BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

GRAMMAR-BASED TRANSLATION FRAMEWORK

PŘEKLADOVÝ FRAMEWORK ZALOŽENÝ NA GRAMATIKÁCH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. RADEK VÍT**
AUTOR PRÁCE

SUPERVISOR **prof. RNDr. ALEXANDER MEDUNA, CSc.**
VEDOUCÍ PRÁCE

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

Master's Thesis Specification

Student: **Vít Radek, Bc.**
Programme: Information Technology Field of study: Information Systems
Title: **Grammar-Based Translation Framework**
Category: Compiler Construction
Assignment:

1. Study translation grammars and their restricted versions. Consult this study with your advisor.
2. Based upon instructions given by your advisor, study algorithms that make use of translation grammars. Focus the study on parsing and translation algorithms, including the algorithms described in the literature below.
3. Inspired by the study in 2, develop new and original translation algorithms. Carefully consult this development with your advisor.
4. Design a translation framework consisting of several translation algorithms designed in 3. Compare its properties with the properties of similar well-known software tools, such as Bison, ANTLR, or PLY.
5. Implement and test the framework designed in 4.
6. Design and implement a syntax-directed translator by using the framework from 5.
7. Summarize the results. Discuss the future investigation concerning this project.

Recommended literature:

- Meduna, A.: Automata and Languages, Springer, London, 2000, ISBN 978-1-85233-074-3
- Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition), Pearson Education, 2006, ISBN 0-321-48681-1
- Meduna, A.: Elements of Compiler Design, New York, Taylor & Francis, 2008
- Denny, J.E., and Malloy, B.A.: IELR(1): Practical LR(1) Parser Tables for Non-LR(1) Grammars with Conflict Resolution. Science of Computer Programming 75(2010), 943-979
- Levine, J.: flex & bison, O'Reilly Media, 2009, ISBN 978-0-596-15597-1

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Meduna Alexander, prof. RNDr., CSc.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 22, 2019
Approval date: October 31, 2018

Abstract

In this thesis, we examine existing parsing algorithms for context-free grammars. Based on these existing algorithms, we design a new model for representing LR automatons and we define a new parsing algorithm LSCELR based on that model. We modify parsing algorithms to create translation algorithms based on translation grammars. We define attribute translation grammars, an extension of translation grammars for defining the relationships between input and output symbols in translation. We implement a translation grammar-based framework ctf that implements the new parsing algorithm. We define a language for describing attribute translation grammars and implement a translator that creates source representation of these grammars for the implemented framework.

Abstrakt

V této práci prozkoumáváme existující algoritmy pro přijímání jazyků definovaných bezkontextovými gramatikami. Na základě těchto znalostí navrhujeme nový model pro reprezentaci LR automatů a s jeho pomocí definujeme nový algoritmus LSCELR. Modifikujeme algoritmy pro přijímání jazyků k vytvoření algoritmů pro překlad založený na překladových gramatikách. Definujeme atributové překladové gramatiky jako rozšířené překladové gramatiky pro definici vztahů mezi vstupními a výstupními symboly překladu. Implementujeme překladový framework ctf založený na gramatikách, který implementuje překlad pomocí LSCELR. Definujeme jazyk pro popis atributových překladových gramatik a implementujeme překladač pro překlad této reprezentace do zdrojového kódu pro implementovaný framework.

Keywords

formal languages, translation grammars, context-free grammars, translation, attribute translation grammars

Klíčová slova

formální jazyky, překladové gramatiky, bezkontextové gramatiky, překlad, atributové překladové gramatiky

Reference

VÍT, Radek. *Grammar-Based Translation Framework*. Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. RNDr. Alexander Meduna, CSc.

Rozšířený abstrakt

V této diplomové práci navrhujeme překladový framework založený na gramatikách. Nejdříve zkoumáme existující algoritmy pro přijímání jazyků definovaných bezkontextovými gramatikami. Sledujeme deterministické algoritmy, které přijímají vstup zleva doprava a nahlíží vždy na jeden následující symbol. Zkoumáme fungování algoritmu pro generování minimálních LR(1) parserů IELR, který je implementovaný v existujícím generátoru parserů GNU Bison. Představujeme také modifikaci LR algoritmů pro řešení konfliktů v tabulkách pro LR(1) parsing.

Na základě těchto existujících algoritmů navrhujeme nový model pro reprezentaci LR automatů. Tento nový model ukládá pro každý svůj stav generované lookahead symboly pro své položky, a ukládá také zdroje pro ostatní lookehead symboly. Tyto informace zachovávají proti běžnému modelu více informací o struktuře automatu a umožňují jednoduchou analýzu propagace těchto symbolů. Tento model také umožňuje jednoduché změny v automatu díky implicitní změně lookahead symbolů při budoucích výpočtech jejich množin. S pomocí tohoto upraveného modelu navrhujeme nový algoritmus pro generování minimálních LR(1) parserů LSCELR. Tento algoritmus využívá vlastností tohoto modelu pro výpočet konfliktů v automatu a rozdělení některých jeho stavů. LSCELR dokáže stejně jako IELR generovat validní parsery i pro gramatiky, které spoléhají na specifické řešení konfliktů pro svoje použití, kde by LALR parsery mohly zmenšit množinu přijímaných jazyků. V této práci popisujeme algoritmy pro vytvoření LSCELR parserů a na vybraných gramatikách demonstrujeme jejich vlastnosti.

V této práci definujeme překladové gramatiky jako prostředky pro generování překladů. Definujeme také atributové překladové gramatiky pro definici překladů v praktickém využití, kde formálně definujeme vztahy mezi atributy vstupních a výstupních terminálů v jednotlivých přepisovacích pravidlech gramatiky. Modifikujeme známé algoritmy pro přijímání jazyků na algoritmy pro překlad, konkrétně prediktivní překlad shora dolů a LR překlad sdola nahoru. Volíme dvouzásobníkový automat jako model pro překlad definovaný překladovými gramatikami. V této práci také představujeme důkaz generativní síly překladových gramatik. Při omezení výstupního jazyka lze využít lineární překladové gramatiky jako prostředek pro generování jazyků se sílou Turingova stroje. Důkaz této síly je založený na simulaci frontových gramatik, které mají stejnou generativní sílu jako Turingovy stroje.

V poslední části práce popisujeme implementaci překladového frameworku na základě algoritmů představených v předchozích částech této práce. Implementovaný framework využívá atributové překladové gramatiky pro definice překladu a využívá LSCELR jako algoritmus pro překlad. Popisujeme rozdíl mezi tradičním přístupem k překladu založeném na bezkontextových gramatikách a představujeme přístup k překladu založeném na překladových gramatikách. Práce předkládá náročné aspekty vývoje výstupních jazyků atributových překladových gramatik pro praktické využití. Představujeme také způsob pro kompresi tabulek pro LR algoritmy, které za cenu horší složitosti přístupu k položkám výrazně snižují paměťové nároky pro reprezentaci všech stavů LR algoritmu v paměti. Popisujeme jazyk pro definici atributových překladových gramatik a asociativitu a precedenci jejich terminálů a pravidel. S pomocí implementovaného frameworku implementujeme nástroj, který překládá tuto reprezentaci do zdrojových kódů pro náš framework.

Pro výslednou implementaci frameworku ve zkratce představujeme způsob reprezentace některých konceptů a požadavky na uživatele tohoto frameworku. Zmiňujeme nedostatky současné implementace a navrhujeme úpravy a vylepšení pro budoucí práce.

Grammar-Based Translation Framework

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of prof. RNDr. Alexander Meduna, CSc. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Radek Vít
May 20, 2019

Acknowledgements

I would like to thank professor Meduna for his mentorship and excellent teaching skills. He made me approach my work on this thesis with care and supported me through the writing process. I would like to thank my colleague, Tomáš Ženčák, for making himself available whenever an outside opinion was needed. I would like to thank my parents for supporting me and giving me a loving home. Achieving this would be impossible without them. Finally, I would like to thank my partner, Mária Adamská, for enduring the time I spent on my computer writing this thesis instead of spending more time with her.

Contents

Chapter 1

Introduction

Translation is an important part of the computer science world; all programming languages have to be translated to machine instructions or to another interpreted language. Even interpreted languages must first be translated to a form that a virtual machine can easily execute. Many storage and serialization formats and various protocol adapters utilize translation as well.

Many translation tools today are based on context-free grammars. They provide a formal specification of the input context-free language, and there exist many algorithms that let us parse the input string and determine whether it's properly formed according to that grammar. Most translation algorithms today only formally specify their input and create their output informally by manually constructing syntax trees upon applying individual productions. In other words, we often formally define what we translate, but leave the specification of the output to the implementation.

In this thesis, we propose and implement a slightly different approach. We define and use attribute translation grammars to formally define both input and output of translations. This way, we are able to reason about the results we get when using modified versions of known parsing algorithms that produce output defined by these grammars. Instead of producing syntax trees based on productions we use, we directly translate our input to a formally defined output. When we use this approach, we are free to use different parsing algorithms without them affecting the design of the rest of our translation tools, since we only need to process the output string. This brings a new set of challenges: we must be able to define usable output formats and then be able to process them efficiently.

We focused our studies on existing parsing algorithms. LR bottom-up parsing represents one of the most powerful approaches to parsing, and there exist many modifications of this algorithms that are used by many existing tools. LR parsers are one of the least limiting for grammar designers and are widely used today. In this thesis, we were able to design an original LR parsing algorithm that is both powerful and memory efficient. We achieved this by modifying the underlying automaton model that LR parsers use and by using this new model to analyze and modify the automaton to create a minimal LR(1) parser.

This master's thesis is a continuation of my bachelor's thesis (see [17]), where some of the concepts we'll be looking at here were first introduced. Parts of the existing concepts have already been covered in this thesis' term project (see [18]). In this thesis, we design an original minimal LR(1) parsing algorithm and an extension of translation grammars for practical use. We also heavily modify the translation framework first implemented for my bachelor's thesis to implement the new translation algorithms and simplify its usage.

In Chapter 2, we discuss some basic concepts from discrete math and language theory. We introduce sets and relations, alphabets and languages, grammars and other language generating and accepting devices. Some prior knowledge of these concepts is required for full understanding of this thesis, as most of these concepts are introduced only briefly as formal definitions.

In Chapter 3, we introduce existing parsing methods. We discuss top-down parsing methods, namely recursive descent parsing and LL parsing. These parsing methods attempt to parse the input by expanding nonterminals so that they recreate the input according to the read symbols. We also take a look at bottom-up LR parsing. We examine existing variants of LR parsing, such as canonical LR, LALR, and IELR parsing. We also discuss conflict resolution methods for LR parsing algorithms and their importance for practical parsing and we briefly examine the traditional construction of syntax trees and transferring attributes through these syntax trees. We briefly discuss the traditional approach to translation using these parsing algorithms.

In Chapter 4, we introduce LSCELR parsing. LSCELR is an original minimal LR(1) parsing algorithm conceptually based on the IELR algorithm. The originality of LSCELR lies in its new automaton model. We explicitly track lookahead dependencies between states and their items, which lets us directly reason about the properties of the automaton without needing to perform additional analysis of the relationships between states. LSCELR generates minimal LR(1) parsers by using this explicit lookahead source model for conflict analysis and generating additional states. Like IELR, LSCELR can generate parsers for ambiguous grammars with conflict resolution without changing the size of the accepted language compared to canonical LR(1) parsers. The main advantage of LSCELR over IELR is the transparency of the approach, making it potentially easier to understand and teach. We also demonstrate how to obtain some traditional LR parsers using our new automaton model.

In Chapter 5, we discuss translation grammars and their properties. We introduce them as a model for formal translation and introduce algorithms for both top-down and bottom-up translation based on well established parsing algorithms. We define attribute translation grammars as an extension of translation grammars to formally define the relationships between the input and output attributes in individual productions. This extension allows us to use translation grammars in practical algorithms where we need to consider attributes of input symbols in addition to their identity. We also discuss the properties of translation grammars as language generation devices. By restricting the accepted output languages of specific linear translation grammars, we are able to generate recursively enumerable languages. We provide a proof of their language generation power by simulating queue grammars.

Finally in Chapter 6, we design a translation framework based on translation grammars and the original parsing algorithms introduced in Chapter 4. We define a language for describing attribute translation grammars. This will allow the users of our framework to define translations conveniently. We demonstrate the differences between traditional context-free grammar-based tools and the proposed approach to translation based on attribute translation grammars. We propose a method of compressing LR parsing tables to reduce the space requirements for representing parsing tables in memory. We discuss some implementation details and the application program interface of the translation framework. We also provide an overview of the tools implemented for this framework and we examine the approach to translation using this framework by taking a closer look at the implementation of one of its tools.

Chapter 2

Preliminaries

This chapter reviews some basic concepts from set theory, discrete mathematics and formal languages. This chapter also introduces the terminology used throughout this thesis and introduces some formal language systems. This chapter is an extended version of Chapter 2 from [17].

2.1 Sets and Relations

This section reviews some basic ideas from set theory (see [10]).

Definition 2.1.1. A *set* Σ is a collection of elements taken from some *universe*. If Σ contains a finite amount of elements, it is a *finite set*. A finite set is customarily specified by listing its members: $\Sigma = \{e_1, e_2, ..., e_n\}$, where e_1 through e_n are all members of Σ. If Σ contains an infinite amount of elements, it is an *infinite set*. An infinite set is usually specified by a predicate, π, so that it contains all elements which satisfy this property; this is denoted by $\Sigma = \{x : \pi(x)\}$. The set with zero elements is denoted by $\varnothing$ and is called an *empty set*. $\{\} = \varnothing$. $|\Sigma|$ is the *cardinality* of the set Σ. It is the number of elements in Σ. If Σ contains an element a, we denote this by $a \in \Sigma$ and refer to a as a *member* of Σ. If a is not in Σ, it is denoted by $a \notin \Sigma$. Let A and B be two sets. A is a *subset* of B, symbolically written $A \subseteq B$, when each member of A also belongs to B. A is a *proper subset*, symbolically $A \subset B$, when $A \subseteq B$ and B contains at least one element that doesn't belong to A.
The *union* of A and B, denoted by $A \cup B$, is defined as

$$A \cup B = \{x : x \in A \vee x \in B\}$$

The *intersection* of A and B, denoted by $A \cap B$, is defined as

$$A \cap B = \{x : x \in A \wedge x \in B\}$$

The *difference* of A and B, denoted by $A - B$, is defined as

$$A - B = \{x : x \in A \wedge x \notin B\}$$

If Σ is a set over a universe U, then its *complement*, denoted by Σ', is defined as

$$\Sigma' = U - \Sigma$$

The *power set* of A, denoted by 2^A, is the set of all subsets of A.

Definition 2.1.2. For two objects, a and b, (a, b) denotes the *ordered pair* consisting of a and b, in this order.

Definition 2.1.3. Let A and B be two sets. The *Cartesian product* of A and B, $A \times B$, is defined as

$$A \times B = \{(a, b) : a \in A, b \in B\}$$

Definition 2.1.4. Let A and B be two sets. Any subset $\rho \subseteq A \times B$ is a *binary relation* or, briefly, *relation*. If ρ represents a finite set, it is a *finite relation*. If ρ represents an infinite set, it is an *infinite relation*. To express that $(a, b) \in \rho$, we usually write $a\rho b$. Let A be a set, $a, b \in A$, and the relation $\rho \subseteq A \times A$. For $k \geq 1$, the *k-fold product* of ρ, ρ^k, is recursively defined as

- $a\rho^0 b$ if $a = b$
- $a\rho^1 b$ if $a\rho b$
- $a\rho^k b$ if there exists $c \in A$ such that $a\rho c$ and $c\rho^{k-1} b$ for $k \geq 2$.

The *transitive closure* of ρ, ρ^+, is defined as $a\rho^+ c$ only if $a\rho^k b$ for some $k \geq 1$. The *reflexive and transitive closure* of ρ, ρ^*, is defined as $a\rho^* b$ only if $a = b$ or $a\rho^+ b$.

Definition 2.1.5 (page 3 in [9])**.** A *function* from A to B, denoted as $f : A \to B$, is a relation $f \subseteq A \times B$ such that for any $a \in A$, $|\{b : (a, b) \in f\}| \leq 1$. If $(a, b) \in f$, then $f(a)$ denotes b.

2.2 Alphabets and Languages

This section reviews some basic ideas from language theory (see [10]).

Definition 2.2.1. An *Alphabet* Σ is a finite nonempty set, whose members are called *symbols*.

Definition 2.2.2. A finite sequence of symbols from Σ is a *string* over Σ. ε denotes an *empty string*; the string consisting of zero symbols. By Σ^* we denote the set of all strings over Σ. $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. Let $x, y \in \Sigma^*$. $|x|$ denotes the *length* of x: how many symbols in sequence there are in x. Σ^n denotes $\{z \in \Sigma^* : |z| = n\}$. When $x = ayb$, where $x, a, b, y \in \Sigma^*$, then y is a *substring* of x. $substring(x, y)$ is true when y is a substring of x. $symbol(x, n)$ denotes the *nth leftmost* symbol of the string x. $symbol(x, 1)$ is the first symbol of the string; if $i > |x|$, $symbol(x, i) = \varepsilon$. $symbol(x, |x|)$ is the last symbol in the string. Let $x \in \Sigma^*$, $y \in \Sigma$. $occurences(x, y)$ is the *number of occurences* of y in x. It is the number of times the symbol y is in the string x.

Definition 2.2.3. A *language* over Σ is any subset $L \subseteq \Sigma^*$. When L represents a finite set of strings, it is a *finite language*. When it represents an infinite set of strings, it is an *infinite language*.

Definition 2.2.4. Let $x, y \in \Sigma^*$ and L, K be two languages over Σ. The *concatenation* of x with y, denoted by xy, is the string obtained by appending y to x. For every $x \in \Sigma^*$, $x = x\varepsilon = \varepsilon x$. The *concatenation* of L with K, denoted as LK, is defined as

$$LK = \{xy : x \in L, y \in K\}$$

Definition 2.2.5. The *reversal* of x, denoted by $reversal(x)$, is x written in the reverse order. The *reversal* of L, denoted by $reversal(L)$, is defined as

$$reversal(L) = \{reversal(x) : x \in L\}$$

Definition 2.2.6. Throughout this thesis, D denotes Dyck's language defined as

$$D = \{vw : v \in \{0,1\}^*, w = reversal(v)\}$$

This thesis doesn't examine the properties of Dyck's languages; this rudimentary definition is sufficient for the purposes of this thesis. For a more rigorous definition, see page 603 of [8].

2.3 Grammars

Finite languages can be defined by listing all their components. Infinite languages, however, are impossible to define by enumeration. To define both finite and infinite languages, we introduce *grammars*. Grammars are devices that generate languages and play a major role in formal language theory. The following definitions are cited from [14], except where noted otherwise.

