BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

TRANSLATION GRAMMARS: PROPERTIES AND APPLICATIONS

PŘEKLADOVÉ GRAMATIKY: VLASTNOSTI A APLIKACE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **RADEK VÍT**
AUTOR PRÁCE

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

BRNO 2017

Zadání bakalářské práce/18096/2016/xvitra00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Vít Radek**
Obor: Informační technologie
Téma: **Překladové gramatiky: Vlastnosti a aplikace**
Translation Grammars: Properties and Applications
Kategorie: Teoretická informatika

Pokyny:

1. Seznamte se detailně s překladovými gramatikami.
2. Studujte vlastnosti překladových gramatik dle instrukcí vedoucího. Zaměřte se na studium jejich vstupních a výstupních jazyků.
3. Aplikujte překladové gramatiky v oblasti syntaxí řízeného překladu.
4. Implementujte aplikaci navrženou v předchozím bodě a testujte ji.
5. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

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

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

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstract

The goal of this thesis is to discuss translation grammars and their use in syntax-driven translation. It provides an example of the use of translation grammars in predictive top-down translation and provides formal devices for language accepting and translation output generation. It examines the descriptional complexity of tranlation grammars as language generating devices. An implementation of a translation framework using these devices is described and a simple compiler is designed and implemented using this framework. This compiler translates a new language REON designed in this paper to Python 3.

Abstrakt

Cílem této práce je prozkoumat překladové gramatiky a jejich použití v syntaxí řízeném překladu. Práce ukazuje využití překladových gramatik v prediktivním syntaktickém překladu a definuje formální prostředky pro přijímání jazyků a generování výstupu překladu. Zkoumá sílu překladových gramatik při jejich použití jako prostředku pro generování jazyků. Práce dále popisuje implementaci překladové knihovny používající tyto formální prostředky a popisuje implementaci jednoduchého překladače pomocí této knihovny. Tento překladač překládá nový jazyk REON navržený v této práci do Pythonu 3.

Keywords

translation grammars, syntax analysis, compilers

Klíčová slova

překladové gramatiky, syntaktická analýza, překladače

Reference

VÍT, Radek. *Translation Grammars: Properties and Applications.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Meduna Alexander.

Translation Grammars: Properties and Applications

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.

.....
Radek Vít
May 11, 2017

Acknowledgements

I would like to thank my supervisor prof. RNDr. Alexander Meduna, CSc. for introducing translation grammars to me and his tremendous help with researching this topic.

Contents

Chapter 1

Introduction

Translation is an important part of the computer science world, as all programming languages have to be translated to either machine instructions or another interpreted language. Even though formal devices have been seldom used to define translations in practice, the use of translation grammars can offer several advantages. Translation grammars provide a human-readable way to define equivalency between input and output constructs and they serve as a formal device for translation definitions, which means that for any translation defined by a translation grammar, its correctness can be formally proven.

This paper lays the groundwork for building translation tools using translation grammars. Top-down predictive parsing is one of the widely used parsing methods; the algorithms used for this method of parsing are adapted to work with translation grammars, creating top-down predictive translation.

Chapter 2 reviews some basic ideas from logic, discrete mathematics, and language theory. It contains the definitions and conventions used throughout this paper. It introduces language generating devices in grammars, and language accepting devices in automata.

The core of this paper is in chapter 3. It introduces and defines translation grammars. The methods and algorithms of predictive top-down parsing are adapted for translation grammars and modified so that they not only accept languages, but generate their translations. This chapter also discusses the descriptional complexity of translation grammars and examines the ways to generate languages with high complexity using translation grammars.

In chapter 4, we overview the devices in Python 3 for matching regular expressions. Then, we propose and design the language REON (Regular Expression Object Notation), which serves as a language for expressing regular expressions. Finally, we design a translation grammar that generates translations from REON to Python 3.

Chapter 5 discusses the implementation of a translation framework based on the ideas and algorithms introduced in chapter 3. An application is implemented using this framework and its properties and usage are discussed. This application performs the translation from REON to Python 3 as proposed in chapter 4.

Some basic understanding and knowledge of logic, discrete mathematics (see [7]), language theory (see [1], [2], and [6]), basic algorithms, basic abstract data types, and the languages C++ and Python (see [15]) are required to fully understand this paper.

Chapter 2

Preliminaries

This chapter reviews some basic concepts from set theory, discrete mathematics and formal languages. This chapter also introduces the terminology used throughout this paper and introduces some formal language systems.

2.1 Sets and Relations

This section reviews some basic ideas from set theory (see [9]).

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

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

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

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

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

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

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

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

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

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

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

.

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

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

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

2.2 Alphabets and Languages

This section reviews some basic ideas from language theory (see [9]).

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

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

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

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

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

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

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

.

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

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

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

2.3 Grammars

Finite languages can be defined by listing all their components. Infinite languages, however, are impossible to define by enumeration. To define both finite and infinite languages, we introduce *grammars*. Grammars are devices that generate languages and play a major role in formal language theory. The following definitions are cited from [13].

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

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

where

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

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

$$x \Rightarrow_G y$$

only if $x = x_1 u x_2$, $y = x_1 v x_2$, and $u \to v \in P$, where $x_1, x_2 \in (N \cup T)^*$. The relation $\Rightarrow_G$ is often denoted by $\Rightarrow$ where no ambiguity rises. Let $S \Rightarrow^* x$, $x \in (N \cup T)^*$. Then, x is a *sentential form*. If $x \in T^*$, then x is a *sentence*. If x is is a sentence, then $S \Rightarrow^* x$ is a *successful derivation*.
The *language of* G, denoted by $L(G)$, is the set of all sentences defined as

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Theorem 2.3.11 (Chomsky hierarchy, see [4])**.**

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

Definition 2.3.12 (page 20 of [12])**.** A *queue grammar* is a sextuple

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

where

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

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

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

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

2.4 Automata

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

Definition 2.4.1 (see [13])**.** A *finite state automaton* is a quintuple

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

where

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

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

$$pyx \vdash qx$$

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

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

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

Definition 2.4.2 (see [13])**.** A *pushdown automaton* is a septuple

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

where

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

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

$$xvpay \vdash xwqy$$

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

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

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

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

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

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

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

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

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

where

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

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

$$a_i|a_obc \to d_i|d_oe \in R$$

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

$$s_ii|s_oopay \vdash s_iI|s_iOqy$$

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

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

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

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

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

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

2.5 More Definitions

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

Definition 2.5.1. *Regular expressions* define the family of **REG**. Quoting verbatim from [9], they are recursively defined as follows:

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

Parentheses are ommited when no ambiguity arises.

Chapter 3

Translation Grammars

Definition 3.0.1. A *translation grammar* is a quintuple

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

where

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

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

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

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

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

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

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

The *input language* is defined as

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

The *output language* is defined as

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

Definition 3.0.2. A *linear translation grammar* is a translation grammar as described in definition 3.0.1 with productions in P in the forms

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

and

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

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

Translation grammars are essentially two context-free grammars with a common set of nonterminals and linked sets of productions. Whereas grammars generate languages, translation grammars generate binary relations called *translations*. Each production starts from a single nonterminal and generates two strings. The input and output right-hand sides of each production must contain the same nonterminals in the same order. Translation grammars define translations $T(G)$, and two languages $L_I(G)$ and $L_O(G)$. They can also be used as language generating systems, as discussed in section 3.2.

