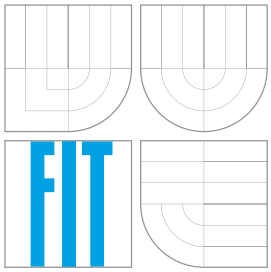
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POKROČILÉ ZOTAVENÍ Z CHYB BĚHEM SYNTAKTICKÉ ANALÝZY ZDOLA NAHORU

ADVANCED ERROR RECOVERY DURING BOTTOM-UP PARSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — DOMINIKA REGÉCIOVÁ
AUTHOR

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

BRNO 2016

Abstrakt

Tato bakalářská práce se zabývá pokročilými metodami zotavení při syntaktické analýze zdola nahoru. Popisuje a srovnává metody s pomocí upravené gramatiky FUN. Vytváří webové rozhraní, Projekt Alan, kde jsou vybrané metody implementovány pro simulaci procesu zotavení.
Autorka také přidává vlastní metodu, kterou následně porovnává s ostatními.

Abstract

This bachelor thesis deals with advanced methods for error recovery during bottom-up parsing. It describes and compares methods with the modified grammar FUN. It creates a web-based interface, Project Alan, where chosen methods are implemented for the simulation of error recovery process.
The author also added her own method and she compared it with others.

Klíčová slova

Syntaktická analýza, zotavování z chyb, gramatika FUN, Projekt Alan, Django, Python

Keywords

Parsing, error recovery, FUN grammar, Project Alan, Django, Python

Citace

REGÉCIOVÁ, Dominika. *Pokročilé zotavení z chyb během syntaktické analýzy zdola nahoru.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Meduna Alexander.

Pokročilé zotavení z chyb během syntaktické analýzy zdola nahoru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením profesora Alexandra Meduny. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Dominika Regéciová
5. července 2016

Poděkování

Ráda bych poděkovala Prof. RNDr. Alexandru Medunovi, CSc. za odborné vedení v průběhu psaní bakalářské práce, za přínosné konzultace a za veškerou trpělivost a ochotu.

Obsah

1 Slovo úvodem **3**

2 Definice základních pojmů **5**
2.1 Slovníček pojmů . . . 5
2.1.1 Abeceda . . . 5
2.1.2 Řetězec . . . 6
2.1.3 Délka řetězce . . . 6
2.1.4 Konkatenace řetězce . . . 6
2.1.5 Mocnina řetězce . . . 6
2.1.6 Podřetězec . . . 6
2.1.7 Jazyk . . . 6
2.1.8 Regulární výraz . . . 7
2.1.9 Konečný automat . . . 7
2.1.10 Deterministický konečný automat . . . 7
2.1.11 Bezkontextová gramatika . . . 7
2.1.12 Bezkontextový jazyk . . . 8
2.1.13 Zásobníkový automat . . . 8
2.1.14 Rozšířený zásobníkový automat . . . 8
2.2 Upravená gramatika FUN . . . 9

3 Teorie překladačů **10**
3.1 Překladač . . . 10
3.2 Lexikální analýza . . . 10
3.3 Syntaktická analýza . . . 11
3.3.1 Derivační strom . . . 11
3.4 Syntaktická analýza zdola nahoru . . . 12
3.4.1 LR tabulka . . . 12
3.5 Kompletní překladač . . . 13

4 Metody zotavení **14**
4.1 Detekce, zotavení, korekce . . . 14
4.2 Panický mód s množinou Follow . . . 15
4.2.1 Množina Follow . . . 15
4.3 Panický mód s množinou First . . . 15
4.3.1 Množina First . . . 15
4.4 Ad-hoc metoda . . . 16
4.5 Metoda Alan . . . 17

4.6 Další zajímavé metody . . . 17
4.6.1 Backward/Forward Move . . . 17
4.6.2 Burke-Fisher metoda . . . 17
4.6.3 Phrase Level Recovery . . . 18

5 Implementace Projektu Alan **19**
5.1 Webové rozhraní . . . 19
5.2 Lexikální analýza . . . 20
5.3 Ukázkové programy . . . 21
5.4 LR tabulka . . . 22
5.5 Syntaktická analýza . . . 22
5.6 Panický mód s množinou Follow . . . 23
5.7 Panický mód s množinou First . . . 23
5.8 Ad-hoc metoda . . . 24
5.9 Alanova metoda . . . 25
5.10 Testování projektu Alan . . . 25
5.10.1 LR tabulka . . . 26
5.10.2 HTML a CSS . . . 26
5.10.3 Python . . . 26
5.10.4 Překladač . . . 26

6 Porovnání metod zotavení **27**

7 Závěr **30**

Literatura **33**

Přílohy **34**
Seznam příloh . . . 35

A Obsah CD **36**
A.1 Technická zpráva . . . 36
A.2 Projekt Alan . . . 36
A.3 Archiv ukázkových programů . . . 36

B Užitečné informace o Projektu Alan **37**
B.1 Adresářová struktura projektu . . . 37
B.2 LR tabulka . . . 38
B.3 LR tabulka pro Ad-hoc metodu . . . 39
B.4 Zprovoznění Projektu Alan . . . 40

Kapitola 1

Slovo úvodem

„Dokonalosti není dosaženo tehdy, když už není co přidat, ale tehdy, když už nemůžete nic odebrat.“

Antoine de Saint-Exupéry

Diplomová práce, obzvlášť jakmile vzniká na půdě technické univerzity, je svázána přísnými představami o svém obsahu a formě. Avšak i když by se mělo jednat o text jednoznačný, přesný a objektivní, proces psaní je práce do značné míry tvůrčí a na každém slově záleží.

Nemusíme souhlasit s výrokem slavného spisovatele, že dokonalost nalezneme v minimalismu, ale kdo více dokáže ocenit genialitu jednoduchosti, než ti, kteří ji sami vytvářejí, ať již v podobě kódu, nebo například v návrhu topologie rozsáhlé sítě.

Abychom zde nezabředli do filozofických debat, bakalářská práce s názvem **Pokročilé zotavení z chyb během syntaktické analýzy zdola nahoru** zřejmě nedosáhne absolutní dokonalosti, ale budeme se snažit projít tímto tématem tak, aby bylo přehledné, názorné a pochopitelné, rozšířilo čtenářovi znalosti a snad ho i ponouklo k dalšímu studiu ve věcech gramatik a překladačů.

Hlavní motivací vzniku práce bylo vytvoření podpůrného materiálu pro studenty především bakalářského programu na *Fakultě informačních technologií* v míře, kterou například kurz *Formální jazyky a překladače* aktuálně nepojímá. Čtenáři nabídneme náhled do další kapitoly o fungování překladačů z pohledu zotavování ze syntaktických chyb. Protože však nic není více názorné, než naše vlastní zkušenost, implementujeme rozhraní, přes které je možné rozebíranou tématiku vyzkoušet a otestovat. Vysvětlíme, proč je potřeba zotavování během syntaktické analýzy a jaké metody jsou k tomuto účelu užívány. U vybraných metod si také ukážeme implementaci.

S dovolením si uveďme ještě jeden citát autora Malého prince: *Technický vývoj směřuje vždy od primitivního přes komplikované k jednoduchému.* Ani tato práce se nebude příčit tomuto vývoji. Nejdříve si představíme velmi primitivní chování syntaktické analýzy, kterou budeme postupně rozvíjet, dělat ji více komplikovanou. A nakonec si představíme metodu, která byla vymyšlena pro účely tohoto projektu a která by měla symbolizovat návrat k jednoduchosti.

Ale popořadě. Nejdříve si proto v kapitole **Definice základních pojmů** [2] vysvětlíme pojmy, které budeme dále hojně používat, a je dobré se sjednotit v jejich významu. Dále si představíme gramatiku [2.2], jenž nás bude provázet až do konce a také uvedeme základní

Teorií překladačů [3], kde zmíníme jeho jednotlivé části a jejich fungování. V kapitole **Metody zotavení**[4] si vysvětlíme rozdíly mezi zotavováním, detekcí a korekcí a seznámíme se s vybranými zástupci. V této a následující kapitole se budeme věnovat konkrétním metodám. Panický mód, Panický mód s využitím množiny First, Ad-hoc metoda a na závěr Alanova metoda, která byla vytvořena pro potřeby této práce. Každá metoda bude představena jak teoreticky [4], tak prakticky [5]. V kapitole **Porovnání metod** [6] provedeme sérií testů, které odhalí slabé i silné stránky porovnávaných způsobů zotavení. V **závěru** [7] si zhodnotíme dosažené cíle, shrneme si, čemu jsme se přiučily a jaké vylepšení a další kroky by mohli být provedeny v budoucnu. V sekci **příloh** [A] tohoto dokumentu může čtenář nalézt další zajímavé informace. Součástí není pouze kompletní implementace včetně technické zprávy a testovacích souborů přiložených na CD, ale také návod na zprovoznění webového rozhraní na lokálním počítači a adresářová struktura projektu, která může pomoci případným zájemcům v orientaci při procházení zdrojových souborů.

Než s čtenáři přejdeme k hlavnímu tématu, je tu ještě jedna drobnost k poznamenání. Myšlenka **Projektu Alan** vznikla na začátku roku 2014, kdy mně, autorce této práce, bylo velkoryse nabídnuto vedení bakalářské práce profesorem Alexandrem Medunou. Protože je téma zotavování z chyb velmi zajímavé a lákavé, rozhodla jsem se vydat právě tímto směrem. Již na první konzultaci jsem vyjádřila touhu vytvořit práci s praktickým využitím například pro budoucí studenty. Výsledkem domluvy byl cíl vytvořit aplikaci, které by prezentovala jednotlivé metody zotavování při syntaktické analýze zdola nahoru. Myšlenka na webové rozhraní se sama nabízela, aby usnadnila přístup uživatelům bez potřeby cokoliv instalovat.

