VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OBECNÝ SYSTÉM PRO TESTOVÁNÍ INTERPRETŮ

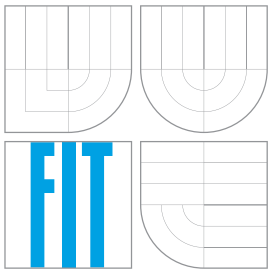
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAELA ŠIKULOVÁ

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OBECNÝ SYSTÉM PRO TESTOVÁNÍ INTERPRETŮ

GENERAL SYSTEM FOR TESTING OF INTERPRETS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MICHAELA ŠIKULOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ROMAN LUKÁŠ, Ph.D.
SUPERVISOR

BRNO 2009

Abstrakt

Práce se zabývá návrhem a implementací systému pro testování interpretů. Testování interpretů je založeno na principu porovnání výsledků interpretace testovacího programu s referenčními výsledky. Sada testovacích programů je získána pomocí překladu testovacích programů v referenčním jazyce na programy v jazyce testovaného interpretu. Testovací systém umožňuje po analýze souboru, který obsahuje popis syntaxe a sémantiky jazyka interpretu, vygenerovat testovací sadu programů pro interpret s odpovídajícími vstupy a referenčními výstupy. Pro tento účel využívá překladu jednoho vyššího programovacího jazyka na jiný.

Abstract

This work deals with concept and implementation of General system for testing of interprets. Testing of interprets is based on the principle of comparation of results of the interpretation carried out by the testing with reference results. The set of testing programmes is acquired by translating testing programmes, in the reference language to programmes in the language of tested interpret. The testing system enables generation of a testing set of programmes for an interpret according input and reference result after a preceding analysis of a file consisting of description of the syntax and semantics of the interpret's language. These actions result is translation of a more complex programming language to another.

Klíčová slova

Bezkontextová gramatika, překlad pomocí párových gramatik, lexikální analýza, rozšířený zásobníkový automat, LR syntaktická analýza, terminál, nonterminál, flex, bison, testování.

Keywords

Context-free grammar, two-grammar translation, lexical analysis, extended pushdown automata, LR syntax analysis, terminal, nonterminal, flex, bison, testing.

Citace

Michaela Šikulová: Obecný systém pro testování interpretů, bakalářská práce, Brno, FIT VUT v Brně, 2009

Obecný systém pro testování interpretů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením Ing. Romana Lukáše, Ph.D. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Michaela Šikulová
18. května 2009

Poděkování

Na tomto místě bych ráda poděkovala Ing. Romanu Lukášovi, Ph.D. za odborné vedení bakalářské práce a také za cenné rady a konzultace při vypracování této práce.

Obsah

Kapitola 1

Úvod

V této bakalářské práci se zabývám návrhem a implementací systému pro testování interpretů. Využívám pro tento účel znalostí získaných z předmětů Formální jazyky a překladače a Výstavba překladačů.

Testování interpretů je založeno na principu porovnání výsledků interpretace testovacího programu s referenčními výsledky. Sada testovacích programů je získána pomocí překladu testovacích programů v referenčním jazyce na programy v jazyce testovaného interpretu.

Testovací systém umožňuje po analýze vstupního souboru, který obsahuje popis syntaxe a sémantiky jazyka interpretu v mém jazyce, vygenerovat testovací sadu programů pro interpret s odpovídajícími vstupy a referenčními výstupy. Pro tento účel využívá překladu jednoho vyššího programovacího jazyka (referenčního) na jiný (jazyk testovaného interpretu). Tím umožňuje vytvoření testovacích programů z programů napsaných v referenčním jazyce.

Technická zpráva je rozdělena do šesti kapitol. Kapitola 2 – Základní definice, je teoretickým úvodem a obsahuje některé základní pojmy, definice a principy, na kterých je tato práce postavena.

Kapitola 3 se zabývá návrhem testovacího systému. Jelikož je testovací systém navržen tak, že se interpret testuje pomocí referenční sady programů, obsahuje tato kapitola i návrh generátoru překladače jednoho programovacího jazyka do jiného (sekce 3.1) a popis jazyka, v němž je možné popsat jednotlivá pravidla pro překlad těchto dvou jazyků (sekce 3.2). Jsou zde vysvětleny principy, na kterých tento překlad funguje a překlad pomocí dvou gramatik je ilustrován v názorném příkladu (sekce 3.1.4).

V kapitole 4 je popis implementace překladače, jehož vstupem je program napsaný v mém jazyce a výstupem jsou programy pro Flex a Bison, s jejichž pomocí je vygenerován překladač vstupního jazyka na výstupní (podle pravidel popsaných v programu v mém jazyce).

Implementaci a použití testovacího systému obsahuje kapitola 5. Je zde vysvětlen princip testování testovacím skriptem a ovládaní testovacího systému. Sekce 5.3 pojednává o ověření správné implementace testovacího systému.

Zhodnocení dosažených výsledků a náměty na rozšíření projektu najdete v závěrečné kapitole 6.

V rámci Semestrálního projektu jsem se seznámila s různými druhy interpretů a principem jejich činnosti. Dále jsem vytvořila stručný návrh, podle kterého je syntax a sémantika jazyka testovaného interpretu popsána párovými gramatikami (návaznost v sekci 3.1.4) a je vztahována k jazyku C – sémantiku jazyka popisuje jazyk C.

Kapitola 2

Základní definice

Základní pojmy a definice jsou přejaty z materiálů k předmětu Formální jazyky a překladače – Přednášky k předmětu [6] a Studijní opora [7], a z materiálů k předmětu Výstavba překladačů – Přednášky k předmětu [8] a Studijní opora [9].

2.1 Základní pojmy

2.1.1 Abecedy, slova, jazyky

Abeceda: *Abeceda* je libovolná neprázdná konečná množina. Prvky abecedy nazýváme *symboly*.

Slovo: Nechť Σ je abeceda. *Slovo*, popř. řetězec, nad Σ je konečná posloupnost symbolů ze Σ. Prázdnou posloupnost budeme nazývat prázdné slovo a označovat jej ε.

1. ε je slovo nad abecedou Σ
2. pokud x je slovo nad Σ a $a \in \Sigma$, potom xa je řetězec nad abecedou Σ

Délka slova: Nechť x je řetězec nad abecedou Σ. Délka slova x, $|x|$, je definována:

1. pokud $x = \varepsilon$, pak $|x| = 0$
2. pokud $x = a_1 \dots a_n$, pak $|x| = n$

Jazyk nad abecedou: Nechť Σ^* značí množinu všech slov nad Σ. Každá podmnožina $L \subseteq \Sigma^*$ je *jazyk nad* Σ. Poněvadž jazyky jsou množiny (slov), lze s nimi provádět množinové operace.

2.1.2 Prostředky pro specifikaci jazyků

Pro specifikaci formálních jazyků potřebujeme konečné prostředky, abychom s nimi mohli v praxi pracovat. Konečné jazyky lze jednoduše popsat výčtem jejich slov. Tato specifikace je však nemožná pro nekonečné jazyky (mezi které patří prakticky všechny programovací jazyky). Jiné prostředky pro specifikaci jazyka jsou gramatiky (v sekci 2.1.2 – Chomského klasifikace gramatik) a automaty (v sekci 2.3.2 – konečné automaty). Jedná se o dva základní typy konečné reprezentace (nejen konečných, ale i nekonečných) jazyků.

Podstatou každé gramatiky je konečná množina pravidel, pomocí kterých lze generovat daný jazyk. Jednotlivá slova tohoto jazyka se vytváří postupným přepisováním řetězců podle těchto pravidel.

Automat definuje jazyk také pomocí konečných prostředků. Jedná se o algoritmus, který pro libovolné slovo nad vstupní abecedou rozhodne, zda patří či nepatří do daného jazyka.

Chomského klasifikace gramatik

Gramatika: Generativní gramatika je čtveřice $G = (N, T, P, S)$, kde

1. N je abeceda nonterminálů
2. T je abeceda terminálů, $N \cap T = \emptyset$
3. P je konečná množina pravidel, $P \subseteq (N \cup T)^* N (N \cup T) \times (N \cup T)^*$
4. S je počáteční symbol, $S \in N$

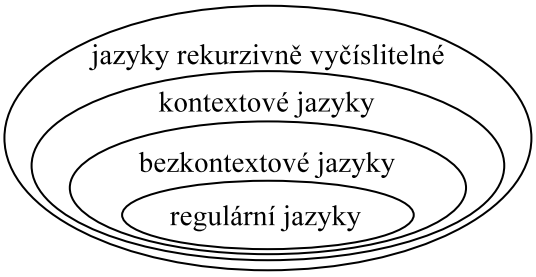
Tuto gramatiku budeme nazývat *gramatika typu 0.*

Gramatika typu 1: Buď $G = (N, T, P, S)$ gramatika typu 0 a pro každé pravidlo $v \to w$ z P platí $|v| \leq |w|$; jedinou výjimkou může být pravidlo $S \to \varepsilon$, jehož výskyt však implikuje, že se S nevyskytuje na pravé straně žádného pravidla. Pak tuto gramatiku G nazýváme *gramatika typu 1*, popř. *kontextová gramatika.*

Gramatika typu 2: Buď $G = (N, T, P, S)$ gramatika typu 0, pro kterou platí $v \to w \in P \Rightarrow v \in N$. Pak G nazýváme *gramatika typu 2*, neboli *bezkontextová gramatika.*

Gramatika typu 3: Buď $G = (N, T, P, S)$ gramatika typu 0 a každé pravidlo z P je tvaru $A \to xB$, nebo tvaru $A \to x$, kde $A, B \in N, x \in T^*$. Pak G nazýváme *gramatika typu 3*, neboli *regulární gramatika.*

Máme-li gramatiku typu i, $i \in \{0, 1, 2, 3\}$, pak jazyk, který tato gramatika generuje, nazýváme jazyk typu i. Podobně jako tomu bylo v případě gramatik, budeme i jazyky typu 2 nazývat bezkontextové a jazyky typu 3 regulární.

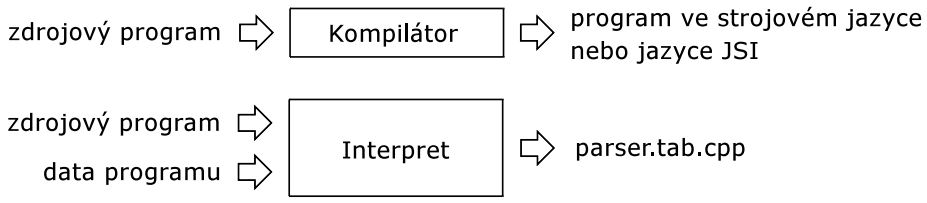


Obrázek 2.1: Chomského hierarchie jazyků

2.2 Překladač

Překladačem označujeme speciální programově realizovaný textový procesor, jenž umožňuje používání programovacího jazyka pro výpočty na počítači. Překladač zpracovává text *zdrojového jazyka* (*zdrojový program*) a převádí ho na sémanticky ekvivalentní text *cílového jazyka* (*cílový program*), nebo interpretuje, tj. provádí zdrojový program a dává výsledky výpočtu. Je-li zdrojovým jazykem vyšší programovací jazyk a cílovým jazykem strojový

jazyk, případně jazyk symbolických instrukcí (JSI), pak takový překladač nazýváme *kompilátorem* a proces překladu *kompilací*. Překladače, které interpretují zdrojový program, nazýváme *interpretační překladače*. Na obrázku 2.2 je schematicky znázorněna funkce kompilátoru a interpretačního překladače.