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

3.1 Syntax-directed translation using translation grammars

To demonstrate the use of translation grammars in syntax-directed translation, this section describes the use of translation grammars by modifying top-down predictive parsing. Top-down predictive parsing is a well-documented method for accepting context-free languages defined by LL grammars and the devices needed for predictive parsing are described in multiple sources (see [9], [11]). This paper modifies these devices to work with LL translation grammars and to generate output, creating *predictive top-down translation*.

3.1.1 Two-stack Pushdown Automaton as a Translation Model for Top-down Translation

Using a two-stack pushdown automaton to simulate translation grammar top-down translation is similar to using a pushdown automaton to simulate top-down parsing of context-free grammars. The following method for constructing two-stack pushdown automation from translation grammars creates a two-stack pushdown automation that accepts a language by empty input pushdown and final state. Algorithm 3.1 describes the construction of such two-stack pushdown automata.

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

The other group of rules simulates the translation grammar's productions. This group of rules simulates the expantion of nonterminals on both input and output stacks. First,

Algorithm 3.1 Two-stack pushdown automation simulating a translation grammar (based on the algorithm on page 47 of [10])

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

$S_I \coloneqq S$
$S_O \coloneqq S$
$Q \coloneqq \{s, -\}$
$\Sigma \coloneqq T_I$
$\tau \coloneqq N \cup T_I \cup T_O \cup \{\#\}$
$F \coloneqq \{s\}$
for all $x \in T_I$, $y \in N \cup T_O$ **do**
    Add $x|ysx \rightarrow |ys$ to R
for all $f, x \in N \cup T_O$ **do**
    Add $x|f- \rightarrow |fx-$ to R
    Add $\#|f- \rightarrow |fs$ to R
for all $Z : (A, A) \rightarrow (b, c) \in P$, where Z is a unique rule label **do**
    Add $+_Z$ to Q
    for all $d \in N \cup T_O$ **do**
        Add $A|ds \rightarrow e\#|d+_Z$ to R, where $e = reversal(b)$
    for all $x \in N \cup T_O \cup \{\#\}$ **do**
        for all $B \in N \cup T_O$, where $B \neq A$ **do**
            Add $x|B+_Z \rightarrow xB|+_Z$ to R
        Add $x|A+_Z \rightarrow x|e-$ to R, where $e = reversal(c)$

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

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

3.1.2 Predictive Top-Down Translation

This section introduces the formal devices and algorithms used for predictive top-down translation using translation grammars. The algorithms and procedures introduced here are modifications of algorithms used for top-down parsing LL grammars (see page 145 in [9]). LL grammars are a proper subset of context-free grammars, which means that the descibed algorithms will only work for a proper subset of translation grammars. However, predictive

parsing is a sufficient demonstration of transforming language accepting devices to language translating devices and the other parsing methods can be transformed using similar methods.

In predictive parsing, a special symbol $\$ \notin N \cup T_I \cup T_O$ is introduced. It is, without any loss of generality, appended to the end of the generated strings and denotes the end of these strings. In two-stack pushdown automation, it denotes the end of input and output. This allows for deterministic behavior of the simulating two-stack pushdown automaton when reaching the end of the input string.

Basic idea. For any translation grammar $G = (N, T_I, T_O, P, S)$ where $\$ \notin N \cup T_I \cup T_O$, add a new nonterminal S', new input and output terminal \$, a production $(S', S') \to (S\$, S\$)$ and make the new nonterminal S' the new starting symbol. The resulting translation grammar

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

generates the same translations, but with the symbol \$ appended to both input and output of each translation.

Definition 3.1.1. Let $G = (N, T_I, T_O, P, S)$ be a translation grammar. For every string $x \in (N \cup T_I)^*$,

$$first(x) = \{a : x \Rightarrow^* w, w \in T_I^*, a \ = symbol(w, 1)\}$$

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

The set $first(x)$ is the predictive set of input terminals or ε that can be the first symbol in a sentential form derived from x. This predictive set will be used later for the construction of other predictive sets. Algorithm 3.2 creates $first(x)$ for every $x \in N \cup T_I \cup \{\varepsilon\}$. Algorithm 3.3 creates $first(x)$ for any string $x \in (N \cup T_I)^*$ when $first(y)$ is known for all $y \in N \cup T_I \cup \{\varepsilon\}$.

Algorithm 3.2 first (based on [11] and [9])

Input: translation grammar $G = (N, T_I, T_O, P, S)$
Output: Set $first(x)$ for each $x \in N \cup T_I \cup \{\varepsilon\}$

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

Definition 3.1.2. For every nonterminal X, we define the set $follow$:

$$follow(X) = \{x \in T_I \cup \{\$\} : substring(F_I(G), Xx)\}$$

Algorithm 3.3 string first (based on [11])

Input: translation grammar $G = (N, T_I, T_O, P, S)$, string $x \in (N \cup T_I)^*$, $first(X)$ for every $X \in N \cup T_I \cup \{\varepsilon\}$
Output: set $first(x)$
  $first(x) \coloneqq first(symbol(x, 1))$
  repeat
    if $\varepsilon \in first(symbol(x, n)$ for $n = 1, \ldots, k-1$ where $k \leq |x|$ **then**
      Add all symbols from $first(symbol(x, k))$ to $first(x)$
  until no change

For every nonterminal, the predictive set *follow* contains all terminals (or \$) that may follow the nonterminal in a sentential form. Algorithm 3.4 creates follow for every nonterminal in the translation grammar G.

Algorithm 3.4 follow (based on [11] and [9])

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

Definition 3.1.3. The *predict* set is defined for every $a = (A, A) \to (i, o) \in P$ as

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

The predictive set *predict* is the set of all terminals that may be the leftmost generated symbol if we apply the production a to the leftmost nonterminal. That is, for all $(xAq, yAr) \Rightarrow^* (xB, yC)$ where $x, B \in T_I^*$, $y, C \in T_O^*$, $A \in N$, $q \in (N \cup T_I)^*$, $r \in (N \cup T_O)^*$, and $a = (A, A) \to (i, o) \in P$ is the first production used in the derivation, $predict(a)$ is the set of all terminals t such that $t = symbol(B, 1)$.

Definition 3.1.4. A translation grammar $G = (N, T_I, T_O, P, S)$ is an *LL translation grammar* if for each $A \in N$, any two different productions $p = (A, A) \to (x, y)$, $q = (A, A) \to (i, o)$, satisfy $predict(p) \cup predict(q) = \varnothing$.

In LL grammars (and LL translation grammars), the first L stands for a left-to-right scan of symbols and the other L stands for leftmost derivation.

Next, we introduce a *predictive table*. This table determines a production for each pair of nonterminal and terminal. In predictive top-down translation, this table determines

Algorithm 3.5 Constructing a predictive table

Input: LL translation grammar $G = (N, T_I, T_O, P, S)$, $predict(x)$ for each $x \in P$
Output: predictive table α

```
for all t ∈ T_I ∪ {$}, n ∈ N do
    α(n, t) := null
for all y ∈ predict(x) for each x = (A, A) → (i, o) ∈ P do
    α(A, y) := x
```

the production that is used for nonterminal expantion. Algorithm 3.5 shows the construction of this table.

Algorithm 3.6 describes the predictive top-down translation method. This method is based on the two-stack automaton described in section 3.1.1 and algorithm 7.17 from [9]. The contents of stacks *ipd* and *opd* are written head-first; the first symbol is the symbol on top of the stack. In addition to regular stack functionality (*POP*, removing the top element from the stack and *PUSH*), both stacks support the operation *REPLACE*. The operation REPLACE(n, x), where $n \in N$, $x \in (N \cup T_I \cup T_O)^*$ replaces the symbol n closest to the top of the stack with the string x. The first symbol in x will be closest to the top of the stack. This behavior can be achieved with two stacks with only PUSH and POP (as described in 3.1.1, but we introduce the operation REPLACE for brevity.