Protože však práce na bakalářce nakonec přesáhla jeden akademický rok, předstihla ji v roce 2015 ukončená práce mé spolužačky Aleny Oblukové, na obdobné téma **Pokročilé zotavení z chyb během syntaktické analýzy shora dolů** [1]. Její zadání vzniklo později a v této práci z ní není vycházeno. V závěru sice nabízí možnost rozšíření projektu směrem, který jsem učinila já (hlavně přidání grafického rozhraní), avšak tato idea byla od začátku má vlastní.

[1] Obluková, A. *Pokročilé zotavení z chyb během syntaktické analýzy shoda dolů.* Brno, 2015. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Meduna Alexander.

Kapitola 2

Definice základních pojmů

„Dějiny člověka jsou dějinami pojmů, o které postupně rozšířil své znalosti.“

Antoine de Saint-Exupéry

Pro pořádek a lepší pochopení následujících kapitol si nejdříve vysvětlíme řadu pojmů. Kdo se cítí být dobře obeznámen s významem slov a spojení, jakými jsou *abeceda*, *jazyk*, *regulární výraz*, či *bezkontextová gramatika*, nechť v klidu tuto část přeskočí a pokračuje dále na podkapitolu **Upravená gramatika FUN** [2.2], kde se dozví více informací o gramatice používané v Projektu Alan. My ostatní však začneme pěkně popořadě.

Následující definice jsou přejaty z knih *Elements of Compiler Design* [7], *Formal Languages and Computation: Models and Their Applications* [8] a z materiálů k předmětům *Výstavba překladačů* [10] a *Formální jazyky a překladače* [9].

2.1 Slovníček pojmů

„Na počátku bylo Slovo,...“ (Jan, 1, 1–3). Neuvádíme zde sice celé znění poměrně známé věty, avšak většina čtenářů jistě uhádne, o kterém „Slově“ Jan Evangelista píše. Pokud však odhlédneme od víry, máme v psané podobě ještě elementárnější prvky – znaky, symboly, písmena. Jestliže si stanovíme sadu těchto symbolů, vzniká *abeceda*.

2.1.1 Abeceda

Definice 2.1.1. Abeceda Σ je konečná neprázdná množina, jejíž prvky nazýváme symboly [8, str. 13].

Ta nám však sama o sobě nestačí. Aby dokázala přenášet informaci, spojujeme je do celků, které označujeme jako *řetězce*, nebo ještě jinak jako slova.

2.1.2 Řetězec

Definice 2.1.2. Konečnou posloupnost symbolů z abecedy Σ nazýváme *řetězcem* nad Σ. Řetězec, který neobsahuje žádné symboly, označujeme jako *prázdný řetězec* pomocí písmene ε a je také řetězcem nad abecedou Σ. [8, str. 13].

2.1.3 Délka řetězce

Definice 2.1.3. Nechť je x řetězcem nad abecedou Σ. *Délka řetězce* $|x|$ značí počet symbolů v x. Prázdný řetězec má nulovou délku. [9, Ifj01-cz.pdf, str. 4]

2.1.4 Konkatenace řetězce

Definice 2.1.4. Nechť jsou x a y řetězce nad abecedou Σ. *Konkatenací* x a y vznikne řetězec xy. [9, Ifj01-cz.pdf, str. 5]

2.1.5 Mocnina řetězce

Definice 2.1.5. Nechť je x řetězcem nad abecedou Σ. Pro $i \geq 0$, i-tá *mocnina* řetězce x, x^i je definována:

1. $x^0 = \varepsilon$
2. pro $i \geq 1$: $x^i = xx^{i-1}$ [9, Ifj01-cz.pdf, str. 6]

2.1.6 Podřetězec

Definice 2.1.6. Nechť jsou x a y řetězce nad abecedou Σ. Pokud existují řetězce a b nad abecedou Σ a platí, že $axb = y$, pak je x *podřetězcem* y. [9, Ifj01-cz.pdf, str. 10]

V této chvíli máme za sebou vyčerpávající výčet různých definic, které se váží k řetězcům. Nyní však můžeme přejít k něčemu daleko zajímavějšímu. Stejně jako v přirozené mluvě či psaném projevu, můžeme řetězce skládat do logických množin. Vytváříme tím jazyk.

2.1.7 Jazyk

Definice 2.1.7. Nechť je Σ^* množinou všech řetězců nad abecedou Σ. Každá podmnožina L je *jazyk* nad Σ. [9, Ifj01-cz.pdf, str. 11]

Všichni intuitivně chápeme význam slova *jazyk*. Můžeme mít na mysli jazyk přirozený (například český či anglický), programátorský (C, Python, Java), či dokonce uměle vytvořený (Quenijština od J. R. R. Tolkiena). Pro naše potřeby je však nutné pokročit trochu dále a definovat si další pojmy, na které se v následujících kapitolách budeme odkazovat.

2.1.8 Regulární výraz

Definice 2.1.8. Nechť je Σ abeceda. Regulární výrazy nad abecedou Σ a jazyky, které tyto výrazy reprezentují, jsou definovány jako:

- $\varnothing$ je regulární výraz označující prázdnou množinu (jazyk)
- ε je regulární výraz označující jazyk $\{\varepsilon\}$
- a, kde $a \in \Sigma$ ke regulární výraz označující jazyk $\{a\}$
- nechť jsou r a s regulární výrazy značící jazyky R, S (po řadě), pak:
  - $(r|s)$ (značeno také $(r + s)$) je regulární výraz značící $R \cup S$
  - (rs) je regulární výraz značící RS
  - (r^*) je regulární výraz značící R^*

Jazyky, které jsou značeny regulárními výrazy nazýváme *regulárními jazyky*. [7, str. 21-22]

2.1.9 Konečný automat

Definice 2.1.9. *Konečný automat* je pětice $M = (Q, \Sigma, R, s, F)$, kde:

- Q je konečná množina stavů
- Σ je vstupní abeceda
- R je konečná množina pravidel ve tvaru: $pa \to q$, kde $p, q \in Q, a \in \Sigma \cup \{\varepsilon\}$
- $s \in Q$ je počáteční stav
- $F \subseteq Q$ je množina koncových stavů [10, VYPe02-en.pdf, str. 7]

2.1.10 Deterministický konečný automat

Definice 2.1.10. Nechť $M = (Q, \Sigma, R, s, F)$ je *Konečný automat bez ε-přechodů*. M je *deterministický konečný automat* (DKA), pokud pro každé $pa \to q \in R$ platí, že množina $R - \{pa \to q\}$ neobsahuje žádné pravidlo s levou stranou pa. [9, Ifj04-cz.pdf, str. 5]

2.1.11 Bezkontextová gramatika

Definice 2.1.11. *Bezkontextová gramatika* je čtveřice $G = (N, T, P, S)$, kde:

- N je abeceda *neterminálů*
- T je abeceda *terminálů*, přičemž $N \cap T = \varnothing$
- P je konečná množina *pravidel* tvaru $A \to x$, kde $A \in N$, $x \in (N \cup T)^*$
- S $\in N$ je *počáteční neterminál* [10, VYPe04-en.pdf, str. 3]

2.1.12 Bezkontextový jazyk

Definice 2.1.12. Nechť je L jazyk. L je *bezkontextový jazyk*, pokud existuje bezkontextová gramatika [2.1.11], která generuje tento jazyk. [10, VYPe04-en.pdf, str. 8]

2.1.13 Zásobníkový automat

Definice 2.1.13. Zásobníkový automat je sedmice $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde

- Q je *konečná množina stavů*
- Σ je *vstupní abeceda*
- Γ je *zásobníková abeceda*
- R je *konečná množina pravidel* tvaru $Apa \to wq$, kde $A \in \Gamma, p, q \in Q, a \in \Sigma \cup \{\varepsilon\}$, $w \in \Gamma^*$
- $s \in Q$ je *počáteční stav*
- $S \in \Gamma$ je *počáteční symbol na zásobníku*
- $F \subseteq Q$ je množina *koncových stavů* [10, VYPe04-en.pdf, str. 21]

2.1.14 Rozšířený zásobníkový automat

Definice 2.1.14. Rozšířený zásobníkový automat je zásobníkový automat, který dokáže z vrcholu zásobníku přečíst celý řetězec (v ZA to byl pouze jeden symbol) [9, Ifj06-cz.pdf, str. 34]

2.2 Upravená gramatika FUN

V projektu využíváme gramatiku, která je inspirovaná gramatikou FUN s laskavým svolením od profesora Alexandra Meduny. Její plnou a nezměněnou podobu lze nalézt v [7, str. 81-82]. Důvodem úprav byla jednoduchost na pochopení, ale i lepší přehlednost.
Nová gramatika má omezený, avšak stále zajímavý základ pravidel, včetně svých specifikací, které si postupně vysvětlíme a na kterých budeme demonstrovat vznik chyb, včetně jejich opravení.