Obrázek 2.2: Kompilátor a interpretační překladač

2.2.1 Překladače a kompilátory

Překladač: Program, který překládá zdrojový program (napsaný ve zdrojovém jazyce) na ekvivalentní cílový program v cílovém jazyce.

Kompilátor: Program, který překládá program napsaný ve vyšším programovacím jazyce na ekvivalentní program napsaný v nižším programovacím jazyce.

Assembler: Program, který překládá program napsaný ve strojovém jazyce (assembleru) na ekvivalentní strojový kód.

Dekompilátor: Program, který překládá program napsaný v nižším programovacím jazyce na ekvivalentní program napsaný ve vyšším programovacím jazyce.

Disassembler: Program, který překládá strojový kód na ekvivalentní program napsaný ve strojovém jazyce (assembleru).

2.2.2 Fáze překladu a jejich funkce

Z logického i implementačního hlediska je účelné proces kompilace strukturovat do určitých podprocesů, které nazýváme *fáze*. Fáze představuje logicky celistvou operaci transformace jedné reprezentace zdrojového programu na jinou reprezentaci. Při praktické implementaci se potom většina těchto fází vzájemně překrývá. Jednotlivé fáze jsou znázorněny na obrázku 2.3.

Lexikální analýza

První fáze, která se nazývá lexikální analýza, identifikuje ve zdrojovém programu podřetězce znaků, jež představují lexikální jednotky zdrojového jazyka. Lexikální jednotka odpovídá identifikátoru, klíčovému slovu, literálu, oddělovači nebo operátoru. Operace lexikální analýzy provádějí transformaci textu zdrojového programu na posloupnost lexikálních jednotek, přesněji na posloupnost jejich reprezentací. Výstupem této fáze je tedy posloupnost jednotek zvané tokeny. Modely pro lexikální analýzu naleznete v sekci 2.3.

Syntaktická analýza

Syntaktická analýza dostává na vstup reprezentaci lexikálních jednotek a v souladu s definicí syntaxe zdrojového jazyka provádí rozklad zdrojového programu, tj. buduje derivační strom zdrojového programu. Modely pro syntaktickou analýzu naleznete v sekci 2.4.

Sémantická analýza

Sémantická analýza zpracovává především informace, které jsou uvedeny v deklaracích, ukládá je do vnitřních datových struktur a na jejich základě provádí sémantickou kontrolu příkazů a výrazů v programu. Důležitou složkou sémantické analýzy je typová kontrola. Výstupem této fáze je tzv. abstraktní syntaktický strom.

Intermediární generování kódu

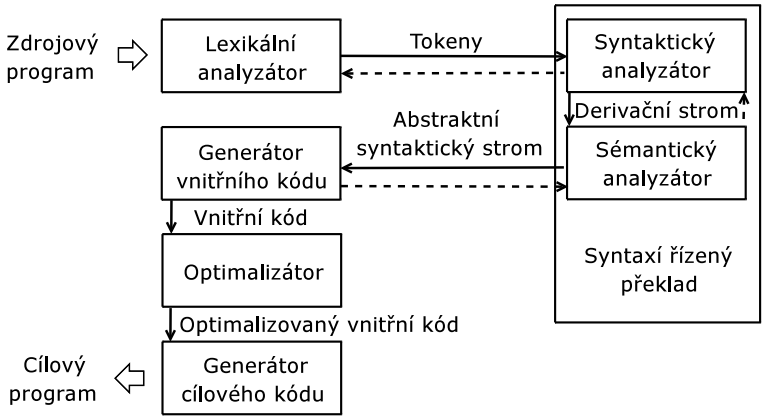
Intermediární generování kódu využívá výsledků syntaktické analýzy a produkuje zdrojovému programu ekvivalentní posloupnost operací. Existuje mnoho způsobů kódování těchto operací; obvykle je každá operace popsána operátorem s několika operandy. Hlavní rozdíl mezi intermediárním kódem a výsledným produktem kompilátoru je v tom, že v intermediárním kódu není ještě specifikováno uložení operandů v paměti ani použití registrů pro implementaci operací. Výstupem této fáze je tedy vnitřní (intermediární) kód.

Optimalizace kódu

Optimalizace kódu je volitelnou fází. Slouží k získání cílového programu, který přestavuje vzhledem k neoptimalizovanému kódu úspory času výpočtu a nebo požadavků na vnitřní paměť. Výsledkem optimalizace kódu je kvalitnější vnitřní (intermediární) kód.

Generování kódu

Závěrečná fáze (generování kódu) již vytváří cílový program. Tato fáze specifikuje paměťová místa pro data, určuje mechanismy zpřístupnění (adresování) těchto dat a vybírá registry, ve kterých se provádí výpočet.



Obrázek 2.3: Fáze překladu

2.3 Modely pro lexikální analýzu

Základními modely pro lexikální analýzu jsou regulární výrazy (v sekci 2.3.1) a konečné automaty 2.3.2.

2.3.1 Regulární výrazy

Lexikální symboly daného programovacího jazyka musí být vytvořeny podle určitých pravidel. Velmi jednoduchým nástrojem, kterým lze požadované vlastnosti lexikálních symbolů zadat, jsou regulární výrazy.

Regulární výraz: Nechť je Σ abeceda. Regulární výraz R nad abecedou Σ je definován následovně

1. $\emptyset$ je regulární výraz označující prázdnou množinu $\emptyset$ (prázdný jazyk)
2. ε je regulární výraz označující jazyk $\{\varepsilon\}$
3. jestliže $a \in \Sigma$, pak a je regulární výraz označující jazyk $\{a\}$
4. jestliže r, s jsou regulární výrazy označující jazyky L_r, L_s, pak
   (a) $r + s$ je regulární výraz označující jazyk $L = L_r \cup L_s$
   (b) rs je regulární výraz označující jazyk $L = L_r L_s$
   (c) r^* je regulární výraz označující jazyk $L = {L_r}^*$

2.3.2 Konečné automaty

Konečné automaty jsou prostředkem pro specifikaci regulárních jazyků. Tyto automaty jsou pro konstrukci kompilátorů velmi důležité, neboť tvoří základ tzv. konečných převodníků, pomocí kterých je realizována lexikální analýza.

Neformálně řečeno, konečný automat má konečnou množinu stavů, v které je specifikován počáteční stav a množina koncových stavů. Pracuje tak, že čte dané slovo nad vstupní abecedou zleva doprava, symbol po symbolu. V prvním kroku je v počátečním stavu, čte nejlevější symbol vstupního slova a přejde do následujícího stavu, v kterém čte druhý symbol, atd., až celé slovo přečte do konce. Automat vstupní slovo akceptuje, jestliže po jeho přečtení skončí v některém koncovém stavu, jinak jej neakceptuje.

Konečný automat: *Konečný automat* M je pětice $M = (Q, \Sigma, R, s, F)$, kde

1. Q je konečná množina stavů
2. Σ je vstupní abeceda
3. R je konečná množina pravidel tvaru: $pa \to q$, kde $p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$
4. $s \in Q$ je počáteční stav
5. $F \subseteq Q$ je množina koncových stavů

Konfigurace konečného automatu: Řetězec qx, kde $q \in Q$, $x \in \Sigma^*$ nazýváme *konfigurací konečného automatu* M.

Přechod konečného automatu: Nechť pax a px jsou dvě konfigurace konečného automatu M, kde $p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$. Nechť $r = pa \to q \in R$ je pravidlo. Potom M může provést *přechod* z pax do qx za použití r; zapsáno $pax \vdash qx\,[r]$ nebo zjednodušeně $pax \vdash qx$.

Sekvence přechodů: Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat.

1. Nechť χ je konfigurace. M provede *nula přechodů* z χ do χ; zapisujeme $\chi \vdash^0 \chi\,[\varepsilon]$ nebo zjednodušeně $\chi \vdash^0 \chi$.
2. Nechť $\chi_0, \chi_1, \ldots, \chi_n$ je sekvence přechodů konfigurací $n \geq 1$ a $\chi_{i-1} \vdash \chi_i\,[r_i]$, kde $r_i \in R$ pro všechna $i = 1, \ldots, n$, což znamená

$$\chi_0 \vdash \chi_1\,[r_1] \vdash \chi_2\,[r_2] \vdash \cdots \vdash \chi_n\,[r_n]\,,$$

   pak M provede n-přechodů z χ_0 do χ_n; zapisujeme $\chi_0 \vdash^n \chi_n\,[r_1, \ldots, r_n]$.
3. Pokud $\chi_0 \vdash^n \chi_n\,[\rho]$ pro $n \geq 1$, zapisujeme $\chi_0 \vdash^+ \chi_n\,[\rho]$.
4. Pokud $\chi_0 \vdash^n \chi_n\,[\rho]$ pro $n \geq 0$, zapisujeme $\chi_0 \vdash^* \chi_n\,[\rho]$.

Jazyk přijímaný konečným automatem: Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat. Jazyk přijímaný konečným automatem M, $L(M)$ je definován

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

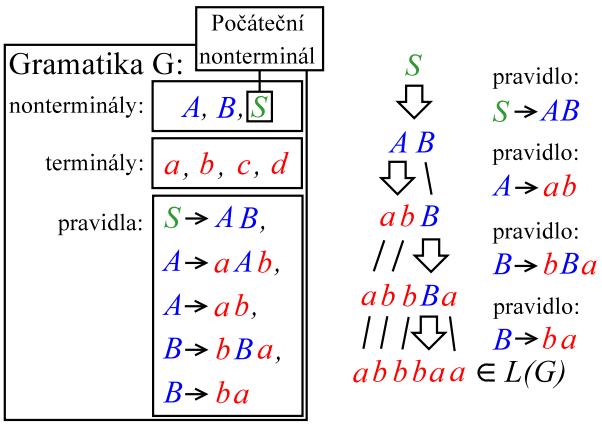
2.4 Modely pro syntaktickou analýzu

Základními modely pro syntaktickou analýzu jsou bezkontextové gramatiky (v sekci 2.4.1), zásobníkové automaty (v sekci 2.4.2) a rozšířené zásobníkové automaty.

2.4.1 Bezkontextové gramatiky

Syntaxe programovacích jazyků se nejčastěji definuje pomocí bezkontextových gramatik. Tyto gramatiky úzce souvisí se zásobníkovými automaty a rozšířenými zásobníkovými automaty, které tvoří teoretický model syntaktických analyzátorů. Platí totiž, že daný jazyk je bezkontextový, právě když jej lze akceptovat vhodným zásobníkovým automatem. Bezkontextové gramatiky jsou definovány v sekci 2.1.2 – Chomského klasifikace gramatik.

Bezkontextová gramatika: Nechť $G = (N, T, P, S)$ je bezkontextová gramatika.



