mathbb{N} \times \mathscr{P}(Var) \times K)^* \\ \langle k_0 \dots k_{n-1}, s \rangle & \mapsto & \bigodot_{0 \le i < n} \left[(i, V, k') \;\middle|\; (V, k') \in d_C(k_i, s) \right] \end{array}$$

We name the three projections of images of d_C:

$$\begin{array}{rcl} \textbf{\textit{dest}} & = & map\ \pi_3 \cdot d'_C : K^* \times S \to K^* \\ \textbf{\textit{vars}} & = & map\ \pi_2 \cdot d'_C : K^* \times S \to \mathscr{P}(Var)^* \\ \textbf{\textit{origin}} & = & map\ \pi_1 \cdot d'_C : K^* \times S \to \mathbb{N}^* \end{array}$$

They represent the destination of a transition, the variables updated along it and the indices of the source continuation from which each continuation of the destination originated.

In the definition of d'_C, the comprehension written with square brackets [...] applies to a sequence and should be understood as preserving order:

- Each continuation k_i is derived, and its index is prepended to the elements of the sequence $d_C(k_i, s)$.
- The sequence of sequences is collapsed and normalised using a variant of the normalising concatenation $\odot$, which we introduce below. The normalisation process ensures that continuations are unique—a continuation appears at most once in the resulting sequence—which is necessary for the procedure to terminate.

The three projections describe the transitions of a deterministic automaton and the auxiliary information associated with them. Since the automaton is deterministic, a transition is described by a source state $\vec{k}$ and a label s. If $\vec{k}' = \mathbf{dest}(\vec{k}, s)$, then:

- $\vec{k}'$ is the target state of transition $(\vec{k}, s)$
- $\mathbf{vars}(\vec{k}, s)$ lists the variables to update for each continuation of $\vec{k}'$
- $\mathbf{origin}(\vec{k}, s)$ is a vector such that $\mathbf{origin}(\vec{k}, s)_j = i$ when $\vec{k}'_j$ is derived from $\vec{k}_i$

Normalising indexed transitions Normalisation proceeds as in 3.3.3 but working on triplets (i, V, k) this time. We use κ, κ' for a sequence of those triplets:

$$\kappa \odot \kappa' \quad = \quad \kappa \cdot \text{prune}(\kappa')$$

where:

$$\begin{aligned} \text{prune}((i, V, k) \cdot \kappa') &= \begin{cases} \epsilon & \text{when } \exists j, i', V',\ \kappa_j = (i', V', \blacksquare) \\ \text{prune}(\kappa') & \text{when } \exists j, i', V',\ \kappa_j = (i', V', k) \\ (i, V, k) \cdot \text{prune}(\kappa') & \text{otherwise} \end{cases} \\ \text{prune}(\epsilon) &= \epsilon \end{aligned}$$

Recognising multiple expressions

To recognise a single expression e, we can now construct a deterministic automaton starting from the initial state $\square \bullet e$ and closing over the transitions described by **dest**.

In practice, it is desirable to build automata that recognise more than one expression. In the opening example, we wanted to recognise two expressions. As pseudo-code:

```
if stack matches /I -> . i then
  report "Expecting a number"
else if stack matches [E/I->(E.)] then
  report "Expecting a closing parenthesis"
```

Rather than testing for both expressions separately with two independent automata, we can generate a single automaton that matches for both at the same time and outputs the index of the one that succeeded.

We achieve this by generalising the continuations $\square$ and $\blacksquare$ to a family $\square_i$ and $\blacksquare_i$ indexed by $i \in \mathbb{N}$. Derivation is such that $d_C(\square_i, s) = (\emptyset, \blacksquare_i)$. All $\blacksquare_i$ are accepting states and normalisation cuts sequences at the first continuation $\blacksquare_i$ for some i.

$$k \in K \quad ::= \quad \ldots \mid \square_i \mid \blacksquare_i \qquad\qquad i \in \mathbb{N}$$

To recognise a vector of expressions $e_0 \ldots e_{n-1}$, we start the determinisation procedure from state $(\square_0 \bullet e_0) \cdot \ldots \cdot (\square_{n-1} \bullet e_{n-1})$.

Related work

To our knowledge, this is the first presentation of deterministic disambiguation as a reformulation of the subset construction on sequences of unique elements.

The determinisation of TNFA in Trofimovich (2019), and Borsotti and Trafimovich (2022) is very close in essence. They use strict weak orderings rather than non-repeating sequences. Strict weak orderings give more degrees of freedom by allowing states and transitions to be partially ordered. We could achieve this in our framework by changing the representation of states from the non-repeating subset of K^* to the subset of $\mathscr{P}(K)^* / \{\emptyset\}$ where a continuation appears at most once. Continuations that belong to the same set are not ordered. Rather than reframing the subset construction on strict weak orderings, they directly give an algorithm to construct strict weak orderings while closing over transitions.

They implement two disambiguation strategies with this representation: leftmost, a variant of ours, and the more challenging POSIX strategy. We did not study whether POSIX could be reframed in our setting. Since we are only interested in leftmost-like disambiguation strategies, it is not clear whether their approach would be beneficial. There are fewer non-repeating sequences than strict weak orderings[2], so in the worst case our determinised automaton would be smaller (still exponential) but their approach might visit fewer states in practice. Picking the best solution would require more experimentation.

Another branch of research explores the problem of full parsing for regular expressions, which constructs a parse tree rather than a valuation. Parsing offers more flexibility but is less performant and can require multiple passes over the input.

To recognise a single expression e, we can now construct a deterministic automaton starting from $\square \bullet e$ and closing over the transitions described by **dest**.

[2]The number of strict weak orderings on n elements is the nth ordered Bell number $\sum_{i=0}^{n} \left\{ {n \atop i} \right\} i!$.

The number of non-repeating sequences of n elements is given by $\sum_{i=0}^{n} \binom{n}{i} i!$.

4.2 Exploration

From the initial state $q_0 = (\square_0 \bullet e_0) \cdot \ldots \cdot (\square_{n-1} \bullet e_{n-1})$ and the transition function **dest**, it is easy to construct an automaton $M = (Q, S, d, q_0, F)$, such that:

- Its states are the subset Q of K^* reachable by following transitions from q_0.
- Its alphabet is the set of LR states.
- Its transition function is **dest**; as defined, the function is total and ϵ acts as a sink state.
- Its initial state is $q_0 \in Q$.
- A state is accepting if it contains an accepting continuation:

$$F = \left\{ q \in Q \;\middle|\; \exists i, j,\ q_i = \blacksquare_j \right\}$$

Formally, Q can be constructed iteratively, starting from the singleton $\{q_0\}$ and growing it with the states reachable by following one transition:

- $Q_0 = \{q_0\}$
- $Q_{i+1} = Q_i \cup \bigcup_{q \in Q_i, s \in S} \mathbf{dest}(q, s)$

The sequence reaches a fixed point such that $\exists j, Q_{j+1} = Q_j$ and we pose $Q = Q_j$. Informally, the existence of a fixed point follows from Antimirov's derivative having finite derivatives, a property preserved by our extensions:

- The reduction continuations are represented as pairs of sets of states of the reduction automaton, so they are finite by construction.
- The second extension is the replacement of the set of continuations with the set of non-repeating sequences. The set of non-repeating sequences of a finite set is bigger than the power set but remains finite.

Thus, we obtain a finite automaton with states Q recognising the initial vector of expressions. In practice, however, this automaton is often impractically large. Fortunately, we can do better: it classifies all elements of S^* while we are only interested in recognising LR stacks. We leverage this to significantly prune the automaton.

4.2.1 The subset reached by viable stacks

The automaton we are looking for is the subset of M that behaves the same as M on the stacks of an LR parser. In Section 2.3.1, we observed that the viable stacks are an approximation of these stacks that can be enumerated from right-to-left by starting from each state and iterating the **pred** function.

To properly classify the errors, LRGrep also requires that no reduction has been applied on a stack since the last shift transition. In such a stack, the last transition

cannot be a goto transition and we can further restrict the enumeration by starting from states whose incoming transitions are not goto transitions. We call these the *wait* states because they are states in which the LR parser waits for the lexer to provide a new input symbol before taking any action[3].

Definition 14 (Wait states) *A wait state is either the initial state or the target of a shift transition. We define the set of wait states* $\boldsymbol{Wait} \subseteq S$ *as:*

$$\boldsymbol{Wait} = \{s_0\} \cup \left\{ s \in S \middle| \boldsymbol{incoming}(s) \in T \right\}$$

We can refine M by computing the subset of states and transitions that are reachable from the stacks ending in a wait state. This construction coincides with an implication rather than an intersection. If $\vec{s}$ is a viable stack, the refined automaton applied to $\vec{s}$ should behave like M; if it is not a viable stack, any behaviour is acceptable. With an intersection, non-viable stacks would be systematically rejected. For two automata A and B on an alphabet Σ, the intersection $A \cap B$ can grow to a size up to $|A| \times |B|$ states, while implication permits decreasing the size of the automaton. At worst, B itself is a valid approximation of $A \implies B$.

Implication is just an optimisation, but a significant one in this domain. In particular, to recognise an occurrence of the reduction operator, the unrestricted definition leads to simulating all sequences of reductions permitted by the grammar and eventually reaching the target. When restricted to viable stacks, only the reductions applicable to the local context are simulated.

The implied automaton We represent the result of the implication with a partial function $\mathbf{Dom} : Q \rightharpoonup \mathscr{P}(S)$ characterising the reachable subset of M. The reachable states are given by $\mathrm{dom}(\mathbf{Dom})$ and (the labels of) the reachable transitions of a state $q \in \mathrm{dom}(\mathbf{Dom})$ by $\mathbf{Dom}(q)$.

We start by considering the graph whose nodes are LR states S with edges $s \xrightarrow{s} s'$ for $s' \in \mathbf{pred}(s)$. The paths leaving an LR state $s \in S$ in this graph enumerate the suffixes of viable stacks whose top state is s. To construct **Dom**, we rely on the dual use of LR states in this graph. A state s is both a node from which we enumerate suffixes and the label of all edges leaving this node.

From this, we deduce the constraints characterising **Dom**:

- The initial state is reachable and recognises the **Wait** LR states:

$$\begin{array}{rcl} q_0 & \in & \mathrm{dom}(\mathbf{Dom}) \\ \mathbf{Wait} & \subseteq & \mathbf{Dom}(q_0) \end{array}$$

- **Dom** is closed for reachable transitions of reachable states. For $q, q' \in Q, s \in S$:

$$\begin{array}{c} q \in \mathrm{dom}(\mathbf{Dom}) \wedge s \in \mathbf{Dom}(q) \wedge q' \in ||\mathbf{dest}(q,s)|| \\ \implies \\ q' \in \mathrm{dom}(\mathbf{Dom}) \wedge \mathbf{pred}(s) \subseteq \mathbf{Dom}(q') \end{array}$$

[3] An exception to this rule is the eager execution of default reductions by Yacc. Integrating LRGrep requires some adaptations, see 9.5.

Taking **Dom** to be the least solution to these equations, we construct the deterministic automaton $N = (Q', S, \mathbf{dest}', q_0, F')$ as the subset of M that we determined to be reachable by viable stacks, where:

$$\begin{array}{rcl} Q' & = & \mathrm{dom}(\mathbf{Dom}) \\ \mathbf{dest}'(q,s) & = & \mathbf{dest}(q,s) \text{ when } s \in \mathbf{Dom}(q) \\ F' & = & F \cap Q' \end{array}$$

4.2.2 Implication on automata

In this subsection, we explore the notion of implication on automata, closely connected to the recently introduced schema-based determinization Niehren et al. (2022). This method simultaneously computes both the implication and the determinization of a DFA and an NFA. For explanatory purposes, we describe the implication of two NFAs as a more fundamental operation. We hope to give some intuition on implication and how it relates to intersection by generalising the solution we have just established for LRGrep to arbitrary automata.

The intersection $A \cap B$ of two non-deterministic automata $A = (S_A, \Sigma, d_A, i_A, F_A)$ and $B = (S_B, \Sigma, d_B, i_B, F_B)$ is another automaton defined as:

$$\begin{array}{rcl} A \cap B & = & (S_A \times S_B, \Sigma, d_{A\cap B}, (i_A, i_B), F_A \times F_B) \\ d_{A\cap B}((q_a, q_b), s) & = & d_A(q_a, s) \times d_B(q_b, s). \end{array}$$

Let us write $s \xrightarrow{w} s'$ when state s' is reachable from s by following transitions labelled $w \in \Sigma^*$. In the intersection automaton, $(i_a, i_b) \xrightarrow{w} (q_a, q_b)$ if and only if $i_a \xrightarrow{w} q_a$ and $i_b \xrightarrow{w} q_b$. If q_a and q_b are final states, then w is recognised by A, B, and also by $A \cap B$. More generally, we have $w \in \mathscr{L}(A \cap B) \iff w \in \mathscr{L}(A) \cap \mathscr{L}(B)$.

Like for the subset construction, the intersection automaton benefits from trimming. Some states might be unreachable from the initial state, or some states might not be able to reach any final state, thus not contributing any word to the recognised language. We write $A \cap^T B$ for the trimmed intersection automaton, and $S_{A\cap^T B}$, $d_{A\cap^T B}$ and $F_{A\cap^T B}$ for the states, transition function and final states of the trimmed intersection automaton.

The implication is characterised by the weaker equation $w \in \mathscr{L}(A) \implies (w \in \mathscr{L}(A \implies B) \Leftrightarrow w \in \mathscr{L}(B))$. If $w \notin \mathscr{L}(A)$, the behaviour is undefined so $A \implies B$ is not unique, but we are looking for the smallest subset of B satisfying the equation.

A suitable automaton $A \implies B$ is the quotient of $A \cap^T B$ obtained by projecting on states of B:

$$\begin{array}{rcl} S_{A\implies B} & = & \{q_b \mid (q_a, q_b) \in S_{A\cap^T B}\} \\ d_{A\implies B}(q_b, a) & = & \{q'_b \mid \exists q_a, (q'_a, q'_b) \in d_{A\cap^T B}(q_a, a)\} \\ F_{A\implies B} & = & \{q_b \mid (q_a, q_b) \in F_{A\cap^T B}\} \end{array}$$

The trimmed intersection automaton tracks the configurations of A and B reachable from the initial state. By projecting on B, the implication automaton tracks the configurations of B reachable from the initial state by an input reaching at least one configuration of A.

Construction of the Implication We can directly construct the relevant subset of B via a function $D : S_B \to \mathscr{P}(S_A)$, such that for all s_a, s_b:

$$s_a \in D(s_b) \iff \exists w, i_B \xrightarrow{w} s_b \wedge i_A \xrightarrow{w} s_a$$

It is the least function such that:

- $i_A \in D(i_B)$, the empty input reaches both initial states.
- D is closed by reachable transitions of reachable states. For all s_a, s_b and $x \in \Sigma$:

$$s_a \in D(s_b) \implies \forall s'_b \in d_B(s_b, x), d_A(s_a, x) \subseteq D(s'_b)$$

The implication automaton is then given by:

$$\begin{array}{rcll} S_{A \Longrightarrow B} & = & \{\, q_b \in S_B & \mid \; D(q_b) \neq \emptyset \,\} \\ d_{A \Longrightarrow B}(q_b, x) & = & \{\, q'_b \in d_B(q_b, x) & \mid \; \exists q_a \in D(q_b), q'_a \in D(q'_b), q'_a \in d_A(q_a, x) \,\} \\ F_{A \Longrightarrow B} & = & \{\, q_b \in F_B & \mid \; D(q_b) \cap F_A \neq \emptyset \,\} \end{array}$$

Constructing D is a bit more complex than **Dom** because we don't have the double vision offered by viable stacks—a state of S_A is not a label in Σ. We track the reachable states and then visit the transitions by quantifying over the alphabet, but the definition is otherwise similar.

Implication and Minimality This construction implements $A \implies B$ but is in general not minimal. First, there might be dead states from which no accepting state is reachable. If those dead states are removed and B is deterministic, then we get the smallest subset of B compatible with $A \implies B$. In the general case, the automaton constructed for $A \implies B$ might be minimisable even if B was minimal to begin with. Minimisation of a non-deterministic automaton is decidable, even though computationally hard (Kameda and Weiner, 1970). But even after minimisation, the automaton might not be the minimal automaton recognising $\mathscr{L}(A \implies B)$ because minimisation preserves the exact behaviour of the automaton. A minimisation algorithm merges states with indistinguishable behaviours. Two states are not merged if they are reachable by some prefixes and there exist suffixes on which they behave differently. But if those prefixes and suffixes do not belong in $\mathscr{L}(A)$, implication does not require the behaviours to be preserved, which could be exploited for more aggressive merging. We do not know if minimisation under those constraints has been studied.

4.3 Two Levels of Detail

Let us go back to the determinised and pruned automaton N. Before processing it further, we want to point out that it can be read at two levels. The usual reading is at the level of states and transitions, where a transition is a directed edge between two

states labelled with an LR(1) state. We write $q \xrightarrow{s} q'$ for an edge from q to q' labelled s; since the automaton is deterministic, a transition is uniquely identified by q and s, and the target state is given by the transition function $\mathbf{dest}'$ such that $\mathbf{dest}'(q,s) = q'$.

The second level is made of continuations and maps between continuations. A state is a sequence of continuations, and a transition between two states relates each continuation in the target state to the continuation in the source state it derives from. A transition also defines a set of variables to update for each target continuation. For an edge $q \xrightarrow{s} q'$, a continuation in q maps to zero, one or more continuations of q' and, conversely, a continuation in q' comes from exactly one continuation in q. The mapping between continuations is given by $\mathbf{origin}(q,s)_i = j$, meaning that q'_i is derived from q_j. The set of variables to update for this continuation is given by $\mathbf{vars}(q,s)_i$.

4.4 Register Allocation

The representation of variables used so far is not suitable for an efficient implementation. We know which variables to update, but we can't tell which variables are already defined at any given point. A simple interpreter could manage this by maintaining and updating dynamic maps from variable names to values. Since variables are local to a continuation, each continuation should have its own map. When following a transition $q \xrightarrow{s} q'$, $\mathbf{origin}(q,s)$ tells which maps should be inherited and $\mathbf{vars}(q,s)$ tells which variables to update. But this approach leads to a lot of wasteful computations. In particular, variable updates tend to be fairly rare, so many continuations have the same bindings.

An efficient implementation can statically determine the number of unique variables and pre-allocate the storage. We solve these problems by analysing the data flow of the automaton. Data-flow analysis was originally introduced to analyse and optimise control flow graphs (Kildall, 1973), which are more general than the automaton we are working on. A data-flow analysis can establish global properties of the automaton by propagating local information along the transitions until reaching a fixed point. We run two analyses:

- One to determine which variables are live in each state. That is, the variables whose value is guaranteed to be used[4] on at least one path leaving this state.
- One to group variables guaranteed to have the same (or no) value. Variables statically known to be equal can share the same storage, or none at all if they have no value.

Definition 15 (Variables assigned by a transition) *$\boldsymbol{assigned}(q,s)$ is the set of the variables defined by the outgoing transition of state q labelled s, together with the index of the continuation for which they are defined.*

$$\boldsymbol{assigned}(q,s) = \left\{ (i,v) \;\middle|\; \forall i, 0 \le i < |q|, v \in \boldsymbol{vars}(q,s)_i \right\}$$

[4] On arbitrary control flow graphs, it is undecidable whether a variable will be used, so live variables are an over-approximation. In our restricted formalism, we can compute an exact solution.

In this section, we often use pairs (i, v) for the occurrence of variable v in the ith continuation.

4.4.1 Live variables

In the end, variables are used by accepting continuations $\blacksquare_i$. If a variable n is bound in expression e_i, it is when reaching the continuation $\blacksquare_i$ that the values associated to n_s and n_e are looked up. Therefore, our analysis starts from the accepting continuations and propagates backward, following transitions to the predecessors, until crossing transitions assigning the variables.

To define the variables used by an accepting continuation, we first need to know all variables bound in the initial expressions. The set of bound variables of an expression is defined by induction on its structure:

$$\begin{aligned}
\mathbf{bv}(\epsilon) &= \emptyset \\
\mathbf{bv}(x) &= \emptyset \\
\mathbf{bv}(n = e) &= \{n_s, n_e\} \cup \mathbf{bv}(e) \\
\mathbf{bv}(e_1 \cdot e_2) &= \mathbf{bv}(e_1) \cup \mathbf{bv}(e_2) \\
\mathbf{bv}(e_1 | e_2) &= \mathbf{bv}(e_1) \cup \mathbf{bv}(e_2) \\
\mathbf{bv}(e^*) &= \mathbf{bv}(e) \\
\mathbf{bv}(/\, A \to \alpha \cdot \beta) &= \emptyset \\
\mathbf{bv}([\, e\,]) &= \emptyset
\end{aligned}$$

bv collects all occurrences of variable binding constructions $n = e$. Note that for the reduction operator $[e]$, the sub-expression is ignored: bindings are not allowed under a reduction.

Definition 16 (Live variables) *The live variables of state q, written $\boldsymbol{live}(q)$, are the least solution to equation:*

$$\boldsymbol{live}(q) = \mathit{immediate}(q) \cup \mathit{successors}(q)$$

where:

$$\begin{aligned}
\mathrm{immediate}(q) &= \left\{ (i, v) \;\middle|\; q_i = \blacksquare_j, v \in \mathbf{bv}(e_j) \right\} \\
\mathrm{successors}(q) &= \bigcup_{q \xrightarrow{s} q'} \mathrm{rename}(q, s, \mathbf{live}(q')) / \mathbf{assigned}(q, s) \\
\mathrm{rename}(q, s, V) &= \left\{ (\mathbf{origin}(q, s)_i, v) \;\middle|\; (i, v) \in V \right\}
\end{aligned}$$

A variable is live if it is used by a continuation $\blacksquare_i$ or if it is live in a successor and not assigned by the transition leading to this successor. An accepting continuation $\blacksquare_i$ uses all variables in $\mathbf{bv}(e_i)$. If a variable live in a successor is assigned on the transition to the successor, then the successor uses this new definition and the use is not propagated further. Since a variable is tagged by the index i of the continuation in which it is live, it is *renamed* during propagation from i to $\mathbf{origin}(q, s)_i$.

Finally, we say that a variable is defined in state q if it is live and assigned on at least one path reaching the state. We write $\mathbf{defined}(q)$ for the set of defined variables.

Definition 17 (Defined variables of a state) ***defined***(q) *is the least solution to equation:*

$$\textbf{\textit{defined}}(q) = \bigcup_{q' \xrightarrow{s} q} \left\{(i,v) \in \textbf{\textit{live}}(q) \middle| (i,v) \in \textbf{\textit{assigned}}(q',s) \vee (\textbf{\textit{origin}}(q',s)_i, v) \in \textbf{\textit{defined}}(q')\right\}$$

Operationally, the variables in **defined**(q) are those that need storage space when reaching state q: they will be used in the future and might have already been assigned some value.

4.4.2 Variable classes

To save storage space, we want to group the defined variables that are known to be equal. We do this using, for each state q, an equivalence relation $\equiv_q$ on variables in **defined**(q) such that two variables (i,v) and (j,w) are in the same class if, on every paths reaching q, they are either both unassigned or both assigned by the same transition.

Definition 18 (Variable equivalence) *Given a state q and $(i,v),(j,w) \in$* ***defined***(q)*, $(i,v) \equiv_q (j,w)$ iff for each transition $q' \xrightarrow{s} q$:*

- *either* $\left\{(i,v),(j,w)\right\} \subseteq$ ***assigned***(q',s)
- *or* $\left\{(i,v),(j,w)\right\} \cap$ ***assigned***$(q',s) = \emptyset \wedge (\textbf{\textit{origin}}(q',s)_i, v) \equiv_{q'} (\textbf{\textit{origin}}(q',s)_j, w)$

Two defined variables are equivalent in a state q if and only if they are equivalent on every incoming transition $q' \xrightarrow{s} q$. Two variables are equivalent on an incoming transition if they are both assigned on this transition, or, if they are not assigned, if they are equivalent in the source state of the transition.

Definition 19 (Variable classes) *We write* ***classes***$(q) =$ ***defined***$(q)/\equiv_q$ *for the set of defined variables quotiented by variable equivalence.*

We see the classes as a set of sets. For instance if **classes**$(q) = \{\{(1,v),(2,v)\},\{(3,v)\}\}$, the state has three defined variables named v and belonging to different continuations. The first two are always equal—they alias each other. The third v can differ.

This is both a data flow and a partition refinement problem. The solution can be computed by specialised algorithms such as those of Paige and Tarjan (1987) but we found that a naive iterative approach is sufficient. Intuitively, we have one partition per state and there are many states, but the domain of each partition, **defined**(q), is small and the iteration converges quickly.

4.4.3 Register allocation & transfers

The number of classes in **classes**(q) tells us how much storage is needed at run time when the state is reached. A state with two classes, say $\{\{(1,v),(2,v)\},\{(3,v)\}\}$, needs two registers. The number of registers needed to run the automaton is the number of classes of the largest partition: $\max_{q\in Q}(|\textbf{classes}(q)|)$.

Register allocation attributes each class of a state to a register via *register mapping*. When a transition is followed, certain registers can be set to the current position and other registers might need to be remapped to match the mapping of the target state. We represent this by a *transfer function*.

Register allocation To allocate registers, we choose an arbitrary total ordering over sets of variables and number each class of a state according to its position in the ordering. For instance in the partition above, if according to the ordering $\{(1,v),(2,v)\} < \{(3,v)\}$, then class $\{(1,v),(2,v)\}$ is attributed register 0 and class $\{(3,v)\}$ is attributed register 1.

Definition 20 (Register mapping) *Given a state q, the **register** function is the bijection between **classes**(q) and the interval $[0;|\textbf{classes}(q)|[$ that preserves the ordering. For the inverse, we write **register**$^{-1}(\textbf{classes}(q),i) \in \textbf{classes}(q)$ for $i \in [0;|\textbf{classes}(q)|[$.*

This very naive ordering completely ignores the data flow of the automaton and is likely to generate a lot of trivial register transfers. This would be a problem for a traditional compiler, but it offers more opportunities for automaton minimisation. Minimisation makes the automaton more compact by merging states that have the same observable behaviour (states that recognise the same inputs and generate the same register transfers). If the register allocation accounted for the flow coming from predecessors, it is more likely that two otherwise equivalent states would be given different register mappings. Compacting the automaton matters more than decreasing the number of register transfer and we found this naive solution to be sufficient in practice.

Variables transfer Knowing the variable classes of each state and where they are stored, we can determine how the storage space should be updated when following a transition.

For each transition $q' \xrightarrow{s} q$, we represent this by a function:

$$\textbf{transfer}(q' \xrightarrow{s} q) : [0;|\textbf{classes}(q)|[\to [0;|\textbf{classes}(q')|[\cup \{\top,\bot\}$$

transfer$(q' \xrightarrow{s} q)(i)$ determines how the register i should be updated when the transition is followed. A register of the target state maps to three possibilities:

- a register of the source state; this case corresponds to a copy.
- the symbol $\top$; this case corresponds to setting the register to the current input position.

- the symbol $\bot$; this case gives the special value $\bot$ to the register, representing an unassigned variable.

Definition 21 (transfer function) *The **transfer** function is defined as follows:*

$$\textbf{\textit{transfer}}(q' \xrightarrow{s} q)(i) = \begin{cases} \top & \text{if } (i,v) \in \textbf{\textit{assigned}}(q',s) \\ \textbf{\textit{register}}(c') & \text{when } c' \in \textbf{\textit{classes}}(q') \text{ and } (\textbf{\textit{origin}}(q',s)_i, v) \in c' \\ \bot & \text{otherwise} \end{cases}$$
$$\text{where } (i,v) \in \textbf{\textit{register}}^{-1}(i)$$

The transfer action for register i depends on $\textbf{register}^{-1}(i)$, the class stored in the register. The equivalence relation ensures that all variables of the class are treated the same, so we pick an arbitrary variable (i,v) of the class to discuss the possibilities. If the variable is assigned by the transition, the register should be set to the current position, represented by $\top$. If the variable is imported from a predecessor, the contents of its register should be copied. Otherwise, the variable is unassigned and marked $\bot$.

The $\bot$ case handles transitions reaching a joint-point where a variable is defined in some but not all branches. Consider a trivial automaton with initial state q_0 and two transitions $q_0 \xrightarrow{s_1,\{v\}} \blacksquare$ and $q_0 \xrightarrow{s_2,\emptyset} \blacksquare$. $\blacksquare$ is reachable by reading s_1 or s_2, but the variable v is set only when reading s_1. $\textbf{defined}(q_0)$ is empty since no variable has been set yet, and $\textbf{defined}(\blacksquare)$ is $\{v\}$. The transfer function for $q_0 \xrightarrow{s_1} \blacksquare$ is $v \mapsto \top$ since the variable is assigned by the transition but for $q_0 \xrightarrow{s_2} \blacksquare$ it is $v \mapsto \bot$.

Together, **register** and **transfer** enable a representation that is suitable for an efficient implementation of recognition:

- registers are small numbers that can be interpreted as indices in an array
- transfers and assignments are trivial operations (permutations and updates of array cells)

We define an extra helper to find the register in which a variable is stored.

Definition 22 ***find-register***(q,i,v) *finds the register in which the variable v of the continuation q_i of state q is stored, as an integer in $[0; \textbf{\textit{classes}}(q)[$, or $\bot$ if it is not defined in this state:*

$$\textbf{\textit{find-register}}(q,i,v) = \begin{cases} \textbf{\textit{register}}(c) & \text{when } (i,v) \in c \text{ and } c \in \textbf{\textit{classes}}(q) \\ \bot & \text{otherwise} \end{cases}$$

As with **transfer**, $\bot$ is used to signal that a value is missing for v.

4.5 Minimisation

For each deterministic automaton, there exists a unique and minimal deterministic automaton that recognises the same language (Hopcroft et al., 2000). This is done (1)

by removing states that are not reachable from the initial state, (2) by removing dead states, from which no accepting state is reachable, and (3) by merging indistinguishable states (states that recognise the same language; in our case, they must also lead to the same register transfers and accepting continuations).

The procedure we used to generate the automaton produces only reachable states, but it often leads to dead and indistinguishable states, so it is worth post-processing the automaton with a minimisation algorithm.