1: $< statement\_list > \rightarrow < statement >;\ < statement\_list >$
2: $< statement\_list > \rightarrow < statement >$
3: $< statement > \rightarrow < statement > r < condition >$
4: $< statement > \rightarrow < statement > \& < condition >$
5: $< statement > \rightarrow < statement > | < condition >$
6: $< statement > \rightarrow < condition >$
7: $< condition > \rightarrow\ ! < condition >$
8: $< condition > \rightarrow < expression >$
9: $< expression > \rightarrow < expression > + < term >$
10: $< expression > \rightarrow < expression > - < term >$
11: $< expression > \rightarrow < term >$
12: $< term > \rightarrow < term > * < factor >$
13: $< term > \rightarrow < term > / < factor >$
14: $< term > \rightarrow < factor >$
15: $< factor > \rightarrow\ (< expression >)$
16: $< factor > \rightarrow\ i$
17: $< factor > \rightarrow\ \#$

Jak můžeme vidět, gramatika stanovuje základní aritmetické operace, jakými jsou sčítání, odčítání, dělení a násobení, využívá také logických operátorů *and*, *or* a *not*. Dále umožňuje použití relačních operátorů. Příkazy lze psát za sebou a oddělovat je pomocí středníku. Poslední řádek však zůstává bez něj. Gramatika rozlišuje pouze proměnné a celá čísla.

Již zde si lze povšimnout pravidel, které mohou vést k chybám z nepozornosti – chybějící středník, nesprávný počet závorek. Pro zdůraznění je třeba uvést, že je zde ještě jedna odchylka od původní verze gramatiky, kdy chybí pravidlo: $< condition > \rightarrow (< condition >)$, což vede k tomu, že například výraz $(!variable)$ je syntakticky neplatný.

Kapitola 3

Teorie překladačů

„Teorie zůstane pouhou teorií, pokud nepřikročíme k činu.“

Jan Amos Komenský

3.1 Překladač

Překladač je nástrojem, který umožňuje programátorům efektivně psát kód na vyšší úrovni abstrakce. Nemusí se zabývat konkrétním hardwarem, či omezeným počtem registrů, protože to a mnoho dalšího za ně řeší překladač, který ze zdrojového programu (napsaného ve zdrojovém jazyce) vytvoří program v jazyce cílovém tak, že jsou oba programy *ekvivalentní* [7, str. 8], což znamená, že vykonávají stejný výpočetní úkon.

Překladač dělíme na několik logických celků: *lexikální analyzátor*, *syntaktický analyzátor*, *sémantický analyzátor*, *generátor vnitřního kódu*, *optimalizátor* a *generátor cílového kódu*.

Jejich úkolem není pouze překlad, ale také kontrola správnosti zdrojového kódu, který musí být napsán dle pravidel zdrojového jazyka. Pokud tomu tak není, překlad nemůže být dokončen a programátor je upozorněn na nedostatky v programu v podobě chybového hlášení.

3.2 Lexikální analýza

Lexikální analýza, neboli *scanner*, představuje část překladače, která zpracovává vstup v podobě zdrojového programu. Rozeznává *lexémy*, což jsou lexikální jednotky. V běžném, například anglickém jazyce si je můžeme představit jako jednotlivá slova a interpunkci.
Pokud nalezne lexém, jenž není součástí gramatiky, říkáme, že analýza objevila lexikální chybu.

Lexikální analýza je často implementována jako závislá na jiné části překladače, nejčastěji na syntaktické analýze. Prochází postupně vstup a vždy když je potřeba, identifikuje další lexém (nebo vrátí chybové hlášení). Rozpoznaný lexém odešle v podobě *tokenu*. Token obsahuje lexém a další informace, které jsou potřeba (např. ukazatel do paměti, hodnotu čísla).

Lexikální analýza čte zdrojový program stejně jako by činil lidský čtenář — čte znak po znaku ve směru zleva doprava po jednotlivých řádcích. Kromě rozeznávání lexémů vykonává i další činnosti — odstraňuje z dalšího zpracování nepotřebné lexémy, jakými jsou komentáře a bílé znaky. V neposlední řadě je také využívána pro ukládání potřebných dat do tzv. *tabulky symbolů*, datové struktury, jenž obsahuje informace o hodnotách konstant, jménech proměnných, typech parametrů procedur a mnoha dalšího.

Lexémy daného jazyka jsou definované pomocí *regulárních výrazů* [2.1.8]. Pro samotnou implementaci se využívá *Deterministického konečného automatu* (DKA)[2.1.10].

Pro úplnost také uveďme, že existují nástroje pro automatické generování lexikální analýzy z regulárních výrazů, jmenovitě například *Lex* [6].

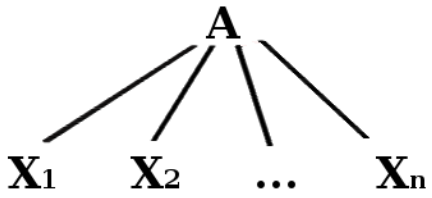
3.3 Syntaktická analýza

Syntaktická analýza, neboli *parser*, kontroluje, zda posloupnost tokenů představují správě syntakticky napsaný program. Řídí se množinou gramatických pravidel. Pokud se mu podaří sestavit *derivační strom*, program je po gramatické stránce správný, v opačném případě překladač zjistí *syntaktickou chybu*.

3.3.1 Derivační strom

Definice 3.3.1. Derivační strom (v angličtině *Parse tree*) podle bezkontextové gramatiky je strom s následujícím vlastnostmi:

- Kořen stromu je označen počátečním symbolem
- Každý list je označen terminálem nebo ε
- Vnitřní uzly jsou označeny neterminály
- Pokud je A neterminál označující vnitřní uzel a $X_1, X_2, \ldots, X_n$ označují potomky tohoto uzlu zleva doprava, pak musí existovat pravidlo $A \to X_1, X_2, \ldots, X_n$. Pak každý symbol z $X_1, X_2, \ldots, X_n$ označuje terminál, či neterminál. Ve zvláštním případě, kdy $A \to \varepsilon$, má tento uzel pouze jednoho potomka značeného jako ε. [1, str. 45]



Obrázek 3.1: Derivační strom pro pravidlo $A \to X_1, X_2, \ldots, X_n$.

Při vytváření derivačního stromu jsou voleny dva přístupy: *shora dolů*, či *zdola nahoru*. Záleží, zda nejdříve pracujeme s kořenem stromu, nebo jeho listy.

3.4 Syntaktická analýza zdola nahoru

V této práci budeme uvažovat pouze syntaktickou analýzou, kterou postupuje metodou zdola nahoru. I zde neexistuje pouze jedna varianta, my však budeme pracovat pouze s **LR syntaktickým analyzátorem**, jenž pracuje nad **LR gramatikou**, což je bezkontextová gramatika, pro kterou lze sestrojit **LR tabulku** [10, VYPe06b-cz.pdf, str. 3]. Mezi další patří například **Precedenční syntaktický analyzátor**, který je však slabší, než LR parser [10, VYPe06a-cz.pdf, str. 3].

Zkratka LR značí ***L****eft-to-right*, tedy že vstup čteme zleva doprava a ***R****ightmost derivation*, tedy že se vždy snažíme nahradit nejpravější neterminál. Pracujeme se zásobníkem, kam ukládáme pravé strany pravidel, neboli *handle* a LR tabulkou.

3.4.1 LR tabulka

Definice 3.4.1. Uvažujeme LR gramatiku $G = (N, T, P, S)$. LR tabulka založená na G se skládá z částí:

- *action* (akce)
- *goto* (přechod)

Řádky jsou označeny symboly z $\Theta = \{\theta_1, \ldots, \theta_m\}$, což je množina stavů rozšířeného zásobníkového automatu. Sloupce části *action* jsou označeny symboly z množiny terminálů a sloupce části *goto* jsou označeny symboly z množiny neterminálů [10, VYPe06b-cz.pdf, str. 5].

Celá LR syntaktická analýza je řízena LR tabulkou. Postup je následující:
Nejdříve načteme token ze vstupu a poté prohledáváme tabulku v části *action* pro určení dalšího kroku. Řádek tabulky určuje stav (na počátku roven 0), sloupec pak určuje aktuálně načtený token. Vybrané políčko obsahuje jednu z následujících možností:

- **s*p***
- **r*q***
- **acc**
- **prázdné políčko**

První možnost, tedy **s*p*** značí pokyn k operaci *shift*, tedy k posunu. Na zásobník vložíme aktuálně načtený token společně s číslem p, které značí nový stav. Aktualizujeme stav a načteme ze vstupu další token.

Druhá varianta, **r*q***, znamená pokyn k provedení *redukce*, neboli aplikaci pravidla. Pokud existuje pravidlo $A \to X$ (kde q značí pořadové číslo tohoto pravidla) a na vrcholu zásobníku nalezneme $<?, p><X, ?>$, pak aktualizujeme stav dle tabulky *goto*[A, p], následně na zásobníku provedeme výměnu $<X, ?>$ za $<A, goto[A, p]>$. Jinak překladač zahlásí chybu.

V případě políčka **acc** syntaktická analýza proběhla úspěšně. Naopak **prázdné políčko** značí objevení syntaktické chyby.

Tento proces opakujeme do okamžiku, kdy je analýza úspěšně dokončena, nebo je na vstupu nalezena chyba.

Stejně jako lexikální analýzu, i tu syntaktickou lze generovat pomocí automatizovaných nástrojů. Příkladem může být například *Yacc*[5] (zkratka značící *Yet Another Compiler-Compiler*) od Stephena C. Johnsona, jenž generuje zdrojový kód v jazyce C.