Definition 2.3.1. A *phrase-structure grammar* is a quadruple

$$G = (N, T, P, S)$$

where

- N is an alphabet of *nonterminals*
- T is an alphabet of *terminals* such that $N \cap T = \varnothing$
- P is a finite relation from $(N \cup T)^* N (N \cup T)^*$ to $(N \cup T)^*$
- S is the *start symbol*

Pairs $(u, v) \in P$ are called *rewriting rules* or *productions*, and are written as $u \to v$. The *direct derivation* relation over $(N \cup T)^*$ is denoted by $\Rightarrow_G$ and is defined as

$$x \Rightarrow_G y$$

only if $x = x_1 u x_2$, $y = x_1 v x_2$, and $u \to v \in P$, where $x_1, x_2 \in (N \cup T)^*$. The relation $\Rightarrow_G$ is often denoted by $\Rightarrow$ where no ambiguity rises. Let $S \Rightarrow^* x$, $x \in (N \cup T)^*$. Then, x is a *sentential form*. If $x \in T^*$, then x is a *sentence*. If x is is a sentence, then $S \Rightarrow^* x$ is a *successful derivation*. We can explicitly denote the production or series of productions by writing $x \ {}_{[p_1 p_2 \ldots p_n]} \Rightarrow^* y$.
The *language of* G, denoted by $L(G)$, is the set of all sentences defined as

$$L(G) = \{w \in T^* : S \Rightarrow_G^* w\}$$

Definition 2.3.2. A *recursively enumerable language* is a language generated by a phrase-structure grammar. The family of recursively enumerable languages is denoted by **RE**.

Definition 2.3.3. A *context-sensitive grammar* is a phrase-structure grammar

$$G = (N, T, P, S)$$

such that every $u \to v \in P$ is of the form $u = x_1 A x_2$, $v = x_1 y x_2$, where $x_1, x_2 \in (N \cup T)^*$, $A \in N$, and $y \in (N \cup T)^+$.

Definition 2.3.4. A *context-sensitive language* is a language generated by a context-sensitive grammar. The family of context-sensitive languages is denoted by **CS**.

Definition 2.3.5. A *context-free grammar* is a phrase-structure grammar

$$G = (N, T, P, S)$$

such that every rule in P is of the form $A \to x$, where $A \in N$ and $x \in (N \cup T)^*$.

Definition 2.3.6. A *context-free language* is a language generated by a context-free grammar. The family of context-free languages is denoted by **CF**.

Definition 2.3.7. A *linear grammar* is a phrase-structure grammar

$$G = (N, T, P, S)$$

such that every rule in P is of the form $A \to x$ or $A \to xBy$, where $A, B \in N$ and $x, y \in T^*$.

Definition 2.3.8. A *linear language* is a language generated by a linear grammar. The family of linear languages is denoted by **LIN**.

Definition 2.3.9. A *regular grammar* is a phrase-structure grammar

$$G = (N, T, P, S)$$

such that every rule in P is of the form $A \to aB$ or $A \to B$, where $A, B \in N$ and $a \in T$.

Definition 2.3.10. A *regular language* is a language generated by a regular grammar. The family of regular languages is denoted by **REG**.

Theorem 2.3.11 (Chomsky hierarchy, see [3])**.**

$$\mathbf{REG} \subset \mathbf{LIN} \subset \mathbf{CF} \subset \mathbf{CS} \subset \mathbf{RE}$$

Definition 2.3.12 (page 20 of [13])**.** A *queue grammar* is a sextuple

$$Q = (V, T, W, F, R, g)$$

where

- V is an alphabet of nonterminals and terminals
- $T \subset V$ is an alphabet of terminals
- W is an alphabet of states. $V \cap W = \varnothing$
- $F \subset W$ is a set of final states
- $R \subseteq (V \times (W - F)) \times (V^* \times W)$ is a finite relation

- $g \in (V - T)(W - F)$ is the starting pair of symbols

If $u = arb$, $v = rxc$, and $((a, b), (x, c)) \in R$, where $r, x \in V^*$, $a \in V$, and $b, c \in W$, then Q makes a derivation step $u \Rightarrow v$ according to $((a, b), (x, c))$ — usually denoted as (a, b, x, c) for brevity.
The language generated by a queue grammar Q, denoted by $L(Q)$, is defined as

$$L(Q) = \{x : x \in T^*, g \Rightarrow_Q^* xf, f \in F\}$$

Queue grammars characterize the family of **RE**.

2.4 Automata

In this section, we define automata. Automata are devices that recognise strings in a given language. They are typically used to accept languages generated by grammars.

Definition 2.4.1 (see [14])**.** A *finite state automaton* is a quintuple

$$M = (Q, \Sigma, R, s, F)$$

where

- Q is a finite set of *states*
- Σ is an *input alphabet*
- $R \subseteq (Q \times (\Sigma \cup \{\varepsilon\})) \times Q$ is the set of *rules* or *transitions*
- $s \in Q$ is the *start state*
- $F \subseteq Q$ is the set of *final states*

Instead of $((p, y), q) \in R$, we write $py \to q \in R$. If $y = \varepsilon$, we write $p \to q \in R$. A *configuration* of M is any string from $Q\Sigma^*$. The relation of a *move*, symbolically denoted by $\vdash_M$ (or $\vdash$ where no ambiguity rises), is defined over $Q\Sigma^*$ as follows:

$$pyx \vdash qx$$

only if $y \in (\Sigma \cup \{\varepsilon\})$, $pyx, qx \in Q\Sigma^*$ and $py \to q \in R$.
The *language* of M is defined as

$$L(M) = \{w \in \Sigma^* : sw \vdash^* f, f \in F\}$$

Finite state automata characterize the family of **REG**.

Definition 2.4.2 (see [14])**.** A *pushdown automaton* is a septuple

$$M = (Q, \Sigma, \tau, R, s, S, F)$$

where

- Q, Σ, s and F are defined as in a finite automaton
- τ is a *pushdown alphabet*

- $R \subseteq (\tau \times Q \times (\Sigma \cup \{\varepsilon\})) \times (\tau^* \times Q)$ is the set of *rules* of *transitions*
- S is the *initial pushdown symbol*

Q and $(\Sigma \cup \tau)$ are always assumed to be disjoint. Instead of $((a, b, c), (d, e)) \in R$, we write $abc \to de$. The configuration of M is any string from $\tau^* Q \Sigma^*$. The relation of a *move*, denoted by $\vdash_M$ or $\vdash$, is defined as

$$xvpay \vdash xwqy$$

where $x, w \in \tau^*$, $v \in \tau$, $p, q \in Q$, $a \in (\Sigma \cup \{\varepsilon\})$, $y \in \Sigma^*$ and $vpa \to wq \in R$.
The *language accepted by M empty pushdown* is defined as

$$L(M) = \{x \in \Sigma^* : Ssx \vdash^* f, f \in Q\}$$

The *language accepted by M final state* is defined as

$$L(M) = \{x \in \Sigma^* : Ssx \vdash^* Zf, Z \in \tau^*, f \in F\}$$

The *language accepted by M empty pushdown and final state* is defined as

$$L(M) = \{x \in \Sigma^* : Ssx \vdash^* f, f \in F\}$$

Pushdown automata define the family of **CF**.

Definition 2.4.3. A *two-stack pushdown automaton* (or 2PDA) is a pushdown automaton with two stacks. A two-stack pushdown automaton M is an octuple

$$M = (Q, \Sigma, \tau, R, s, S_I, S_O, F)$$

where

- Q, Σ, s, F and τ are defined as in a pushdown automaton
- $R \subseteq \tau \times \tau \times Q \times (\Sigma \cup \{\varepsilon\}) \times \tau^* \times \tau^* \times Q$ is the set of *rules* or *transitions*
- S_I is the *initial input pushdown symbol*
- S_O is the *initial output pushdown symbol*

Instead of $((a_i, a_o, b, c), (d_i, d_o, e)) \in R$, we write

$$a_i|a_o bc \to d_i|d_o e \in R$$

where $| \notin (Q \cup \Sigma \cup \tau)$ is a special symbol denoting the border between the two stacks. The *configuration* of M is any string from $\tau^*|\tau^* Q \Sigma^*$. The relation of a *move*, denoted by $\vdash_M$ or $\vdash$, is defined as

$$s_i i|s_o opay \vdash s_i I|s_o Oqy$$

where $i, o \in \tau$, $s_i, s_o, I, O \in \tau^*$, $p, q \in Q$, $a \in (\Sigma \cup \{\varepsilon\})$, $y \in \Sigma^*$ and $i|opa \to I|Oq \in R$.
The *language accepted by M final state* is defined as

$$\{x \in \Sigma^* : S_I|S_O sx \vdash^* s_i|s_o f, s_i, s_o \in \tau^*, f \in F\}$$

The *language accepted by M empty input pushdown* is defined as

$$\{x \in \Sigma^* : S_I|S_O sx \vdash^* |s_o f, s_o \in \tau^*, f \in Q\}$$

The *language accepted by M empty input pushdown and final state* is defined as

$$\{x \in \Sigma^* : S_I|S_O sx \vdash^* |s_o f, s_o \in \tau^*, f \in F\}$$

2.5 More Definitions

Grammars and automata may not always be convenient in written text to express some ideas. This section describes some convenient ways to define languages instead of grammars or automata.

Definition 2.5.1. *Regular expressions* define the family of **REG**. Quoting verbatim from [10], they are recursively defined as follows:

1. $\varnothing$ is a regular expression denoting the empty set.
2. ε is a regular expression denoting $\{\varepsilon\}$.
3. a, where $a \in \Sigma$, is a regular expression denoting $\{a\}$.
4. if r and s are regular expressions denoting the languages R and S, respectively, then
   (a) (rs) is the regular expression denoting RS.
   (b) $(r + s)$ is the regular expression denoting $R \cup S$.
   (c) (r^*) is the regular expression denoting R^*.

Parentheses are omitted when no ambiguity arises. For convenience, we also introduce the repetition operator: for a regular expression r denoting the language R, $r^{n \in \mathbb{N}}$ denotes R^n.

Chapter 3

Parsing

In this chapter, we discuss existing parsing methods. We discuss both top-down and bottom-up parsing algorithms in detail. We take a look at variations of the LR parsing algorithm and review their properties. We discuss conflict resolution in LR parsers and we describe a common method used in practical parsing tools such as GNU Bison. The contents of this chapter were already covered in [18] and most of its contents, with the exception of conflict resolution, are quoted from there.

3.1 Context-free grammars

Parsing methods introduced in this chapter typically work with a subset of context-free grammars. Here, we introduce some additional concepts that will help us talk about context-free grammars in the context of parsing.

Definition 3.1.1. For any context-free grammar $G = (N, T, P, S)$ and a series of productions $a \in P^*$, we can construct a *parse tree*. Intuitively, it is a tree where its nodes are terminals and nonterminals, and each symbol's parent node is the nonterminal of the production it was added in.

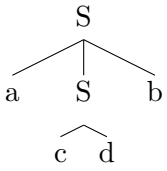

Figure 3.1: The parse tree of $S \Rightarrow aSb \Rightarrow acdb$

Figure 3.1 shows a parse tree for a context-free grammar

$$G = (\{S\}, \{a, b, c, d\}, \{S \to aSb, S \to cd\}, S)$$

Each non-leaf node is a nonterminal, and all leaf nodes are terminals.

Definition 3.1.2 (Definition 3.4 in [9])**.** Let $G = (N, T, P, S)$ be a context-free grammar. A *leftmost derivation* over $(N \cup T)^*$ is denoted by $\Rightarrow_G^{lm}$ and is defined as

$$x_1 u x_2 \Rightarrow_G^{lm} x_1 v x_2$$

only if $u \to v \in P$, where $x_1 \in T^*$ and $x_2 \in (N \cup T)^*$. A sequence of productions $\alpha \in P^+$ such that $S \underset{[\alpha]}{\Rightarrow}^{lm*} t$, where $t \in L(G)$ is a sentence, is called a *left parse* of G. A *rightmost derivation* over $(N \cup T)^*$ is denoted by $\Rightarrow_G^{rm}$ and is defined as

$$x_1 u x_2 \Rightarrow_G^{rm} x_1 v x_2$$

only if $u \to v \in P$, where $x_1 \in (N \cup T)^*$ and $x_2 \in T^*$. A sequence of productions $\alpha \in P^+$ such that $S \underset{[\alpha]}{\Rightarrow}^{rm*} t$, where $t \in L(G)$ is a sentence, is called a *right parse* of G.

Definition 3.1.3 (Definition 3.6 in [9])**.** Let $G = (N, T, P, S)$ be a grammar. G is *ambiguous* if there exists a sentence $w \in L(G)$ and two different sequences $a, b \in P^+, a \neq b$, such that $S \underset{[a]}{\Rightarrow}^{lm*} w$ and $S \underset{[b]}{\Rightarrow}^{lm*} w$. It is *unambiguous* otherwise.

Definition 3.1.4. Let $G = (N, T, P, S)$ be a context-free grammar. An *augmented context-free grammar* is a context-free grammar

$$G' = (N \cup \{S'\}, T \cup \{\$\}, P \cup \{S' \to S\$\}, S')$$

where $S' \notin N$, $\$ \notin T$ and $\$$ is the end of input symbol. Augmented grammars generate the same language as the grammar they were created from, but each sentence has the special end of input symbol $\$$ appended to it.

3.2 Predictive sets

Both top-down and bottom-up parsing algorithms use two predictive sets first and follow associated with a context-free grammar G. The definitions, algorithms and procedures introduced here are cited from [10].

Definition 3.2.1. Let $G = (N, T, P, S)$ be a context-free grammar. For every string $x \in (N \cup T)^*$,

$$first(x) = \{a : x \Rightarrow^* aw, (w \in (N \cup T)^*, a \in T \vee a = aw = \varepsilon)\}$$

If $x \Rightarrow^* \varepsilon$, where $x \in (N \cup T)^*$, then $\varepsilon \in first(x)$; as a special case, for $x = \varepsilon$, $first(x) = \{\varepsilon\}$.

The set $first(x)$ is the predictive set of terminals or ε that can be the first symbol in a sentential form derived from x. This predictive set will be used later for the construction of other predictive sets. Algorithm 3.1 creates $first(x)$ for every $x \in N \cup T \cup \{\varepsilon\}$. Algorithm 3.2 creates $first(x)$ for any string $x \in (N \cup T)^*$ when $first(y)$ is known for all $y \in N \cup T \cup \{\varepsilon\}$.

Definition 3.2.2. For every nonterminal X, we define the set $follow$:

$$follow(X) = \{x \in T : Xx \text{ is a substring of a sentential form of G}\}$$

For every nonterminal N, the predictive set *follow(N)* contains all terminals that may follow the nonterminal N in a sentential form.

Algorithm 3.3 creates follow for every nonterminal in the translation grammar G. We add symbols following nonterminals in productions to their follow sets until no more modifications are possible.

Algorithm 3.1 first (based on [12] and [10])

Input: context-free grammar $G = (N, T, P, S)$
Output: set $first(x)$ for each $x \in N \cup T \cup \{\varepsilon\}$

```
for all x ∈ N do
    first(x) := ∅
for all x ∈ T do
    first(x) := {x}
first(ε) := {ε}
repeat
    for all A → i ∈ P do
        Add all symbols from first(symbol(i, 1)) to first(A)
        if ε ∈ first(symbol(i, n)) for n = 1, ..., k − 1 where k ≤ |i| + 1 then
            Add all symbols from first(symbol(i, k)) to first(A)
until no change
```

Algorithm 3.2 string first (based on [12])

Input: context-free grammar $G = (N, T, P, S)$, string $x \in (N \cup T)^*$, $first(X)$ for every $X \in N \cup T \cup \{\varepsilon\}$
Output: set $first(x)$

```
first(x) := first(symbol(x, 1))
if ε ∈ first(symbol(x, n) for n = 1, ..., k − 1 where k ≤ |x| then
    Add all symbols from first(symbol(x, k)) − {ε} to first(x)
if ε ∈ first(symbol(x, n) for n = 1, ..., |x| then
    Add all ε to first(x)
```

3.3 Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string (see [1]), starting from the root nonterminal. We are constructing the series of rules in a left-most derivation of the input string.

3.3.1 Recursive Descent

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. General recursive-descent parsers may need to implement backtracking to scan over the input multiple times, but this is not necessary for most programming language constructs (see page 219 of [1]).

The idea for recursive-descent parsers is manually implementing the parser to avoid tables and an explicit pushdown. Instead, the pushdown is simulated by the programming language's recursion mechanism, and the selection of productions to apply is left to the programmer. This may, of course, result in differences from the context-free grammar if there are errors or deliberate differences in the implementation.

The main disadvantage of recursive-descent parsers is the need to manually rewrite the individual procedures whenever the grammar is changed; this means that the resulting parser is less flexible than parsers with generated tables. Since implementing recursive

Algorithm 3.3 follow (based on [12] and [10])

Input: context-free grammar $G = (N, T, P, S)$ and $first(x)$ for each $x \in N \cup T \cup \{\varepsilon\}$.
Output: $follow(x)$ for each $x \in N$
Require: $\$ \notin N \cup T$
  $follow(S) \coloneqq \{\$\}$
  repeat
    for all aBy, where $A \to aBy \in P$, $a, y \in (N \cup T)^*$, $B \in N$ **do**
      if $y \neq \varepsilon$ **then**
        Add all symbols from $first(y) - \{\varepsilon\}$ to $follow(B)$
      if $\varepsilon \in first(y)$ **then**
        Add all symbols from $follow(A)$ to $follow(B)$
  until no change

descent parsers does not generally benefit from using any parsing frameworks, we will not discuss them further.

3.3.2 LL(1) Parsing

LL(1) parsing is a table method for top-down parsing. We construct a table which lets us decide which leftmost production is possible based on a single lookahead symbol. The first L stands for a left-to-right scan of symbols and the other L stands for leftmost derivation.

Definition 3.3.1. Let $G = (N, T, P, S)$ be a context-free grammar. The *predict* set (see [10]) is defined for every $a = A \to i \in P$ as

1. If $\varepsilon \notin first(i)$, then $predict(a) = first(i)$
2. If $\varepsilon \in first(i)$, then $predict(a) = (first(i) - \{\varepsilon\}) \cup follow(A)$

The predictive set *predict* is the set of all terminals that may be the leftmost generated symbol if we apply the production a to the leftmost nonterminal. That is, for all

$$xAq \,_{[a\alpha]} \Rightarrow^{lm*} xB$$

where $x, B \in T^*$, $A \in N$, $q \in (N \cup T)^*$, and $a = A \to i \in P$ is the first production used in the derivation, $predict(a)$ is the set of all terminals t such that t is the first symbol of B.

Definition 3.3.2. A context-free grammar $G = (N, T, P, S)$ is an *LL(1) context-free grammar* if for each $A \in N$, any two different productions $p = A \to x$, $q = A \to y$, satisfy $predict(p) \cap predict(q) = \varnothing$.

Next, we introduce a *predictive table*. This table determines a production for each pair of nonterminal and terminal. In predictive top-down parsing, this table determines the production that is used for nonterminal expansion. Algorithm 3.4 shows the construction of this table.

Algorithm 3.5 describes the predictive top-down parsing method. This method is based on Algorithm 7.17 from [10]. The contents of the stack *ipd* is written head-first; the first symbol is the symbol on top of the stack. In addition to regular stack functionality (*POP*, removing the top element from the stack and *PUSH*), both stacks support the operation *REPLACE*. REPLACE(n, x) replaces the symbol n closest to the top of the stack with the string x. The first symbol in x will be closest to the top of the stack. This behavior can be achieved with a stack with only PUSH and POP, but we introduce REPLACE for brevity.