Note that while minimisation is well-defined given a deterministic automaton and a fixed alphabet, in our case some arbitrary choices already affected the definition of the automaton and the alphabet. In particular, three factors of our generation procedure affect the effectiveness of the minimisation procedure on the automaton: the use of wildcards in the derivation of reductions (Section 2.4.1), the exploration conditioned by viable stacks (Section 4.2), and the register allocation and transfer functions (Section 4.4.3). Assuming these are fixed, the minimisation procedure yields a minimal automaton, but it might be possible to produce a smaller automaton by making different choices.

With these precautions in mind, we can apply a standard, off-the-shelf, minimisation procedure. We experimented successfully with Valmari's variant of Hopcroft's algorithm (Hopcroft, 1971; Valmari, 2012). These algorithms work by maintaining a partition which keeps indistinguishable states in the same class. The initial partition is a coarse over-approximation (with distinguishable states in the same class), that is progressively refined until all distinguishable states are separated. Jacobs and Wissmann (2022) take a modern look at the problem and give a high-level description that leads to a fast and generic algorithm.

Minimising the extended DFA The characteristics of our automaton necessitate modifications to the standard approaches to achieve correct and optimal results:

- The initial partition is refined to distinguish the different accepting states, based on their accepting continuation and the mapping of registers used by the accepting continuation.
- The alphabet is extended to also include the transfer functions, since two transitions $q_1 \xrightarrow{s} q_2$ and $q_3 \xrightarrow{s} q_4$ can be distinguished if the functions $\mathbf{transfer}(q_1 \xrightarrow{s} q_2)$ and $\mathbf{transfer}(q_3 \xrightarrow{s} q_4)$ differ, even though they are both labelled s

That is, for a transition $q \xrightarrow{s} q'$, the label is taken to be $(s, \mathbf{transfer}(q \xrightarrow{s} q'))$ and not simply s. The initial partition is the coarsest partition such that:

- states without an accepting continuation ($\nexists(i,j), q_i = \blacksquare_j$) are in the same class
- two states q and q' such that $\exists i, i', k,\ q_i = q'_{i'} = \blacksquare_k$ are in the same class iff $\forall v \in \mathbf{bv}(e_j), \mathbf{find\text{-}register}(q,i,v) = \mathbf{find\text{-}register}(q',i',v)$.

Output of minimisation Minimisation is an optimisation that is transparent to the rest of the pipeline: it is not mandatory to minimise the automaton. But if one wants to use it, a few definitions have to be adapted.

Let us write R and d_R for the states and transition function of the minimised automaton. We have that:

- a state $r \in R$ is a set of equivalent states $q \in Q$ from the source automaton.
- transitions are labelled by pairs (s, f) of an LR state and a transfer function.

A state r is accepting clause j if there exists $q \in r$ such that $q_i = \blacksquare_j$. When this is the case, we simply write $r_i = \blacksquare_j$ and define **find-register**$(r, i, v) =$ **find-register**(q, i, v). The initial refinement used for minimisation guarantees that the partition of states is compatible with these definitions.

We decompose the transitions into an input label and a separate transfer function. When $d_R(r, (s, f)) = r'$, we write $r \xrightarrow{s} r'$ and **transfer**$(r \xrightarrow{s} r') = f$. The specification of the minimisation problem guarantees that f is functionally determined by s (there is no $f' \neq f$ on which $d_R(r, (s, f'))$ is defined).

Finally, we need the **Dom** function. **Dom**(q) characterises the subset of S from which transitions leaving q are reachable. A transition leaving r is reachable if it is reachable from at least one state in the equivalence class:

$$\mathbf{Dom}(r) = \bigcup_{q \in r} \mathbf{Dom}(q).$$

4.6 Code Generation

In this section, we convert an automaton into a program for a basic abstract machine, a format easy to interpret with or compile to general-purpose programming languages.

Input Automaton We start from an automaton with a set of states Q recognising a vector of expressions e_i. We write $\blacksquare_i = q_j$ if the jth continuation of state q accepts clause i and $q \xrightarrow{s} q'$ for a transition from q to q' labelled by an LR state $s \in S$. We also assume a few functions refining the behaviour of transitions:

- the register transfers for each transition are specified by a finite map **transfer**$(q \xrightarrow{s} q') : [0; m[\rightharpoonup \mathbb{N}[0; n[\cup \{\top, \bot\}$ for some $m, n \in \mathbb{N}$.
- **Dom**$(q) \subseteq S$ specifies the reachable subset of outgoing transitions from q. This is used to distinguish rejecting and undefined transitions. When in state q and looking ahead at s, if there is no transition $q \xrightarrow{s} q'$, the default behaviour is to reject the input. If $s \notin$ **Dom**(q), code generation knows that this situation cannot happen with a valid input (an LR stack), and is free to choose a different action.

- **find-register**$(q, i, v) \in \mathbb{N} \cup \{\bot\}$ finds the register in which variable v of the ith continuation of state q is stored, or $\bot$ if the variable is not defined in q.

This automaton can be the unoptimised M we got after determinisation, its subset N produced by exploring the viable stacks, or the result of the minimisation procedure. To use M, **Dom** is taken to be $\_ \mapsto S$.

Actions A state q often has many transitions with the same transfer function and target state. That is, if $q \xrightarrow{s} q'$, there are generally other s' such that $q \xrightarrow{s'} q'$ and **transfer**$(q \xrightarrow{s} q')$ = **transfer**$(q \xrightarrow{s'} q')$; we say that s and s' have the same *action*.

We call these pairs of a transfer function and target state *actions*, and we write $a \in$ **Actions** $= \left\{ (\textbf{transfer}(q \xrightarrow{s} q'), q') \;\middle|\; \forall q, q', s, q \xrightarrow{s} q' \right\}$.

Output Program The program consumes the input states of the LR stack one by one and from right to left. When clause i is found to match with variables bound to values $\vec{v}$, it emits the pair $(i, \vec{v})$. A single execution can produce zero, one or more matches.

The program is made of two components:

- a map **code** : $Q \cup$ **Actions** $\to$ *Instruction** associating each state and action to a sequence of instructions.
- a partial function **table** : $Q \times S \rightharpoonup$ **Actions** $\cup \{ \mathit{Halt} \}$ representing the transition table.

The program starts by executing **code**(q_0), the sub-program of the initial state.

4.6.1 Abstract Machine

The abstract machine contains a fixed number of registers, identified by an integer from 0 to n, and tracks the position of the current input symbol. A symbol is a state of the LR automaton, represented as a number. We write *r1*, *r2*, ... for meta variables ranging over registers. The value of a register is either a natural number, denoting a position in the input, or a distinguished value $\bot$ representing an unassigned variable.

Instructions

There are register and control instructions. Register instructions change the value of a register, and execution moves to the next instruction in sequence. Control instructions can change the next instruction to execute.
The register manipulation instructions are:

- `Store` *r1*: save the current position in register *r1*
- `Move` *r1*, *r2*: set `r1` to the value of *r2*

- `Clear` *r1*: set `r1` to $\bot$
- `Swap` *r1*, *r2*: exchange the values of *r1* and *r2*

The control instructions are:

- `Goto` q: move input to next symbol and move execution to the sub-program associated with state q
- `Match` q: query the transition table at cell $(q, \textit{current input symbol})$. If it is an action a, execution continues with the sub-program associated with a. If it is *Halt*, execution stops; if it is not defined, execution continues with the next instruction in sequence.
- `Accept` c, $r1|\bot, \ldots, rn|\bot$: signals that clause number c succeeded with variables bound to the values of registers *r1* to *rn*, or $\bot$ for variables that are statically known to be unbound. c has to have arity n (i.e. $|\mathbf{bv}(e_c)| = n$). Execution continues with the next instruction. Multiple `Accept` instructions can be executed if an input has multiple matches.
- `Halt`: stop execution.

4.6.2 Default Actions and Transition Table

To keep the code of an automaton compact, we want the transition table to be sparse; defined on the smallest domain possible. In practice, the outgoing transitions from a state often share the same actions, such that they can be grouped by identical behaviour. A standard solution to make the table sparse is to treat the most common action separately and only store the transitions that differ from it (Aho et al., 1986).

Definition 23 (Default transitions) *We write* $\boldsymbol{default}(q)$ *for a largest subset of* $\mathbf{Dom}(q)$ *such that, for all* $s_1, s_2 \in \boldsymbol{default}(q)$*, either:*

- *both* $q \xrightarrow{s_1} q_1$ *and* $q \xrightarrow{s_2} q_2$ *are defined and they have the same action:*

$$q_1 = q_2 \wedge \boldsymbol{transfer}(q \xrightarrow{s_1} q_1) = \boldsymbol{transfer}(q \xrightarrow{s_2} q_2)$$

- *both are undefined*

If multiple subsets of $\mathbf{Dom}(q)$ satisfy these conditions, an arbitrary one is chosen.

Definition 24 (Sparse transition table) *The transition table is represented by a partial function* $\boldsymbol{table} : Q \times S \rightharpoonup \boldsymbol{Actions} \cup \{\, \textit{Halt} \,\}$.

It is defined on the pairs (q, s) *where* $s \in \mathbf{Dom}(q)$ *and* $s \notin \boldsymbol{default}(q)$ *by:*

$$\boldsymbol{table}(q, s) = \begin{cases} \boldsymbol{transfer}(q \xrightarrow{s} q'), q') & \textit{when } q' = d'(q, s) \textit{ is defined} \\ \textit{Halt} & \textit{otherwise} \end{cases}$$

4.6.3 Translation of transfer functions

A transfer function $f : [0; dst[\to [0; src[\cup \{\top, \bot\}$ shuffles, copies and updates certain registers. After execution of the function, we want the first *dst* registers to evaluate either to the value a register had before execution (if the image is an integer), to store the current position (if the image is $\top$) or to be cleared (if the image is $\bot$). To translate it into a sequence of primitive register instructions, we have to carefully order the operations to avoid overwriting the value of a register that is later needed.

This is an instance of the well-studied parallel move problem. We give a simple, greedy solution as pseudo-code in appendix A and refer the reader to Rideau et al. (2008) for an optimal solution (using an auxiliary register instead of `Swap` instructions).

4.6.4 Translation of states and actions

We can now define the sub-programs for each state and action. For an action (f, q), we emit the instructions for the transfer function f followed by `Goto` q.

For a state q, we start by handling accepting continuations. For each i, j such that $q_i = \blacksquare_j$, we emit `Accept` `j`$, \vec{r}$. The vector $\vec{r}$ gives values to the variables of clause j. Let $\vec{v}$ be a vector of the variables $\mathbf{bv}(e_j)$ with an arbitrary ordering (i.e. for each $v \in \mathbf{bv}(e_j), \exists! k, \vec{v}_k = v$). Then $\vec{r}_k = \textbf{find-register}(q, i, \vec{v}_k)$.

After that, we emit `Match q` to look up the transitions for state q, followed by the code for the action of the default group $\mathbf{default}(q)$. We pick an arbitrary $s \in \mathbf{default}(q)$ and:

- if $q \xrightarrow{s} q'$ is defined, we emit the code for the transfer function $\textbf{transfer}(q \xrightarrow{s} q')$ followed by `Goto` q'.
- if $q \xrightarrow{s} q'$ is not defined, we emit `Halt`.

4.6.5 Representing a sparse transition table

table is a partial function with a limited domain. Efficient representation of these functions has been the subject of various studies. The Dragon Book (Aho et al., 1986) cites a simple solution in $O(n^2)$, where n is the size of the domain, for lexer generators that works well in practice.

LRGrep uses this solution. It has been sufficient so far; however, we have observed it becoming a bottleneck when scaling to larger specifications. In our tests, a rule with around one thousand clauses translated into around 10^6 transitions and packing the transition table was the slowest part of the pipeline, taking a few seconds.

We are looking to replace this algorithm in the future. The problem is studied by Tarjan and Yao (1979) who offer a more involved solution with strong guarantees. Driesen (1999) compares common solutions to the problem of efficient dynamic dispatch in object-oriented programming languages; this turns out to be the same problem, replacing transitions indexed by a state and symbol with method implementations indexed by a class and a method name.

4.7 Conclusion

In this chapter, we started from a derivative operator d_C for our dialect of regular expressions and constructed an efficient deterministic automaton. In Section 4.1, we adapted the subset construction to obtain a determinisation procedure that is compatible with the rules for disambiguation and variable captures, resulting in a deterministic transition function **dest**.

We observed that the automaton obtained by following all derivatives from an initial state is unnecessarily large since it classifies all inputs in S^*, while we are only interested in classifying the stacks reachable by an LR parser. Section 4.2 introduced the notion of automaton implication to characterise the relevant subset of the automaton and restricted the transition function to d', which still correctly classifies the LR stacks and yet leads to a smaller automaton.

To implement variable captures efficiently, in Section 4.4.3 we applied data flow analyses on the smaller automaton to store variable values in a minimum number of registers and determined the registers to transfer or update for each transition.

This results in a deterministic automaton whose transitions are given by **dest**$'$ and labelled by register **transfer** functions. This automaton can be further minimised by adapting standard techniques (Section 4.5).

Finally, Section 4.6 suggests a simple abstract machine and a translation scheme to represent the automaton as a program.

Chapter 5

Fast reachability analysis for LR(1) Parsers

This chapter is self-contained and has been published separately in Bour and Pottier (2021). The results presented here are re-used in Chapter 6.

5.1 Introduction

The Reachability Problem An LR(1) automaton is a state machine equipped with a stack, which stores a sequence of past states. The state found at the top of the stack is the current state. At each step, the automaton's action is determined by its current state and the lookahead symbol, that is, the first unconsumed input symbol. The automaton's palette of actions includes consuming the first input symbol, following a transition to a new state (which is then pushed onto the stack), and popping a number of states off the stack (thus changing the current state).

A pair of a state s and a symbol z is *reachable* if there exists an input sentence that leads the automaton from its initial configuration to a configuration where the current state is s and the current lookahead symbol is z. In its simplest form, the *reachability problem* can be formulated as follows:

Given: a state s; a terminal symbol z.
Find: whether (s, z) is reachable.

The problem can also be stated in a more elaborate form as follows:

Given: a state s; a transition tr that leaves the state s; two terminal symbols a and z.
Find: whether there exists a sentence w such that:

- the first symbol of wz is a, and
- whenever the automaton is in state s and its input begins with wz, the automaton consumes w, leaves z unconsumed, and takes the transition tr.

In short, the second problem asks under what conditions each transition can be taken. A solution of this problem yields a solution of the first problem. Indeed, a

solution of the second problem allows constructing a directed graph G, whose vertices are pairs (s, z), such that the first problem boils down to ordinary reachability in the graph G. The role played by the terminal symbol a may not be clear to the reader at this point; we return to it in §5.3.

By an "LR(1) automaton", we mean a state machine equipped with a stack and that can peek at the first unconsumed input symbol. There are many methods for constructing such an automaton, including Knuth's canonical method (Knuth, 1965), SLR (DeRemer, 1971), LALR (DeRemer and Pennello, 1982; DeRemer, 1969), Pager's method (Pager, 1977), and IELR(1) (Denny and Malloy, 2010). Furthermore, this construction process can be influenced by user-provided precedence declarations, which indicate how shift/reduce and reduce/reduce conflicts must be resolved. Because we state the reachability problem in terms of the automaton, not in terms of a grammar, we do not care how the automaton is constructed. Our reachability algorithm is compatible with all of the construction methods cited above.

Applications Solving the reachability problem can be useful in a range of situations. One simple application is error diagnosis, that is, producing a diagnostic message when a syntax error is encountered. Jeffery (2003) suggests producing a message based solely on the current state of the automaton while disregarding its stack. This is done by setting up in advance a mapping of states to messages. Because state numbers are opaque and brittle, the user is in fact expected to set up a mapping of erroneous input sentences to messages. Pottier (2016) notes that this table must be *correct* (every sentence is erroneous), *irredundant* (no two sentences lead to the same state), and *complete* (every state where an error can be triggered corresponds to some sentence). While correctness and irredundancy are easy to enforce, completeness is a more challenging property. Constructing a complete collection of erroneous input sentences or just checking that a collection of erroneous input sentences is complete requires the ability to compute the *reachable error states*, that is, to determine which states s are such that some input sentence triggers an error in state s. This requires solving the reachability problem.

Pottier (2016) proposes a reachability algorithm and implements it in the Menhir parser generator (Pottier and Régis-Gianas, 2021). Menhir's support for Jeffery's error diagnosis methodology is exploited in production by several compilers, such as CompCert C (Leroy, 2021), Catala (Merigoux et al., 2021), and Stan 3 (Talts, 2019).

A reachability algorithm has applications beyond error diagnosis. It could be used, for instance, to test an LR(1) parser by generating a set of input sentences that reaches every state or one that triggers every reduction in every state. It could also be used as a component in syntax error recovery algorithms and in syntactic completion algorithms.

Challenges and Contributions The reachability problem in pushdown systems is a well-known challenge. While a general $O(n^3)$ solution is well-established, the fine-grained complexity remains under investigation—many special cases admit faster algorithms, though their classification remains incomplete (Chistikov et al., 2022;

Koutris and Deep, 2023). This work provides an efficient solution for the case of LR automata with conflict resolution but does not investigate the relation with other classes of context-free languages (CFLs).

The question that appears in the second problem statement above is parameterised with a transition tr and with two terminal symbols a and z. Thus, the number of questions that may be asked is $|Tr| \times |T|^2$, where Tr is the set of all transitions and T is the set of all terminal symbols. An analysis of the problem reveals that these questions seem interdependent, so answering one of them potentially requires answering many of them. Without even analysing the cost of answering one question, this suggests that the complexity of a reachability algorithm can be high. We analyse the complexity of a naïve reachability algorithm later on (§5.3.3).

Pottier (2016) proposes a reachability algorithm whose implementation in Menhir (Pottier and Régis-Gianas, 2021) has been in use for five years. We find that, in practice, this algorithm does not scale as gracefully as one might wish. For the OCaml grammar and parser, which involve 121 terminal symbols, 230 nonterminal symbols, and 1672 LR(1) states, we find that this algorithm can require over 5 gigabytes of memory and 5 minutes of run time. Although the space requirement is tolerable, such a time requirement is much too large to achieve a smooth edit-compile-debug cycle.

Contribution We present a new reachability algorithm which we find is much more efficient in practice than Pottier's algorithm. Our algorithm processes the OCaml parser in 50 megabytes of memory and half a second, improving over Pottier by a factor of 10^2 in space and 6×10^2 in time (§5.7).

We derive this new algorithm in two main steps. First, we propose a matrix-based formulation of the reachability problem, which requires computing a *cost matrix* for every transition. Then, we remark that cost matrices are redundant in two ways: (a) they contain many identical columns; (b) they contain many rows whose content is not relevant, insofar as only the *combination* of these rows via the function "pointwise minimum" is of interest. This suggests that it is possible to gain space (and save time) by merging certain rows and columns. We propose a method for determining ahead of time (before the cost matrices are computed) which rows and columns can be merged, as well as a direct method for constructing *compact cost matrices* where these rows and columns are merged.

These ideas, combined with a number of other optimisations, allow us to report excellent performance in practice. The largest grammar in our test suite, a grammar for the C++ programming language (Hashimoto, 2021), is processed in 2.6 gigabytes of memory and under one minute, whereas it could not be processed by Pottier's algorithm.[1]

[1] Pottier's algorithm, as implemented by him in Menhir, is limited to 256 terminal symbols. This is caused by the choice of a low-level representation that involves packing several pieces of information in a single machine word. Even if this limitation was removed, our performance evaluation (§5.7) shows that Pottier's algorithm would not be able to handle this grammar.

Outline After introducing some notation (§5.2), we present our formulation of the reachability problem in terms of cost matrices (§5.3). Then (§5.4), we explain several ways in which this formulation can be improved, including merging certain rows and columns. This leads to a reformulation of the problem in terms of compact cost matrices (§5.5). The paper ends with a discussion of implementation details (§5.6), an experimental evaluation of the algorithm's performance (§5.7), a review of the related work (§5.8), and concluding words (§5.9).

5.2 Notation

We assume that a context-free grammar $\mathscr{G}$ has been fixed and that an LR(1) automaton has been constructed for this grammar. We assume the reader is familiar with these concepts (Aho and Ullman, 1972; Grune and Jacobs, 2008). In the following, we recall some standard notation and propose specific notation useful in this paper.

LR(1) Automata We write:

- $a, b, c, z \in T$ for terminal symbols,
- $A \in N$ for nonterminal symbols,
- $x \in X$, where X is $T \cup N$, for arbitrary symbols,
- $Z \in \mathscr{P}(T)$ for sets of terminal symbols,
- $w \in \vec{T}$ for sentences (sequences of terminal symbols),
- $\alpha, \beta \in \vec{X}$ for sentential forms (sequences of symbols),
- $s \in S$ for automaton states.

A transition in the automaton is a directed edge from a source state s to a target state s'. Every transition is labelled with a symbol x. The "shift" transitions are the transitions whose label is a terminal symbol; the "goto" transitions are those whose label is a nonterminal symbol. We write Tr for the set of all transitions.

Because the automaton is deterministic, the target state s' of a transition is determined by s and x. Thus, a transition can be identified by the pair (s, x), and Tr can be viewed as a subset of $S \times X$.

The target state of a transition is given by the function $target : Tr \to S$, which can also be viewed as a partial function $target : S \times X \rightharpoonup S$. We generalise this function so it accepts a state s and a sequence of symbols α as arguments. Thus, $target(s, \alpha)$ is the state reached by following the path labelled α out of the state s, if such a path exists. We write $\mathbf{incoming}(s')$ for the set of transitions that enter the state s', that is, for the set $\{tr \in Tr \mid target(tr) = s'\}$.

We assume that the reduction actions of the automaton are given by a function *reduce*, which maps a pair of a state s and a production $A \to \alpha$ to a set of terminal

symbols. When the automaton is in state s, if the next input symbol (the "lookahead symbol") is a member of $\mathit{reduce}(s, A \to \alpha)$, then the automaton will reduce the production $A \to \alpha$.

Costs A cost is either a non-negative natural number or ∞. We write $\bar{\mathbb{N}}$ for $\mathbb{N} \cup \{\infty\}$, and equip it with semiring structure. We use the min-plus semiring $(\bar{\mathbb{N}}, \oplus, \otimes)$, where:

- $x \oplus y = \min\{x, y\}$,
- the unit of $\oplus$ is $+\infty$,
- $x \otimes y = x + y$,
- the unit of $\otimes$ is 0.

Matrices We use "cost matrices", that is, matrices whose elements are costs. We write "$\cdot$" for the matrix product over the cost semiring. The rows and columns of cost matrices are indexed with terminal symbols (§5.3). Later on, we introduce "compact cost matrices" whose rows and columns are indexed with sets of terminal symbols (§5.5).

Partitions A *partition* of the terminal symbols is a set P of non-empty subsets of T such that every terminal symbol is a member of exactly one element of P. We write $\mathrm{Part}(T)$ for the set of all partitions of T. We let P, Q range over partitions, and p, q range over *classes*, that is, elements of a partition.

A partition P *refines* a partition Q if every element of P is a subset of some element of Q. We write $P \preceq Q$ when this is the case. Equipped with this partial order, $\mathrm{Part}(T)$ forms a lattice. Its bottom element $\bot$ is the finest partition, $\{\{a\} \mid a \in T\}$. Its top element $\top$ is the coarsest partition, $\{T\}$. The meet of two partitions P and Q, written $P \wedge Q$, is the coarsest partition that refines both P and Q. The join of two partitions P and Q, written $P \vee Q$, is the finest partition that is refined both by P and by Q.

If Z is a set of terminal symbols, we write $Z?$ for the partition $\{Z, T \setminus Z\} \setminus \{\emptyset\}$. This partition has at most two classes, namely Z and $T \setminus Z$. If one of these sets is empty, then it has only one class. This partition distinguishes members versus non-members of Z.

We let $Z!$ stand for $(\{T \setminus Z\} \cup \bigcup\{\{a\} \mid a \in Z\}) \setminus \{\emptyset\}$. The classes of this partition are $T \setminus Z$ (if it is non-empty) and the singletons $\{a\}$, where a ranges over Z. This partition distinguishes all members of Z, and places all non-members of Z in a single class. We have $Z! \preceq Z?$.

Finally, if P is a partition, we define the partition $P \downarrow Z$, pronounced "P inside Z", as $(P \wedge Z?) \vee Z!$. Taking the meet of P and $Z?$ refines P by distinguishing members versus non-members of Z. Then, taking a join with $Z!$ coarsens the result by conflating all non-members of Z into a single class. We have $Z! \preceq P \downarrow Z \preceq Z?$.

5.3 Problem Specification

5.3.1 Transition Costs

Let us briefly explain again what problem we wish to solve. In short, for each transition in the automaton, we would like to determine under what conditions this transition can be taken. So, we might wish to ask:

Question 1 *Let tr be a transition. What input sentences w allow tr to be taken and are consumed when it is taken?*

In the case of a shift transition (s, a), naturally, the answer is immediate: this transition can be taken when the first input symbol is a, and this symbol is consumed in the process.

In the case of a goto transition (s, A), answering is more difficult. To take such a transition, the automaton must consume a sequence w of input symbols so as to reach a state where one of the productions associated with the symbol A, say $A \to \alpha$, can be reduced. There, the automaton must reduce α to A, which is permitted only if the input symbol that follows w, say z, allows such a reduction. The symbol z is known as the *lookahead symbol*: it is consulted before the reduction, but not consumed.

Because the answer to the question depends on z, the first input symbol that follows w, and because we seek a pleasant decomposition of the main problem into subproblems, the need naturally arises to refine our question by imposing a constraint on the *first symbol a* of the sentence wz.

Question 2 *Let tr, a, and z be a transition and two terminal symbols. What sentences w allow tr to be taken, under the assumption that the first input symbol following w is z, and under the constraint that the first symbol of wz must be a?*

For greater simplicity, instead of answering Question 2 with a set of sentences, we focus on computing the *minimum length* of a sentence in this set, a number in $\bar{\mathbb{N}}$. Once the cost of every transition is known, it is a routine exercise to reconstruct a minimum-cost sentence that allows each transition to be taken.

The answer to Question 2 is the *cost* of transition *tr* with respect to the initial symbol a and the lookahead symbol z. We write $cost(tr)_{a,z}$ for this cost; it is a number in $\bar{\mathbb{N}}$.

It is useful to think of transition costs as matrices. For each transition *tr*, $cost(tr)$ can be viewed as a *cost matrix* whose rows are indexed by a and whose columns are indexed by z. This matrix can be viewed as the answer to Question 1. Whereas Question 2 is a family of questions, indexed by a and z, each of which is answered by a single cost, Question 1 is a single question that is answered by a cost matrix.

5.3.2 A Characterisation of Transition Costs

We now characterise the cost matrices $cost(tr)$ by providing a family of equations that these matrices satisfy.

The cost of a shift transition (s, a) is easy to express. It is 1 if the constraint imposed on the initial input symbol allows taking this transition; it is ∞ otherwise. Thus:

$$\begin{aligned} cost(s,a)_{b,z} &= 1 \quad \text{if } a = b \\ cost(s,a)_{b,z} &= \infty \quad \text{otherwise} \end{aligned}$$

In other words, the matrix $cost(s, a)$ contains a row of 1's at index a and contains ∞ everywhere else:

$$cost(s,a) = \begin{matrix} \vdots \\ a \\ \vdots \end{matrix} \begin{bmatrix} \infty & \dots & \infty \\ 1 & \dots & 1 \\ \infty & \dots & \infty \end{bmatrix}$$

The cost of a goto transition $cost(s, A)$ is characterised by the following equations:

$$cost(s,A) = \bigoplus_{A \to \alpha} \begin{cases} let\ s' = target(s,\alpha)\ in \\ cost(s, A \to \epsilon \bullet \alpha) \cdot \\ \Delta\, reduce(s', A \to \alpha) \end{cases} \tag{5.1}$$

$$cost(s, A \to \alpha \bullet x\beta) = \begin{cases} cost(s,x) \cdot \\ cost(target(s,x), A \to \alpha x \bullet \beta) \end{cases} \tag{5.2}$$

$$cost(s, A \to \alpha \bullet \epsilon) = I_T \tag{5.3}$$

Figure 5.1: A Characterisation of Transition Costs

Equation 5.1 Equation 5.1 expresses the fact that the cost of the transition (s, A) is the minimum, over all productions $A \to \alpha$, of the costs of the paths that begin in state s, follow a sequence of transitions labelled α, and reach a state where $A \to \alpha$ can be reduced.

Equation 5.1 involves the auxiliary matrix $cost(s, A \to \alpha \bullet \beta)$, defined by Equations 5.2 and 5.3, which denotes the cost of a path that begins in state s, is labelled β, and reaches a state where the production $A \to \alpha\beta$ can be reduced.

Equation 5.1 involves a multiplication with the matrix ΔZ, where Z is the set of lookahead symbols that allow reducing $A \to \alpha$ in state s'. The matrix ΔZ, pronounced "filter Z", is defined as follows:

$$\begin{aligned} (\Delta Z)_{a,z} &= 0 \quad \text{if } a = z \text{ and } a \in Z \\ (\Delta Z)_{a,z} &= \infty \quad \text{otherwise} \end{aligned}$$

We note that ΔT is the identity matrix. In general, ΔZ is a variant of the identity matrix where the diagonal entries whose index lies outside of Z are set to ∞ instead of 0.

A multiplication $M \cdot \Delta Z$ produces a copy of the matrix M where all columns whose index lies outside Z are set to ∞. The multiplication with ΔZ in Equation 5.1

reflects the idea that if reduction is permitted, then it has zero cost since no input symbol is consumed; if it is not permitted, then it has infinite cost; and whether it is permitted depends on the lookahead symbol.

Equations 5.2 and 5.3 Equation 5.2 concerns the case of a non-empty path, labelled with the sentential form $x\beta$. The cost matrix for the first transition, which leaves the state s and is labelled x, and the cost matrix for the remaining transitions, which begin in the state $\mathit{target}(s,x)$[2] and follow the path labelled β, are combined via matrix multiplication. Indeed, multiplication expresses the idea that the "lookahead symbol" that is chosen while examining the first transition must also be the "first symbol" that is chosen while examining the remaining transitions.

Equation 5.3 concerns the case of an empty path, labelled with the sentential form ϵ. The cost matrix for this path is the identity matrix I_T.

Recursion Equations 5.1–5.2 are mutually recursive. They are monotonic with respect to the ordering $(\bar{\mathbb{N}}, \leq)$, lifted pointwise to matrices. That is, when the cost matrices that appear in the right-hand side of an equation grow, the value of this right-hand side grows as well. This ensures that Equations 5.1–5.3 have least and greatest solutions. The desired cost matrices form the least solution.