3.5 Kompletní překladač

Jak jsme si již uvedli na začátku této kapitoly, lexikální a syntaktická analýza jsou jen části skutečného překladače. Abychom skutečně naplnili podstatu překladače – přeložení zdrojového programu do cílového jazyka, je potřeba provést více kroků.

Můžeme si představit situaci, kdy se dítě učí číst. Nejdříve rozeznává jednotlivá písmena. *M-á-m-a-m-e-l-e-m-a-s-o*. Poté dokáže přečíst jednotlivá slova. *Máma-mele-maso*. Rozumí jednotlivých slovům, ale ještě nevnímá význam věty, vyjádření její činnosti. V této fázi se nachází překladač po syntaktické analýze (za předpokladu, že překlad neprobíhá paralelně s dalšími částmi). Abychom se dopracovali k pochopení významu, potřebujeme **sémantickou analýzu**. Až v tomto bodě víme, že *Máma vaří a připravuje jídlo*.

V případě překladače kontrolujeme, zda jsou instrukce v souladu se sémantikou, případně upravujeme kód, aby tomu tak bylo. Kontrolujeme deklarace proměnných, provádíme typovou kontrolu, případné přetypování proměnných. Překladače také obvykle generují **vnitřní kód**, který je vhodný především pro svoji jednotnost, která usnadňuje *optimalizaci* a následný překlad do cílového jazyka. **Optimalizátor** není nepostradatelnou součástí překladače a řada z nich žádný proces optimalizace ani nemá, avšak umožňuje nám efektivněji pracovat s pamětí a odhalit instrukce či proměnné, které jsou nadbytečné a nevyužívané. Vytváří tak *optimalizovaný vnitřní kód*. Teprve až tento zdrojový kód převádí na **cílový jazyk**. Tím většinou bývá strojový kód.

V této bakalářské práci se však budeme zabývat pouze lexikální a především syntaktickou analýzou. Pokud by měl čtenář zájem dozvědět se více o fungování překladu jako celku, či ho zaujala část, které zde není věnováno místo, nechť nahlédne do seznamu použité literatury, konkrétně například do knihy *Elements of Compiler Design* [7].

Kapitola 4

Metody zotavení

„Nehovořte o chybách, ty budou hovořit samy za sebe.“

Winston Churchill

Metod zotavování z chyb během syntaktické analýzy je poměrně velké množství. Většina z nich se však liší pouze jménem, přičemž si zachovávají svůj základní princip. Často se také můžeme setkat se situací, kdy má jedna metoda více pojmenování v závislosti na použité literatuře. V této práci čtenář nalezne takové metody, které reprezentují základní postupy, nebo naopak přinášejí originální způsob řešení.

4.1 Detekce, zotavení, korekce

Pokud hovoříme o metodách zotavování z chyb, je dobré si vysvětlit, co zotavování přesně znamená a jak se liší od detekce a korekce.

Detekce značí odhalení přítomnosti chyby (chyb) ve vstupu. Syntaktická analýza rozpozná nesprávné použití gramatiky. Zjistí, že vstup není větou jazyka, který gramatika popisuje a obsahuje syntaktickou chybu. Schopnost detekce má téměř každý parser bez nutnosti rozsáhlé modifikace.[4, str. 229]. Výjimkou je například precedenční syntaktická analýza, která nedokáže detekovat chybějící neterminál [4, str. 191].

Při **zotavení** je analýza schopna pokračovat až do konce vstupu a tím potencionálně dokáže odhalit všechny chyby v programu. Cílem je také snaha o zamezení vzniku falešných chybových hlášení, které mohou vzniknout vinou zotavovacího procesu. Problém nastává při vytváření derivačního stromu a dodržování sémantických akcí, čehož syntaktická analýza se zotavováním z chyb často není schopna [4, str. 230].

Korekce opraví vstup do té míry, že je syntaktická analýza schopna dokončit tvorbu derivačního stromu a provést sémantické akce, jako kdyby byl vstup v pořádku. Využívá různých postupů, kdy odstraňuje, vkládá, či mění symboly.
Nutné podotknout, že korekce nedokáže vstup opravit tak, jak si ho představoval sám autor zdrojového programu, ale to není ani snahou těchto metod. V literatuře proto také mohou být označovány jako *error repair*, neboli „opravující chyby“, což je přesnější popis než „korekce“ [4, str. 230].

4.2 Panický mód s množinou Follow

Jako první si představíme metodu nejčastěji nazývanou *Panický mód*. Přesněji Panický mód s využitím množiny Follow, což je asi nejběžnější podoba.

4.2.1 Množina Follow

Definice 4.2.1. Nechť $G = (N, T, P, S)$ je Bezkontextová gramatika. Pro všechna $A \in N$ definujeme množinu $Follow(A)$: $Follow(A) = \{a : a \in T, S \Rightarrow^* xAay, x, y \in (N \cup T)^*\} \cup \{\$: S \Rightarrow^* xA, x \in (N \cup T)^*\}$ [10, VYPe05-cz.pdf, str. 28]
Pomocí *Follow()* zjistíme všechny terminály, které mohou být vpravo od A.

Panický mód již podle názvu nepatří mezi nejvíc sofistikované postupy zotavování z chyb. Naopak, je označována za možná nejprimitivnější metodu vůbec [4, str. 534]. Přesto je až překvapivě efektivní a nedá se jí upřít nápaditost. V čem tedy spočívá? V situaci, kdy parser objeví chybu, lidově řečeno zpanikaří. A začne hledat záchytný bod, díky kterému by se mohl opět uklidnit. Tím záchranným bodem je tzv. **synchronizační token** na vstupu z množiny, kterou si definujeme. Jedná se o ukončovače logických celků programu – například středník, závorky nebo klíčové slovo *end*.

Překladač pak v okamžiku nalezení chyby začne prohledávat vstup, aby našel co nejkratší řetězec do záchytného bodu. Jakmile nalezne tento synchronizační bod (byť by se jednalo o konec vstupního programu), odstraní vstup až k tomu tokenu. Jaký bod má na vstupu v daný moment hledat mu prozradí právě množina Follow(). Protože pracujeme s LR syntaktickou analýzou, je potřeba také zkontrolovat zásobník a případně odstranit tokeny, které tam již díky zotavení nepatří. Pokud bychom tak neučinili, hrozí upadnutí do nedefinovaného stavu.

I přes jednoduchou myšlenku má metoda překvapivou účinnost na nejrůznější typy chyb. Nevýhodou jsou případy, kdy se během zotavování přeskočí velký kus vstupu, hlavně pokud je množina synchronizačních tokenů omezeného počtu. Je pak velmi pravděpodobné, že syntaktická analýza nedetekuje všechny chyby v daném logickém celku.

V knize *Compilers: principles, techniques & tools* [1, str. 196] je pak zmíněná ještě jedna skutečnost, která přidává metodě na zajímavosti a užitečnosti: metodě (na rozdíl od mnohých jiných) nehrozí upadnutí do nekonečné smyčky.

4.3 Panický mód s množinou First

4.3.1 Množina First

Definice 4.3.1. Nechť $G = (N, T, P, S)$ je Bezkontextová gramatika. Pro každé $x \in (N \cup T)^*$ definujeme množinu $First(x) : First(x) = a : a \in T, x \Rightarrow^* ay, y \in (N \cup T)^*$. [10, VYPe05-cz.pdf, str. 4]
Pomocí *First()* zjistíme všechny terminály, kterými může začínat větná forma derivovatelná z x.

Alternativní podoba Panického módu vznikla pro otestování odlišného přístupu k odstranění nejkratšího chybového řetězce na vstupu. Zatímco jsme však v předchozím případě pátrali po konci logického celku, nyní hledáme začátek, na který lze navázat. Teoreticky stačí opět přeskočit část vstupu dle synchronizačního tokenu a pak zkontrolovat zásobník, avšak již zde si můžeme prozradit výsledky reálné implementace, která ukazuje, že situace není tak jednoduchá.

Množina First je vhodná spíše pro **LL syntaktickou analýzu**, která vytváří derivační strom směrem shora dolů a provádí nejlevější derivaci. Začíná s počátečním neterminálem a pomocí aplikací pravidel simuluje vytvoření posloupnosti terminálů na vstupu. Jaké pravidlo zvolit, určuje právě množina First. Další informace lze nalézt například v [9, Ifj07-cz.pdf]. Pro LL syntaktickou analýzu by byly tyto úpravy pravděpodobně dostačující, to však neplatí pro LR parser. Jak se ukázalo, syntaktická analýza se může vlivem zotavení dostat do nedefinovaného stavu, protože na vstupu nalezne tokeny, se kterými si neví rady. Proto bylo potřeba doplnit metodu o druhou kontrolu vstupu, tentokrát pro množinu Follow. Po odstranění tokenů, které by vedli ke špatným výsledkům, může překlad bezpečně pokračovat.

4.4 Ad-hoc metoda

Metoda Ad-hoc již svým názvem prozrazuje, že je sestavena pro určitou syntaktickou analýzu, přesněji pro její gramatiku. Její přesný popis se v záležitosti na použité literatuře různí, někdy dokonce popisuje celou skupinu metod zotavování. My budeme uvažovat variantu, která je popsáná v *Elements of Compiler Design* [7, str. 178].