Algorithm 3.4 Constructing a predictive table

Input: LL(1) context-free grammar $G = (N, T, P, S)$, $predict(x)$ for each $x \in P$
Output: predictive table α
  for all $t \in T \cup \{\$\}$, $n \in N$ **do**
    $\alpha(n, t) \coloneqq null$
  for all $y \in predict(x)$ for each $x = A \to i \in P$ **do**
    $\alpha(A, y) \coloneqq x$

Algorithm 3.5 Predictive top-down parsing (see algorithm 7.17 in [10])

Input: an augmented context-free grammar $G = (N \cup \{S'\}, T \cup \{\$\}, P \cup \{S' \to S\$\}, S')$, its predictive table α and the input string $x\$$, where $x \in (T - \{\$\})^*$
Output: **SUCCESS** or **ERROR**
  $ipd \coloneqq S\$$
  $n \coloneqq 1$
  repeat
    let X denote the current ipd top symbol
    let t be the nth symbol in the input string
    switch X**:**
      case $X = \$$**:**
        if $t = \$$ then **SUCCESS**, else **ERROR**
      case $X \in T$**:**
        if $t = X$ then increment n and POP ipd, else **ERROR**
      case $X \in N$**:**
        if $\alpha(X, t) = null$ **then**
          ERROR
        else
          $\alpha(X, t) = X \to i$
          REPLACE(X, i) in ipd
  until SUCCESS or **ERROR**

3.4 Bottom-up Parsing

Bottom-up parsing takes the opposite approach in comparison to top-down parsing; the parse tree is constructed starting from the terminals, reaching the top starting nonterminal at the very end. Although there exist simpler bottom-up parsing algorithms (e.g. precedence parsing, see page 159 in [9]), we will focus on the LR family of parsing algorithms. We will introduce SLR parsing, canonical LR(1) parsing, LALR parsing, and finally IELR parsing. Each subsequent algorithm makes heavy use of the concepts used in the previous algorithms. We assume all context-free grammars are augmented, and we will always append the end of input symbol \$ to the end of each input string.

3.4.1 LR parsing

The LR parsing algorithm is the same for all algorithms below; where they differ is in the construction of the *action* and *goto* tables. The action table contains a parser action (shift s, reduce $X \to Y$, success, or error) for each pair of state and terminal. The goto table contains

Algorithm 3.6 LR parsing (see page 251 in [1])

Input: action and goto tables, an input string $w\$$
Output: a reversed rightmost parse of the input string or **ERROR**

```
s_0 is the contents of the stack (s_0 is the initial state)
let a be the first symbol of input w$
while true do
    let s be the state on top of the stack
    if action[s, a] = shift t then
        push t onto the stack
        let a be the next input symbol
    else if action[s, a] = reduce A → B then
        pop |B| symbols off the stack
        let state t now be on top of the stack
        push goto[t, A] onto the stack
        output the production A → B
    else if action[s, a] = accept then
        break (parsing is done)
    else
        return ERROR (or error recovery)
```

either a state or nothing for each pair of state and nonterminal. The parsing method is shown in Algorithm 3.6.

3.4.2 SLR Parsing

SLR parsing is the simplest LR parsing method, and is thus a good starting point for studying LR parsing.

First, we will introduce LR(0) tables and LR(0) automata as introduced in [1]. An LR parser makes shift-reduce decisions by maintaining states to track where we are in a parse. Every state is a collection of items.

Definition 3.4.1. An *LR(0) item* is a rule from a context-free grammar G with a dot at some position of the body. For example, a production $A \to bC$ yields three items $A \to \cdot bC$, $A \to b \cdot C$ and $A \to bC \cdot$.

Algorithm 3.7 closure (see page 245 in [1])

Input: a set of items I
Output: a set of items O

```
O := I
repeat
    for all items A → α·Bβ in O (where α, β ∈ (N ∪ T)*) do
        for all rules B → X ∈ P (where X ∈ (N ∪ T)*) do
            add B → ·X to O
until no more items are added in an iteration
```

Definition 3.4.2. A *closure* of a set of items I is constructed in the way described in Algorithm 3.7. If the dot symbol is in front of a nonterminal in one of the items,

all rules where that nonterminal is on the left-hand side are added to the closure set with the dot before the first symbol on the right-hand side. This is repeated until there are no more changes.

Definition 3.4.3. The *goto* function is constructed in the following way: $goto(I, X)$ is the closure of all items $A \to \alpha X \cdot \beta$ such that $A \to \alpha \cdot X\beta \in I$.

Algorithm 3.8 LR(0) automaton (see page 246 in [1])

Input: An augmented grammar G
Output: A LR(0) automaton (C, E)
$C \coloneqq closure(\{S' \to \cdot S\$\})$. This first state is stored as I_0.
$E \coloneqq \varnothing$
repeat
  for all sets of items I in C **do**
    for all symbols $X \in N \cup T \cup \{\$\}$ **do**
      if $goto(I, X)$ is not empty **then**
        if $goto(I, X)$ is not in C **then**
          add $goto(I, X)$ to C
          insert $I \xrightarrow{X} goto(I, X)$ to E
until no new sets of items are added to C in an iteration

We start constructing the parsing tables by generating an LR(0) automaton. An LR(0) automaton is a graph, where nodes are sets of items and edges are marked with the symbol that has been recognized. The construction of an LR(0) automaton is described in Algorithm 3.8. The initial state is the closure of the initial item $S' \to \cdot S\$$. Then, from each state, the dot is moved over each symbol in the state and new states are created this way. If (S_1, S_2) is an edge over the symbol X, we denote this by $S_1 \xrightarrow{X} S_2$. If abc is a series of symbols, we may denote a series of transitions $S_1 \xrightarrow{a} S_2 \xrightarrow{b} S_3 \xrightarrow{c} S_4$ as $S_1 \xrightarrow{abc} S_4$.

Algorithm 3.9 SLR action and goto tables

Input: an LR(0) automaton with the states $\{I_0, I_1, \dots\}$, the follow set for each nonterminal
Output: the action and goto tables
State i is constructed from I_i. The parsing actions for state i are determined as follows:
All action entries are initialized as an error.
If $A \to \alpha \cdot a\ \beta$ is in I_i, $a \neq \$$ and $I_i \xrightarrow{a} I_j$, set $action[i, a] \coloneqq$ shift j
If $A \to \alpha \cdot$ is in I_i and $A \neq S'$, set $action[i, a] \coloneqq$ reduce $A \to \alpha$ for all $a \in follow[A]$.
If $S' \to S \cdot \$$ is in I_i, set $action[i, \$] \coloneqq$ accept
If any conflicting actions arise from the above rules, we say the grammar is not SLR(1).
The goto transitions are created using the rule: If $I_i \xrightarrow{X \in N} I_j$, then $goto[i, X] = j$.
The initial state I_0 is the one constructed from $S' \to \cdot S\$$.

The SLR algorithm constructs the goto action tables from the LR(0) automaton and the follow set for each nonterminal. The method is described in Algorithm 3.9.

3.4.3 Canonical LR(1) Parsing

Canonical LR(1) parsing creates an LR(1) automaton with LR(1) items. An LR(1) item contains a lookahead symbol $s \in T$ that denotes which symbol may be the first read symbol in the input string after a reduction is done. The following is cited from [1].

Canonical LR(1) parsers are the strongest LR parsers with the lookahead size of 1, but lead to very large automatons, and thus to very large parsing tables. Until recently, the size of the tables made it impossible to use this algorithm in practice, and it is not practical to this day. Most practical grammars are LALR, and those that aren't can be parsed with the IELR algorithm with the same strength as canonical LR(1) parsing, but much smaller tables (see Section 3.4.5).

Definition 3.4.4. An *LR(1) item* is an LR(0) item with a terminal symbol (or \$) attached to it. For example, if we have an LR(0) item $A \to B{\cdot}CD$, we can create an LR(1) item $[A \to B{\cdot}CD, a]$. The lookahead has no effect in an item where the dot isn't at the last position, but when it is, it tells us which terminal can follow the production in order for it to be reduced. For example, if we have a state containing only $[A \to BC{\cdot}, a]$, we are only compelled to reduce BC to A if the next lookahead symbol is exactly a.

Algorithm 3.10 LR(1) closure

Input: a set of LR(1) items I, a context-free grammar $G = (N, T, P, S)$
Output: O: the closure of I

```
O := I
repeat
    for all items [A → α·Bβ, a] ∈ O do
        for all production B → γ ∈ P do
            for all b ∈ first(βa) do
                add [B → ·γ, b] to O
until no more items are added to O
```

Algorithm 3.11 LR(1) goto

Input: a set of items I, a symbol $X \in N \cup T \cup \{\$\}$
Output: a set of items O

```
for all items [A → α·Xβ, a] do
    add item [A → αX·β, a] to O
O := closure(O)
```

With LR(1) items, we can construct a LR(1) automaton. Algorithm 3.10 contains a procedure for the closure of sets of LR(1) items, Algorithm 3.11 contains the goto function for sets of LR(1) items, and finally Algorithm 3.12 shows the creation of a LR(1) automaton. The process is largely the same as with LR(0) items, but the modification of the closure and goto functions ensures that the lookahead symbols are computed. As a result of the lookahead symbols being added to items, more states with different lookahead sets can be created.

The creation of the action and goto tables for canonical LR parsers is described in Algorithm 3.13. The goto table is constructed the same way as in previous algorithms, using the modified *goto* function. In the action table, the reduce action is added to the items with the terminal symbols matching the lookahead symbols in the state's items.

Algorithm 3.12 LR(1) automaton

Input: a context-free grammar $G = (N, T, P, S)$
Output: an LR(1) automaton (O, E) with the states $\{I_0, I_1, \dots\}$
initialize C to $closure([S' \to S\$, \$])$ (this initial item is I_0)
repeat
for all sets of items I in C **do**
for all symbols $X \in N \cup T \cup \{\$\}$ **do**
if $goto(I, X)$ is not empty **then**
if $goto(I, X)$ is not in C **then**
add $goto(I, X)$ to C
insert $I \xrightarrow{X} goto(I, X)$ to the set E
until no new sets of items are added to C in an iteration

Algorithm 3.13 LR(1) action and goto

Input: an augmented context-free grammar G, a LR(1) automaton with the states $\{I_0, I_1, \dots\}$
Output: the action and goto tables
State i is constructed from I_i. The parsing actions for state i are determined as follows:
All action entries are initialized as an error.
If $[A \to \alpha \cdot a\ \beta, b]$ is in I_i, $a \in T$, $a \neq \$$ and $I_i \xrightarrow{a} I_j$, set $action[i, a] :=$ shift j
If $[A \to \alpha \cdot, a]$ is in I_i and $A \neq S'$, set $action[i, a] :=$ reduce $A \to \alpha$
If $[S' \to S \cdot \$, \$]$ is in I_i, set $action[i, \$] :=$ accept
If any conflicting actions arise from the above rules, we say the grammar is not LR(1).
The goto transitions are created using the rule: If $I_i \xrightarrow{X \in N} I_j$, then $goto[i, X] = j$.
The initial state I_0 is the one constructed from $[S' \to \cdot S\$, \$]$.

3.4.4 LALR Parsing

In this section, we will examine the construction of action and goto tables for the LALR (look-ahead LR) parsing algorithm from [1] and [16]. LALR parsing is a middle ground between the relatively small recognition power of the SLR algorithm and the large state count of canonical LR(1) parsing. It utilizes a lookahead set for each state that lets us better predict when a reduction is possible but still uses the much smaller LR(0) automaton, leading to the same number of states as the SLR algorithm. There exist LR(1) grammars for which LALR parsing will result in conflicts; there will be different LR(1) states that are merged in LALR (with the same LR(0) core, but different lookahead sets, called *isocores*), potentially creating new reduce-reduce conflicts. The construction of the action and goto tables is the same as in Algorithm 3.9, but the construction of reduce rules is different. If $A \to \alpha \cdot$ is in I_i and $A \neq S'$, set $action[i, a] :=$ reduce $A \to \alpha$ for all $a \in LA(i, A \to \alpha)$.

There exist multiple ways of computing the LA (lookahead) set that is needed for the construction of LALR tables. The simplest is to create the LR(1) automaton and merge all states with the same LR(0) item cores, but different lookahead symbols (see [1]). We present two better approaches to constructing LALR parsers. Both construct the LR(0) automaton and then compute the sets of lookahead symbols. The first approach is a straight-forward construction from the LR(0) automaton, where we propagate lookaheads through the au-

tomaton. The second approach is an efficient method of obtaining sets of lookahead symbols utilized by the IELR algorithm.

Definition 3.4.5. A *kernel* of an LR state S is a subset of items, where the $LR(0)$ portion of each item has the form $A \to \alpha \cdot \beta$, where $\alpha \neq \varepsilon$, or as a special case $S' \to \cdot S\$$. The full state S can be computed from its kernel K as $S = CLOSURE(K)$.

Lookahead Propagation

Algorithm 3.14 Determining lookaheads (see Algorithm 4.62 in [1])

Input: the kernel K of a set of LR(0) items I and a grammar symbol X
Output: the lookaheads spontaneously generated by items in I for kernel items in $GOTO(I, X)$ and the items in I from which lookaheads are propagated to kernel items in $GOTO(I, X)$

for all each item $A \to \alpha \cdot \beta \in K$ **do**
  $J \coloneqq CLOSURE(\{[A \to \alpha \cdot \beta, \#]\})$
  if $[B \to \gamma \cdot X\delta, a] \in J$ and $a \neq \#$ **then**
    Conclude that a is generated spontaneously for item $B \to \gamma X \cdot \delta$ in $GOTO(I, X)$.
  if $[B \to \gamma \cdot X\delta, \#] \in J$ **then**
    Conclude that lookaheads propagate from $A \to \alpha \cdot \beta \in I$
    to $B \to \gamma X \cdot \delta$ in $GOTO(I, X)$.

The first approach (see Section 4.7 in [1]) lets us get a set of lookaheads generated for each successor in an LR(0) automaton. We also obtain the set of successor states where lookaheads will be propagated. Algorithm 3.14 gives us the set of generated lookaheads for each transition in the LR(0) automaton and how lookaheads are propagated through that transition.

Algorithm 3.15 Computation of the kernels of the LALR(1) automaton (see Algorithm 4.63 in [1]

Input: an augmented context-free grammar G'
Output: kernels of the LALR(1) states.

1. Construct the kernels of the sets of LR(0) items for G'. A simple approach would be computing the full states and removing nonkernel items.
2. Apply Algorithm 3.14 to each kernel and grammar symbol to obtain the sets of generated lookaheads and the lookahead propagation items.
3. Initialize a table that gives, for each kernel item in each set of items, the associated lookaheads. Initially, each item has associated with it only those lookaheads that were generated according to the previous step.
4. Make repeated passes over the kernel items in all sets. When we visit an item i, we look up the kernel items to which i propagates its lookaheads, using information from step 2. The current set of lookaheads of i is added to those items. We continue making passes until no more new lookaheads are propagated.

Algorithm 3.15 describes the process of obtaining LALR(1) kernels. The main issue with this approach is its ineffectiveness if the order of propagation is chosen poorly. After

using this approach and completing the full LALR(1) states as closures of the kernels, we construct the parser tables in the same way as for canonical LR parsers.

An Alternative Approach to LALR Lookahead Computation

An alternative to using the approach from the previous section exists. We will introduce the computation from [16], creating the lookahead sets from the LR(0) automaton. Here, $X =_s F(X)$ denotes that X is the smallest set satisfying $F(X)$. $\bigcup\{A, B, \ldots, D\}$ denotes $A \cup B \cup \cdots \cup D$. Below, we will define the sets and relations that are used to calculate the LA set from a $LR(0)$ automaton as described in [16].

Definition 3.4.6. $(p, A \in N)$ *reads* (r, C) iff $p \xrightarrow{A} r \xrightarrow{C}$ and $C \Rightarrow^* \varepsilon$.

Definition 3.4.7. We introduce the set of symbols *direct read* for each nonterminal transition (p, A), denoted by $DR(p, A)$:

$$DR(p, A \in N) = \{t \in T : p \xrightarrow{A} r \xrightarrow{t}\}$$

Direct read symbols are simply terminals that may be read after A is read in p.

Definition 3.4.8.

$$Read(p, A) =_s DR(p, A) \cup \bigcup\{Read(r, C) : (p, A) \textbf{ reads } (r, C)\}$$

Read is the set of all terminals that can be read before any phrase including A is reduced. This includes the direct read symbols, as well as symbols read indirectly (via the reads relation).

Definition 3.4.9. (p, A) *includes* (p', B) iff

$$B \to XA\gamma, \gamma \Rightarrow^* \varepsilon, p' \xrightarrow{X} p$$

where $X \in (T \cup N)^*$.

If γ may be reduced to ε, then terminals that follow B in state p' may also follow A in state p. We can identify such relations by the includes relation.

Definition 3.4.10.

$$Follow(p, A) =_s Read(p, A) \cup \bigcup\{Follow(p', B) : (p, A) \textbf{ includes } (p', B)\}$$

The follow set is the set of terminals that may follow after reducing to A in state p.

Definition 3.4.11. $(q, A \to \omega)$ *lookback* (p, A) iff $p \xrightarrow{\omega} q$ where $\omega \in (N \cup T)^*$.

The lookback relation tells us whether there exists a path from state p to q through the string ω. After reducing ω to A from state q, p will be one of the possible states after ω is popped from the stack.

Definition 3.4.12. $LA(q, A \to \omega) = \bigcup\{Follow(p, A) : (q, A \to \omega) \textbf{ lookback } (p, A)\}$, where $\omega \in (N \cup T)^*$.

When the parser reduces ω to A in state q, each state p is a possible state after ω is popped. Then, the parser must read A, with some terminal t the first of the input.

The computation of these sets and relations is described in detail using the *digraph* algorithm in [16] and will be skipped here for brevity.

3.4.5 IELR Parsing

IELR (inadequacy elimination LR) parsing was first introduced in [4], and this section is cited from there. Here, we introduce the core concepts of IELR parsing. This algorithm has the same strength as canonical LR(1) parsing (see [4]) with fewer parsing states. The IELR algorithm first computes LALR parsing tables and then computes which inadequacies in the state machine are LR(1) inadequacies, and splits those states accordingly.

Conflicts that are created by merging states are all present in LALR. We can mark states that contribute to these conflicts with their lookahead symbols. When splitting states, we can compare the immediate contributions to potential conflicts to decide whether any two states are compatible.

The IELR algorithm consists of 6 phases, which we label *Phase 0* through *Phase 5*:

- *Phase 0*:
  Compute the LALR parsing tables as described in Section 3.4.4 (as sets and relations, e.g. reads or includes).
- *Phase 1*:
  Compute auxiliary tables from some of the LALR parsing tables (to be used in later phases).
- *Phase 2*:
  Compute annotations. Using the information from phases 0 and 1, we identify each conflict in the LALR parsing tables. When such a conflict is detected, we trace back through states that contribute to this conflict and record the nature of the contributions.
- *Phase 3*:
  Split states. The algorithm effectively splits states to eliminate LR(1) inadequacies. The algorithm effectively recomputes the state machine in a way similar to that of phase 0, but creates more different states based on the annotations from phase 2.
- *Phase 4*:
  Compute reduction lookahead (LA) sets. In this phase, we compute the lookahead sets for the new created states. This is done in the same way as in phase 0.
- *Phase 5*:
  Resolve remaining conflicts. In this phase, the remaining parsing conflicts are resolved. We describe a form of conflict resolution in Section 3.4.6.

Phases 1 through 3 are the core of IELR; phases 1 and 2 recognize and annotate LR(1) inadequacies, and phase 3 splits the states to avoid them.

Phase 1

Phase 1 calculates three tables for the next phases: *predecessors*, *follow_kernel_items* and *always_follows*.

The predecessors table simply denotes the predecessors of each state in the state machine. $predecessors(I_i) = \{I_n : I_n \xrightarrow{X \in (N \cup T)} I_i\}$. Note that I_0 never has any predecessors. The follow_kernel_items table holds boolean values; if $follow\_kernel\_items[(I_i, I_n), k]$ is true, then the lookahead sets of I_n depend on the kth item of I_i. Finally, the always_follows table holds all terminals that will be always generated as lookahead for all edges of the LALR automaton. This is analogous to computations from Section 3.4.4.

