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 2017

Zadání bakalářské práce/19042/2016/xsabol01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Sabol Matúš**
Obor: Informační technologie
Téma: **Gramatická formalizace překladu a její implementace**
A Grammatical Formalization of Translation and Its Implementation
Kategorie: Teoretická informatika

Pokyny:

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

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Meduna Alexander, prof. RNDr., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstract

This thesis serves as an introduction to the topic of formal translations. It introduces the reader to essential theory and then uses knowledge of said theory to create a translator based on a particular translation. First part defines the essentials of the formal languages theory, which is a prerequisite for understanding the formal translation theory, whose essentials are explained after it. The second part describes the translation itself, firstly with a theoretical model, then with a computational model. Key implementation details are explained and briefly discussed. The proof-of-concept translator is successfully created and some of its possible improvements, as well as ways of expanding the formal translation topic are discussed.

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, ktorá je nutná pre pochopenie teórie formálneho prekladu, ktorej základne prvky sú vysvetlené následne. Druhá časť popisuje samotný preklad, najprv matematickým, následne výpočetným modelom. Sú spomenuté a vysvetlené kľúčové prvky implementácie. Prekladač, ktorý slúži ako overenie konceptu, je úspešne vytvorený. Na záver sú spomenuté niektoré možnosti vylepšenia samotného prekladaču, a taktiež aj možnosti ďalšieho rozvoja v tématike formálnych prekladov.

Keywords

finite automaton, formal language, formal grammar, pushdown automaton, translation grammar, syntax-directed translation schema, pushdown transducer, translator

Klíčová slova

konečný automat, formálny jazyk, formálna gramatika, zásobníkový automat, prekladová gramatika, syntaxovo riadené prekladové schéme, zásobníkový prevodník, prekladač

Reference

SABOL, Matúš. *A Grammatical Formalization of Translation and Its Implementation.* Brno, 2017. 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
May 14, 2017

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. I would also like to thank PhD. Křivka for providing me with study material..

Contents

Chapter 1

Introduction

The field of formal system theory, especially formal language theory had 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 – an ability of a great potential.

This thesis is concieved as an introduction to the formal translation theory and as such provides basic theoretical knowledge required for understanding it. The theory provided is used to construct a translator to show by example how to describe a translation theoretically, how to transform it into a computable model and how to implement it. Note that the translator is by not meant to be an industrial-level solution, but rather a proof-of-concept program that promotes superieor readability and simplicity over superior performance. It also shows some considerations concerning design and implementation choices for a translation.

Thesis is divided into two thematical parts:

The first part, Chapter 2, will introduce base definitions used throughout the whole formal language theory: language, grammar, and pushdown automaton as well as a definition of postfix notation. It further introduces means of formalizing translations, translational models and transformation between the two. It is presented in a rigorous manner with several examples make sure that the reader understands it.

The second part, Chapter 3, provides a *de facto* method for constructing a translator. It begins on a more theoretical level with formally defining a particular translation and transforming it into a computational model using structures and algorithm defined in the previous part. It then shifts into more practical tone when providing the reader with details of its actual implementation, such as programmatical structures and algorithms used to implement the translator based on the provided computational model.

The reader is expected to have fundamental knowledge of mathemathical sets: what is a set, element inclusion, subset, union, intersection, and Carthesian product. At least basic knowledge of finite automata is required as well: states, transitions, finishing states. These are the two cornerstones of formal languages theory that are mainly expanded upon. There is also a mention of algorithm time complexity and the Big-O notation in several 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 matter.

Definitions, examples and algorithms are numbered sequentially within chapters and they are concluded using symbol ■. Another practice in this thesis is emphasizing new or important terminology by *italicizing* it. Please make note that phrases of latin origin, such as *et cetera* or *verbatim* are italicized as well. This is not meant as emphasis, but it is a common practice to italicize such phrases in written text.

Chapter 2

Theory

This chapter is dedicated to introducing all the theory necessary for understanding the topic of this thesis. It will cover basic theory elements of formal systems and some of the theory needed to understand translations.

2.1 Preliminaries

In this section, we will be going over some of the essentials required to understand the subject of this thesis. While not at all difficult, a proper understanding of these is absolutely crucial for proper understanding not only translations, but formal systems theory as a whole.
Definitions are taken from [3] and [4]. Further reading on this topic can also be concluded in [5], chapters 1 to 3 and [2], chapters 1 to 7.

There will be mentions of *reverse Polish* (will be referred to as *postfix* later on) notation in various places throughout this thesis, and as such a definition is required. This definition is taken *verbatim* from [3].

Definition 2.1. Let Σ be an alphabet, whose symbols denote operands. The *reverse Polish* expressions are defined recursively as follows:

1. If a is an infix expression and $a \in \Sigma$, then a is also the reverse Polish expression of a.
2. If U and V are infix expressions denoted by reverse Polish expressions X and Y, respectively, and o is an operator such that $o \in \{+, -, *, /\}$, then XYo is the reverse Polish expression denoting UoV.
3. If (U) is an infix expression, where U is denoted by the reverse Polish expression X, then X is the reverse Polish expression denoting (U).

■

2.1.1 Alphabets

The very building blocks of formal systems are alphabets. They contain all symbols that the system is „allowed“ to use. We will now define some key terms regarding alphabets.