3.2 Descriptional Complexity of Translation Grammars

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

Lemma 3.2.1. Recall Lemma 2.38. in [12]. Let Q' be a queue grammar. Then, there exists a queue grammar

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

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

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

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

Lemma 3.2.2. For every queue grammar Q, there exists a linear translation grammar G such that

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

Proof. Without any loss of generality, assume that the queue grammar Q satisfies the conditions described in Lemma 3.2.1.

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

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

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

Algorithm 3.6 Predictive top-down translation

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

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

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

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

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

Basic idea. The constructed translation grammar G simulates the queue grammar Q that satisfies the properties described in Lemma 3.2.1. The production from 1, applied only once, initialises the derivation. The production 4 terminates the derivation. The productions

from 2 simulate the rules applied by Q. Finally, productions from 3 generate the simulated terminals to the input string in the order generated by Q.

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

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

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

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

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

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

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

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

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

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

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

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

Proof of claim B. Recall that Q satisfies the properties given in Lemma 3.2.1. These properties imply that Claim B holds.

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

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

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

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

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

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

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

Lemma 3.2.2 implies that any queue grammar can be simulated by a linear translation grammar. Since linear queue grammars define the family of recursively enumerable languages **RE**, linear translation grammars can be used to define any recursively enumerable language. Because of this descriptional complexity, any grammar or rewriting system defining any subset of recursively enumerable languages can be simulated by a linear queue grammar.

Chapter 4

Designing a Translation

In this chapter, we design the translation from REON to Python 3.5. The first section describes the subset of Python 3.5, and its module `re`. For brevity, in the rest of this paper, Python 3.5 will be referred to as Python 3 or simply Python. The second section defines the language REON. It is a simple JSON-based language defining regular expressions. The third and final section describes the translation between the two languages.

4.1 Python 3 and its Regular Expression Notation

In Python 3, regular expressions are strings that describe the regular expression pattern. The `re` module uses these patterns for its string matching operations. This section does not provide a comprehensive description of the language and neither does it describe the entirety of the `re` package. Instead, it focuses on the understanding of the basics of Python's regular expression notation. The description of the regular expression constructs in this section is quoted essentially verbatim from [14].

Regular expression patterns are stored in variables as strings. The recommended way to store RE patterns is using raw strings, denoted by `r"..."`. These strings treat backslashes as literal characters and do not process any escaped characters.

Python identifiers are an alphabetic character followed by any number of alphanumeric characters or underscores. Defining a variable in python is done the following way:
`var = expr`
This one-line expression assigns the result of the expression *expr* to variable *var*.

A regular expression (or RE) specifies a set of strings that matches it. The functions in `re` let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if A and B are both regular expressions, then AB is also a regular expression. In general, if a string p matches A and another string q matches B, the string pq will match AB. This holds unless A or B contain low precedence operations; boundary conditions between A and B; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like `'A'`, `'a'`, or `'0'`, are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so `last` matches the string `'last'`.

Some characters, like `'|'` or `'('`, are special. Special characters either stand for classes of ordinary characters or affect how the regular expressions around them are interpreted. Regular expression pattern strings may not contain null bytes but can specify the null byte using a `\number` notation such as `'\x00'`.

Repetition qualifiers (`*`, `+`, `?`, `{m, n}`, etc.) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix `?`, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression `(?:a{6})*` matches any multiple of six `'a'` characters.

The special characters are:

- `'.'`
  This matches any character except a newline. If the *DOTALL* flag has been specified, this matches any character including a newline.

- `'^'`
  Matches the start of the string, and in *MULTILINE* mode also matches immediately after each newline.

- `'$'`
  Matches the end of the string or just before the newline at the end of the string, and in *MULTILINE* mode also matches before a newline. `foo` matches both `'foo'` and `'foobar'`, while the regular expression `foo$` matches only `'foo'`. More interestingly, searching for `foo.$` in `'foo1\nfoo2\n'` matches `'foo2'` normally, but `'foo1'` only in *MULTILINE* mode; searching for a single `$` in `'foo\n'` will find two (empty) matches: one just before the newline, and one at the end of the string.

- `'*'`
  Causes the resulting RE to match 0 or more repetitions of the preceding RE.

- `'+'`
  Causes the resulting RE to match 1 or more repetitions of the preceding RE.

- `'?'`
  Causes the resulting RE to match 0 or 1 repetitions of the preceding RE.

- `*?`, `+?`, `??`
  The `'*'`, `'+'`, and `'?'` qualifiers are all greedy; they match as much text as possible. Adding `?` after the qualifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched.

- `{m}`
  Specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match.

- `{m,n}`
  Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible. Omitting m specifies a lower bound of zero, and omitting n specifies an infinite upper bound.

- `{m,n}?`
  Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier.

- '\'
  Either escapes special characters (permitting you to match characters like '*', '.', and so forth), or signals a special sequence; special sequences are discussed below.

- []
  Used to indicate a set of characters. in a set:

  - Characters can be listed individually, e.g. [amk] will match 'a', 'm', or 'k'.
  - Ranges of characters can be indicated by giving two characters and separating them by a '-'. If - is escaped or if it's placed as the first or last character, it will match a literal '-'.
  - Special characters lose their special meaning inside sets.
  - Character classes such as \w or \S (defined below) are also accepted inside a set, although the characters they match depends on whether ASCII or LOCALE mode is in force.
  - Characters that are not within a range can be matched by complementing the set. If the first character of the set is '^', all the characters that are not in the set will be matched. ^ has no special meaning if it's not the first character in the set.
  - to match a literal ']' inside a set, precede it with a backslash, or place it at the beginning of the set.

- '|'
  A|B, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the '|' in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by '|' are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. To match a literal '|', use \|, or enclose it inside a character class, as in [|].

- (...)
  Matches whatever expression is inside the parentheses and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the \number special sequence, described below. To match the literals '(' or ')', use \(or \), or enclose them inside a character class: [(] [)]. If the group is repeated (for example, with the operator +), only the last iteration is captured.

- (?...)
  This is an extension notation (a '?' following a '(' is not meaningful otherwise). The first character after the '?' determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; (?P<name>...) is the only exception to this rule. Following are the currently supported extensions.

- (?aiLmsux)
  (One or more letters from the set 'a', 'i', 'L', 'm', 's', 'u', 'x'.) The group matches the empty string; the letters set the corresponding flags: re.A (ASCII-only matching), re.I (ignore case), re.L (locale dependent), re.M (multi-line), re.S (dot matches all), and re.X (verbose), for the entire regular expression. (The flags are

described in Module Contents.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a flag argument to the re.compile() function.

- `(?:...)`
  A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group cannot be retrieved after performing a match or referenced later in the pattern.

- `(?P<name>...)`
  Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name *name*. Group names must be valid Python identifiers, and each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.

- `(?P=name)`
  A backreference to a named group; it matches whatever text was matched by the earlier group named *name*.

- `(?#...)`
  A comment; the contents of the parentheses are simply ignored.

- `(?=...)`
  Matches if `...` matches next, but doesn't consume any of the string. This is called a *lookahead assertion*.

- `(?!...)`
  Matches if `...` doesn't match next. This is a *negative lookahead assertion*.