Phase 2

In phase 2, we detect states with shift-reduce and reduce-reduce conflicts, and we annotate those states with the nature of these conflicts.

The annotations themselves take the following form:

$$annotation(I_i, t \in T) = (i, t, \text{number of actions}, \gamma)$$

γ is the inadequacy contribution matrix. $|\gamma|$ = number of actions, and $\forall 1 \leq i \leq |\gamma| : \gamma[i]$ is either:

- Undefined, if the LR(0) item always generates that action (all shift actions).
- A boolean sequence such that $|\gamma[i]| = |I_i|$. If $\gamma[i][j]$ is true, then the jth item of I_i can generate action i if t is in the lookahead set of that item.

Annotations for predecessors are added to the annotations using an annotation propagation procedure. The computation of these annotations is described in detail in [4].

Phase 3

In phase 3, we construct the IELR automaton. The construction of the automaton is very similar to that of LALR parsers. New sets of items are calculated; this time, however, not all items with the same LR(0) core are merged; instead, a new state with the same LR(0) items may be created if the merge wouldn't pass a compatibility test based on the annotations from phase 2.

The algorithm starts with the already computed LR(0) state machine and its lookaheads and propagates those lookaheads through different states. If a compatible state isn't found, a new state is created and the edge is modified so that the transition is made to the new state. If a compatible state is found, its lookahead set is set to be recalculated. State compatibility is examined by their contributions based on the current lookahead sets; if two states merging would cause an inadequacy, they will not be merged. The complete algorithm is in [4].

3.4.6 Conflict Resolution in LR parsers

Many practical grammars are ambiguous. Because of that, it is highly desirable to be able to resolve shift-reduce and possibly reduce-reduce conflicts. Although there exist many other approaches, we will take a look at one specific conflict resolution method used in GNU Bison: operator associativity and precedence (see [5]).

Operator associativity and precedence is a concept employed in operator precedence parsing (see Section 5.1 in [9]). We define precedence levels and associativity for terminals and productions and the decisions whether to shift or reduce those symbols on conflicts is made according to those properties. In order to use this modification, we first define precedence levels and associativity for a subset of terminals. We also associate specific terminals with productions to assign them that terminal's precedence and associativity.

Definition 3.4.13. The *precedence and associativity annotation* of a context-free grammar $G = (N, T, P, S)$ is a triple (p, a, s) , where

- $p : T \to \mathbb{N}$ is an *operator precedence* function. A terminal $x \in T$ has a *higher precedence* than a terminal $y \in T$ if $a(x) < a(y)$.

- $a : \mathbb{N} \to \{left, right, none\}$ is an *operator associativity* function. Each level of precedence is given an associativity.
- $s : P \to T$ is a function assigning terminals to productions. These terminals determine the precedence and associativity of productions. Typically, $s(A \to B)$ will be the last terminal on the right-hand side of the production — we will only mention the associated terminal if it is different.

Operator precedence determines the priority for all terminals within shift-reduce conflicts. If we can either shift or reduce a production $p \in P$ on $t \in T$, we choose the shift action if $p(t) < p(s(p))$, and we choose the reduce action if $p(t) > p(s(p))$. If $p(t) = p(s(p))$, which means that both the terminal and the production have the same precedence, we choose according to that level's associativity. If $a(p(t)) = none$, that precedence level is not associative, and we cannot make a decision. This would result in an error, just like if there was no conflict resolution. If $a(p(t)) = left$, we choose the reduce action, and if $a(p(t)) = right$, we choose the shift action.

The full specification for all terminals would be very large for many grammars and only a small subset of terminals will typically have a meaningful precedence and associativity (such as operators). For most terminals, we choose some $n \gg |T|$ and we set $a(n) = none$. This becomes a default lowest precedence for most terminal symbols so that not all terminals have to be explicitly assigned a precedence.

If there are two reduce actions on $t \in T$, we can simply warn the user and choose the production that was defined earliest in the grammar. Reduce-reduce conflicts are typically a sign of an error in the grammar, so a reduce-reduce conflict could be considered an error. However, there might exist grammars where the shift action is chosen and the remaining reduce-reduce conflict may cause an error on an otherwise valid grammar, even if the reduce-reduce conflict ultimately doesn't affect the parser. If reduce-reduce conflicts are considered to be errors, it is desirable to at least leave the option to opt-out of specific (or all) reduce-reduce errors.

3.5 Translation with Context-free Grammars

In this section, we will describe the approach most compilers use to generate translations when working with context-free grammars as the formal basis. We will focus on syntax-directed translation as described in [9].

When a rule is applied in the parsing algorithm (whether in top-down or bottom-up algorithms), a part of the parse tree is created. Most practical compilers do not create the parse tree, but create the *syntax tree* instead. This tree has the same basic structure as the parse tree, but may omit superfluous information that is not needed for the creation of target code. Because of this, the syntax tree is a more efficient representation of the source program. As an alternative, compilers may instead generate three-address code or expressions in postfix notation. Below, we will assume the generation of syntax trees for simplicity.

Whenever a rule is reduced or expanded (depending on the algorithm), we can create a part of the syntax tree. The creation of the subtree is invoked by the parsing algorithm as a specific action. This generation is not formally defined by the context-free grammar; when we employ translation grammars, we will be able to formally define the output of the parsing algorithms.

3.5.1 Attributes

The input of a parsing algorithm is usually a string of tokens, extracted from the input characters by a lexical analyzer. These tokens are terminals with the possibility of other attributes attached to them (e.g. a terminal for an identifier may have an attribute with its name attached). Attributes are associated with the input terminals and will be put on the pushdown along with them.

3.5.2 Bottom-up actions

In bottom-up parsing, whenever we reduce a specific rule $A \to aBC$, we will perform an *action*, where the parameters of that action are the attributes of the individual symbols that we reduce to a single nonterminal. The action will create a part of the syntax tree and will attach a new attribute a_A to A that will be added to the pushdown. A future reduction $X \to YAZ$ will take this new attribute as one of the parameters. When a reduction is made, a part of the syntax tree is created from bottom-up. This means that a new node will be created, and the reduced symbols will be its successors in the syntax tree.

3.5.3 Top-down actions

In top-down translation, we recognize two types of attributes for each symbol: Synthesized attributes, that are transferred from the children to the parent, and inherited attributes, which we transfer from the parent to the children. The nodes of the syntax tree have unknown synthesized parameters when a rule is first applied. We can, however, mark the locations of the future attribute destinations in the syntax tree on the pushdown and whenever a terminal is popped, the token's attribute may be put in the correct place in the syntax tree. When a synthesized attribute is updated, inherited attributes may be modified as a result. Individual actions, applied when a rule is expanded on the pushdown, will reserve the space for the information obtained later in the parsing process in addition to creating the part of the syntax tree.

Chapter 4

LSCELR Parsing

In this chapter, we introduce LSCELR (Lookahead Source Conflict Elimination LR) parsing. LSCELR is a new algorithm for generating minimal LR(1) parsers. It was inspired by IELR parsing, and it was developed to have the same power as IELR and canonical LR parsers with conflict resolution. Even non-LR grammars can be parsed with the help of conflict resolution. The top-level algorithm is similar to IELR: we create an LALR automaton and we identify all LR conflicts. Then, we will annotate states that contribute to those conflicts and split them where needed.

The new algorithm's originality is in its new model of the LR automaton. The traditional LR(1) automaton stores lookaheads for its LR(0) items and doesn't track the origin of those lookaheads. Although there exist algorithms to recompute that information, we already know this information during the construction of the automaton. We introduce an explicit lookahead model that represents the relationships between individual items' lookaheads within the items. When creating the automaton, we save the propagation paths of all lookahead symbols in individual items, which allows us to easily identify all conflicts created by merging isocores. IELR has to recompute these dependencies in the following phases of the algorithm, whereas LSCELR never throws away this information. In this chapter, we will introduce this explicit model and we will introduce the algorithms needed to create LSLALR (Lookahead Source LALR) and LSCELR parsers.

Throughout this chapter, we will show examples of automatons and algorithms on two context-free grammars. Figure 4.1 shows a context-free grammar with four sentences $aaa\$$, $aaaa\$$, $bab\$$ and $baab\$$. If we consider terminal a to be left-associative, we can accept three sentences $aaa\$$, $bab\$$ and $baab\$$ with canonical LR(1) parsers. LALR parsers are no longer able to parse $baab\$$ because of state merging. We will demonstrate that LSCELR doesn't have this deficiency. Figure 4.2 shows a context-free grammar that has a reduce-reduce

$$P = \left\{ \begin{array}{l} 0 : S' \to S\$, \\ 1 : S \to aAa, \\ 2 : S \to bAb, \\ 3 : A \to a, \\ 4 : A \to aa \end{array} \right\}$$

Figure 4.1: An unambiguous grammar (see Fig. 1 in [4])

$$P = \left\{ \begin{array}{ll} 0 : S' \to S\$, & 5 : B \to Cb, \\ 1 : S \to aA, & 6 : B \to Da, \\ 2 : S \to bB, & 7 : C \to xx, \\ 3 : A \to Ca, & 8 : D \to xx \\ 4 : A \to Db, & \end{array} \right\}$$

Figure 4.2: An unambiguous grammar with a reduce-reduce conflict in LALR

conflict in LALR parsers. We will demonstrate that LSCELR eliminates these conflicts by splitting states of the LALR automaton.

4.1 LS Items and the LS Automaton

First, we will introduce *LS items* and the *LS automaton*. We modify regular LR(1) items so that they contain the marked production, generated lookahead symbols and references to other items where the rest of lookaheads are propagated from.

Definition 4.1.1. An *LS item* is a triple $I = (A \to \alpha \cdot \beta, g, s)$, denoted by $[A \to \alpha \cdot \beta, g, s]$, where

- $A \to \alpha \cdot \beta$ is an LR(0) item
- $g \subseteq T$ is the set of generated lookahead symbols
- $s \subseteq \mathbb{N} \times \mathbb{N}$ is the set of references to LS items that are sources of lookaheads for this item. The first number in the pair references a state in the automaton, and the second references an item in that state.

Unlike an LR(1) item, not all lookaheads are directly enumerated in its lookahead set. Instead, we reference other items that lookaheads are propagated from.

Definition 4.1.2. An *LS state* is a pair $S = (I, GOTO)$, where

- $I = \{I_1, I_2, \ldots, I_{|I|}\}$ is an LS item
- $GOTO : (T \cup N) \to \mathbb{N}$ is the transition function.

LS items in I are explicitly marked by their numerical identifier. For an LS state $S = (I, GOTO)$ and $i \in \langle 1, |I| \rangle$, $S[i]$ denotes $I_i \in I$. We say that two states $S_i = (I_i, GOTO_i)$ and $S_j = (I_j, GOTO_j)$ are *isocores* iff $|I_i| = |I_j| \wedge \forall [p, a, b] \in I_i : \exists [p, a, c] \in I_j$.

Definition 4.1.3. An *LS automaton* is a set of LS states $S = \{S_1, S_2, \ldots, S_{|S|}\}$.

LS states in S are explicitly marked by their numerical identifier: For an LS automation S and $i \in \langle 1, |S| \rangle$, $S[i]$ denotes $S_i \in S$. For two LS states $S_i = (I_i, GOTO_i) \in S$ and $S_j = (I_j, GOTO_j) \in S$, $S_i \xrightarrow{x} S_j$ denotes $GOTO_i(x) = j$.

Algorithm 4.1 $lookaheads(S, [p, g, s], examined)$

Input: LS automaton S, LS item $[p, g, s]$, a set of examined LS items $examined$
Output: a set of lookahead symbols $la \subseteq T$

```
if [p, g, s] ∈ examined then
    return ∅
e := examined ∪ {[p, g, s]}
return g ∪ ⋃_{(state,item)∈s} lookaheads(S, S[state][item], e)
```

The full lookahead set for each LS item is calculated by traversing the item's predecessors. We introduce the procedure for computing the lookahead set for LS items in Algorithm 4.1. In order to obtain the set of lookahead symbols for an item i, we must call this procedure with an empty set as the last argument: $lookaheads(automaton, i, \varnothing)$. The last argument is

the set of already examined states, and is needed to stop infinite recursion in case of a cyclical dependency of lookahead sources. To obtain the lookahead set for an LS item, we mark this item as examined and we return the union of this item's generated lookaheads and the lookaheads of its lookahead sources. Note that the lookahead sets obtained when calling this procedure with a non-empty examined set don't necessarily match the actual lookahead sets of those items: consider items with circular dependencies when one of the items has been already been examined up the recursion stack.

Basic idea. Items contain their generated lookahead symbols and the source items of the rest of their lookahead symbols. We get the full lookahead set by obtaining the lookahead sets of the sources and adding them to the set of generated lookaheads. If there is a circular dependency, we need stop infinite recursion by examining each state at most once in each dependency path. If we examine any state for the second time, we can return the empty set (any subset of g can be returned here): the same state is already being examined up in the recursion stack, and its lookaheads will be added properly there.

When computing the full lookahead sets for the whole automaton, we can do this efficiently by computing the lookahead sets in ascending order. Then, the full lookahead sets for previous states have already been computed and don't require the recursive procedure for their computation. Alternatively, we could change the order of computation after the first state so that we always prioritize states in $GOTO$ of the current state.

Algorithm 4.2 $closure(I)$

Input: an ordered set of LS items $I = \{I_1, I_2, \ldots, I_{|I|}\}$
Output: LS state $S = (I, GOTO)$
  repeat
    for all items $[A \to \alpha{\cdot}B\beta, g, s]$ in I, where $B \in N$, $\alpha, \beta \in (N \cup T)^*$ **do**
      for all rules $B \to C \in P$, where $C \in (N \cup T)^*$ **do**
        $f = first(\beta)$
        if $\varepsilon \in f$ **then** $S = s$ and $G = g \cup f - \{\varepsilon\}$
        else $S = \varnothing$ and $G = f$
        if $\exists I_x = [B \to {\cdot}C, g_x, s_x] \in I$ **then**
          $I_x \coloneqq [B \to {\cdot}C, g_x \cup G, s_x \cup S]$
        else
          let $I_{|I|+1} = [B \to {\cdot}C, G, S]$
          add $I_{|I|+1}$ to I
  until no more items are added or modified in an iteration
  return $(I, \varnothing)$

In Algorithm 4.2, we modify Algorithm 3.10 to account for the different form of items. The item set will contain the appropriate generated lookaheads for each item, and will link to lookahead sources where the item's lookahead is reliant on previous states.

Basic idea. The closure of a set of LS items is computed in the same way a closure of LR(1) items would be. Because we store sets of lookaheads and lookahead sources, we check whether an item with the same production part is in the set already, and we add the generated lookaheads and lookahead sources instead of inserting a new item. If either the set of generated lookaheads or the set of lookahead sources is modified, or a new item has been added we continue the closure procedure. After several iterations, there will be no more productions to add and all generated lookaheads and lookahead sources will have been

distributed to their respective items. If the first set of the suffix of a production contains ε, the lookahead symbols of the original item become lookahead symbols of the closure item as well. We achieve this by adding both the generated lookaheads and lookahead sources to the new item.

4.2 The LSCELR Parser

We create the LSCELR parser in the following phases:

1. Generate the LSLALR automaton
2. Detect all states with shift-reduce and reduce-reduce conflicts.
3. Mark the conflict contribution sources in the state machine.
4. Split states potentially contributing to conflicts.
5. Calculate the final lookahead set for all states and resolve the remaining conflicts.

We begin by creating the LSLALR automaton as described in Section 4.2.1. If the LSLALR automaton is constructed for an LALR grammar, no conflicts will be present and steps 2, 3 and 4 will be skipped. If there are any reduce-reduce or shift-reduce conflicts, we will go backwards through the automaton, marking the potential conflict contributions in each state. All states along that path that can be reached from multiple other states will be marked, and those transitions will be reexamined. Only a single predecessor will be preserved, and only states that make the same conflict contributions to their successor reduce states will be merged. The resulting automaton does not contain any conflicts a canonical LR(1) parser doesn't have, but the number of states will be smaller for most non-trivial grammars.

The phases of this algorithm, as well as its general idea, were heavily influenced by the IELR algorithm by Denny and Malloy (see [4]). The originality of our approach is in the LS automaton model. We employ the explicit lookahead source model to determine the conflicts and their sources instead of the analysis methods used by Denny and Malloy. In principle, the resulting parsers should be nearly identical, although we have not considered any minimization of the parser after conflict resolution. Thanks to the explicit state dependency tracking, this approach for creating minimal LR(1) parsers is arguably easier to follow than IELR and might prove useful for teaching the principles of creating minimal LR(1) parsers. It is also subjectively easier to reason about the correctness of the approach, although no formal proof of correctness is made for LSCELR here.

4.2.1 Phase 1: The LSLALR Automaton

In the first phase, we generate the LS equivalent of the LALR automaton. Unlike LALR, we don't discard the information about where each lookahead symbol comes from. This lets us obtain an explicit enumeration of all lookahead sources for each item in the automaton. We will use that information in the next phase, when we mark the conflicts each state potentially contributes to.

Algorithm 4.3 shows the creation of the successor states of an LS state. In traditional parsing algorithms, this was referred to as the GOTO function. When we move over a symbol in a state, we create a link to that state in items of the successor state. This means that the set of generated lookahead symbols for successor states is always empty here, and we

Algorithm 4.3 next(S_n)

Input: LS state $S_n = (I, GOTO)$
Output: a set of successor transitions $(y \in (N \cup T), S_o)$, where S_o is an LS state
  $transitions := \varnothing$
  for all $I_x = [A \to \alpha \cdot y\beta, g, s] \in I$, where $y \in (N \cup T) - \{\$\}$, $\alpha, \beta \in (N \cup T)^*$ **do**
    let $nextitem = [A \to \alpha y \cdot \beta, \varnothing, (n, x)]$
    if $\exists (y, W) \in transitions$ **then**
      $W \coloneqq W \cup nextitem$
    else
      insert $(y, \{nextitem\})$ to $transitions$
  for all $(symbol, W) \in transitions$ **do**
    $W \coloneqq closure(W)$
  return $transitions$

reference the original state in the set of lookahead sources. This way, when any state changes its set of lookahead symbols, its successors in the automaton will be updated implicitly: when the set of lookahead symbols for any successor is calculated, lookaheads from all predecessors are added.

Algorithm 4.4 expand(S, n)

Input: LS automaton S, $n \in \mathbb{N} : S_n \in S$
Output: expanded LS automaton S
  let $S_n = (I, GOTO)$
  for all $(x, (I', GOTO')) \in next(S_n)$ **do**
    if $\exists j : compatible((I', GOTO'), S_j)$ **then**
      for all $I_k = [p, g, s] \in S_j$ **do**
        let $(p, g, s_1) \in I'$
        $I_k \coloneqq [p, g, s \cup s_1]$
      insert (x, j) to $GOTO$
    else
      insert $(x, |S| + 1)$ to $GOTO$
      insert $S_{|S|+1} = (I', GOTO')$ to S (it becomes $S_{|S|}$)
      $S \coloneqq expand(S, |S|)$
  return S

Algorithm 4.4 describes the recursive procedure for generating states in the LS automaton. We use a compatibility test to find an existing state that a successor state could be merged with. We construct the compatibility tests so that there always exists at most one compatible existing state. If a compatible state is found, we merge the lookahead source sets for each item. This way, the lookaheads from both predecessors are found for this items and all its successors in the automaton.