Definition 2.2. We define an *alphabet* Σ as a finite non-empty set, whose members are called *symbols*.

■

Definition 2.3. We define a *string* w over Σ as:

$$w = a_1 a_2 a_3 ... a_n; a_i \in \Sigma, i \in \mathbb{N}$$

, a sequence of symbols from Σ. A special string contaning zero symbols is called an *empty string* and is denoted by ε. Such string is still a string over Σ.

■

Definition 2.4. We denote Σ^* a set of all strings over Σ. We define Σ^+ as $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

■

2.1.2 Languages

Languages are one step above alphabets. They are sets of strings over an alphabet, both finite and infinite, defined as follows:

Definition 2.5. We define a *language* L over Σ^* as a set $L \subseteq \Sigma^*$. If L is a finite set, we call it a *finite language*, otherwise it's an *infinite language*.

■

We will now show an example of alphabet and a language over said alphabet.

Example 2.6. Let there be an alphabet $\Sigma = \{a, b\}$. Let us define a language $L = \{a^n b^n; n \in \mathbb{N}\}$. 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, it is an alphabet over Σ.

■

2.1.3 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.

Definition 2.7. A *context-free grammar* is a 4-tuple $G = (N, T, R, S)$, whose elements are defined as follows:

N is a finite set of *non-terminal* symbols

T is a finite set of *terminal* symbols

R is a finite set of *productions rules* in form of $A \to w; A \in N; w \in (N \cup T)^*$

S is the starting non-terminal, $S \in N$

■

It should be noted that while there also exist *context-sensitive* grammars for whose production rules holds true that $A \in (N \cup T)^*$. However, those are of no interest for us, and as such we will only consider context-free grammars for the purposes of this thesis.

The way grammars work is by transforming the output, that starts as the non-terminal S using production rules from R, until no non-terminals remain in the output. This means that in every *derivation step* they choose a non-terminal from the current output, and apply any applicable rule (any rule whose right side is equal to the chosen non-terminal) by replacing the non-terminal with the sequence of terminals on the left side of the production rule. We will now properly define these actions:

Definition 2.8. A *derivation step* is a transition $x \xrightarrow{A \to w} y$ using the production rule $A \to w$, where

$$x = x_1 A x_2; A \in N; x_1, x_2 \in (N \cup T)^*$$

and

$$y = x_1 w x_2; x_1, x_2, w \in (N \cup T)^*$$

The production rules have often times assigned their ordinal number. In such case, we can also write $x \xrightarrow{n} y$, with n being the ordinal number of said production rule.

■

While by strict definition the order of non-terminals chosen is irrelevant, in praxis we always use one of two approaches:

1. Always picking the left-most non-terminal, this results in a *left-most derivation*
2. Always picking the right-most non-terminal, this results in a *right-most derivation*

For the purposes of this thesis, we will always consider a derivation to be left-most unless specifically said otherwise.

Definition 2.9. We define a *k-step* derivation $S \to^{=k} w$ a derivation, that applies exactly k production rules to the output, to produce a valid output string w.
We define a derivation $S \to^* w$, as a k-step derivation, where $k \geq 0$.
We define a derivation $S \to^+ w$, as a k-step derivation, where $k \geq 1$.

■

Definition 2.10. We define w a *string generated by grammar G*, denoted $S \Rightarrow^G w$, if and only if there exists such production in G, for which

$$S \to^* w; w \in T^*$$

holds true.

■

Definition 2.11. A language $L(G)$, *generated by grammar* G, is a language

$$L(G) = \{w | w \in T^*, S \Rightarrow^G w\}$$

, which is a subset of all strings generated (accepted) by grammar G.
■

With all the important terminology defined, we will now show an example of a context-free grammar defining a language $L(G) = \{a^n b^n | n \in \mathbb{N}\}$.

Example 2.12. We have a grammar $G = (\{S\}, \{a, b\}, R, S)$, with R defined as:

$S \to aSb$

$S \to ab$

We will now show that this grammar really generates language $L(G) = \{a^n b^n | n \in \mathbb{N}\}$. We start with the non-terminal S, which we can replace either with aSb or ab, according to the production rules for S. If we choose the latter, we have run out of non-terminals, and have generated the string ab. Otherwise, we are left with the string aSb and we can keep applying productions $S \to aSb$ for $n - 2$ more times, until we are left with $a^{n-1}Sb^{n-1}$. At this point, we use the production $A \to ab$, which transforms the string into $a^n b^n$. This show, that the language generated by this grammar is in fact $L(G) = \{a^n b^n | n \in \mathbb{N}\}$.

■

2.1.4 Pushdown automata

A pushdown automaton is a finite automaton extended by an infinite stack, which is in formal system theory referred to as a *pushdown*. A pushdown automaton uses not only the input, but also the top of the pushdown to decide the next transition.

Definition 2.13. A *pushdown automaton* is a 7-tuple

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

, whose elements are defined as follows:

Q is a finite set of states

Σ is an *input alphabet*

Γ is a *stack alphabet*

δ is a finite set of rules, mappings from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

q_0 is the initial state, $q_0 \in Q$

Z_0 is the initial stack symbol, $Z_0 \in \Gamma$

F is a finite set of terminal states, $F \subseteq Q$

■

Each transition, in addition to changing the state of the automaton, also pushes a string over the stack alphabet Γ.
We use configurations to express a state of a pushdown automaton. This is most commonly used to express its current state, or as a means to define transitions from δ.

