BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

A GRAMMATICAL FORMALIZATION OF TRANSLATION AND ITS IMPLEMENTATION

GRAMATICKÁ FORMALIZACE PŘEKLADU A JEJÍ IMPLEMENTACE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MATÚŠ SABOL**
AUTOR PRÁCE

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

BRNO 2019

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

22466

Student:	**Sabol Matúš**
Program:	Informační technologie
Název:	**Gramatická formalizace překladu a její implementace**
	A Grammatical Formalization of Translation and Its Implementation
Kategorie:	Teoretická informatika

Zadání:

1. Dle instrukcí vedoucího se seznamte s překladovými gramatikami.
2. Dle instrukcí vedoucího studujte vlastnosti překladových gramatik.
3. Formalizujte syntakticky řízený překlad překladovými gramatikami.
4. Implementujte formalizace navržené v bodě 3. Testujte implementaci na řadě příkladů.
5. Zhodnoťte dosažené výsledky. Diskutujte další vývoj projektu.

Literatura:

- Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1-3, Springer, 1997, ISBN 3-540-60649-1 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

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce:	**Meduna Alexander, prof. RNDr., CSc.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1. července 2019
Datum odevzdání:	31. července 2019
Datum schválení:	9. července 2019

Abstract

This thesis serves as an introduction to the topic of formal translation. The reader is introduced to important theory and this theory is subsequently used to construct a particular translator. The first part defines the base formal languages theory. These findings are followed upon by the second part that firstly concerns itself with bottom-up parsing in more detail. A canonical LR(1) parser is introduced as a practical bottom-up parser. A translator of infix mathemetical expressions to postfix is constructed as an example. The translator core is subsequently implemented as a library that allows specifying any LR(1) translation. The library functionality is tested by implementing the before constructed translator and by its subsequent testing of the correctness translation outputs for various inputs.

Abstrakt

Táto práca slúži ako úvod do problematiky formálneho prekladu. Čitateľovi predstavuje podstatnú teóriu a následne používa jej poznatky na vytvorenie konkrétneho prekladača. V prvej časti sú definované základy teórie formálnych jazykov. Na tieto poznatky nadväzuje druhá časť, ktorá vo väčšej hĺbke rozoberá spracovanie zdola-hore. Je predstavený kanonický LR(1) parser ako konkrétny praktický parser zdola-hore. Ako príklad je zostrojený prekladač matematických vzorcov z infixovej na postfixovú notáciu. Jadro prekladaču je následne implementované ako knižnica, ktorá dovoľuje špecifikovať ľubovoľný LR(1) preklad. Funkcionalita knižnice je testovaná implementáciou predom zostrojeného prekladaču a následného testovania správnosti výsledkov prekladu rôznych vstupov.

Keywords

formal language, finite automaton, context-free grammar, pushdown automaton, translation schema, syntax-directed translation, bottom-up parsing, LR parser, translator

Klíčová slova

formálny jazyk, konečný automat, bezkontextová gramatika, zásobníkový automat, prekladová schéma, syntakticky riadený preklad, spracovanie zdola-hore, LR parser, prekladač

Reference

SABOL, Matúš. *A Grammatical Formalization of Translation and Its Implementation.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Prof. RNDr. Alexander Meduna, CSc.

A Grammatical Formalization of Translation and Its Implementation

Declaration

Hereby I declare that this bachelor'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.

....
Matúš Sabol
July 31, 2019

Acknowledgements

I would like to express my deepest gratitude to Prof. Meduna, who provided me with guidance, and encouragement throughout the whole making of this thesis, and to Ing. Jirák for consultation and eager willingness to help. I would also like to thank PhD. Křivka for providing study material.

Contents

Chapter 0

Introduction

The field of formal language theory has had a great impact on whole computer science as it provided a base for compilation and interpretation of higher languages, which marked a great milestone in computer science history. One part of formal language theory – formal translation theory – may not have recieved as much notice as some other parts of the formal language theory but its importance should not be diminished, as it provides us with the ability to translate one formal language into another. It is heavily used in compiler construction to translate a human-readable language into a machine-readable one, to give one example.

This thesis is written as a quick introduction to the formal translation theory and as such provides basic theoretical knowledge required for firstly understanding basics of formal language theory, which is further built upon to establish practical ways of processing languages, as well as basic formalisms and computational models for translations. This effort culminates in a creation and implementation of a translator on the top of a canonical LR(1) parser. Please note that the final translator is not meant to be necessarily a generic, industry-level solution, but rather a proof of concept, that can be read, understood, and extended easily.

The thesis is divided into three thematically distinct parts:

Chapter 1 introduces the reader into the fundamentals of formal language theory, defining concepts of alphabets and languages. Further introduced are formal methods of practically representing languages via the means of generation by grammars, respectively. Lastly, formalisms for translations are defined by extending the already mentioned formalisms of language representations.

Chapter 2 delves into more practical aspects of formal languages. Firstly, two major ways of parsing languages – top-down and bottom-up – are presented. The bottom-up parsing is further elaborated upon, which leads up to introduction of an important class of bottom-up parsable grammars - LR grammars, together with methods of constructing a parser for such grammars. These are again extended to allow definition and construction of LR translators. A proof of concept LR(1) translator for infix to postfix expression translation is constructed as an example.

Chapter 3 considers particular implementation details and decisions of the translator constructed in the previous chapter. The core of the parsing and translation stays common across all LR(1) grammars, and as such is abstracted into a library, which is then used to implement the particular translation of infix expressions to postfix. This implementation is then tested for correct behavior.

The reader is expected to have fundamental knowledge of mathemathical sets: what is a set, element inclusion, subset, union, intersection, Carthesian product, *etc.*; knowledge of basing data structures and their properties, such as a stack; a basic understanding of finite state automata is assumed as well: states, transitions, finishing states, *etc.*. These three are the cornerstones of formal languages theory that are mainly expanded upon. There are also several mentions of algorithm time complexity and the Big-O notation in later places of the thesis. While knowledge of these is not essential for understanding the topic on hand, it provides the acknowledged reader with more insight on the final product of this thesis.

Definitions, examples, and algorithms are numbered sequentially within chapters and are concluded using symbol $\square$. Important terms will be emphasized by *italicizing* on their first mention - usually during their definition. Please make note that phrases of latin origin, such as *et cetera* or *verbatim* are italicized as well, as it is common practice to do so.

Chapter 1

Preliminary theory

Before one can delve into the realm of formal language translations, he must first understand the theory leading up to it. Therefore, this initial chapter is dedicated to providing mathematical foundations for the theory that are eventually expanded upon to formalize and construct translations.

Firstly, the reader is introduced to the fundamental building blocks of formal languages – languages themselves and their composition.

Theory around languages is further developed by describing methods of representing said languages in a finite manner. One such method – generation by grammars – is defined and further explored. Grammars are then extended to define grammatically directed translations.

1.1 Polish expression notations

There is a useful way of representing ordinary (*infix*) arithmetic expressions without using parentheses. This notation is referred to as *Polish notation*[1].

The preliminary text, definition and example are taken from [2, sec. 3.1.1].

Definition 1.1. Let Θ be a set of binary operators, and let Σ be a set of operands. Two forms of Polish expression notation, *prefix* Polish and *postfix* Polish are defined recursively as follows:

1. If an infix expression E is a single operand $a \in \Sigma$, then both the prefix Polish and postfix Polish representation of E is a.

2. If $E_1 \, \theta \, E_2$ is an infix expression, where θ is an operator, and E_1 and E_2 are infix expressions and operands of θ, then

   (a) $\theta \, E_1' \, E_2'$ is the prefix Polish representation of $E_1 \, \theta \, E_2$, where E_1' and E_2' are the prefix Polish representations of E_1 and E_2, respectively, and

   (b) $E_1'' \, E_2'' \, \theta$ is the portfix Polish representation of $E_1 \, \theta \, E_2$, where E_1'' and E_2'' are the postfix Polish representations of E_1 and E_2, respectively

3. If (E) is an infix expression, then

[1]This notation was originally described by Polish mathematician Jan Łukasiewicz. Due to the international public having trouble pronouncing his name, the notation is commonly called “Polish” instead.