- `(?<=...)`
  Matches if the current position in the string is preceded by a match for `...` that ends at the current position. This is called a *positive lookbehind assertion*. The contained pattern must only match strings of some fixed length, meaning that `abc` or `a|b` are allowed, but `a*` and `a{3,4}` are not.

- `(?<!...)`
  Matches if the current position in the string is not preceded by a match for `...`. This is called a *negative lookbehind assertion*. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length.

- `(?(id/name)yes-pattern`no-pattern)|
  Will try to match with yes-pattern if the group with given *id* or *name* exists, and with no-pattern if it doesn't. no-pattern is optional and can be omitted.

The special sequences consist of `'\'` and a character from the list below. If the ordinary character is not an ASCII digit or an ASCII letter, then the resulting RE will match the second character. For example, `\$` matches the character `'$'`.

- `'\number'`
  Matches the contents of the group of the same number. Groups are numbered starting from 1. This special sequence can only be used to match one of the first 99 groups. If the first digit of number is 0, or number is 3 octal digits long, it will not be interpreted

as a group match, but as the character with octal value number. Inside the '[' and ']' of a character class, all numeric escapes are treated as characters.

- \A
  Matches only at the start of the string.

- \b
  Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of Unicode alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore Unicode character. By default Unicode alphanumerics are the ones used, but this can be changed by using the *ASCII* flag. Inside a character range, \b represents the backspace character, for compatibility with Python's string literals.

- \B
  Matches the empty string, but only when it is not at the beginning or end of a word.

- \d
  Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes [0-9], and also many other digit characters. If the *ASCII* flag is used only [0-9] is matched.

- \D
  Matches any character which is not a Unicode decimal digit. This is the opposite of \d. If the *ASCII* flag is used this becomes the equivalent of [^0-9].

- \f
  Matches the form feed character.

- \n
  Matches the line feed character.

- \r
  Matches the carriage return character.

- \s
  Matches Unicode whitespace characters. If the *ASCII* flag is used, only [\t\n\r\f\v] is matched.

- \S
  Matches any character which is not a Unicode whitespace character. This is the opposite of \s.

- \t
  Matches the tab character.

- \v
  Matches the vertical tab character.

- \w
  Matches Unicode word characters; this includes most characters that can be part of a word in any language, as well as numbers and the underscore. If the *ASCII* flag is used, only [a-zA-Z0-9_] is matched.

- `\W`
  Matches any character which is not a Unicode word character. This is the opposite of `\w`.

- `\Z`
  Matches only at the end of the string.

4.2 The REON language

This section describes the language REON (Regular Expression Object Notation). It is based on the JSON markup language. REON, however, is not a subset of the JSON language. It supports escaped characters not supported in JSON and it allows trailing commas at the end arrays. This description is derived from the JSON language description (see [3]). Although REON does not conform to the JSON specification, its syntax was heavily inspired by it.

Its purpose is to describe regular expressions within an object notation. This form of regular expression notation clearly defines the order of operations and thus doesn't require its user to remember the operations' precedence and associativity. Note that this language describes regular expressions with potentially higher complexity than regular expressions defined in 2.5.1.

4.2.1 REON Tokens

A REON text is a sequence of tokens. The set of tokens contains three literal names, six structural characters, strings, numbers and special identifiers. Whitespace characters are ignored when parsing REON. The structural characters are `{`, `}`, `[`, `]`, `:` and `,`. The three literal names are `null`, `true` and `false`. They must be lowercase.

Numbers are respresented in base 10 using decimal digits. It contains an integer component that may be prefixed with an optional minus sign and may be followed by a fraction and/or an exponent part. Leading zeros are not allowed. An exponent part begins with the letter E in upper or lower case, which may be followed by a plus or minus sign. The E and optional sign are followed by one or more digits. The representation of numbers is visualised in figure 4.1.

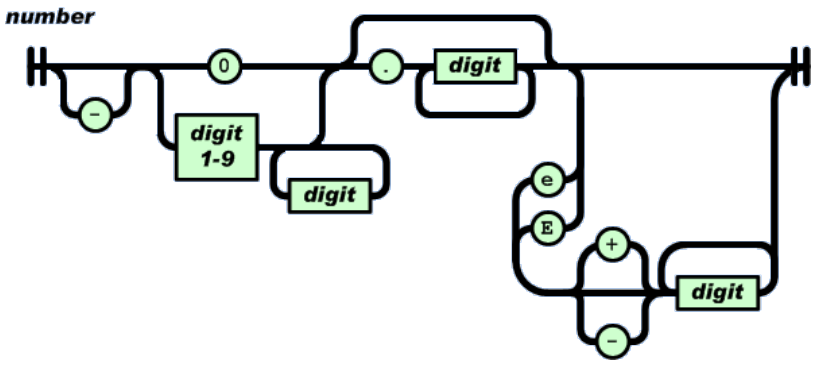

Figure 4.1: The representation of a number in both JSON and REON, from [5].

The representation of strings is similar to conventions used in the C family of programming languages. A string begins and ends with quotation marks. All characters may be placed

within the quotation marks, with the following exceptions: quotation mark (") and reverse solidus (\) must be escaped, and the control characters (0x00 through 0x1F) are forbidden.

Some strings may be interpreted as *keywords* instead. A string is a keyword only if followed by a comma and is one of the following:

- `"repeat rep"`
  The repeat keyword followed by a repetition parameter *rep*. *rep* is one of the following:
  - `*`
    Any number of repetitions.
  - `+`
    One or more repetitions.
  - `?`
    Zero or one repetition.
  - `n`
    Exactly n repetitions.
  - `n-m`
    A range of repetitions. Either n, or m can be omitted for a zero lower bound and an infinite upper bound, respectively.
- `"non-greedy repeat rep"`
  rep is the number of repetitions, same as above.
- `"set"`
- `"!set"`
- `"alternatives"`
- `"group name"`
  The optional *name* defines the group identifier. *name* must be a sequence of alphanumeric or underscore characters starting with an alphabetic character.
- `"match group"`
- `"comment"`
- `"lookahead"`
- `"!lookahead"`
- `"lookbehind"`
- `"!lookbehind"`
- `"if"`
- `"then"`
- `"else"`

4.2.2 REON Matching

This section describes the structure of REON and the semantics of its constructs.

A REON expression defines a regular expression (RE). A RE is any value. Values are one of the following: Object, array, string, number, null, true or false.

The literal true matches the empty string, the literals null and false never match.

Arrays are represented as square brackets surrounding zero or more values. Values are separated by commas. There is no requirement for the values to be of the same type. The last value of an array may have a trailing comma. Arrays represent either a series of REs that are appended (denoted by any free array), or a series of alternatives (see the alternatives object, described below)

Some characters can be escaped in strings, which causes them to match a character, class of characters or empty strings at specific places in a string. The following list contains characters that can be escaped along with the semantics of these escaped sequences.

- `\.`
  Matches any character.
- `\"`
  Matches the double quote character.
- `\\`
  Matches the reverse solidus character.
- `\A`, `\^`
  Matches the start of the string.
- `\b`
  Matches the empty string at the beggining or end of a word. A word is a sequence of alphanumeric or underscore characters.
- `\B`
  Matches the empty string only where `\b` does not match.
- `\d`
  Matches any decimal digit.
- `\D`
  Matches any character that is not a decimal digit.
- `\f`
  Matches the form feed character.
- `\n`
  Matches the line feed character.
- `\r`
  Matches the carriage return character.
- `\s`
  Matches any whitespace character.
- `\S`
  Matches any character that is not whitespace.

- `\t`
  Matches the tab character.

- `\v`
  Matches the vertical tab character.

- `\w`
  Matches an alphanumeric character or an underscore.