Algorithm 4.5 shows the creation of the LSLALR automaton. We begin by generating the first state as the closure of $[S' \to \cdot S\$, \{\$\}, \varnothing]$. Then, we recursively generate the new states' successors and selectively merge them. Here, we set the compatibility test to

$$compatible(S_x, S_y) \iff isocores(S_x, S_y)$$

Algorithm 4.5 LSLALR automaton

Input: augmented context-free grammar $(N \cup \{S'\}, T \cup \{\$\}, P, S')$
Output: LSLALR LS automaton S
$S := \{S_1 = closure(\{[S' \to \cdot S\$, \{\$\}, \varnothing]\})\}$
$S := expand(S, 1)$
return S

to ensure that all isocores are merged.

Basic idea. The construction of the LSLALR automaton is analogous to Algorithm 3.15. Instead of constructing sets of LR(0) items, we merge states with matching LR(0) items. Generated lookaheads don't play a role in this matching, because they will always be the same for the same set of LR(0) items. They are the same because the procedure next never transfers any generated lookaheads (see Algorithm 4.3), and the rest of generated lookaheads is only generated from the LR(0) items contained in the LS items. Instead of figuring out which lookaheads are generated, we already have this information enumerated in the items. The third and fourth phases of 4.63 are simulated when getting the full lookahead sets of each item.

state	$[p,$	$g,$	$s]$	next state lookaheads
1.	$S \to \cdot aAa,$	$\{\$\},$	$\varnothing$	**S3**
	$S \to \cdot bAb,$	$\{\$\},$	$\varnothing$	**S8**
	$S' \to \cdot S\$,$	$\{\$\},$	$\varnothing$	**G2**
2.	$S' \to S \cdot \$,$	$\varnothing,$	$\{(1,3)\}$	$\{\ \$\ \}$
3.	$S \to a \cdot Aa,$	$\varnothing,$	$\{(1,1)\}$	**G6**
	$A \to \cdot a,$	$\{a\},$	$\varnothing$	**S4**
	$A \to \cdot aa,$	$\{a\},$	$\varnothing$	**S4**
4.	$A \to a\cdot,$	$\varnothing,$	$\{(3,2),(8,2)\}$	$\{a, b\}$
	$A \to a \cdot a,$	$\varnothing,$	$\{(3,3),(8,3)\}$	~~**S5**~~
5.	$A \to aa\cdot,$	$\varnothing,$	$\{(4,2)\}$	$\{a, b\}$
6.	$S \to aA \cdot a,$	$\varnothing,$	$\{(3,1)\}$	**S7**
7.	$S \to aAa\cdot,$	$\varnothing,$	$\{(6,1)\}$	$\{\$\}$
8.	$S \to b \cdot Ab,$	$\varnothing,$	$\{(1,2)\}$	**G9**
	$A \to \cdot a,$	$\{b\},$	$\varnothing$	**S4**
	$A \to \cdot aa,$	$\{b\},$	$\varnothing$	**S4**
9.	$S \to bA \cdot b,$	$\varnothing,$	$\{(8,1)\}$	**S10**
10.	$S \to bAb\cdot,$	$\varnothing,$	$\{(9,1)\}$	$\{\$\}$

Table 4.1: LSLALR automaton for the grammar from Figure 4.1

In Table 4.1, we show a LSLALR for the grammar from Figure 4.1. We can see that each state contains all predecessors that affect its items' lookahead sets. We can trace back through the states to obtain the full lookahead states. For example, the first item of state 4 has the lookahead set $\{a, b\}$, where lookahead symbol a was generated in state 3 and b was generated in state 8. Notice that if we applied left associativity to a to resolve the shift-reduce conflict in state 4, state 5 becomes unreachable.

4.2.2 Phase 2: Conflict Detection

In the second phase, we detect all conflicts in all states. Only states with at least one reduce item (that is, with an item where the dot is at the last position of the production) is present. We calculate the lookahead set for each LS item and we decide which parser actions this LALR parser would take. If there are any conflicts, we mark all lookaheads that cause a reduce action in all of these conflicts for each item in these states. We will obtain a set of conflicted symbols, which we will use in phase 3 to mark the conflicted lookahead propagation paths.

Algorithm 4.6 Conflict detection

Input: LS automaton S
Output: a set of conflicted state indices C, and a function $conflicts : \mathbb{N} \times \mathbb{N} \to 2^T$

```
C := ∅
conflicts(x, y) := ∅
for all S_i = (I, GOTO) ∈ S do
    if there is at least one reduce item in I then
        let actions = {(s, j) : I_j = [X → Y·, x, y] ∈ I, s ∈ lookaheads(S, I_j, ∅)}
        let c = {(s, j) : ∃k ≠ j : (s, k) ∈ actions ∨ I_k = [X → Y · sZ, x, y] ∈ I}
        if c ≠ ∅ then
            C := C ∪ {i}
            conflicts(i, j) := {s : (s, j) ∈ c}
return (C, conflicts)
```

We obtain the set of states with conflicts and which symbols the conflicts are on in Algorithm 4.6. We examine all states with at least one reduce state. If there are multiple actions possible on any symbol, we mark that state as conflicted and we add the conflicted symbols to the reduce items where the symbol's presence in the lookahead set causes that conflict.

As an example, in Table 4.1, there is a shift-reduce conflict on a in state 3. Algorithm 4.6 will recognize the conflicted state S_3, and will mark $conflicts(4, 1) = \{a\}$. Because the shift action will always be generated in all isocores, $conflicts(4, 2)$ will be empty.

4.2.3 Phase 3: Marking Conflict Contributions

In this phase, we will mark items with all symbols that they can potentially contribute to conflicts with. We recursively iterate over lookahead sources and we mark non-generated lookahead symbols that lead to conflicts in the LSLALR automaton. We will be able to use this information to keep some states merged, and to split states that contribute to different conflicts, or that change already existing conflicts.

Algorithm 4.7 defines the procedure for marking conflicts for individual items in the LS automaton. We first check whether there are any lookahead sources for this item and whether the set of conflicted symbols *contributions* isn't empty. In case there are no lookahead sources, this item's lookahead set is fully dependent on its set of generated lookaheads. This means that this item is not affected by any state merging. If the set of conflicted symbols is empty, we have no more conflicts to mark. In either of these cases, we can return without any modifications to the *contributions* function.

We remove all generated lookaheads from the set *contribution*, because these lookaheads are always propagated to successor states regardless of lookahead sources. We then add

Algorithm 4.7 mark(S, *contributions*, i, j, *contribution*)

Input: LS automaton S, $\mathit{contributions} : \mathbb{N} \times \mathbb{N} \to 2^T$, $i \in \mathbb{N}$, $j \in \mathbb{N}$, $\mathit{contribution} \in 2^T$
Output: $\mathit{contributions} : \mathbb{N} \times \mathbb{N} \to 2^T$

```
let [p, g, s] = S[i][j]
if s = ∅ ∨ contribution = ∅ then
    return contributions
contribution := contribution − g
contributions(i, j) := contributions(i, j) ∪ contribution
if no new contributions were added to contributions(i, j) then
    return contributions
for all (state, item) ∈ s do
    contributions := mark(S, state, item, contribution)
return contributions
```

contribution to this item's *contributions*. If no symbols were added, they have already been marked in the item's lookahead sources as well. Finally, we propagate *contribution* to all lookahead source states by recursively calling this procedure, modifying *contributions* in every call.

Algorithm 4.8 Marking Conflict Contributions

Input: LS automaton S, a set of conflicted state indices C, and a function $\mathit{conflicts} : \mathbb{N} \times \mathbb{N} \to 2^T$
Output: $\mathit{contributions} : \mathbb{N} \times \mathbb{N} \to 2^T$

```
contributions(x, y) := ∅ for all x, y ∈ ℕ
for all i ∈ C do
    let S_i = (I, GOTO)
    for all I_j ∈ I do
        contributions := mark(S, contributions, i, j, conflicts(i, j))
```

We describe the method for marking all conflict contributions for all items in Algorithm 4.8. We call *mark* for each item in all conflicted states.

Basic idea. At the end of this procedure, we have the function *contributions*. The set $\mathit{contributions}(s, i)$ is the set of terminals that contribute to a potential conflict, if they are passed to that item from any lookahead source. When $j \in \mathit{contributions}(s, i)$, then if the terminal j is in the lookahead set of $I_i \in S_s$, there exists an item $I_k \in S_l$ in the LSLALR automaton where a reduce on j will happen because of I_i. Not all such terminals are marked here, just the ones that lead to a conflict in the LSLALR automaton. Items with no lookahead sources aren't affected by any state merging, so they aren't marked.

Consider Table 4.2. The reduce-reduce conflict on a and b in state 7 has two lookahead sources and neither a, nor b are generated lookahead symbols. We add a and b to the set of conflict contributions for items 1 and 2 in state 7. State 6 has no generated lookaheads, so both items are marked with the same conflict contributions. Items 4 and 5 in states 3 and 11 have no lookahead sources: even if some contributions remain after subtracting the set of the individual items' generated lookaheads, no state merging affects the lookaheads of these items.

state	$[p,$	$g,$	$s]$	contributions
			...	
3.	$S \to a \cdot A,$	$\varnothing,$	$\{(1,1)\}$	$\varnothing$
	$A \to \cdot Ca,$	$\varnothing,$	$\{(1,1)\}$	$\varnothing$
	$A \to \cdot Db,$	$\varnothing,$	$\{(1,1)\}$	$\varnothing$
	$C \to \cdot xx,$	$\{a\},$	$\varnothing$	$\varnothing$
	$D \to \cdot xx,$	$\{b\},$	$\varnothing$	$\varnothing$
			...	
6.	$C \to x \cdot x,$	$\varnothing,$	$\{(3,4),(11,4)\}$	$\{a,b\}$
	$D \to x \cdot x,$	$\varnothing,$	$\{(3,5),(11,5)\}$	$\{a,b\}$
7.	$C \to xx\cdot,$	$\varnothing,$	$\{(6,1)\}$	$\{a,b\}$
	$D \to xx\cdot,$	$\varnothing,$	$\{(6,2)\}$	$\{a,b\}$
			...	
11.	$S \to b \cdot B,$	$\varnothing,$	$\{(1,2)\}$	$\varnothing$
	$B \to \cdot Cb,$	$\varnothing,$	$\{(1,2)\}$	$\varnothing$
	$B \to \cdot Da,$	$\varnothing,$	$\{(1,2)\}$	$\varnothing$
	$C \to \cdot xx,$	$\{b\},$	$\varnothing$	$\varnothing$
	$D \to \cdot xx,$	$\{a\},$	$\varnothing$	$\varnothing$
			...	
16.			...	

Table 4.2: Conflict contributions in LSLALR automaton for the grammar from Figure 4.2

4.2.4 Phase 4: Splitting States

Finally, we use the information obtained in the previous phases to split some states of the LSLALR automaton to create the LSCELR automaton. We will reexamine transitions to all states that contribute to a conflict and have multiple predecessors, and only merge them with existing states if their lookahead symbols contribute to the same conflicts on the same items.

In order not to introduce any new conflicts by merging two potentially conflicted states, we must ensure that any conflict contributions in both states' items are exactly the same. If they are different, merging them will introduce or modify conflicts in one of their successors. Recall that potential contributions affect conflicts created by merging states; in order to keep the power of canonical parsers, only isocores with the same potential contributions may be merged.

In this phase, we need to change the compatibility test. To determine whether two states S_x and S_y are compatible, we need to examine their lookaheads and compare their potential conflict contributions. If they are the same for all items, we can merge them. Algorithm 4.9 describes the compatibility test for the fourth phase of the LSCELR algorithm. We use the set of conflict contributions calculated from the LSLALR automaton to determine whether the two states have identical contributions to potential conflicts. We find the isocore state from that automaton by finding the isocore with the smallest index. From that index, we are able to get the set of potentially conflicted lookaheads for each item. After performing set intersection between this set and the full set of lookahead symbols for each item, we are able to compare them between the two states.

Algorithm 4.9 Compatibility test for phase 4 of LSCELR

Input: LS automaton S, $contributions : \mathbb{N} \times \mathbb{N} \to 2^T$, and LS states $S_x = (I_x, GOTO_x)$ and $S_y = (I_y, GOTO_y)$
Output: **True** or **False**
  if S_x and S_y are not isocores, **return False**
  let z be the smallest number such that S_x and $S_z = (I_z, GOTO_z) \in S$ are isocores
  for all $I_j = [i, g, s_z] \in I_z$ **do**
    let $[i, g, s_x] \in I_x$
    let $[i, g, s_y] \in I_y$
    $l_1 = lookaheads(S, [i, g, s_x], \varnothing) \cap contributions(z, j)$
    $l_2 = lookaheads(S, [i, g, s_y], \varnothing) \cap contributions(z, j)$
    if $l_1 \neq l_2$ **then**
      return False
  return True

Algorithm 4.10 Splitting States

Input: LSLALR automaton S, $contributions : \mathbb{N} \times \mathbb{N} \to 2^T$
Output: the set of split lookahead sources $split \subseteq \mathbb{N} \times \mathbb{N}$, LS automaton S
  $split = \varnothing$
  for all $S_n = (I, GOTO) \in S$ **do**
    let $I[x] = [p_x, g_x, s_x]$ be a kernel item
    if $\exists I_i = [p_i, g_i, s_i] \in I : contributions(n, i) \neq \varnothing \wedge |\{s : (s, c) \in s_x\}| > 1$ **then**
      let a be the smallest number such that $(a, b) \in s_i$
      for all $I_j = [p, g, s] \in I$ **do**
        if $s \neq \varnothing$ **then**
          if $j = 1$ **then**
            $split \coloneqq split \cup \{s - \{(a, c) \in s\}\}$ for all c
          $s \coloneqq \{(a, c) \in s\}$ for all c
  return $split$, S

We remove all but the first lookahead source in states where there exists an item with some potential conflict contributions, and there are at least two different lookahead source states in the first kernel item. States with no conflict contributions don't need to be split, and states with only one lookahead source cannot be split, since they only have a single predecessor in the automaton. States that satisfy both of these conditions could possibly remove some conflicts if they were split into multiple different states. In Algorithm 4.10, we iterate over all states that satisfy both of these conditions. We only keep the lookahead source to the first source state, and we return the rest of the lookahead sources in the *split* set.

The *split* set now contains all states that need to be reexamined. Algorithm 4.11 shows the procedure of adding the split states back to the automaton, creating the LSCELR automaton. From the marked item, we learn the symbol that was skipped to reach the state with conflict contributions and multiple lookahead sources. We generate the successor state over that symbol from this state and we reexamine it using the new compatibility test. If we cannot merge it with any existing state, we add this new state to the automaton and call *expand* to generate its successors. *expand* uses the new compatibility test to merge

Algorithm 4.11 Regenerating States

Input: LS automaton S, *contributions* : $\mathbb{N} \times \mathbb{N} \to 2^T$, the set of split lookahead sources *split* $\subseteq \mathbb{N} \times \mathbb{N}$
Output: LSCELR automaton S

```
for all (i, j) ∈ split do
    let S[i] = (I, GOTO)
    let I[j] = [X → Y · zZ, g_j, s_j], z ∈ N ∪ T
    let (z, S_next = (I_next, GOTO_next) ∈ next(S[i])
    if ∃S_k = (I_k, GOTO_k) ∈ S : compatible(S_k, S_next) then
        for all I_i = [p, g, s_i] ∈ I_k do
            let [p, g, s_next] ∈ I_next
            s_i := s_i ∪ s_next
        GOTO(z) := k
    else
        insert new state S_next to S (it becomes S_|S|)
        GOTO(z) := |S|
        S := expand(S, |S|)
return S
```

new states as well, ensuring that no extra conflicts are created. In either case, we update the *GOTO* function for the reexamined state so that it contains the correct successor state.

Basic idea. We split all but one source from states that contribute to conflicts in the LSCELR automaton. The compatibility test doesn't let us merge two states unless it contributes the same lookaheads to potential conflicts. If we cannot merge a state, we insert it as a new state to the automaton, and it becomes a new option for future state merging. Lookaheads that are not marked as contributions in the original LALR state don't matter when merging states, because they didn't cause any conflicts even when all isocores were merged. If a new state is added, we expand it to insert its successors into the automaton. If its successor states can be merged with the existing states, they will be. If they cannot, they will be added as new states and recursively expanded further. This way, we avoid all conflicts created or modified by merging two states.

In Table 4.3, we demonstrate splitting states as a continuation of Table 4.2. We remove all lookahead sources from state 6's items but the first, because these items have both nonempty conflict contributions and they have multiple lookahead sources. We now generate the successor state for items 4 and 5 in state 11. This generated state's contributions don't match the contributions of the stripped state 6, so we add the new state to the state machine. We modify state 11's shift actions accordingly. Its successor state cannot be merged with 6 for the same reasons and state 18 is created. By splitting these two states, we have eliminated the reduce-reduce conflict.

Because we only merge two states if their potentially conflicted lookaheads match, we can safely cache the lookaheads of existing states and we don't have to recompute them each time we try to merge a new state with them. This is an important optimization of the algorithm, since computing the full set of lookaheads is an expensive operation (we have to traverse the predecessor states). Another minor optimization is early exit for the lookahead retrieval algorithm when all potentially conflicted lookaheads have been found. Since potentially conflicted lookaheads for items are usually a relatively small subset of terminals, we can

state	$[p,$	$g,$	$s]$	next state lookaheads	potential contributions
			...		
3.	$S \to a \cdot A,$	$\varnothing,$	$\{(1,1)\}$	**G8**	$\varnothing$
	$A \to \cdot Ca,$	$\varnothing,$	$\{(1,1)\}$	**G9**	$\varnothing$
	$A \to \cdot Db,$	$\varnothing,$	$\{(1,1)\}$	**G4**	$\varnothing$
	$C \to \cdot xx,$	$\{a\},$	$\varnothing$	**S6**	$\varnothing$
	$D \to \cdot xx,$	$\{b\},$	$\varnothing$	**S6**	$\varnothing$
			...		
6.	$C \to x \cdot x,$	$\varnothing,$	$\{(3,4), \sout{(11,4)}\}$	**S7**	$\{a,b\}$
	$D \to x \cdot x,$	$\varnothing,$	$\{(3,5), \sout{(11,5)}\}$	**S7**	$\{a,b\}$
7.	$C \to xx\cdot,$	$\varnothing,$	$\{(6,1)\}$	$\{a, \sout{b}\}$	$\{a,b\}$
	$D \to xx\cdot,$	$\varnothing,$	$\{(6,2)\}$	$\{\sout{a}, b\}$	$\{a,b\}$
			...		
11.	$S \to b \cdot B,$	$\varnothing,$	$\{(1,2)\}$	**G14**	$\varnothing$
	$B \to \cdot Cb,$	$\varnothing,$	$\{(1,2)\}$	**G15**	$\varnothing$
	$B \to \cdot Da,$	$\varnothing,$	$\{(1,2)\}$	**G12**	$\varnothing$
	$C \to \cdot xx,$	$\{b\},$	$\varnothing$	**~~S7~~ S17**	$\varnothing$
	$D \to \cdot xx,$	$\{a\},$	$\varnothing$	**~~S7~~ S17**	$\varnothing$
			...		
17.	$C \to x \cdot x,$	$\varnothing,$	$\{(11,4)\}$	**S18**	$\{a,b\}$
	$D \to x \cdot x,$	$\varnothing,$	$\{(11,5)\}$	**S18**	$\{a,b\}$
18.	$C \to xx\cdot,$	$\varnothing,$	$\{(17,1)\}$	$\{b\}$	$\{a,b\}$
	$D \to xx\cdot,$	$\varnothing,$	$\{(17,2)\}$	$\{a\}$	$\{a,b\}$

Table 4.3: Splitting states to create the LSCELR automaton for the grammar from Figure 4.2

check if we haven't already found all of them before continuing recursively looking up more lookahead symbols.