(a) The prefix Polish representation of (E) is the prefix Polish representation of E, and

(b) The postfix Polish representation of (E) is the postfix Polish representation of E

□

Example 1.2. Consider the infix expression $(a + b) * c$. This expression is of the form $E_1 * E_2$, where $E_1 = (a + b)$ and $E_2 = c$. Thus, the prefix and postfix Polish expressions for E_2 are both c. The prefix expression for E_1 is the same as that for $a + b$, which is $+ab$. Thus the prefix expression for $(a + b) * c$ is $* + abc$.

Similarly, the postfix expression for $a + b$ is $ab+$, so the postfix expression for $(a + b) * c$ is $ab + c*$. □

Throughout the thesis, we will be omitting the word "Polish" when referring to infix and postfix expressions.

1.2 Alphabets and languages

This section describes the bare essentials required for understanding of the subject of this thesis and the formal language theory as a whole: laguages and their building blocks. These theoretical cornerstones should be simple to understand, nonetheless, it is imperative that they are fully understood, as all further concepts rely heavily upon them.

Definitions in this section are taken from [5] and [6]. Further reading on this topic can also be conducted in [2, chap. 0], [7, chap. 1 to 3], and [4, chap. 1 to 7].

Alphabets

The fundamental building block of any formal language is a set of basic symbols called *alphabet*. An alphabet contains all symbols the system is "allowed" to use. Symbols from an alphabet can be sequentially "strung together" to form a *string* over said alphabet – much like letters of english alphabet can be put together in such manner to create words.

Definition 1.3. We define an *alphabet* Σ as a finite non-empty set, whose members are called *symbols*.

We define a *string* α over Σ as

$$\alpha = a_1a_2a_3...a_n; n \in \mathbb{N}, 1 \leq i \leq n, a_i \in \Sigma$$

a sequence of symbols from Σ. A special string contaning zero symbols is called an *empty string* and is denoted by greek letter ε[2]. Such string is still a string over Σ[3].

We denote Σ^* a set of all strings over Σ. We define Σ^+ as $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. □

Example 1.4. Let us define an alphabet $\Sigma = \{0, 1\}$. Strings '11011' and '' (empty string) are both strings over Σ. String 12021 is not a string over Σ, as $2 \notin \Sigma$. □

[2]Epsilon. While various empty string representations can be found across literature, one will mostly see either latin e or greek ε (or its variation ϵ) used.

[3]Strings are also commonly referred to as "words" of the language.

Languages

Definition 1.5. A *language* $L(\Sigma)$ over an alphabet Σ is a set defined as $L(\Sigma) \subseteq \Sigma^*$. If $L(\Sigma)$ is a finite set, we call it a *finite language*, otherwise it's an *infinite language*. □

In layman's terms, language $L(\Sigma)$ is a set of words (strings) that can be formed by alphabet Σ.

Example 1.6. Let there be an alphabet $\Sigma = \{a, b\}$. Let us define a language $L(\Sigma) = \{a^n b^n; n \in \mathbb{N}\}$. This would create an *infinite* language of strings in form *ab*, *aabb*, *aaabbb*, etc.. This is obviosly a subset of Σ^* (all strings possibly made with only a and b), and thus by definition, an alphabet over Σ. □

1.3 Representations of languages

With defining language L a set of strigs over some alphabet Σ comes the problem of representing L. While finite languages can be listed by enumeration, L being finite is rarely the case in practice. Obviously, it is not possible to finitely enumerate an infinite language, therefore a different representation has to be defined.

There are two mainly used methods of finitely defining a potentially infinite language. One of those is using a generative system called a *grammar*. Grammars use defined rules to construct each sentence of a language described by said grammar. One advantage of defining a language by a grammar is that the rules of the grammar impart structure to the sentences defined by them, making parsing and translation easier. [2, sec. 2.1.1]

Other such method uses *recognizers* – finite machines that, given an input, can decide whether that input is a sentence of the recognized language [2, sec. 2.1.4]. They provide a model for parsing languages and usually take form of a finite automaton enhanced with some sort or internal memory.

Definitions and observations in this section are taken from [2, chap. 2, 4] and [1, sec. 2.2]. Various observations and statements will be cited more specifically.

1.3.1 Grammars

In the field of natural languages, grammars set the rules that generate the structure of the language: how words are formed, the word order, where the commas go, *et cetera*. They do not define the meaning of said words or sentences, however. In formal languages, this is very much same, as formal grammars lay down the rules for constructing languages. In other words, a formal grammar *generates* a formal language.

A grammar for a language L uses two finite disjoint alphabets - an alpabet of *nonterminal* symbols N and an alphabet of *terminal* symbols Σ. The terminal symbols alphabet is the alphabet over which L is defined, while the alphabet of nonterminals is used for generation of words in L. This will be described in detail later in this section.

The core of a grammar is a finite set P of formation rules commonly called *productions*, that describe how sentences of the language should be generated. A production is, in its essence, a pair of strings in which the first string can be any string containing at least one nonterminal, while the second string can be any string. [2, sec. 2.1.2]

Definition 1.7. A *grammar* is a 4-tuple $G = (N, \Sigma, P, S)$, whose elements are defined as follows:

N is a finite set of *nonterminals*.

Σ is a finite set of *terminals*, such that $\Sigma \cap N = \emptyset$.

P is a finite subset of $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$.

S is the *starting* nonterminal, $S \in N$.

An element $(\alpha, \beta) \in P$ will be written as $\alpha \to \beta$ and called a *production*. We will call α the "head" of the production, while β will be called the "body". □

Nonterminals are symbols – variables that denote sets of strings. These sets of strings help define the generated language by imposing a hierarchical structure on the language that is key to syntax analysis and translation. One nonterminal in a grammar in distinguished as a starting nonterminal, and the set of strings it denotes is the language generated the grammar. Usually, productions for the starting nonterminal are listed first.

Terminals are the basic symbols from which strings are formed. The set of a grammar's terminals is the alphabet of the language it generates. Another common name for a terminal used throughout literature is "token". [1, sect. 4.2.1]

A grammar defines a language in a recursive manner.

Definition 1.8. A string called *sentential form* of a grammar $G = (N, \Sigma, P, S)$ is defined recursively as follows:

1. S is a sentential form.
2. If $\alpha\beta\gamma; \alpha, \beta, \gamma \in (N \cup \Sigma)^*$ is a sentential form and $\beta \to \delta \in P$, then $\alpha\delta\gamma$ is also a sentential form.

A sentential form of G containing only terminals is called a *sentence generated by grammar G*. The *language generated by grammar G*, denoted $L(G)$, is a set of all sentences generated by grammar G. □

Definition 1.9. A *derivation step* of grammar G is a transition $\Rightarrow$ defined as follows: If $\alpha\beta\gamma \in (N \cup \Sigma)^*$ and $\beta \to \delta \in P$, then $\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$. This is read as "$\alpha\beta\gamma$ directly derives $\alpha\delta\gamma$".

We define a *k-step* derivation $\beta \overset{k}{\Rightarrow} \delta$ a k-fold product of the relation $\Rightarrow$, such that there is a chain of productions *exactly* k number of productions resulting in β being derived into δ.

We define derivation $\beta \overset{*}{\Rightarrow} \delta$ as a k-step derivation, where $k \geq 0$.