Definition 2.14. We define a *configuration* of a pushdown automaton $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ a 3-tuple

$$(q, w, \gamma)$$

, where $q \in Q$ is the current state, $w \in \Sigma^*$ is the current input string, and $\gamma \in \Gamma^*$ is the current pushdown.

We denote a pushdown automaton *transition* defined by a rule

$$(q, x, \alpha) \to (r, \beta)$$

, where $q, r \in Q$, are automaton states, $x \in \Sigma$ is the current input character, α is the current pushdown top, and $\beta \in \Gamma^*$ is a string over the stack alphabet as a mapping

$$(q, xw, \alpha\gamma) \vdash (r, w, \beta\gamma)$$

and in definitions we denote it as

$$\delta(q, x, \alpha) \ni (r, \beta)$$

.

By the example set in 2.9, we define a transition

$$(q, w, \gamma) \vdash^* (r, w', \gamma')$$

as a k-step transition, where $k \geq 0$.

■

As pushdown automata are finite automata, there is a need to decide when its operation is correctly finished; string that leads to the automaton finishing correctly is also called a *string accepted by the automaton*. This can be determined using one of 3 methods:

- in finishing state and empty input string
- in finishing state and empty pushdown
- in finishing state, empty input string and pushdown

In the first two methods the state of pushdown and input string, respectively, is not relevant. If the automaton cannot apply any of the rules before the chosen condition is met, the automaton fails meaning that the input string is not accepted by the automaton. It should be noted that in practice, it is not uncommon that the „in finishing state“ condition is disregarded. Often time it is formalized as $F = \emptyset$.

As an example of a pushdown automaton, we will construct one that accepts the same language as the grammar in example 2.12.

Example 2.15. Denote a pushdown automaton

$$P = (\{q\}, \{a, b\}, \{S, E, b\}, \delta, q, S, \{q\})$$

that accepts strings by empty input string and pushdown, where rules δ are defined as:

$$\begin{aligned}
\delta(q, \varepsilon, S) &= \{(q, \varepsilon)\} \\
\delta(q, a, S) &= \{(q, E)\} \\
\delta(q, a, E) &= \{(q, Eb)\} \\
\delta(q, b, E) &= \{(q, \varepsilon)\} \\
\delta(q, b, b) &= \{(q, \varepsilon)\}
\end{aligned}$$

We can see right away from the first rule that this pushdown automaton will accept empty strings. In the case that the input string is a string in form $a^n b^n$ we first apply transition $(q, a, S) \to (q, E)$ followed by $n - 1$ transitions $(q, a, E) \to (q, Eb)$, where n is the total number of a symbols in the input. At this moment, the input and output look like this: (b^n, Eb^{n-1}). On the first occurence of symbol b we can only use the transition $(q, b, E) \to (q, \varepsilon)$ which leaves us with state (b^{n-1}, b^{n-1}). At this point, we apply the transition $(q, b, b) \to (q, \varepsilon)$ for another $n - 1$ times, until we are left with empty input and empty pushdown, signifying that the string $a^n b^n$ is really accepted by the pusdown automaton. If the input string was in the form $a^n b^m; n \neq m$, then there would be an imbalance between stack symbols which would lead either to non-empty input or non-empty pushdown after we can no longer apply rules, which means that no such string can be accepted.

■

Keep in mind that this is not the only pushdown automaton that accepts such language. In fact, there is an infinite amount of such automata, but most of them do not concern us for purposes of this thesis.

2.2 Formalisms for translations

Having covered the basic theory for understanding formal systems, we can now move on to defining theory for translation itself. The preliminary text and definitions were taken from [1], sections 3.1.2 and 3.1.4.

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.

2.2.1 Syntax-Directed Translation Schemata

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 for purposes of this thesis.

Definition 2.16. A *syntax-directed translation shcema* (SDTS for short) is a 5-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 non-terminals in α
5. S is the starting non-terminal, $S \in N$

■

As we can see from the definition, a SDTS structurally almost identical to a regular grammar, with the exception that every rule now has two outputs - first one representing the input language grammar and the second one representing the ouput language grammar.

For further formal needs, we will also define syntax-directed translation, input, and output grammar.

Definition 2.17. Define a SDTS $T = (N, \Sigma, \Delta, R, S)$. The grammar

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

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

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

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

■

Let us explain these definitions on a simple example:

Example 2.18. Let $T = (\{E\}, \{a, +, *\}, \{a, +, *\}, R, E)$ be a SDTS. R is defined as follows:

$E \to E + E, EE+$

$E \to E * E, EE*$

$E \to a, a$

The input grammar G_i of this SDTS is defined as:

$$G_i = (\{E\}, \{a, +, *\}, P, E)$$

where P contains rules:

$E \to E + E$

$E \to E * E$

$E \to a$

The output grammar G_o of this SDTS is defined as:

$$G_o = (\{E\}, \{a, +, *\}, P', E)$$

where P' contains rules:

$E \to EE+$

$E \to EE*$

$E \to a$

For example, the translation of $a * a + a$ according to this SDTS would be $aa * a+$.

■

From these facts, we can tell that this particular SDTS translates an expression in infix notation into an expression in postfix notation.