Obrázek 2.4: Ilustrace bezkontextové gramatiky

Přímá derivace: Nechť $u, v \in (N \cup T)^*$ a $p = A \to x \in P$ je pravidlo. Pak říkáme, že uAv *přímo derivuje* uxv podle pravidla p, a zapisujeme $uAv \Rightarrow uxv$.

Nejlevější derivace: Nechť $u \in T^*$, $v \in (N \cup T)^*$ a $p = A \to x \in P$ je pravidlo. pak říkáme, že uAv *přímo derivuje v nejlevější derivaci* uxv podle pravidla p a zapisujeme $uAv \Rightarrow_{lm} uxv\,[p]$ nebo zkráceně $uAv \Rightarrow_{lm} uxv$.

Nejpravější derivace: Nechť $u \in (N \cup T)^*$, $v \in T^*$ a $p = A \to x \in P$ je pravidlo. pak říkáme, že uAv *přímo derivuje v nejpravější derivaci* uxv podle pravidla p a zapisujeme $uAv \Rightarrow_{rm} uxv\,[p]$ nebo zkráceně $uAv \Rightarrow_{rm} uxv$.

Derivace v 0-krocích: Nechť $u \in (N \cup T)^*$. Pak říkáme, že u *derivuje v 0-krocích* u a zapisujeme $u \Rightarrow^0 u\,[\varepsilon]$ nebo zkráceně $u \Rightarrow^0 u$.

Derivace v n-krocích: Nechť $u_0, u_1, \ldots, u_n \in (N \cup T)^*$ a nechť pro všechna $i = 1, \ldots, n$ platí $u_{i-1} \Rightarrow u_i\,[p_i]$. Pak říkáme, že u_0 *derivuje v n-krocích* u_n a zapisujeme $u_0 \Rightarrow^n n\,[p_1 p_2 \ldots p_n]$ nebo zkráceně $u_0 \Rightarrow^n u_n$.

Netriviální derivace: Nechť $u \Rightarrow^n v\,[\pi]$ pro $n \geq 1$; $u, v \in (N \cup T)^*$. Pak říkáme, že u *netriviálně derivuje* v a zapisujeme $u \Rightarrow^+ v\,[\pi]$ nebo zkráceně $u \Rightarrow^+ v$.

Derivace: Nechť $u \Rightarrow^n v\,[\pi]$ pro $n \geq 0$; $u, v \in (N \cup T)^*$. Pak říkáme, že u *derivuje* v a zapisujeme $u \Rightarrow^* v\,[\pi]$ nebo zkráceně $u \Rightarrow^* v$.

Generovaný jazyk: Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. *Jazyk generovaný bezkontextovou gramatikou* G, $L(G)$, je definován $L(G) = \{w : w \in T^*, S \Rightarrow^* w\}$.

Derivační strom

Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Strom je derivační strom v G, jestliže

1. každý uzel je ohodnocen symbolem z $N \cup T$,
2. kořen je ohodnocen symbolem S,
3. jestliže uzel má nejméně jednoho následovníka, pak je ohodnocen symbolem z N,
4. jestliže $b_1, b_2, \ldots, b_k$ jsou přímý následovníci uzlu a, jenž je ohodnocen symbolem A, zleva doprava s ohodnocením $B_1, B_2, \ldots, B_k$, pak $A \to B_1 B_2 \ldots B_k \in P$.

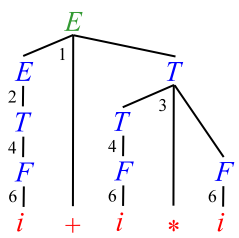
Ke každé vytvořené derivaci v bezkontextové gramatice G můžeme tedy vytvořit odpovídající derivační strom.

Příklad: $G = (N, T, P, E)$, kde $N = \{E, F, T\}$, $T = \{i, +, *, (,)\}$, $P = \{1 : E \to E{+}T$, $2 : E \to T$, $3 : T \to T{*}F$, $4 : T \to F$, $5 : F \to (E)$, $6 : F \to i\}$.

Pravá derivace:

$$\begin{aligned}
\underline{E} &\Rightarrow_{rm} E + T && [1]\\
&\Rightarrow_{rm} E + T * F && [3]\\
&\Rightarrow_{rm} E + T * i && [6]\\
&\Rightarrow_{rm} E + F * i && [4]\\
&\Rightarrow_{rm} E + i * i && [6]\\
&\Rightarrow_{rm} T + i * i && [2]\\
&\Rightarrow_{rm} F + i * i && [4]\\
&\Rightarrow_{rm} i + i * i && [6]
\end{aligned}$$

Derivační strom:



Nejednoznačnost bezkontextových gramatik

Nejednoznačnost dané bezkontextové gramatiky způsobuje při syntaktické analýze značné praktické potíže. Tato vlastnost totiž umožňuje odvodit z určitého řetězce generovaného jazyka různé derivační stromy.

Nejednoznačná bezkontextová gramatika: Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Pokud existuje řetězec $x \in L(G)$ s více jak jedním derivačním stromem, potom G je *nejednoznačná*.

Jazyk L je *vnitřně nejednoznačný*, pokud L není generován žádnou jednoznačnou gramatikou.

2.4.2 Zásobníkové automaty

Podobně jako konečný automat, má i zásobníkový automat konečně stavovou řídící jednotku a na vstupní pásce má zapsané vstupní slovo, které čte pomocí čtecí hlavičky. Navíc je však vybaven (potencionálně nekonečnou) zásobníkovou pamětí, jejíž vrchol ovlivňuje každý přímý přechod.

Zásobníkový automat: Zásobníkový automat je sedmice $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde

1. Q je konečná množina stavů
2. Σ je vstupní abeceda
3. Γ je zásobníková abeceda
4. R je konečná množina pravidel tvaru $Apa \to wq$, kde $A \in \Gamma$, $p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Gamma^*$
5. $s \in Q$ je počáteční stav
6. $S \in \Gamma$ je počáteční symbol na zásobníku
7. $F \subseteq Q$ je množina koncových stavů

Jazyk přijímaný zásobníkovým automatem M přechodem do koncového stavu, značen jako $L(M)_f$, je definován

$$L(M)_f = \{w : w \in \Sigma^*, Ssw \vdash^* zf, z \in \Gamma^*, f \in F\}.$$

Jazyk přijímaný zásobníkovým automatem M vyprázdněním zásobníku, značen jako $L(M)_\varepsilon$, je definován

$$L(M)_\varepsilon = \{w : w \in \Sigma^*, Ssw \vdash^* zf, z = \varepsilon, f \in Q\}.$$

Jazyk přijímaný zásobníkovým automatem M přechodem do koncového stavu a vyprázdněním zásobníku, značen jako $L(M)_{f\varepsilon}$, je definován

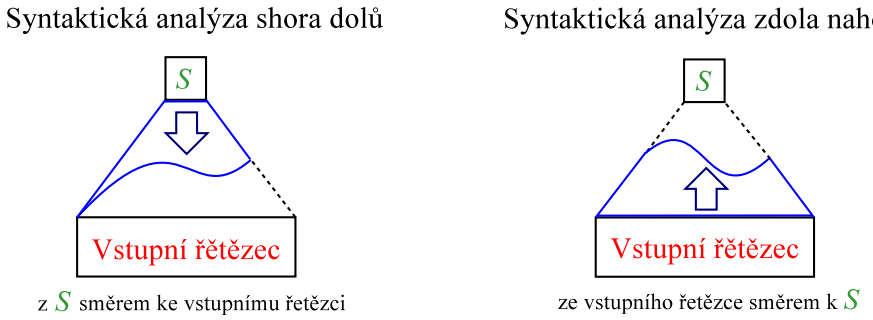
$$L(M)_{f\varepsilon} = \{w : w \in \Sigma^*, Ssw \vdash^* zf, z = \varepsilon, f \in F\}.$$

2.4.3 Rozšířený zásobníkový automat

V zásobníkovém automatu je v každém přechodu nahrazen právě jeden symbol (vrchol) zásobníku. V rozšířeném zásobníkovém automatu v jednom přechodu bude možné nahradit celý řetězec symbolů uložených v zásobníku.

2.4.4 Základní přístupy k syntaktické analýze

Obecně můžeme rozlišit dva způsoby tvorby derivačního stromu. Buď syntaktický analyzátor vytváří strom postupně od kořene k jeho listům (budeme nazývat syntaktickou analýzou shora dolů a budeme simulovat pomocí zásobníkového automatu) nebo naopak syntaktický analyzátor vytváří strom postupně od jeho listů ke kořeni (budeme nazývat syntaktickou analýzou zdola nahoru a budeme simulovat pomocí rozšířeného zásobníkového automatu). Oba případy jsou ilustrovány na obrázku 2.5.



Obrázek 2.5: Dva základní přístupy k syntaktické analýze

Simulace syntaktické analýzy shora dolů

1. Vstup: bezkontextová gramatika $G = (N, T, P, S)$
2. Výstup: zásobníkový automat $M = (Q, \Sigma, \Gamma, R, s, S, F)$, $L(G) = L(M)_{\varepsilon}$
3. Metoda:

   $Q = \{s\}$;
   $\Sigma = T$;
   $\Gamma = N \cup T$;
   Konstrukce množiny R:
   for each $a \in \Sigma$: přidej $aSa \to s$ do R;
   for each $A \to x \in P$: přidej $aS \to ys$ do R, kde $y = \text{reversal}(x)$;
   $F = \emptyset$;

Simulace syntaktické analýzy zdola nahoru

1. Vstup: bezkontextová gramatika $G = (N, T, P, S)$
2. Výstup: zásobníkový automat $M = (Q, \Sigma, \Gamma, R, s, \#, F)$, $L(G) = L(M)_{f}$
3. Metoda:

   $Q = \{s, f\}$;
   $\Sigma = T$;
   $\Gamma = N \cup T \cup \{\#\}$;

Konstrukce množiny R:

for each $a \in \Sigma$: přidej $sa \to as$ do R;

for each $A \to x \in P$: přidej $xs \to As$ do R;

přidej $\#Ss \to f$ do R;

$F = \{f\}$;

2.4.5 Překladová párová gramatika

Překlad: Nechť Σ a Δ jsou abecedy. Abecedu Σ nazvěme *vstupní abecedou*, Δ *výstupní abecedou*. *Překladem* jazyka $L_1 \subset \Sigma^*$ do jazyka $L_2 \subset \Delta^*$ nazveme relaci $TRAN : L_1 \to L_2$. Je-li $[x, y] \in TRAN$, pak řetězec y nazýváme *výstupem* pro řetězec x.

Překladová párová gramatika: Překladová párová gramatika je pětice $V = (N, \Sigma, \Delta, P, S)$, kde

1. N je konečná množina nonterminálních symbolů
2. Σ je konečná vstupní abeceda
3. Δ je konečná výstupní abeceda
4. P je množina přepisovacích pravidel
5. $S \in N$ je startovací (nonterminální) symbol

Pravidla mají tvar $A \to \alpha, \beta$, kde $A \in N$, $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$ a nonterminály v řetězci β jsou permutací nonterminálů z řetězce α.