We define derivation $\beta \overset{+}{\Rightarrow} \delta$ as a k-step derivation, where $k \geq 1$. □

Example 1.10. Let us define a grammar $G = (\{S\}, \{0, 1\}, P, S)$ with productions in P defined as follows:

$S \to 1S1$

$S \to 0$

as well as the following derivation: $S \Rightarrow 1S1 \Rightarrow 11S11 \Rightarrow 11011$. We can say that $S \overset{3}{\Rightarrow} 11011$, $S \overset{*}{\Rightarrow} 11011$, $S \overset{+}{\Rightarrow} 11011$ as well as $11011 \in L(G)$. It can be seen that $L(G) = \{1^n 0 1^n | n \geq 1\}$. □

For notational convenience, throughout this thesis and other literature as well, productions with the same head nonterminal can have their bodies grouped, separated by the symbol |, read as "or" [1, sec. 2.2.1]. In the light of this, we can rewrite previous productions as $S \to 1S1 \,|\, 0$.

Context-free Grammars

Having provided a general definition of a grammar, it is important to note that this general definition would make for an impractical implementation, due to minimal constraints resulting in many edge-case scenarios. There are various classes of grammars incurring certain restrictions to their productions. Further reading on these classes can be conducted in [2, sec. 2.1.3]. However, one such class of restricted grammars is especially important in formal language theory as its restrictions allow for it to effectively specify most of a programming language's structure, and is also used as a basis of various schemes for specifying translations [2, sec. 2.4].

Definition 1.11. A *context-free grammar* $G = (N, \Sigma, P, S)$ is a grammar with each production in P being in the form $A \to \alpha; A \in N, \alpha \in (N \cup \Sigma)^*$.

A language L generated by a context-free grammar (CFG) is called a *context-free language.* □

While in theory the order of nonterminals chosen for production is not relevant, it is desirable to follow a certain order in practice. There are two main ways in which we can deterministically choose which nonterminal should be used for derivation:

- *leftmost* derivation - always choosing the leftmost nonterminal for production. Denoted $\beta \underset{lm}{\Longrightarrow} \gamma$. Any sentential form derived by a leftmost derivation $S \overset{*}{\underset{lm}{\Longrightarrow}} \alpha$ is called a *left sentential form.*
- *rightmost* derivation - always choosing the rightmost nonterminal for production. Denoted $\beta \underset{rm}{\Longrightarrow} \gamma$. Any sentential form derived by a rightmost derivation $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha$ is called a *right sentential form.*

A convenient way of visualizing derivations on CFGs are *derivation trees* (or *parse trees*).

Definition 1.12. A labeled ordered tree D is a *derivation tree* (or *parse tree*) of a CFG $G = (N, \Sigma, P, S)$ if

- The root of D is S
- If $D_1, ..., D_n$ are subtrees of the direct children of the root, and the root of D_i is labeled X_i , then $S \to X_1, ..., X_n \in P$. If X_i is a nonterminal, D_i is a derivation tree of for $G = (N, \Sigma, P, X_i)$. If X_i is a terminal, D_i is a single node labeled X_i.
- If D_1 is the only subtree of the root of D and $D_1 = \varepsilon$, then $S \to \varepsilon$

□

Example 1.13. Let us define a CFG $G = (\{E, T, F\}, \{a, +, *\}, P, E)$ with productions in P defined as follows:

$E \to E + E \,|\, E * E \,|\, a$

It is immediately apparent that this CFG lets us describe very simple mathematical expressions. Now let's try to derive the string $a_1 + a_2 * a_3$ by this CFG using left-most derivations:

$$E \underset{lm}{\Longrightarrow} E + E \underset{lm}{\Longrightarrow} a_1 + E \underset{lm}{\Longrightarrow} a_1 + E * E \underset{lm}{\Longrightarrow} a_1 + a_2 * E \underset{lm}{\Longrightarrow} a_1 + a_2 * a_3$$

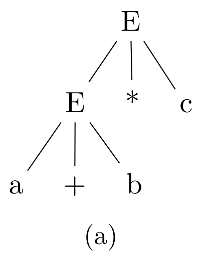
This creates derivation tree figure 1.1a.

So far so good, but the derivation above is not the only way to derive $a_1 + a_2 * a_3$ by this CFG using left-most derivations. Another possible way to do so is:

$$E \underset{lm}{\Longrightarrow} E * E \underset{lm}{\Longrightarrow} E + E * E \underset{lm}{\Longrightarrow} a_1 + E * E \underset{lm}{\Longrightarrow} a_1 + a_2 * a_3 \underset{lm}{\Longrightarrow} a_1 + a_2 * a_3$$

This creates derivation tree figure 1.1b.

This CFG is *ambiguous* – meaning that it produces more than one leftmost or more than one rightmost derivation for the same sentence [1, sec. 4.2.5]. This is also true the other way around: if the grammar produces at most one leftmost and at most one rightmost derivation for a single sentence, it is *unambiguous*. □



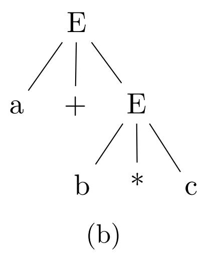


Figure 1.1: Parse trees of $a+b*c$ produced by G in example 1.13 using leftmost derivations

1.3.2 Recognizers

The second common method for finitely specifying a possible infinite language is to define a recognizer for it. There are three main parts to a recognizer – an input tape, a finite state control, and an auxillary memory. A recognizer operates by making a sequence of *moves*. Each move consists of moving the input head one cell to the left, one cell to the right, or not moving the head at all, reading the symbol from the input tape, storing the input into the memory and changing the state of the control to determine its next action.

The current state of a recognizer can be described by a *configuration*. A configuration contains

- The state of the finite control.
- The status of the input tape.
- The state of the auxillary memory.

We assume that the input tape is read from left to right. It is said that a recognizer is *one-way* if the input head cannot move to the left. Normally, it is assumed that the input tape is read-only, meaning no changes can be made to the input by the recognizer.

It is said that a recognizer is *deterministic* if in each configuration there is at most one possible move. Otherwise the recognizer is *nondeterministic*. While nondeterministic recognizers are a convenient abstraction, they are often difficult to simulate in practice. [2, sec. 2.1.4]

Finite automata

We will first introduce the *finite automaton* – the simplest form of recognizer. Its auxillary memory is null. A finite automaton is a one-way recognizer that is required to move its head on each move. It can be nondeterministic as well as deterministic, but we will consider only deterministic finite automata for the purposes of this thesis. [2, sec. 2.2.3]

Definition 1.14. A *deterministic finite automaton* (FA) is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

Q is a finite set of states

Σ is a finite set of *input symbols*

δ is a mapping $(Q \times \Sigma) \to Q$, called *state transition function*

$q_0 \in Q$ is the *initial state* of the finite state control

$F \subseteq Q$ is the set of *final states*

$\square$

Definition 1.15. We define a *configuration* of FA $M = (Q, \Sigma, \delta, q_0, F)$ a pair (q, w) where $q \in Q$ is the current state of M and $w \in \Sigma^*$ is the remaining input.

A configuration (q_0, w) is called an *initial configuration*.

A configuration (q_F, ε) where $q_F \in F$ is called a *final configuration*.

A *move* by M is represented by a binary relation $\vdash$ on configurations. If $\delta(q, a) \ni q'$, then $(q, aw) \vdash (q', w)$. $\square$

Although it is notable that FA are recognizers for an important class of languages called *regular languages* [2, sec. 2.2], this is not important for the purposes our thesis. The FA was defined since it's the simplest of recognizers and all other recognizers in this thesis "extend" this simple recognizer in one way or another.