4.2.5 Phase 5: Generating the Parser and Conflict Resolution

As a final step, we generate the parser. First, we calculate all lookahead symbols for all reduce states using Algorithm 4.1 efficiently in ascending order, as described in Section 4.1. The rest of the parser generation is similar to the existing algorithms described in the previous chapter. The creation of the action and goto tables for LSCELR parsers is described in Algorithm 4.12. The goto table is created from the GOTO function in each LS item. In the action table, the reduce action is added to the items with the terminal symbols matching the lookahead symbols in the state's items.

4.3 Properties of LSCELR Parsers

With LSCELR parsers, we are able to achieve the power of canonical LR(1) parsers. Canonical parsers often have a very large number of states, which makes them unsuitable for practical use, and most practical applications use LALR parsing instead. LSCELR

Algorithm 4.12 LSCELR action and goto

Input: augmented context-free grammar G, LSCELR automaton S
Output: the action and goto tables

The parsing actions for all states $S_i = (I, GOTO) \in S$ are determined as follows:
$la \coloneqq lookaheads(S, S_i, \varnothing)$
All action entries are initialized as an error.
If $[A \to \alpha \cdot a\ \beta, g, s]$ is in I, $a \in T$, $a \neq \$$ and $GOTO(a) = j$, set $action[i, a] \coloneqq$ shift j
If $[A \to \alpha \cdot, g, s]$ is in I, $a \in la$ and $A \neq S'$, set $action[i, a] \coloneqq$ reduce $A \to \alpha$
If $[S' \to S \cdot \$, \{\$\}, \varnothing]$ is in I, set $action[i, \$] \coloneqq$ accept
If any conflicting actions arise from the above rules and they cannot be resolved, we say the grammar is not LSCELR.
The goto transitions are created using the rule: If $S_i \xrightarrow{X \in N} S_j$, then $goto[i, X] = j$.
The initial state is S_1.

parsers do not suffer from the deficiencies of LALR parsing, but they achieve this with an order of magnitude fewer states than canonical parsers.

By changing the automaton model, we were able to avoid some explicit analysis of the LALR automaton compared to IELR. Because of this change, the analysis of the automaton is relatively straightforward and some relationships between items are preserved instead of needing to be recomputed. We were able to successfully generate minimal LR(1) parsers using this new model. The core idea of the minimal LR(1) parser however remains: all states that don't change the nature of their conflicts are merged and all states that do are split.

grammar	LALR	canonical LR	LSCELR
Figure 4.1	10	12	11
Figure 4.2	16	18	18
GAWK	435	4861	613
ANSI C	479	2623	512

Table 4.4: Numbers of states for LALR, canonical and LSCELR parsers

In Table 4.4, we compare the number of states for existing parsing algorithms. The first two rows contain the number of states for the two grammars from this chapter. These are small grammars, so the number of states is comparable to canonical parsers. GAWK is a text processing tool from the GNU collection. We have constructed the automatons for its grammar obtained from [15]. Finally, we have the numbers of states for the latest standard of ANSI C from 2011. The grammar was obtained from [2]. All of the automatons were constructed using the implementation described in the last two chapters of this thesis. We can see that the hypothesis of greatly reduced parser sizes is true for these practical grammars. The ANSI C grammar doesn't rely on associativity and precedence conflict resolution (it has these concepts encoded in the grammar itself), and its LSCELR parser is nearly identical to its LALR parser. The differences might indicate some bugs or reliance on default Bison behavior in the grammar.

LSCELR parsers can be used to generate minimal LR(1) parsers that guarantee no created conflicts over canonical parsers. Like IELR parsers, they work well with non-LR(1)

grammars with conflict resolution and they don't lose any accepted sentences by reducing the number of states.

In its current implementation, the generation of LSCELR parsers is quite slow for large grammars compared to IELR (benchmarks of IELR are in [4]). This only becomes an issue for grammars with canonical LR parsers that have thousands of states. Even very large IELR parsers can be generated in a fraction of a second, where LSCELR takes tens of seconds on newer processors. There exist multiple areas for optimization in the LSCELR algorithm that could resolve this issue, however. A hybrid approach where the explicit lookahead model is used in an IELR implementation exclusively to avoid recomputing the preserved properties of the automaton could solve this issue completely. We circumvent this issue in our framework by providing a convenient way to generate a text representation of the parsing tables as an additional optional step after translating translation grammars from their text representation. The parsing tables can then be initialized from this representation, removing the need to recompute the LSCELR parser on each initialization.

Like Denny and Malloy, we consider conflict resolution that is merge-stable. If conflict resolution isn't merge-stable (it makes decisions based on the full lookahead sets regardless of the actual conflicted symbols), LSCELR could potentially merge two states that have the same potential conflict contributions, but conflict resolution would behave differently based on the other lookahead symbols. The conflict resolution used in Bison and in our implementation is merge-stable, so this issue is not relevant here.

4.4 Other Parsing Algorithms via the LSCELR algorithm

Other parsers can be obtained by modifying the LSCELR algorithm. Here, we demonstrate how to create the equivalents of LALR and Canonical LR parsers through the LSCELR algorithm.

4.4.1 LSLALR Parsers

If we always skip phases 2, 3 and 4 from Section 4.2, we will obtain the LSLALR parser. This parser has the same properties as a LALR parser, but takes advantage of the explicit model during parser table construction. The original approach propagates lookaheads forwards through the automaton, whereas we propagate lookaheads by references to predecessors. Since LSCELR parsers are identical to LALR parsers for LALR grammars, generating LALR parsers this way poses a risk: grammars that are not LALR may lose the ability to accept some input sentences and new reduce-reduce conflicts may appear.

4.4.2 LS Canonical LR Parsers

If we skip phases 2, 3 and 4 from Section 4.2, and we set the compatibility test in phase 1 so that two states are compatible when they are isocores and their corresponding items have the same full lookahead sets, we obtain the LS canonical LR(1) parser. Here, the explicit model doesn't provide any advantages over canonical LR parsing. We do, however, get the lookahead propagation paths if we want to inspect them during grammar construction or debugging. The usefulness of this approach is mainly in studying canonical parsers. Canonical parsers will yield many more states for the same recognition power as LSCELR parsers.

Chapter 5

Translation Grammars

This chapter introduces translation grammars and some of their properties. The following is quoted from [17], with the exceptions of Section 5.1 and Section 5.2.3. The following definition is a modification of the definition from [8], separating the input and output terminal sets and changing the notation slightly.

Definition 5.0.1. A *translation grammar* is a quintuple

$$G = (N, T_I, T_O, P, S)$$

where

- N is an alphabet of *nonterminals*
- T_I is an input alphabet of *input terminals* such that $T_I \cap N = \varnothing$
- T_O is an input alphabet of *output terminals* such that $T_O \cap N = \varnothing$
- P is a finite set of *productions* in the form

  $$(A, A) \to (u_0 B_1 u_1 \dots B_n u_n, v_0 B_1 v_1 \dots B_n v_n)$$

  for $j = 1, \dots, n$, $B_j \in N$, for $i = 0, \dots, n$, $u_i \in T_I^*$ and $v_i \in T_O^*$ ($n = 0$ implies $(A, A) \to (u_0, v_0)$)
- S is the *start symbol*

Let $(A, A) \to (i, o) \in P$, where $i \in (N \cup T_I)^*$ and $o \in (N \cup T_O)^*$. Then, $A \to i$ is the *input production*, $A \to o$ is the *output production*, A is the *left-hand side* of the production, i is the *input right-hand side* of the production, and o is the *output right-hand side* of the production.
The *direct derivation* relation, denoted by $\Rightarrow$ and defined from $N \times N$ to $(N \cup T_I)^* \times (N \cup T_O)^*$, is defined as $(q, r) \Rightarrow (x, y)$ only when $q, x \in (N \cup T_I)^*$, $r, y \in (N \cup T_O)^*$, $(A, A) \to (i, o) \in P$, $q = x_0 A x_1$, $r = y_0 A y_1$, $x = x_0 i x_1$, $y = y_0 o y_1$, $occurences(x_0, n) = occurences(y_0, n)$ and $occurences(x_1, n) = occurences(y_1, n)$ for all $n \in N$.
The set of all *input sentential forms* of G if defined as

$$F_I(G) = \{i : i \in (N \cup T_I)^*, o \in (N \cup T_O)^*, (S, S) \Rightarrow^* (i, o)\}$$

The *translation* defined by G is denoted by $T(G)$ and defined as

$$T(G) = \{(i, o) : i \in T_I^*, o \in T_O^*, (S, S) \Rightarrow_G^* (i, o)\}$$

The *input language* is defined as

$$L_I(G) = \{i : (i, o) \in T(G)\}$$

The *output language* is defined as

$$L_O(G) = \{o : (i, o) \in T(G)\}$$

The *input grammar* is a context-free grammar

$$I_G = (N, T_I, \{A \to i : (A, A) \to (i, o) \in P\}, S)$$

Definition 5.0.2. A *linear translation grammar* is a translation grammar as described in Definition 5.0.1 with productions in P in the forms

$$(A, A) \to (u, x)$$

and

$$(A, A) \to (uBv, xBy)$$

where $A, B \in N$, $u, v \in T_I^*$ and $x, y \in T_O^*$

Translation grammars are essentially two context-free grammars with a common set of nonterminals and linked sets of productions. Whereas grammars generate languages, translation grammars generate binary relations called *translations*. Each production starts from a single nonterminal and generates two strings. The input and output right-hand sides of each production must contain the same nonterminals in the same order. Translation grammars define translations $T(G)$, and two languages $L_I(G)$ and $L_O(G)$. They can also be used as language generating systems, as discussed in Section 5.3.

The complexity of the input and output languages of translation grammars and linear translation grammars is the same as that of context-free and linear grammars, respectively. This can be easily proven by comparing the definitions of translation grammars and context-free grammars, or linear translation grammars and linear grammars. This implies that on their own, input and output languages generated by translation grammars define the family of context-free languages **CF**, and that input and output languages generated by linear translation grammars define the family of linear languages **LIN**.

Definition 5.0.3. Let $G = (N, T_I, T_O, P, S)$ be a translation grammar. An *augmented translation grammar* is a translation grammar

$$G' = (N \cup S', T_I \cup \{\$\}, T_O \cup \{\$\}, P \cup \{(S', S') \to (S\$, S\$)\}, S')$$

where $S' \notin N$, $\$ \notin T_I \cup T_O$ and $\$$ is the end of input symbol.

5.1 Attribute Translation Grammars

In this thesis, we introduce attribute translation grammars. In practical translation, it is often necessary to attach additional attributes to terminals. For example, programming languages often use integer literals, where the actual numeric value of that input token is attached as an additional attribute. Here, we define a formal basis for working with these attributes in the context of translation grammars. We extend the rule definition by adding a function mapping input attributes to output attributes.

Definition 5.1.1. An *attribute translation grammar* is a quintuple

$$G = (N, T_I, T_O, P, S)$$

where

- N, T_I, T_O and S have the same meaning as in translation grammars.
- P is a finite set of productions in the form

  $$[(A, A) \to (u_0 B_1 u_1 \dots B_n u_n, v_0 B_1 v_1 \dots B_n v_n), F]$$

  for $j = 1, \dots, n$, $B_j \in N$, for $i = 0, \dots, n$, $u_i \in T_I^*$ and $v_i \in T_O^*$ ($n = 0$ implies $(A, A) \to (u_0, v_0)$). $F : \mathbb{N} \to 2^{\mathbb{N}}$ is the attribute relay function. $F(x) = \varnothing$ for all x, where $x > |u_0 B_1 u_1 \dots B_n u_n|$ or $symbol(u_0 B_1 u_1 \dots B_n u_n, x) \in N$. For all other x, $F(x)$ is the set of indices of output terminals in $v_0 B_1 v_1 \dots B_n v_n$. During translation, the attribute from x is relayed to all output terminals marked in $F(x)$.

5.2 Syntax-directed translation using translation grammars

To demonstrate the use of translation grammars in syntax-directed translation, this section describes the use of translation grammars by modifying top-down predictive LL(1) parsing and bottom-up LR(1) parsing. We have previously discussed both LL and LR parsing in chapter 3. We modify these algorithms to both parse input and create output defined by a translation grammar, creating *predictive top-down translation* and *LR translation*.

5.2.1 Two-stack Pushdown Automaton as a Translation Model

Using a two-stack pushdown automaton to simulate translation grammar top-down translation is similar to using a pushdown automaton to simulate top-down parsing of context-free grammars. The following method for constructing two-stack pushdown automation from translation grammars creates a two-stack pushdown automation that accepts a language by empty input pushdown and final state. Algorithm 5.1 describes the construction of such two-stack pushdown automata.

Basic idea. There are two groups of rules in R. If an input terminal symbol is on the top of the input stack, one of the comparing rules in the form $x|ysx \to |ys \in R$, where $x \in T_I, y \in N \cup T_O, s \in Q$ is applied. These rules compare the input symbol to the top of the input nonterminal.

The other group of rules simulates the translation grammar's productions. This group of rules simulates the expansion of nonterminals on both input and output stacks. First, the input of the production is reversed and pushed to the input stack, followed by a special symbol #. Second, the correct nonterminal symbol is moved to the top of the output stack by moving the top terminals and nonterminals from the top of the output stack to the top of the input stack. When the correct nonterminal is on top of the output stack, the output right-hand side of the production is reversed and pushed to the output stack. Finally, the symbols moved to the input stack from the output stack are moved back to the output stack until the special symbol # is encountered and removed from the input stack. This group of rules ensures that the correct nonterminal is expanded in the output stack. Note that this set of rules for each production in translation grammar G does not rely on the absolute position of the expanded nonterminal in the output stack.

Algorithm 5.1 Two-stack pushdown automation simulating a translation grammar (based on the algorithm on page 47 of [11])

Input: translation grammar $G = (N, T_I, T_O, P, S)$
Require: $\# \notin (N \cup T_I \cup T_O)$
Output: two-stack pushdown automaton $M = (Q, \Sigma, \tau, R, s, S_I, S_O, F)$

```
S_I := S
S_O := S
Q := {s, −}
Σ := T_I
τ := N ∪ T_I ∪ T_O ∪ {#}
F := {s}
for all x ∈ T_I, y ∈ N ∪ T_O do
    Add x|ysx → |ys to R
for all f, x ∈ N ∪ T_O do
    Add x|f− → |fx− to R
    Add #|f− → |fs to R
for all Z : (A, A) → (b, c) ∈ P, where Z is a unique rule label do
    Add +_Z to Q
    for all d ∈ N ∪ T_O do
        Add A|ds → e#|d+_Z to R, where e = reversal(b)
    for all x ∈ N ∪ T_O ∪ {#} do
        for all B ∈ N ∪ T_O, where B ≠ A do
            Add x|B+_Z → xB|+_Z to R
        Add x|A+_Z → x|e− to R, where e = reversal(c)
```

After the successful derivation of M, the output produced by the simulated translation grammar is on the output stack. The first terminal of the generated sentence is on the top of the output stack.

5.2.2 Predictive Top-Down Translation

This section introduces the formal devices and algorithms used for predictive top-down translation using translation grammars. LL grammars are a proper subset of context-free grammars, which means that the described algorithms will only work for a proper subset of translation grammars.

Definition 5.2.1. A translation grammar $G = (N, T_I, T_O, P, S)$ is an *LL translation grammar* if its input grammar $G_I = (N, T_I, P', S)$ is an LL grammar and $|P| = |P'|$.

Algorithm 5.2 describes the predictive top-down translation method. This method is based on the two-stack automaton described in section 5.2.1 and algorithm 7.17 from [10]. The contents of stacks *ipd* and *opd* are written head-first: the first symbol is the symbol on top of the stack. In addition to regular stack functionality (*POP*, removing the top element from the stack and *PUSH*), both stacks support the operation *REPLACE*. The operation REPLACE(n, x), where $n \in N$, $x \in (N \cup T_I \cup T_O)^*$ replaces the symbol n closest to the top of the stack with the string x. The first symbol in x will be closest to the top of the stack. This behavior can be achieved with two stacks with only PUSH and POP (as described in 5.2.1), but we introduce the operation REPLACE for brevity.

Algorithm 5.2 Predictive top-down translation

Input: translation grammar $G = (N, T_I, T_O, P, S)$, its predictive table α_G and the input string x, where $x \in T_I^*$.
Output: output string $r \in T_O^*\{\$\}$ if $x \in L_I(G)$, **ERROR** otherwise

```
ipd := S$, opd := S$
r := ε
n := 1
repeat
    let X denote the current ipd top symbol
    let t := symbol(x$, n)
    switch X:
        case X = $:
            if t = $ then SUCCESS, else ERROR
        case X ∈ T_I:
            if t = X then increment n and POP ipd, else ERROR
        case X ∈ N:
            if α(X, t) = null then
                ERROR
            else
                α(X, t) = (X, X) → (i, o)
                REPLACE(X, i) in ipd
                REPLACE(X, o) in opd
until SUCCESS or ERROR
if SUCCESS then
    repeat
        let s denote the current opd top symbol
        r := rs
        POP opd
    until opd is empty
```

5.2.3 LR(1) Translation

Algorithm 5.3 describes the predictive bottom-up translation method. This is an application of the LR(1) algorithm as introduced in this thesis. The stacks *ipd* and *opd* are the same as in the previous section.

Basic idea. The first half of the algorithm is an implementation of the LR(1) parsing algorithm. Its output are the productions to achieve the rightmost parse in reverse. If the parsing portion of the algorithm has been successful, we now have the reversed rightmost parse productions on the output stack. We reverse them by putting them on the input stack one by one behind a special separator. We now have the rightmost parse productions available on the input stack.

Then, we start applying the productions on the output stack. As it is a rightmost parse, we store the output of each productions on the output stack reversed. That way, we always replace the rightmost nonterminal. After we are done applying the productions, we now have the reversed output sentence stored on the output stack. We can now pop them one by one and add the symbols to the beginning of the output sentence. After we have emptied the output stack, we now have the output of the translation.

Algorithm 5.3 Bottom-up LR translation

Input: translation grammar $G = (N, T_I, T_O, P, S)$, its $action_G$ and $goto_G$ tables and the input string x, where $x \in T_I^*$.

Output: output string $r \in T_O^*\{\$\}$ if $x \in L_I(G)$, **ERROR** otherwise

```
ipd ≔ 0, opd ≔ ε
r ≔ ε
n ≔ 1
repeat
    let state denote the current top of ipd
    let t ≔ symbol(x$, n)
    switch action_G[state, t]:
        case SUCCESS:
            SUCCESS
        case error:
            ERROR
        case shift i:
            PUSH i to ipd and increment n
        case reduce X → (i, o):
            PUSH X → (i, o) to opd
            POP |i| symbols from ipd
            let state denote the current top of ipd
            PUSH goto_G[state, X] to ipd
until SUCCESS or ERROR
if SUCCESS then
    push a separator ‖ to ipd
    move the contents of opd to ipd, reversing them in the process
    opd ≔ S$
    repeat
        let X → (i, o) denote the current ipd top symbol
        REPLACE(X, reverse(o))
        POP ipd
    until ‖ is at the top of ipd
    while opd is not empty do
        let s be the top symbol of opd
        r ≔ sr
        POP opd
```

5.3 Descriptional Complexity of Translation Grammars

This section demonstrates that translation grammars can be used to define any recursively enumerable language (**RE**) by simulating queue grammars.