Skutečnost, že je Ad-hoc metoda „šitá na míru“ překladači dává prostor pro lepší optimalizaci pro konkrétní potřeby syntaktické analýzy, její síla je však limitována schopností autora a jeho porozumění gramatice.

Pracuje s LR tabulkou, přesněji s částí *action*, kde dochází k detekci chyby při přechodu do stavu s prázdným políčkem. Zde tato metoda přichází s rutinami, které by měli vést k zotavení překladače. Rutin je hned několik a nemusí být nijak složité – například vložení symbolu na vstup, nebo jeho přeskočení. Důležité je určit, jakou rutinu je potřeba použít, tedy určit teoretickou příčinu, jež vedla k detekci chyby. Proces doplňování tabulky o tyto rutiny je časově náročný a vyžaduje dobrou znalost jazyka a gramatiky. Některé políčka v tabulce nelze dosáhnout, u takových pak nezáleží, jakou rutinu zde umístíme.

Pokud autor dokáže správně odhadnout povahu chyb, je tato metoda velmi rychlá a efektivní. V opačném případě může způsobit nedefinovaný stav překladače, či dokonce uvíznutí v nekonečné smyčce.

4.5 Metoda Alan

Jak jsme si již řekli v dřívějších kapitolách, metoda Alan vznikla pro potřeby této bakalářské práce. Má některé společné rysy například s Panickým Módem, avšak vychází z jiného postupu.

Klíčovou informací pro tuto metodu je skutečnost, že pokud dojde k zastavení syntaktické analýzy během detekce chyby, na zásobníku již máme část *handle*, tedy část pravé strany pravidla. Co to značí? Že překladač sice možná neví, co chtěl autor programu příkazem v místě chyby říci, avšak může zjistit, co je potřeba pro pokračování syntaktické analýzy. A přesně toho my využijeme. Přesněji vezmeme *handle*, nalezneme pravidlo, které bychom pravděpodobně použili, kdyby ve vstupu nebyla chyba a provedeme vnucenou redukci. Tím upravíme zásobník, ale jak jsme uvedli u předchozí metody, musíme také kontrolovat vstup. Zde opět volíme množinu Follow pro neterminál z levé strany vybraného pravidla.

Tato metoda opět není nijak komplikované myšlenky, avšak pokud je synchronizační pravidlo vybráno dobře, dokáže rychle opravit velké množství chyb. Postup výběru synchronizačního pravidla v Projektu Alan si ukážeme v následující kapitole v části [5.9].

4.6 Další zajímavé metody

4.6.1 Backward/Forward Move

První zajímavou metodou pro doplnění je tzv. **Backward/Forward Move** od autorů Susan L. Graham a Stevena P. Rhodese [3]. Ti ji sice původně pojmenovali jako *Graham-Rhodens Method*, avšak ve většině literatury je odkazována s pomocí výše uvedeného označení.

Prvním krokem po detekci chyby je analýza kontextu okolí chyby během tzv. *condensation phase*. Po níž následuje *correction phase*, která upravuje zásobník a vstup, aby mohla syntaktická analýza pokračovat v překladu.

V první fázi redukujeme zásobník, abychom se dostali do stavu před chybou. Tomuto kroku se říká *Backward Move* a výsledkem je množina redukcí, s jejichž pomocí lze provést zotavení. *Forward move* je pak proveden, pokud je na vstupu další chyba, či je redukce možná pouze s obsahem zásobníku před úpravou, tedy obsahující chybu.

Tato metoda se nejvíce hodí pro precedenční syntaktickou analýzu [4, str. 530], avšak lze ji uplatnit i na LR parser. Zde však nepotřebujeme *Backward Move*, protože možné redukce jsou zřejmé z vrcholu zásobníku a dalšího symbolu na vstupu[4, str. 532].

4.6.2 Burke-Fisher metoda

Zajímavá metoda je také **Burke-Fisher**, která částečně vychází z Backward/Forward Move. Využívá současného zpracování vstupu dvěma parsery, přičemž jeden je o několik tokenů napřed. V situaci, kdy první parser nalezne chybu, využije se oblast, která je mezi oběma analyzátory k zjištění kontextu chyby a k modifikaci vstupu. Rozšířená Burke-Fisher metoda se používá například pro LALR syntaktickou analýzu [4, str. 532].

4.6.3 Phrase Level Recovery

Posledním zajímavým zástupcem je metoda, která je použita v mnoha překladačích a která dokáže opravit každý vstupní řetězec [1, str. 196]. Upravuje vstup změnou prefixu, tak aby mohla syntaktická analýza pokračovat i po nalezení chyby. Typickými změnami v této metodě jsou přidání/odebrání středníku, změna čárky na středník a podobné změny, které reprezentují nejčastější chyby programátorů zaviněné nepozorností, či překlepy. Všechny tyto změny musí navrhnout autor parseru. Opravy také musí být voleny s citem, protože zde hrozí uváznutí v nekonečné smyčce. Metoda má také potíže se vstupem, kdy se skutečná chyba nachází ještě před bodem detekce.

Kapitola 5

Implementace Projektu Alan

„Nevidíme daleko do budoucnosti, ale stejně před sebou zříme hromadu věcí, které je třeba udělat.“

Alan Turing

Projekt Alan je pojmenovaný po slavném matematikovi Alanovi Turingovi (1912 – 1954). Samotná aplikace nemá sice přímou souvislost s tímto velmi nadaným mužem, avšak snad volba jména nebude nikoho pohoršovat.

Hlavním smyslem webových stránek je poskytnout čtenáři možnost vyzkoušet si reálné fungování metod zotavování, ideálně bez nutnosti instalace dalších programů a aplikací. V následujících podkapitolách si uvedeme rozsah projektu a implementační detaily jak k překladači, tak k samotným metodám. Na konci kapitoly si také prozradíme, jaké nástroje byly použity pro testování.

5.1 Webové rozhraní

Pro implementaci webového rozhraní byl vybrán open-source framework Django [1] ve verzi 1.8, založený na programovacím jazyce Python. Překladač a řídící část projektu byly napsány v Pythonu ve verzi 3.4.

Django se může jevit jako příliš robustný pro projekt, kde je webové rozhraní prostředkem pro dostupnost aplikace, nikoliv jejím hlavním smyslem, avšak umožňuje vytvářet přehledný a udržovatelný kód, což může usnadnit případné budoucí rozšíření. Dalším důvodem volby frameworku je samotný programovací jazyk Python, který je populární jak na vědeckém poli, tak u velké skupiny programátorů.

Jako databáze pro uložení pravidel je využita implicitní databáze SQLite, jenž je součástí frameworku a tudíž není nutné instalovat další nástroje.

Aplikaci samotná je dostupná na webovém serveru: http://alanproject.pythonanywhere.com/. Pokud by byl například výpadek, či byly stránky nedostupné z jiných důvodů, v příloze **Zprovoznění projektu Alan** [B.4] je k nalezení kompletní návod, jak získat přístup na lokálním počítači bez nutnosti internetového serveru.

[1] Django (Version 1.8) [Computer Software]. (2015). Retrieved from https://djangoproject.com.

Obrázek 5.1: Úvodní stránka s informacemi o projektu

5.2 Lexikální analýza

V podkapitole [3.2] jsme si vysvětlili princip fungování lexikální analýzy. Nyní si ukážeme, jak vypadá v Projektu Alan, včetně praktického příkladu použití. Nejprve ze všeho ale musíme definovat vstup, který chceme zpracovávat.

Děje se tak na stránce http://alanproject.pythonanywhere.com/alan/, která je přístupná přes hlavní menu položkou Alan.

Obrázek 5.2: Stránka pro vložení zdrojového programu.

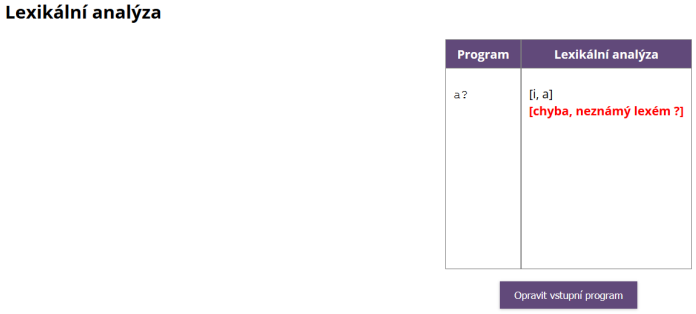
Zdrojový program můžeme vložit několika způsoby. Buď jej přímo vypíšeme do textového pole, který na počátku obsahuje pouze komentář `{ Zde napište svůj kód }`, nebo jej lze zkopírovat a následně vložit do pole. Třetí možností je nahrání souboru v podobě prostého textu (.txt). V takovém případě je potřeba nejdříve zvolit cestu k souboru a pak volbu

potvrdit tlačítkem **Nahrát soubor**.

Pokud jsme se vstupem spokojeni, můžeme použít tlačítko **Spustit lexikální analýzu**. Dle již zmiňované teoretické podkapitole bychom mohli očekávat, že nyní práci převezme syntaktická analýza a bude si postupně žádat další a další tokeny. V našem případě je ale situace jiná.

První velkou odlišností je skutečnost, že překladač není řízený syntaktickou analýzou a jednotlivé části jsou na sobě víceméně nezávislé. Spolupráce je zajištěna přes framework, který zajišťuje předávání informací, díky tomu také nebylo potřeba implementovat tabulku symbolů. Vstup scanner zpracovává najednou, nikoliv do nalezení dalšího lexému, či chyby a na výstup předává již kompletní seznam tokenů, který reprezentuje celý zdrojový kód. Token má tvar *[lexém, další informace]*, přičemž mezi další informace řadíme hodnotu čísla, nebo jméno proměnné.