5.3.3 Computing Transition Costs

The equations 5.1–5.3 form a system of monotonic equations whose variables are matrices of costs and whose right-hand sides involve operations on matrices, such as multiplication $\cdot$ and minimum $\oplus$.

To solve these equations, one approach is to use a generic least fixed point computation algorithm: see, e.g., Pottier (2009) and references therein. However, such an approach is very naïve. Indeed, working at the level of matrices is too coarse-grained: when a matrix is updated, every right-hand side where this matrix appears must be re-evaluated. Thus, even a tiny update (one that affects only a few matrix entries) requires a large amount of computation.

A better approach is to work at the level of matrix entries. Equations 5.1–5.3 can be reformulated as a system of equations whose variables are costs and whose right-hand sides involve operations on costs, such as minimum and addition. The benefit of such a reformulation is twofold. First, this reduces the amount of computation that a generic fixed point computation algorithm must perform. Second, because addition is a superior function,[3] the reformulated problem is of a specific form that can be efficiently solved by Knuth's generalisation of Dijkstra's algorithm Knuth (1977). This algorithm is potentially more efficient than a generic fixed point algorithm, and its worst-case time complexity is easier to analyse.

[2] It may be the case that there is no transition labelled x out of the state s, in which case $\mathit{target}(s,x)$ is undefined. Then, we consider that the right-hand side of Equation 5.2 yields a matrix where every entry is ∞.

[3] A function f of type $\bar{\mathbb{N}} \times \bar{\mathbb{N}} \to \bar{\mathbb{N}}$ is superior if $f(x,y) \geq \max(x,y)$ holds.

Unfortunately, this improved approach remains naïve: its space and time complexities are quite poor.

The space complexity of this approach can be assessed as follows. The number of matrices of the form $cost(tr)$ is $|Tr|$, the number of transitions in the automaton. The number of matrices of the form $cost(s, A \to \alpha \bullet \beta)$ is the total star size $\mathscr{S}$ Pottier (2016). We believe that $|Tr| \leq \mathscr{S}$ holds. Thus, the number V of cost variables is $O(\mathscr{S} \times |T|^2)$. This is also the space required by the cost matrices alone.

Provided a suitable "priority queue" data structure is used, the time complexity of Knuth's algorithm is $O(V \log V + E)$ where E is the number of dependencies between variables that the equations exhibit. Here, some variables participate in B dependency relations, where B is the maximum number of productions associated with each nonterminal symbol; some participate in $|T|$ relations. Assuming that B is smaller than $|T|$, which is usually true in practice, we find that E is $O(V \times |T|)$. Assuming that $\log \mathscr{S}$ is smaller than $|T|$, which is usually true in practice, we find that $V \log V$ is dominated by E, so the time complexity of this approach is $O(\mathscr{S} \times |T|^3)$. This analysis seems to be supported by our experiments (§5.7).

In practice, what does this mean? The largest grammar in our test suite, a grammar for C++ Hashimoto (2021), has 422 terminal symbols and 755 nonterminal symbols. The LR(1) automaton produced by the Menhir parser generator has over 10^4 states and 5×10^5 transitions. Its total star size $\mathscr{S}$ is over 2×10^6. This is the number of cost matrices that we wish to compute. Since the size of a matrix is $422^2 = 1.78 \times 10^5$, the total size of the matrices is over 3.63×10^{11}. Assuming that a matrix entry occupies a 64-bit word, this requires at least 2.6 terabytes of memory. Considering today's economic constraints on the price and availability of memory, this is not realistic: the naïve algorithm does not scale well. The optimisations that we describe reduce the space consumption of the algorithm, in this case, by a factor of more than 10^4.

5.4 Overview of Optimisations

Four main ideas allow us to reduce the space and time requirements of the naïve algorithm, namely: *merging of identical columns*, *merging of peer rows*, *matrix chain multiplication optimisation*, and *maximal sharing of matrix multiplications*. These optimisations reduce the size of the cost matrices that we wish to compute; therefore, the computation time is reduced as well. The computation itself is performed in the manner that was suggested (§5.3.3), that is, by formulating the problem as a system of equations bearing on cost variables, and by using Knuth's algorithm to solve these equations.

Merging Identical Columns In general, at each step, the lookahead symbol determines the behaviour of the LR(1) automaton: to shift a terminal symbol, to reduce a production, or to report a syntax error. However, it is often the case that two distinct lookahead symbols dictate the same behaviour. For instance, if the terminal symbols z_1 and z_2 are both members of the set $reduce(s', A \to \alpha)$, then, in state s', the distinction between these symbols is irrelevant: both allow reducing the production $A \to \alpha$.

This implies that some cost matrices have repeated columns. Our first optimisation is to *precompute which columns must be identical* and *merge* them ahead of time so that they are computed only once.

Merging Peer Rows Suppose that the cost computation involves a matrix multiplication $M \cdot M'$, which arises as an instance of Equation 5.2. Thanks to the previous optimisation, we may happen to know ahead of time that the columns indexed by b_1 and b_2 in the matrix M must be equal. This implies that the rows indexed by b_1 and b_2 in the matrix M' participate in the multiplication as *peers*: they always appear in an expression of the form $(M_{a,b_1} \otimes M'_{b_1,c}) \oplus (M_{a,b_2} \otimes M'_{b_2,c})$, where the equality $M_{a,b_1} = M_{a,b_2}$ holds, so this expression is equal to $M_{a,b_1} \otimes (M'_{b_1,c} \oplus M'_{b_2,c})$. Thus, to compute the result of the multiplication, it is not necessary to have separate access to the rows $M'_{b_1,\_}$ and $M'_{b_2,\_}$. Instead, it suffices to have access to the combination of these rows by the function "minimum", that is, $M'_{b_1,\_} \oplus M'_{b_2,\_}$. This indicates that, even though these rows are not identical, they can be merged. Instead of allocating space for two rows, one can allocate space for a single row and use it to store $M'_{b_1,\_} \oplus M'_{b_2,\_}$.

This reasoning assumes that the matrix M' participates in only one multiplication, namely $M \cdot M'$. If M' participates in several multiplications (a situation that typically arises when a state has several incoming transitions) then the rows $M'_{b_1,\_}$ and $M'_{b_2,\_}$ can be merged only if every matrix that appears on the left-hand side of a multiplication $\_ \cdot M'$ is known to have identical columns at indices b_1 and b_2.

Our second optimisation, therefore, is to *determine ahead of time which rows are peers* and *merge* them.

Matrix Chain Multiplication Optimisation Once certain columns and rows have been merged, as described above, we no longer work with cost matrices whose dimensions are $|T|$ by $|T|$. Instead, we compute compact cost matrices, which are rectangular and can have various dimensions. Therefore, our third optimisation is to *identify chains of multiplications* and *optimise each such chain* by exploiting the associativity of multiplication. A standard dynamic programming algorithm is used to solve the Matrix Chain Ordering Problem (Cormen et al., 2009, §15.2) Wikipedia (2021c).

As a consequence of this optimisation, we do not necessarily compute the auxiliary matrices $cost(s, A \to \alpha \bullet \beta)$ described in the previous section. Because we optimise the chain of multiplications expressed by Equation 5.2, we may go through a different family of auxiliary matrices, while preserving the end result of the chain, namely the matrix $cost(s, A \to \epsilon \bullet \alpha)$, which appears in Equation 5.1.

Maximal Sharing of Matrix Multiplications Often, two paths of interest in the automaton share a suffix. This in turn means that two matrix multiplication chains share a suffix.

For instance, suppose $target(s_1, \alpha) = target(s_2, \alpha) = s'$, that is, the path labelled α out of s_1 and the path labelled α out of s_2 lead to a common state s'. Suppose that

there exists a production $A \to \alpha\beta$ and that the states s_1 and s_2 have outgoing transitions labelled A. Then, we would like to compute the cost matrices $cost(s_1, A \to \epsilon\bullet\alpha\beta)$ and $cost(s_2, A \to \epsilon\bullet\alpha\beta)$. Each of these matrices is the result of a multiplication chain, and these two chains share a common suffix: indeed, both of them involve the matrix $cost(s', A \to \alpha\bullet\beta)$.

This raises a question: when we optimise two multiplication chains that share a suffix, should we optimise each chain separately, ignoring the presence of a shared suffix, or should we decompose the two chains into three subchains (two prefixes and a shared suffix) that do not overlap and can be optimised independently? The first option yields greater freedom in optimising long multiplication chains, but destroys sharing that is present in the initial formulation of the problem and thereby potentially creates redundant computation. The second option preserves sharing but gives up opportunities for optimising long chains of multiplications.

After experimenting with both options, we choose the first one, which seems most beneficial. To make up for the loss of sharing that it causes, *we explicitly impose maximal sharing:* that is, after each multiplication chain has been separately optimised, we use hash-consing Filliâtre and Conchon (2006); Wikipedia (2021a) to discover matrix multiplication trees that appear several times, and share them again. This is our fourth and last major optimisation.

In the next section (§5.5), we describe the first two key optimisations, namely the merging of rows and columns, which allows us to compute compact cost matrices. Then (§5.6), we describe some implementation details that lead to increased efficiency in practice.

5.5 Merging Rows and Columns

The cost matrices have dimension $|T| \times |T|$: their rows and columns are indexed with terminal symbols. In this section, we would like to merge certain rows and certain columns. We wish to do so a priori: that is, before any matrices are constructed, we wish to compute which rows and which columns can be merged. This allows us to directly construct *compact cost matrices* whose rows and columns are indexed by *equivalence classes* of terminal symbols.

We proceed as follows for each transition (s, x):

1. We compute two partitions, named $\mathbf{first}(s)$ and $\mathbf{follow}(s, x)$. The partition $\mathbf{first}(s)$ tells which rows in the cost matrix associated with this transition can be merged; the partition $\mathbf{follow}(s, x)$ tells which columns can be merged.

2. We compute a compact cost matrix $\mathbf{ccost}(s, x)$ whose row indices range over $\mathbf{first}(s)$ and whose column indices range over $\mathbf{follow}(s, x)$. An entry in this matrix is identified by two equivalence classes $p \in \mathbf{first}(s)$ and $q \in \mathbf{follow}(s, x)$.

These two phases are described in the next two subsections.

5.5.1 Which Rows and Columns Can Be Merged?

To begin, we define the functions **first** : $S \to \text{Part}(T)$ and **follow** : $Tr \to \text{Part}(T)$. These functions determine which columns and rows we can safely merge.

As explained earlier (§5.4), we merge two columns when we know ahead of time that they would be identical. Thus, we need **follow** to satisfy the following soundness property:

$$\forall q \in \mathbf{follow}(tr) \quad \forall b,\ b' \in q \quad \forall a \quad cost(tr)_{a,b} = cost(tr)_{a,b'}$$

That is, if the partition **follow**(tr) indicates that the terminal symbols b and b' are equivalent (members of the same class), then the columns indexed by b and b' in the matrix $cost(tr)$ must be identical. The reverse implication is not true: the columns indexed by b and b' may happen to be identical even when this is not predicted by the partition **follow**(tr), that is, even when b and b' are not members of the same class. Thus, the partition **follow**(tr) is a conservative prediction of which columns will be identical.

Another way to think about **follow**(tr) is that this partition indicates which distinctions between lookahead symbols definitely have no influence on how the transition tr can be taken.

It does not seem easy to state a soundness property that **first** must satisfy. As explained earlier (§5.4), two rows are merged not when they are identical, but when we know in advance that only the combination of these rows by the function "minimum" is relevant. So, we state no such property.

The partitions **first**(s) and **follow**(s, x) are characterised by a set of recursive monotone equations: they form the greatest fixed point of these equations. Any fixed point is sound; the greatest fixed point allows the greatest amount of merging. The equations are as follows:

$$\mathbf{follow}(s, a) = \top \tag{5.4}$$

$$\mathbf{follow}(s, A) = \bigwedge_{A \to \alpha} \begin{cases} \textit{let } s' = \textit{target}(s, \alpha) \textit{ in} \\ \textit{let } Z = \textit{reduce}(s', A \to \alpha) \textit{ in} \\ \mathbf{first}(s') \downarrow Z \end{cases} \tag{5.5}$$

$$\mathbf{first}(s) = \bigwedge_{tr \in \mathbf{incoming}(s)} \mathbf{follow}(tr) \tag{5.6}$$

Figure 5.2: Which Rows and Columns Can Be Merged

Equation 5.4 indicates that the partition **follow**(s, a) is $\top$. This is sound because all columns in the matrix $cost(s, a)$ are identical (§5.3.2). An equivalent statement is that **follow**(tr) is $\top$ when tr is a shift transition.

Equation 5.5 Equation 5.5 defines **follow**(s, A), where we assume that the state s has an outgoing transition labelled A. In doing so, it aims to answer the question: "which lookahead symbols influence the manner in which this transition can be

taken?" If two symbols b and b' lie in the same class, then the input sentences that allow taking this transition are the same, regardless of whether the lookahead symbol is b or b'.

This transition can be taken if there is a production $A \to \alpha$ such that (1) the sentential form α can be recognised, taking the automaton from state s to state $s' = \mathit{target}(s, \alpha)$, and (2) in state s', the production $A \to \alpha$ can be reduced. Any such production can be chosen, hence Equation 5.5 involves a meet over all such productions. Let us now focus on one such production and examine the right-hand side of Equation 5.5.

In the state s', reducing the production $A \to \alpha$ is permitted if and only if the lookahead symbol is a member of the set $\mathit{reduce}(s', A \to \alpha)$. Let us refer to this set as Z. Then, there definitely is *no need to distinguish between non-members of Z*: indeed, all lookahead symbols outside Z forbid reducing the production $A \to \alpha$. This explains why a partition of the form "$\ldots \downarrow Z$" appears in Equation 5.5.

There remains to explain what partition to choose in place of "$\ldots$". This determines which terminal symbols inside Z must be distinguished.

Choosing $\bot$ means that all symbols inside Z must be distinguished. That would be sound but pessimistic, largely preventing the merging of rows and columns.

Choosing $\top$ means that no distinction inside Z is necessary. That would be unsound. Indeed, before reducing $A \to \alpha$, the right-hand side α must be recognised, and this process may be influenced by the lookahead symbol.

What partition describes this influence in a sound way, and is not as pessimistic as $\bot$? We remark that, in order to go from the state s along the path labelled α to the state s', the automaton must (as its final step) enter the state s', and to do so, it must follow some transition $tr \in \mathbf{incoming}(s')$. Therefore, two lookahead symbols must be distinguished only if this distinction influences some transition $tr \in \mathbf{incoming}(s')$. In other words, a sound choice is the meet, over all such transitions, of the partitions $\mathbf{follow}(tr)$. This happens to be precisely $\mathbf{first}(s')$.

Equation 5.6 Equation 5.6 defines the partition $\mathbf{first}(s)$ as the meet of the partitions $\mathbf{follow}(tr)$, where tr ranges over the incoming transitions of the state s. The reason for this is as follows. The partition $\mathbf{first}(s)$ is supposed to determine which rows in a cost matrix $\mathit{cost}(s, x)$ can be merged. As explained earlier (§5.4), two rows in such a matrix can be merged if this matrix is always multiplied (on its left) with matrices where the corresponding columns are identical. This is the case if every incoming transition tr of state s has a cost matrix $\mathit{cost}(tr)$ where these columns are identical. Thus, for every $tr \in \mathbf{incoming}(s)$, we need $\mathbf{first}(s) \preceq \mathbf{follow}(tr)$ to hold. This explains Equation 5.6.

Because all of the incoming transitions of a state s carry the same symbol, either all of them are shift transitions, labelled with a terminal symbol a, or all of them are goto transitions, labelled with a nonterminal symbol A. In the former case, Equation 5.6 can be simplified: by Equation 5.4, every incoming transition tr satisfies $\mathbf{follow}(tr) = \top$, so $\mathbf{first}(s)$ is $\top$.

5.5.2 A Characterisation of Compact Cost Matrices

Compact Cost Matrices For each transition (s, x), we wish to compute a compact cost matrix $\mathbf{ccost}(s, x)$, whose row indices are members of $\mathbf{first}(s)$ and whose column indices are members of $\mathbf{follow}(s, x)$.

We now provide several equations that characterise these matrices, and allow them to be computed.
These equations involve a family of auxiliary matrices $ccost(s, A \to \alpha\bullet\beta)$ of dimensions $\mathbf{first}(s)$ by $\mathbf{first}(target(s, \beta))$.

The compact cost matrix $\mathbf{ccost}(s, a)$ associated with a shift transition is defined as follows:

$$\begin{array}{ll} \mathbf{ccost}(s,a)_{p,q} = 1 & \text{if } a \in p \\ \mathbf{ccost}(s,a)_{p,q} = \infty & \text{otherwise} \end{array}$$

Since $\mathbf{follow}(s, a)$ is $\top$, a one-class partition, this matrix has width one: it is a column matrix. It has exactly one "1" entry, found at the unique index p such that $a \in p$ holds, and "∞" entries everywhere else.

The compact cost matrix $\mathbf{ccost}(s, A)$ associated with a goto transition and the auxiliary matrices $ccost(s, A \to \alpha\bullet\beta)$ are defined as follows:

$$\mathbf{ccost}(s, A) = \bigoplus_{A \to \alpha} \begin{cases} ccost(s, A \to \epsilon\bullet\alpha) \cdot \\ creduce(s, A \to \alpha) \end{cases} \tag{5.7}$$

$$ccost(s, A \to \alpha\bullet x\beta) = \begin{cases} let\ s' = target(s, x)\ in \\ \mathbf{ccost}(s, x) \cdot \\ \mathbf{coerce}(\mathbf{follow}(s, x), \mathbf{first}(s')) \cdot \\ ccost(s', A \to \alpha x\bullet\beta) \end{cases} \tag{5.8}$$

$$ccost(s, A \to \alpha\bullet\epsilon) = I_{\mathbf{first}(s)} \tag{5.9}$$

Figure 5.3: A Characterisation of Compact Cost Matrices

Equation 5.7 Equation 5.7 resembles Equation 5.1 in Figure 5.1, but the square matrix $\Delta reduce(target(s, \alpha), A \to \alpha)$ that is visible in Equation 5.1 is replaced with the rectangular matrix $creduce(s, A \to \alpha)$, whose dimensions are $\mathbf{first}(target(s, \alpha))$ by $\mathbf{follow}(s, A)$. This matrix is defined as follows:

$$\begin{array}{ll} creduce(s, A \to \alpha)_{p,q} = 0 & \text{if } q \subseteq Z \wedge q \subseteq p \\ & \text{where } Z = reduce(s', A \to \alpha) \\ & \text{and } s' = target(s, \alpha) \\ creduce(s, A \to \alpha)_{p,q} = \infty & \text{otherwise} \end{array}$$

This generalises the definition of the matrix ΔZ to a setting where, instead of two terminal symbols a and z, we consider two classes $p \in \mathbf{first}(target(s, \alpha))$ and $q \in \mathbf{follow}(s, A)$.

It is worth noting that, by virtue of Equation 5.5, we have $\mathbf{follow}(s, A) \preceq Z?$. This implies that q must lie either entirely within Z or entirely outside of Z. Thus, the test $q \subseteq Z$ is cheap: it suffices to pick an arbitrary element of q and to test whether it is a member of Z.

Also by virtue of Equation 5.5, we find that $\mathbf{follow}(s, A)$ refines $\mathbf{first}(\mathit{target}(s, \alpha)) \downarrow Z$, from which we can deduce that $\mathbf{follow}(s, A) \downarrow Z$ refines $\mathbf{first}(\mathit{target}(s, \alpha)) \downarrow Z$. This implies that the condition $q \cap Z \subseteq p \cap Z$ can be decided by picking an arbitrary element of q and testing whether it is a member of $p \cap Z$. Assuming that $q \subseteq Z$ holds, this can be rephrased as follows: $q \subseteq p$ can be decided by picking an arbitrary element of q and testing whether it is a member of p.

Equation 5.8 Equation 5.8 is analogous to Equation 5.2, but a *coercion matrix* must be inserted in the middle of the matrix multiplication expression, because the two matrices $\mathbf{ccost}(s, x)$ and $\mathit{ccost}(\mathit{target}(s, x), A \to \alpha x \bullet \beta)$ do not have matching numbers of columns and rows. The columns of the former matrix are indexed with classes in $\mathbf{follow}(s, x)$, while the rows of the latter are indexed with classes in $\mathbf{first}(\mathit{target}(s, x))$. There is a refinement relation between these partitions: as a consequence of Equation 5.6, we have $\mathbf{first}(\mathit{target}(s, x)) \preceq \mathbf{follow}(s, x)$. In such a situation, it is possible to construct a coercion matrix, whose definition and meaning are explained next.

Coercion Matrices Let $P, Q \in \mathrm{Part}(T)$ be two partitions such that $Q \preceq P$ holds, that is, Q refines P. The matrix $\mathbf{coerce}(P, Q)$, whose rows are indexed with classes in P and whose columns are indexed with classes in Q, is defined as follows:

$$\mathbf{coerce}(P, Q)_{p,q} = \begin{cases} 0 & \text{if } q \subseteq p \\ \infty & \text{otherwise} \end{cases}$$

Because Q refines P, every class $q \in Q$ is a subset of some class in P. Thus, for every $p \in P$ and $q \in Q$, the class q is a subset of either p or $T \setminus p$. Thus, the condition $q \subseteq p$ can be decided by testing an arbitrary element of q.

In the special case where P and Q are the same partition, the coercion matrix $\mathbf{coerce}(P, Q)$ is square: it is an identity matrix. Otherwise, the matrix $\mathbf{coerce}(P, Q)$ is rectangular: it has more columns than rows. The row identified by the class p has a "0" entry at each column q such that the class q is a piece of the class p, and "∞" entries elsewhere. Since every class $q \in Q$ is a subset of exactly one class in P, on every column, there is exactly one "0" entry.

The matrix $\mathbf{coerce}(P, Q)$ has the remarkable property that, when used in a matrix multiplication, it serves as an adapter: it can expand a matrix from dimension P to dimension Q and shrink a matrix from dimension Q to dimension P.

Indeed, the product $M \cdot \mathbf{coerce}(P, Q)$ expands the matrix M from width P to width Q by duplicating columns: the column found at index p in the matrix M is copied in the matrix $M \cdot \mathbf{coerce}(P, Q)$ at every index q such that $q \subseteq p$ holds.

In the opposite direction, the product $\mathbf{coerce}(P, Q) \cdot M$ shrinks the matrix M from height Q to height P by merging rows: the row found at index p in the product

coerce$(P, Q) \cdot M$ is the pointwise minimum of the rows found in the matrix M at indices q such that $q \subseteq p$ holds.

Soundness of Compact Cost Matrices The cost matrix $cost(s, x)$ and the compact cost matrix **ccost**(s, x) are related as follows:

Claim 2 *Let* (s, x) *be a transition. Then, we have:*

$$\begin{aligned} \textbf{\textit{coerce}}(\textbf{\textit{first}}(s), \bot) \cdot cost(s, x) = \\ \textbf{\textit{ccost}}(s, x) \cdot \textbf{\textit{coerce}}(\textbf{\textit{follow}}(s, x), \bot) \end{aligned}$$

The multiplication by **coerce**(**follow**$(s, x), \bot$) on the second line duplicates some columns of the compact cost matrix, so as to undo the merging of identical columns.

The multiplication by **coerce**(**first**$(s), \bot$) on the first line compresses some rows of the cost matrix by computing their pointwise minimum. This reflects the fact that the compact matrix contains less information than the cost matrix: some information is lost when peer rows are merged.

5.6 Implementation Details

We now propose an overview of several implementation details and optimisations that, together, yield significant additional performance improvements.

5.6.1 Representing Partitions

The computation of **first** and **follow** partitions (§5.5.1) requires an efficient representation of partitions of the set T of terminal symbols. Several options come to mind. Following Lee (1981), one could represent a partition as an array of size $|T|$, where each terminal symbol is mapped to a representative element of its block. Or, following Hopcroft (1971) and Paige and Tarjan (1987), one could adopt the mutable data structures used in partition refinement algorithms, which involve doubly-linked lists of elements and doubly-linked lists of blocks. In our setting, a persistent and compact representation is desirable. We represent a partition as an (unordered) linked list of blocks, and represent a block as a sparse bit set.[4] In the worst case, a partition occupies $O(|T|)$ space in memory; yet, in several common cases, it can require significantly less space. Indeed, we allow one block to be omitted from the linked list of all blocks. This causes no loss of information: the omitted block is the complement of the blocks that do appear in the list. For example, in the representation of the partitions $Z?$ and $Z!$, the block $T \setminus Z$ can be omitted. Thus, these partitions occupy only $O(|Z|)$ space.

Our implementation of sparse bit sets is able to efficiently compute a "unique-or-shared prefix" of two nonempty bit sets Z_1 and Z_2. Let us write $Z < Z'$ when

[4] Whereas an ordinary bit set would be represented as an array of machine words and would have size $O(|T|)$, regardless of its cardinality, a sparse bit set is represented as linked list of pairs of an integer offset and a machine word, and has size $O(n)$, where n is the cardinality of the bit set.

every element of Z is less than every element of Z', according to an arbitrary, fixed total order on T. Then, the unique-or-shared-prefix operation produces either (a) a nonempty subset Z'_1 of Z_1 such that $Z'_1 < Z_1 \setminus Z'_1$ and $Z'_1 < Z_2$ are satisfied; or (b) symmetrically, a nonempty subset Z'_2 of Z_2 such that $Z'_2 < Z_1$ and $Z'_2 < Z_2 \setminus Z'_2$ are satisfied; or (c) a nonempty subset of $Z_1 \cap Z_2$ such that $Z'_1 < Z_1 \setminus Z'_1$ and $Z'_2 < Z_2 \setminus Z'_2$.

This operation is used in the construction of a carefully optimised implementation of $\bigwedge$, the meet of a family of k partitions. This algorithm is inspired by the standard k-way merge algorithm Wikipedia (2021b), whose main loop involves a priority queue.

In our setting, the priority queue is populated with pairs of a block and its provenance. The priority of such a pair is the minimum element of the block. A provenance is a set of identifiers, where initially each block in each of the k partitions receives a unique identifier. Pairing a block with a provenance allows us to record which initial blocks this block is a subset of. When the priority queue is first populated, the provenance of each block is a singleton set containing just the identifier of this block.

The algorithm's main loop proceeds as follows. As long as the queue has size greater than one, the two blocks with minimum priority, say Z_1 and Z_2, are extracted, and their unique-or-shared prefix, say Z, is computed. If it is unique, then it is set aside, together with its provenance; if it is shared, then it is inserted into the queue again with an updated provenance, the union of the provenances of Z_1 and Z_2. In either case, the remainders of the two blocks, namely $Z_1 \setminus Z$ and $Z_2 \setminus Z$, are inserted into the queue (if they are non-empty).

Once this loop terminates, the blocks that have been set aside are sorted according to their provenance, and those with the same provenance are merged. The list of blocks thus obtained is the meet of the k input partitions.

5.6.2 Approximating first and follow Partitions

We have presented the computation of **first** and **follow** as the computation of a greatest fixed point. A standard fixed-point computation algorithm Pottier (2009), based on a chaotic iteration scheme, can be used to compute this fixed point in an exact way. In our experience, however, chaotic iteration can be slow, so this computation can become a bottleneck. To address this issue, we use a faster algorithm, which visits each cost variable at most twice and computes a post-fixed point, that is, a solution of the inequations obtained by replacing equality "=" with the refinement relation "$\preceq$" in Equations 5.7–5.9. Because every post-fixed point refines the greatest fixed point, the partitions that we obtain are an under-approximation of **first** and **follow**. Such an approximation does not endanger the correctness of our approach; it just means that we do not merge rows and columns as aggressively as we could with the greatest fixed point.

Our algorithm examines the dependency graph formed by all instances of Equations 5.7–5.9, computes its strongly connected components, and processes one component at a time, in topological order. Inside each component, it proceeds as follows. By inspection of Equations 5.7–5.9, one can see that the partition variables that be-

long in this component form two families $X_1, \ldots, X_m$ and $Y_1, \ldots, Y_n$[5] and that the inequations these variables must satisfy are of four kinds:

$$\begin{array}{lll} X_i \preceq P_i & \text{where } P_i \text{ is a constant partition} & (A) \\ X_i \preceq Y_j \downarrow Z_{ij} & \text{where } Z_{ij} \text{ is a constant set} & (B) \\ Y_j \preceq Q_j & \text{where } Q_j \text{ is a constant partition} & (C) \\ Y_j \preceq X_i & & (D) \end{array}$$

Our algorithm solves these constraints as follows:

1. Compute $P = \bigwedge_i P_i \wedge \bigwedge_{ij} Z_{ij}? \wedge \bigwedge_j Q_j$.
2. Assign to each X_i the value $P_i \wedge \bigwedge_j (P \downarrow Z_{ij})$.
3. Assign to each Y_j the value $Q_j \wedge \bigwedge_i X_i$.

It is not difficult to check that this assignment satisfies all of the inequations A–D. The second step is where each X_i is assigned an under-approximation of its ideal value; in the third step, each Y_j receives a value as accurate as possible, considering the approximation made in the previous step.

In our experience, this approximation leads to an increase of less than 2% in the size of the compact cost matrices, due to the fact that the merging of rows and columns is no longer as aggressive as it could be. This does not lead to an observable slowdown in the computation of compact cost matrices. On the other hand, the approximate computation of **first** and **follow** can be 10 times faster than the exact computation. As a result, it is no longer a bottleneck.

5.6.3 Replacing Knuth's Priority Queue with a FIFO

We have indicated earlier (§5.3.3) that the computation of the compact cost matrices (§5.5.2) can exploit Knuth's algorithm Knuth (1977), which has known asymptotic worst-case complexity. This is attractive, because it makes it easy to assess the time complexity of our algorithm (§5.3.3). However, we find that a FIFO queue offers similar performances in practice while being simpler to implement. Knuth's algorithm guarantees that every matrix entry is visited at most once, so that, when an entry is visited, it is assigned its final value. By using a FIFO queue instead of a priority queue, we lose this property: an entry may be visited several times. In practice, we find that entries are visited 1.1 times on average. This is a small price to pay. In return, a FIFO queue is easier to implement and can be (marginally) more compact and faster.