Nechť $V = (N, \Sigma, \Delta, P, S)$ je překladová párová gramatika, přičemž $N \cap \Delta = \emptyset$ a množina P obsahuje pouze pravidla tvaru

$$A \to x_0 B_1 x_1 B_2 \dots B_k x_k y_0 B_1 y_1 B_2 \dots B_k y_k$$

pro $x_i \in \Sigma^*$, $y_i \in \Delta^*$, $0 \leq i \leq k$. Pak *překladová gramatika* G_V příslušející gramatice V je pětice $G_V = (N, \Sigma, \Delta, P', S)$, kde množina P' obsahuje pouze pravidla ve tvaru

$$A \to x_0 y_0 B_1 x_1 y_1 B_2 \dots B_k x_k y_k$$

odvozená z původních pravidel.

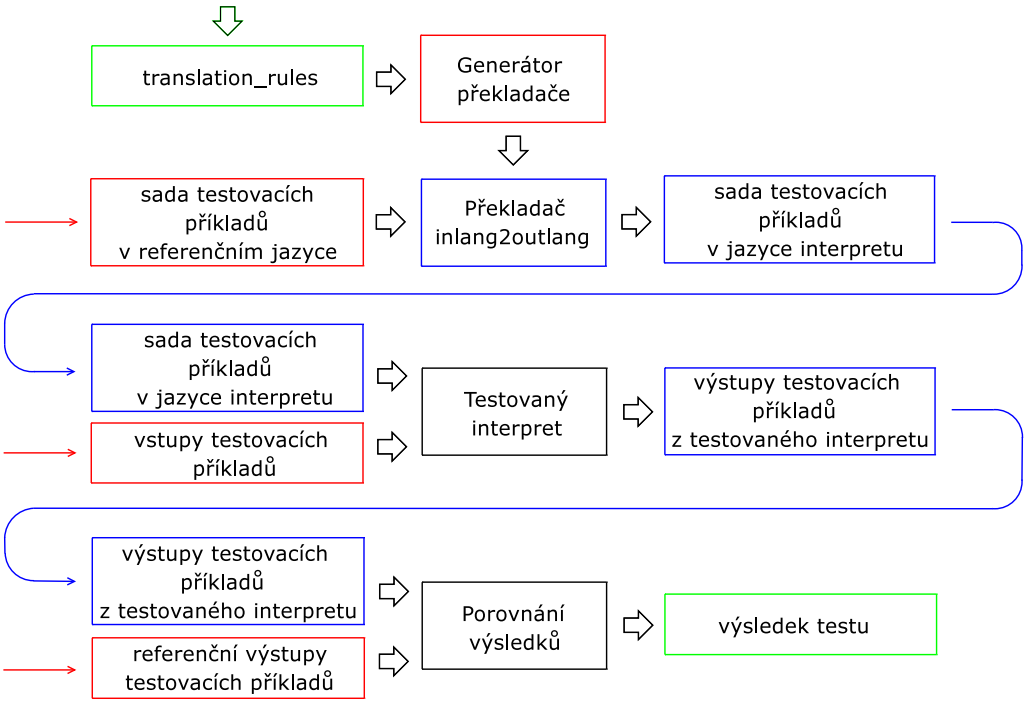
Konkrétní příklad použití překladové párové gramatiky je v následující kapitole na obrázku 3.6.

Kapitola 3

Návrh testovacího systému

Systém pro testování interpretů obsahuje nástroje pro specifikaci jazyka interpretu a pro vytvoření překladače referenčního jazyka na jazyk interpretu, sadu testovacích programů v referenčním jazyce, včetně jejich vstupů a odpovídajících výstupů a nástroj pro automatické testování.

Schéma testovacího systému je na obrázku 3.1. Uživatel specifikuje syntaxi jazyka do souboru translation_rules, podle kterého generátor překladače vytvoří překladač inlang2outlang z referenčního jazyka do jazyka testovaného interpretu. O návrhu tohoto překladače se dočtete v sekci 3.1. Tímto překladačem lze přeložit sadu testovacích programů do jazyka testovaného interpretu. Interpretu se na vstup dá program v jeho jazyce a vstupy programu. Výsledek řešení interpretem – výstup – je pak porovnáván s výstupy referenčního programu. Pokud se shodují, test proběhl úspěšně. O testovacím systému se dočtete v kapitole 5.



Obrázek 3.1: Schéma testovacího systému

3.1 Návrh generátoru překladače

3.1.1 Popis syntaxe jazyka

Při implementaci překladače se obvykle používá jednoho ze dvou základních přístupů – překladu shora dolů nebo zdola nahoru. Tyto názvy odpovídají postupu při vytváření derivačního stromu.

Syntaktická analýza shora dolů

Syntaktický analyzátor začíná ze startovacího symbolu gramatiky a snaží se jej převést na terminální symboly (odpovídající vstupu) postupnou expanzí nonterminálních symbolů. Překlad shora dolů se dá popsat jako proces vytváření derivačního stromu počínaje jeho kořenem. Parser analyzuje vstup zleva doprava a konstruuje nejlevější derivaci. Tomuto přístupu odpovídá třída LL gramatik, která se používá pro analyzátory implementované ručně.

Syntaktická analýza zdola nahoru

Syntaktický analyzátor začíná z terminálních symbolů ze vstupu a snaží se jej redukovat na startovací symbol gramatiky. Překlad zdola nahoru se dá popsat jako proces budování derivačního stromu ze vstupního řetězce směrem ke kořenu. Jedním ze způsobů syntaktické analýzy zdola nahoru je LR analýza. Parser analyzuje vstup zleva doprava a vytváří se pravá derivace v opačném pořadí.

Výhodou tohoto přístupu je, že analyzátory LR lze vytvořit k rozpoznání téměř všech možných konstrukcí programovacích jazyků, které jsou syntakticky definovány bezkontextovými gramatikami. Jedná se tedy o obecnější přístup než je tomu u LL analyzátoru.

Nevýhodou je značná náročnost manuální konstrukce LR analyzátoru pro gramatiky typických programovacích jazyků. Je však možné využít generátor LR analyzátorů, např. Bison, což se stává oproti LL analyzátoru opět výhodou.

3.1.2 Použití generátorů syntaktické a lexikální analýzy

Pokud chceme vytvořit LR syntaktický analyzátor, můžeme použít některý z generátorů parserů. Pro potřeby tohoto testovacího systému jsem vybrala program GNU Bison, který je dostupný na školním serveru merlin.fit.vutbr.cz a je v základní nabídce balíčků většiny distribucí GNU Linux. Parser vygenerovaný programem Bison vyžaduje lexikální analyzátor, pro který použiji generátor lexikálních analyzátorů GNU Flex. O generátoru parserů Bison a generátoru lexikálních analyzátorů Flex se dočtete níže.

Flex

Flex[1] je generátor lexikálních analyzátorů. Podle zadaných pravidel vytvoří kód v jazyce C a uloží ho do hlavičkového souboru. Tento soubor obsahuje postup pro stavový automat, který provádí lexikální analýzu. O použití překladače Flex jsem se dočetla na manuálových stránkách programu[3][4].

Flex dostane na vstup program v jazyce Flexu. Tento program se skládá ze tří částí. První část slouží pro deklaraci proměnných, pojmenovaných konstant a regulárních definic. Další oddíl obsahuje vlastní definici lexikální struktury jazyka a činnosti analyzátoru

[1] celý název: Flex - fast lexical analyzer generator

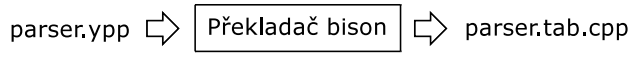
formou překladových pravidel. Poslední část obsahuje pomocné procedury a akce zapsané v jazyce C. Z tohoto oddílu Flex pouze kopíruje veškerý text do výstupního souboru. Znázornění překladu programu napsaného v jazyce Flexu (scanner.l) do jazyka C (lex.yy.c) je na obrázku 3.2.

Obrázek 3.2: Znázornění vytvoření souboru s kódem pro lexikální analýzu

Bison

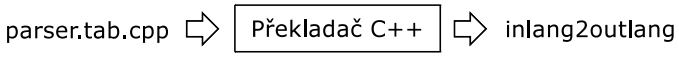
Bison[2] je generátor LR syntaktických analyzátorů. Z popisu bezkontextové gramatiky vytvoří kód v jazyce C (případně C++) a uloží ho do hlavičkového souboru. Tento soubor obsahuje příslušný postup pro stavový automat, který simuluje vytváření derivačního stromu. O použití překladače Bison jsem se dočetla na manuálových stránkách programu[5][2].

Nejprve je třeba připravit zdrojový program v jazyce Bisonu. Uvedený program má tři části podobně jako program napsaný v jazyce Flexu. Do deklarační části se přidá připojení knihovny obsahující lexikální analyzátor. Prostřední část obsahuje překladová pravidla bezkontextové gramatiky. Každé takové pravidlo může obsahovat sémantickou akci (čehož využijeme pro překlad na výstupní jazyk). Znázornění překladu programu napsaného v jazyce Bisonu (parser.ypp) do jazyka C++ (parser.tab.cpp) je na obrázku 3.3.



Obrázek 3.3: Znázornění vytvoření souboru s kódem pro syntaktickou analýzu

Z takto vygenerovaného kódu v jazyce C++ je už jednoduché vytvořit spustitelný syntaktický analyzátor (inlang2outlang) pomocí překladače jazyka C++, např. g++. Znázornění tohoto překladu je na obrázku 3.4.



Obrázek 3.4: Znázornění vytvoření spustitelného syntaktického analyzátoru

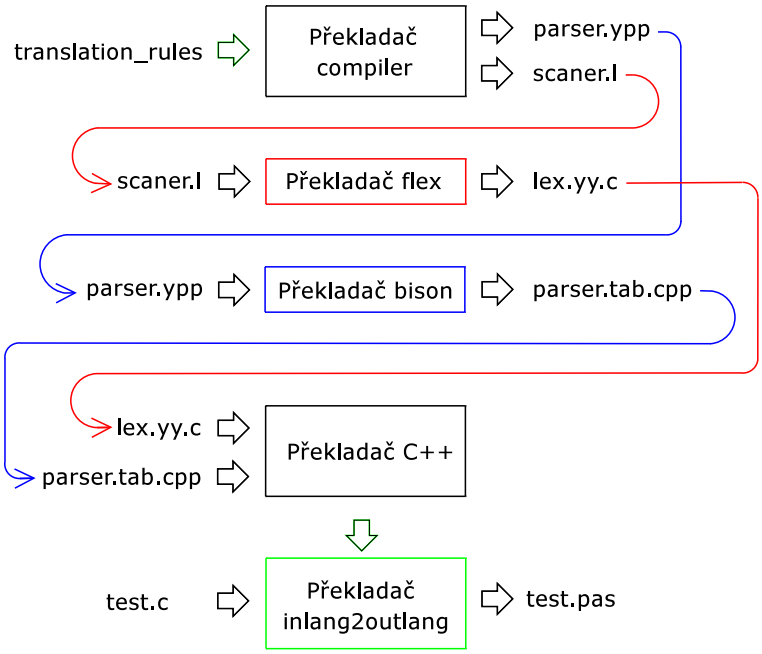
3.1.3 Překladač programu napsaného v mém jazyce

Třetím stavebním kamenem – prvotně funkčním – je překladač *compiler*, který přeloží program napsaný v mém jazyce do jazyka Flexu a Bisonu. Program napsaný v mém jazyce obsahuje pravidla pro přeložení vstupního jazyka testovacího systému na jazyk výstupní. Analýzou jednotlivých pravidel má být *compiler* schopný přepsat je na pravidla v jazyce Bisonu.