2.2.2 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.

Definition 2.19. 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 2.20. 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

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

As mentioned after definition 2.14, the $q \in F$ is often times disregarded in practice.

Having introduced both syntax-directed translation schemata used to formally describe translations and pushdown transducers to formalize translation implementation, it would be beneficial for us to have an algorithm for constructing a PDT from a SDTS. One such algorithm is provided here:

Algorithm 2.21. Let alphabet Δ' be defined as:

$$a' \in \Delta' \iff a \in \Delta$$

Let homomorphism h be defined by

$$\forall a \in \Delta : h(a) = a'$$

Let us define a PDT $P = (\{q\}, \Sigma, N \cup \Sigma \cup \Delta', \Delta, \delta, q, S, \emptyset)$, where δ is defined as follows:

1. $\forall A \to x_0 B_1 x_1 ... B_k x_k, y_0 B_1 y_1 ... B_k y_k \in R, k > 0 : \delta(q, \varepsilon, A) \ni (q, x_0 y'_0 B_1 x_1 y'_1 ... B_k x_k y'_k, \varepsilon)$, $y'_i = h(y_i), 0 \leq i \leq k$
2. $\forall a \in \Sigma : \delta(q, a, a) = \{(q, \varepsilon, \varepsilon)\}$
3. $\forall a \in \Delta : \delta(q, \varepsilon, a') = \{(q, \varepsilon, a)\}$

■

Proof of this can be found in [1], Lemma 3.2.

Let us explain this algorithm on an example where we construct PDT that implements SDTS from example 2.18 using algorithm 2.21:

Example 2.22. Let $P = (\{q\}, \{a, +, *\}, \{E, a, a', +, +', *, *'\}, \{a, +, *\}, \delta, q, E, \emptyset)$ be a PDT. We define δ as:

$$\delta(q, \varepsilon, E) = \{(q, E + E+', \varepsilon), (q, E * E*', \varepsilon), (q, aa', \varepsilon)\}$$

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

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

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

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

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

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

Let us look at the steps done by the PDT on the same input as in example 2.18:

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

As we see, this translator produced output $aa*a+$ for input $a*a+a$, the same as the aforementioned STDS from example 2.18.

■

Chapter 3

Implementation

The knowledge of theory from the previous chapter was utilized to create a tool called `if2pf` using C++ programming language. Standing for „infix to postfix“, that does exactly what it says on the tin – it transforms an expression in infix notation into an equivalent expression in postfix notation.

This chapter is dedicated to description of the tool and explanation of the implementation decisions made in its making.

3.1 Underlying SDTS and PDT

The very base stone of any translation is the SDTS which defines the language that's being translated from and the language that's being translated to.

The STDS that I ultimately settled for after some fixes, tweaks and changes was:

$$T = (\{E\}, \{a, +, -, *, /, (,)\}, \{a, +, -, *, /\}, R, E)$$

, with rules defined as:

1. $E \to E + E, EE+$
2. $E \to E - E, EE-$
3. $E \to E * E, EE*$
4. $E \to E/E, EE/$
5. $E \to (E), E$
6. $E \to a, a$

Example 3.1. This example will demonstrate that this SDTS does in fact describe a translation from infix expressions to postfix.
(The input/output string pairs have been encapsulated in brackets for better readability of this example.)

$$[E,\ E] \xrightarrow{3} [E * E,\ EE*] \xrightarrow{7} [a * E,\ aE*] \xrightarrow{6} [a * (E),\ aE*] \xrightarrow{1}$$
$$\xrightarrow{1} [a * (E + E),\ aEE + *] \xrightarrow{7} [a * (b + E),\ abE + *] \xrightarrow{7} [a * (b + c),\ abc + *]$$

■

From the final output string we can see that $bc+$ will be computed first, followed by $ad*$, where $d = bc+$. This is effectively the same as $a * (b + c)$, meaning that the output string is in fact a translation of the input string

3.1.1 Input grammar

As we are translating from postfix expressions, our input grammar G_i must generate infix expressions. One of such grammars, and probably the simplest one of them is:

$$G = (\{E\}, \{a, +, *\}, R, E)$$

, where R contains rules as follows:

1. $E \to E + E$
2. $E \to E * E$
3. $E \to a$

This grammar is the most basic grammar that only produces expressions like $a+a$, $a*a$ and combinations of thereof. You can see it cannot do subtractions, divisions or paretheses. As the grammar is very simple, expanding it to accomodate for these requirements is very straightforward.
Note that while we are formally using the rule $E \to a$, but the actual implementation uses rules from $E \to a$ to $E \to z$, but for the sake of not writing out 25 more rules, we will only write the rule $E \to a$, which will also include the other 25 rules for the rest of variables.

With all this said, we can define our final input grammar G_i:

$$G_i = (\{E\}, \{a, +, -, *, /, (,)\}, R_i, E)$$

with the rules in R_i defined as

1. $E \to E + E$
2. $E \to E - E$
3. $E \to E * E$
4. $E \to E/E$
5. $E \to (E)$
6. $E \to a$

We can see that the grammar is *left-recursive*, which means that the leftmost symbol from w in $A \to w$ is the same as A. While this trait is often times undesirable, especially in grammars used to construct compilers, it doesn't concern us in the case of translation from infix to postfix.