5.6.4 Representing Dependencies

During the computation of compact cost matrices (§5.5.2), an efficient representation of the dependencies between matrix entries is needed, so that, when an entry is updated, the entries that depend on it can be enqueued for re-examination. There

[5] The X_i's correspond to **follow** partitions, while the Y_j's correspond to **first** partitions.

is a space-time trade-off between an explicit representation, where the dependency edges form a graph that is explicitly stored in memory, and an implicit representation, where the dependency edges are not stored in memory, but can be computed on demand.

The number of matrix entries can be huge and is the dominating factor that determines the memory requirement of this computation. Thus, we strive to maintain as little information per matrix entry as possible, and cannot use a fully explicit representation of dependency edges.

To every entry in every compact cost matrix, we assign a unique number. Then, with each entry, we associate three pieces of information, which occupy three words in memory. One piece is the content of this matrix entry, that is, a cost, represented as a machine integer. The second piece is a reference to the matrix to which this entry belongs. (Because entries are numbered sequentially and because the dimensions of every matrix are known, this information suffices to also recover the coordinates of this entry within its matrix.) The third and last piece is either *null* or the number of another matrix entry: this way, the FIFO queue of all "dirty" matrix entries is represented as a linked list.

To encode the dependencies between matrix entries, we combine a coarse-grained explicit scheme and a fine-grained implicit scheme. At the outer level, dependencies between matrices are explicitly stored in memory. At the inner level, dependencies between matrix entries are computed on demand. Given the coordinates of an entry and the (outer-level) dependencies of the matrix in which it appears, we are able to enumerate the matrix entries that depend on this entry.

5.6.5 Representing Constant Matrices

In general, a cost matrix is represented as an array of entries. This can be quite expensive in terms of space. We identify several cases where a specialised representation is beneficial.

The matrices $\mathbf{ccost}(s, a)$ are column matrices where a single row contains a "1" entry. We store just the index of this row.

The auxiliary matrices $\mathit{ccost}(s, A \to \alpha \bullet \epsilon)$ need not be represented at all. Indeed, they are identity matrices. If they appear in a multiplication, then this multiplication can be simplified. If they appear in a minimum $\oplus$, then the result matrix is eagerly initialised with appropriate "0" entries.

The coercion matrices that appear in products of the form $M_1 \cdot \mathbf{coerce}(P, Q) \cdot M_2$ have two (redundant) representations. One representation allows an efficient propagation of an update in M_1 to the result of the multiplication; the other representation allows an efficient propagation of an update in M_2. Both are sparse representations of the coercion matrix; one representation offers efficient access to the "0" entries along every row; the other offers efficient access to the "0" entries along every column.

Because the matrices $\mathit{creduce}(s, A \to \alpha)$ always appear on the right-hand side of a matrix multiplication, a single sparse representation, which allows efficient access to the "0" entries along every row, suffices.

5.6.6 Testing Block Inclusion

The construction of the *creduce* and **coerce** matrices (§5.5.2) involves many tests of the form $q \subseteq p$, where it is known that either $q \subseteq p$ or $q \cap p = \emptyset$ must hold. Indeed, q and p are blocks in two partitions Q and P such that $Q \preceq P$. Therefore, the condition $q \subseteq p$ can be decided by picking an arbitrary member of q and testing whether it is a member of p.

5.7 Experimental Evaluation

5.7.1 Comparison of Three Algorithms

We compare the performance of three algorithms, namely Pottier's algorithm Pottier (2016), dubbed "P", the naïve algorithm based on cost matrices (§5.3), dubbed "CM", and our optimised algorithm based on compact cost matrices (§5.5), dubbed "CCM". We run each algorithm on the 394 grammars of Menhir's test suite. This suite contains only a handful of small artificial grammars; the majority of the grammars in it are real-world grammars, ranging from small to large grammars. The tests are run sequentially and repeated five times; only the best time is kept. We use an Intel Xeon E7-4870 machine running at 2.4Ghz, with 1TB of RAM.

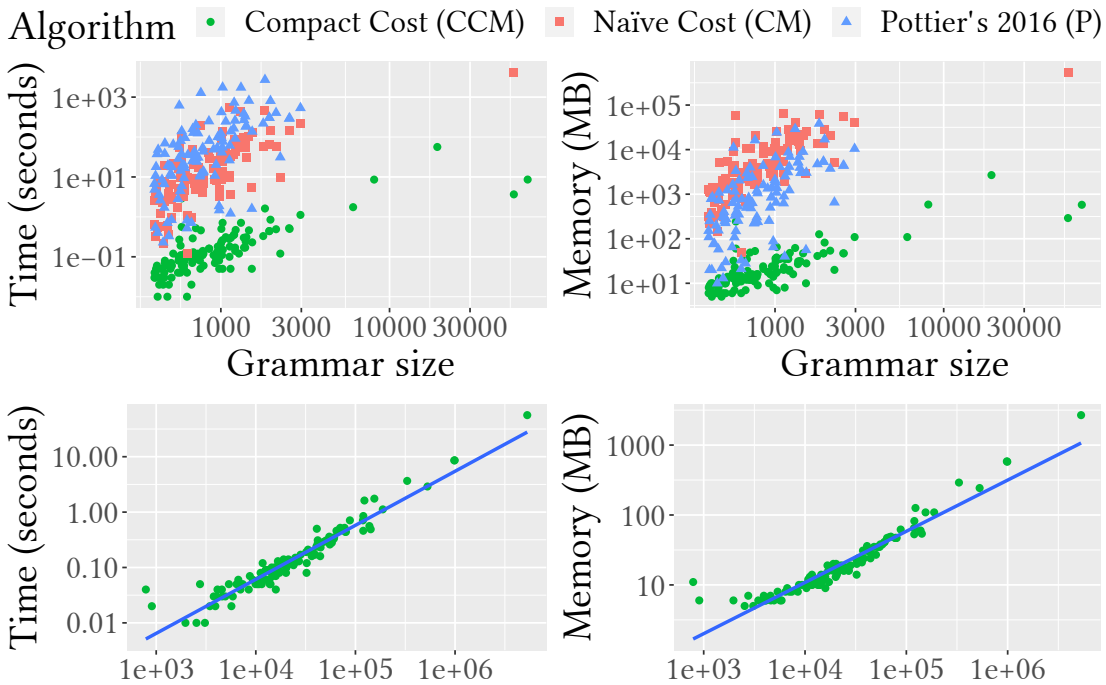


Figure 5.4: Performance Aspects of Reachability Algorithms

Our performance results are summarised in Figure 5.4 in four graphs. Each graph uses a logarithmic scale along each axis.

The top-left and top-right graphs show the time and memory consumption of the three algorithms. To reduce the noise, only grammars whose size[6] is larger than 400 are taken into account when an average[7] is computed. A first finding is that CM can be competitive with P: on average, it is 3 times faster, but consumes 5 times more memory. A second finding is that CCM is the clear winner. On average, it is 95 times faster and uses 215 times less memory than CM. Compared with P, it is on average 300 times faster and uses 42 times less memory. In fact, the larger the grammar, the greater the gain: our algorithm can be up to 2700 times faster than Pottier's, and can use as little as 470 times less memory.

It is worth mentioning that, as a result of its much smaller time and space requirements, CCM is able to handle much larger grammars than P and CM.

The bottom-left and bottom-right graphs in Figure 5.4 are an attempt to predict CCM's time and memory requirements. We find that both are correlated[8] with the parameter $\mathcal{U}$, which we define as follows:

$$\mathcal{U} = \sum_A edges(A) * prodsize(A)$$

where $edges(A)$ is the number of transitions labelled A and $prodsize(A)$ is the sum of the lengths of the productions associated with the nonterminal symbol A.

5.7.2 Impact of Successive Optimisations

Table 5.1 shows the impact of several optimisations (§5.4) that form a path from Algorithm CM to Algorithm CCM. The three optimisations that we consider are: (1) merging rows and columns, (2) reordering matrix products, and (3) sharing common subexpressions in matrix products. Each optimisation requires the previous one: reordering applies after merging; sharing applies after reordering.

As explained earlier (§5.4), before applying optimisations (1), (2), and (3), we take a first step, namely: (0) abandon the sharing of cost matrices that is inherent in Equations 5.1–5.3. This has a negative impact in terms of space and time, but opens the way to the steps that follow, which have positive impact. The four lines in Table 5.1 show the impact of steps (0), (1), (2), (3) respectively.

We measure the impact of each optimisation on the number of matrix entries, V, and on the number of dependencies between matrix entries, E. These quantities correspond to the parameters V and E that appear in the complexity analysis of Knuth's algorithm (§5.3.3). The parameter V is directly related to the memory requirement of Knuth's algorithm, while the parameter E is directly related to its time requirement. Algorithm CM is an instance of Knuth's algorithm, so its space and time complexity is directly related with V and E. Algorithm CCM consists of several precomputation phases (namely merging, reordering, sharing), followed with a run of Knuth's algorithm. The complexity of this last phase is still directly related with V

[6] The size of a grammar is the sum of the lengths of its productions.

[7] When an "average" is mentioned, a geometric mean is meant.

[8] Both have a Pearson correlation coefficient of 0.998 with $\mathcal{U}$.

and E. We find that, in practice, the pre-computation phases and the last phase have roughly comparable execution time.

Table 5.1: Impact of Successive Optimisations

	Impact on V	Impact on E
Unsharing	on avg. $\times 1.53$ up to $\times 2.59$	on avg. $\times 2.08$ up to $\times 3.72$
Merging	on avg. $/ 5.59 \times 10^3$ up to $/ 9.04 \times 10^4$	on avg. $/ 2.10 \times 10^5$ up to $/ 1.55 \times 10^7$
Reordering	on avg. $/ 0.89$ up to $/ 1.73$	on avg. $/ 2.17$ up to $/ 7.25$
Sharing	on avg. $/ 1.18$ up to $/ 1.92$	on avg. $/ 1.32$ up to $/ 2.52$

Each cell of Table 5.1 shows the impact of an optimisation, that is, the ratio between a quantity before applying this optimisation and this quantity after it has been applied. For instance, the top left cell shows "on avg. $\times 1.53$". This cell appears in the column entitled "Impact on V". This means that, on average, the number of matrix entries is multiplied by 1.53 when sharing is abandoned in step (0). The cell below this one shows "on avg. $/ 5.59 \times 10^3$". This means that, on average, the number of matrix entries is divided by 5.59×10^3 when merging is applied in step (1). This cell also contains "up to $/ 9.04 \times 10^4$", which indicates that, in some cases, the number of matrix entries is divided by 9.04×10^4. Naturally, the impact of an optimisation grows as grammars grow larger.

The second line of Table 5.1 shows that merging alone has a dramatic impact. The last two lines show that reordering and sharing both allow reaping an additional (albeit much smaller) performance improvement. The cumulative effect of all four steps can be computed by multiplying the factors shown in each column.

Interestingly, reordering slightly increases the number of matrix entries on average, as indicated by the number " / 0.89" in the third line of the first column. Indeed, what is optimised is not the total size of the matrices that are being multiplied, but the number of elementary operations required for the multiplications. One can see, on the third line in the second column, that reordering has a positive impact on the parameter E.

5.8 Related Work

The reachability problem that we consider seems closely related to a shortest paths problem, which is well-known to be expressible in terms of cost matrices (Lehmann, 1977). It may be possible to reduce our reachability problem to a shortest paths problem in a suitably constructed graph. We have not attempted to do so; instead, we have expressed the reachability problem directly in terms of cost matrices, and

we have proved that the equations characterising these cost matrices (§5.3) can be solved by using Knuth's generalisation (Knuth, 1977) of Dijkstra's algorithm.

We speed up this naïve algorithm by remarking that, in the specific setting of the LR(1) reachability problem, many rows and columns of the cost matrices can be merged, letting us compute much smaller "compact cost matrices" (§5.4). Several additional optimizations help improve the algorithm's performance in practice (§5.6).

Pottier's reachability algorithm (Pottier, 2016) seems related to the naïve algorithm that we have described: indeed, it uses a priority queue, very much in the same way as Knuth's algorithm (Knuth, 1977), to compute and process a set of reachability facts. Its presentation, however, is not matrix-based, and it does not have the ability to handle several terminal symbols together, as an equivalence class: each reachability fact concerns just one terminal symbol.

The LR(1) reachability problem can be considered a special case of the reachability problem for pushdown systems, which has been thoroughly studied, as it has many applications in model-checking and in program analysis (Bouajjani et al., 1997; Esparza et al., 2000; Lal and Reps, 2006; Reps et al., 2005; Suwimonteerabuth et al., 2006). Although our merging technique is probably specific to LR(1), it would be interesting to draw a detailed comparison with this line of work.

Minamide and Mori (2012) formalise an HTML5 parser as a conditional pushdown system (Li and Ogawa, 2010), that is, a pushdown system extended with the ability to test whether the stack matches a regular expression. They propose a new reachability algorithm, based on earlier algorithms by Bouajjani et al. (1997) and Esparza et al. (2000). They use this algorithm to prove that certain branches in the parser's code are dead, to prove that the parser never attempts to pop an item off an empty stack, and to generate a set of input sentences that achieves good coverage of the parser's code and can be used for testing the HTML5 parsers found in industrial browsers. Although their work is similar to ours in its motivation, the HTML5 parser they consider is different from (and more complex than) an LR(1) parser. As a result, their algorithm is more costly: a 438-line fragment of the HTML5 grammar is analysed in 82 minutes.

By speeding up the computation of the reachable states in an LR(1) parser, our algorithm makes Jeffery's approach to error diagnosis (Jeffery, 2003) more scalable. Indeed, as argued by Pottier (2016), this approach requires computing which error states are reachable and which (minimal) erroneous input sentences lead to these states. As a grammar evolves over time, the maintainer of the grammar must recompute this information over and over. In order to achieve a smooth edit-compile-debug cycle, a fast reachability algorithm is required. Although Pottier (2016) successfully applies this technique to the CompCert C parser, his algorithm requires several minutes to process somewhat larger grammars, such as the OCaml grammar, which our algorithm handles in half a second, and is unable to process much larger grammars, such as the C++ grammar (Hashimoto, 2021) that our algorithm handles in a little less than a minute.

A broad survey of error diagnosis techniques, and of the quality of the diagnostic

messages produced by various techniques and tools, is offered by Becker et al. (2019).

5.9 Conclusion

We present a new algorithm for the reachability problem in LR(1) parsers. This algorithm vastly outperforms the state of the art, which, as far as we know, is represented by Pottier's algorithm (Pottier, 2016). We have not proved the correctness of our algorithm, but we have implemented it in the Menhir parser generator Pottier and Régis-Gianas (2021) and have experimentally verified, using a test suite of 390 grammars, that it produces the same results as Pottier's earlier algorithm.

We expect our new algorithm to make Jeffery's approach to error diagnosis (Jeffery, 2003) more scalable. Furthermore, we believe that a fast reachability algorithm has many potential applications beyond computing the reachable error states. In the future, we wish to investigate its application to more powerful error diagnosis schemes, to error recovery, and to syntactic completion.

Chapter 6

Static analyses

In this chapter, we refine the concepts of viable stacks and reduction automata to precisely characterise the stacks reachable by an LR(1) parser. Our procedure begins with a target state and grows a suffix of such reachable stacks. For each stack suffix, we identify the lookahead symbols that cause parsing failure. Additionally, for reporting purposes, we can generate sentence prefixes leading to these configurations.

By composing these tools, we introduce two static analyses aimed at enhancing error messages for LR grammars:

- **Enumeration**: This method generates a finite set of invalid sentences capable of exercising all reduce-filter patterns.
- **Coverage Analysis**: This checks whether an error specification adequately covers all failing configurations that can be reached by the parser (in other words, all invalid inputs are given at least one explanation).

6.1 Failing Configurations

In Chapter 2, we introduced the set of viable stacks by enumerating paths in the LR(1) automaton and used it to construct the reduction automaton **RA**, which enumerates the sequences of reductions applicable on a viable stack. However, if the LR(1) automaton was pruned to resolve conflicts, these viable stacks can be over-approximations; some viable stacks and their associated sequences of reductions might not be reachable in practice.

We leverage the reachability analysis from Chapter 5 to refine the set of viable stacks and the reduction automaton. This refinement yields the reachable stacks and the reachable reductions automaton, which accurately capture the behaviours of an LR(1) parser with truncated actions.

6.1.1 Reachable Stacks

When compiling an LRGrep expression to an automaton, we used the viable stacks twice: to construct the reduction automaton and then to derive only the subset of

the automaton covering the stacks. The over-approximation is fine in these contexts since it only leads to a marginally bigger automaton. For a static analysis, we need an exact enumeration to avoid reporting uncovered cases that are actually unreachable.

Using the cost matrices of the reachability analysis, we can remove the viable stacks that are unreachable. The *graph of reachable stacks* **RS** is a directed graph which refines the graph of the LR(1) automaton by pairing each state with a class of lookahead terminals and removing the transitions which cannot be taken for the source and target classes.

There is an initial node written `Initial` and the other nodes are pairs (s, c) of an LR(1) state s and a class $c \in \mathbf{first}(s)$. There is an edge $(s, c) \rightarrow_{\mathbf{RS}} (s', c')$ when there is a transition $s \xrightarrow{x} s'$ in the LR(1) automaton and the reachability analysis determined that taking this transition with these classes has a finite cost. Formally, **RS** is the least graph satisfying:

1. $(s_0, T) \in \mathbf{RS}$, the initial LR state is injected into the graph
2. for each $(s, c) \in \mathbf{RS}$, we have $(s', c') \in \mathbf{RS}$ and an edge $(s, c) \rightarrow_{\mathbf{RS}} (s', c')$ if there is a transition $s \xrightarrow{x} s'$ in the LR automaton, if $c' \in \mathbf{first}(s')$, and if the cost is finite:

$$\left(\mathbf{ccost}(s, x) \cdot \mathbf{coerce}(\mathbf{follow}(s, x), \mathbf{first}(s'))\right)_{c,c'} < \infty$$

The reachable stacks are obtained by taking the paths of **RS** starting from the initial node (s_0, T) and projecting each node (s, c) on s, the LR state it refines.

Since all nodes are reachable from the initial node, the graph of reachable stacks can also be visited backward. Starting from any node and visiting predecessors until reaching the initial node produces a reachable stack in reverse.

6.1.2 Reachable Reduction Automaton

By intersecting the reduction automaton **RA** with reachable stacks **RS**, we obtain a non-deterministic automaton enumerating suffixes of reachable stacks and the sequence of reductions applicable to them. We write NRA for this construction but ultimately, we are interested in its determinisation **RRA**, the reachable reduction automaton.

A state is a pair $(r, v) \in \mathrm{NRA} \subseteq (\mathbf{RS} \cup \{\texttt{Initial}\}) \times \mathbf{RA}$, and `(Initial,Initial)` is the initial state. The transitions are of three forms:

1. `(Initial,Initial)` $\xrightarrow{s}$ $((s, T), v)$ for $s \in \mathbf{Wait}$, $(s, T) \in \mathbf{RS}$ and `Initial` $\xrightarrow{s}_{\mathbf{RA}} v$
2. $((s_1, T_1), v_1) \xrightarrow{s_2} ((s_2, T_2), v_2)$ iff there is a transition $(s_2, T_2) \rightarrow_{\mathbf{RS}} (s_1, T_1)$ in **RS** and a transition $v_1 \xrightarrow{s_2}_{\mathbf{RA}} v_2$ in the reduction automaton[1]
3. $((s_1, T_1), v_1) \xrightarrow{\epsilon} ((s_1, T_1), v_2)$ iff there is a transition $v_1 \xrightarrow{\epsilon}_{\mathbf{RA}} v_2$

[1] Or a transition $v_1 \twoheadrightarrow_{\mathbf{RA}} v_2$ when using the wildcard optimisation (see 2.4.1)

Since we are interested only in analysing the stacks of a waiting parser, we restrict the initial transitions to those reaching **Wait** states. The class of a wait state is always T: if $s \in \mathbf{Wait}$, we have $\mathbf{first}(s) = \{T\}$ by equations 5.4 and 5.6 because a wait state is either initial or has a terminal as incoming symbol. The other transitions are the transitions of **RA** restricted to reachable stacks.

We obtain $\mathbf{RRA} \subseteq \mathscr{P}(\text{NRA})$ by applying the subset construction to determinise NRA and we write $R_0 = \{(\texttt{Initial}, \texttt{Initial})\}$ for the initial state.

Simulated Suffixes Paths of the **RRA** starting from R_0 represent exactly the stack suffixes reachable by sequences of reductions starting from a waiting parser. Given a path:

- The labels annotating the transitions give the suffix as a sequence of LR state starting from the top of the stack.
- States in the path track the applicable reductions or the status of ongoing reductions.

A state $q \in \mathbf{RRA}$ is a set of NRA states, each of them tracking a different sequence of reductions. States of the form $(r, \texttt{Suffix}(s, \vec{s}', T'))$ are particularly interesting. They represent the configurations in which an LR parser has to decide which action to take, while the current state is determined to be $\text{top}(s\vec{s}')$ and the looking ahead symbol belongs to T'.

Definition 25 (Stack Suffixes) *We write* $\boldsymbol{suffixes} : \boldsymbol{RRA} \to \mathscr{P}(\boldsymbol{Suffixes})$ *for the suffix states in an* ***RRA*** *state:*

$$\begin{array}{rcl} \boldsymbol{suffixes}(q) & = & q \cap \boldsymbol{Suffixes} \\ \boldsymbol{Suffixes} & = & \left\{ n \in NRA \;\middle|\; \exists r, s, \vec{s}', T', n = (r, \texttt{Suffix}(s, \vec{s}', T')) \right\} \end{array}$$

Given $s \in \boldsymbol{Suffixes}$*, we define the projections* ***suffix*** *and* ***lookaheads****:*

$$\begin{array}{rcl} \boldsymbol{suffix}(r, \texttt{Suffix}(s, \vec{s}', T')) & = & s\vec{s} \\ \boldsymbol{lookaheads}(r, \texttt{Suffix}(s, \vec{s}', T')) & = & T' \end{array}$$

In non-initial NRA states, the first component r is a node of **RS** from which one can enumerate all stack prefixes that complete the suffix into a reachable stack.

Definition 26 (Stack Prefixes) *We write* $\boldsymbol{prefix} : NRA \rightharpoonup \boldsymbol{RS}$ *and* $\boldsymbol{prefixes} : \boldsymbol{RRA} \to \mathscr{P}(\boldsymbol{RS})$ *for the projection of* ***RS*** *nodes from NRA and* ***RRA*** *states, defined as:*

$$\begin{array}{rcl} \boldsymbol{prefix}(r, \_) & = & r \\ \boldsymbol{prefixes}(q) & = & \bigcup_{n \in q} \boldsymbol{prefix}(n) \end{array}$$

Connection to the reduction relation The **suffix** of a state q is made up of the last state consumed from the stack being recognised followed by the simulated suffix (the states pushed by following the goto transitions of the simulated reductions).

We can connect this to the reduction relation introduced in 2.2.2. If $(\alpha)\beta \downarrow \gamma$ then there is a reachable state $q \in$ **RRA** with a suffix $s \in$ **suffixes**(r) such that:

- there is a path p in **RS** reaching **prefix**(s) such that **incoming**$(p) = \alpha$
- there is a path p' in **RRA** from R_0 to q such that **incoming**$(p') = \beta$
- $\gamma =$ **incoming**(tail(**suffix**(s)))

The symbols **lookaheads**(s) are not directly connected to the reduction relation, but they tell us that the LR automaton realises the actions described (reducing a stack from $\alpha\beta$ to $\alpha\gamma$) when looking ahead at any symbol in **lookaheads**(s).

Connection to reduce-filter patterns A reduce-filter pattern $[\gamma / I]$ matches exactly the stacks with a suffix reaching the states $q \in$ **RRA** such that $\exists s \in$ **suffixes**(q) satisfying:

$$\gamma = \textbf{incoming}(\text{tail}(\textbf{suffix}(s))) \wedge I \subseteq \textbf{items}(\text{top}(\textbf{suffix}(s)))$$

6.1.3 Failing lookahead symbols

A failing configuration is a pair of a stack and a lookahead symbol that causes the LR parser to reject the input. The rejection might not happen immediately: the parser sometimes has to reduce a few productions before reaching a state that rejects the lookahead symbol.

Definition 27 (rejected lookahead symbols) *We write* ***reject***(s) *for the terminals that cause an LR parser in state s to reject, and generalise the definition to* ***RRA*** *states:*

$$\textbf{\textit{reject}}(q) = \bigcup_{s \in \textbf{\textit{suffixes}}(q)} \textbf{\textit{reject}}(top(\textbf{\textit{suffix}}(s))) \cap \textbf{\textit{lookaheads}}(s)$$

The rejections of an **RRA** state are defined by looking for any suffix reached in this state and intersecting the rejections of an LR parser with this suffix by the lookaheads that permitted reaching it.

Indirect rejections The lookahead symbols rejected when reaching state q are given by **reject**(q). However, for a given sentence, different sequences of reductions can be exercised depending on the lookahead token, possibly delaying the time at which a rejection is detected.

Consider a path $R_0 \xrightarrow{s_1} q_1 \xrightarrow{\cdots} q_i \xrightarrow{s_n} q_n$ in the reachable automaton. It captures the behaviours of LR parsers whose stack ends in $s_n \ldots s_1$. When looking ahead at a terminal $a \in T$, an LR parser might reject immediately, in which case $a \in$ **reject**(q_1), or engage in reducing some production, in which case the rejection is detected in

some state q_i, for $1 < i \leq n$. The lookahead symbols that are guaranteed to fail for this path are given by $\bigcup_i \textbf{reject}(q_i)$.

Now, let us look at what this means for a reduce-filter pattern. We observed that it can be interpreted as a subset of states $\bar{q} \subseteq \textbf{RRA}$. The lookahead symbols that can make the pattern fail are not only those immediately rejected, $\textbf{reject}(q)$ for $q \in \bar{q}$, but also all those rejected by paths containing a state in $\bar{q}$. We reason about these by collecting, for a state q, the lookahead symbols rejected on at least one path reaching q from the initial state, and those rejected on at least one path leaving q.

Definition 28 (reject-before and reject-after) *$\textbf{\textit{reject-before}}(q)$ and $\textbf{\textit{reject-after}}(q)$ are the least functions satisfying:*

$$\begin{aligned} \textbf{\textit{reject-before}}(q) &= \textbf{\textit{reject}}(q) \cup \bigcup_{q' \xrightarrow{s} q} \textbf{\textit{reject-before}}(q') \\ \textbf{\textit{reject-after}}(q) &= \textbf{\textit{reject}}(q) \cup \bigcup_{q \xrightarrow{s} q'} \textbf{\textit{reject-after}}(q') \end{aligned}$$

The failing lookahead symbols of state q are then $\textbf{reject-before}(q) \cup \textbf{reject-after}(q)$. For a reduce-filter pattern mapping to $\bar{q}$, they are $\bigcup_{q \in \bar{q}} \textbf{reject-before}(q) \cup \textbf{reject-after}(q)$.

Normalisation of rejected lookaheads Optimisations of the LR automaton, notably for compressing the transition table, are allowed to delay rejections, which can make **reject** and **reject-before** pessimistic. Consider a lookahead symbol not rejected on a path $R_0 \xrightarrow{\cdots} q_i$, but rejected on all possible continuations.

reject and **reject-before** are correct in that no failing lookahead symbol is missed, but some failing symbols might be detected later.

It is possible to detect these delayed rejections using the least solution to:

$$\text{delayed-reject}(q) = (T / \text{shift}(q)) \cap \bigcap_{q \xrightarrow{s} q'} \text{delayed-reject}(q')$$

where $\text{shift}(q)$ is the set of terminals shifted in the LR configuration described by q:

$$\text{shift}(q) = \bigcup_{s \in \textbf{suffixes}(q)} \left\{ a \in \textbf{lookaheads}(s) \;\middle|\; \exists s', \text{top}(\textbf{suffix}(s)) \xrightarrow{a}_S s' \right\}$$

A rejection is delayed if the lookahead is not shifted in the current state nor in any of the states reached by reducing.

When working with an automaton on which such optimisation has been applied, taking delayed-reject instead of **reject** marginally improves the output of static analyses by detecting rejection earlier in a path. This permits to produce shorter examples to illustrate a failing case to the end-user. In particular, using delayed-reject reverses the effects of the *default reduction* optimisation[2] on failing lookahead symbols.

[2]Used, for instance, by Menhir, Yacc and Bison.

6.1.4 Generating sentences

To report a failure to the end-user, it is nice to produce a sentence illustrating the problem.

Given a path in **RRA** and a lookahead symbol a rejected on this path, we want to produce an invalid sentence that will cause the LR parser to fail in a configuration described by this path and lookahead symbol. We start by constructing a matching LR stack, and then, using the accessibility information, turn this stack into a sentence ending in a.

The suffix of this stack is directly given by the labels $\vec{s}$ of the transitions of the path, in reverse. To complete $\vec{s}$, we look at the last state q of the path. The valid prefixes are the paths from the initial node of **RS** to any $r \in$ **prefixes**(q). To obtain the prefix $\vec{p}$ that expands into the shortest sentence, one can apply a shortest-path algorithm (Dijkstra, 1959), using the costs of the reachability matrices as weights.

The stack $\vec{p}\vec{s}$ can then be translated into a sentence prefix by replacing each state whose incoming transition is a nonterminal with the shortest sequence of states labelled by terminals it expands to, as determined by the cost matrices. Let's call this process *expand*.

The invalid sentence is then **incoming**$(map(expand, \vec{p}\vec{s})) \cdot a$.

6.2 Enumeration

The goal of enumeration is to produce a list of minimal invalid sentences that cover all pairs of a reduce-filter pattern and a failing lookahead symbol.

Let us illustrate with an arithmetic example. In grammar G_A, the sentences matching $[E/E \to T{+}E\cdot]$ can fail for lookahead symbols i, (, and). While i and (fail systematically,) depends on the context, as shown by these two sentences:

- 1 + 2 matches the pattern and fails with all three lookahead symbols.
- (1+2 also matches but fails only for i and (.

With this simple grammar, the sentence 1+2 alone is sufficient to cover all failing lookaheads of $[E/E \to T{+}E\cdot]$; with richer grammars, a single sentence per reduce-filter pattern rarely suffices. However, the second example also covers the pattern $[E/(E \to)\cdot]$: while a single sentence is unlikely to cover all failing lookaheads of a pattern, it usually covers some lookaheads of multiple patterns.

In light of this example, we can see the production of a small list of sentences covering all failing lookaheads of all patterns as an optimisation problem—finding sentences that cover as many cases as possible or focusing on cases that are difficult to cover first, while balancing the length and number of sentences. Producing a minimal list of sentences seems computationally hard, but we propose a greedy approach which has proved sufficient in our tests.