- `\W`
  Matches any character that is not alphanumeric or an underscore.

- `\Z`, `\$`
  Matches the end of the string.

In JSON, objects are unordered sets of name–value pairs. in REON, they may only contain predetermined sets of name–value pairs in a specified order to express specific regular expression operations. All objects start with the structural character `{` and end with `}`. Tokens are represented as `<token>`. The contents within the parentheses may be one of the following:

- `<repeat> : RE`
  The RE inside this object is matched the number of times specified by the repeat token. This is a greedy operation, as many repetitions as possible will be matched.

- `<non-greedy repeat>: RE`
  The RE inside this object is matched the number of times specified by the repeat token. This is not a greedy operation, as few repetitions as possible will be matched.

- `<set> : <string>`
  The string specifies a set of characters. This expression matches any the characters from the set. Character ranges can be specified by `a-z`. To match the character `-` inside a set, put `\-` in the string or as the first or last character in the string. Some escaped characters retain their meaning inside sets, but sequences `\^` and `\$` only match the characters `^` and `$`.

- `<!set> : <string>`
  Same as above, but this expression matches any characters that are not in the set.

- `<alternatives> : [RE, ...]`
  Matches one of the REs in the list. The first expression in the list has the highest matching priority; even if a subsequent expression could result in a longer match, the first expression that can be matched will be the result match.

- `<group name> : RE`
  Matches the expression defined by RE and stores the last match. It can later be retreived by its number or its optional identifier.

- `<match group> : ref`
  Retreives the last match of the group specified by *ref* — a number or string identifying an existing group.

- `<comment> : <string>`
  The contents of the string are ignored.
- `<lookahead> : RE`
  Matches an empty string when followed by the expression defined by RE.
- `<!lookahead> : RE`
  Matches an empty string when not followed by RE.
- `<lookbehind> : RE`
  Matches an empty string when it is preceded by RE. The RE must be a fixed length expression.
- `<!lookbehind> : RE`
  Matches an empty string when it is not preceded by RE. The RE must be a fixed length expression.
- `<if> : ref , <then> : RE , <else> : RE2`
  The conditional expression. If a group specified by *ref* was matched, the expression RE will be matched. If it was not matched, it will match RE2. The else name–value pair of this object is optional.

4.3 Translation from REON to Python 3

This section describes the design of the translation from REON to Python 3. The language REON was designed to be mostly compatible with the Python way of expressing regular expressions. In this section, the REON objects will be referred to by their first token names (e.g. *set*, *!lookbehind*, ...). The use of translation grammars allows us to define the equivalent string representations of REON expressions; each production in the resulting translation grammar will reflect these equivalencies in its input and output terminals.

4.3.1 Translating Structured Expressions

This section discusses the translation of structured REON expressions to Python.

REON append lists concatenate the expressions in the list. Appending expressions in Python is achieved by putting them in sequence. Translating this requires no special treatment.

The expression defined by a *string* result in an output terminal *re*. When generating this terminal, all REON escaped characters retain their meaning as defined above. The REON literals `\^` and `\$` do not semantically match `^` and `$` in Python, but the Python literals `\A` and `\Z` match the semantic meaning of REON's `\^` and `\$`. All other escaped literals are compatible. Characters that have a special meaning in Python must be escaped in the result Python pattern to match the literal characters. To ensure that the Python literal `.` matches all characters, the control sequence `(?s)` is inserted to the very beginning of the pattern, applying the DOTALL flag to the result pattern.

The *repeat* and *non-greedy repeat* objects can be transformed to Python by enclosing them into a non-capturing group and stating the correct repetition operator. Enclosing them in a non-capturing group is necessary; it guarantees that the repetition operator only applied to the expression within the object.

The *set* and *!set* objects contain only a string literal. The string contents — if properly escaped, as discussed before — must only be enclosed in `[...]` or `[^...]` to be valid Python sets. Characters that would end or otherwise manipulate the Python set — such as `]` or `^` — must be escaped in the translation output. To indicate this, the output terminal for the set contents is named *set*.

The *alternatives* object denotes a series of alternative expressions. This is denoted as `A|B` in Python. The output expression is enclosed in a non-capturing group to ensure the correct order of operations. Then, the first expression in the list is output as usual. Every next expression is divided from the previous by the `|` operator.

The *group* object can be defined with an optional identifier. Since Python denotes groups with and without identifiers in different ways, the named and anonymous group objects must be differentiated in the translation grammar. Group objects without identifiers can simply be enclosed in `(...)`. A special symbol is output to indicate the definition of a group. The named group objects will be enclosed in `(?P<name>...)` to specify the *name* of the object.

Match group objects reference previously declared groups. Number and identifier references are denoted differently in Python, so the translation of matching groups must be split into number references and identifier references. The number references will be translated to `\number` and identifier references will be translated to `(?P=name)`.

Comment objects have the direct equivalent in `(?#...)`. In comment strings, only the output `)` characters must be additionally escaped. To indicate this, the output terminal is named *comment*.

The *lookahead* and *!lookahead* objects can be expressed as `(?=...)` and `(?!...)`, respectively. *Lookbehind* and *!lookbehind* objects are expressed as `(?<=...)` and `(?<!...)`. In lookbehind expressions, the regular expression within must be of fixed length. To satisfy this, a special symbol must output before and after the closing parentheses to check the length of the expression within. When outputting other expressions, alternatives and variable length repeat operators are forbidden.

Finally, the *if* object is output as `(?(id/name)yes|no)`. The *id* or *name* reference is output differently than when matching groups; references in *if* objects must have a separate set of productions in the resulting translation grammar. If the optional `, "else": no` is omitted, the `|no` portion of the output is omitted.

4.3.2 Translation Grammar

The resulting translation grammar is not described here; in the following chapter, another component — attribute actions — will be added to its productions. To avoid duplicity, the full translation grammar as designed in this section is omitted. The translation grammar as implemented in chapter 5 is in appendix B. This translation grammar is based on the design in this chapter; only attribute actions were added.

Chapter 5

Implementation

This chapter's purpose is twofold. First, it describes the architecture and implementation of the translation framework ctf based on the predictive translation design in section 3.1. This framework aims to provide a simple automated interface for translation based on translation grammars. Second, it describes the implementation of the proposed translation designed in 4.3 from the language REON to Python 3 using the framework ctf.

5.1 Ctf

Ctf (ctf ain't a Translation Framework) is a C++ header-only translation framework. Its goal is to provide a quick way to define translations with automated creation of predictive tables and automatic translation directing.

The translation is divided into three parts: lexical analysis, syntax analysis and output generation. Lexical analysis supplies syntax analysis with tokens. Syntax analysis translates the input string of tokens to the output string of tokens according to a given translation grammar. Output generation generates text or binary output. Semantic analysis is done both during syntax analysis (see 5.1.2) and output generation.

5.1.1 Implementation details

Ctf is implemented in C++ under the standard C++14. C++ was chosen as the implementation language because of its efficiency, its vast array of procedural, functional, and object-oriented features, the good usability of its standard library and, in part, due to personal preference.

The library is implemented as header-only due to its small size and locality of use in larger projects. Even after potential future implementation other translation methods, its size should not become such that it would significantly affect compilation times. Implementing the library as header-only has the benefit of easy library integration and addresses the issue of ABI changes between different C++ compilers and their different versions.

For the full overview of implementation details, see the generated program documentation in the included CD (see appendix A).