We will show on an example that this grammar does, in fact, produce infix expressions:

Example 3.2. We begin with the non-terminal E and we apply productions, choosing the leftmost non-terminal every time:

$$E \xrightarrow{3} E * E \xrightarrow{6} a * E \xrightarrow{5} a * (E) \xrightarrow{1} a * (E + E) \xrightarrow{6} a * (b + E) \xrightarrow{6} a * (b + c)$$

We can clearly see, that this expression is in infix form. Any other expression would be constructed in a similar fashion and would also be in infix form. Note that this is in now way a rigorous proof, but a demostration of the grammar. Also note that we also used letters b and c to denote variables – this is for proper identification of said variables, as we will later on construct a postfix equivalent of this expression.

■

3.1.2 Output grammar

Having described the input grammar, the output grammar is very similar except as it needs to generate postfix expressions, the operator have been moved from between non-terminals to behind of them. Also one big change is that postfix expressions do not have the need for parentheses, which makes them excellent for computer processing. These changes turn the grammar G_i into grammar G_o as such:

$$G_o = (\{E\}, \{a, +, -, *, /\}, R_o, E)$$

with the rules in R_o defined as

1. $E \to EE+$
2. $E \to EE-$
3. $E \to EE*$
4. $E \to EE/$
5. $E \to a$

As with the input grammar, we will now show on an example that this grammar ideed produces postfix expressions. We will create a postfix equivalent of the expression created in example 3.2

Example 3.3. We begin with the non-terminal E and we apply productions, choosing the leftmost non-terminal every time:

$$E \xrightarrow{3} EE* \xrightarrow{5} aE* \xrightarrow{1} aEE+* \xrightarrow{5} abE+* \xrightarrow{5} abc+*$$

■

We can see that because of the way the expression is constructed, $bc+$ will be computed first, followed by $ad*$, where $d = bc+$. This is effectively the same as $a * (b + c)$.

3.1.3 Considerations about used grammars

While in construction of compilers we often strive for deterministic *LL(1) grammars* (a $LL(1)$ grammar is a grammar that reads input from left to right, produces the leftmost derivation with lookup of 1 character - that means it only needs the current symbol under the reading head to decide the next derivation), you could determine that the grammars G_i and G_o we use are neither deterministic, nor $LL(1)$. and I'm actually quite sure they're not even $LL(k)$ grammars.

Even though the chosen grammars are far from what would be considered ideal when creating compilers, they work just fine in translations. In fact, I found it impossible to construct a working SDTS using $LL(1)$ grammars for this particular trasnlation. This is mainly due to the nature of such translation: the position, and possibly even order of operators within the string is changed due to how operator priorities are handled. This holds especially true for expressions in parentheses, as they can move operators with lesser priority before operators with greater priority, which would otherwise come before said lower priority operators.

With all this said, the method I devised to make this work despite all the complications will be covered in further part of this paper dedicated to implementation, as it has nothing to do with the theoretical part of the program.

I also took this opportunity to show that one doens not require grammars with perfect attributes to achieve good results.

3.1.4 The underlying PDT

As we showed earlier, our chosen STDS describes the translation we seek, this means all that is left now is to construct a PDT that is equivalent to it.
The PDT P constructed from the SDTS for this translation using algorithm 2.21:

$$P = (\{q\}, \{a, +, -, *, /, (,)\}, \{E, a, a', +, +', -, -', *, *', /, /', (,)\},$$
$$\{a, +.-, *, /\}, \delta, q, E, \emptyset)$$

with transition δ defined as:

$$\begin{aligned}
\delta(q, \varepsilon, E) = \{ & \\
& (q, E + E+', \varepsilon), \\
& (q, E - E-', \varepsilon), \\
& (q, E * E*', \varepsilon), \\
& (q, E/E/', \varepsilon), \\
& (q, (E), \varepsilon), \\
& (q, aa', \varepsilon) \\
\} & \\
\delta(q, a, a) = \{(q, \varepsilon, \varepsilon)\} & \\
\delta(q, +, +) = \{(q, \varepsilon, \varepsilon)\} & \\
\delta(q, *, *) = \{(q, \varepsilon, \varepsilon)\} & \\
\delta(q, (, () = \{(q, \varepsilon, \varepsilon)\} & \\
\delta(q,),)) = \{(q, \varepsilon, \varepsilon)\} & \\
\delta(q, \varepsilon, +') = \{(q, \varepsilon, +)\} & \\
\delta(q, \varepsilon, *') = \{(q, \varepsilon, *)\} & \\
\delta(q, \varepsilon, a') = \{(q, \varepsilon, a)\} &
\end{aligned}$$

As was said before, the transitions that contain the terminal a in any way or form stand for all such transitions from a to z, but only the a ones are shown for the sake of brevity. We are now going to show, for the sake of completeness, that this PDT is equivalent to the aforementioned STDS.