Pushdown automata

We will now introduce one such extension – the *pushdown automaton* – a recognizer that models context-free language parsers. The pushdown automaton is a one-way nondeterministic recognizer with infinite storage that consists of a single stack (pushdown list). [2, sec. 2.5]

Definition 1.16. A *pushdown automaton* (PDA) is a 7-tuple

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where items common with FA retain their meaning. Additionally

Γ is a finite *pushdown alphabet*

δ is a state transition function $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$

$Z_0 \in \Gamma$ is the *initial pushdown symbol*

□

Definition 1.17. We define a *configuration* of PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ a triple (q, w, α) such that $q \in Q$ is the current finite control symbol, $w \in \Sigma^*$ is the remaining input, and $\alpha \in \Gamma^*$ are the contents of the pushdown list.

Configuration (q_0, w, Z_0) is the initial configuration of PDA P. Configuration $(q_F, \varepsilon, \alpha)$ where $q_F \in F, \alpha \in \Gamma^*$ is the final configuration of PDA P.

A move by P is represented by a binary relation $\vdash$ on configurations. We write

$$(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha)$$

if $\delta(q, a, Z) \ni (q', \gamma)$. □

1.4 Formalisms for translations

Having covered the basic representations of languages in the form of grammars and recognizers, we now move onto describing basic representations of translation between languages. We will see that these representations are extensions of the already defined language representations – syntax-directed translation schemas extend grammars with a second set production rule bodies representing the output grammar, while pushdown transducers extend pushdown automata with an output tape.

These formalisms will be elaborated upon in chapter 2 to define proper translators.

1.4.1 Syntax-Directed Translation Schema

A syntax-directed translation schema is essentially a grammar with translation elements provided with each rule. Every time a certain rule is used in the input derivation step, the translation element is used to determine a part of the output associated with the input generated by that rule. They are often times also called *translation grammars*, but we will use the term syntax-directed translation schema in this thesis.

All definitions, examples and statements are taken from [2, sec. 3.1.2].

There are several desirable features in translation definitions, two of them being:

1. It should be easy to determine the translation pairs.
2. It should be possible to construct a translator directly from the definition using an algorithm.

As with translation definitions, there are some particular features that are desirable in translators. Some of them are:

1. Time efficiency - their time to process string of length n should be $O(n)$.
2. Small size.
3. Ability to create small finite test such that if the translator passes this test, it would guarantee correct working on all inputs.

While there may be several ways to formally describe translations, in this thesis we will only consider syntax-directed translation schemata and pushdown transducers as means of doing so.

Definition 1.18. A *syntax-directed translation schema* (SDTS for short) is a 6-tuple

$$T = (N, \Sigma, \Delta, R, S)$$

where

1. N is a finite set of *nonterminal symbols*
2. Σ is a finite *input alphabet*
3. Δ is a finite *output alphabet*
4. R is a finite set of *rules* of the form $A \to \alpha, \beta$, where $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$, and the nonterminals in β are a permutation of the nonterminals in α
5. S is the starting nonterminal, $S \in N$

$\square$

Let $A \to \alpha, \beta$ be a rule. To each nonterminal of α there is *associated* an identical nonterminal of β. If a nonterminal B appears only once in α and β, the association is obvious. If B appears more than once, we use integer superscripts to indicate the association. This association is an intimate part of the rule.

A SDTS defines a translation in a recursive manner.

Definition 1.19. A pair called *translation form* of a SDTS $T = (Q, \Sigma, \Delta, R, S)$ is defined recursively as follows:

1. (S, S) is a translation form and the two S's are said to be *associated.*

2. If $(\alpha A\beta, \alpha' A\beta')$ is a translation form, in which the two explicit instances of A are associated, and if $A \to \gamma, \gamma' \in R$, then $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ is a translation form. The nonterminals of γ and γ' are associated in the translation form exactly as they are associated in the rule.

If the forms $(\alpha A\beta,\ \alpha' A\beta')$ and $(\alpha\gamma\beta, \alpha'\gamma'\beta')$, together with their associations, are related as above, then we write $(\alpha A\beta, \alpha' A\beta') \Rightarrow (\alpha\gamma\beta, \alpha'\gamma'\beta')$. We then define $\overset{k}{\Rightarrow}$, $\overset{*}{\Rightarrow}$, $\overset{+}{\Rightarrow}$ similarly as in definition 1.9.

The *translation defined by* T, denoted $\tau(T)$ is the set of translation forms such that

$$\{(x, y) \mid (S, S) \overset{*}{\Rightarrow} (x, y), x \in \Sigma^*, y \in \Delta^*\}$$

□

As we can see from the definitions, a SDTS structurally very similar to a CFG, with the exception that every rule now has two bodies - first one representing the input grammar rule body and the second one representing the ouput grammar rule body.

For further formal needs, we will also define input and output grammar.

Definition 1.20. Define a SDTS $T = (N, \Sigma, \Delta, R, S)$. The grammar

$$G_i = (N, \Sigma, P, S)$$

where $P = \{A \to \alpha \mid A \to \alpha, \beta \in R\}$ is called *underlying* (or *input*) grammar of T. The grammar

$$G_o = (N, \Sigma, P', S)$$

where $P' = \{A \to \beta \mid A \to \alpha, \beta \in R\}$ is called the *output grammar* of T. □

Definition 1.21. A SDTS $T = (N, \Sigma, \Delta, R, S)$ such that in each rule $A \to \alpha, \beta \in R$, associated nonterminals occur in the same order in α and β is called a *simple* SDTS. The translation defined by a simple SDTS is called a *simple syntax-directed translation* (simple SDT). □

The simple SDTs are important because for each simple SDT we can easily construct a translator consisting of a PDT 1.23. Many, but not all, useful translations can be described as a simple SDT.

Example 1.22. Let there be a SDTS $T = (\{E\}, \{+, *, a\}, \{+, *, a\}, R, E)$, with R containing following rules:

$E \to E^1 + E^2,\ E^1E^2+$

$E \to E^1 * E^2,\ E^1E^2*$

$E \to (E),\ E$

$E \to a,\ a$

We can see from definition that the underlying grammar defines infix expressions over alphabet $\{+, *, a\}$, while the output grammar defines postfix expressions over the same alphabet. By definition 1.21, T is a simple SDTS.

Consider the following derivation of expression $a_1 * a_2 + a_3$. Terminals a have been indexed with a subscript to show their order after translation.

$$\begin{aligned}(E, E) &\Rightarrow (E^1 + E^2,\, E^1E^2+)\\ &\Rightarrow (E^1 * E^2 + E^3,\, E^1E^2 * E^3+)\\ &\Rightarrow (a_1 * E^1 + E^2,\, a_1E^2 * E^2+)\\ &\Rightarrow (a_1 * a_2 + E,\, a_1a_2 * E+)\\ &\Rightarrow (a_1 * a_2 + a_3,\, a_1a_2 * a_3+)\end{aligned}$$

We see that the infix expression $a_1 * a_2 + a_3$ has been translated to a posfix expression $a_1a_2 * a_3+$. As per definition 1.1, we can see that this translation is correct. □

1.4.2 Pushdown transducers

We will now introduce an important class of translators called pushdown transducers. Pushdown transducer are obtained by providing a pushdown automaton with an output, that is, on each step the automaton is allowed to emit a finite-length output string. All definitions, statements, and examples are taken from [2, sec. 3.1.4].

Definition 1.23. A *pushdown transducer* (PDT) P is an 8-tuple

$$P = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$$

where all symbols have the same meaning as for a pushdown automaton, except that Δ is an *outptut alphabet* and δ is now mapping from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^* \times \Delta^*$. □

The configuration and transition is defined similarily to a pushdown automaton, although with the difference of adding the current state of the output.

Definition 1.24. We define a *configuration* of P as a 4-tuple (q, w, γ, y), where q, w, and γ are the same as for a PDA and y is the output string emmited to this point. If $\delta(q, x, Z) \ni (r, \alpha, z)$, then we write $(q, aw, Z\gamma, y) \vdash (r, w, \alpha\gamma, yz)$ for all $w \in \Sigma^*, \gamma \in \Gamma^*$, and $y \in \Delta^*$.