5.1.2 Attribute actions

During syntax analysis, there is need for some semantic actions when applying productions. In ctf, this is achieved by adding *attribute actions* to each rule in translation grammars. Attribute actions define the relationship between input and output terminals and their

attributes. Each input token's attribute may be distributed to any number of output terminals, and an output terminal may receive any number of attributes from the input terminals. The user defines these relationships to propagate the attributes obtained during lexical analysis to semantic analysis.

The following example demonstrates the use of attribute actions. Consider the following translation grammar $G = (N, T_I, T_O, P, S)$, where

- $N = \{E, E', T, T', F\}$
- $T_I = \{i, +, *, (,)\}$
- $T_O = T_I$
- $P = \{$

$$
\begin{aligned}
(E, E) &\to (TE', TE'),\\
(E', E') &\to (\varepsilon, \varepsilon),\\
(E', E') &\to (+TE', T + E'),\\
(T, T) &\to (FT', FT'),\\
(T', T') &\to (\varepsilon, \varepsilon),\\
(T', T') &\to (*FT', F * T'),\\
(F, F) &\to ((E), E),\\
(F, F) &\to (i, i)
\end{aligned}
$$

  $\}$
- $S = E$

This translation grammar translates very simple infix expressions to postfix expressions. The terminal i represents a variable. If the lexical analyzer provides each of the i tokens attributes representing the variable name, then the corresponding output tokens should receive these attributes.

In ctf, to achieve this we construct `ctf::TranslationGrammar::Rule` with attribute actions to guarantee the attribute transfer during syntax analysis. `Rule` is then constructed as follows: `ctf::TranslationGrammar::Rule r{"F"_nt, {"i"_t}, {"i"_t}, {{0}}}`. The last element is a vector of sets of attribute actions. Attribute actions are only defined for input terminals and the targets may only be terminals or special symbols.

5.1.3 Class Model

This section describes some important classes in ctf. All symbols in ctf are in namespace `ctf`. For brevity, this namespace classification will be ommited when discussing the following classes.

For the purposes of translation, a new container was created. It is an extention of a stack with added functionality, including replacing a symbol in the stack with a string of symbols. This is implemented in the template class `tstack`, which is implemented using a double-linked list from the C++ standard library.

The various errors during the translation process throw exceptions. All ctf exceptions are subclasses of class `TranslationError`. Class `LexicalError` indicates an error in lexical

analysis, class `SyntaxError` indicates a translation error and `SemanticError` indicates a semantic error.

Symbols are represented by class `Symbol`. It represents nonterminal symbols, terminal symbols, the end of file symbol, and special symbols. The class can be constructed with an unknown type. Each `Symbol` object has a type, a name and an attribute. Both name and attribute are strings representing the symbol's name and attribute. Type determines which kind of symbol the object represents. The symbols with the type `SPECIAL` serve as special semantic markers. When generating output, terminal symbols could be used for this purpose instead, but the special symbol distinction provides a clear indicator of the difference in semantics.

The class `TranslationGrammar::Rule` represents individual translation grammar productions. The first component is the left-hand side nonterminal of the production. The second component is a vector of `Symbol`s representing the input right-hand side of the production. The third, optional component is a vector of `Symbol`s representing the output right-hand side of the production. If this component is missing when constructing the `Rule` object, the output right-hand side is identical to the input right-hand side. The last, optional component are attribute actions. It is a vector of sets of indices to the output vector of symbols. Attribute actions are defined for all terminals in the input and can be targeted to output terminals or special symbols. If the third component is not specified when the constructer is called, implicit attribute actions are created to transfer attributes to all corresponding terminals from input to output.

The class `TranslationGrammar` represents a translation grammar. It is defined by its (input) terminals, its nonterminals, a vector containing all its productions, and its starting symbol. The preffered way to construct this class is to only list the productions and the starting symbol. In this case, the sets of terminals and nonterminals are automatically deduced from the production contents.

Pure abstract class `TranslationControl` serves as the central translation controller for class `Translation`. Its subclass `LLTranslationControl` implements predictive top-down translation. Its output is a `tstack` filled with output symbols.

The class `Translation` is the main point of interaction for the framework's users. The user constructs the class with a lexical analyzer, translation grammar, translation control and output generator. The lexical analyzer and output generator are specified with C++ callables (`std::function`) and any function pointer or reference, lambda or callable object can be assigned as the lexical analyzer or output generator if conforming to their respective function signature requirements. The user chooses the translation method from the methods implemented in ctf (currently only predictive top-down translation with the name `"ll"`) or creates a custom `TranslationControl` subclass.

5.1.4 Translation Algorithms

To implement predictive top-down translation, algorithms described in section 3.1.2 were used. Slight modifications were made to the algorithms for the sake of efficiency and due to the object design of symbols. Algorithm 3.6 was modified to implement attribute actions. Its modification with attribute actions is described in algorithm 5.1.

Basic idea. When an expantion rule is used, attribute actions are created for each terminal in the input right-hand side of the production and are pushed in reverse order to the attribute action pushdown. When a comparing rule is used, the input symbol's attribute is distributed to the selected output terminals generated by the associated expantion rule.

Algorithm 5.1 Predictive top-down translation with attribute actions

Input: translation grammar $G = (N, T_I, T_O, P, S)$, its predictive table α_G, the input string $x\$$, where $x \in T_I^*$ and each symbol t in x has an associated attribute and a vector of sets of attribute actions for each production in $a \in P$.

Output: output string $r \in T_O^*\{\$\}$ with associated attributes for each symbol s in r if $x \in L_I(G)$, **ERROR** otherwise

```
ipd ≔ S$, opd ≔ S$
actionspd ≔ ε
result ≔ ε
n ≔ 1
repeat
    let X denote the current ipd top symbol
    let t ≔ symbol(x$, n)
    switch X:
        case X = $:
            if t = $ then SUCCESS, else ERROR
        case X ∈ T_I:
            if t = X then
                increment n and POP ipd
                let at denote the current actionpd top symbol
                add the attribute of t to all terminals in opd specified by at
                POP actionpd
            else
                ERROR
        case X ∈ N:
            if α(X, t) = null then
                ERROR
            else
                α(X, t) = (X, X) → (i, o)
                REPLACE(X, i) in ipd
                REPLACE(X, o) in opd
                for all s = symbol(i, n) for n = |i|, ..., 1, where s ∈ T_I do
                    let t denote the attribute actions in o for the nth symbol in i
                    t points to the terminals from o in opd where last placed
                    PUSH t to actionpd
until SUCCESS or ERROR
if SUCCESS then
    repeat
        let s denote the current opd top symbol
        r ≔ rs
        POP opd
    until opd is empty
```

5.1.5 Testing

Ctf was tested using a combination of unit testing and integration testing. The tests were written using the testing framework Catch. Because of its header-only nature, the objects that were not tested could not be replaced with mockups. Performance testing was not performed, as the framework is a prototype and its main goal was to prove the translation grammar related concepts introduced throughout this paper.

Unit testing was performed for low-level classes and containers included in ctf. A test suite for each translation unit was created to satisfy prime path coverage of roughly 80%. Integration testing was performed for the framework's high-abstraction classes and they cover most of ctf's implemented functionality. The classes with integration tests are `Translation` and `LLTranslationControl`.

5.2 Result Application

The translation from REON to Python 3 was implemented using the framework ctf. The resulting application is named reon (the language REON is written in capital letters for distinction). This section describes the result application and its components.

The application reon translates reon input to Python 3. The translation is implemented using the framework ctf. The lexical analyzer converts the input strings to tokens as described in section 4.2.1, the translation grammar is constructed according to the design in section 4.3 and the output generator is implemented so that the output is conforming to the Python `re` module requirements, as discussed in section 4.3. The output generator keeps track of the groups defined during output and reports invalid group references.