[2]celý název: Bison - GNU Project parser generator (yacc replacement)

Zvažovala jsem možnost, že by uživatel klíčová slova vstupního jazyka zadal do souboru, který by dostal *compiler* na vstup spolu s programem s pravidly. Tato klíčová slova by se pak přidala při generování programu v jazyce Flexu. Avšak nabídla se druhá varianta, kdy se překladač „učí“ klíčová slova analýzou pravidel (co není nonterminál, je klíčové slovo). Tato varianta se mi zdála pro uživatele testovacího systému přívětivější.

Schéma, které znázorňuje postup vytvoření překladače vstupního jazyka testovacího systému na výstupní, je na obrázku 3.5. *Compiler* dostane na vstup soubor s překladovými pravidly translation_rules. Pravidla analyzuje, zpracuje a jeho výstupem jsou dva soubory – scaner.l v jazyce Flexu a parser.ypp v jazyce Bisonu. Soubor scanner.l se přeloží překladačem Flex na program v jazyce C – soubor lex.yy.c. Soubor parser.ypp se přeloží překladačem Bison na program v jazyce C++ parser.tab.cpp. Parser.tab.cpp má jako knihovnu připojený kód lexikálního analyzátoru lex.yy.c. Tyto programy se přeloží kompilátorem jazyka C++. Tím vznikne výsledný spustitelný soubor inlang2outlang, který je překladačem vstupního jazyka testovacího systému na výstupní podle pravidel zapsaných v souboru translation_rules (na obrázku je je příklad překladu z jazyka C do jazyka Pascal).



Obrázek 3.5: Schéma postupu vytvoření překladače

3.1.4 Překlad pomocí dvou gramatik

Jasným příkladem překladu je přepis infixového zápisu aritmetického výrazu na postfixový. Stejně jako tomu bylo u syntaktické analýzy, jsou i zde dva možné přístupy – prostřednictvím gramatiky nebo automatu.

Překladová párová gramatika je založena na dvou vzájemně spojených bezkontextových gramatikách. První z nich, tzv. vstupní gramatika, popisuje jazyk tvořený všemi větami zdrojového jazyka $L1$. Druhá, výstupní gramatika $L2$, popisuje jazyk tvořený všemi výstupy pro řetězce jazyka $L1$. Tento přístup je v testovacím systému využit pro specifikaci gramatiky jazyka interpretu.

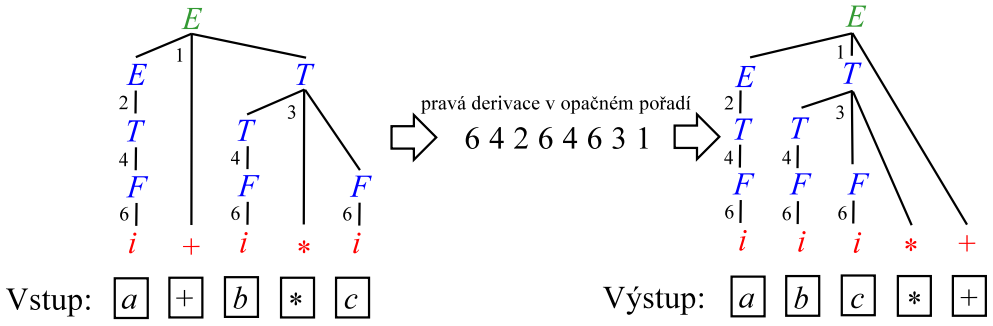
Druhý přístup ke specifikaci překladu využívá překladového automatu, který je získán rozšířením konečného nebo zásobníkového automatu o výstupní pásku a výstupní funkci, která předepisuje výstup automatu. Překlad definovaný překladovým automatem je množina dvojic řetězců $[x, y]$ takových, že automat přijme řetězec x a na výstup vyšle řetězec y. Překladač *compiler* převede překladovou párovou gramatiku specifikovanou ve vstupním souboru na pravidla v jazyce Bisonu. Přepis na výstupní řetězec je zapsán v části pro sémantické akce. Výsledný stavový automat při simulaci derivačního stromu (analýza vstupního řetězce) na zásobník vkládá řetězce přepsané podle pravidel výstupního jazyka.

Příklad: Překlad pomocí dvou gramatik

Pravidla vstupní gramatiky	Pravidla výstupní gramatiky	Sémantická akce
$1 : E \Rightarrow E{+}T$	$1 : E \Rightarrow ET{+}$	přepiš na $ET+$
$2 : E \Rightarrow T$	$2 : E \Rightarrow T$	přepiš na T
$3 : T \Rightarrow T{*}F$	$3 : T \Rightarrow TF{*}$	přepiš na $TF*$
$4 : T \Rightarrow F$	$4 : T \Rightarrow F$	přepiš na F
$5 : F \Rightarrow (E)$	$5 : F \Rightarrow E$	přepiš na E
$6 : F \Rightarrow i$	$6 : F \Rightarrow i$	přepiš na i

Pravá derivace:

$$
\begin{aligned}
\underline{E} &\Rightarrow_{rm} E{+}\underline{T} && [1]\\
&\Rightarrow_{rm} E{+}T{*}\underline{F} && [3]\\
&\Rightarrow_{rm} E{+}\underline{T}{*}i && [6]\\
&\Rightarrow_{rm} E{+}\underline{F}{*}i && [4]\\
&\Rightarrow_{rm} \underline{E}{+}i{*}i && [6]\\
&\Rightarrow_{rm} \underline{T}{+}i{*}i && [2]\\
&\Rightarrow_{rm} \underline{F}{+}i{*}i && [4]\\
&\Rightarrow_{rm} i{+}i{*}i && [6]
\end{aligned}
$$



Obrázek 3.6: Derivační strom

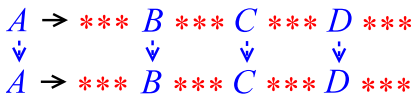
3.1.5 Analýza pravidel

Vstupní řetězec lze přeložit na výstupní podle různých modelů. Jejich výhody a nevýhody objasním na následujících řádcích.

Model 1: Při analýze podle vstupního pravidla jsou všechny nonterminály na výstup přepisovány právě v takovém pořadí, v jakém byly přijaty podle vstupního pravidla. Tento model plně odpovídá modelu párových gramatik, avšak oproti následujícímu modelu, je málo obecný.

Vstup: $< \text{stat} > \rightarrow \text{if} < \text{expr} > < \text{stat} > \text{else} < \text{stat} >$

Výstup: $< \text{stat} > \rightarrow \text{IF} < \text{expr} > \text{THEN} < \text{stat} > \text{ELSE} < \text{stat} >$



Obrázek 3.7: Ilustrace analýzy podle modelu 1

Model 2: Ve vstupním i výstupním pravidle musí být stejný počet odpovídajících si nonterminálů. Nonterminály jsou odlišeny názvem, aby bylo možné je přepsat na výstup v pořadí různém od pořadí na vstupu. Pak je možné například přepsat příkaz if na příkaz ifnot.

Vstup: $< \text{stat} > \rightarrow \text{if} < \text{expr} > < \text{stat\_1} > \text{else} < \text{stat\_2} >$

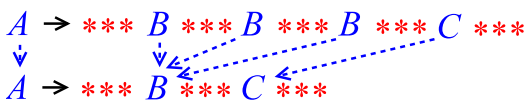
Výstup: $< \text{stat} > \rightarrow \text{IFNOT} < \text{expr} > \text{THEN} < \text{stat\_2} > \text{ELSE} < \text{stat\_1} >$

Obrázek 3.8: Ilustrace analýzy podle modelu 2

Model 3: Oba předchozí modely však nejsou dostatečně obecné, protože neumožňují např. přepis příkazu for z jazyka C do jazyku Pascal a naopak. V příkazu for v jazyce C se tentýž identifikátor vyskytuje třikrát, přitom v jazyce Pascal je odpovídající identifikátor pouze jednou. Proto je nutné zavést kopírování (případně redukci) nonterminálů. Odpovídající nonterminál se pak přepíše na výstup jednou až n-krát podle toho, jak to pravidlo vyžaduje. Tento model je použitý při návrhu jazyka a implementaci překladače *compiler*.

Vstup: $< \text{stat} > \rightarrow \text{for(} < \text{id} > = < \text{expr\_1} > \text{;} < \text{id} > <= < \text{expr\_2} > \text{;} < \text{id} > \text{++)} < \text{stat} >$

Výstup: $< \text{stat} > \rightarrow \text{FOR} < \text{id} > := < \text{expr\_1} > \text{TO} < \text{expr\_2} > \text{DO} < \text{stat} >$



Obrázek 3.9: Ilustrace analýzy podle modelu 3

3.2 Jazyk pro popis jazyka interpretu

Podoba jazyka pro popis jazyka interpretu je odvislá od návrhu generátoru překladače. Tímto jazykem jsou popsána pravidla LR gramatiky vstupního jazyka a jim odpovídající pravidla výstupní gramatiky. Příklady převodu gramatiky na LR můžete najít například na webových stránkách *School of Computer Science, University of Manchester – Converting grammars to LR*[11], kde jsem se sama inspirovala.

Zápis pravidla v mém jazyce má tvar:

<levá strana pravidla>`:`*<pravá strana vstupního pravidla>*`$`*<pravá strana výstupního pravidla>*

přičemž levá strana pravidla je název nonterminálu (např. stat) a pravé strany obsahují klíčová slova daných jazyků a nonterminály ohraničené závorkami '<' a '>'. Na pravé straně vstupního pravidla jsou klíčová slova od sebe oddělena mezerami. Překladač *compiler* podle nich vytváří tabulku klíčových slov. Na pravé straně výstupního pravidla jsou mezery tam, kde klíčová slova a nonterminály potřebují být oddělena mezerami.

```
stat: if ( <expr> ) <stat> $ IF <expr> THEN <stat>
```

Při použití pravidla s více nonterminály stejného typu (výraz, příkaz), např. příkaz if-else, je třeba tyto nonterminály odlišit a doplnit pravidla pro jejich přepis.

```
stat: if ( <expr> ) <stat_1> else <stat_2> $ IF <expr> THEN <stat_1> ELSE <stat_2>
stat_1: <stat> $ <stat>
stat_2: <stat> $ <stat>
```

Tento jazyk připouští celořádkové komentáře, uvozené znaky „//“. Znaky, které jsou použité pro oddělení jednotlivých částí pravidel (tedy ':' a '$') a závorky '<' a '>' sloužící pro označení nonterminálu lze do pravidla vložit přes escape sekvenci:

`"\D"` je přepsáno na `':'`,

`"\S"` je přepsáno na `'$'`,

`"\<"` je přepsáno na `'<'`,

`"\>"` je přepsáno na `'>'`.