Example 3.4.

$$
\begin{aligned}
(q, a*(b+c), E, \varepsilon) &\vdash (q, && a*(b+c), && E*E*', && \varepsilon) \\
&\vdash (q, && a*(b+c), && aa'*E*', && \varepsilon) \\
&\vdash (q, && *(b+c), && a'*E*', && \varepsilon) \\
&\vdash (q, && *(b+c), && *E*', && a) \\
&\vdash (q, && (b+c), && E*', && a) \\
&\vdash (q, && (b+c), && (E)*', && a) \\
&\vdash (q, && b+c), && E)*', && a) \\
&\vdash (q, && b+c), && E+E+')*', && a) \\
&\vdash (q, && b+c), && bb'+E+')*', && a) \\
&\vdash (q, && +c), && b'+E+')*', && a) \\
&\vdash (q, && +c), && +E+')*', && ab) \\
&\vdash (q, && c), && E+')*', && ab) \\
&\vdash (q, && c), && cc'+')*', && ab) \\
&\vdash (q, &&), && c'+')*', && ab) \\
&\vdash (q, &&), && +')*', && abc) \\
&\vdash (q, &&), &&)*', && abc+) \\
&\vdash (q, && \varepsilon, && *', && abc+) \\
&\vdash (q, && \varepsilon, && \varepsilon, && abc+*)
\end{aligned}
$$

■

As we can see, the result is the same as in example 3.1, meaning that this PDT is really equivalent with the infix to postfix SDTS defined earlier in this chapter, which means that implementing this PDT will implement a translation of expressions from infix to postfix notations. This observation concludes this chapter, as now we are ready to implement this transducer.

3.2 Translator implementation

Having figured the PDT for the translation, the only thing left is to implement it; however such implementation is generally no menial task. If your PDT is created from a $LL(1)$ SDTS, then it is of course easy: you can implement it the same way you would implement a recursive descent complier, except you would include pushdown and output manipulation.

As was stated in section 3.1.3, our chosen SDTS is not created using $LL(1)$ grammars, which means that this solution is not usable. The fact that we are dealing with a formal language that factors precedence is also one of the reasons that the recursive descent solution may not have worked anyway.

3.2.1 Input preparation

To deal with this problem, an input preparation algorithm, that reads the input expression and returns a list of operators sorted by their priorities in *ascending* order was devised. This list is used to choose transitions that produce a correct output.

Example 3.5.
An input string „$a * (b + c) * d + e$“ would be transformed to list „$+ * *(+$“.
An input string „$a + b - c$“ would be transformed to list „$-+$“.

■

You can see that the closing parenthesis is omitted as it only caused problems with the implementation and otherwise was not a factor in any shape or form. You can also see that the order of operators in the second example is mirrored, this is because we are using a pushdown (a stack), which is a LIFO container in nature. If the string „$a + b - c$“ was transformed into „$+-$“ (that is, without mirroring the order of equiprecedent operators), it would lead to an incorrect translation. We will explain why in Algorithm section. Also note that the original input string is conserved.

Algorithm 3.6.
Define a set of operators $O = \{+, -, *, /, (,)\}$.
Define an input string w.
Define a stack S that holds positions of starting parentheses.
Define a list P of pairs of indices.
Define a list L of sorted operators.

For all characters in w:

1. read $i - th$ character a_i from w
2. if $a_i = ($ then push index of k into S, where $k = |L| + 1$
3. if $a_i =)$ then pop index l from S and put a pair (l, k) into P
4. if $a_i \in O$ then add a_i to L, go back to step 1.

In the end, we also add a pair $(1, |L|)$ into P.

After this is done, we are left with a list of operators that is not sorted yet. The next part will sort them by precedence.

For all pairs $(j, i) \in P$:

1. sort all operators in interval from j to i in L by precedence in ascending order, treat expressions in parentheses as a single operator
2. mirror sequences of equiprecedent operators, leaving out expressions in parentheses
3. remove the closing parenthesis

After all these steps, the list L is in fact a list of operators sorted ascending by their precedence, as showed in example 3.5.

■

To make things clear, we will show an example of how such list is constructed from an output.

Example 3.7.

Define input string $w = a*(b+c*(d-e+f))/g$. After the first part of the algorithm is done, $w = *(+*(-+))/$. We have 3 intervals defined in P, those are $(1, 10)$, $(2, 9)$, and $(5, 8)$.

After applying the second part on the interval $(1, 10)$, we are left with $w = /*(+*(-+))$.

Next is the interval $(2, 9)$ after which $w = /*(+*(-+)$.

After the algorithm is applied for the last interval, we are left with $w = /*(+*(+-$.

■

3.2.2 Structures

An important part of implementing a translation, or any formal system for that matter, is representation of transitions. For example, in the case of aforementioned $LL(1)$ grammars transitions are represented by methods that form a recursive descent parser. Unfortunately, we cannot use such approach for our particular translator. What we can use is storing the transitions in a structure similar to the mathemathical representation of transition. While such approach may be rather naïve and most likely not very time-efficient compared to other possible means of representing transitions, it is also the simplest and most readable. Since our goal is not to create a highly efficient industrial solution but rather to show a proof of concept I decided to use exactly such solution.