5.2.1 Usage

After compilation, an executable file `reon` is created. The application reon has the following optional arguments:

- `-h`
  prints help
- `-i file`
  sets the input to the specified file. The default input is stdin.
- `-o file`
  sets the output to the specified file. The default output is stdout.
- `-v variable`
  sets the Python variable name in the output to the specified name. The name must be a sequence of lowercase alphabetic characters. The default name is *re*.

5.2.2 Input and Output

The application reads characters from a file. If the string read is a member of the REON language (as described in section 4.2.2), the application performs a translation to Python 3. If any lexical, syntax or semantic error is encountered, the application outputs an empty string and the appropriate error code is returned.

The output is a valid Python 3.5 program, which defines a single variable with a string representing the regular expression defined by the input REON string. This variable may be used in other Python programs by importing the file containing it and using its contents in the matching and searching operations of the module `re`.

5.2.3 Translation Components

The lexical analyzer is implemented as a recursive descent lexical analyzer. It puts all characters from the input to a string buffer before transforming them to a string of input terminals. The lexical analyzer is largely based on the JSON language description. However, modifications were made to the specification to allow REON escapes only and to recognise REON keywords and REON keywords with parameters. When appropriate, the state representing methods of the class `ReonLexicalAnalyzer` iterate within themselves to avoid unneccesary overhead. Character lookahead is used when recognising keywords (such as `"group"`, etc.) in order to only recognise them as keywords when followed by a `:` character. Thus, strings matching a keyword but not followed by a `:` are regular strings. This distinction allows for a slight simplification of the translation grammar when a string object is expected. The lexical analyzer provides all tokens with attributes. The tokens with attributes distributed to the output string are `string`, `number`, `repeat`, `non-greedy repeat`, `named group` and `comment`.

The translation grammar is based on the design in section 4.3. Attribute actions were added to productions containing terminals with attributes to reflect those arguments in the output. Special symbols were added to the output strings to perform semantic checks or output application defined strings. The full `ctf::TranslationGrammar` as used in the resulting application is in appendix B.

The output generator generates Python 3 text output from the tokens obtained from the finished translation. It keeps track of all defined groups throughout the output process and checks whether a referenced group has been defined. When outputting a lookbehind or a negative lookbehind expression, the Python specification demands that the expression must have fixed length. This is achieved by checking the expression and throwing `ctf::SemanticError` when encountering any variable repetition or alternation.

5.2.4 Testing

Acceptance testing was performed, where the application specification is to translate REON input to Python 3. The tests were written as a shell script using standard Linux utilities. The individual tests were written as groups of files containing test input, expected output and additional reon arguments. The application was tested by translating a set of example input files covering all REON language features. Basic tests are in place to check whether the translation fails if it should.

Chapter 6

Conclusion

Translation grammars define translations between context-free languages. Defining formal devices for the use of translation grammars in syntax-directed parsing and translation allows for building translation tools based on formal devices and represents a clearly defined approach to translation.

Translation grammars are currently not widely used for translation. There exist translation frameworks (such as cdec) using transducers as formal translation defining devices. These frameworks, in most cases, require the developers to define the input and output languages with context-free grammars. This is only a step away from translation grammars and offers no advantage over defining translation grammars directly.

The advantages of using a formal definition of a translation over parsing the input and synthesising the output in an improvised way are clear; using translation grammars provides a provable platform with transparent equivalency between the input and output terminals in each production. For translating some languages there may be issues where the syntactic structures of the translated languages may be out of order, but addressing these issues with advanced output generation techniques using output hints and semantic markers can be used to circumvent these issues.

The potential generative power of translation grammar was proven to be the same as queue grammars, implying that translation grammars can be used to generate recursively enumerable languages. This property of translation grammars has not been proven and this paper is the first to suggest this property of translation grammars.

Concerning the much more practical uses of translation grammars, the formal devices used for predictive top-down parsing of context-free grammars were succesfully modified to work with translation grammars and to produce the output language output. The approach for modifying parsing methods to create translation methods used in this paper can be applied to most context-free grammar parsing methods to create other translation methods. Top-down predictive translation is suitable for a vast array of translation applications and is an important step for a more widespread use of translation grammars.

The language REON is an alternative look at defining regular expressions; most other notations define regular expressions in a way where their structure is very difficult to decipher for any programmer. REON defines clearly structured regular expressions and has most of the features of other modern extended regular expression notations.

The framework ctf designed in this paper has great potential for creating formal system-based translation tools. Manual creation of translation controlling tables is error-prone and tedious; ctf creates all translation controlling tables automatically at runtime with minimal machine time cost. The biggest strength of ctf is the short time it takes to define

a translation with it. The initial implementation of reon took only around three days to complete and the core of the translation — syntax analysis — only took around two to three hours to define.

The application reon may fill a niche in the computing world. It is, however, unlikely that the language REON will become popular or widely used for regular expression definition. The tool is primarily a working example of both translation definition with translation grammars and the usage of the framework ctf. When used by a shell a script, for example, it can produce a Python source file containing many automatically translated regular expression pattern variables, which may be used in a larger project requiring many regular expression matching patterns.

Since the generational power of translation grammars is the same as Turing machines, there must exist a way to simulate any formal system generating recursively enumarable languages by translation grammars. Future research can lead to discovering those methods, translation grammars' similarity with two-stack pushdown automata could potentially lead to viable solutions of those problems.

Other context-free grammar parsing methods still need to be modified to create translation methods. Bottom-up translation methods are useful for defining translations of expressions with large sets of operators with operator precedences and varying associativities, where top-down parsing is clearly not the best solution.

The framework ctf is already a functioning tool for defining and prototyping translations. However, a number of components need improvements. Some form of syntax error recovery must be put in place to allow analyzing the whole input string; reporting only the first error in the input string is an unacceptable way for the user to learn about the errors in his input; no state of the art tools behave this way. A rigid system for defining syntax error messages during syntax analysis needs to be put in place to allow developers to give the user proper information about the errors in the input. Bottom-up parsing methods such as LR and operator precedence need to be implemented based on the future modifications of these parsing methods to allow better definitions of expressions. A system for switching between top-down and bottom-up parsing must be put in place to allow a convenient way to translate programming languages. Finally, a lexical analyzer system using finite or symbolic automata can be considered for ctf to provide a unified and less error-prone lexical analysis interface than requiring a programmer-supplied function or callable object to perform the analysis.

Bibliography

[1] Aho, A. V.; Lam, M. S.; Ullman, J. D.; et al.: *Compilers: Principles, Techniques, and Tools*. Pearson. 2011. ISBN 0321486811.

[2] Aho, A. V.; Ullman, J. D.: *The Theory of Parsing, Translation, and Compiling (Volume I: Parsing)*. Prentice Hall. 1972. ISBN 0139145567.

[3] Bray, T.: RFC7149. IETF RFC 7149. March 2014. [Online; accessed 2017-04-11]. Retrieved from: https://tools.ietf.org/html/rfc7159.html

[4] Chomsky, N.: On certain formal properties of grammars. *Information and Control*. vol. 2, no. 2. 1959: pp. 137 – 167. ISSN 0019-9958. doi:http://dx.doi.org/10.1016/S0019-9958(59)90362-6. Retrieved from: http://www.sciencedirect.com/science/article/pii/S0019995859903626

[5] json.org: JSON number representation. 2017. [Online; accessed 2017-04-30]. Retrieved from: http://json.org/number.gif

[6] Martin, J.: *Introduction to Languages and the Theory of Computation*. McGraw-Hill Education. 2010. ISBN 0073191469.