Formally, we aim to produce a list such that, for all terminals a and reduce-filter patterns $[\alpha/I]$ that fail when looking ahead at a, there is a minimal invalid sentence ua such that the prefix u matches the pattern.

Working state by state A reduce-filter pattern translates to a set of **RRA** states; if we cover all **RRA** states and failing lookahead symbols, all reduce-filter patterns will be covered too. We consider each q individually, producing enough sentences to cover failing lookahead symbols, given **reject-before**$(q) \cup$ **reject-after**(q).

Let us start by handling symbols in **reject-before**(q). Each of them is rejected by a state q', possibly q itself, on a path from R_0 to q. We generate an invalid sentence from this path and lookahead symbol, following the procedure from Section 6.1.4.

We still need to cover the lookahead symbols in **reject-after**(q)/**reject**(q). By definition, for each symbol there is a state q', reachable from q, rejecting this symbol. We pick an arbitrary path from R_0 to q, complete it with a path from q to q', and again generate an invalid sentence from this path and lookahead.

Shortest paths We said that, given a source and target state, we picked arbitrary paths. In practice, one wants paths minimising the length of the sentences. These are not necessarily the shortest paths because we have to account for nonterminals expanding into sequences of terminals. As in Section 6.1.4, those can be found by a shortest-path algorithm using the cost matrices to weight the edges.

Filtering redundant sentences The set of sentences we just described cover all failing lookaheads for all **RRA** states, but can be redundant in two ways:

1. Some cases are likely to be covered by multiple sentences, such that some sentences can be removed without affecting coverage.

2. **RRA** states are too fine-grained; we were looking for reduce-filter patterns to begin with.

The reduce-filter patterns form a partial order such that $[\gamma / I] \subseteq [\gamma / I']$ when $I' \subseteq I$. In other words, when more items are required, fewer situations match the reduce-filter pattern. The minimal elements that recognise at least one sentence are exactly the $[$**incoming**(tail(**suffix**(s)))/**items**(top(**suffix**(s)))$]$, for $s \in$ **Suffixes**. Let us call this set *Targets*:

- By mapping **incoming** on the suffix, it identifies the sequence of reductions producing the same sequence of nonterminals.

- By keeping only the **items** of the last state, it identifies all LR(1) states that come from the same LR(0) state.

Each sentence covers one lookahead symbol for one or more *Targets*, found by looking at the states in the **RRA** path used to generate the sentence.

To filter the sentences, we maintain a finite map from *Targets* to $\mathscr{P}(T)$. Initially, all sets of terminals are empty. Before outputting a sentence with failing lookahead symbol a, we add a in the finite map for all *Targets* reached by the sentence. If they were already all covered, we omit the sentence.

Minimality of the solution Filtering sentences is an instance of the *set cover problem*, which is NP-hard. Our solution is very naive and depends entirely on the order in which the candidate sentences are visited but has proved sufficient for our test cases. Cormen et al. (2009) gives a better and simple greedy algorithm. Better solutions have been deeply researched by the optimisation community (Korte and Vygen, 2012).

But selecting a minimal subset is not sufficient to guarantee an optimal enumeration, as the generation procedure also affects the quality of the solution. There are multiple ways to improve the set of candidates. By picking the shortest paths to **reject-before** and **reject-after** lookahead symbols, the set favours having more shorter sentences over possibly fewer longer sentences. When multiple shortest paths exist, certain might lead to quicker coverage than others. Finally, we generated one sentence per lookahead, but it can be more efficient and pleasant for the end user to pair a sentence prefix with all the lookaheads that cause it to fail.

6.3 Coverage

The coverage checker finds the parts of the grammar that are not covered by an error message. The analysis applies to a deterministic automaton (Q, S, d, q_0, F) that realises an error specification for an LR automaton with initial state $s_0 \in S$, as produced in Chapter 4. Coverage can be checked immediately after determinisation, exploration or minimisation. It is done by looking for fallible stack suffixes that are not recognised by the automaton Q. A state is final if it has an accepting continuation: $q \in F \Leftrightarrow \exists i, j, q_i = \blacksquare_j$.

We write $q \xrightarrow{s}_Q q'$ iff $d(q, s) = q'$.

6.3.1 Fallible stacks

We start by constructing a non-deterministic automaton **FS**, for fallible stacks, that recognises the reachable stacks that can fail. They generally coincide with the reachable stacks, except if in some configurations, the LR automaton rejects none of the terminals.

The states **FS** are pairs in $(\mathbf{RS} \uplus \mathbf{RRA}) \times \mathscr{P}(T)$. The first component is the disjoint union of the reachable stacks and the reachable reduction automaton; the second component tracks the rejected lookahead symbols. Tracking the rejected lookahead symbols is not strictly necessary for checking coverage but is convenient to inform the user, when an uncovered sentence has been found, of the lookaheads that cause it to fail.

The initial state is $((\texttt{Initial}, \texttt{Initial}), \emptyset)$, representing a configuration in the initial state of **RRA** when no lookahead symbol has been rejected yet. For each transition $(\texttt{Initial}, \texttt{Initial}) \xrightarrow{s}_{\mathbf{RRA}} r$ such that $\mathbf{reject\text{-}after}(r) \neq \emptyset$, there is a transition:

$$((\texttt{Initial}, \texttt{Initial}), \emptyset) \xrightarrow{s}_{\mathbf{FS}} (r, \mathbf{reject}(r)).$$

The transitions for the other states $r \in RRA$ are as follows:

- If there are no transitions leaving r in RRA, e.g. $\nexists s, r', r \xrightarrow{s}_{RRA} r'$, then for each $p \in \textbf{prefixes}(r)$ there is a transition:

$$(r, T') \xrightarrow{\epsilon}_{\textbf{FS}} (p, T')$$

  This handles the case of reaching the end of the sequence of reductions applicable to a stack suffix by switching to enumerating all prefixes of reachable stacks ending in p.

- Otherwise for each transition $r \xrightarrow{s}_{\textbf{RRA}} r'$ there is a transition:

$$(r, T') \xrightarrow{s}_{\textbf{FS}} (r', T' \cup \textbf{reject}(r'))$$

  if $T' \neq \emptyset$ or $\textbf{reject-after}(r') \neq \emptyset$. That is, either we have found lookaheads that are rejected or there are some lookaheads that can still be rejected.

For prefixes $p \in \textbf{RS}$, the transitions are reversed: if $p' \rightarrow_{\textbf{RS}} p$, then:

$$(p, T') \xrightarrow{\pi_1(p')}_{\textbf{FS}} (p', T')$$

This edge is labelled by $\pi_1(p')$, which is the state at the top of the stacks described by p' in **RS**.

Paths in **FS** follow reductions that can fail, accumulate the rejected lookaheads and switch to enumerating stack prefixes at the end of reduction sequences.

6.3.2 Intersecting with fallible stacks

Coverage is determined by intersecting Q and **FS**, looking for transitions that are defined only in **FS**.

We construct the intersection as the least graph with states $I \subseteq \textbf{FS} \times Q$ such that $i_0 = (((\texttt{Initial}, \texttt{Initial}), \emptyset), q_0) \in I$ and for each $(f, q) \in I$:

- If q is final, there are no transitions; we have reached an accepting state of Q, the configuration described by f is covered on this path.
- If q is not final and $f \xrightarrow{\epsilon}_{\textbf{FS}} f'$, then $(f', q) \in I$ and $(f, q) \xrightarrow{\epsilon}_I (f', q)$
- If q is not final and $f \xrightarrow{s}_{\textbf{FS}} f'$:
  - if $\nexists q', q \xrightarrow{s}_Q q'$ or, if $\exists q', q \xrightarrow{s}_Q q'$, $s = s_0 \wedge q' \notin F$, we have found a transition from f labelled s that is not covered by q
  - otherwise, $(f', q') \in I$ and $(f, q) \xrightarrow{s}_I (f', q')$ where $q \xrightarrow{s}_Q q'$

The two cases for non-covered transitions are either if the transition is missing, $\nexists q', q \xrightarrow{s}_Q q'$, or if the transition exists but is labelled by the initial LR state s_0 and reaches a non-final state, $q' \notin F$. The initial state is the bottom of the LR stack, so the scan ends here, but no message has been found yet. This situation can happen when the error automaton Q is built with a sink state, consuming all states until the end of

the stack without accepting, or as a by-product of optimisations for compacting the automaton (e.g. minimisation of wildcard transitions).

When discovering an uncovered transition $f \xrightarrow{s}_{\mathbf{FS}} f'$ from a state (f, q), we can construct an example to report to the user. The labels along the path from i_0 to (f, q) give a stack suffix, which can be completed by generating a prefix from $\pi_1(f')$ (the target of the transition), following the same recipe as in Section 6.1.4.

The lookahead symbols that caused the suffix to fail are given by $\pi_2(f)$, which can be empty. In this case, $\pi_1(f) \in RRA$ and the lookaheads potentially causing failure are **reject-after**$(\pi_1(f))$. The suffix is too short to determine the exact failure. By following outgoing transitions of f', it can be completed to a more specific suffix guaranteed to fail.

We can also report the reduce-filter patterns that would solve the issue to the user. For this we look for the suffix states in **suffixes**$(\pi_1(f'))$ or **suffixes**$(\pi_1(f''))$ for all (f'', q') on paths from i_0 to (f, q).

A suffix s can be turned into a reduce-filter pattern:

$$[\,\textbf{incoming}(\text{tail}(\textbf{suffix}(s)))\ /\ \textbf{items}(\text{top}(\textbf{suffix}(s)))\,]$$

If the reduce-filter pattern appears on all paths reaching (f, q), handling it is guaranteed to solve the issue. If it appears only on some paths, only some of the cases will be covered. A more sophisticated reporting could suggest examples specialised for the different cases.

Dead clauses and patterns The analysis can also be used to find unreachable patterns and clauses. This is done by exploring I and looking for clauses that are accepted on no paths. Reporting unreachable clauses proves very valuable in keeping an error specification synchronised with an evolving grammar.

A refinement of the approach is to also report sub-patterns that do not contribute to any new case being covered. This happens particularly with branches, e.g. $(e_1|e_2)$ when all cases matching e_2 are covered by a clause of higher priority. Our implementation tracks symbol patterns x and reductions as the unit "of contribution": a symbol pattern or a reduction that do not contribute is reported.

6.4 Conclusion

In Chapter 2, we saw that paths in the LR automaton could be used to enumerate LR stacks and derived the viable reduction automaton to reason about possible sequences of reductions. Unfortunately, this enumeration is an over-approximation if the actions of the LR automaton were truncated by conflict resolution.

Chapter 5 developed an efficient strategy for characterising the impact of truncation on the reachability of LR configurations. This chapter uses that result to enumerate failing configurations and suggests two applications:

- **Enumeration**: Produces a set of invalid sentences illustrating the ways in which grammar constructions can be misused.

- **Coverage**: Takes an error specification and finds invalid sentences that lack an error message.

Although we have not proven this result, both methods always terminate and are sound and complete. They do not report failures that are unreachable and do not miss any reachable failures. The enumeration reports all the reduce-filter patterns that can be covered. If coverage analysis does not find any uncovered sentences, the error specification covers all possible errors.

Chapter 7

The LRGrep implementation

7.1 From the calculus to an implementation

In this chapter, we explore the practical implementation of the calculus, addressing real-world scenarios that benefit from extensions of our matching language. We introduce features like constraints on lookahead symbols and user-provided predicates, which were not covered in the theoretical framework. We then present the tools provided by the implementation of LRGrep and the concrete syntax of LRGrep files.

7.1.1 Predicated clauses

In the calculus, the applicability of a clause is entirely determined by the pattern. In practice, we found two situations in which it is desirable to refine this behaviour: conditioning clause application on certain lookahead terminals or on a user-provided predicate. We did not introduce these subtleties earlier as they do not directly affect the theory we developed.

The general syntax for clauses has the form:

$$\begin{array}{rcl} \text{clause} & ::= & \text{pattern} \quad (@\ \text{lookaheads}^*)^? \quad \texttt{\%partial}^? \quad \{\,\text{action}\,\} \\ \text{lookaheads} & ::= & a \mid \text{first}(A) \end{array}$$

The two fixed parts of the clause are still the pattern and the semantic action, but they can be complemented with an optional *lookahead specification* and an optional `%partial` qualifier.

The lookahead specification is a set of terminals. If present, it restricts the clause to apply only when the parser fails while looking ahead at one of those terminals. Terminals are specified directly by giving their names or by using the syntax $\text{first}(A)$ which denotes the set of all terminals that can appear at the beginning of a word recognised by a nonterminal A:

$$\text{first}(A) = \{a \in T \mid \exists w,\ a \cdot w \in \mathscr{L}(A)\}$$

If the `%partial` qualifier is present, the semantic action is allowed to give up. It evaluates to an option. If this optional result is `Some`(v), it behaves like a normal

action evaluating to v. If it is `None`, the clause does not apply and matching resumes, looking for the next candidate.

We say that a clause is total if it is not `%partial` and has no lookahead specification; otherwise, it is non-total.

Impact on LRGrep's pipeline This change affects two parts of the compiler. First, when reaching an accepting state for a non-total clause, we no longer prune the continuations with lesser-priority because these might contribute reachable candidates if the clause does not succeed. Second, an accepting state can have more than one clause associated with it. Previously, if multiple clauses succeeded at the same time, only the one with the highest priority was retained. Now, the total clause with the highest priority and all the non-total ones with higher priorities are retained.

During execution, an LRGrep-automaton produces a list of candidates from the stack, at most one per clause. The candidates are ranked according to the clause priority and tried in order until a successful candidate is found.

Impact on coverage analysis

The main impact is on the coverage analysis. Partial clauses can simply be ignored by the analysis. We do not make any assumptions about the semantic action, so the conservative approximation is that a partial clause always gives up and does not contribute to coverage.

However, the analysis can be refined to account for lookahead specifications. In the original definition of coverage analysis, accepting states were provided as a subset $F \subseteq Q$. Traversal stopped when reaching an accepting state $q \in F$.

Acceptance is now modelled as a function $\mathit{accepted} : Q \to \mathscr{P}(T)$, indicating the lookahead terminals accepted by this state. $\mathit{accepted}(q)$ is defined as the union of the lookahead specifications of all clauses accepted in state q; a total clause handles all terminals.

The construction of the intersection I is augmented to accumulate the handled terminals: $I \subseteq \mathbf{FS} \times Q \times \mathscr{P}(T)$. A state is now a triple (f, q, h); in the initial state, h is $\mathit{accepted}(q_0)$. A transition $(f, q, h) \xrightarrow{s} (f', q', h')$ is such that $h' = h \cup \mathit{accepted}(q')$.

Transitions are followed until reaching a state (f, q, h) such that:

$$\textbf{reject-after}(\pi_1(f)) \cup \pi_2(f) \subseteq h$$

reject-after$(\pi_1(f))$ lists the lookahead symbols that can still fail in some transitive successors, while $\pi_2(f)$ are the lookahead symbols that have already failed. If all of those are in h, all possible failures in this suffix have been handled.

Similarly, when reporting a non-covered transition, the failing lookahead symbols are $(\pi_2(f) \cup \textbf{reject-after}(\pi_1(f))) \setminus h$.

7.1.2 Multiple entry points

The work we presented assumes a grammar with a single start symbol and an automaton with a single initial state. In practice, parser generators support more than

one entry point, to define multiple languages that share common definitions.

For instance, ML languages have interface and implementation files which are recognised by two different entry points but otherwise share common constructions. Some languages have a batch compiler and an interactive interpreter which recognise slightly different dialects.

A grammar author might want to provide different error messages for the different cases. The analyses we introduced can easily be adapted to work on specific entry points by applying them on the subset of the automaton reachable from the corresponding initial states. The wait states, the enumeration of viable stacks, and therefore the reduction automata and the reachable stacks directly depend on the set of states considered.

7.1.3 Integration with OCaml

The reference implementation of LRGrep works with the OCaml programming language; therefore, the pieces of code are expected to be valid OCaml code:

- Identifiers in an LRGrep specification should follow OCaml lexical rules.
- The specification can contain initialisation code which is interpreted as a piece of an OCaml structure.
- Semantic actions should be valid OCaml expressions.

The semantic actions of all clauses should have the same type—wrapped in an option for partial actions. In the generated code, LRGrep uses pattern matching to extract information from the stack and paste the code of semantic actions in the branches.

Portability across languages and parser generators

The compilation process is parameterised by the LR automaton being analysed, but is otherwise similar to the generation of a lexer from a specification: a specialised compiler takes the description and translates it into a program in a general-purpose language.

The reference implementation consumes the LR automata produced by Menhir, the reference LR parser generator for OCaml. Besides that, there is nothing specific to the OCaml language in LRGrep's design. The compiler can, in principle, be retargeted to other programming languages and parser generators. The only assumptions are that the automaton is an LR(1) automaton, and that, at runtime, the parser stack can be accessed undamaged when the input is rejected.

By undamaged, we mean that the parser stack should be in the state it had after the most recent shift transition. We expect this to be the main difficulty when porting LRGrep to another parser generator. High-performance LR implementations make a destructive use of the stack, and spurious reductions are a natural consequence of a non-canonical (compressed) automaton. This is unfortunately at odds with our

approach for producing good error messages: to maximise sharing, the compression process throws away information prior to detecting an error. This information is irrelevant when processing valid inputs but can be valuable for assessing the situation when parsing fails.

Finding implementation strategies suitable for high-performance implementations and evaluating their impact has been relegated to future work; see 9.5. Meanwhile, our reference relies on Menhir's support for taking persistent snapshots of a stack to undo undesirable reductions after a parse error has been detected.

Working with Menhir extensions to LR grammars

Menhir provides two unusual features to help engineer an LR grammar: higher-order and inlined nonterminals.

Higher-order nonterminals allow writing generic grammatical definitions by abstracting over symbols (including other higher-order nonterminals). This behaves like a C++ template or macro, where a definition contains named holes that are substituted by concrete definitions when instantiated. Like C++ templates, the higher-order nonterminals are compiled by monomorphisation and each instantiation is given a unique name derived from the actual parameters.

Each occurrence of an inlined nonterminal in a rule is substituted by the different productions of the nonterminal.

Both features are handled in a pre-processing step by rewriting the grammar to a plain, first-order, LR grammar. Since LRGrep receives the automaton after it has been processed by Menhir, it has no knowledge of the higher-order and of the inlined nonterminals. For higher-order nonterminals, it sees the monomorphised instances, and treats each of them as a distinct nonterminal. The only change is that, syntactically, a symbol name can look like a function application. For instance `list(definitions)` is the result of instantiating `list`, a higher-order nonterminal, at `definitions`, a plain nonterminal. LRGrep treats it like a normal nonterminal, as if the grammar author had manually written a `list_definitions` nonterminal. Inlined nonterminals have completely disappeared from the automaton. To write error rules matching situations involving an inline nonterminal, the grammar author should do the substitution themselves by matching for each expansion of the nonterminal.

7.1.4 Support for longest-match operators

We observed that allowing extraction of values from the stack makes matching ambiguous. We are no longer asking only whether a certain pattern matches, but also *how* it matches. To solve this problem, we introduced disambiguation rules, imposing an order on the different derivations of the same input.

For the iteration operators, e^* and $[e]$, the semantics we gave favour the shortest match: the minimum number of iterations for e^*, and the sequence of reductions consuming the shortest suffix of the stack for $[e]$. The implementation features longest-match variants written e^{**} and $[[e]]$. They can be implemented with minor changes:

- Relying on the disambiguation rules for |, e^{**} expands to $e^{**} \cdot e|\epsilon$ (whereas e^* expands to $\epsilon|e^* \cdot e$).
- By replacing $d_C(k,s) \odot k'$ by $k' \odot d_C(k,s)$ in the derivation of the reduction operator (Figure 3.4, page 72). In this line, $d_C(k,s)$ is the continuation to derive when a match has been identified, while k' continues to look for a longer reduction sequence. By putting k' first, we give a higher priority to the branch looking for a longer suffix than to the success continuation.

Longest variants do not increase the expressiveness of the language. They are offered as an option to the grammar author to refine the positions reported in an error message. We can give an artificial example on G_A to illustrate the case. Let us assume a situation in which one wants to match and report the position of a factor F in the input $1+2*3*4$. With pattern $p = [F]$, the position reported covers only 4, as it is the shortest suffix that reduces to a factor; with pattern $[[F]]$ the position reported covers the whole product $2*3*4$.

7.1.5 Practical usage of variable bindings

The subsequence bound by a variable binding $n = e$ is not directly exposed to the user. We expect the user to be interested in the semantic values and the source code locations rather than stack offset.

Following Menhir's conventions, we provide special syntaxes `$startpos(`n`)` and `$endpos(`n`)` to extract the starting and ending position of the sequence that matched. If e covers a single symbol (e.g., $n = x$), its semantic value is bound to n.

This covers the two use cases where we found variable bindings to be useful: referring to a source code location and accessing a semantic value. We did not find any reason to allow direct access to the stack, though this can be done in principle.

For instance, a concrete implementation for an unclosed parenthesis looks like:

```
| l=LPAREN; [expr /LPAREN expr . RPAREN] {
  printf "Expecting ')' to close the parenthesis opened line %d column %d"
    (line $startpos(l)) (column $startpos(l))
}
```

Reconstructing types To decide at which type a variable should be exposed, we have to consider two pieces of information:

1. The type of the semantic values that can be bound to this variable, by looking at the transitions binding it.
2. Whether it is guaranteed to be bound when reaching the semantic action.

For a simple pattern, like $n =$ `expr`, we know that n is always captured when matching an `expr` and that there is no way for the pattern to match without binding n. In the semantic action, n can be given the type of the semantic values of nonterminal `expr`.

More problematic patterns are n_1 = `expr`|n_1 = `pattern` or n_2 = `expr`|ϵ. For the first one, we fall back to an opaque type representing a Menhir stack frame, because n_1 could be bound either to an `expr` or a pattern `pattern`. For the second one, we keep the type of `expr` but wrap it in an option.

To solve 1., we look at all the transitions where a variable is bound. If all the transitions that bind the variable are labelled with the same symbol, we give the variable the type of the semantic values of this symbol. Otherwise, we use the generic stack frame type. For expert users, a few functions are provided to extract information from this representation, though we did not find practical cases where this was necessary.

For 2., we check if the variable is bound on all succeeding paths. If so, we directly expose the value at the type we have determined in 1. If not, then there is a path from the initial state to the accepting state which does not go through a transition binding the variable. In this case, we wrap the type in an option.

Runtime representation During matching, semantic values are stored in registers. The mapping from a register to a typed variable is done just before executing a semantic action. A register has the most general type: an optional stack frame.

When generating code, we prepend a small prologue to the semantic actions that extracts the precise representation from the generic one. If we determined that the variable is bound on all paths, we can unwrap the option (it cannot be `None`: if this is the case we abort the computation and signal an internal compiler error). If we found a more precise type to give to the semantic value, we check that the LR(1) state of the stack frame matches our expectations, just to be safe, and then cast the semantic value to the expected type.

More precise types? We could leverage the OCaml type system to degrade the types more gracefully than directly falling back to the generic stack frame type. For instance, n_1 could be given the type `[`Expr of ty_expr | `Pattern of ty_pattern]`. However, we did not encounter situations where this was useful in practice, or could not be worked around.

For this example, we suggest rewriting the pattern as *ne* = `expr`|*np* = `pattern` which binds the variables *ne*: `ty_expr option` and *np*: `ty_pattern option`.

7.2 Concrete syntax of LRGrep files

An LRGrep error specification is written in a `.mlyl` file. Even though the syntax of patterns almost directly follows from the calculus from Chapter 3, the implementation has to handle practical details that we did not consider so far, and also features syntactic sugars to ease writing. We review these differences now and introduce the surface syntax of `.mlyl` files.

Figure 7.1 gives the structure of an `.mlyl` file. It is a sequence of rules with optional heading and trailing blocks of OCaml code wrapped between curly brackets.

```
mlyl:
  header
  rule*
  trailer

header: action?
trailer: action?

action: "{" OCaml code ... "}"
```

Figure 7.1: Structure of an `.mlyl` file

```
rule: "rule" ident ident* "="
        "parse" "error"? entrypoints? clause*

entrypoints: "(" symbol ( "," symbol )* ")"

ident: ["a"-"z" "A"-"Z" "_"]
       ["a"-"z" "A"-"Z" "0"-"9" "_" "'"]*

symbol: ident
      | ident "(" (symbol ("," symbol)* )? ")"
```

Figure 7.2: Syntax of LRGrep rules

The heading and trailing code, if present, should be syntactically valid OCaml structures.

Rules The syntax of rules is given in Figure 7.2. A rule starts with the keyword `rule`, followed by a list of identifiers: the mandatory name of the rule and zero or more formal parameters, delimited by `= parse`. Identifiers follow OCaml lexical conventions. The `error` keyword specifies that the rule is meant to apply only to failing parsers, which is the intended use case for LRGrep. It is nonetheless optional: if the keyword `error` is omitted, LRGrep disables the optimisations that make assumptions on the configuration of parsers being analysed. The rule can also be restricted to only certain entry points by specifying them between parentheses; LRGrep makes use of this information to prune the automaton more aggressively and to prevent the coverage checker from reporting missing cases reachable only from other entry points. After that comes the list of clauses.

Grammatical symbols follow Menhir's conventions:

- A terminal starts with an upper-case letter, like `INT`.
- A nonterminal starts with a lower-case letter, like `expr`.
- Instances of higher-order nonterminals use function application notation; they

```
clause: ("|" pattern lookahead_constraints?)+
        (action | "%partial" action | "{" "." "}")

lookahead_constraints: "@" lookahead ("," lookahead)*

lookahead: terminal | "first" "(" nonterminal ")"
```

Figure 7.3: Syntax of clauses

start with a nonterminal name followed by a comma-separated list of symbols between parentheses. E.g., `list(INT)`.

Although in the majority of cases a single rule covering all error messages is sufficient, multiple rules can often be useful to factor information extraction. For instance, syntax error messages of the reference Elm compiler (see Section 8.3) are sometimes split into two levels of analyses:

- The first, finer-grained, level focuses on the grammatical construction that failed to parse, such as the current expression or pattern.
- The second level locates the definition in which the failure occurred in the current module (e.g., "While parsing function `foo`" or "Inside type `bar`")

To reproduce this behaviour it can be convenient to split the analysis into two rules, one for extracting the current definition and analysing the local context.

Clauses Figure 7.3 gives the syntax of clauses. A clause is a "|" delimited list of one or more patterns with optional lookahead constraints ending with an action.
There are three kinds of action:

- **Total action**: A total action is a piece of OCaml code wrapped between curly brackets that is executed when the clause matches. The rule ends by returning the value to which the OCaml code evaluates. This is the most common kind of action.
- **Partial action**: Sometimes, semantic information is needed to decide whether a clause applies or not. LRgrep provides partial actions, specified with the `%partial` keyword for this case. In a partial action, the OCaml code should evaluate to an optional value. When it is `None`, matching resumes, trying to recognise a clause with lower priority. When it is `Some v`, matching ends, as in the total case, returning `v`.
- **Unreachable action**: The syntax `{ . }` is provided to ensure that a clause never matches. LRGrep checks that this is the case at compile-time. This is useful for guaranteeing that certain patterns are already covered by clauses with higher priority.

```
base:
  | symbol                 — a specific symbol
  | "_"                    — any symbol
  | "[" pattern "]"        — reduction to the shortest stack suffix
  | "[[" pattern "]]"      — reduction to the longest stack suffix
```

Figure 7.4: Base patterns, can be bound to variable

The lookahead constraints restrict the patterns preceding `@` to match only for certain lookahead terminals. The terminals are specified as a comma-separated list of names or using the special syntax `first(nt)`.

Patterns The pattern language is decomposed into two syntactic categories: the "base" patterns, listed in Figure 7.4, which can be bound to variables, and the other operators from Figure 7.5.

A base pattern can be a single grammatical symbol, a wildcard _ that matches any symbol or an application of the reduction operator. The rest of the pattern language contains filters and the usual regular operators: sequence, disjunction, repetition (Kleene-star), a base pattern that can be bound to a variable, and ϵ, which consumes nothing.

Filters The base syntax for a filter is a single item but a few shortcuts are provided for convenience. The shortcuts specify a set of items using a syntax inspired by glob patterns, which are interpreted as a disjunction of filters for each item.

To understand the set of items denoted by the generalised syntax of filters, it is simpler to start from the set of all items and keep only those satisfying a series of constraints:

1. An optional left-hand side symbol. If provided, only items with this left-hand side are kept.

2. A right-hand side specified as a sequence of symbols, wildcards "_" acting as placeholders for any symbol, and wildcard sequences "_*" for any sequence of symbols. Only the items with a matching right-hand side are kept.

3. One or more dots `"."`, representing the different positions in which a dot is allowed in the items.
   E.g. `. a . b .` is interpreted as `. a b` or `a . b` or `a b .`.

For instance, a filter `/_* . expr . _*` matches any item with a dot before or after an `expr`, and no constraints on what comes after or before.

```
pattern:
  |                          — empty pattern, always matching
  | "/" filter               — item filtering
  | (ident "=")? base        — binding a variable
  | pattern "*"              — shortest repetition
  | pattern "**"             — longest repetition
  | pattern "?"              — optional pattern
  | pattern ";" pattern      — pattern sequence
  | pattern "|" pattern      — pattern disjunction
  | "(" pattern ")"          — parentheses for disambiguation
filter: ( symbol ":" )?  ( "." | "_" | "_*" | symbol )*
```

Figure 7.5: Operators for composing patterns

7.3 The LRGrep tool

LRGrep is distributed as an OCaml package which provides an `lrgrep` library and an `lrgrep.runtime` library. `lrgrep` receives a Menhir automaton as an argument and offers three sub-commands:

`compile` takes an error specification and produces an OCaml module, and if opted-in, checks for exhaustive coverage of errors.

`enumerate` emits a list of sentences covering all reduce-filter patterns.

`interpret` reads an invalid sentence and produces an annotated stack trace detailing the parser's configuration at the point of failure.

The Menhir automaton is represented by a `.cmly` file produced by Menhir when compiling a grammar with the `--cmly` flag. The `compile` command is intended to be invoked by a build system as part of a larger compilation process, though the coverage feature can be used on its own when a developer is working on error messages or during continuous integration to ensure that error messages are synchronised with the grammar. The `enumerate` and `interpret` commands are provided as tools to help authoring error messages.