Protože některé skriptovací jazyky využívají pro ukončení příkazu odřádkování (například interpret Bash použitý při ověření funkčnosti testovacího systému), bylo nutné do pravidel pro překlad povolit značky formátování výstupního kódu (v levé straně pravidla výstupního jazyka). Pro odřádkování je použita sekvence znaků „\X“[3]. Dá se toho využít i při odřádkování připojení knihoven v jazyce C.

```
stat: if ( <expr> ) <stat> $ if [ <expr> ] ; then \X <stat> \X fi
```

Sekvence znaků „\X“ se dá použít i pro zvýšení přehlednosti výstupního kódu například v jazyce Pascal.

```
stat: if ( <expr> ) <stat> $ IF <expr> THEN \X <stat>
```

[3]Sekvence znaků „\n“ není použita z důvodu přehlednějšího zpracování řetězců překladačem *compiler*

Kapitola 4

Implementace překladače

Překladač mého jazyka pro popis syntaxe a sémantiky jazyka interpretu do jazyka generátoru LR syntaktické analýzy Bison je implementován v jazyce C++. Pro použití některých tříd, které jsou součástí knihoven C++, jsem čerpala informace z webových stránek *C++ reference*[10], kde jsou popsány význam a syntaxe těchto tříd a jejich metod. Činnost programu *compiler*, důležité třídy, metody a funkce jsou popsány v následujících sekcích.

4.1 Činnost překladače

Překladač *compiler* musí ze vstupního souboru s pravidly popisujícími LR gramatiku vygenerovat soubory pro Flex a pro Bison. Začne otevřením vstupního souboru pro čtení a předá tento soubor funkci readRulesFromFile() – popsána v sekci 4.1.1, která naplní vektor objektů třídy rule (vPairedRules) – popsána v sekci 4.3, pravidly rozdělenými na levé strany pravidel, pravé strany pravidel vstupního jazyka a pravé strany pravidel výstupního jazyka.

Následně je nad každým objektem z vektoru vPairedRules volána metoda createRuleLineForBison() – popsána v sekci 4.3.1, která naplní atribut LineForBison pravidlem pro Bison. Z takto zpracovaného vektoru vPairedRules již lze generovat soubory pro vstup generátoru lexikálních analyzátorů Flex – popsáno v sekci 4.1.4, a generátoru LR syntaktických analyzátorů Bison – popsáno v sekci 4.1.5. Program má tedy dva výstupní soubory, scanner.l a parser.ypp.

4.1.1 Funkce readRulesFromFile()

Tato funkce načítá po řádku pravidla ze vstupního souboru. Řádek rozdělí funkcí isolateParts() – v sekci 4.1.2. Levou stranu pravidla, pravou stranu vstupního pravidla a pravou stranu výstupního pravidla vloží do objektu třídy rule (popsána v sekci 4.3) a objekt přidá do vektoru. Takto plní vektor pro každý řádek s pravidlem ze vstupního souboru.

4.1.2 Funkce isolateParts()

Funkce isolateParts() dostane na vstup řetězec – jeden řádek ze vstupního souboru, který zpracuje. Pokud je řádek prázdný (obsahuje pouze mezery, „\f\n\r\t\v “) nebo obsahuje celořádkový komentář (uvozený „//“), pak funkce vrátí hodnotu 1 (ve funkci readRulesFromFile(), v sekci 4.1.1, je pak řádek vynechán). Vstupní řetězec rozdělí na tři dílčí řetězce podle znaků ':' a '$' v tomto pořadí. Pokud se rozdělení nezdaří, funkce skončí chybou (návratová hodnota -1). Dílčí řetězce upraví pomocí funkce cutOffSpaces(), která odstraní

mezery na začátku a na konci řetězce, a vloží každý z řetězců do vektoru řetězců, na nějž funkce dostala referenci.

4.1.3 Funkce fillTableOfKeyWords()

Funkce dostane na vstup referenci na tabulku klíčových slov reprezentovanou vektorem objektů třídy keyWords (třída keyWords je popsána v sekci 4.2). Úkolem funkce fillTableOfKeyWords() je tuto tabulku předvyplnit pro alespoň částečné zpřehlednění pravidel vygenerovaných pro Bison.

Tokeny jsou vytvořeny pro středník ';' a závorky '(' a ')', '<' a '>', '{' a '}'. O další doplnění tabulky se stará metoda třídy rule utillizeKeyWord() – v sekci 4.3.4.

4.1.4 Funkce createOutForFlex()

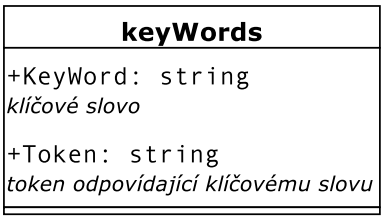
Funkce createOutForFlex() vytvoří soubor scanner.l ve formátu pro vstup generátoru lexikálních analyzátorů Flex. Do tohoto souboru jsou zapsána pravidla, podle kterých vygenerovaný lexikální analyzátor vrátí klíčová slova jako odpovídající tokeny (podle vektoru klíčových slov) a ostatní lexémy jako token IDENT.

4.1.5 Funkce createOutForBison()

Tato funkce vytvoří soubor parser.ypp ve formátu pro vstup generátoru LR syntaktických analyzátorů Bison. Do souboru parser.ypp se zapíší pravidla tak, jak je přepsala metoda třídy rule createLineForBison() – v sekci 4.3.1 – pro každé pravidlo. Jednotlivá pravidla jsou ve vektoru objektů třídy rule, na nějž funkce dostala referenci na vstup.

4.2 Třída keyWords

Třída keyWords má dva veřejné atributy, jak můžete vidět na UML diagramu, obrázek 4.1. Atribut KeyWord obsahuje klíčové slovo jazyka (řetězec) a atribut Token obsahuje token odpovídající klíčovému slovu (rovněž řetězec).



Obrázek 4.1: UML diagram třídy keyWords

Příklad použití:

```
keyWords word;                                              pro klíčové slovo
word.KeyWord = "while";  word.Token = "T_KEYWORD_while";

keyWords nonterm;                                           pro nonterminál
nonterm.KeyWord = "expr";  nonterm.Token = "$3";
```

4.3 Třída rule

Třída rule obsahuje několik atributů a metod. UML diagram této třídy je na obrázku 4.2 (strana 25). Atribut Label slouží pro uložení řetězce levé strany pravidla, atributy InRule, resp. OutRule, slouží pro uložení řetězce pravé strany pravidla vstupního, resp. výstupního, pravidla. Atributy InLine a OutLine obsahují číslo řádku ve vstupním souboru, na kterém je toto pravidlo. Důležité metody jsou popsány v následujících sekcích.

4.3.1 Metoda createRuleLineForBison()

Tato metoda dostane na vstup referenci na tabulku klíčových slov (vektor objektů třídy keyWords). Jejím úkolem je z atributů Label, InRule a OutRule přeložit pravidlo tak, jak jej přijme Bison. K tomu potřebuje vektor nonterminálů, objektů třídy keyWords.

Metoda createRuleLineForBison() zavolá metodu inLineScanner() – popsaná v sekci 4.3.2, která pravou stranu pravidla vstupního jazyka převede na tokeny a uloží do vektoru řetězců. Následně volaná metoda outLineScanner() – popsaná v sekci 4.3.3, funguje obdobně pro pravou stranu pravidla výstupního jazyka, avšak provádí jiné akce s tabulkou klíčových slov a tabulkou nontreminálů.

Ve chvíli, kdy je naplněn vektor tokenů pravé strany pravidla vstupního jazyka a vektor pravé strany pravidla výstupního jazyka, je možné vytvořit pravidlo pro Bison. Řetězec s pravidlem je vytvořen následujícím způsobem:

1. Do řetězce s pravidlem se uloží levá strana pravidla (atribut Label), shodná s levou stranou pravidla ve vstupním souboru, a znak ':'.

2. Do řetězce s pravidlem se přidá každý token z vektoru tokenů pravé strany pravidla vstupního jazyka – syntax.

3. Jako sémantická akce pro pravidlo je překlad na pravidlo výstupního jazyka: na zásobník se vloží řetězec z každého klíčového slova a ukazatele na nonterminál v pořadí odpovídajícím pravidlu. Do řetězce s pravidlem se přidá mezi závorkami '{' a '}' kód v jazyce C++, který řetězec z klíčových slov a ukazatelů na nonterminály spojí a vloží na zásobník.

4. Pravidlo je ukončeno středníkem – do řetězce s pravidlem se přidá znak ';'.

4.3.2 Metoda inLineScanner()

Metoda inLineScanner() rozděluje pravou stranu vstupního pravidla na lexémy. Jednotlivá klíčová slova jsou oddělena mezerou, nonterminální lexémy jsou v závorkách '<' a '>'. Při načtení klíčového slova je volána metoda utillizeKeyWord() – popsána v sekci 4.3.4, která klíčové slovo zpracuje a vrátí token odpovídající klíčovému slovu. Tento token (řetězec) je přidán do vektoru tokenů pravé strany pravidla vstupního jazyka.

Při načtení nonterminálního lexému je provedena kontrola, je-li už ve vektoru nonterminálů (vektor objektů třídy keyWords). Pokud tam není, metoda ho tam přidá – atributu KeyWord je přiřazen řetězec lexému a atributu Token je přiřazen ukazatel na nonterminál (obsahuje pořadí tokenu v pravidle, např. pro třetí token je to řetězec „$3“). Řetězec lexému je přidán do vektoru tokenů pravé strany pravidla vstupního jazyka.

4.3.3 Metoda outLineScanner()

Metoda outLineScanner() rozděluje pravou stranu výstupního pravidla na lexémy. Jednotlivá klíčová slova jsou oddělena mezerou, nonterminální lexémy jsou v závorkách '<' a '>'. Při načtení klíčového slova je tento řetězec přidán do vektoru tokenů pravé strany pravidla výstupního jazyka.

Při načtení nonterminálního lexému je provedena kontrola, jestli už je ve vektoru nonterminálů (vektor objektů třídy keyWords). Atribut Token objektu třídy keyWords (tedy řetězec ve kterém je ukazatel na nonterminál v pravé straně pravidla vstupního jazyka) je přidán do vektoru tokenů pravé strany pravidla výstupního jazyka.

Pokud není načtený nonterminál ve vektoru nonterminálů, překlad skončí chybou, neboť pro každý nonterminál je potřeba mít pár mezi nonterminály v pravé straně pravidla vstupního jazyka.

4.3.4 Metoda utillizeKeyWord()

Tato metoda dostane na vstup klíčové slovo a referenci na tabulku klíčových slov (vektor objektů třídy keyWords). Když klíčové slovo najde v tabulce, vrátí jemu odpovídající token.