We say that y is an output for w if $(q_0, w, Z_0, \varepsilon) \vdash^* (q, \varepsilon, \alpha, y)$ for some $q \in F$ and $\alpha \in \Gamma^*$. The *translation defined by P* , denoted $\tau(P)$, is

$$\{(x, y) \,|\, (q_0, w, Z_0, \varepsilon) \vdash^* (q, \varepsilon, \alpha, y), q \in Q, \alpha \in \Gamma^*\}.$$

□

As with pushdown automata we can say that y is an output for x *by empty pushdown list* if $(q_0, x, Z_0, \varepsilon) \vdash^* (q, \varepsilon, \varepsilon, y), q \in Q$. The *translation defined by P by empty pushdown list* is

$$\{(x, y) \mid (q_0, w, Z_0, \varepsilon) \vdash^* (q, \varepsilon, \varepsilon, y), q \in F\}.$$

Example 1.25. Let there be a PDT $P = (\{q\}, \{a, +, *\}, \{+, *, E\}, \{a, +, *\}, \delta, q, E, \{q\})$ with δ defined as follows:

$$\delta(q, a, E) = \{(q, \varepsilon, a)\}$$

$$\delta(q, +, E) = \{(q, EE+, \varepsilon)\}$$

$$\delta(q, *, E) = \{(q, EE*, \varepsilon)\}$$

$$\delta(q, \varepsilon, +) = \{(q, \varepsilon, +)\}$$

$$\delta(q, \varepsilon, *) = \{(q, \varepsilon, *)\}$$

With the input $+*aaa$, P makes the following sequence of moves:

$$\begin{aligned}(q, +*aaa, E, \varepsilon) &\vdash (q,\ *aaa, EE+, \varepsilon)\\ &\vdash (q,\ aaa, EE*E+, \varepsilon)\\ &\vdash (q,\ aa, E*E+, a)\\ &\vdash (q,\ a, *E+, aa)\\ &\vdash (q,\ a, E+, aa*)\\ &\vdash (q,\ \varepsilon, +, aa*a)\\ &\vdash (q,\ \varepsilon, \varepsilon, aa*a+)\end{aligned}$$

Thus a translation by empty pushdown list of $+*aaa$ is $aa*a+$. It can be verified that $\tau_\varepsilon(P)$ is the set:

$$\begin{aligned}&\{(x,y) \mid x \text{ is a prefix arithmetic expression over } \{+, *, a\}\\ &\text{and } y \text{ is the corresponding postfix expression}\}\end{aligned}$$

□

Chapter 2

Parsing and translation

Having introduced fundamentals of formal languages, their representations and translation formalisms, we will now be presenting practical methods of language recognition via grammatical parsers, and translation.

Firstly we define two main approaches to practically parsing grammars – top-down and bottom-up. Top-down parsing is only briefly mentioned as it's not the focus of this thesis. On the other hand, bottom-up parsing is explored more deeply. This culminates in definition of a special class of bottom-up parsable grammars – the LR grammars.

LR grammars are further explored and a method of creating an LR grammar parser is defined. This LR parser construction process is then expanded to allow for creation of LR translators.

In the last part of this chapter we formally construct a particular LR translator that translates infix expressions into postfix, step by step. This constructed parser, will be then implemented in chapter 3.

2.1 Parsing methods

In compiler design, parsing is often referred to as "syntax analysis". This means that the parser is fed tokens – terminals from the input and tries to construct a parse tree of the input. If the input is well formed according to the parsed language grammar, a parse tree is successfuly constructed by the parser. While this parse tree can be actually "physically" constructed in the parser's memory, most of the time its construction is only simulated. [2, sec. 3.4.1]

Firstly, the top-down method is briefly explained, as to give the reader at least an idea of it. We stop at that, since its deeper understanding is not necessary for this thesis. Then the bottom-up method is explored more in depth, since we are more interested in the bottom-up methods for reasons that will become clear later on.

2.1.1 Top-down parsing

Top-down parsing can be viewed as finding a leftmost derivation for an input string. Equivalently, we can look at top-down parsing as the problem of constructing a parse tree for the input string, starting from the root and creating the parse tree nodes in a depth-first manner [1, sec. 4.4]. For this short overview, however, we will only consider the first way of understanding top-down parsing.

There exists a class of grammars that can be naturally parsed in a top-down fashion. They are called *LL(k) grammars*, meaning that they scan the input from **l**eft to right producing a **l**eft parse, using k symbols of lookahead at each step to make parsing action decisions.

The class of LL(1) grammars is rich enough to cover most programming constructs, although consideration is required for writing a proper LL(1), or any LL(k) grammar, since the grammar must be unambiguous and not left-recursive to be parsed naturally by a LL(k) parser.

The topic of LL(k) grammars will not be discussed further in this thesis, since it would be unnecessary to introduce them more formally and rigorously because we will not be using them further down the line. Further reading on these, however, can be conducted in [2, sec. 3.4.2] and [1, sec. 4.4.3], which is also the source of claims and "definitions" in this subsection.

2.1.2 Bottom-up parsing

As opposed to top-down parsing where inputs are parsed by conducting leftmost derivations, or by creating parse trees from their roots; bottom-up parsers work in a fundamentally opposite manner. Bottom-up parser conduct rightmost derivations, creating a *right parse*, and contruct parse trees from leaves, working their way up to the root.

We will now define a PDT which implements a SDTS T_r which maps words from language L to their right parses. We shall define an extended PDT, which will serve as a model for bottom-up parsers.

Definition 2.1. An *extended PDT* (EPDT) is an 8-tuple $P = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$ where all symbols retain their meaning from the definition of PDT, with the exception of δ which now maps a finite subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*$ to the finite subsets of $Q \times \Gamma^* \times \Delta^*$. Configurations are defined as before, but with the pushdown top on the right, and we say that $(q, aw, \beta\alpha, x) \vdash (p, w, \beta\gamma, xy) \iff \delta(q, a, \alpha) \ni (p, \gamma, y)$.

The EPDT P is deterministic if:

1. $\forall q \in Q, a \in \Sigma \cup \{\varepsilon\}, \alpha \in \Gamma^* : |\delta(q, a, \alpha)| \leq 1$, and
2. $\delta(q, a, \alpha) \neq \emptyset \wedge \delta(q, b, \beta) \neq \emptyset$, with $b = a \vee b = \varepsilon$, then neither of α and β is a suffix of the other.

□

Definition 2.2. Let $G = (N, \Sigma, P, S)$ be a CFG. Let M_r^G be the nondeterministic EPDT $(\{q\}, \Sigma, N \cup \Sigma \cup \{\$\}, \{1, ..., p\}, \delta, q, \$, \emptyset)$. The pushdown top is on the right, and δ is defined as follows:

1. $\delta(q, \varepsilon, \alpha) \ni (q, A, i)$ if production i in P is $A \to \alpha$.
2. $\delta(q, a, \varepsilon) = \{(q, a, \varepsilon)\}, \forall a \in \Sigma$.
3. $\delta(q, \varepsilon, \$S) = \{(q, \varepsilon, \varepsilon)\}$.

This EPDT embodies the elements of what is known as a *shift-reduce* parsing algorithm. Under rule 2, M_r shifts input symbols onto the top of the pushdown list. Whenever a handle appears on top of the pushdown, M_r can reduce the handle under rule 1 and emit the number of the production used to reduce the handle. M_r may then shift more input symbols onto the pushdown, until the next handle appears on top of it. The handle can then be reduced and the production number emitted, *et cetera.* M_r continues to operate in this fashion until the pushdown contains only the nonterminal on top the end of pushdown marker. Unde rule 3 M_r can then enter a configuration in which the pushdown is empty. $\square$

Example 2.3. Let there be a CFG $G_{if} = (\{E, T, F\}, \{+, *, a, (,)\}, P, E)$ where P containing following rules

(1) $E \to E + T$

(2) $E \to T$

(3) $T \to T * F$

(4) $T \to F$

(5) $F \to (E)$

(6) $F \to a$

This grammar denotes infix expressions with addition, multiplication and parentheses.