7.3.1 Compiling an LRGrep specification

Integration with Dune The standard build system for OCaml is Dune. It reads its instructions from `dune` files that are present in all directories containing OCaml sources.

Figure 7.6 gives a minimal `dune` file for compiling an LRGrep specification. Dune knows about OCamllex and Menhir, so it has built-in directives to automate most of the work. `(ocamllex (modules lexer))` instructs Dune to generate a `lexer.ml` module from the `lexer.mll` specification using OCamllex.

```
(ocamllex
  (modules lexer))
(menhir
  (modules parser)
  (flags :standard --inspection --table --cmly))
(rule
  (targets errors.ml)
  (deps parser.cmly errors.mlyl)
  (action
     (run lrgrep errors.mlyl -g parser.cmly -o errors.ml)))
(executable
 (name main)
 (libraries menhirLib lrgrep.runtime))
```

Figure 7.6: Compiling an LRGrep specification with Dune

`(menhir (modules parser))` does the same to generate a `parser.ml` module from the `parser.mly` grammar.

We have to pass extra flags to Menhir to integrate with LRGrep:

`--table` produces a parser that uses the table-based interpreter rather than the native code backend; this is required to hook LRGrep's failure handling code.

`--inspection` produces detailed definitions necessary to interpret the contents of the stack.

`--cmly` saves the automaton in the `parser.cmly` file.

We need to give detailed instructions for building LRGrep specifications. The custom rule states that invoking `lrgrep compile errors.mlyl -g parser.cmly -o errors.ml` produces `errors.ml` using the `parser.cmly` and `errors.mlyl` files.

The last directive builds a `main` executable linked to Menhir's and LRGrep's runtime libraries.

Integration with a Menhir parser Now that we have an `Error` module, it must handle the failures of the parser. This is done by using a custom interpreter for the Menhir parser, provided by LRGrep.

A normal invocation of Menhir to parse the contents of a buffer `lexbuf` using the `main` entry point with lexer `Lexer.token` is written:

```
Parser.main Lexer.token lexbuf
```

A successful parse returns the semantic value produced by `main`, while an invalid input causes the parser to raise an exception. To let LRGrep handle errors using a rule named `error_message`, one replaces the invocation with:

```
Errors.parse_with_error_message
  Parser.Incremental.main Lexer.token lexbuf
```

This tries to parse the input like before, but the result is now wrapped in a result type:

- A successful parse returns `Ok r` where `r` is the semantic value produced by `main`.
- When provided with an invalid input, the parser's configuration is analysed according to the error specification and `Error (Some e)` is returned if a clause applied or `Error None` if there was no handler for the error.

7.3.2 Understanding a failure with the interpreter

The interpreter can guide the creation of a new error clause from a sample input. It parses the input and prints a detailed trace of the configuration of the LR parser at the end of the input or at the first error.

Let us illustrate the workflow using an unclosed parenthesis for arithmetic as an example. We start the interpreter by providing the Menhir automaton, and we pass the argument - to read the sample on standard input:

```
$ lrgrep interpret -g calc.cmly -
main LPAREN INT PLUS INT
```

The input starts with the name of the entry point followed by the tokens. The interpreter has no knowledge of the lexical conventions of the language being analysed, so the input is provided using the terminal names. This input is a valid prefix and is consumed until the end; the interpreter then answers:

```
Parser stack (most recent first):
    [ / expr: INT .]
- line 1:21-24 INT
   ↱ expr
     [expr / expr: expr . DIV expr
           / expr: expr . TIMES expr
           / expr: expr . MINUS expr
           / expr: expr PLUS expr .
           / expr: expr . PLUS expr]
- line 1:16-20 PLUS
- line 1:5-15 expr
   ↱ expr
     [expr / expr: expr . DIV expr
           / expr: expr . TIMES expr
           / expr: expr . MINUS expr
           / expr: expr . PLUS expr
           / expr: LPAREN expr . RPAREN]
- line 1:5-11 LPAREN
- line 1:0-4 main
```

We see that it is possible to reduce the input to an expression in two different ways:

- Immediately after reading the `INT` at the top of the stack (reducing `expr: INT`)
- After reading `INT`, `PLUS`, and `expr` (further reducing `expr PLUS expr`)

We are interested in the second one, and we can use the reduce-filter patterns printer by the interpreter to match this specific reduction. To get a non-ambiguous match it is sufficient to filter for `expr : LPAREN expr . RPAREN`, the item with the longest prefix.

We can directly use `[expr / expr: LPAREN expr . RPAREN]`, or bind the preceding symbol if we want to refer to the location of the opening parenthesis:

```
lp=LPAREN; [expr / expr: LPAREN expr . RPAREN]
```

7.3.3 Working with enumerated sentences

Enumeration works by listing sentences that cover the different ways a parser can fail. Of course, there is generally an infinite number of invalid sentences, so we aim to cover the reduce-filter patterns and the lookaheads that can make them fail. This is a finite set that we can exhaustively cover, even though it can require a large number of sentences. We use the strategy described in Section 6.2 to produce reduce the number of sentences possible, grouping each sentence with all the lookaheads that can make it fail.

By limiting the coverage to reduce-filter patterns, many invalid sentences are considered equivalent. For instance:

- The exact path that led to a failure is not considered. For instance, with OCaml grammar, `let x = 5 when` and `let x = 5 let y = 6 when` are considered equivalent. The error is the keyword `when`, which is not allowed in an expression. The fact that the second example is preceded by another definition is ignored.
- Two sentences going through the same reductions are considered equivalent, even if the number of reduction differ; for instance `x * 2 if` and `x * 2 * 3 if` are considered equivalent.

These equivalences reduce the amount of noise with only a slight loss in precision. In practice, the sentences produced by enumeration are sufficient for covering the most common syntax errors. It is helpful to get an idea of the possible failures of a grammar and to start writing an error specification.

The calc example We illustrate the use of enumeration by examining the failures of `calc`. It is the *hello world* of grammars provided as an example by Menhir and LRGrep to demonstrate their features.

The relevant parts of the grammar are given in Figure 7.7; the complete example can be found at https://github.com/let-def/lrgrep/tree/master/examples/calc.

Let us analyse the grammar:

```
$ menhir --table --inspection --cmly calc.mly
$ lrgrep enumerate -g calc.cmly
```

```
%left PLUS MINUS        /* lowest precedence */
%left TIMES DIV         /* medium precedence */
%nonassoc UMINUS        /* highest precedence */
%%
main: e = expr EOL { e }
expr:
| i = INT                     { i }
| LPAREN e = expr RPAREN      { e }
| e1 = expr PLUS e2 = expr    { e1 + e2 }
| e1 = expr MINUS e2 = expr   { e1 - e2 }
| e1 = expr TIMES e2 = expr   { e1 * e2 }
| e1 = expr DIV e2 = expr     { e1 / e2 }
| MINUS e = expr %prec UMINUS { - e }
```

Figure 7.7: The *calc* LR grammar

The first lines of output look like:

```
main INT MINUS @ TIMES RPAREN PLUS EOL DIV # [ /expr: expr MINUS . expr ]
main INT DIV @ TIMES RPAREN PLUS EOL DIV # [ /expr: expr DIV . expr ]
main INT PLUS @ TIMES RPAREN PLUS EOL DIV # [ /expr: expr PLUS . expr ]
main LPAREN INT RPAREN @ LPAREN INT # [ /expr: LPAREN expr RPAREN . ]
...
```

Each line consists of a valid prefix and its failing lookaheads, formatted as:

```
<entrypoint> <symbol>* @ <lookahead>* # <reduce-filter pattern>*.
```

In these examples, the entry point is `main`, followed by symbols that form a valid sentence prefix, delimited by `@`. After that comes the list of lookahead symbols that cause the valid prefix to fail. For instance, the valid prefix in the first line is `INT MINUS` and the invalid sentences are `INT MINUS TIMES`, `INT MINUS RPAREN`, `INT MINUS PLUS`, `INT MINUS EOL`, and `INT MINUS DIV`.

The next two lines illustrate variants of the same issue: writing two consecutive operators (except minus, which is allowed as a prefix operator), or an operator followed by a closing parenthesis. Enumeration attempts to group related failures together.

The last part of the line, delimited by #, is a list of reduce-filter patterns matching the sample sentence. These patterns can be used directly in an error specification, e.g.:

```
| /expr: expr MINUS . expr { "Expecting an expression after '-'." }
```

In more complex cases, multiple reduce-filter patterns may appear on the same line. These are different targets visited by reduction sequences applicable to the viable prefix. Handling a single pattern is usually sufficient for an example. If one writes multiple entries in the error specification to handle multiple patterns, the first one to succeed for any input is retained. This has to be kept in mind when writing

the specification: the most specific error messages and patterns should be placed first, to avoid a very generic error message taking precedence over them.

Here are two other lines of output, formatted for better display:

```
main LPAREN LPAREN INT RPAREN @ EOL
  [ /expr: LPAREN expr RPAREN . ]
  [ expr /expr: expr . DIV expr
         /expr: expr . TIMES expr
         /expr: expr . MINUS expr
         /expr: expr . PLUS expr
         /expr: LPAREN expr . RPAREN ]
main LPAREN INT RPAREN @ RPAREN
  [ /expr: LPAREN expr RPAREN . ]
  [ expr /main: expr . EOL
         /expr: expr . DIV expr
         /expr: expr . TIMES expr
         /expr: expr . MINUS expr
         /expr: expr . PLUS expr ]
```

They both identify a failure following a closing parenthesis, but they call for different explanations, so the first pattern `[/expr: LPAREN expr RPAREN .]` should not be used. Whether another closing parenthesis is allowed depends on context that is discovered only after reducing `expr: LPAREN expr RPAREN`. Generally speaking, it is rarely a good idea to use a reduce-filter pattern with an item ready to be reduced (the `.` at the end of the rule); instead, it should be reduced to reveal more context that can be used to improve the error report.

Here are some possible clauses for these situations:

```
| pos=[expr /expr: LPAREN expr . RPAREN ]
  { "Expecting a closing parenthesis for expression started at " ^
     print_position $startpos(pos) }
| pos=[expr] @ RPAREN
  { "Found a closing parenthesis. \
     Expecting a corresponding opening parenthesis at " ^
     print_position $startpos(pos) }
```

Fuzzing a parser with enumeration See our case study of the Elm language in Section 8.3 for an advanced application of enumeration. The error specification for Elm is automatically generated by testing the reference implementation on the output of the enumeration.

7.3.4 Working with coverage report

Let us take another look at writing error messages for the arithmetic example using the coverage check. We start from the error specification given in Figure 7.8. It covers our two guiding examples: a missing integer and an unclosed parenthesis.

To run the analysis, we execute:

```
rule error_message = parse error
| / . INT
  { "Expecting an integer" }
| lpos=LPAREN; [expr / _* . RPAREN]
  { "Expecting a closing parenthesis" ... }
```

Figure 7.8: A tentative specification for the calc grammar of Figure 7.7.

```
$ lrgrep compile -coverage errors.mlyl -g calc.cmly
...
Sentence 1, failing when looking ahead at [RPAREN;LPAREN;INT]:
- RPAREN
  [ /expr: LPAREN expr RPAREN .]
- expr
- LPAREN
- main:
  [expr /main: expr . EOL
        /expr: expr . DIV expr
        /expr: expr . TIMES expr
        /expr: expr . MINUS expr
        /expr: expr . PLUS expr]
Sentence 2, failing when looking ahead at [RPAREN;LPAREN;INT]:
- INT
  [ /expr: INT .]
- main:
  [expr /main: expr . EOL
        /expr: expr . DIV expr
        /expr: expr . TIMES expr
        /expr: expr . MINUS expr
        /expr: expr . PLUS expr]
...
```

LRGrep reports two viable prefixes that cause the parser to fail when looking ahead at `RPAREN`, `LPAREN` or `INT`, annotated with reduce-filter patterns to cover these cases. The two sentences are actually two instances of the same issue: invalid continuations of expressions that are not handled. We complete the coverage by discriminating on the lookahead terminal:

```
| [expr] @ LPAREN { "Unexpected '('. Maybe you forgot to add an operator?" }
| [expr] @ INT { "Unexpected integer. Maybe you forgot to add an operator?" }
| [expr] @ RPAREN { "This closing parenthesis does not match any opening \
                     parenthesis" }
```

Conclusion

In summary, this chapter has provided a comprehensive overview of the implementation details and practical considerations for using LRGrep. From the refinement of clause applicability to the integration with OCaml, we have covered the essential aspects that enable effective error specification and analysis. The tools and techniques discussed here should equip users with the knowledge needed to leverage LRGrep for developing robust and user-friendly error messages in their parser generators.

Chapter 8

Case studies

LRGrep was applied to the parsers of three programming languages, each application illustrating a different approach.

The first two languages, OCaml and Catala, already have parsers written using Menhir.

OCaml served as the primary experience guiding the design of LRGrep. We developed error messages from scratch with the objective of addressing problematic examples reported by the language's user community.

Catala utilised Menhir's existing error message infrastructure. This provided an opportunity to stress-test the reduce-filter pattern dialect and work on compatibility, since nearly all of Menhir's error messages can be automatically imported. While the patterns themselves could be generated automatically, we manually ordered some clauses as this process is not yet fully automated. We refined the order until the behaviour matched that of the reference parser.

Elm is a widely recognised general-purpose programming language known for the quality of its error messages. Its parser was written by hand with the aid of a Haskell-based parsing combinator library, providing specialised primitives for handling syntactic errors.

Our approach to Elm aimed at automatically generating an error specification replicating the behaviour of the reference parser. The initial goal was to validate LRGrep's expressiveness by having a declarative specification produce error messages comparable in quality to those produced by a hand-written parser recognised as state-of-the-art. A secondary objective was to demonstrate the utility of our enumeration concept, showing that LR grammars could be used to test an existing analyser and automatically generate an error specification while treating it as a black box.

In summary:

- Manually crafting high-quality error messages for OCaml
- Testing compatibility with Menhir's approach in Catala
- Matching the quality of an established implementation and leveraging declarative specifications for automation in Elm

```
1. let x = 1 in print_int x
2. let x = 1
3. let y = print_int x;
4.   let z = 2 in
5.   x + z
```

Figure 8.1: Correct but confusing OCaml constructions

8.1 OCaml

OCaml (Leroy et al., 2019) is a functional programming language. It is the language in which Menhir and LRGrep are implemented. The reference implementation uses an LALR(1) parser generated by Menhir from a grammar. This grammar has conflicts and uses Yacc-style precedence rules for resolution.

A distinctive feature of the OCaml grammar is its limited use of delimiters and redundant constructions. Most syntactic structures are not explicitly terminated; instead, a definition concludes by beginning a new one. This characteristic often leads users to inadvertently write programs that generate ambiguous and confusing error messages. The ambiguity stems from potential misinterpretations in two directions: an accidental sequence of tokens may be incorrectly parsed as a continuation of the current definition when the user intended to start a new one, or it may be interpreted as initiating a new definition when the user's intent was to extend the previous one.

Figure 8.1 lists correct OCaml code that exhibits some confusing constructions. When executed, it prints `"1"` twice and defines the values `x` to `1` and `y` to `3`.

Line 1 does not define any value: `x` is local to the expression `print_int x`. Line 2 defines the value `x`: just omitting `in` turns the local variable into a module-level definition. Syntactically, both constructions share the same let-binding prefix. In practice, an expression can span many lines, and one must reach the end of the bindings to see whether they are local or module-level.

On line 3, things get trickier. The indentation is intentionally misleading to make one think that `y` is bound to the result of `print_int x`. However, the `;` is a sequencing operator, forcing the expression to continue on the following line. The `let` on line 4 is let-binding local to the definition of `y`.

These syntactic choices lead to files with a regular structure and minimal noise when used correctly. However, when an error occurs—such as placing an `in` or a `;` in the wrong location—the reported error can be extremely confusing. In practice, experienced programmers use careful indentation and rely on tools for automatic indentation. By comparing the proposed indentation of nested constructions with their intended structure, they are better able to identify and resolve issues.

For reference, other languages generally do not suffer from this problem to the same extent. C-like languages use braces to make the nesting structure explicit, while indentation-based languages such as Python, F#, and Haskell visually clarify nested constructions by requiring deeper indentation for inner structures.

F# is particularly noteworthy because it evolved from OCaml and supports both

```
| LPAREN seq_expr error
    { unclosed "(" $loc($1) ")" $loc($3) }
```

Figure 8.2: A rule of the OCaml parser to report an unclosed parenthesis

```
| self BAR pattern
    { Ppat_or($1, $3) }
| self BAR error
    { expecting $loc($3) "pattern" }
```

Figure 8.3: A standard rule to parse a pattern and an error rule to report that a pattern is expected

syntactic styles. The "light" syntax in F# utilises indentation to mitigate the issues described above, whereas the "verbose" syntax remains compatible with OCaml but retains similar problems. Standard ML adopted a different approach by using distinct keywords for module-level definitions (`val` and `fun`) and local definitions (`let ... in ... end`), explicitly delimiting them to make the programmer's intent clearer.

Modifying the OCaml syntax would break too much existing code, but improving error messages is a feasible and valuable alternative. The development of LRGrep was driven by these requirements: providing more comprehensive error explanations by considering not only the immediate context but also the broader surroundings of an error. This approach aims to account for the path that led to an error and to utilise indentation to offer more precise diagnostics.

8.1.1 The current situation

Today, syntax errors are produced directly by the OCaml parser using specially crafted rules:

- Either by making some legitimate rules more permissive and rejecting incorrect use in the semantic actions
- Or by using the special `error` symbol to catch unterminated constructions or common misuses.

However, only some rules can utilise the `error` symbol due to the rapid introduction of ambiguities. When multiple error clauses apply to a single situation, which one should prevail? As a result, most cases are not covered, and the parser defaults to a generic "Syntax error" message.

Figure 8.2 and 8.3 show how the `error` symbol can be used to handle an unclosed parenthesis and how to handle a missing syntactic construction.

Figure 8.4 shows a permissive rule to parse injective type variance annotations. Only `!+` and `!-` are accepted, but the `PREFIXOP` terminal represents these and many

```
| PREFIXOP
  { if $1 = "!+" then Covariant, Injective else
    if $1 = "!-" then Contravariant, Injective else
    expecting $loc($1) "type_variance" }
```

Figure 8.4: A permissive rule to parse injective type variance annotations

more. The lexer could have been refined to distinguish these from other prefix operators, but this would complicate both the lexer and the parser. Instead, all prefix operators are accepted at the grammar level, and the invalid ones are rejected later.

Incorrect use of Yacc-style error symbol As mentioned in Section 1.7.2, the semantics of the `error` symbol in Yacc can be surprising. The LR rule handling unclosed parentheses in older versions of OCaml is a particularly poor example of this issue, as illustrated in Figure 1.5.

8.1.2 Towards LRGrep-powered error handling

As part of this work, an OCaml parser with LRGrep error handling has been developed as a fork. The long-term goal is to merge it into the official implementation once sufficient quality has been attained.

In this fork, all rules involving the `error` symbol have been removed. More permissive rules have been retained for now, but their status may be reconsidered on a case-by-case basis. Our objective is to centralise syntax error handling in an LRGrep specification.

Apart from these changes, the patch is relatively small—the primary modification involves wrapping the parser invocation to allow LRGrep-generated handlers to manage errors. We also specialised the LRGrep interpreter presented in Section 7.3.2 to directly consume OCaml files.

The majority of the effort has been dedicated to developing new error messages, a process largely informed by issues reported by OCaml users. We now examine some common problems and their translation into LRGrep clauses.

8.1.3 Direct errors

In the analysis of erroneous situations, two distinct categories emerged: direct errors and complex errors. Direct errors arise from immediate violations of grammatical rules, such as the omission of a necessary delimiter or the incorrect use of a keyword. Complex errors, on the other hand, result from unexpected combinations of otherwise grammatically correct constructions, making it difficult to pinpoint a single, clear cause.

This section focuses specifically on general errors, which can be systematically identified by adhering to the structure of the grammar. These errors exhibit a pre-

dictable pattern: "if stuck in this position, suggest this solution", where "position" can be as simple as a grammatical item.

The error messages generated by the current OCaml implementation fall into this category. Similarly, hand-written recursive descent parsers often produce errors of this type. When such parsers are unable to identify an expected construction, they naturally report this expectation. This aligns well with the control flow utilised in direct-style LL parsers, which are typically implemented as mutually recursive functions. Each function recognises a specific construction but operates without knowledge of the broader context.

Unterminated constructions

Error handling within the *if-then-else* construct serves as a case study to illustrate our approach, which can be generalised to other grammatical structures. The integration of error rules into existing grammars adheres to a systematic methodology:

- Examine the syntactic structure and identify potential points for relevant error messages.
- Prioritise these potential errors to ensure that the most critical issues are addressed first.
- Utilise the enumeration mechanism to address overlooked or missing cases.

In OCaml grammar, *if-then-else* is represented by two rules:

```
| IF ext list(attribute) seq_expr THEN expr
| IF ext list(attribute) seq_expr THEN expr ELSE expr
```

The `ext` and `list(attribute)` symbols are advanced use cases. They are nullable and can safely be ignored in the error messages.

During parsing, errors may occur at various stages: immediately following an `IF`, preceding a `THEN`, among others. Each of these scenarios is modelled by filtering the corresponding item, with patterns of the form `[_* / α . β]`. These patterns can be read as "situations where α . β is effective after following any sequence of reductions (the `_*` in brackets)".

The reduction part is not necessary for items in which the dot is positioned immediately after a terminal, e.g., `/ IF . _* | / IF _* THEN . _*`. A reduction cannot produce a terminal, so in these cases `_*` would only match the empty sequence. But when the dot is after a nonterminal, this pattern lets LRGrep find the appropriate sequences, automatically reducing the nullable nonterminals; when unsure, `[_* / ...]` is guaranteed to match as many situations as possible.

Figure 8.5 shows clauses to produce syntax errors when stuck after an *if-then-else* keyword. Even though the patterns are simple, wording the error messages requires setting some conventions upfront. Reminding the user of the syntax assumes that some concise notations for communicating the formal syntax have been established. And in case of ambiguity, should we remind the common prefix `if` ... `then` ... or hint that `else` ... is also possible? In any case, the error message should not be too

```
| / IF . _*
  { "Expecting an expression after 'if'.\n\
     The syntax is 'if <expression> then <expression>'." }
| / IF _* THEN . _*
  { "Expecting an expression after 'then'.
     The syntax is 'if <expression> then <expression>'." }
| / IF _* ELSE . _*
  { "Expecting an expression after 'else'.
     The syntax is 'if <expression> then <expression> else <expression>'." }
```

Figure 8.5: Handling errors after *if-then-else* keywords

```
| [_* / IF _* seq_expr . THEN _*]
  { "Expecting 'then' after the expression.
     The syntax is 'if <expression> then <expression>'." }
| [_* / IF _* . seq_expr THEN _*]
  { "Expecting an expression.
     The syntax is 'if <expression> then <expression>'." }
```

Figure 8.6: Handling errors between `if` and `then`

verbose. It is also worth having a notation to link to a more detailed explanation, such as a dedicated section in the reference manual of the language.

These questions are not settled. Our work concentrates on demonstrating the technical feasibility of these fine-grained error messages. Deciding on a uniform wording for errors for the OCaml grammar is left to future work.

Figure 8.6 adds clauses handling errors happening between `if` and `then`. Being stuck before the `else` is a bit more contentious: when a suffix of the stack reduces to `IF ext list(attribute) seq_expr THEN expr`, a valid *if-then* has been recognised and can be reduced. We can mention that `else <expression>` is a valid continuation:

```
| [_* / IF _* . ELSE _*]
  { "Expecting 'else' after the expression.
     The syntax is 'if <expression> then <expression> else <expression>'." }
```

But this rule might take precedence over a more relevant rule. For instance, with the input `(if foo then bar`, is it more relevant to point to the unclosed parenthesis or to suggest adding an `else`? We decided to ignore this case and not report `else` as a possible continuation. If one wants to handle a case like this, a way out is to put the `else` clause before the unclosed parenthesis one.

An alternative approach is shown in Figure 8.7. A partial rule with side-effects is used to remember that `else` is a valid continuation and to decide later which message is appropriate.

```
| [_* / IF _* . ELSE _*]
  %partial { else_valid := true; None }
  ...
| [seq_expr / LPAREN seq_expr . RPAREN]
  { report "Expecting a closing parenthesis";
    if !else_valid then report " or an 'else'" }
```

Figure 8.7: Detecting *if-then* in unclosed parentheses

Unclosed parentheses

A typical case worth examining is the unclosed parenthesis. We follow a similar approach as for *if-then-else*: filtering by items, using the grammar and enumeration to find relevant cases.

However, the OCaml grammar allows parentheses in many contexts (expressions, patterns, types, modules, ...) and for various purposes: for disambiguation, as in `(1 + 2) * 3`, for applicative notation `Foo(Bar)`, or for specialised constructions like `(`e `:` τ `:>` τ'`)`, a separate rule for the coercion of `e` from type τ to type τ'.

We use a single clause that catches all these cases:

```
| lp=LPAREN; _*; [_* / _* LPAREN _* . RPAREN _*]
  { "Unclosed parenthesis at " ^ line_and_char $startloc(lp) }
```

The pattern is general enough to cover unclosed parentheses in all constructions of the OCaml language. It works by following all reductions that end up in an item where the dot immediately precedes a right parenthesis. It then scans the stack for a left parenthesis whose position is reported in the error message.

A limitation with this approach is that `LPAREN` and `RPAREN` must appear literally. Hiding a parenthesis behind a nonterminal, for instance using `lp` defined by the rule `lp: LPAREN { ... }` prevents the pattern from matching.

A rule with multiple sets of parentheses, such as hard-coding the nesting of two parentheses: `LPAREN LPAREN expr RPAREN RPAREN`, will also confuse the pattern as it matches either `RPAREN` but always captures the location of the rightmost `LPAREN` on the stack.

8.1.4 Complex errors

The second category of errors stems from unforeseen interactions between grammatical structures. Such errors often arise from partially overlapping constructions. If distinct contexts recognise marginally different languages, end-users who are oblivious to these nuances can easily become confused.

A typical example in OCaml is the first one introduced in this chapter. At the beginning of a structure[1], an expression and a value definition are both valid con-

[1] A structure defines the body of a module. The `structure` nonterminal recognises a list of definitions: types, values, sub-modules, etc.

tinuations. Both can start with a `let`-binding: terminated with `in` for the expression and not delimited for the value definition.

We will examine this and some other examples in detail. The main difference from the previous category is that these cases were not found by analysing the grammar but through practice. They were brought to our attention by the OCaml user community—novice students seeking assistance and seasoned users identifying apparent parser limitations.

Despite our tools' inability to predict such issues proactively, we were pleased that the pattern language was expressive enough to handle all reported scenarios. Furthermore, the tooling facilitated the creation of precise patterns from the provided examples.

Value definitions and expressions

The prototypical example is:

```
1. let x = 5;
2. let y = 6
3. let z = 7
```

The upstream compiler rejects this input with the message:

```
File "sample.ml", line 3, characters 0-3:
3 | let z = 7
    ^^^
Error: Syntax error
```

The error is reported on the `let` on line 3 because it is possible to terminate the let-binding on line 2 to get a valid expression: `let x = 5; let y = 6 in foo`. Though it is more likely that the user did not intend to add `;` on line 1. `let x = 5` is a valid value definition, but `5;` is an unterminated sequence that expects an expression. We want to detect this `;` and notify the user.

We use the LRGrep OCaml interpreter to get an idea of what a pattern for this case could look like. The trace is provided in Figure 8.8. The first line indicates that parsing failed on the `let` of line 3, as expected. The rest of the output lists the viable prefix recognised at the point of failure, from right to left.

The top of the stack is too precise for the situation we are trying to identify, so we look for a relevant reduction, indicated by the ↱ … lines. The bottom-most one, `let_bindings(ext)`, is a good candidate: it reduces just before the `SEMI` we are interested in. The pattern `semi=SEMI; [let_bindings(ext)]` captures a semicolon that precedes a well-formed let-binding.

However, a finer-grained pattern is needed for the situation we want to identify. For instance, the input `let () = let x = 5; let y = 6 let` … fails and matches the pattern, but suggesting to remove `;` is incorrect. Without `;`, the parser would expect an `in` to complete the unterminated expression `let x = 5`.

We want to report the `;` only if omitting it resolves the issue; that is, when the symbols that precede form a valid definition. A second use of the reduction operator

```
$ lrgrep-ocaml-interpreter sample.ml
File "sample.ml", line 3, characters 0-3,
parser stack (most recent first):
- line 2:8-9  INT
                [constant: INT .]↱
                constant↱
                simple_expr↱
                expr↱
                seq_expr
- line 2:6-7  EQUAL↱
                strict_binding
- line 2:4-5  val_ident↱
                let_binding_body_no_punning↱
                 list(post_item_attribute)↱
                let_binding_body
- line 2:3-3  rec_flag
- line 2:3-3  list(attribute)
- line 2:3-3  ext
- line 2:0-3  LET↱
                let_bindings(ext)
- line 1:9-10 SEMI
- line 1:8-9  expr
- line 1:6-7  EQUAL
- line 1:4-5  val_ident
- line 1:3-3  rec_flag
- line 1:3-3  list(attribute)
- line 1:3-3  ext
- line 1:0-3  LET
- entrypoint  implementation
```

Figure 8.8: Annotated trace of `sample.ml`'s failure

```
$ echo "let x = 5" | lrgrep-ocaml-interpreter
File "<stdin>", line 2, characters 0-0,
parser stack (most recent first):
- line 1:8-9  INT
                [constant: INT .]↱
                constant↱
                simple_expr↱
                expr↱
                seq_expr
- line 1:6-7  EQUAL↱
                strict_binding
- line 1:4-5  val_ident↱
                let_binding_body_no_punning↱
                 list(post_item_attribute)↱
                let_binding_body
- line 1:3-3  rec_flag
- line 1:3-3  list(attribute)
- line 1:3-3  ext
- line 1:0-3  LET↱
                let_bindings(ext)↱
                 list(structure_element)↱
                structure_item↱
                list(structure_element)↱
                structure
- entrypoint  implementation
```

Figure 8.9: Trace of the parser's configuration immediately before "`;`"

can solve this. We query the interpreter again for the possible reductions by feeding it the prefix of the input that precedes the `;`. The trace is given in Figure 8.9.