As we can see in definition 2.19 a PDT transition is effectively a mapping $(q, x, \alpha) \to (r, \beta, y)$ where q, r are states, x is an input character, y is an output string, α is a character on pushdown top and β is the string that is pushed into the pushdown. Representing such transition by a single tuple would be advantageous for programmatical purposes. Since the PDT we use only has one state, this implies that $q = r$ and hence the state is not a factor. The final representation chosen was (x, α, y, β, h), as the elements are ordered in a way where the input elements come before the output ones. There is also an extra element h called *hint* which is used in transitions that would otherwise be non-deterministic. If a transition is deterministic, then $h = \varepsilon$. The C++ class that achieves this is:

```
class PDTtransition {
    PDTsymbol         string_input;
    PDTsymbol         stack_input;
    PDTsymbolString   string_output;
    PDTsymbolString   stack_output;
    PDTsymbol         hint;
};
```

It is apparent that this class truly represents the tuple (x, α, y, β, h). Note that all class methods are omitted from this listing and will be omitted from any other code listing for sake of brevity.

You can see that that the `PDTtransition` class is composed of 5 members of 2 types: class `PDTsymbol` that represents a single symbol of an alphabet and class `PDTsymbolString` which is a string of such symbols.

```
class PDTsymbol {
    std::string symbol;
};

class PDTsymbolString {
    std::vector<PDTsymbol> symbolString;
};
```

The class `PDTsymbol` was chosen to represent a single symbol with a C++ string because this way you could represent symbols like a' literally by string `"a'"`, without any need to look for substitute characters. This is, of coure, not an optimal solution, as every string operation takes at least $O(n)$ time. It is, however, the most readable, and readability is one of the goals of this thesis.

3.2.3 Translation algorithm

Having defined our structures and a way to prepare the input to help us correctly translate it, we can now move on to the algorithm that performs the actual translation. Note that the implemented PDT accepts strings by empty pushdown.

Until now, we have been only considering the syntax of both input and output language for the translation grammar. But since a program cannot, unlike humans, choose correct transitions in correct order to produce a correct translation by means of logical thinking, we need to start looking at the semantics as well. Let us expand on definition 2.1. If we had an infix expression Ao_1Bo_2C, and precedence of o_1 and o_2 was the same, then the postfix equivalent would be ABo_1Co_2; this means that operations would be calculated in the order they are written. If, however, the precedence of o_2 was greater than that of o_1, it would mean that operation Bo_2C would precede Ao_1X; this would be translated as $ABCo_2o_1$. This means that more precedent operators need to appear before less precedent ones in postfix notation.

Since we are implementing a PDT, which contains a pushdown – a LIFO structure – we need to push the less precedent operators before the more precedent ones, which would result in more precedent operators being translated before the less precedent ones, which is exactly what we want. This is the reason we created a list of operators sorted by their precedence using alorithm 3.6; so that it can help us deterministically choose the tranitions to apply to create a correct translation.

Algorithm 3.8.
Define `current` currently first symbol of the input string. If the input string is empty then `current = ""`.
Define `peek` a symbol right after `current`. If the input string length is 1 and no symbol after `current` can be acquired then `peek = ""`.
Define pushdown P, whose initial element is E.
Define L list of operators sorted by precedence.

Define function $f_p(x)$ a priority function, such that $f_p(a) = f_p(b) = ... = f_p(z) < f_p(+) = f_p(-) < f_p(*) = f_p(/) < f_p('(') = f_p(')') < p(the\ rest)$.
Define hint h.

While P is not empty:

1. read `current` and `peek` from input, move in input by one symbol
2. acquire top symbol `stackTop` from P
3. if `stackTop != E` declare $h = \varepsilon$ and jump to step 6.
4. if $f_p(\texttt{current}) \geq f_p(first\ element\ of\ L)$ or $f_p(\texttt{peek}) \geq f_p(first\ element\ of\ L)$, then

   4a. if $|L| > 0$, then $h = first\ element\ of\ L$ and remove first element of L from L

   4b. else $h = \texttt{current}$
5. otherwise $h = \texttt{current}$
6. find transition $(\texttt{current}, \texttt{stackTop}, y, \beta, h)$, if no transition is found then end with error
7. push β to P, write y to output

■

The most important part of the algorihm is step 4. where it checks whether the input (since one of `current` or `peek` will *always* be an operator in a correctly written expression) includes operators with lesser or equal priority than the operator currently on the beggining of L. This ensures that the operators in output will be placed in the correct order, as in infix notation the operations with lower priority occur after the operations with higher priority when reading from left to right. This will in effect cause the translator to first use production rules that create a „base“ for higher priority operations and then repeats the process for each „operand“ (either a variable or an expression in parentheses) in the base expression. Only by doing this we can achieve the correct order of output operands.

Another fact that should be mentioned that due to the nature of `PDTtoken` using a C++ string, the only reasonable search method in step 6. is linear seach - meaning that every time we search for the transition we iterate the whole list of transitions, comparing strings on every iteration. This gives this search algorithm a terrible worst-case time complexity $O(n.m)$, where n is the number of transitions and m is the maximum length of a string, whose comparison takes $O(m)$. While this is the theoretical time complexity, the effective time complexity would be $O(1)$, although with a very large constant, as n stays constant (89) during the translation and m is always ≤ 2 in this particular translator.