Lexikální analýza také zpracovává celý vstup, bez ohledu na lexikální chyby. Ty mají speciální tvar tokenu *[chyba, doplňující informace o chybě]*. Uživatel tedy vidí kompletní seznam chyb, které může opravit návratem přes tlačítko **Opravit vstupní program**.



Obrázek 5.3: Lexikální analýza nalezla neznámý lexém „?“

Důvodů k těmto změnám bylo několik. Lexikální analýza není hlavním tématem práce, tím je zotavování ze syntaktických chyb. Proto je dobré předem vyloučit chyby v lexémech a umožnit i uživateli více se zaměřit na to podstatné. Implementace byla také ovlivněna podobou prezentace výsledků přes jednotlivé webové stránky a větší modularita napomohla v možnosti lépe znázorňovat informace o průběhu překladu.

5.3 Ukázkové programy

Pro potřeby projektu byla vytvořená sada ukázkových souborů, kterou lze stáhnout v komprimovaném archivu na manuálové stránce. Tyto soubory jsou využity i pro testovací účely aplikace. Pro lepší přehlednost mají názvy souborů předpony, pokud obsahují nějakou chybu: „le“ v případě lexikální chyby, „se“ s chybou syntaktickou.

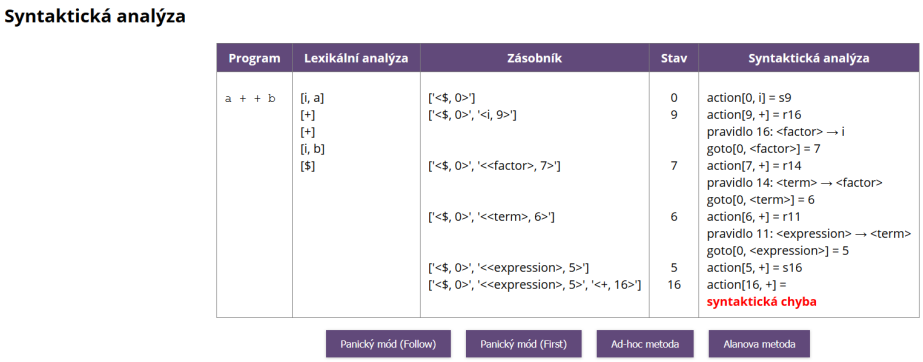
5.4 LR tabulka

Pro vytvoření LR tabulky existuje vícero algoritmů. Mezi základní zástupce uvedeme ty, které nalezneme i v materiálech pro předmět *Formální jazyky a překladače* [9, Ifj08-cz.pdf, str. 18].

- Simple LR (SRL) – jednoduchý algoritmus, který vytváří málo stavů, ale nemusí být dostatečně silný pro vytvoření tabulky pro všechny LR gramatiky
- Canonical LR – silnější než SLR, na druhou stranu vytváří mnoho stavů
- Lookahead LR (LARL) – nejsilnější a zároveň vytváří stejný počet stavů jako SLR

V našem případě budeme používat SLR algoritmus pro jeho jednoduchost a malý počet vytvořených stavů. Upravená gramatika FUN [2.2] je dostatečně zjednodušená, abychom byly schopni vytvořit tabulku pouze se SLR algoritmem. Postup je ukázán například v prezentaci [10, VYPe06b-cz.pdf, str. 11–24]. Úplnou tabulku lze nalézt v příloze [B.2].

5.5 Syntaktická analýza



Obrázek 5.4: Syntaktická analýza nalezla chybu ve vstupu „a + + b“

Jak funguje LR syntaktická analýza jsme uvedli již v teoretické části. Překlad je řízen LR tabulkou, která je uložena v seznamu (v Pythonu označovaném jako *List*), kdy každý záznam reprezentuje řádek, uložený v podobě slovníku (*Dictionary*). Části *action* i *goto* jsou tvořeny samostatnými seznamy. Výhodou tohoto uspořádání dat je možnost přistupovat k políčkům v tabulce pomocí `action[číslo][token]` a `goto[číslo][neterminál]`. Tabulka je uložena hned dvakrát. Jednou pro potřeby syntaktické analýzy, podruhé pro načtení a zobrazení v podobě HTML tabulky.

Na stránce http://alanproject.pythonanywhere.com/alan/parser/ je znázorněna tabulka s několika sloupci. Prvním je vstupní program, následovaný seznamem tokenů z lexikální analýzy. Třetím sloupcem znázorňujeme aktuální obsah zásobníku. Stav překladače, který používáme pro čtení z LR tabulky je ve čtvrtém sloupci. V posledním je pak výpis informací, který komentuje postup překladu a načtené hodnoty z tabulky. Pokud je na vstupu nalezena syntaktická chyba, uživateli je nabídnut rozcestník v podobě tlačítek pro určení metody zotavení.

5.6 Panický mód s množinou Follow

Pro implementaci této metody bylo využito materiálů [10, VYPe06b-cz.pdf, str. 27]. Jak jsme si uvedli v teoretickém představení Panického módu, je potřeba si stanovit logické celky programu, podle kterých bude parser prohledávat vstup. Nebudeme však určovat přímo synchronizační tokeny, ale neterminály, které symbolizují jednotlivé podčásti programu.

Pro náš případ jsme vybrali následující množinu: *{<expression>, <condition>, <statement>, <statement_list>}*. Ke každému neterminálu nyní určíme hodnoty Follow:

- **<statement_list>**: {$}
- **<statement>**: {$, ;, *r*, &, |}
- **<condition>**: {$, ;, *r*, &, |}
- **<expression>**: {$, ;, *r*, &, |, +, −,)}

Následně prohledáváme zásobník a hledáme shodu. Vše, včetně nalezeného neterminálu, odstraníme. Poté si určíme nahrazující neterminál, který představuje logicky vyšší celek. Až teprve z tohoto náhradníka použijeme množinu Follow pro prohledání vstupu a jeho zkrácení. Chceme tím simulovat pokrytí chyby, kdy ji nejen přeskočíme, ale také simulujeme provedení pravidla, čímž se snažíme minimalizovat hrozbu nekonzistentního stavu překladu. Aktualizujeme stav, na zásobník vložíme náhradní neterminál spolu s novým stavem a překlad může dále pokračovat.

5.7 Panický mód s množinou First

Obdoba klasické verze Panického módu byla implementována mírně odlišným způsobem. Ze začátku si opět volíme záchytnou množinu neterminálů, která zůstává stejná i pro tuto variantu. Poté si určíme hodnoty First:

- **<statement_list>**: {!, (, *i*, #}
- **<statement>**: {!, (, *i*, #}
- **<condition>**: {!, (, *i*, #}
- **<expression>**: {(, *i*, #}

Nezačneme však hledat shodu na zásobníku, nýbrž procházíme vstup a pátráme po synchronizačním tokenu s průniku všech množin First. Vše do tohoto tokenu přeskočíme. Poté určíme, který z neterminálů budeme hledat na zásobníku. Pokud by byl nalezený token „!“, pak by to znamenalo vyřazení možnosti *<expression>*. Prohledáváme tedy zásobník a vše až po shodu odstraníme.

V první variantě byl aktualizován vstup dle nalezeného neterminálu, avšak testování ukázalo nestabilitu tohoto řešení, která vedla i k uváznutí v nekonečné smyčce. Proto byla metoda rozšířena o další úpravu vstupu, tentokrát s množinou Follow pro nalezený neterminál na zásobníku. Tím byla chyba v metodě opravena, i za cenu ztráty přímočarosti postupu.

5.8 Ad-hoc metoda

Přestože má tato metoda poměrně jednoduchou myšlenku – doplnění prázdných políček v části *action* LR tabulky o zotavovací rutiny, jednalo se o časově nejnáročnější metodu na implementaci. Nejdříve bylo potřeba projít každé políčko tabulky a zjistit, při jaké chybě(chybách) se zde syntaktická analýza ocitne. Poté bylo potřeba vymyslet rutinu, která povede k zotavení a nakonec bylo zapotřebí řady testů, abychom se ujistily, že se parser nedostane do nedefinovaného stavu, či neskončí v nekonečné smyčce.

Situaci komplikovala i skutečnost, že část políček není reálně dosažitelná. Pro tento případ je zde rutina, která je však spíše kontrolní. Pokud by se totiž spustila, znamená to, že se nám překlad dostal do nesmyslného stavu, který již neodpovídá námi definovanému jazyku.

Zotavovacích rutin máme 6:

- **1**: Odebrání tokenu ze zásobníku
- **2**: Přidání proměnné i do vstupního řetězce
- **3**: Odstranění tokenu ze vstupního řetězce
- **4**: Přidání uzavírací) do vstupního řetězce
- **5**: Přidání + do vstupního řetězce
- **6**: Nedostupný stav

První rutina, tedy odebrání tokenu ze zásobníku se používá, pokud je například program ukončen středníkem, který tam však nepatří. Program $a + b; a;$ není syntaktický správný. Překladač rozpozná chybu, action[11, \$] = prázdné políčko, v tomto případě odkaz na rutinu f1. Zásobník je nyní obsahuje tyto položky: [<\$, 0>, < <statement>, 2>, <;, 11>, < <statement>, 2>, <;, 11>]. Pro aplikaci 2. pravidla $< statement\_list > \rightarrow < statement >$ je potřeba odstranit přebytečný středník. Poté může analýza dále pokračovat. Tato rutina se dá jistě použít i pro jiné druhy chyb, v této práci však autorka dávala ve většině případů přednost jiným rutinám, jedná se ale pouze o osobní preference.