Lemma 5.3.1. Recall Lemma 2.38. in [13]. Let Q' be a queue grammar. Then, there exists a queue grammar

$$Q = (V, T, W' \cup \{\$, f\}, \{f\}, R, g)$$

such that $L(Q') = L(Q)$, where $W' \cap \{\$, f\} = \varnothing$, each $(a, b, x, c) \in R$ satisfies $a \in V - T$ and

- either $b \in W'$, $x \in (V - T)^*$, $c \in W' \cup \{\$, f\}$
- or $b = \$$, $x \in T$ and $c \in \{\$, f\}$

The symbol $\$ \notin W'$ denotes the state where only terminals are generated and the symbol $f \notin W'$ is the new and only final state.

Lemma 5.3.2. For every queue grammar Q, there exists a linear translation grammar G such that

$$L(Q) = \{x : (x, y) \in T(G), y \in D\}$$

Proof. Without any loss of generality, assume that the queue grammar Q satisfies the conditions described in Lemma 5.3.1.

$$Q = (V, T, W' \cup \{\$, f\}, \{f\}, R, g)$$

Then we construct a linear translation grammar G in the following way:

$$G = (W' \cup \{\$, f, S\}, T, \{0, 1\}, P, S)$$

where $S \cup W' = \varnothing$. Set $n = |V|$.
Introduce the bijective homomorphism α from V to $\{0, 1\}^n \cap 0^+10^*1$. Expand its domain to V^* so that $\alpha(ab) = \alpha(a)\alpha(b)$, for any $a \in V, b \in V^*$ and $\alpha(\varepsilon) = \varepsilon$. Let ω be the bijective homomorphism defined as $\omega(x) = reversal(\alpha(reversal(x)))$, where $x \in V^*$.
P is constructed as follows:

1. For $g = kl$, $k \in V$, $l \in W'$, add $(S, S) \Rightarrow (l, \alpha(k)l)$ to P.
2. For each $(a, b, x, c) \in R$, where $a \in V - T$, $b \in W' \cup \{\$\}$, $x \in (V - T)^* \cup T$ and $c \in W \cup \{f\}$, add $(b, b) \Rightarrow (c, \alpha(x)c\omega(a))$ to P.
3. For each $t \in T$, add $(f, f) \Rightarrow (tf, f\omega(t))$ to P.
4. Finally, add $(f, f) \Rightarrow (\varepsilon, \varepsilon)$ to P.

Basic idea. The constructed translation grammar G simulates the queue grammar Q that satisfies the properties described in Lemma 5.3.1. The production from 1, applied only once, initialises the derivation. The production 4 terminates the derivation. The productions from 2 simulate the rules applied by Q. Finally, productions from 3 generate the simulated terminals to the input string in the order generated by Q.

Claim A. G can generate every $(x, y) \in T(G)$ where $y \in D$ in this way:

$$\begin{array}{lllr} & (S, & S &) \\ \Rightarrow & (l, & \alpha(k)l &) \\ \Rightarrow^* & (\$, & \alpha(a_0..a_k..a_{k+m})\$ & \omega(a_k..a_0)) \\ \Rightarrow^* & (f, & \alpha(a_0..a_{k+m}x_1..x_m)f & \omega(a_{k+m}..a_0)) \\ \Rightarrow^* & (xf, & \alpha(a_0..a_{k+m}x_1..x_m)f & \omega(x_m..x_1a_{k+m}..a_0)) \\ \Rightarrow & (x, & \alpha(a_0..a_{k+m}x_1..x_m) & \omega(x_m..x_1a_{k+m}..a_0)) \end{array}$$

Where $k, m \geq 1$, $a_i \in V - T$ for $i = 0, \ldots, k+m$; $g = kl : k \in V - T, l \in W'$; $a_0 = k$; $x = x_1..x_m$, $x_i \in T^*$ for $i = 1, \ldots, m$;

$$y = \alpha(a_0..a_{k+m}x_1..x_m)\omega(x_m..x_1a_{k+m}..a_0)$$

$$y = vw, w = reversal(v)$$

Proof of claim A. Examine the construction of G. Observe that every successful derivation begins with application of production 1, followed by applying productions from 2, followed by applying productions from 3. Every successful derivation ends with a single application of production 4.

Observe that during application of rules in 2 and 3, the output side is as follows: $w_i \in V$ for $i = 0, \ldots, k+m$; $k, m \geq 1$; $N \in W' \cup \{\$, f\}$

$$\alpha(w_0..w_k..w_{k+m})N\omega(w_l..w_0)$$

To satisfy $y \in D$, only productions that put $\omega(w_{k+1})$ to the right of the nonterminal are applicable.

Claim B. Q generates every $h \in L(Q)$ in this way: $g = a_0q_0$

$$\begin{array}{ll} a_0q_0 & \\ \Rightarrow a_1y_1q_1 & (a_0, q_0, z_0, q_1) \\ \Rightarrow a_2y_2q_2 & (a_1, q_1, z_1, q_2) \\ \ldots & \\ \Rightarrow a_{k+1}y_{k+1}q_{k+1} & (a_k, q_k, z_k, q_{k+1}) \\ \Rightarrow a_{k+2}y_{k+2}x_1\$ & (a_{k+1}, q_{k+1}, x_1, \$) \\ \ldots & \\ \Rightarrow a_{k+m}x_1 \ldots x_{m-1}\$ & (a_{k+m-1}, \$, x_{m-1}, \$) \\ \Rightarrow x_1 \ldots x_m f & (a_{k+m}, \$, x_m, f) \end{array}$$

where $k, m \geq 1$; $a_i \in V - T$ for $i = 0, \ldots, k+m$; $y_i \in (V - T)^*$ for $i = 1, \ldots, k+m-1$; $x_j \in T^*$ for $j = 1, \ldots, m$; $z_0 = a_1y_1$; $y_iz_i = a_{i+1}y_{i+1}$ for $i = 1, \ldots, k$; $g = a_0q_0$; $q_i \in W'$ for $i = 0, \ldots, k$;

Proof of claim B. Recall that Q satisfies the properties given in Lemma 5.3.1. These properties imply that Claim B holds.

Claim C. Let G generate $(h, y) \in T(G)$, $y \in D$ in the way described in Claim A; then, $h \in L(Q)$.

Proof of claim C. Let $(h, y) \in T(G)$, $y \in D$. Consider the generation of h as described in Claim A. Examine the construction of P to see that at this point R contains (a_0, q_0, z_0, q_1), $\ldots$, (a_k, q_k, z_k, q_{k+1}), $(a_{k+1}, q_{k+1}, x_1, \$)$, $\ldots$, $(a_{k+m}, \$, x_m, f)$, where $z_0, \ldots, z_k \in (V - T)^*$, and $x_1, \ldots, x_m \in T^*$. Then, Q makes the generation of h in the way described in Claim B.

Claim D. Let Q generate $h \in L(Q)$ in the way described in Claim B; then, $(h, y) \in T(G)$, $y \in D$.

Proof of claim D. Let $h \in L(Q)$. Consider the generation of h as described in Claim B. Examine the construction of P to see that at this point P contains

$$
\begin{aligned}
(S, S) \Rightarrow & \qquad (q_0, \alpha(a_0)q_0),\\
(q_0, q_0) \Rightarrow & \qquad (q_1, \alpha(z_0)q_1\omega(a_0)),\\
\ldots, & \\
(q_k, q_k) \Rightarrow & \qquad (q_{k+1}, \alpha(z_k)q_{k+1}\omega(a_k)),\\
(q_{k+1}, q_{k+1}) \Rightarrow & \qquad (\$, \alpha(x_1)\$\omega(a_k + 1)),\\
\ldots, & \\
(\$, \$) \Rightarrow & \qquad (f, \alpha(x_m)f\omega(a_{k+m})),\\
(f, f) \Rightarrow & \qquad (x_1 f, f\omega(x_1)),\\
\ldots, & \\
(f, f) \Rightarrow & \qquad (x_m f, f\omega(x_m))
\end{aligned}
$$

where $z_0, \ldots, z_k \in (V - T)^*$, and $x_1, \ldots, x_m \in T^*$. Then, G makes the generation of (h, y) in the way described in Claim A.

Claims A through D imply that $L(Q) = \{x : (x, y) \in T(G), y \in D\}$, so this lemma holds. □

Lemma 5.3.2 implies that any queue grammar can be simulated by a linear translation grammar. Since linear queue grammars define the family of recursively enumerable languages **RE**, linear translation grammars can be used to define any recursively enumerable language. Because of this descriptional complexity, any grammar or rewriting system defining any subset of recursively enumerable languages can be simulated by a linear queue grammar.

Chapter 6

Translation Framework

For this thesis, we implemented a grammar-based translation framework `ctf` (*C*tf ain't a *T*ranslation *F*ramework). In this chapter, we will discuss its features, its translation specification language and we will compare the approach to translation to existing tools like GNU Bison.

6.1 Translation Framework Features

The framework is designed as a runtime library for modern C++. Because non-general top-down predictive translation covers a relatively small subset of translation grammars, `ctf` implements bottom-up LR(1) translation as its primary translation algorithm. We let the user choose between canonical LR, LSCELR and LALR translation, and we default to LSCELR. The input languages for both canonical LR and LSCELR translation are relatively unrestricted and don't present many obstacles for grammar designers. Resolving conflicts with precedence and associativity annotations adds even more convenience for the development of grammars and even reduces the number of steps the translator has to make when accepting grammars that specify expressions — only a single nonterminal represents the expression instead of one for each precedence level.

In `ctf`, translation is defined by an attribute translation grammar with precedence and associativity for conflict resolution. This leads to a slightly different approach to syntax-directed translation compared to existing context-free grammar-based tools. We discuss these differences in Section 6.3. The user provides three components to define a full translation: a lexical analyzer, an attribute translation grammar and an output generator.

The lexical analyzer divides the input into lexemes and translates them into tokens, some of which have attributes (see page 11 in [9]). The framework does not implement a lexical analyzer generator. It does, however, provide a base class for user implemented lexical analyzers that reads text input and keeps track of input position automatically. Errors and warnings are also handled by this base class. It is recommended that users implement a recursive descent lexical analyzer or a state machine lexical analyzer.

The attribute translation grammar defines the core of the translation. We choose attribute translation grammars for their ability to transfer attributes from the input string of tokens to the output string of tokens. Users can use the text format described in Section 6.2 to specify these grammars. The translation algorithm uses this grammar to translate the string of input tokens provided by the lexical analyzer to a string of output tokens. The output tokens contain output terminals and their attributes.

The output generator receives the output of the translation defined by the translation grammar and produces final output. For straightforward translation, such as translating between two markup languages, this could be as simple as printing the text representation of output tokens. For more complicated applications such as compilers, the output generator will likely contain a semantic analyzer, a code generator etc. Because output is only available when the input has been parsed successfully, output has a well defined form defined by the output grammar. The output generator needs to correctly parse the output string to form its output. If the associated translation grammar was written well, parsing the output will be relatively straightforward: users are free to put separation markers between sections and other arbitrary symbols to their output languages to make parsing the output easier. For applications where a syntax tree is necessary, it is possible to define the output to be in postfix notation that is then converted to the syntax tree by the output generator as a preparation step. The framework provides a base class for output generators with convenience methods for text output, warnings and errors handling.

Since `ctf` is not yet as mature as existing tools and frameworks, there exist large grammars for which the runtime cost of creating the parser tables is very high (almost 30 seconds for the GAWK grammar when using LSCELR). `ctf` provides the option of saving and loading parsing tables to and from text representation. This operation is very fast and bypasses the issues with potentially slow parser construction. The user can add custom parsing error messages by constructing the translation objects with their custom error message function as an optional parameter. Error recovery is currently not implemented, but is ready for use if any users implement it in a subclass of the translation control classes.

6.2 Grammar Specification

Specifying a translation grammar in the form of a programming language source file is neither easy, nor maintainable. We introduce a language for specifying attribute translation grammars for our framework. This language was inspired by the BISON grammar specification language, as well as the YAML markup language (see [6] and [19]).

$$ar = \{(1, \{1\})\} \cup \{(x, \varnothing) : x \neq 1\}$$

$$P = \begin{cases} 0 : [(S', S') \to (A, A)\$, \varnothing], \\ 1 : [(A, A) \to (a, a), ar], \\ 2 : [(A, A) \to (aAB, cAdBe), ar], \\ 3 : [(B, B) \to (b, b), ar] \end{cases}$$

Figure 6.1: An attribute translation grammar for $\{(a^{n+1}b^n\$, c^n a(dbe)^n\$) : n \geq 0\}$

```
grammar grammar_name

A:
    'a'
    'a' A B | 'c' A 'd' B 'e'
        1

B:
    'b'
```

Figure 6.2: Representation of Figure 6.1

Each grammar specification file (`*.ctfg`) contains the name of the translation grammar, its optional precedence and associativity specifications, and its productions. The name of the translation grammar is in `snake_case` and determines the name of the output source files and the namespace they'll be put in. The starting nonterminal is always the left-hand side nonterminal of the first production. Nonterminals are always `CamelCase`, and we can put the symbol `'` in the name, except for the first position. Terminals are denoted

as `'terminal'`, where we can put almost any symbols between the `'` characters. Whitespace symbols such as newlines or tabs are forbidden in terminal names, with the exception of spaces. Indentation is always done using tab characters, and comments start with the `#` character and end at the end of line. The full grammar for translating this format is specified in this format in Appendix B.

6.2.1 Precedence

```
precedence: # optional precedence specifications
    none 'unary -' # highest precedence
    right '^'
    left '*' '/'
    left '+' '-' # lowest precedence
```

Figure 6.3: Example of precedence specification

Precedence can be specified after the grammar name and before grammar productions as a list of descending precedence levels, specifying precedence and associativity for groups of terminals. Each precedence layer has an associativity specification as discussed in Section 3.4.6. Figure 6.3 shows a specification of precedence levels.

6.2.2 Productions

```
Expression:
    'integer' # input and output are the same and implicitly connected
    Expression '-' Expression |
        Expression Expression '-'
        3 # we connect the '-' terminals
    '-' Expression | Expression 'unary -'
        precedence 'unary -' # different precedence for this production
        - # we explicitly don't connect anything to the input '-'
    'float cast' 'integer' | 'integer' 'float cast'
        2
        1
```

Figure 6.4: Example of production specification

Figure 6.4 shows an example of a group of productions. We group productions with the same left-hand side nonterminals in the same group. A group of productions starts with the left-hand side nonterminal of the production, followed by a `:` character. Then follow the productions, indented by one tab and separated by newlines. There exist two forms of production specification: if both input and output right-hand sides of the production are the same, we only specify the input right-hand side, and the output is set to the same string of terminals and nonterminals. All attribute relays in such rules are automatically set so that the corresponding terminals are connected. The second form specifies both input and output, separated by the `|` symbol. The output part of a rule can be put on a new line,

indented by one extra tab symbol. We explicitly denote the empty string in either input or output as `-`.

Rules have the implicit precedence of their last input terminal, or the end of input terminal if there aren't any. We can explicitly specify a different terminal symbol as the production precedence. This is useful when two different symbols share the same terminal on input, but have different semantic meaning (for example negation and subtraction).

We can specify attribute relays for terminals. Like precedence specification for productions, they follow the input and output on next lines, indented by an extra tab. We can specify attribute relays for all (or a prefix of) terminals in the input. Each attribute relay is a list of integers separated by commas. Each number represents the corresponding symbol in the output and it must be a terminal. The attribute from that input terminal will be distributed to all target nonterminals. We can explicitly give empty relay sets to terminals by using the `-` symbol. If multiple output terminals are targets of multiple relays, the behavior is undefined. If both precedence specification and attribute relays need to be specified, precedence specification always goes first.

6.3 Syntax-directed Translation

We redefine the concept of syntax-directed translation as understood from the perspective of syntax-directed parsing (see Figure 1.8 in [9]) for the context of translation grammars. In this section, we will discuss the approach to translation in this framework and compare it to GNU Bison as a representative of existing parsing tools.

GNU Bison associates applications of productions with user-defined function calls. This lets us create the syntax tree and, in general, perform almost any action upon reducing any rule. This can lead to semantics checks during the parsing process. In `ctf`, we simply define the relationship between the input and output tokens and their attributes. Semantic meaning of language constructs is separated from translation itself completely and only has effect when generating output after the translation has been successful. This feature is admittedly an issue for some languages that inherently rely on their semantics for parsing (e.g. C, C++ or Java), but for other applications, this strict separation of syntax from semantics could lead to better maintainability of the translator. The output generator in translation grammar-defined translation is a component that is completely separated from the translation algorithm: the same output generator can be used for both top-down and bottom-up translation algorithms.

In traditional parsers, the parse tree is always constructed according to the specification of the context-free grammar. On the other hand, the actions on each reduce or the construction of the abstract syntax tree are not formally specified. When using translation grammars as the formal base, we generate output according to that formal specification. A different challenge with parsing that formally defined output arises: parsing the output string is left up to the output generator, becoming the informally defined part of the translator. For most applications, postfix notation can be used for expressions and other similar concepts, so that their syntax trees can be trivially constructed from the output string. Generating the postfix notation does not need to correspond to the syntax tree with translation grammars, so we gain some flexibility in defining translations this way. Most other concepts can be clearly communicated in parts of the output separated by special output terminals. The difficulty of parsing the output is determined by the formal specification of the translation itself. A potential advantage of this can be that the grammar can be heavily restructured, but

when the output language stays the same, we can reuse existing output generators without any changes.

The set of input grammars we can parse in `ctf` with LSCELR is the same as the set of context-free grammars we can parse in GNU Bison when using IELR. There are two main differences between the two tools. The first difference is the inherent difference in approach to translation. There exist applications where it is desirable to be able to run custom code on reducing a production, and there exist applications where processing a flat translation output is more convenient (such as simple translators). For most applications, both approaches are applicable and are simply a matter of preference. The second difference is more substantial: GNU Bison is simply a parser generator, whereas `ctf` is a runtime C++ framework. While it is certainly possible and convenient to construct programs that do what a Bison-generated parser would do, we also have the option of constructing translations during runtime.

The use of modern C++ could also prove to be an advantage in the context of practical compilers. Many compilers use the LLVM framework (see [7]) for target code generation. While it is possible to emit text representation of the LLVM intermediate code, the native C++ interface is bound to be more convenient for compiler developers.

6.4 Table Compression

Fully enumerated LR parsing tables are mostly filled with error or empty items. Practical implementations (such as GNU Bison) don't typically store LR parsing tables as 2D arrays of actions. In this section, we propose a method of LR table compression that keeps all enumerated actions, while reducing the size of the tables for most practical parsers.

We store the action and goto tables as an array of sorted arrays. Each table is implemented with two arrays: the first array a stores pairs of symbols and their actions or gotos where only nonempty items are stored (error items are considered empty in the action table). The second array i stores the beginning and end indices for each state. For example, items for state n are stored between $a[i[n]]$ and $a[i[n+1]]$. Items are stored sorted by their key (terminals for the action table and nonterminals for the goto table).