Example 3.9.
Let us have an input string $w = a * (b + c)$. By applying algorithm 3.6 to it, we now have a sorted list of operators $L = *(+$. Pushdown P is starting with only symbol E. Translation

process will be shown as a sequence of rows with each row showing the state of the translator before applying a transition that would lead into the row below it.

```
input=a*(b+c) | current=a | peek=* | L=*(+ | P=E          | hint=* | output="
input=a*(b+c) | current=a | peek=* | L=(+  | P=E*E*'      | hint=a | output="
input=a*(b+c) | current=a | peek=* | L=(+  | P=aa'*E*'    | hint=" | output="
input=*(b+c)  | current=* | peek=( | L=(+  | P=a'*E*'     | hint=" | output="
input=*(b+c)  | current=* | peek=( | L=(+  | P=*E*'       | hint=" | output=a
input=(b+c)   | current=( | peek=b | L=(+  | P=E*'        | hint=( | output=a
input=(b+c)   | current=( | peek=b | L=+   | P=(E)*'      | hint=" | output=a
input=b+c)    | current=b | peek=+ | L=+   | P=E)*'       | hint=+ | output=a
input=b+c)    | current=b | peek=+ | L="   | P=E+E+')*'   | hint=b | output=a
input=b+c)    | current=b | peek=+ | L="   | P=bb'+E+')*' | hint=" | output=a
input=+c)     | current=+ | peek=c | L="   | P=b'+E+')*'  | hint=" | output=a
input=+c)     | current=+ | peek=c | L="   | P=+E+')*'    | hint=" | output=ab
input=c)      | current=c | peek=) | L="   | P=E+')*'     | hint=c | output=ab
input=c)      | current=c | peek=) | L="   | P=cc'+')*'   | hint=" | output=ab
input=)       | current=) | peek=" | L="   | P=c'+')*'    | hint=" | output=ab
input=)       | current=) | peek=" | L="   | P=+')*'      | hint=" | output=abc
input=)       | current=) | peek=" | L="   | P=)*'      | hint=" | output=abc+
input="       | current=" | peek=" | L="   | P=*'       | hint=" | output=abc+
input="       | current=" | peek=" | L="   | P="        | hint=" | output=abc+*
```

■

Note that the last three rows of the sequence are misaligned to the rest on purpose so that the ouput could fit well inside the page.

If you compare the sequence of colums `input`, `P`, and `output` with example 3.4, you can see that they have the same values through the whole sequence of transitions. This further shows that the implemented translator is in fact equivalent to the PDT defined in section 3.1.4, and hence equivalent to the SDTS defined in 3.1, and hence exactly the translator we have been seeking to implement.

Chapter 4

Conclusion

By creating a software translator from a formal description of a translation I have successfully achieved the main goal of this thesis. The existence of such translator provides not only a proof of concept of easily implementing $LL(k)$-based translations for large or variable k (maybe even non-LL-based translations); but even more importantly an easily understandable and expandable example for a translator.

As was mentioned before, the `if2pf` in its current state is rather poorly optimized for performance, which provides much opportunity for optimization even in its base form.

One possible improvement would be not to use `std::string` for encoding symbols, but rather representing each symbol as an arbitrary combination of bits stored in a primitive type. Each combination's meaning would then be represented by a lookup table pair *bit_combination* $\leftrightarrow$ *symbol*. This would allow for a reasonable amount of represented symbols and increased speed as it would no longer be necessary to compare strings. The memory footprint of such solution would probably be lower as well, as there would be no need to store a string object, but only a few bytes for each symbol. In case of great number of symbols, this advantage might be partially offset by a large lookup table that would be needed.

Other such improvement would be to merge stack top symbol and hint into a single field, effectively creating a unique key for each transition. The list of such "compacted" transitions could then be sorted by the stack-hint field, which would make it possible to use binary search, which would bring down worst-case time complexity of finding a transition during translation from $O(n.m)$ to $O(log\ n)$ where n is number of transitions.

Expanding on the concept of formal translation is the logical next step. I see that there are two essential ways to expand in this area:

1. "into the depth": by describing translations using more complex mathemathical models, such and transforming them into more complex computational models that would allow even more complex languages to be translated with similar amounts of effort; or

2. "into the width": taking inspiration from [6], chapter 4, utilizing grammar systems in translation and creating *translation systems* – systems of several simpler translation grammars working together achieving the effect of a more complex translation grammar.

Bibliography

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

[2] 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.

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

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

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

[6] ROZENBERG, G.; SALOMAA, A.: *Handbook of Formal Languages, Volume 2.* Springer. 1997. ISBN 978-3-662-07675-0.

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 translator `if2pf`. It will contain two folders as follows:

if2pf contains source files and Makefile of the `if2pf` program, as well as its Windows and Linux executables

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

`if2pf` manual

The folder if2pf contains four files in total: `if2pf.cc` which is its source code; `Makefile` used to build the program; `if2pf_win.exe` which is a Windows executable built using MinGW G++ compiler on Windows 10; 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++11 -O2 if2pf.cc -o if2pf`; the warning and optimization flags being optional. Please note that `if2pf` requires C++11 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++11 compatible compilers. After the executable is created the program is ready for use.

Usage

The program is invoked by command `./if2pf` with no parameters. User then provides an input string in infix form that meets the following requirements:

1. only operands allowed are in form of „variables“ represented by lowecase letters of English language
2. there is no whitespace between any of the operands or operators

Otherwise the input string is equivalent to an expression in infix notation that uses only operators $+$, $-$, $*$, / and parentheses that can be nested. The output of `if2pf` is in the same format as its input.

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
> a+b
> ab+

> ./if2pf
> a+b*c
> abc*+

> ./if2pf
> a+ b
> ERROR MESSAGE
```

■

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