Pokud klíčové slovo dosud není v tabulce klíčových slov, přidá ho tam. Atributu KeyWord přiřadí klíčové slovo a atributu Token přiřadí nově vytvořený token. Pokud klíčové slovo obsahuje pouze písmena, číslice nebo znaky '_', pak je nový token složen z řetězců „T_KEYWORD_“ a klíčového slova. Například pro klíčové slovo „while“ je odpovídající token „T_KEYWORD_while“. Pokud klíčové slovo obsahuje i jiné znaky, pak je nový token složen z řetězců „T_KEYWORD_“ a pořadí nového klíčového slova ve vektoru klíčových slov. Například pro klíčové slovo „+=“, které by bylo třínácté v pořadí v tabulce klíčových slov, je odpovídající token „T_KEYWORD_13“.

4.4 Kompilace a spuštění překladače compiler

Překladač compiler se skládá z pěti dílčích souborů main.cpp, rule.hpp (deklarace třídy rule), rule.cpp (definice metod třídy rule), keyWords.hpp (deklarace třídy keyWords) a Makefile.

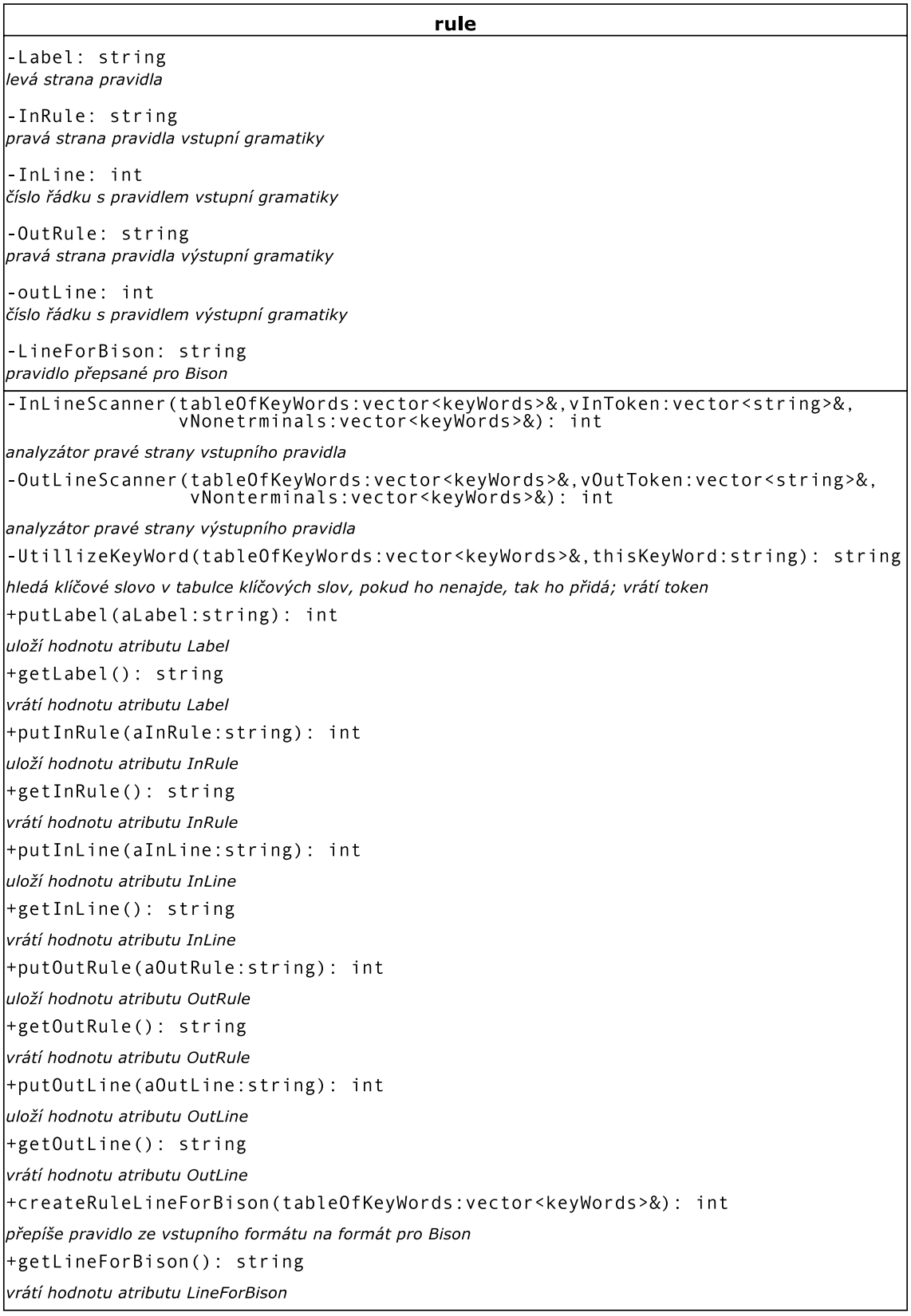
Kompilace: K přeložení překladače *compiler* je zapotřebí překladač g++ z balíku GNU GCC a program GNU Make. Samotná kompilace je pak vyvolána příkazem:

```
make
```

Spuštění: Programu *compiler* se při spuštění může předat jeden vstupní parametr, a to název souboru s překladovými pravidly. Pokud tento parametr není zadán, *compiler* implicitně použije soubor translation_rules ve stejném adresáři. Samotné spuštění překladače *compiler* se na unixových systémech provede příkazem:

```
./compiler c2pascal_rules        spuštění s parametrem
./compiler                       spuštění bez parametru
```

Výstupy: Výstupy programu jsou dva soubory scanner.l, který je program v jazyce překladače Flex, a parser.ypp, který je program v jazyce překladače Bison.



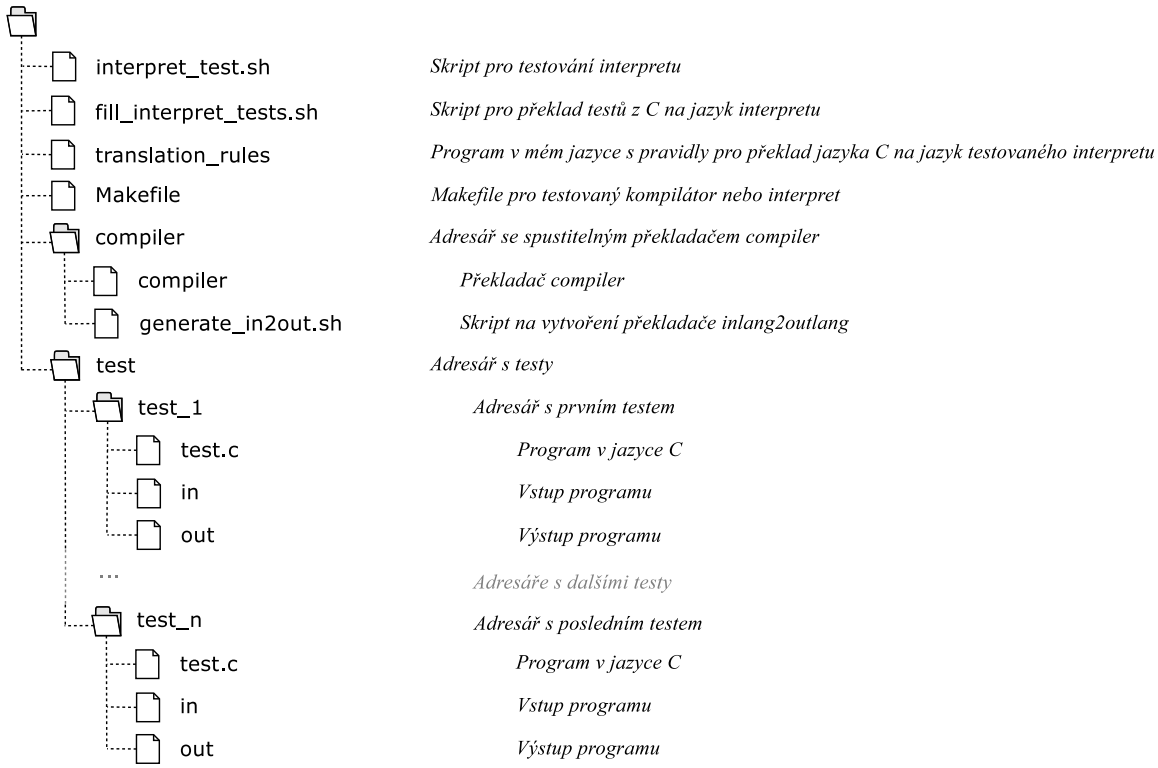
Obrázek 4.2: UML třídy rule

Kapitola 5

Testovací systém

Testování interpretu je založeno na principu porovnání výsledků interpretace testovacího programu s referenčními výsledky. Jelikož je systém pro testování interpretů navržen pro použití v operačním systému Linux, využívá skriptů napsaných pro interpret Bash. O testování interpretu se stará skript interpret_test.sh, o jehož funkci se dočtete v sekci 5.2.

Jako referenční jazyk systému pro testování interpretů jsem zvolila jazyk C. Sadu testovacích programů v jazyce testovaného interpretu získá systém přeložením testovacích programů v jazyce C na programy v jazyce interpretu. Zprostředkování tohoto překladu obstarávají skripty fill_interpret_tests.sh a generate_in2out.sh, o kterých se dočtete v sekci 5.1. Návrh adresářové struktury testovacího systému, o které se dále zmiňuji, je zobrazen na obrázku 5.1.



Obrázek 5.1: Adresářová struktura testovacího systému

5.1 Skripty fill_interpret_tests.sh a generate_in2out.sh

Skript fill_interpret_tests.sh napřed vytvoří překladač inlang2outlang skriptem (umístěným v adresáři compiler/) generate_in2out.sh. Ten postupuje podle schématu na obrázku 3.5 uvedeného v kapitole Návrh testovacího systému. Ve čtyřech krocích překládá program translation_rules (nebo jiný zadaný parametrem skriptu fill_interpret_tests.sh) překladači *compiler*, Flex, Bison a g++. Příklad vygenerování překladače inlang2outlang:

```
./compiler
flex scanner.l
bison parser.ypp
g++ parser.tab.cpp -o inlang2outlang
```

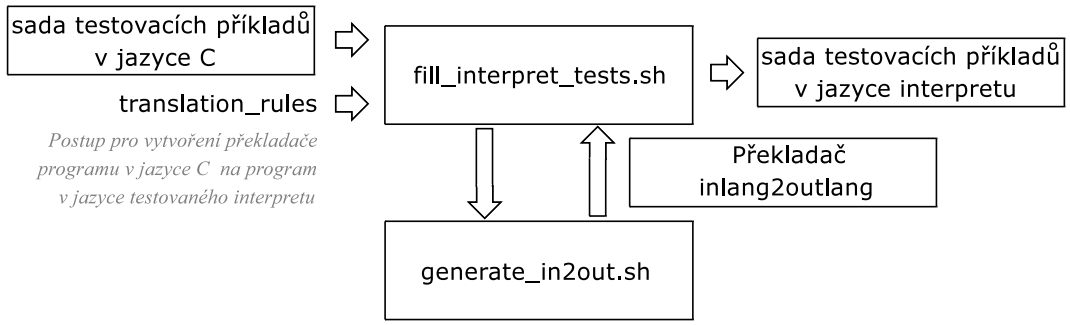
Následně skript fill_interpret_tests.sh prochází adresář s jednotlivými testy test/. V každém adresáři s testem (např. test_1/) přeloží program test.c v jazyce C na test2.out v jazyce testovaného interpretu (přípona souboru může být zadána parametrem skriptu). Činnost těchto skriptů je znázorněna na obrázku 5.2.