The right parser for G_{if} would be

$$M_r^{G_{if}} = (\{q\}, \Sigma, N \cup \Sigma \cup \{\$\}, \{1, ..., |P|\}, \delta, q, \$, \emptyset)$$

where

$$\begin{aligned}
\delta(q, \varepsilon, E + T) &= \{(q, E, 1)\} \\
\delta(q, \varepsilon, T) &= \{(q, E, 2)\} \\
\delta(q, \varepsilon, T * F) &= \{(q, T, 3)\} \\
\delta(q, \varepsilon, F) &= \{(q, T, 4)\} \\
\delta(q, \varepsilon, (E)) &= \{(q, F, 5)\} \\
\delta(q, \varepsilon, a) &= \{(q, F, 6)\} \\
\delta(q, t, \varepsilon) &= \{(q, t, \varepsilon)\}; \forall t \in \Sigma \\
\delta(q, \varepsilon, \$E) &= \{(q, \varepsilon, \varepsilon)\}
\end{aligned}$$

With input $a_1 + a_2 * a_3$, $M_r^{G_{if}}$ could make the following sequence of moves, among others:

$$\begin{aligned}
(q, a_1 + a_2 * a_3, \$, \varepsilon) &\vdash (q, +a_2 * a_3, \$a_1, \varepsilon) \\
&\vdash (q, +a_2 * a_3, \$F, 6) \\
&\vdash (q, +a_2 * a_3, \$T, 64) \\
&\vdash (q, +a_2 * a_3, \$E, 642) \\
&\vdash (q, a_2 * a_3, \$E+, 642) \\
&\vdash (q, *a_3, \$E + a_2, 642) \\
&\vdash (q, *a_3, \$E + F, 6426) \\
&\vdash (q, *a_3, \$E + T, 64264) \\
&\vdash (q, a_3, \$E + T*, 64264) \\
&\vdash (q, \varepsilon, \$E + T * a_3, 64264) \\
&\vdash (q, \varepsilon, \$E + T * F, 642646) \\
&\vdash (q, \varepsilon, \$E + T, 6426463) \\
&\vdash (q, \varepsilon, \$E, 64264631) \\
&\vdash (q, \varepsilon, \varepsilon, 64264631)
\end{aligned}$$

Thus, $M_r^{G_{if}}$ would produce the right parse 64264631 for the input string $a_1 + a_2 * a_3$. □

Definition 2.4. A SDTS is *semantically unambiguous* if there are no two distinct rules of the form $A \to \alpha, \beta$ and $A \to \alpha, \gamma$. □

A semantically unambiguous SDTS has exactly one translation element for each production of the underlying grammar.

The definitions and examples in this subsections were taken from [2, sec. 3.4.3].

2.2 LR(k) parsing and translation

The previous section we saw definition for general bottom-up parsing that serves as a more accurate model of the workings of a bottom-up parser, more precisely an undeterministic shift-reduce parser. Moving on into more practical circumstances, only deterministic parsers will be of interest to us.

Just as LL(k) is a class of CFGs that was mentioned to be deterministically parsable by top-down parsers in subsection 2.1.1, there exists a similar class of grammars that is deterministically parsable reading the input **l**eft to right, producing **r**ightmost derivations, called *LR(k)* grammars. As was the case with LL(k), k is the number of lookahead symbols required to make parsing action decisions.

We will cover mainly the most robust type of LR(k) parsers, so-called *canonical LR(k) parsers*. Simpler and in turn, weaker types or LR(k) parsers – *SLR* and *LALR* will be mentioned as well.

Definitions, statements and examples from this preliminary text as well as the following subsections were taken mostly from [1, sec. 4.6, 4.7] and some from [2, sec. 5.2].

2.2.1 LR(k) grammars

In this subsection we will define a large class of grammars for which we can always construct deterministic right parsers. These grammars are the LR(k) grammars.

Informally, we say that a grammar is LR(k) if given a rightmost derivation, we can isolate the handle of each right-sentential form and determine which nonterminal is to replace the handle by scanning the remaining input from left to right, but only going at most k symbols into the remaining input.

Before we define the term LR(k) grammar, let us introduce the simple concept of an augmented grammar.

Definition 2.5. Let $G = (N, \Sigma, P, S)$ be a CFG. We define the *augmented grammar* derived from G as $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S')$. The augmented grammar G' is merely G with a new starting production $S' \to S$, where $S' \notin N$ is a new start symbol. We assume that $S' \to S$ is the zeroth production in G' and that the other productions of G are numbered $1, 2, ..., p$. We add the starting production so that when a reduction using the zeroth production is called for, we can interpret this "reduction" as a signal to accept. □

We shall now give a precise definition of an LR(k) grammar.

Definition 2.6. Let $G = (N, \Sigma, P, S)$ be a CFG and let $G' = (N', \Sigma, P', S')$ be its augmented grammar. We say that G is LR(k), $k > 0$, if the three conditions

1. $S' \underset{G'\,rm}{\overset{*}{\Longrightarrow}} \alpha A w \underset{G'\,rm}{\Longrightarrow} \alpha\beta w$,
2. $S' \underset{G'\,rm}{\overset{*}{\Longrightarrow}} \gamma B y \underset{G'\,rm}{\Longrightarrow} \gamma\beta x$, and
3. $\text{FIRST}_k(w) = \text{FIRST}_k(y)$

imply that $\alpha A y = \gamma B x$, meaning that $\alpha = \gamma$, $A = B$, and $x = y$.

A grammar is LR if it is LR(k) for some k. □

This definition says that if $\alpha\beta w$ and $\alpha\beta y$ are right-sentential forms of the augmented grammar with $\text{FIRST}_k(w) = \text{FIRST}_k(y)$ and if $A \to \beta$ is the last production used to derive $\alpha\beta w$ in a rightmost derivation, then $A \to \beta$ must also be used to reduce $\alpha\beta y$ to $\alpha A y$ in a right parse. Since A can derive β independently of w, the LR(k) conditions says that there is sufficient information in $\text{FIRST}_k(w)$ to determine that $\alpha\beta$ was derived from αA. Thus there can never be any confusion about how to reduce any right-sentential form of the augmented grammar. In addition, with an LR(k) grammar we will always know whether we should accept the present input string, or continue parsing. If the start symbol doesn't appear in the body of any production, we can alternatively defina an LR(k) grammar $G = (N, \Sigma, P, S)$ as one in which the three conditions

1. $S \underset{G'\,rm}{\overset{*}{\Longrightarrow}} \alpha A w \underset{G'\,rm}{\Longrightarrow} \alpha\beta w$,
2. $S \underset{G'\,rm}{\overset{*}{\Longrightarrow}} \gamma B y \underset{G'\,rm}{\Longrightarrow} \gamma\beta x$, and
3. $\text{FIRST}_k(w) = \text{FIRST}_k(y)$

imply that $\alpha Ay = \gamma Bx$.

The reason we cannot always use this definition is that if the start symbol appears in the body of some production we may not be able to determine whether we have reached the end of the input string and should accept or whether we should continue parsing.

We will now show for each LR(k) grammar $G = (N, \Sigma, P, S)$ we can construct a deterministic right parser which behaves in the following manner.

First of all, the parser will be constructed from the augmented grammar G'. The parser will behave very much like the shift-reduce parser introduced in example 2.3, except that the LR(k) parser will put special information symbols, called LR(k) tables, on the pushdown list above each grammar symbol on the pushdown list. These LR(k) tables will determine whether a shift move or a reduce move is to be made and, in the case of a reduce move, which production is to be used.