Looking at the reductions, `structure_item` is a sensible choice. This nonterminal is responsible for recognising a single definition in a structure. If a prefix reduces to a `structure_item`, it forms a valid definition. We get this clause:

```
| [structure_item]; semi=SEMI; [let_bindings(ext)]
  { "Expecting 'in' after let binding(s) in an expression,\n\
    though the error might be caused by the ';' at " ^
     string_of_location $startloc(semi) }
```

Indentation-based heuristic Figure 8.10 shows a refinement using indentation to guess the user's intent. If the let-bindings are indented at the same level as the previous definition, the `;` is likely the culprit. But if the let-bindings are more indented, maybe they really are intended to be local definitions and `in` is missing.

We decided not to use this heuristic and always give the same message. This can be reconsidered if users report that the situation is still confusing.

```
| l1=[structure_item]; semi=SEMI; l2=[let_bindings(ext)]
  { if $startloc(l1).pos_cnum < $startloc(l2).pos_cnum then
      (* less-indented: 'in' is more likely *)
       "Expecting 'in' after let binding(s) in an expression,\n\
        though the error might be caused by the semicolon at " ^
       string_of_location $startloc(semi)
    else
      (* same indentation: focus on ';' *)
       "The ';' at " ^ string_of_location $startloc(semi) ^
       " is likely to be incorrect. Otherwise, \
         'in' is expected after the let binding."
  }
```

Figure 8.10: Using indentation to choose an explanation

A problematic variation The following code[2] exhibits a closely related problem that is not covered by our previous pattern:

```
let x = foo; 5 +
let y = 6
let z = 7
```

The error is still detected when reaching the third `let`. Finishing the second let-binding is a possible but improbable fix. Nesting let-bindings inside operator application is very unusual in OCaml:

```
let x = foo; 5 + let y = 6 in y
```

A more likely explanation is that the programmer was interrupted while writing the addition on the first line. The right hint is to finish it by adding an expression. Guided by the interpreter, we find that the pattern `expr; PLUS; [let_bindings(ext)]` recognises this situation. However, it is too specific: what if the user had written `5 -` instead?

We decided to point out the location that precedes the let-binding. The user is more likely to find a solution if they know where an expression was expected. This more general clause captures this:

```
| loc=_; / expr : _* . expr; [let_bindings(ext)]
  { "Expecting 'in' after let binding(s) in an expression.\n\
    This might be due to the unterminated expression at " ^
     string_of_location $endloc(loc) }
```

It recognises incomplete let-bindings in a context where an expression is expected and reports the location of whatever precedes the let-bindings.

Parameters of a data constructor

Defining a new sum type in OCaml looks like:

[2] Thanks to Ty Overby for reporting this case

```
type a_new_type = A of string | B of string * string
```

The sum type is named `a_new_type`, and it has two data constructors: `A` that expects a string argument, e.g. `A("foo")`, and `B` that expects two string arguments, e.g. `B("bar", "baz")`. The syntax of parameters looks a lot like a type expression, so a user might be tempted to use an arrow type. Let's try with the OCaml top-level:

```
# type another_type = C of int -> int;;
                               ^^
Error: Syntax error
```

The example is rejected, even though `int -> int` is a valid OCaml type denoting functions from integers to integers. In reality, only a subset of type expressions are accepted in this position. To understand why, let us take another look at `B`: we said that it expects two arguments. But `string * string` commonly denotes a tuple of two strings: a single argument that is a pair of strings. The syntax of parameters repurposes `*` for the introduction of multiple parameters. Since in the syntax of types, `*` binds tighter than `->`, it is safer to reject `->`. Otherwise, consider this type definition:

```
type worse_type = D of int * int -> int * int
```

Is `D` expecting three arguments (an `int`, an `int -> int` function and another `int`), or a single function of type `(int * int) -> (int * int)`? The grammar was designed to reject this ambiguous notation, forcing the user to add parentheses. However, there is no error message to explain this decision to the user, which can arguably be confusing.

As before, we give the input to the interpreter to visualise the viable prefix and get a sense of the possible reductions:

```
$ lrgrep-ocaml-interpreter -
type worse_type = D of int * int -> int * int
File "<stdin>", line 1, characters 33-35,
parser stack (most recent first):
- line 1:29-32        LIDENT
                [mk_longident(mod_ext_longident,LIDENT): LIDENT .]r
                mk_longident(mod_ext_longident,LIDENT)r
                type_longidentr
                atomic_type
- line 1:27-28        STAR
- line 1:23-26        reversed_separated_nonempty_llist(STAR,...)r
                constructor_argumentsr
                reversed_separated_nonempty_llist(STAR,atomic_type)
- line 1:20-22        OF
...
```

The confusing dialect of type expressions starts after the `of` keyword, so there is no point in looking deeper in the stack. We can see that the input reduces to `constructor_arguments`. This nonterminal recognises the dialect we are interested in: sequences of types separated by `*`, without arrows.

If parsing failed after recognising valid arguments and looking ahead at `->`, represented by the terminal `MINUSGREATER` in OCaml grammar, then we can confidently suggest the user put parentheses:

```
| [constructor_arguments] @ MINUSGREATER
  { "In constructor arguments, arrow types should \
     be wrapped between parentheses." }
```

Robust handling of the `constructor_arguments` dialect

Now that we have identified a subtlety in the grammar, we take some time to consider the possible variations.

The syntax of arrow types also recognises labelled arguments, as in `a:int -> int`, and optional arguments, as in `?a:int -> int`. By reading the grammar and looking at the output of the interpreter, we end up with two additional clauses:

```
| [constructor_arguments]; STAR; LIDENT @ COLON
  { "This looks like a labelled argument. \
     In constructor arguments, arrow types should \
     be wrapped between parentheses." }
| [constructor_arguments]; STAR @ QUESTION
  { "This looks like an optional argument. \
     In constructor arguments, arrow types should \
     be wrapped between parentheses." }
```

We do not know if the user intended to introduce an arrow type: the parsing failed before reaching the `MINUSGREATER` terminal. Our message starts by explaining what we identified, a labelled or optional argument, to prevent confusion.

OCaml also supports GADT constructors, introduced by a slight syntax variant. We are reminded of them by searching for the references to `constructor_arguments` in the grammar:

```
| OF constructor_arguments
| COLON constructor_arguments MINUSGREATER
    atomic_type %prec below_HASH
```

The second flavour also repurposes `MINUSGREATER`. In this case, the parser accepts the `MINUSGREATER` and fails after, so our first clause does not cover the case. We can anticipate some possible failures:

- The rule for GADT allows a single `MINUSGREATER` (e.g., `type a = D : t1 -> t2`). A second `MINUSGREATER` after the atomic type is rejected, as in:
  `type a = D : t1 -> t2 -> t3`.
- Tuples are not recognised by `atomic_type` either, so the parser also fails if the user writes a tuple, introduced by `*`.

```
| COLON; constructor_arguments; MINUSGREATER;
    [atomic_type] @ MINUSGREATER
  { "In constructor arguments, arrow types should \
     be wrapped between parentheses." }
| COLON; constructor_arguments; MINUSGREATER; [atomic_type] @ STAR
  { "A tuple cannot appear in this position. \
     If you intend to introduce a functional argument \
     returning a tuple, wrap the function type between \
     parentheses." }
```

8.1.5 Future work: wording of error messages

In this case study, we concentrated on developing patterns to accurately characterise errors identified through user reports in the context of the OCaml language. The design of LRGrep evolved as a direct outcome of this endeavour.

Our initial focus was not on refining the wording of error messages; rather, we established a robust framework capable of producing precise error reports, which we validated through prototyping. However, significant work remains in crafting messages that maximise user comprehension. Balancing clarity and brevity while catering to both novice and experienced users presents a complex challenge that requires meticulous attention.

Future refinement of the error messages will be approached incrementally, incorporating feedback from the broader OCaml community to enhance usability and effectiveness. The real difficulty lies in providing accurate explanations for complex errors. While we are capable of effectively handling these errors once they are reported by a user, anticipating and preemptively addressing them remains a significant challenge. This ongoing process will involve continuous improvement based on empirical data and community feedback to refine the precision and clarity of our error messages.

8.2 Catala

Catala (Merigoux et al., 2021) is a domain-specific language for implementing legislative texts. The current applications focus on executable translations of tax code.

Its implementation and distribution closely follows those of general-purpose languages: it comes with a compiler, Catala, and a build system, Clerk. The compiler is designed around the textbook compilation pipeline: lexer, parser, intermediate representations, and multiple backends. It is implemented in OCaml, and the parser uses Menhir.

However, the target audience includes not only programmers but also lawyers and policy makers. Consequently, the syntax is relatively verbose, with constructions inspired by the natural language used in tax code. Attention is paid to ensure that the tooling is user-friendly for non-experts. The parser relies on Menhir's message infrastructure to provide syntax error messages.

Another peculiarity is the multilingual aspect of the pipeline. The language aims to be usable by non-English speakers, and as of today, the frontend supports French, English, and Polish dialects. This is achieved by using language-specific lexers targeting the same set of terminals; the parser itself remains unchanged.

Here is an excerpt of Catala code, taken from their homepage:

```
scope QualifiedEmployeeDiscount :
  definition qualified_employee_discount
    under condition is_property consequence
  equals
    if employee_discount >=
      customer_price * gross_profit_percentage
    then customer_price * gross_profit_percentage
    else employee_discount
```

To validate the design of LRGrep, we translated the error messages of Catala to an LRGrep specification. There is no plan for the upstream implementation to change in the short term, but the development team is interested in potential improvements. More concretely:

- The existing error infrastructure adds maintenance work when evolving the grammar; it has now stabilised, but in anticipation of future evolutions, removing friction would be beneficial.
- The expressiveness of the current system is limited and rigid; in particular, it makes experimentation with more specific error messages difficult and leaves little room for improvement.

The situation is not dire overall. Catala has exhaustive coverage using handwritten error messages and no further work is required as long as the grammar does not change. When it does change, Menhir can tell at compile-time how the existing error messages are affected.

Our work consisted solely in translating the existing error messages to an LRGrep specification, directly addressing the maintenance concern. As it stands, our fork offers no observable difference to the user of Catala. For the implementor, however, our specification is more concise. A single LRGrep pattern can identify a family of errors that was previously specified by a list of invalid sentences. The expressiveness concern is also addressed in theory, opening the way for new experiments, but we limited our work to reproducing existing results.

8.2.1 The current design

The parser and its error messages can be found in the directory `compiler/surface`, implemented in the two files `parser.mly` and `parser.messages`.

The parser does not use the `%on_error_reduce` directive. When asked, the developers of Catala reported that, even after thoroughly reviewing Menhir's manual, the advantages and application scenarios of the `%on_error_reduce` directive remained

unclear. Therefore, considering Menhir's automaton generation strategy, the only reductions that can happen prior to an error being reported are default reductions.

The rest of the specification is the `.messages` file, a Menhir error message file. It is a list of erroneous inputs followed by messages to report. An entry looks like:

```
source_file: BEGIN_CODE DECLARATION ENUM UIDENT COLON ALT UIDENT CONTENT
             TEXT YEAR
##
## Ends in an error in state: 362.
##
## list(enum_decl_line) -> enum_decl_line . list(enum_decl_line)
##   [ SCOPE END_CODE DECLARATION ]
##
## The known suffix of the stack is as follows:
## enum_decl_line
##

expected another enum case, or a new declaration or scope use
```

The first line means that when trying to parse a `source_file` (one of the entry-points of the grammar), the parser fails when reading the sequence of terminals after the `:`. More precisely, `BEGIN_CODE DECLARATION ENUM UIDENT COLON ALT UIDENT CONTENT TEXT` is a valid prefix, but looking ahead at `YEAR` immediately after causes the failure. The comment lines, beginning with `#`, are generated by Menhir to help make sense of the situation. This case failed in state number 362. In this state, `list(enum_decl_line) -> enum_decl_line . list(...)` is an active item, and `SCOPE`, `END_CODE` and `DECLARATION` are lookahead tokens inherited from the context. In this state, we know that `enum_decl_line` is a suffix of the stack.

The message follows. It is hand-written and it accurately reflects the information known in this state of the automaton: the item tells us that we can recognise another `enum_decl_line`, and the inherited look-aheads hint at the other possible continuations.

There are around 260 messages like this one, totalling 4400 lines.

Autosuggestions To run the example through Catala, we translate the terminals into a valid sentence:

```
declaration enumeration X : -- X content text year
```

The actual output is:

```
[ERROR] Syntax error at token "year"
Message: expected another enum case, or a new declaration or scope use
Autosuggestion: did you mean "declaration", or maybe "--", or maybe "scope"?
```

The textual error message is completed with suggestions. This part is generated automatically by querying the parser for valid continuations. We preserve this behaviour as is—actually, our work does not require any change to the autosuggestion algorithm.

Such suggestions have been formally studied by Sasano and Choi (2021), which hints at possible improvements such as reminding the user of the syntax of a construction rather than suggesting a single symbol. Some of this information is already computed by LRGrep and could be exposed to the grammar author in the future. See Section 9.3.

8.2.2 Prerequisite: preventing regressions

Our goal is to preserve the existing behaviour. The first step is to create a testsuite to record current messages and check future changes.

The testsuite directly follows the structure of the `.messages` file. For each entry:

- we generate a Catala file that translates to the same sequence of terminals
- we record the output of the compiler for that failure

Every time the parser changes, we re-run the compiler on each input to check if the error message has been affected.

Unreachable entries A minor annoyance is that some entries are actually unreachable for lexical reasons. Menhir is correct to report that a specific sequence of tokens will cause a failure, but the actual sequence cannot be produced by Catala's lexer. The reason is that Catala supports literate programming. Some tokens change the behaviour of the lexer, switching it back-and-forth between a writing mode, in which free text is accepted, and a programming mode in which the lexical conventions of Catala apply.

8.2.3 Translating the Specification

To get a rough LRGrep specification from a `.messages` file, the first step is to translate each entry into an LRGrep pattern that matches the same state. This is done by turning the items, listed by Menhir in the comments, into filters. This part can be done mechanically.

Looking for Terminals

When the dot is on the right side of a terminal, the translation is almost immediate. For instance, let us consider this entry:

```
source_file: BEGIN_CODE DECLARATION ENUM UIDENT COLON ALT UIDENT
             YEAR
##
## Ends in an error in state: 357.
##
## enum_decl_line -> ALT UIDENT . option(preceded(CONTENT,addpos(typ)))
##   [ SCOPE END_CODE DECLARATION ALT ]
##
```

```
## The known suffix of the stack is as follows:
## ALT UIDENT
##

expected a payload for your enum case, or another case or declaration
```

The item translates to filter:

```
/enum_decl_line: ALT UIDENT . option(preceded(CONTENT,addpos(typ)))
```

We also decided to turn the known suffix of the stack into direct matching of symbols. This is not necessary as filtering is enough to guarantee the intended behaviour. But it can come in handy to extract semantic information in the future. The resulting clause is:

```
| ALT; UIDENT /enum_decl_line: ALT UIDENT . option(preceded(CONTENT,typ))
  { "expected a payload for your enum case, or another case or declaration" }
```

It can also be the case that multiple items are active in the state reached by an entry. In this case, we filter for each item. For instance:

```
| struct_scope_base; DEPENDS
  / struct_scope: struct_scope_base DEPENDS . separated_nonempty_list(...)
  / struct_scope: struct_scope_base DEPENDS . LPAREN ... RPAREN
  { "expected the type of the parameter of this struct data function" }
```

Looking for Nonterminals

States having a nonterminal as incoming symbol (on the top of stack) require slightly more work. A parser waiting for input is either in its initial state or has just shifted a terminal. For a nonterminal to be on the stack, some reductions had to occur and we simulate them with the reduction operator.

Aside from that, the translation is similar: after reducing, we filter for items. For instance, this entry:

```
source_file: BEGIN_CODE DECLARATION STRUCT UIDENT COLON CONDITION LIDENT
             YEAR
##
## Ends in an error in state: 300.
##
## struct_scope -> struct_scope_base . DEPENDS separated_nonempty_list(...)
##  [ SCOPE END_CODE DECLARATION DATA CONDITION ]
## struct_scope -> struct_scope_base . DEPENDS LPAREN ... RPAREN
##  [ SCOPE END_CODE DECLARATION DATA CONDITION ]
## struct_scope -> struct_scope_base .
##  [ SCOPE END_CODE DECLARATION DATA CONDITION ]
##
## The known suffix of the stack is as follows:
## struct_scope_base
##
```

```
expected a new struct data, or another declaration or scope use
```

Which we translate to the clause:

```
| [struct_scope_base
  / struct_scope: struct_scope_base . DEPENDS separated_nonempty_list(...)
  / struct_scope: struct_scope_base . DEPENDS LPAREN ... RPAREN
  / struct_scope: struct_scope_base . ]
  { "expected a new struct data, or another declaration or scope use" }
```

When the known prefix contains multiple symbols, one has to be more careful. Only nonterminals should be included in the reduction. For instance, see the rule looking for an unclosed parenthesis:

```
| LPAREN; [expression / expression: LPAREN expression . RPAREN]
  { "unmatched parenthesis that should have been closed by here" }
```

The `LPAREN` is also matched, but left out of the reduction. It becomes trickier when the suffix is made up of multiple nonterminals. This is a problem if either of these nonterminals recognises ϵ: we cannot tell if both of them are the result of the reduction, or if the first one was already on the stack before reducing. In this case, the simplest is to use a wildcard and let LRGrep figure out the possible sequences of reductions, e.g.:

```
  | [_* /scope_decl_item: scope_decl_item_attribute lident . CONTENT ...
       /scope_decl_item: scope_decl_item_attribute lident . CONDITION ...]
    { "expected either 'condition', or 'content' \
       followed by the expected variable type" }
```

Ordering Clauses

After the translation, we have a list of clauses, one for each `.messages` entry. Each clause identifies the same state as the entry it is translated from, but it might also recognise more states. In this case, the textual ordering rule applies: the first rule to match a situation is retained.

This is not the meaning of the original `.message` entries which matched a single state each. To recover this behaviour, we used a few heuristics. We put the clause not involving reduction (those matching terminals) before those matching reductions (matching nonterminals). Reducing generalises a pattern and is likely to match more situations. Then, inside each group, we put the clauses matching more items first. Pattern `/A: a . b /B: a . c` is strictly more restrictive than pattern `/A: a . b`. If `/A: a . b` comes first, the second will never match.

After that, we proceeded by trial and error, reordering clauses until the behaviour was identical to the reference tests.

The problem has to do with reductions: Menhir reduces before looking for an error message. The default reduction optimisation imposes a first set of reductions. Then the parser can specify and order other reductions that should be applied with the `%on_error_reduce` directive. The sequence of reductions is fixed at compile-time

and independent of the error message. LRGrep takes a different stance: rather than ranking reductions, the specification ranks the error messages.

We managed to reproduce the original behaviour after a few iterations. While the translation of the error cases of a `.messages` file to error patterns can be done mechanically, we do not have an automated solution to order the resulting clauses yet. This is not an ideal situation but it was sufficient and quick enough for this problem. In principle, it is possible to automate this part and we leave this for future work.

Simplifying and sharing patterns

Once we are sure to preserve the existing behaviour, we can simplify the specification to make it more readable.

Although the `.messages` had 260 entries, it had only 160 unique error messages. We used *or-patterns* to factorise clauses with the same message. For instance, clauses catching an error after a `PLUS` or a `MULT` were shared using:

```
| PLUS          / expression: expression PLUS . expression
| MULT          / expression: expression MULT . expression
  { "expected an expression" }
```

Another simplification was to remove useless item filters. For instance, for an *if-then-else*, we had the following clause:

```
[expression
/ expression: expression . DOT qlident
/ expression: expression . OF funcall_args
/ expression: expression . WITH constructor_binding
/ expression: expression . CONTAINS expression
/ expression: expression . FOR lident AMONG expression
/ expression: expression . MULT expression
/ expression: expression . DIV expression
... (* going through all binary operators *)
/ expression: IF expression . THEN expression ELSE expression
/ expression: expression . FOR lident AMONG expression SUCH THAT expression]
{ "expected the \"then\" keyword as the conditional expression is complete" }
```

The only relevant item is `expression: IF expression . THEN expression ELSE expression`, and this is reflected in the error message: we are looking for a `then` keyword, the rest is an artefact of the automaton extraction. In principle, LRGrep could warn the user when some items do not contribute to reducing a set of situations, but we have not implemented this feature yet. This clause simplifies to:

```
[expression / expression: IF expression . THEN expression ELSE expression]
{ "expected the \"then\" keyword as the conditional expression is complete" }
```

Finally, repeating all symbols in an item that come after a `.` is rarely useful. A pattern like:

```
| /expression: lident AMONG expression SUCH . THAT expression IS MAXIMUM OR IF
    COLLECTION EMPTY THEN expression
```

```
  /expression: lident AMONG expression SUCH . THAT expression IS MINIMUM OR IF
    COLLECTION EMPTY THEN expression
```

Simplifies to:

```
| /expression: lident AMONG expression SUCH . THAT expression ...
```

Final specification

We end up with a 550 lines long specification file. It is a list of clauses, whose error messages read like a transcription of their pattern most of the time. Here is a short excerpt:

```
...
| /assertion: VARIES . _ WITH_V expression _
  { "expecting the name of the varying variable" }
| /scope_item: ASSERTION . assertion
  { "expected an expression that should be asserted during execution" }
| /definition: _ _ DEFINITION _ _ _ _ DEFINED_AS . expression
  { "expected an expression for the definition" }
...
```

The specification is short and declarative enough to be readable without having to continuously look at the parser's grammar to figure out what is going on. Maintenance has been light so far, though our experience is too brief to draw any conclusion.

8.3 Elm

Elm (Czaplicki and Chong, 2013) is a general-purpose, purely functional programming language targeting web applications. It gained recognition for popularising the "Elm architecture", a widely adopted approach to building reactive user interfaces. Our focus here lies on its compiler. To ensure maximum accessibility, special attention has been given to error messages, both syntactic and semantic. Notably, the compiler endeavours not only to explain the detected errors but also to clarify the expected usage of the language within the identified context.

The compiler's error messages are highlighted as one of its key strengths, which prompted us to investigate whether we could reproduce these syntax errors with a declarative specification. We deemed it worthwhile to explore whether LRGrep could effectively replicate Elm's approach in presenting clear and informative error notifications.

8.3.1 Elm's Syntax

There is no established reference grammar for the Elm programming language; it is entirely defined by its implementation and evolves with each version, rendering it somewhat unstable as new versions may introduce significant syntax changes, particularly in advanced features. The core remains relatively stable, however. This

makes it challenging to develop an independent parser, as attempts at formalising the grammar available online are limited and outdated. Official documentation is minimal, focusing primarily on examples without explicitly outlining the syntax of various constructs.

Consequently, our study involved a detailed examination of the Elm compiler's implementation to gain a comprehensive understanding of its grammar.

Syntax errors are represented by an appropriate algebraic type and generated by the parser upon encountering an error. These errors are then propagated to be handled by an error management infrastructure shared by the different compilation passes.

8.3.2 An LR Grammar for Mini-Elm

Given that LRGrep is specifically designed for LR grammars, we created our own for the Elm language. The development process involved several sources:

- Incomplete existing grammars: we started by examining these resources as a foundation.
- The compiler's changelog: this provided insights into the evolution of the language syntax.
- Popular libraries, maintained by active Elm users: these served as practical examples and validation for our work.

The libraries not only helped us to adapt and refine the grammar but also validated our alternative implementation, which we named Mini-Elm. Our focus was on the subset of the language used in those libraries, though many features were covered, indicating that a significant portion of Elm's syntax is represented.

Thus, we have developed an LR grammar for (a subset of) Elm, which, to the best of our knowledge, represents the closest approximation of a reference grammar for this language. In the following section, we look into some unique aspects of this implementation.

Indentation-sensitive Grammar

The Elm programming language exhibits sensitivity to indentation, as some syntactic structures depend on the column at which they begin. This is a lexical consideration that goes beyond the formalism of context-free grammars. Fortunately, these can be encoded by representing some indentation conditions using special terminals.

Beginning of Line An Elm source file is composed of a header followed by a sequence of declarations. These constructs are mandated to start at the first column, and they are the only language elements that can begin on the first column.

```
grammar:
| BOL module_decl
  imports_and_decls
  {}
;

imports_and_decls:
| BOL import imports_and_decls
| decls
  {}
;
```

```
decls:
| EOF
| BOL EOF
| BOL decl decls
  {}
;
```

Figure 8.11: Encoding of beginning-of-line constraints

To encode this structural requirement, we introduce a terminal `BOL`, short for beginning-of-line. It signifies that the subsequent terminal starts at the beginning of the line: `BOL` is emitted before every terminal that initiates a new line (either at the beginning of the file or after a `"\n"` line delimiter) and is not preceded by any whitespace. Figure 8.11 shows the uses of this terminal in the grammar.

Grammar constructs that can only appear at the beginning of a line are preceded by the `BOL` terminal. Inserting these `BOL` terminals is straightforward: we intercept the tokens produced by the lexer and insert a `BOL` whenever a terminal is detected at the beginning of a line. This adds minimal complexity to the lexer and parser implementations and ensures that a declaration is accepted if, and only if, it begins a new line. Any other construct is rejected, eliminating ambiguities.

Nested Layout Contexts Indentation also plays a role in delimiting the branches of a `case` analysis and the bindings of a `let` expression. These constructs do not have explicit closing delimiters; they are automatically closed when followed by a terminal less indented than the one that opened the structure. This indentation-based mechanism facilitates nesting of such constructions. For instance:

```
turnLeft direction =
    case direction of
                North -> West
                West  -> South
                South -> East
                East  -> North

average numbers =
    let
       count = List.length numbers
       total = List.sum numbers
    in
    if count > 0 then total / count else 0
```

```
expr:
| "case" expr "of" layout_sensitive(case_branch)+ {...}
| "let" layout_sensitive(let_def)+ "in" expr      {...}
```

Figure 8.12: Declarative specification of layout-sensitive constructions

In a branch of a case analysis, there is no delimiter to separate the expression from the next pattern. Typically, the expression should be indented more than the pattern. The first pattern determines the indentation level, and following patterns have to begin on the same column. Similarly, in the `let` binding, there is no explicit separation between the definitions; `numbers` continues the previous expression because it is more indented than `count`, while `total` begins a new definition because it is aligned with `count`.

To capture this in an LR grammar, we introduce the concept of "layout context". A new context starts when entering an indentation-sensitive construct and finishes when exiting this construct. Each definition of a let-binding has its own context.

When entering a new context, the column of the first terminal of the construct is recorded. A context is exited when the syntactic construction is complete and followed by a terminal on the same or a less-indented column. This necessitates collaboration between the lexer and parser: the indentation column is determined during lexical processing, but entering and exiting a layout context is governed by the parser.

Layout contexts can be nested. The parser maintains a stack of columns for the active layout contexts. The lexer stores the indentation column of the current lookahead symbol in `layout_column` and introduces a `LAYOUT_LEAVE` terminal when this column is less than or equal to the column at the top of the layout stack. The indentation stack is managed by these two ϵ-rules:

```
layout_enter: /*empty*/ { push layout_context !layout_column };
layout_leave: /*empty*/ { pop layout_context };
```

A higher-order rule is used to enforce a well-bracketed use of the stack:

```
layout_sensitive(X): layout_enter x=X layout_leave LAYOUT_LEAVE { x };
```

Parsing an indentation-sensitive construct starts by pushing the indentation level on the layout stack, and the construct is terminated by a `LAYOUT_LEAVE` token. Since the lexer decides to introduce `LAYOUT_LEAVE` by looking only at the column, it is possible to introduce one in incorrect situations. The parser is designed to ignore a `LAYOUT_LEAVE` in an incorrect context.

Put together, this permits a concise grammar for cases and let-bindings shown in Figure 8.12.

Operator precedences

Older versions of Elm allowed users to introduce custom operators and to customise their associativity and priority. To support this in an LR grammar, the expressions are

parsed in two passes. The LR grammar does not need to be aware of the customisations: applications of operators are simply parsed as a list of operands and operators, an "operator soup". Once recognised, a specialised precedence parser such as Pratt (1973), or Dijkstra's *shunting yard algorithm*, turns the soup into a proper syntax tree. Such a parser can be efficiently reconfigured dynamically.

Our frontend is still designed to support this feature, even though it was removed in Elm 0.19. We kept this implementation because it allows for a simple specification where the LR grammar is unaware of operator precedence. We care only about error messages which are not affected by the actual priorities in the current implementations (neither in the reference nor in our prototype).

Thus, all infix binary operators are handled by a single grammatical rule:

```
expr: app_expr appop expr {}
```

8.3.3 Error messages for Mini-Elm

In this section, we present a method for automatically generating an error specification from a declarative grammar and a reference compiler implementation.

Utilising our LR grammar for Elm along with the enumeration process detailed in Section 6.2, we produce a set of incorrect sentences that encompass all reduce-filter patterns (specifically, pairs of these patterns and a corresponding failing lookahead terminal).

These generated sentences are fed into the Elm compiler using a minimal printer tailored for invalid inputs. We then analyse the error messages returned by the compiler. By relating the error message used to explain an invalid sentence with the reduce-filter patterns matching this sentence, we are able to infer an error specification for the grammar, mimicking the behaviour of the reference Elm compiler.

Elm compiler error reports Figure 8.13 shows the output of the Elm compiler explaining a syntax error in a simple file. The explanation starts with a review of the construction being analysed by the parser; here, a `let` expression. The token `as` is highlighted as the invalid lookahead symbol, and the parser suggests that in this context, the expected keyword is `in`. Finally, a footnote restates the indentation rules relevant to this particular syntactic setting.

Printing (invalid) Elm

The enumeration of the automaton's sentence fragments generates a list of terminals, which must be transformed into source code for transmission to the Elm compiler. Most terminals are straightforward to represent: keywords are written according to Elm lexical rules (e.g., the `LET` terminal is displayed as `let`), lowercase identifiers are denoted by `lident`, uppercase identifiers by `UIDENT`, and so on. Semantic values are not relevant for our purpose—printing enough code to get a syntax error—so dummy values are used (`1` for an integer, etc.).

```
$ cat Test.elm
module Test.Test exposing (test)

test = let x = 'a' as
       x
$ elm make Test.elm
Detected problems in 1 module.
-- LET PROBLEM ------------------------------------------- src/Test.elm

I was partway through parsing a `let` expression, but I got stuck here:

3| test = let x = 'a' as
                       ^
Based on the indentation, I was expecting to see the `in` keyword next.
Is there a typo?

Note: This can also happen if you are trying to define another value within
the `let` but it is not indented enough. Make sure each definition has
exactly the same amount of spaces before it. They should line up exactly!
```

Figure 8.13: Submitting an invalid input to the Elm compiler

However, since the syntax is indentation-sensitive, proper handling of whitespace is crucial. Most terminals are simply separated by a single space, but we must interpret terminals like `BOL` and `LAYOUT_LEAVE` as constraints on indentation.