Item lookup is implemented as binary search in the subrange of a determined by the state and has the time complexity $O(log(M))$, where $M = |T|$ for action tables and $M = |N|$ for goto tables. In practical grammars, very few states have more than $x \geq \frac{|T|}{2}$ actions or $x \geq \frac{|N|}{2}$ gotos. This lets use less total memory, while preserving all nonempty table items. Total time for lookup of an item for state n, where there are $j \cdot M$ non-empty items is $log_2(j \cdot M)$ and assuming the storage needed to save an item is the same as storage needed to store a terminal or a nonterminal, we save $(1 - 2j) \cdot M$ units of memory. With a conservative average of $j = 0.3$, this reduces memory usage to 0.6 times the original space requirements. Further space conservation could be achieved by having a flat representation of actions and gotos in states where more than half of symbols yield a valid action or goto. This was not deemed necessary, as states where that is true are extraordinarily rare for tested practical grammars.

Compressed LR tables are implemented in `ctf_lr_table.hpp`. Each table contains both goto and action tables in compressed form as described in this section. LR tables have the type of automaton from which they're constructed as a template parameter. Class aliases for LALR, LSCELR and Canonical LR parsing tables are defined in that source file, with variants with and without conflict resolution. Table classes can be saved to a file

and loaded using a simple serialization format. Table deserialization is not checked for correctness, and manually changing the saved parser tables will lead to undefined behavior. This allows for faster deserialization.

6.5 Implementation

This section contains a brief description of the implementation of `ctf` and its associated tools. We also discuss some features of the framework that are not a core part of its design. A practical use of the framework is demonstrated in the description of the `grammarc` syntax-directed translator.

We were able to use the implementation of the translation framework from [17] as a basis for the implementation for this thesis. The framework has since been heavily modified and the supported translation algorithms have been changed completely. The final implementation is in C++17 and will likely be compatible with newer standards of C++ in the future.

The framework is designed to be easy to add as a dependency. For that reason, the whole implementation can be included as a single header file. This removes any potential issues with different compiler or ABI versions and can be used with any compiler that supports C++17. This design decision has a negative impact on compile times: the whole framework is compiled for every translation unit it's included in. This is a deliberate tradeoff: translation is an unlikely operation to be needed in large parts of any large system. When the framework is included exclusively in files that handle translation, the ease of use outweighs the build times issue.

6.5.1 Application Program Interface

The complete program documentation is available in the Doxygen-generated documentation. The project has a Makefile target for generating the documentation, which can be run as `make doc`. We represent terminals and nonterminals as instances of class `Symbol`. Symbols are often used to index tables, so they are represented by unsigned integers, where the two most significant bits denote whether they are terminals, nonterminals or the end of input symbol. Tokens are instances of class `Token`, and they contain the symbol they represent, their program location and an optional attribute. Attributes are stored as `Attribute` and can store any arbitrary type. The standard library type `std::any` is used as the underlying storage for attributes; this makes sure that access to stored attributes is type-safe.

Translations are instances of class `Translation` and can be run with different sets of input and output streams. The lexical analyzer, translation algorithm and output generator types are template type parameters of `Translation`, and any instance takes ownership of all three of these objects. The constructor also makes a copy of the translation grammar that defines the translation. The function `load` makes it convenient to supply saved parser tables instead of constructing the translation from the translation grammar itself. When an instance of `Translation` has been constructed, it is ready to run translation. We can also optionally supply a function for printing symbols as their text representation; this is useful both during debugging, when we can see which symbols cause issues, and during production: the default parsing error messages use this function to communicate which symbol was unexpected and which symbols would be acceptable. The method `run` takes three streams as parameters: the input, output and error streams. We can also optionally provide the name of the input stream, which is added to symbol locations.

The project contains classes `LexixalAnalyzer` and `OutputGenerator`. These classes contain the default implementations of their respective functionalities. They provide a convenient interface for implementing the user-defined lexical analyzers and output generators. Both classes provide a mechanism for warnings, errors and unrecoverable errors that print additional info with the user-supplied messages. The lexical analyzer class also provides an input buffer that stores the input and automatically tracks the symbol locations, as well as a method for constructing tokens with this information automatically added.

6.5.2 Tools

There exist two `ctf` tools that make using it more convenient: `grammarc` and `parsergen`. The former lets us specify attribute translation grammars in the format specified in Section 6.2 and translate that representation to a pair of C++ source files. The latter is a convenience tool for large parsers; it takes source files from `grammarc` and converts them to text representation of parsing tables for a chosen translation algorithm.

Grammar Translation Tool

Section 6.2 specifies a translation grammar text representation format. The framework doesn't currently support loading translation grammars from files; `grammarc` is a tool that converts this input into the appropriate C++ source files. The framework is designed around this tool; it is possible to use it without this tool, but it is far less convenient. We generate the grammar itself, as well as several convenience functions. The name of the grammar dictates the names of the generated source files and the namespace all symbols will be put in: grammar `example` will generate two source files `example.cpp` and `example.h`.

The generated header file contains custom string literal functions `""_t` and `""_nt` for both terminals and nonterminals defined in the translation grammar. This can be useful for lexical analyzers and output generators, as they can use these literals to construct symbols. We generate a function `to_string` for printing symbols that associates symbols with their string representation. Finally, the header file contains a declaration of the grammar itself. The generated source file contains the translation grammar itself. The construction uses the custom literals defined in the header to make the program representation slightly more legible.

Because this tool performs translation, it is implemented with `ctf`. The translation grammar it uses is in Appendix B and translates the representation to an easy to parse sequence of output tokens.

The lexical analyzer is implemented as a recursive descent lexical analyzer. Because the input language uses indentation, we must also keep track of the indentation level in the lexical analyzer. Special tokens `'INDENT'` and `'DEDENT'` represent the start and end of a specific indentation level. When a new line starts with more tab symbols than the previous one, we cache and subsequently emit that many `'INDENT'` tokens after the `'NEWLINE'` token, and we do the same with `'DEDENT'` tokens when there are fewer tabs on the beginning of a line. The rest of the lexical analyzer is implemented in standard fashion. Attributes of tokens are either missing, strings or unsigned integers.

The output generator performs two passes over the output: In the first pass, we parse the optional precedence declaration and collect all nonterminals and input terminals and output terminals (terminals used purely for precedence are considered output terminals). We then map the symbols to their integer representations. With this information, we are able to generate the header file. In the second pass, we generate the translation grammar

itself. We pass over the individual rules and generate the contents of the source file. The grammar outputs special tokens that denote ends of major sections of the output, such as ends of individual rules, ends of groups of rules etc. These separators greatly simplify the generation of output. The tool checks for correctness in the individual rules: nonterminals in input and output must match.

The first iteration of this tool was implemented with the program representation of the translation grammar. Now, the implementation is self-hosting and its translation grammar is generated from a source file in its own format. The implementations of both the lexical analyzer and the output generator, as well as creating and running the translation are located in `tools/grammar/main.cpp` and they provide an example of usage of `ctf`.

Parser File Generator

LSCELR parsing tables are slow to create for grammars that yield hundreds of states. To support loading parser tables created from text representation of translation grammars, `parsergen` can create either of the chosen translation algorithms and create the saved parsing table with the help of source files created by `grammarc`. These generated parsers can be used as text input in raw string literals when creating translations, or they can be loaded from files. The parser tables are created by generating a short `ctf` program using the two source files from `grammarc` in a temporary directory and saving the parser tables to a file.

Chapter 7

Conclusion

In this thesis, we provided a short overview of the existing parsing algorithms for languages generated by context-free grammars. The LR family of parsing algorithms covers the largest set of these languages; because of this, we have chosen LR parsing as the basis for our translation algorithms. The traditional approach to translation, where we create an abstract syntax tree from the productions used in parsing and transfer input tokens' attributes, is used by most existing tools today. We provided a short summary of the techniques that are used in these kinds of parsers.

We designed an original minimal LR(1) parsing algorithm LSCELR based on the IELR parsing algorithm. We designed a new representation of the LR automaton, the LS automaton. We embedded additional information about the relationships between items in the LS automaton, which allow us to perform analysis of the automaton's properties. We provided a complete set of procedures and algorithms necessary for creating and working with LS automatons. The LS automaton is the model for both LSLALR and LSCELR parsing. The structure of LS automatons is well-suited for splitting states, which we utilize in the creation of the LSCELR parser. We detect all conflicts in the LSLALR parser, mark all items that contribute to those conflicts and finally split states that cause different conflicts. LSCELR parsers are able to parse the same subset of grammars as canonical LR parsers, but with an order of magnitude fewer states for most practical grammars.

We provided a comprehensive definition of translation grammars. As a new addition, we also introduced attribute translation grammars for practical use in translation. Practical translation often defines attributes for some input terminals and the ability to additionally define the relationships between input and output symbols' attributes is important for practical translation tools. Attribute translation grammars allow us to formally define these relationships in the scope of individual productions. We designed a modification of selected parsing algorithms to create translation algorithms: top-down predictive translation and bottom-up LR translation. We presented the use of two-stack automata for simulating translation grammars. We presented a formal proof of the generational power of translation grammars first introduced in [17] that suggests that translation grammars can generate recursively enumerable languages.

We designed a translation framework based on attribute translation grammars and the original parsing and translation algorithms discussed in this thesis. The C++17 translation framework `ctf` is a runtime framework for translation that implements LSCELR, LSLALR and canonical LR translation. We designed a language for specifying attribute translation grammars for this framework. This language lets us specify the productions and attribute relays of attribute translation grammars, as well as the precedence and associativity of their

terminals and productions. We implemented a compiler `grammarc` from this language to the source representation for `ctf`. We used the framework itself to perform the translation, making `grammarc` self hosting. We check semantic correctness of the grammar specification, and emit appropriate error messages for specific semantic errors. We implemented a tool for creating saved parsing tables from the output of `grammarc` that lets us efficiently initialize translations for even large grammars, where the speed of the current implementation could become an issue.

As far as we know, this definition of attribute translation grammars hasn't been used for defining translation in existing translation tools or frameworks. The success of this approach to translation, as opposed to using context-free grammars and constructing abstract syntax trees, depends largely on the quality of translation output languages. A comprehensive set of recommendations for constructing easily parseable output languages and formal devices for parsing these output languages (such as predicting possible next symbols based on output context) could be a large contribution to the popularity of translation grammars in practical tools. This could be a subject of future work on this project, as the construction of output languages is currently left up to the creators of translation grammars.

Further improvements and extensions to the framework can be made. Here, we will detail other tasks for future work on this project. The implemented parsing techniques only work with a subset of translation grammars. Although most practical concepts can be easily defined for LSCELR translation by attribute translation grammars with conflict resolution, there may still exist use cases where a general translation algorithm is needed. Implementing general or backtracking versions of either LL or LR parsing algorithms would allow us to translate all languages generated by translation grammars. LSCELR parsing would be a suitable basis for a general or backtracking parsing algorithm because of its minimization of both conflicts and invalid actions.

We currently don't implement error recovery in `ctf`. Future work should provide a suitable implementation of some existing LR error recovery algorithms for our LR translation algorithms. A mechanism for selecting the error recovery algorithm should be a part of the framework after implementing error recovery. Another area of improvement is in the framework's error messages. The current implementation enumerates the expected terminal symbols on error. Although some practical parsers use this approach, it would be desirable to provide a convenient way to generate custom error message functions for our framework. We suggest implementing a tool similar to `merr` that generates the error function from short samples of source code with the help of the parser and the lexical analyzer. Since we already support generating parsing tables from the text representation of attribute translation grammars, generating the error message functions could also be handled by a similar tool.

We prioritized correctness over efficiency when developing the LSCELR algorithm. As a result of this prioritization, there unfortunately exist inefficiencies in both its design and implementation. Future work could improve the utilization of LS automatons for their benefits and getting rid of the inefficiencies in the generation of LSCELR parsers. One possible approach would be to adopt some of the approaches and analysis algorithms from IELR parsing and using LS automata purely to preserve the lookahead propagation information so that no recomputation of these dependencies is required in phase 1 of the IELR algorithm. Alternatively, modifications to the current implementation and efficient caching of the full lookahead sets and potential contributions could yield better results even with the current design of the LSCELR algorithm.

The translation framework as-is should serve as a practical and usable tool for creating parsers and translators. We acknowledge that the existing tools and parser generators are widely used and powerful, although they generally represent a single approach to translation. We provide a fresh take on translation with the use of attribute translation grammars, and the approach we introduce here may prove more fitting for many translation purposes over the use of context-free grammars. Our framework should hopefully make the development of both simple and complex translation tools easier and ultimately prove useful in the large field of competitors.

Bibliography

[1] Aho, A. V.; Lam, M. S.; Ullman, J. D.; et al.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson. 2006. ISBN 0-321-48681-1.

[2] ANSI C Yacc grammar. 2012. [Online; accessed 2019-04-14].
Retrieved from: http://www.quut.com/c/ANSI-C-grammar-y-2011.html

[3] Chomsky, N.: On certain formal properties of grammars. *Information and Control*. vol. 2, no. 2. 1959: pp. 137 – 167. ISSN 0019-9958.
Retrieved from:
http://www.sciencedirect.com/science/article/pii/S0019995859903626

[4] Denny, J. E.; Malloy, B. A.: IELR(1): Practical LR(1) Parser Tables for Non-LR(1) Grammars with Conflict Resolution. *Science of Computer Programming*. vol. 75. 2010: pp. 943 – 979.

[5] The Bison Parser Algorithm. [Online; accessed 2019-04-27].
Retrieved from: https://www.gnu.org/software/bison/manual/html_node/How-Precedence.html#How-Precedence

[6] Levine, J.: *flex & bison: Text Processing Tools*. O'Reilly Media. 2009. ISBN 0596155972.

[7] The LLVM Compiler Infrastructure. [Online; accessed 2019-04-21].
Retrieved from: https://llvm.org/

[8] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer. 2005. ISBN 81-8128-333-3.

[9] Meduna, A.: *Elements of Compiler Design*. Taylor & Francis. 2008.

[10] Meduna, A.: *Formal Languages and Computation: Models and Their Applications*. Auerbach Publications. 2014. ISBN 978-1-4665-1345-7.

[11] Meduna, A.: IFJ - Kapitola VI. Modely pro bezkontextové jazyky. Fakulta informačních technologií. Vysoké učení v Brně. 2015.

[12] Meduna, A.: IFJ - Kapitola VII. Syntaktická analýza shora dolů. Fakulta informačních technologií. Vysoké učení v Brně. 2015.

[13] Meduna, A.; Techet, J.: *Scattered context grammars and their applications*. WIT. 2010. ISBN 978-1-84564-426-0.

[14] Meduna, A.; Zemek, P.: *Regulated Grammars and Automata*. Springer. 2014. ISBN 978-1-4939-0368-9.

[15] opengroup.org: awk - pattern scanning and processing language. [Online; accessed 2019-04-14].
Retrieved from: https://pubs.opengroup.org/onlinepubs/7908799/xcu/awk.html

[16] Pennello, T. J.; DeRemer, F.: Efficient Computation of LALR(1) Look-ahead Sets. *SIGPLAN Not.* vol. 39, no. 4. April 2004: pp. 14–27. ISSN 0362-1340. doi:10.1145/989393.989396.
Retrieved from: http://doi.acm.org/10.1145/989393.989396

[17] Vít, R.: *Translation Grammars: Properties and Applications*. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. 2017.
Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=18096

[18] Vít, R.: *Grammar-Based Translation Framework*. Term project. Brno University of Technology, Faculty of Information Technology. 2018.

[19] The Official YAML Web Site. [Online; accessed 2019-04-20].
Retrieved from: https://yaml.org/

Appendix A

Contents of the Included CD

- **ctf** - this folder contains the implementation of the translation framework `ctf`
- **ctf/tools/grammar** - this folder contains the implementation of the translator for attribute translation grammars using `ctf`
- **ctf/README.md** - This file describes the dependencies of `ctf` and the steps needed to test the framework and to compile and run `grammarc`. It also contains a short tutorial for creating translation tools with `ctf`.
- **grammars** - this folder contains the translation grammars used in Table4.4. Only the input languages are specified here without any conflict resolution.

Appendix B

Translation Grammar for grammarc

```
# CTF Grammar Tool Grammar
# translation from this grammar format to C++
# author: Radek Vit
grammar ctfgc

# Keywords
## grammar, precedence, none, left, right
# Tokens in Python Regex
## 'grammar name': [a-z](_[a-z]+)*_?
## 'nonterminal': [A-Z][a-zA-Z']*
## 'terminal': '([^\s\"\\]| |\\[bfnrt\"'\\])+?'

# disambiguate empty lines before optional precedence declaration
precedence:
       none 'NEWLINE'

# rules and precedence levels are always indented with the 'tab' character

GrammarC:
       'NEWLINE' GrammarC | GrammarC
       'grammar' 'grammar name' 'NEWLINE' Precedence Rules |
              'grammar' Precedence Rules
              -
              1

# 'NEWLINE' has higher precedence than EOF, so the second rule
# will be picked before the first rule if both are viable.
Precedence:
       -
       'NEWLINE' Precedence | Precedence
       'precedence' ':' 'NEWLINE' 'INDENT' PrecedenceLevels 'DEDENT' |
              'precedence' PrecedenceLevels 'precedence end'
              1
```

```
PrecedenceLevels:
       -
       'NEWLINE' | -
       Associativity TokenList 'NEWLINE' PrecedenceLevels |
              Associativity TokenList 'level end' PrecedenceLevels

Associativity:
       'none'
       'left'
       'right'

TokenList:
       'terminal'
       'terminal' TokenList

Rules:
       'NEWLINE' Rules | Rules
       Rule
       Rule Rules'

Rules':
       'NEWLINE' | -
       'NEWLINE' Rules' | Rules'
       Rule
       Rule Rules'

Rule:
       'nonterminal' ':' 'NEWLINE' 'INDENT' RuleClauses 'DEDENT' |
              'nonterminal' RuleClauses 'rule block end'
              1
              -
              -
              -
              3

RuleClauses:
       -
       # empty line or comment
       'NEWLINE' RuleClauses | RuleClauses
       RuleClause RuleClauses | RuleClause 'rule end' RuleClauses

RuleClause:
       String 'NEWLINE' | String
       String 'NEWLINE' 'INDENT' PrecedenceAttribute 'DEDENT' |
              String PrecedenceAttribute
       String '|' OutputString
```

```
String:
       '-' | 'string end'
              1
       'terminal' | 'terminal' 'string end'
              1, 2
       'nonterminal' | 'nonterminal' 'string end'
              1, 2
       'nonterminal' String
       'terminal' String

OutputString:
       'NEWLINE' 'INDENT' String 'NEWLINE' 'DEDENT' | String
       'NEWLINE' 'INDENT' String 'NEWLINE' Attributes 'DEDENT' | String Attributes
       String 'NEWLINE' | String
       String 'NEWLINE' 'INDENT' Attributes 'DEDENT' | String Attributes

Attributes:
       RulePrecedence | 'attributes' RulePrecedence 'attribute list end'
       AttributeList | 'attributes' AttributeList 'attribute list end'
       RulePrecedence AttributeList |
              'attributes' RulePrecedence AttributeList 'attribute list end'

PrecedenceAttribute:
       RulePrecedence | 'attributes' RulePrecedence 'attribute list end'

RulePrecedence:
       'precedence' 'terminal' 'NEWLINE' | 'precedence' 'terminal'
              1
              2

AttributeList:
       Attribute
       Attribute AttributeList

Attribute:
       '-' 'NEWLINE' | 'attribute end'
              -
              1
       IntList 'NEWLINE' | IntList 'attribute end'
              2

IntList:
       'integer'
       'integer' ',' | 'integer'
              1
       'integer' ',' IntList | 'integer' IntList
              2
```