An LR(k) parser for a CFG G is nothing more than a set of rows in a large table, where each row is called an "LR(k) table". One row is distinguished as the initial LR(k) table. Each LR(k) table consists of two functions – a *parsing action function* f and a *goto function* g:

1. A parsing action function f takes a string $u \in \Sigma^{*k}$ as argument (this string is called the *lookahead string*), and the value of $f(u)$ is either **shift** t, **reduce** i, **error**, or **accept**.
2. A goto function g takes a symbol $X \in N$ as argument and has as value either the name of another LR(k) table, or **error**.

Construction of LR(k) tables will be explained further down the line.

The LR parser behaves as a shift-reduce parsing algorithm, using a pushdown list, an input tape, and an output buffer. At the start, the pushdown list contains the initial LR(k) table T_0 and nothing else. The input tape contains the word to be parsed, and the output buffer is initially empty. If we assume that the input word to be parsed is $aabb$, then the parser would initially be in configuration

$$(T_0, aabb, \varepsilon)$$

Parsing then proceeds by performing the following algorithm:

Algorithm 2.7. LR(k) parsing algorithm.

Input: A set $\Im$ of LR(k) tables for an LR(k) grammar $G = (N, \Sigma, P, S)$, with $T_0 \in \Im$ designated as the initial table, and an input string $z \in \Sigma^*$, which is to be parsed.

Output: If $z \in L(G)$, the right parse of G. Otherwise, an error indication.

Method: Perform steps 1 and 2 until acceptance occurs or an error is encountered. If acceptance occurs, the string in the output buffer is the right parse of z.

1. The lookahead string u, consisting of the next k input symbols is determined.
2. The parsing action function f of the table on the top of the pushdown list is applied to the lookahead string u.

(a) If $f(u) =$ **shift** i, then the next input symbol, say a, is removed from the input and shifted onto the pushdown list. Then, the argument i determines which new table should be placed on top of the pushdown list. We then return to step 1. If there is no next input symbol, halt and declare error.

(b) If $f(u) =$ **reduce** i and production i is $A \to \alpha$, then $2|\alpha|$ symbols[1]are removed from the top of the pushdown list, and production number i is placed in the output buffer. A new table T' is then exposed as the top table of the pushdown list, and the goto function of T' is applied to A to determine the next table to be placed on top of the pushdown list. We place A and this new table on top of the pushdown list and return to step 1.

(c) If $f(u) =$ **error**, we halt parsing.

(d) If $f(u) =$ **accept**, we halt and declare the string in the output buffer to be the the right parse of the original input string.

$\square$

We shall now develop the theory necessary to construct LR(k) parsers.

Definition 2.8. Suppose that $S \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha\beta w$ is a rightmost derivation in grammar G. We say that a string γ is a *viable prefix* of G if γ is a viable prefix of $\alpha\beta$. That is, γ is a string which is a prefix of some right-sentential form but which does not extend past the right end of the handle of that right-sentential form.

Definition 2.9. Let $G = (N, \Sigma, P, S)$ be a CFG. We say that $[A \to \beta_1 \bullet \beta_2, u]$ is an LR(k) item (for k and G, but we usually omit reference to these parameters when they are understood) if $A \to \beta_1\beta_2 \in P$ and $u \in \Sigma^{*k}$. We say that LR(k) item $[A \to \beta_1 \bullet \beta_2, u]$ is valid for $\alpha\beta_1$, a viable prefix of G, if there is a derivation $S \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha\beta_1\beta_2$ such that $u = \text{FIRST}_k(w)$. Note that fix may be ε and that every viable prefix has at least one valid LR(k) item. $\square$

The LR(k) items associated with the viable prefixes of a grammar are the key to understanding how a deterministic right parser for an LR(k) grammar works. In a sense we are primarily interested in LR(k) items of the form $[A \to \beta\bullet, u]$, where the dot is at the right end of the production. These items indicate which productions can be used to reduce right-sentential forms. The next definition and theorem are at the heart of LR(k) parsing.

Definition 2.10. We define the *ε-free first function*, $\text{EFF}^G_k(\alpha)$ as follows (we shall delete the k and/or G when clear):

1. If α begins with a terminal, then $\text{EFF}_k(\alpha) = \text{FIRST}_k(\alpha)$.

2. If α begins with a nonterminal, then

$$\text{EFF}_k(\alpha) = \{w \mid \exists \alpha \underset{rm}{\overset{*}{\Longrightarrow}} \beta \underset{rm}{\Longrightarrow} wx, \forall A \in N : \beta \neq Awx\}, \text{ and } w = \text{FIRST}_k(wx)$$

$\square$

[1]If $\alpha \to X_m, ..., X_r$, at this point the top of the pushdown list will be of the form $T_0X_1T_1...X_r, T_r$. Removing $2|\alpha|$ symbols removes the handle from the top of the pushdown list along with any intervening LR tables

Thus, $\mathrm{EFF}_k(\alpha)$ captures all members of $\mathrm{FIRST}_k(\alpha)$ whose derivation does not involve replacing a leading nonterminal by ε (equivalently, whose rightmost derivation does not use an ε-production at the last step, when α begins with a nonterminal).

LR(k) parsing techniques are based on the following theorem.

Theorem 2.11. A grammar $G = (N, \Sigma, P, S)$ is LR(k) if and only if the following condition holds for each $u \in \Sigma^{*k}$. Let $\alpha\beta$ be a viable prefix of a right-sentential form $\alpha\beta w$ of the augmented grammar G'. If LR(k) item $[A \to \beta\bullet, u]$ *is valid* for $\alpha\beta$, then there is no other LR(k) item $[A_i \to \beta_1 \bullet \beta_2, v]$ which is valid for $\alpha\beta$ with $u \in \mathrm{EFF}_k(\beta_2 v)$. (Note that β_2 may be ε.) □

The proof of this theorem can be found in [2, Theorem 5.9].

The construction of a deterministic right parser for an LR(k) grammar requires knowing how to find all valid LR(k) items for each viable prefix of a right-sentential form.

Definition 2.12. Let G be a CFG and γ a viable prefix of G. We define $V_k^G(\gamma)$ to be the set of LR(k) items valid for γ, with respect to k and G. We again delete k and/or G if understood. We define S_V as the collection of the sets of valid LR(k) items for G. S_V contains all sets of LR(k) items which are valid for some viable prefix of G.

Due to lack of time, this section was not finished. If it were, however, it would continue defining algorithms for construction of LR(k) parsers from [2, sec. 5.2], which would be used in section section 2.3 to show a step-by-step construction of LR(1) parser as an example and a testament to the complexity of constructing one.

2.2.2 LR(k) translation

Due to lack of time, this section could not even be started. If there was enough time for it, however, it would contain definitions from [3, sec. 9.2.1, sec. 9.2.3] describing translators with underlying LR(k) grammars and their construction. These, together with definitions and algorithms from 2.2.1 would be used to implement robust translation part of the output library implemented in 3.

2.3 Infix to postfix translator

This sections was supposed to show a step-by-step creation of LR(k) parser and translator from underlying SDTS for translation of infix expressions to postfix, to provide basis for an implementable example in the next chapter. Unfortunately, due to lack of time to finish the thesis, you will only see its extremely abridged version: the definition of the underlying SDTS and the final LR(1) parse tables.

This section will see the definition of a SDTS for translation of simplified methematical expressions from infix to postfix and LR(1) parsing table resulting from the underlying grammar of the SDTS.