A `BOL` simply introduces a newline, positioning the next terminal at the start of the line.

Handling of `LAYOUT_LEAVE` is more intricate. The printer receives a stream of terminals and knows that a `LAYOUT_LEAVE` indicates the end of a layout-sensitive construction, but it has no idea where it started and what was the associated indentation.

We make a simplifying assumption here. Our printer does not handle exiting from nested layout contexts, which is acceptable because the sentences we generate only enter nested layout contexts on the path to an error. In practice, this means that we might miss Elm error messages that depend on an indented construction to be completely recognised in the valid prefix prior to failing. This was not necessary to cover all reduce-filter patterns, so we skipped this case.

With this limitation and since layout contexts only occur after terminals `LET` and `OF`, we remember the column at which the most recent of these terminals ended. A `LAYOUT_LEAVE` terminal is then printed as a newline followed by the appropriate number of spaces to ensure that subsequent tokens align with the stored column.

For instance, `BOL MODULE UIDENT BOL LIDENT EQUAL LET LIDENT EQUAL LIDENT LAYOUT_LEAVE LIDENT EQUAL LIDENT LAYOUT_LEAVE IN LIDENT` is printed:

```
module UIDENT
```

```
lident = let lident = lident
             lident = lident
             in lident
```

The line breaks and indentation constraints are satisfied; however, no additional effort or aesthetic refinement is made.

Enumerating Elm sources

The declarative aspect of the grammar makes it easy to use as a generation device. Since we are interested in the error messages, we use the procedure for enumerating failing configurations. Indentation sensitivity is transparent to this procedure: `BOL` and `LAYOUT_LEAVE` are treated like other terminals and are automatically placed at the appropriate locations. The only difficulty was in printing them.

The core loop of the *fuzzer* proceeds as follows:

- The enumeration procedure from Section 6.2 emits a valid sentence prefix and lists the lookahead terminals that cause it to fail. The sentence prefix is annotated by the reduce-filter patterns that match it.
- A source file is generated for each lookahead by printing the prefix followed by the lookahead.
- The compiler is invoked on each source file. We check that all invocations fail at the expected position. If not, this indicates an error in our grammar and we have to update it. It could also be that the reference compiler did not satisfy the viable prefix property, though this never occurred in practice.
- The error messages are parsed, extracting the core of the message from the variable parts (name of the file, references to positions or identifiers in the sample source code). The core part is used to populate an *error* mapping from an error message to the set of reduce-filter patterns matching sentences that produce this message.

This pass ends when all sentences have been consumed and classified.

The second pass operates by repeatedly identifying the patterns associated with a single message in the *error* mapping.

When a pattern is associated with a single message, it means that all sentences matched by this pattern receive the same error message. We say that it is a “critical pattern” since it uniquely determines the error message received by the sentences. We emit a clause from the pattern and the error message, and remove the message from the mapping. This potentially unlocks new critical patterns, and we repeat the procedure until there are no more.

For the reference Elm compiler, the procedure terminates with an empty set of messages. This indicates that:

- The inferred specification is able to reproduce all syntax error messages.

- The reduce-filter patterns are sufficiently fine-grained to unambiguously pinpoint an error message. If at some point during the procedure, we had pending error messages but no critical pattern, it would mean that some information not captured by a reduce-pattern was used by the Elm compiler to pick an error message.

This outcome provides strong evidence supporting our approach's efficacy, though it does not address all limitations.

First, the precision is inherently limited. Only the sentences in the generated set were tested—by design of the enumeration procedure, it should be representative of the language but this offers no guarantee of completeness beyond reduce-filter patterns. For example, the *complex errors* we wrote for the OCaml language are beyond the recognition power of reduce-filter patterns, which means that (1) the enumeration procedure is unlikely to generate a sentence stressing them, and (2) they would not have a "critical pattern", preventing a clause from being inferred.

Second, while we can cover all error messages generated from our test set, exhaustive coverage of errors is not guaranteed. The enumeration procedure produces sentences sufficient for covering all reduce-filter patterns but the subset of critical patterns chosen to associate with specific error messages might not cover all cases. This was not an issue in our (Mini-)Elm case study, nor did we further investigate this possibility.

If it becomes problematic for another grammar, it should be possible to refine the enumeration procedure to generate sentences avoiding the critical patterns. The new sentences lead to a broader and more complete set of test cases, and the procedure is repeated until exhaustivity is achieved. (The procedure terminates since there is a finite set of reduce-filter patterns to begin with, and each iteration covers at least one new pattern).

Chapter 9

Future work

9.1 Tooling: importing Menhir error messages

When porting Catala error handling from Menhir error messages to an LRGrep error specification, we could automate most of the translation but not the ordering of clauses. We proceeded by trial and error, refining the order until our specification reproduced the original behaviour.

This is not a fundamental problem though: it is possible, in principle, to automate the translation of most error messages to error specifications. We say most because two cases can be problematic:

- Having two distinct messages for two LR(1) states with the same LR(0) core.
- Not being able to translate certain `%on_error_reduce` directives because the reduction operators are too coarse-grained.

These problems can be worked around, possibly at the cost of making patterns more complex, but more work is required to give a definitive answer to these questions and to automate the process.

9.2 Alternative operator: hypothetical future

The present work focused on characterising LR stacks using the "filter" and "reduce" operators. Filter identifies a situation based on the immediate past and future, expressed as an LR(0) item. Reduce applies reductions to generalise a situation before further analysis.

Very early, we observed that composing both operators often led to some redundancies, as in the $[E\,/\,F \to (E\cdot)]$ example. The fact that we are looking for stacks ending with E is specified twice: first for reducing, then as part of matching the immediate past. This is effectively redundant because the only way for $/\,F \to (E\cdot)$ to succeed is to reduce E first. We can even let LRGrep infer that by rewriting the pattern as $[\_^*\,/\,F \to (E\cdot)]$. This new formulation can be read as "situations matching $F \to (E\cdot)$,

for any sequence of reductions", and LRGrep will find that the only solution is when the placeholder _* is instantiated to E.

A compelling alternative could be a *hypothetical future* operator taking a sequence of symbols and which matches prefixes for which these symbols constitute a valid continuation. This generalises the filter operator by looking for arbitrarily long suffixes and not only following immediate transitions (shifts and gotos) but also allowing reductions, as long as they permit the sequence to match.

If we write $\#\alpha$ for the *hypothetical future* α, our two introductory examples become as simple as $\#i$, looking for an integer, and `#)`, looking for a closing parenthesis. The richer closing pattern parenthesis can be written as $l = (E\#)$. For these simple cases, the specification seems very natural and should be intuitive for grammar authors, even if they are not familiar with the intricacies of LR parsing.

But it is not clear if this generalises as well as our current approach. The complex OCaml example where we want to detect when a `;` changed the meaning of a 'let' and inform the user can, for instance, be rewritten as:

$$(\texttt{\#LET});\ \mathit{semi} = \texttt{SEMI};\ \texttt{LET};\ \texttt{\_}^*;\ \texttt{\#IN}$$

The right part tells us that we are looking for a situation where we expect an `IN` prefixed by any construction beginning with a `LET` after a semi-colon, which could be omitted: before this semi-colon, `LET` was a valid hypothetical future (the parentheses are used to avoid an ambiguity in notation).

Compare this to the current pattern using reductions:

$$\texttt{[structure\_item]};\ \mathit{semi} = \texttt{SEMI};\ \texttt{[let\_bindings(ext)]}$$

The new operator can avoid referring to the nonterminals `structure_item` and `let_bindings(ext)`, and use a notation closer to a "lexical" regular expression for matching the situation. This would be simpler for a beginner user; however, a grammar author should already be familiar with the nonterminals they use in their grammar, and the interpreter facility can helpfully remind them of the nonterminals relevant to a specific situation. However, we also lose some precision here: the loose specification `LET;_*;#IN` could accidentally match more than expected, for instance when multiple `LET` are nested on the stack.

Still, we feel that it is worth exploring alternative ways for characterising stacks, especially if they make simple and common situations simpler to specify.

9.3 Reduction graph and syntactic completions

Sasano and Choi (2021) introduce a framework for suggesting syntactically valid completions of the current context by dynamically exploring the reductions allowed by the grammar.

They start by generating a set of *simple candidates*, made up of the items reachable from a stack after following any possible sequence of reductions. By iterating the procedure, they get *nested candidates*: for each simple candidate, they simulate

the impact of the possible completions on the stack (by appending the suffix of the item to the stack prefix) and generate a new set of simple candidates from the new stack.

This approach is entirely dynamic, simulating all reductions at runtime. They use two heuristics to keep this method responsive:

1. One to cut off the work before looking too deep in the stack.
2. One to avoid expansions growing the stack too much by exploring a given branch too deeply (e.g., suggesting $+i+i+i+\ldots$ as a completion for i).

We suspect that our reduction automata could be a helpful addition to this approach. The set of *simple candidates* is trivial to compute from the reduction automaton, in a time linear to the depth of the LR stack. For nested candidates, we conjecture that the reduction automaton could be extended to produce a finite description of the set of all candidates, providing an exact solution rather than relying on two heuristics.

Simulating the completion of all items can be achieved by adding transitions $\texttt{Suffix}(s, \vec{s}', T'') \xrightarrow{\epsilon} \texttt{Reduce}(s, \vec{s}', T, A, \alpha)$ for all $A \to \alpha \cdot \beta \in \textbf{items}(\text{top}(s\vec{s}')), \alpha \neq \epsilon$ to the reduction automaton. For each item, the α part represents a prefix that was already recognised, and β a candidate for completion.

Given a stack $\vec{s}$, the nested candidates can be constructed by following all paths labelled $\vec{s}$ (from right to left) and, along each path, collecting the β parts of the reductions visited.

9.4 Application to GLR

GLR extends the recognition power of LR parsers to all context-free grammars by allowing non-deterministic actions. Conflicts are kept and, at runtime, conflicting actions are explored in parallel: the stack is cloned, and the different continuations are explored independently. The parser configuration consists of a set of stacks rather than a single stack, with a clever representation that is compact and shares as much work as possible between the different continuations (Tomita, 1991).

For each input token, all stacks are advanced in a lockstep fashion. Parsing succeeds when at least one of the stacks accepts the input, though multiple ones might do so, and the result is generally a set of semantic values issued from a forest of derivation trees.

If one of the stacks rejects a token, it is removed from the set. Parsing fails if the set of stacks ends up empty. The configuration immediately preceding the failure—the one that we would like LRGrep to analyse to produce an error message—contains a non-empty set of stacks.

LRGrep as it is should adapt quite seamlessly to this setting by treating each stack independently:

- The compilation strategy and, more generally, the analysis of reductions are not affected by the move to GLR; even though there is a set of stacks, an individual stack is still a well-formed LR stack. That multiple actions might apply at any given time might lead to marginally larger graphs but does not fundamentally change the analyses.

- Matching multiple stacks leads to multiple error message candidates, though the existing candidate ranking mechanism (using textual precedence) should extend reasonably well to this setting. If the highest ranked candidate matches multiple stacks, picking one arbitrarily leads to selecting the same error message, though the semantic values might be affected.

However, the current LRGrep is completely oblivious to correlations between the different stacks. It can happen that two or more stacks, even if they are treated independently by the parser, can only appear or disappear from the set of active stacks at the same time. Although the example is a bit contrived, consider an arithmetic grammar not enforcing the associativity of the + operator. For every nested occurrence of +, a GLR parser will consider both right-associative and left-associative interpretations. The set of stacks will grow exponentially in the number of +, but the correct language will be recognised.

In fact, past the initial divergence, the suffix of the input is interpreted identically, and this high correlation allows a GLR parser to handle this case efficiently.

LRGrep cannot be statically aware of this correlation, and it is likely impossible to properly account for it. GLR parsers can handle arbitrary CFGs. In general, establishing static correlations between the stacks occurring in non-deterministic parsing would allow reasoning about ambiguities, which is known to be undecidable for CFGs. In LRGrep, this will mainly manifest in the exhaustive coverage analysis. For instance, if we provide an error message focusing specifically on the left-associative interpretation of +, the coverage analysis will complain that there is no error message when a stack that followed the right-associative interpretation fails. Even though, at runtime, there will always be a left-associative stack in the set from which to produce an error message.

So, in the GLR setting, we have to give up the soundness of the exhaustivity analysis and accept some false negatives. We have not experimented with this, but it is unlikely to be a problem in practice, especially for realistic grammars and not contrived examples that are purposefully ambiguous.

On the other hand, if a case can be made that accounting for the correlation between stacks allows for significantly improving error messages in specific situations, it should be possible to extend the matching language with an operator for parallel matching of stacks. For instance, with a pattern $p_1 \& p_2$ that succeeds when one stack of the set matches p_1 and another matches p_2. This would significantly complicate the run-time matching strategy but we suspect the impact on the rest of the theory to be quite contained. Such a pattern would be out of reach of the coverage analysis.

It is worth mentioning Brunsfeld et al. (2020), a full-featured implementation

of GLR with incremental and error-resilient parsing, supported by GitHub. It has received a lot of attention lately and is becoming a solution of choice for providing syntax highlighting and other syntax-directed services to programmers' editors. Adapting LRGrep to Tree-sitter runtime would allow providing detailed syntax-error messages in these contexts, benefiting a much wider audience than the present experiment.

9.5 Imperative Implementation

An LRGrep matching program expects to be fed the stack as it was after the last shift transition, prior to any reduction (a configuration in which the current state is what we called a **Wait** state). This is trivially the case for canonical parsers which only reduce when it is known that this will lead to a valid outcome—eventually shifting the lookahead token or accepting the input. However, practical LR parsers resort to optimisations that result in reductions being applied before the validity of the lookahead token can be determined:

- Default reductions, which are independent of the lookahead symbol and can be applied even before the parser is fed the next lookahead token.
- "Fall back" or "spurious" reduction actions used to make the transition table sparser; these reductions are safe because even if it is not known at the time of the reduction whether the lookahead token is valid, it will be checked again after reducing; the worst that can happen is that an invalid lookahead symbol is rejected after a few extra reductions.

These two optimisations conflict with the LRGrep approach, as they damage the stack. This is a serious problem in practice because the main implementations of LR parsers are imperative and use a destructive implementation of the stack for performance reasons.

We do not have a clear answer to this problem, but we have two approaches in mind which should have a limited impact on performance:

- Remembering the damage done to the stack by keeping a scratch buffer while reducing and, in case of an error, applying LRGrep to the scratch buffer first (and switching to the main stack once reaching the end of the damaged suffix).
- Not applying reductions on the stack but instead preparing an "action plan" on the side, and committing it only when determining that the input token is valid; if it is not, the undamaged stack can be sent to LRGrep.

Both approaches should be reasonably efficient, but actual performance comes down to implementation details and microarchitectural considerations: remembering the damage translates to extra stores, but they do not introduce new dependencies and do not affect control flow; preparing an action plan should fit into a tight loop that does not need to call external code (e.g., no semantic actions), but this

loop depends heavily on the transition table and lookups might cause stalls and offer fewer opportunities for instruction-level parallelism than interleaving transitions with semantic actions.

As usual, ϵ-rules complicate matters: without them, the stack decreases monotonically during reductions; with ϵ-rules, the stack can temporarily grow. Fortunately, this can be worked around by performing an ϵ-closure of the involved reductions during compilation, as is done when constructing the reduction automaton. An LR parser generator tuned for high performance should have similar facilities that could be leveraged when integrating LRGrep.

Conclusion

This dissertation challenges the notion that LR parsers are less capable than alternatives for reporting accurate syntax errors for programming languages. Our central thesis posits that the LR formalism not only enables efficient parsing of practical languages but also retains decidable properties that make it suitable for dynamic and static analysis of syntax errors. I introduce a dialect of regular expressions for specifying and handling these errors, demonstrating the potential of LR parsers to offer precise and maintainable error messages.

The core components of this DSL—filters and reductions—allow for fine-grained classification and generalisation of grammatical contexts, enabling the identification of specific positions within parsing rules while abstracting over specific nonterminals. This classification is applied to the stack of a failing LR parser to select an error message. The efficient recognition of these constructs through compilation to finite automata shows that our approach is amenable to a compact and performant implementation.

After recognition, I tackled the problem of statically reasoning about syntax errors. Addressing the challenge posed by conflict resolution (which removes certain transitions and actions in an LR parser), I have developed an efficient reachability analysis that identifies the reachable configurations of LR parsers. This enables the complete and sound analysis of LR stacks and failing configurations, with two practical applications: enumeration of erroneous sentences and checking the exhaustive coverage of syntax errors.

I implemented these compilation and static analysis schemes and validated them with three case studies:

1. Enhancing the syntax error reporting capabilities of OCaml's reference parser.

2. Porting Catala's error handling mechanisms from Menhir to LRGrep, achieving compatibility with a semi-automatic import of Menhir's example-based error message specifications.

3. Replicating Elm's renowned syntax error reports, using our tools to infer an error specification while using the reference implementation as a black box.

These case studies validate the effectiveness of our proposed methods and illustrate their broad applicability across different programming languages and environments, paving the way for improved parser design and syntax error handling in future language implementations. The findings herein advocate for a reevaluation

of LR parsers as valuable tools not only for parsing efficiency but also for sophisticated error reporting, bridging a gap between the formalism and the practical needs of compiler developers.

Bibliography

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Aho, A. V. and Ullman, J. D. (1972). *The theory of parsing, translation, and compiling*. Prentice Hall.

Antimirov, V. (1996). Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319.

Barik, T., Ford, D., Murphy-Hill, E., and Parnin, C. (2018). How should compilers explain problems to developers? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 633–643, New York, NY, USA. Association for Computing Machinery.

Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P., Pearce, J. L., and Prather, J. (2019). Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, pages 177–210.

Berry, G. and Sethi, R. (1986). From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126.

Borsotti, A. and Trafimovich, U. (2022). A closer look at TDFA. arXiv:2206.01398 [cs].

Bouajjani, A., Esparza, J., and Maler, O. (1997). Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory (CONCUR)*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer.

Bour, F. and Pottier, F. (2021). Faster reachability analysis for LR(1) parsers. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, pages 113–125, Chicago IL USA. ACM.

Bour, F., Refis, T., and Scherer, G. (2018). Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages*, 2(ICFP):103:1–103:15.

Brunsfeld, M., Thomson, P., Hlynskyi, A., et al. (2020). Tree-sitter.

Brzozowski, J. A. (1964). Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494.

Chistikov, D., Majumdar, R., and Schepper, P. (2022). Subcubic certificates for CFL reachability. *Proc. ACM Program. Lang.*, 6(POPL):41:1–41:29.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Coquand, T. and Siles, V. (2011). A Decision Procedure for Regular Expression Equivalence in Type Theory. In Jouannaud, J.-P. and Shao, Z., editors, *Certified Programs and Proofs*, pages 119–134, Berlin, Heidelberg. Springer.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms (Third Edition)*. MIT Press.

Czaplicki, E. and Chong, S. (2013). Asynchronous functional reactive programming for guis. *ACM SIGPLAN Notices*, 48(6):411–422.

Denny, J. E. and Malloy, B. A. (2010). The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution. *Science of Computer Programming*, 75(11):943–979.

DeRemer, F. and Pennello, T. (1982). Efficient computation of $LALR(1)$ look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649.

DeRemer, F. L. (1969). Practical translators for LR(k) languages. Technical Report MIT-LCS-TR-065, Massachusetts Institute of Technology.

DeRemer, F. L. (1971). Simple $LR(k)$ grammars. *Communications of the ACM*, 14(7):453–460.

Diekmann, L. and Tratt, L. (2020). Don't panic! better, fewer, syntax errors for LR parsers. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 166 of *LIPIcs*, pages 6:1–6:32.

Dijkstra, E. W. (1959). A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271.

Donnelly, C. and Stallman, R. (2021). *Bison*.

Driesen, K. (1999). Software and Hardware Techniques for Efficient Polymorphic Calls. page 204.

Esparza, J., Hansel, D., Rossmanith, P., and Schwoon, S. (2000). Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer.

Filliâtre, J.-C. and Conchon, S. (2006). Type-safe modular hash-consing. In *ACM Workshop on ML*, pages 12–19.

Gallier, J. and Quaintance, J. (2021). Introduction to the theory of computation languages, automata and grammars some notes for cis511.

Glushkov, V. M. (1961). The Abstract Theory of Automata. *Russian Mathematical Surveys*, 16(5):1–53.

Grune, D. and Jacobs, C. J. H. (2008). *Parsing techniques: a practical guide, second edition*. Monographs in computer science. Springer.

Hashimoto, M. (2021). The Code Continuity Analysis Framework. https://github.com/codinuum/cca.

Hopcroft, J. E. (1971). An $n \log n$ algorithm for minimizing states in a finite automaton. In Kohavi, Z. and Paz, A., editors, *Theory of Machines and Computations*, pages 189–196. Academic Press.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2000). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Jacobs, J. and Wissmann, T. (2022). Fast Coalgebraic Bisimilarity Minimization. arXiv:2204.12368 [cs].

Jeffery, C. L. (2002). *Merr User's Guide*.

Jeffery, C. L. (2003). Generating LR syntax error messages from examples. *ACM Transactions on Programming Languages and Systems*, 25(5):631–640.

Johnson, S. C. (1975). Yacc: Yet another compiler-compiler. Computing Science Technical Report 32, Bell Laboratories.

Kameda, T. and Weiner, P. (1970). On the State Minimization of Nondeterministic Finite Automata. *IEEE Transactions on Computers*, C-19(7):617–627.

Kildall, G. A. (1973). A unified approach to global program optimization. In *Principles of Programming Languages (POPL)*, pages 194–206.

Knuth, D. E. (1965). On the translation of languages from left to right. *Information & Control*, 8(6):607–639.

Knuth, D. E. (1977). A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5.

Korte, B. and Vygen, J. (2012). *Combinatorial Optimization: Theory and Algorithms*. Springer Publishing Company, Incorporated, 5th edition.

Koutris, P. and Deep, S. (2023). The fine-grained complexity of cfl reachability. *Proc. ACM Program. Lang.*, 7(POPL).

Lal, A. and Reps, T. W. (2006). Improving pushdown system model checking. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 343–357. Springer.

Lang, B. (1974). Deterministic Techniques for Efficient Non-Deterministic Parsers. In Loeckx, J., editor, *Automata, Languages and Programming*, pages 255–269, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lee, T. T. (1981). Order-preserving representations of the partitions on the finite set. *Journal of Combinatorial Theory, Series A*, 31(2):136–145.

Lehmann, D. J. (1977). Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76.

Leroy, X. (2021). The CompCert C compiler. http://compcert.org/.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. (2019). The OCaml system: documentation and user's manual.

Li, X. and Ogawa, M. (2010). Conditional weighted pushdown systems and applications. In *ACM Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 141–150.

Mamouras, K. and Chattopadhyay, A. (2024). Efficient Matching of Regular Expressions with Lookaround Assertions. *Proceedings of the ACM on Programming Languages*, 8(POPL):2761–2791.

Merigoux, D., Chataing, N., and Protzenko, J. (2021). Catala: a programming language for the law. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29.

Midtgaard, J., Nielson, F., and Nielson, H. R. (2016). A Parametric Abstract Domain for Lattice-Valued Regular Expressions. In Rival, X., editor, *Static Analysis*, volume 9837, pages 338–360. Springer Berlin Heidelberg, Berlin, Heidelberg. Series Title: Lecture Notes in Computer Science.

Minamide, Y. and Mori, S. (2012). Reachability analysis of the HTML5 parser specification and its application to compatibility testing. In *Formal Methods (FM)*, volume 7436 of *Lecture Notes in Computer Science*, pages 293–307. Springer.

Moseley, D., Nishio, M., Perez Rodriguez, J., Saarikivi, O., Toub, S., Veanes, M., Wan, T., and Xu, E. (2023). Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1026–1049.

Nederhof, M.-J. and Sarbo, J. J. (1996). Increasing the Applicability of LR Parsing. In Bunt, H. and Tomita, M., editors, *Recent Advances in Parsing Technology*, volume 1, pages 35–57. Springer Netherlands, Dordrecht. Series Title: Text, Speech and Language Technology.

Niehren, J., Sakho, M., and Al Serhali, A. (2022). Schema-Based Automata Determinization. *Electronic Proceedings in Theoretical Computer Science*, 370:49–65.

Owens, S., Reppy, J., and Turon, A. (2009). Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2):173–190.

Pager, D. (1977). A practical general method for constructing $LR(k)$ parsers. *Acta Informatica*, 7:249–268.

Paige, R. and Tarjan, R. E. (1987). Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989.

Paxson, V., Estes, W., and Millaway, J. (2007). Lexical analysis with flex. *University of California*, page 28.

Pottier, F. (2009). Lazy least fixed points in ML. Unpublished.

Pottier, F. (2016). Reachability and error diagnosis in LR(1) parsers. In *Compiler Construction (CC)*, pages 88–98.

Pottier, F. and Régis-Gianas, Y. (2005–2021). The Menhir parser generator. https://gitlab.inria.fr/fpottier/menhir/.

Pratt, V. R. (1973). Top down operator precedence. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73*, pages 41–51, Boston, Massachusetts. ACM Press.

Rabin, M. O. and Scott, D. (1959). Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3(2):114–125.

Reps, T. W., Schwoon, S., Jha, S., and Melski, D. (2005). Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1-2):206–263.

Rideau, L., Serpette, B. P., and Leroy, X. (2008). Tilting at windmills with Coq: formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning*, 40(4):307–326.

Régis-Gianas, Y., Jeannerod, N., and Treinen, R. (2020). Morbig: A Static parser for POSIX shell. *Journal of Computer Languages*, 57:100944.

Sasano, I. and Choi, K. (2021). A text-based syntax completion method using LR parsing. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 32–43, Virtual Denmark. ACM.

Suwimonteerabuth, D., Schwoon, S., and Esparza, J. (2006). Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In *Automated Technology for Verification and Analysis (ATVA)*, volume 4218 of *Lecture Notes in Computer Science*, pages 141–153. Springer.

Talts, S. (2019). stanc3: rewriting the Stan compiler. https://statmodeling.stat.columbia.edu/2019/03/13/stanc3-rewriting-the-stan-compiler/.

Tarjan, R. E. and Yao, A. C.-C. (1979). Storing a sparse table. *Communications of the ACM*, 22(11):606–611.

Thiemann, P. (2016). Derivatives for Enhanced Regular Expressions. *arXiv:1605.00817 [cs]*. arXiv: 1605.00817.

Thompson, K. (1968). Programming Techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.

Tomita, M., editor (1991). *Generalized LR Parsing*. Springer US, Boston, MA.

Trofimovich, U. (2019). Tagged Deterministic Finite Automata with Lookahead. *arXiv:1907.08837 [cs]*. arXiv: 1907.08837.

Valmari, A. (2012). Fast brief practical DFA minimization. *Information Processing Letters*, 112(6):213–217.

Wikipedia (2021a). Hash-consing. https://en.wikipedia.org/wiki/Hash_consing.

Wikipedia (2021b). *k*-way merge algorithm. https://en.wikipedia.org/wiki/K-way_merge_algorithm.

Wikipedia (2021c). Matrix chain multiplication. https://en.wikipedia.org/wiki/Matrix_chain_multiplication.

Wrenn, J. and Krishnamurthi, S. (2017). Error messages are classifiers: a process to design and evaluate error messages. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 134–147, Vancouver BC Canada. ACM.

Zimmerman, J. (2022). langcc: A Next-Generation Compiler Compiler.

Appendix A

A greedy solution for parallel moves

We propose a translation scheme to implement parallel moves with the instruction set of the abstract machine from Section 4.6. The code produced uses the `Swap` instruction and is linear in the size of the (graph of the) transfer function. It is not optimal as it can generate some unnecessary temporary moves, but it is simple and sufficient for our needs. It emits instructions that realise a finite map $f : [0; m[\to [0; n[\cup \{\top, \bot\}$ that describes register transfers and updates, for some $m, n \in \mathbb{N}$ (introduced in Section 4.4.3).

We split the work into two passes: the first to handle shuffling of registers, and the second one to implement setting, duplicating and clearing the other registers.

In the intermediate steps, some registers might have been shuffled from their original position. The main idea is to maintain an auxiliary array `aux` of n integers to record where the original value of a register can be found. We also use two arrays of n booleans: `mark` to track if a register value is used at least once and `dup` for register values used more than once.

Initialisation code is provided in Figure A.1. First, `aux` is set such that all register values are at their original position; `mark` and `dup` are cleared.

The transfer function is scanned to determine which register values are needed once or more.

After that we can emit instructions:

- The first pass in Figure A.2 moves values used at least once to their first target register.
- The second pass in Figure A.3 updates the remaining registers, copying their value from the first target for values used multiple times, and updating or clearing the other ones.

```
var aux: array[n] of integer
    mark, dup: array[n] of boolean
for i = 0 to n do
  aux[i] = i
  mark[i] = false
  dup[i] = false
done
```

```
for i = 0 to m do
  let j = f(i)
  if j ∈ ℕ then
    if not mark[j] then
      mark[j] = true
    else dup[j] = true
  end
done
```

Figure A.1: Initialisation

```
for i = 0 to m do
  let j = f(i)
  if j ∈ ℕ and mark[j] then
    mark[j] = false
    j = aux[j]; Find the actual position of the value
    if i != k then ; The value is not in the expected position
      if i <= n and (mark[i] or dup[i]) then ; Value i is needed later:
        emit Swap(i, j)  ; Swap it
        aux[i] = j       ; Remember the new position of value i
      else
        emit Move (i, j) ; Overwrite register i
      end
      aux[j] = i; Remember the position of value j
    end
  end
done
```

Figure A.2: First pass: move values used at least once to their first target register

```
for i = 0 to m do
  let j = f(i)
  if j ∈ ℕ then
    if not mark[j] then
      mark[j] = true; Skip the first occurrence
    else
      j = aux[j]; Find the actual position of the value
      emit Move (i, j)
    end
  else if j = ⊥ then emit Clear i
  else if j = ⊤ then emit Store i
  end
done
```

Figure A.3: Second pass: set, clear or duplicate remaining registers