Druhá rutina, která znamená přidání proměnné i do vstupního řetězce značí chybějící proměnnou, či naopak nadbytečný operátor. Například ve vstupu $a+b+$ buď chybí operand, například c, nebo je zde přebývající +. Pokud se spustí druhá rutina, do vstupu je přidána proměnná.

Třetí rutina, odstranění tokenu ze vstupního řetězce je způsobena například přebytečnou pravou závorkou. V tomto případě se překladač zastaví závorkou a je nutné ji přeskočit, aby překlad mohl pokračovat. Příkladem může být vstup $(a + b))$.

Čtvrtá rutina je jakoby opakem té třetí, protože na vstup naopak pravou závorku přidáme.

Pátá rutina, přidání + do vstupního řetězce, značí, že na vstupu jsou dvě proměnné hned za sebou, což gramatická pravidla nepovolují. Bylo by možné zde přidat jakýkoliv operátor, autorka si vybrala +. Poznamenáme, že nelze očekávat, že je to oprava, kterou by si autor programu přál, ale to není ani cílem metody. Zde je klíčové umožnit parseru dokončit jeho práci a je na programátorovi, aby si vstup opravil do kýžené podoby.

Šestá rutina je, jak jsme již zmiňovaly, kontrolního rázu. Jsou jimi označené nedostupné stavy, do kterých by se syntaktická analýza neměla správně nikdy dostat. Pokud však tato situace nastane, víme, že máme problém a náš překladač se nachází v nedefinovaném stavu.

Upravenou LR tabulku, přesněji doplněnou část *action* lze nalézt v příloze [B.3].

5.9 Alanova metoda

V tomto případě hledáme na zásobníku částečnou shodu pravé strany synchronizačního pravidla a provedeme redukci. To by byla hrubě načrtnutá teoretická představa Alanovy metody.

Pokud se však budeme chtít touto metodou zabývat blíže, například pro ověření její schopnosti zotavení, musíme si nejdřív položit několik otázek. První otázkou je, jak dle zásobníku zjistím nejvhodnější pravidlo. Druhou otázkou je, zda pouhá redukce bude stačit. Neuvedeme tím překladač do nedefinovaného stavu? Nevzniknou nám další chyby? Jaká bude časová náročnost těchto oprav?

Začneme od začátku. Tedy vlastně od konce. V takovém pořadí totiž budeme prohledávat pravidla. Důvodem je snaha minimalizovat násilnou opravu parseru. Hledáme také první částečnou shodu, protože víme, že nám na zásobníku něco chybí. Pokud najdeme vhodné pravidlo, vyprázdníme zásobník a vložíme do něj neterminál z levé strany pravidla se stavem dle *goto* části LR tabulky. Je třeba zdůraznit, že ne vždy takové pravidlo nalezneme. Tento stav je vidět hlavně s chybami způsobenými závorkami. Přebývající, či scházející závorky jsou chyby, se kterými si Alanova metoda neumí poradit. Důvodem je odstraněné pravidlo, které jsme zmiňovali v podkapitole [2.2].

Dále také poněkud zjednodušeně můžeme říci, že chyby ve vstupním programu jsou dvojího typu – přebytečný, či chybějící lexém. Simulace provedení redukce by mělo pokrýt chyby způsobené vynecháním lexémů, jakým je například **;**. Pro druhý typ chyb je potřeba dodatečně zkontrolovat vstup, zda nehrozí nedefinovaný stav překladače. Takže odpověď na druhou otázku je ano, je potřeba ještě dodatečně kontrolovat vstup. Kvůli dobré zkušenosti v případě Panického módu byla pro tento účel vybrána množina Follow, která byla určena pro neterminál levé strany synchronizačního pravidla.

5.10 Testování projektu Alan

Pro testování bylo využito více nástrojů a programů, abychom docílili co největšího pokrytí všech případných chyb a nedostatků. Nejprve jsem testovali grafické rozhraní, které bylo implementované pomocí HTML a CSS. Zde se zaměříme na přehlednost výsledného rozhraní, jeho funkčnost a dodržování standardů.

Samozřejmě nesmíme opomenout jádro projektu, překladač, či řídící kód, obojí napsané v jazyce Python. I přesto, že je mnoho potencionálních problémů kontrolováno přes framework, stále musíme otestovat, zda jednotlivé části překladače pracují, jak mají.

5.10.1 LR tabulka

Vytváření LR tabulky i s použitím SLR algoritmu je poměrně náročný proces a je jednoduché udělat chybu, která negativně ovlivní chování celého překladače. Pro kontrolu jak finální verze, tak postupu vytváření lze použít automatické nástroje pro tvorbu LR tabulek. V našem případě bylo využito webových stránek http://jsmachines.sourceforge.net/machines. Výhodou toto nástroje je možnost volby algoritmu a upozornění na konflikty typu shift-redukce, či redukce-redukce, pokud není gramatika jednoznačná (je víceznačná), to je, že parser asociuje k větě více než jeden derivační strom. Upravená gramatika FUN je jednoznačná, avšak pokud by čtenář někdy potřeboval řešit tvorbu tabulky pro víceznačnou gramatiku, nechť nahlédne do podkapitoly *3.4.5 Užití víceznačných gramatik* v učebním textu **Překladače**[11], kde jsou popsány velmi zajímavé možnosti řešení těchto konfliktů.

5.10.2 HTML a CSS

Pro zkontrolování HTML kódu byl použit automatický validátor od W3C, obdobně byl použit nástroj pro kontrolu kaskádových stylů W3C-CSS. Výsledná podoba stránek také byla kontrolována na vícero prohlížečích (Mozilla Firefox, Google Chrome, Internet Explorer, Safari) pro kontrolu jednotnosti ve vzhledu i funkcionalitě.

5.10.3 Python

Pro kontrolu dodržení stylistické konzistence a detekci potencionálních chyb se skvěle osvědčil nástroj **flake8**, který kontroluje dodržení jednotného stylu, ale i třeba nevyužité proměnné, či připojené knihovny.

5.10.4 Překladač

Pro testování samotného překladače byly vytvořeny testovací soubory, umístěné v adresáři **tests**. Každý logické celek je spuštěn se sadou příkladů se snahou o maximální pokrytí zdrojových kódů a detekci co největšího množství potencionálních chyb.

Kapitola 6

Porovnání metod zotavení

„Každý posudek je celkové nebo částečné porovnání dvojího systému údajů – systému člověka posuzovaného a systému člověka posuzujícího.“

Boris Vian

Uvedli jsme si několik metod zotavování při LR syntaktické analýze, vysvětlili jsme si, jaké principy využívají a u vybraných jsme si opsali implementační detaily.

Jak ale ve skutečnosti fungují? Jsou dostatečně silné pro zotavení? Jaké jsou jejich omezení? A která z nich je „nejlepší“?

V této kapitole metody podrobíme testům, abychom zjistili, jaké jsou jejich schopnosti při procesu zotavování a jak si vedou v porovnání se svými kolegy. Výsledky jsou taktéž umístěny na webových stránkách: http://alanproject.pythonanywhere.com/alan/comparison/

Pro potřebu porovnávání byla vytvořena sada testovacích vstupů obsahující syntaktickou chybu. Každá metoda byla poté opakovaně spouštěna nad těmito vstupy za cílem získat co nejpřesnější informace. Zohledňované aspekty hodnocení jsou následující:

- Zvládla metoda opravit chybu na vstupu?
- Kolik cyklů metody bylo zapotřebí?
- Celkový čas analýzy [ns]
- Celkové hodnocení

Nejdříve se ptáme, zda je metoda schopna chybu vůbec opravit. Pokud tomu tak není, je důležité, aby nedostala překlad do nedefinovaného stavu, či dokonce do nekonečné smyčky. V našem případě, pokud není metoda dostatečně silná na opravu, je syntaktická analýza zastavena a uživatel je informován, že na vstupu je chyba, se kterou si daná metoda neumí poradit. Poté zkoumáme počet oprav nutných pro další pokračování analýzy, tedy počet volání metody zotavení. Opakovaný zásah je nutný, pokud je na vstupu více chyb, ale také pokud metoda nesprávně vyhodnotí příčinu chyby. V nejhorším případě může „vytvořit“ falešné chyby nevhodnou úpravou vstupu či zásobníku. Časové údaje nechť laskavý čtenář uvažuje s rezervou, protože se jedná o přibližné hodnoty získávané opakovaným spouštěním a měření je ovlivněno mnoha faktory, jakými jsou specifikace testovacího stroje či aktuální

vytížení procesoru. Přesto však nejsou nic neříkající a mohou nám napovědět dost o chování jednotlivých metod. Důležitějším faktorem je celkové hodnocení, které se zaměřuje na kvalitu opravy. Tří stupňové hodnocení je inspirované článkem **A practical method for LR and LL syntactic error diagnosis and recovery**[2].

- **1** neboli **vynikající** – metoda opraví chybu tak, jak by učinil lidský čtenář
- **2** neboli **dobré** – rozumná oprava, která nevede k nekonzistentnímu stavu
- **3** neboli **špatné** – nedostatečná oprava, která může vést k vytvoření dalších chyb

Nyní si projdeme všechny testovací vstupy a popíšeme si, jak se jednotlivé metody chovaly a jaké byly jejich výsledky.