The SDTS for translation of infix expression to postfix is

$$T_{if2pf} = (\{S, E, T, F\}, \{id, +, *, (,)\}, \{id, +, *, (,)\}, R, S)$$

with R defined as:

$$\begin{aligned}
&S \to E, \varepsilon \\
&E \to E + T, + \\
&E \to T, \varepsilon \\
&T \to T * F, * \\
&T \to F, \varepsilon \\
&F \to (E), \varepsilon \\
&F \to id, id
\end{aligned}$$

where translation production bodies specify outputted terminals, as the current implentation only supports terminal outputting after the production had been applied.

Following the algorithms in section 2.2.1, a set of LR(1) tables was created for T_{if2pf}.

	ACTION						GOTO			
	+	*	(	)	**id**	$	S	E	T	F
T0			s4		s5			1	2	3
T1	s6					acc				
T2	r2	s7				r4		8	9	10
T3	r4	r4				r4				
T4			s11		s12			8	9	10
T5	r6	r6				r6				
T6			s4		s5				13	3
T7			s4		s5					14
T8	s16			s15						
T9	r2	s17		r2						
T10	r4	r4		r4						
T11			s11		s12			18	9	10
T12	r6	r6		r6						
T13	r1	s7				r1				
T14	r3	r3				r3				
T15	r5	r5				r5				
T16			s11		s12				19	10
T17			s11		s12					20
T18	s16			s21						
T19	r1	s17		r1						
T20	r3	r3		r3						
T21	r5	r5		r5						

Table 2.1: LR(1) tables of our infix grammar parsing

Chapter 3

Translator implementation

3.1 `ptlib`

The most important output of this thesis is parsing and translation library – `ptlib`. It was implemented in C++, mainly for performance reasons, but also due to author's personal preference.

The library has 3 main components - the lexer which reads tokens defined by regular expressions from the input, the parser that implements an LR(1) parser which uses these tokens to decide what move to take. Lastly, there is the translation part that executes translation actions on parser's reduction moves.

Unfortunately, as I ran out of time to finish the thesis (as you've probably read several times at this point), I will be omitting more detailed description of the `ptlib` components. If there was time to finish it, each previously described component would be described from its "grand scheme" standpoint, together with more detailed explanations of particular implementation details.

3.2 `if2pf`

3.2.1 Implementation

The LR(1) parser constructed in section 2.3 was implemented by providing constructed LR(1) tables, input, and output grammar productions to the classes implemented by `ptlib`. Due to the library hiding all the parsing logic behind a few setup calls and a single function call for translation, and as such there is not much to talk about here.

3.2.2 Testing

The `if2pf` program was tested first manually on some inputs, and eventually a bash script was created together with a file containing a list of inputs as well as expected program outputs and it would automatically run all these tests and check results. Test cases started from simple and progressed to more and more complex. However, there is only so much complexity one can get from addition, multiplication and parentheses in mathematical expressions.

Chapter 4

Conclusion

In this thesis, the reader was introduced to the basic theory of formal languages and translations. We have defined formalisms for parsing and introduced LR(k) parsers to implement deterministic bottom-up parsing and translation with. We have then constructed a canonical LR(1) translator for translation of infix expressions to postfix. `ptlib` – a LR(1) parsing framework had been implemented in C++ and this infix to postfix translation had been realized within said framework, which was then tested with various inputs for correctness of its output.

Creating a canonical LR(1) translator is a daunting task due to the massive amount of required theory and a vast amount of steps necessary to create all facilities for the parser and translator to work properly – which was further deepened by time constraints of the author. The biggest thing `ptlib` fails to do is to take this work is to unload this tedious parser construction work from the user and requires already constructed parse tables to be operating properly. Construction of these is algorithmized, and while it is not necessarily straightforward to implement, it is possible to do so. This would be the first and most improvement `ptlib` should receive.

Another possible improvement on the translating side of things is to make the definition of translations more powerful as of now it only allows outputting terminal strings on reductions. This can be improved into output grammar parse tree simulation, or actual in-memory parse tree construction and traversal.

The lexer can be improved by ditching sets of C++ regexes and constructing a single robust regular automaton from defined regular expressions for reading tokens from input.

The entire library can further be made more robust by ditching the loading of production rules *etc.* via means of formatted strings, and have it done by definition of specific object classes, so that C++ compilers could further optimize the parsing and translation by taking care of its definition during compile time.

Overall, there is even more to improve on this translation library, which was unfortunately not possible at the time of writing due to various time constraints weighing down the author.

I find it most unfortunate that I ran out of time to finish this thesis properly, but that is the consequence of grossly overestimating my capabilities and working speed – deciding to take on such a massive task of recreating an entire brand new bachelor's thesis in little over a month, especially since I work full time as well. Initially, the estimated changes and fixes to the old thesis looked managable, but the more I delved into the formal language translation again, the more I saw the fundamental error in the old thesis, and decided to change the entire idea of it.

Bibliography

[1] AHO, A. V.; LAM, M. S.; SETHI, R.; et al.: *Compilers, Principles, Techniques and Tools, Second Edition*. Addison-Wesley. 2006. ISBN 0-321-48681-1.

[2] AHO, A. V.; ULLMAN, J. D.: *Theory of Parsing, Translation and Compiling, Volume 1*. Prentice-Hall, Inc.. 1972. ISBN 0-13-914556-7.

[3] AHO, A. V.; ULLMAN, J. D.: *Theory of Parsing, Translation and Compiling, Volume 2*. Prentice-Hall, Inc.. 1973. ISBN 0-13-914564-8.

[4] HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D.: *Introduction to Automata Theory, Languages, and Computation, Second Edition*. Addison-Wesley. 2001. ISBN 0-201-44124-1.

[5] MEDUNA, A.: *Automata and Languages: Theory and Applications*. Springer. 2000. ISBN 978-1-4471-0501-5.

[6] MEDUNA, A.: *Elements of Compiler Design*. Auerbach Publications. 2008. ISBN 978-1-4200-6323-3.

[7] ROZENBERG, G.; SALOMAA, A.: *Handbook of Formal Languages, Volume 1*. Springer. 1997. ISBN 978-3-642-59136-5.

Appendix A

Attachments

Contents

This thesis includes a memory medium (an SD card) which holds this thesis in PDF format, its LaTeX sources, as well as the created translation framework `ptlib` together with its example usage program `if2pf`.

src contains `ptlib` source file, as well as the source file and Makefile of the `if2pf` program and its Linux executable.

doc contains HTML documentation for `ptlib` created by Doxygen.

thesis contains source files of this thesis, as well as the thesis in PDF format.

`if2pf` manual

The folder src contains four files in total: `translator.hh` which containst `ptlib` implementaton; `if2pf.cc` which is its source code; `Makefile` used to build the program; and finally `if2pf` which is a Linux executable built using GNU G++ compiler.

Installation

All that is required to build the program is to navigate into its folder in your CLI of choice and run command `make`. By invoking command `make clean` you delete the built executable. Alternatively, you could just invoke command `g++ -Wall -std=c++17 -O3 if2pf.cc -o if2pf`; the warning and optimization flags being optional. Please note that `if2pf` requires C++17 compatible compiler to build correctly. As `if2pf` does not use any platform-specific libraries or function calls, it is fully portable across all platforms that have C++17 compatible compilers. After the executable is created the program is ready for use.

Usage

The program is invoked by command `./if2pf` with one parameter - the string to translate.

Examples. These few examples show the functionality on three simple examples: first two providing a correct input, while the thirt once provides an incorrect input by adding a whitespace into it. The first row of each example shows invokation of the program; the second row is the input provided by the user and the third one is the ouput provided by `if2pf`.

```
> ./if2pf 'var1 + var2'
> var1 var2 +

> ./if2pf 'a1+(a2*a3)'
> a1 a2 a3 * +


> ./if2pf 'var2 ()'
> ERROR: parsing failed
```

There is not much more to it, as `if2pf` is quite a bare-bones tool, but provides great extensibility and moddability in return.