[7] Matousek, J.; Nesetril, J.: *An Invitation to Discrete Mathematics*. Oxford University Press. 2008. ISBN 0198570422.

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

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

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

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

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

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

[14] Python Software Foundation: 6.2. re — Regular expression operations. 2017. [Online; accessed 2017-03-29].
Retrieved from: https://docs.python.org/3.5/library/re.html

[15] Python Software Foundation: Python 3.5.3 documentation. 2017. [Online; accessed 2017-05-01].
Retrieved from: https://docs.python.org/3.5/

Appendices

Appendix A

Contents of the Included CD

This appendix describes the contents of the included CD.

- **ctf** - This folder contains the framework ctf.
- **reon** - This folder contains the application reon.
- **README.txt** - This file describes ctf and reon dependencies and the steps needed to test ctf and to compile, test and run reon.

Appendix B

REON to Python 3 Translation Grammar

This is the translation grammar from REON to Python 3 as used in the utility reon. The literals `""_nt` denote nonterminal symbols, the literals `""_t` denote terminal symbols and literals `""_s` denote special symbols defined by the application (in the case of `"variable"_s`), or symbols only used for semantic analysis.

The individual rules are defined in the following way: The first argument of the constructor is the left-hand side nonterminal of the production. The second argument of the constructor is the input right-hand side of the production. The optional third argument is the output right-hand side of the production. Finally, the last optional argument of the constructor is the vector of attribute actions for this rule, as discussed in 5.1.2.

```
const TranslationGrammar reonGrammar{
  // rules
  {
    // first derivation
    {"E"_nt, {"RE"_nt}, {"variable"_s, " = r\"(?s)"_t, "RE"_nt, "\"\n"_t}},
    // empty regular expression
    {"RE"_nt, {}},
    // regular expression with some reon content
    {"RE"_nt, {"REFULL"_nt}},
    // match empty string
    {"REFULL"_nt, {"true"_t}, {"re"_t}},
    // don't match ever
    {"REFULL"_nt, {"false"_t}, {"(?!)"_t}},
    // don't match ever
    {"REFULL"_nt, {"null"_t}, {"(?!)"_t}},
    // string only RE
    {"REFULL"_nt, {"string"_t}, {"re"_t}, {{0}}},
    // append list
    {"REFULL"_nt, {"["_t, "RE-listE"_nt, "]"_t}, {"RE-listE"_nt}},
    // object
    {"REFULL"_nt, {"{"_t, "OBJ"_nt, "}"_t}, {"OBJ"_nt}},
    {"OBJ"_nt,
     {"repeat"_t, ":"_t, "RE"_nt},
```

```
 {"(?:"_t, "RE"_nt, ")"_t, "repeat"_t},
 {{3}, {}}},
{"OBJ"_nt,
 {"non-greedy repeat"_t, ":"_t, "RE"_nt},
 {"(?:"_t, "RE"_nt, ")"_t, "repeat"_t, "?"_t},
 {{3}, {}}},
{"OBJ"_nt,
 {"set"_t, ":"_t, "string"_t},
 {"["_t, "set"_t, "]"_t},
 {{}, {}, {1}}},
{"OBJ"_nt,
 {"!set"_t, ":"_t, "string"_t},
 {"[^"_t, "set"_t, "]"_t},
 {{}, {}, {1}}},
{"OBJ"_nt,
 {"alternatives"_t, ":"_t, "["_t, "RE-AlistE"_nt, "]"_t},
 {"RE-AlistE"_nt}},
{"OBJ"_nt,
 {"group"_t, ":"_t, "RE"_nt},
 {"("_t, "group"_s, "RE"_nt, ")"_t}},
{"OBJ"_nt,
 {"named group"_t, ":"_t, "RE"_nt},
 {"(?P<"_t, "named group"_t, ">"_t, "RE"_nt, ")"_t},
 {{1}, {}}},
{"OBJ"_nt, {"match group"_t, ":"_t, "Ref"_nt}, {"Ref"_nt}},
{"OBJ"_nt,
 {"comment"_t, ":"_t, "string"_t},
 {"(?#"_t, "comment"_t, ")"_t},
 {{}, {}, {1}}},
{"OBJ"_nt, {"lookahead"_t, ":"_t, "RE"_nt}, {"(?="_t, "RE"_nt, ")"_t}},
// negative lookahead
{"OBJ"_nt, {"!lookahead"_t, ":"_t, "RE"_nt},
 {"(?!"_t, "RE"_nt, ")"_t}},
{"OBJ"_nt,
 {"lookbehind"_t, ":"_t, "RE"_nt},
 {"(?<="_t, "fixed_length_check"_s, "RE"_nt, "end_check"_s, ")"_t}},
{"OBJ"_nt,
 {"!lookbehind"_t, ":"_t, "RE"_nt},
 {"(?<!"_t, "fixed_length_check"_s, "RE"_nt, "end_check"_s, ")"_t}},
// if-then[-else]
{"OBJ"_nt,
 {"if"_t, ":"_t, "IfRef"_nt, ","_t, "then"_t, ":"_t, "RE"_nt,
  "Else"_nt},
 {"(?("_t, "IfRef"_nt, ")"_t, "RE"_nt, "Else"_nt, ")"_t}},
// number reference
{"Ref"_nt, {"number"_t}, {"\\"_t, "nref"_t}, {{1}}},
// identifier reference
{"Ref"_nt, {"string"_t}, {"(?P="_t, "ref"_t, ")"_t}, {{1}}},
```

```
  // number reference in if-then[-else]
  {"IfRef"_nt, {"number"_t}, {"nref"_t}, {{0}}},
  // identifier reference in if-then[-else]
  {"IfRef"_nt, {"string"_t}, {"ref"_t}, {{0}}},
  // no else
  {"Else"_nt, {}},
  // optional else
  {"Else"_nt, {","_t, "else"_t, ":"_t, "RE"_nt}, {"|"_t, "RE"_nt}},
  // empty append list
  {"RE-listE"_nt, {}},
  // first element in append list
  {"RE-listE"_nt, {"REFULL"_nt, "RE-list"_nt}},
  // no more elements in append list
  {"RE-list"_nt, {}},
  // elements in append list
  {"RE-list"_nt, {","_t, "RE-list-comma"_nt}, {"RE-list-comma"_nt}},
  // no element after trailing comma in append list
  {"RE-list-comma"_nt, {}},
  // element after last comma in append list
  {"RE-list-comma"_nt, {"REFULL"_nt, "RE-list"_nt}},
  // empty alternation list
  {"RE-AlistE"_nt, {}},
  // first element in alternation list
  {"RE-AlistE"_nt,
   {"REFULL"_nt, "RE-Alist"_nt},
   {"(?:"_t, "REFULL"_nt, "RE-Alist"_nt, ")"_t}},
  // no more elements in alternation list
  {"RE-Alist"_nt, {}},
  // elements in alternation list
  {"RE-Alist"_nt, {","_t, "RE-Alist-comma"_nt}, {"RE-Alist-comma"_nt}},
  // no element after trailing comma in alternation list
  {"RE-Alist-comma"_nt, {}},
  // element after comma in alternation list
  {"RE-Alist-comma"_nt,
   {"REFULL"_nt, "RE-Alist"_nt},
   {"|"_t, "REFULL"_nt, "RE-Alist"_nt}},
},
// starting nonterminal
"E"_nt};
```