Parametry spuštění skriptu fill_interpret_tests.sh

`-g=` Nastaví vstupní soubor s popisem překladové gramatiky
`-p=` Přípona programu v jazyce testovaného překladače

Implicitní nastavení parametrů je:

```
fill_interpret_tests.sh -g=translation_rules -p=out
```



Obrázek 5.2: Činnost skriptu fill_interpret_tests.sh

5.2 Skript interpret_test.sh

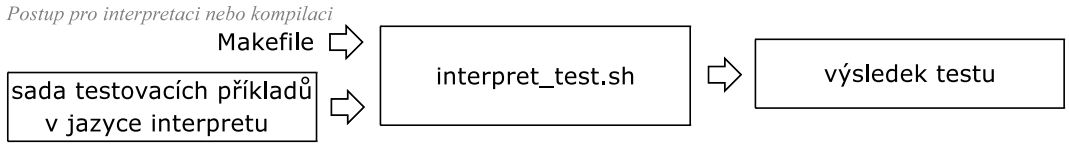
Tento skript prochází adresář s jednotlivými testy test/. V každém adresáři s testem (např. test_1/) spustí interpretaci testovacího programu (v případě, že se testuje kompilátor, program je přeložen a spuštěn) podle postupu uvedeného v Makefile. Programu předá vstupy ze souboru in z adresáře s aktuálním testem.

Výstup programu uloží do souboru out2 v adresáři s aktuálním testem. Výstup out2 a referenční výstup out porovná programem diff[1], a pokud se shodují, test proběhl úspěšně.

[1] Celý název: diff - compare files line by line

O výsledku jednotlivého testu je podána zpráva na standardní výstup. Na závěr skript informuje o celkové úspěšnosti testů.

Činnost tohoto skriptu je znázorněna na obrázku 5.3, propojení se skripty fill_interpret_tests.sh a generate_in2out.sh na obrázku 5.4



Obrázek 5.3: Činnost skriptů interpret_test.sh

Parametry spuštění skriptu interpret_test.sh

Skript interpret_test.sh lze spustit s několika parametry upřesňujícími postup testování.

`-t=` Nastaví zda se testuje kompilátor nebo interpret. Možné hodnoty:
- `c` pro kompilátor
- `i` pro interpret

`-f` Spustí vytvoření testovacích příkladů

`-g=` Nastaví vstupní soubor s popisem překladové gramatiky

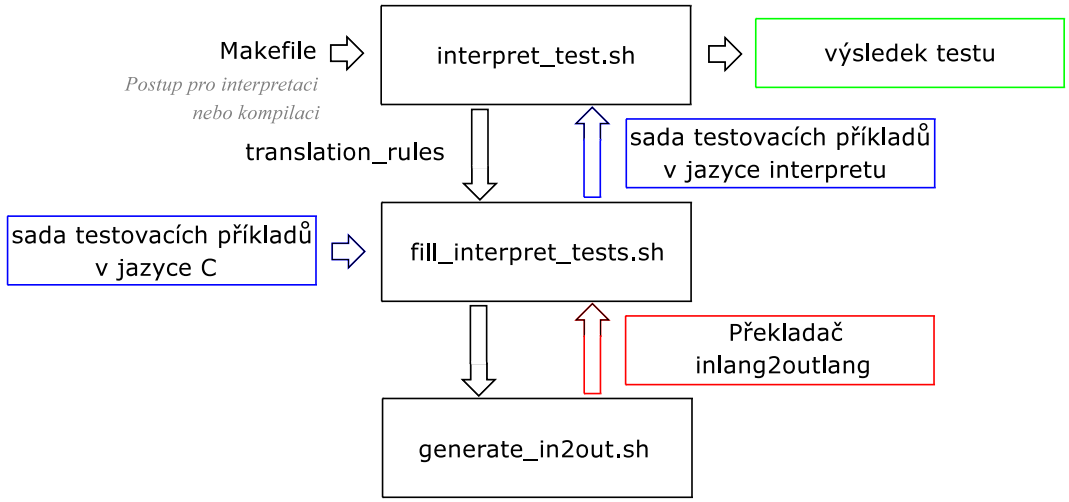
`-p=` Přípona programu v jazyce testovaného překladače

Implicitní nastavení parametrů je pro interpret Bash:

```
interpret_test.sh -t=i
```

Příklad spuštění s vytvořením testovacích příkladů a testování:

```
interpret_test.sh -t=i -f -g=c2bash_rules -p=sh
```



Obrázek 5.4: Spolupráce skriptů pro testování

5.3 Ověření správné implementace testovacího systému

Ověření správné implementace testovacího systému proběhlo testováním některých funkčních překladačů a interpretů, které jsou dostupné na studentském serveru merlin.fit.vutbr.cz.

Překlad jazyka C do jazyka C

Napřed byla otestována funkčnost překladu z jazyka C do jazyka C a testování překladačem gcc. Popis gramatiky jazyka C jsem vytvořila podle popisu základních řídících struktur v knize Učebnice jazyka C[1]. Jednalo se o test zda generátor překladače funguje správně. Překlad jazyka C do jazyka C a testování překladače gcc proběhlo úspěšně.

Překlad jazyka C do jazyka Pascal

Následně byl testován překlad jazyka C do jazyka Pascal a testování překladače fpc. Jelikož jazyk Pascal je strukturou i použitím podobný jazyku C (univerzální programovací jazyky), nebyl problém napsat pravidla pro překlad mezi těmito jazyky. Pravidla pro překlad gramatiky do jazyka Pascal jsem vytvořila také podle Učebnice jazyka C[1], kde jsou i úseky programů v jazyce Pascal. Překlad jazyka C do jazyka Pascal a testování překladače fpc také proběhlo úspěšně.

Překlad jazyka C do jazyka Bash

Překlad jazyka C do jazyka Bash byl obtížnější než jsem doufala. Je tomu tak proto, že jazyk C (kompilovaný jazyk, typovaný) a jazyk interpretu Bash (proměnné v Bashi jsou pouze jednoho datového typu – řetězec znaků) jsou dva odlišné jazyky s různými účely použití.

Definice překladové gramatiky je proto jen omezená a programy takto vytvořené nevystihují podstatu použití interpretu Bash. Pro testování interpretů doporučuji jako referenční jazyk zvolit jazyk interpretu s podobnou strukturou a použitím.

Kapitola 6

Závěr

Výsledkem této práce je funkční systém pro testování interpretů implementovaný pro použití v prostředí operačního systému Linux. Testovací systém je založen na principu porovnání výsledků interpretace a referenčních výsledků. Systém obsahuje překladač potřebný k přeložení programů v referenčním jazyce na jazyk interpretu a tím umožní jednoduché získání programů, vstupů a výstupů pro otestování interpretu z již existujících testovacích programů napsaných v referenčním jazyce a jejich vstupů a výstupů. Za referenční jazyk doporučuji zvolit jazyk s podobnou strukturou a účelem použití jazyku testovaného interpretu.

Systém je rozšířen o možnost přímého testování jak interpretu, tak i kompilátoru.

Do systému je implementován popis základních řídících struktur jazyka C a pravidla pro jejich překlad na jazyk Pascal a na jazyk interpretu Bash. Systém byl implementován tak, aby doplněním popisu o další řídící struktury jazyka C a pravidla pro přepis na jazyk testovaného interpretu bylo možné dosáhnout širších možností využití. Je jednoduché tento popis přepsat pro jazyky mnoha dalších interpretů a kompilátorů, avšak úplně obecný není. Například jazyk python, který pro odlišení podprogramů používá odsazení od začátku řádku (mezery nebo tabulátory), je takto popsat velmi obtížné, ne-li nemožné.

Tento systém je implementován s ohledem na to, aby se dala každá jeho část upravit pro potřebu uživatele. Po úpravě popisu překladové gramatiky se dá použít pro automatické testování projektů do předmětu Formální jazyky a překladače (IFJ). Referenčním jazykem byl zvolen jazyk C právě proto, že je blízký jazyku interpretu, který má být výsledkem tohoto projektu v předmětu IFJ.

Jako další rozšíření systému by bylo vhodné doplnit jej o skript, který programy v referenčním jazyce ze zadaného adresáře přesune tak, aby odpovídaly adresářové struktuře systému (automatické plnění testů).

Při zpracování tohoto tématu jsem si zopakovala formální modely potřebné pro vytváření překladačů, blíže jsem se seznámila s kompilátory a interprety, principem jejich činnosti a různými programovacími jazyky. Vyzkoušela jsem si implementaci generátoru překladače jednoho vyššího programovacího jazyka do jiného a práci s generátory lexikální a syntaktické analýzy (Flex a Bison), které jsou používány při tvorbě mnoha překladačů.

Literatura

[1] Herout, P.: *Učebnice jazyka C*. Kopp, 2005, iSBN 80-7232-220-6.

[2] Manuálové stránky: Bison. `man bison`.

[3] Manuálové stránky: Flex. `man flex`.

[4] Manuálové stránky: Lex. `man lex`.

[5] Manuálové stránky: Yacc. `man yacc`.

[6] Meduna, A.; Lukáš, R.: *Formální jazyky a překladače – IFJ, Přednášky k předmětu*. 2006.

[7] Meduna, A.; Lukáš, R.: *Formální jazyky a překladače – IFJ, Studijní opora*. 2006.

[8] Meduna, A.; Lukáš, R.: *Výstavba překladačů – VYP, Přednášky k předmětu*. 2006.

[9] Meduna, A.; Lukáš, R.: *Výstavba překladačů – VYP, Studijní opora*. 2006.

[10] WWW stránky: C++ Reference. `http://www.cppreference.com/`.

[11] WWW stránky: Converting grammars to LR. School of Computer Science, University of Manchester. `http://www.cs.man.ac.uk/~pjj/complang/g2lr.html`.

Příloha A

Obsah CD

Pro každý program a skript je ve složce s ním soubor README, který obsahuje jeho stručný popis a návod na použití.

A.1 Písemná zpráva

Písemná zpráva ve formátu pdf a její zdrojový text pro LaTeX je na přiloženém CD v adresáři pisemna_zprava/.

A.2 Testovací systém

Testovací systém se svojí adresářovou strukturou obsahující testovací skripty, testy a spustitelný překladač compiler (kompilovaný na studentském serveru merlin.fit.vutbr.cz) najdete v adresáři system_pro_testovani_interpretu/. Definice základních překladových gramatik z jazyka C na jazyk Pascal a na jazyk intepretu Bash (pouze omezený) je v adresáři system_pro_testovani_interpretu/translation_rules/.

A.3 Překladač compiler

Zdrojové kódy překladače compiler jsou v adresáři compiler/ a jsou i zařazeny v adresářové struktuře testovacího systému.