Prázdný program
Příklad: „“
Jako první zde máme zajímavý případ a to prázdný program. Některé jazyky, jako je C, C++ nebo Java považují prázdný program za chybný, protože neobsahuje klíčové slovo `main`. Skriptovací jazyky, jakými jsou PHP či Python naopak nemají s prázdným vstupem problém a jejich spuštění proběhne bez chybového hlášení.

V našem případě považujeme prázdný program za chybný, tudíž voláme metodu zotavení, aby situaci napravila. Úspěch má ovšem pouze Ad-hoc metoda, což není zase tak překvapivé. Právě Ad-hoc metoda je jediná, která má dost informací, aby provedla náležitou opravu – vložení proměnné do vstupu jako minimálního prvku tvořící správný zdrojový kód. Zbylé metody vyžadují kontext chyby (obsah zásobníku, vstup) a v tomto případě nemají s čím pracovat. Na jejich obranu je nutno dodat, že existuje jednoduchá náprava - vložením podmínky, která kontroluje délku vstupu vyřešíme schopnost detekce této chyby a metoda je poté schopna opravit vstup podle představ autora překladače.

Přebytečný středník
Příklad: „`a;`“ a „`a;;b`“
Další dva testovací příklady jsou ukázkou přebytečného středníku.
Ad-hoc i Alanova metoda si vedly velmi dobře – chybu opravily rychle a tak, jak by učinil lidský čtenář. Panický mód s množinou Follow také zotavení zvládl, avšak v případě `a;;b` přeskočil vstup až na konec, což není příliš šetrná oprava. Panický mód s množinou First pro změnu nezvládl opravu vstupu `a;`.

Chyby operátoru
Příklad: „`a+`“, „`a b`“, „`a + + b`“ a „`a<`“
Chybějící či přebývající operátory jsou dalším případem testování. Ad-hoc a Alanova metoda si opět vedly velmi dobře, Panický mód s množinou Follow si však nedokázal poradit se vstupem `a b`. Důvodem je „příliš brzký“ výskyt chyby, kdy na zásobníku nejsou potřebné informace. Pokud bychom vstup rozšířili, byť jen na podobu `a; a b`, Panický mód si s ní již dokáže poradit. Pro obě metody Panického módu by bylo možná spravedlivější, abychom testovali komplexnější příklady, ale zde aspoň vidíme jejich hlavní omezení. Poněkud překvapivá byla skutečnost, že Panický mód s množinou First zvládl opravit vstup `a + + b`.

Chyby v závorkách
Příklad: „`(a`“, „`(a))`“, „`)a;b`“ a „`()`“
Závorky, i vzhledem k podobě gramatiky, dělaly velké problémy většině metod. Alanova metoda dokázala opravit pouze vstup `(a`, Panický mód s množinou Follow pouze vstup `(a))` a její druh s množinou First propadl na celé čáře. Ad-hoc si poradila se všemi příklady tohoto typu bez problému.

Chybějící pravidlo se závorkami
Příklad: „`(!a)`“
Jak jsme již zmiňovali v podkapitole *Upravená gramatika FUN* [2.2], během úprav gramatiky bylo odstraněno pravidlo $< condition > \rightarrow (< condition >)$, které způsobuje, že vstup `(!a)` není syntakticky správný. S touto chybou si opět dokázala poradit pouze Ad-hoc metoda.

Zákeřné chyby
Příklad: „`(a+b)c)`“
Tento vstup je trochu zákeřný, protože zde máme chybějící operátor a ještě k tomu přebytečnou uzavírající závorku. Kromě již opakovaně chválené metody Ad-hoc zde zazářil Panický mód s množinou First. Oprava byla sice hodnocená jako dobrá, protože odstranila přebytečnou závorku i s proměnou `c`, ale zato tak učinila v jednom volání metody (na rozdíl od Ad-hoc, která potřebovala 2 volání), takže byla ve výsledku i o něco málo rychlejší.

	Chybný vstup	Panický mód (Follow)	Panický mód (First)	Ad-hoc metoda	Alanova metoda
1		ne / 1x / 0,015 ns / 3	ne / 1x / 0,010 ns / 3	ano / 1x / 0,383 ns / 1	ne / 1x / 0,010 ns /3
2	a;	ano / 1x / 0,262 ns / 1	ne / 1x / 0,362 ns / 3	ano / 1x / 0,200 ns / 1	ano / 1x / 0,424 ns / 1
3	a;;b	ano / 1x / 0,261 ns / 2	ano / 1x / 0,240 ns / 2	ano / 1x / 0,298 ns / 1	ano / 1x / 0,520 ns / 1
4	a+	ano / 1x / 0,263 ns / 2	ne / 1x / 0,210 ns / 3	ano / 1x / 0,264 ns / 1	ano / 1x / 0,235 ns / 1
5	a b	ne / 1x / 0,107 ns / 3	ne / 1x / 0,108 ns / 3	ano / 1x / 0,201 ns / 2	ano / 1x / 0,368 ns / 2
6	a + + b	ano / 1x / 0,350 ns / 1	ano / 1x / 0,297 ns / 2	ano / 1x / 0,259 ns / 1	ano / 1x / 0,559 ns / 1
7	a<	ano / 1x / 0,166 ns / 1	ne / 1x / 0,142 ns/ 3	ano / 1x / 0,211 ns / 1	ano / 1x / 0,477 ns / 1
8	(a	ne / 1x / 0,278 ns / 3	ne / 1x / 0,170 ns / 3	ano / 1x / 0,364 ns / 1	ano / 1x / 0,374 ns / 1
9	(a))	ano / 1x / 0,389 ns / 1	ne / 1x / 0,237 ns / 3	ano / 1x / 0,481 ns / 1	ne / 1x / 0,288 ns / 3
10	)a;b	ne / 1x / 0,176 ns / 3	ne / 1x / 0,102 ns / 3	ano / 1x / 0,367 ns / 1	ne / 1x / 0,087 ns / 3
11	()	ne / 1x / 0,126 ns / 3	ne / 1x / 0,165 ns / 3	ano / 1x / 0,507 ns / 1	ne / 2x / 0,398 ns / 3
12	(!a)	ne / 1x / 0,092 ns / 3	ne / 1x / 0,107 ns/ 3	ano / 1x / 0,513 ns / 1	ne / 2x / 0,443 ns / 3
13	(a+b)c)	ne / 1x / 0,177 ns / 3	ano / 1x / 0,443 ns/ 2	ano / 2x / 0,604 ns / 2	ne / 2x / 0,656 ns / 3

Obrázek 6.1: Tabulka výsledků porovnávacích testů

Kapitola 7

Závěr

„Na počátku bylo slovo, ale konec je stále ještě v nedohlednu.“

Wolfgang Eschker

A jsme na konci. Tedy ne tak docela, protože právě nyní je čas udělat si rekapitulaci a zhodnotit dosažené výsledky.

Na začátku jsme si vysvětlili základní pojmy týkající se překladačů a gramatik. Koho by daná problematiky lákala a chtěl by do ní nahlédnout více, nechť využije seznamu literatury, která obsahuje řadu velmi poučných a zajímavých děl.

Seznámili jsme se s upravenou gramatikou FUN, poznali jsme jednotlivé části překladače se zaměřením na lexikální a syntaktickou analýzu. Také jsme si definovali, co je to LR syntaktická analýza a jak pracuje se zásobníkem a LR tabulkou.

Poté jsme se vydali prozkoumat taje metod zotavování. Vysvětlili jsme si rozdíl mezi detekcí, korekcí a zotavením. Z výčtu mnoha metod zotavení jsme si vybrali několik zajímavých zástupců – Panický mód hned ve dvou variantách (s využitím množin Follow a First), Ad-hoc metodu, a zmínili jsme i další zajímavé metody (například Backward/Forward Move).

Představili jsme si také metodu, která byla vytvořena za účelem této bakalářské práce – Alanovu metodu, pojmenovanou po slavném matematikovi Alanu Turingovi.

Kromě teoretického základu jsme si také přiblížili způsob implementace přes webové rozhraní Projektu Alan, které umožňuje reálně vyzkoušet jednotlivé kroky překladu během lexikální a syntaktické analýzy a v případě syntaktické chyby i způsob zotavení pomocí výše uvedených metod. Zde jsme si přidali další praktické informace k jednotlivým částem překladače a metodám a také jsme si popsali způsob testování projektu.

Každou metodu jsme také vyzkoušely na testovací sadě a porovnali je mezi sebou.

Zjistili jsme velké výhody metody Ad-hoc, která si rychle a efektivně dokázala poradit s testovacími příklady. Protože využívá implicitní informace od autora překladače, nemusí prohledávat vstup či zásobník a dobře zvolenou rutinou provede zotavení. Její nevýhoda, jak jsme si uvedli již v teoretickém představení, spočívá v náročnosti sestavit množinu zotavovacích rutin, které by byly optimální pro konkrétní programovací jazyk a také v náročnosti doplnění LR tabulky o tyto rutiny. Patří ji však zasloužené první místo.

Druhou nejlepší byla Alanova metoda. Na zotavení potřebuje více času, než Ad-hoc metoda, ale stále má poměrně velké procento úspěšnosti. Největší potíž měla při opravování vstupu se závorkami. Tato skutečnost byla zapříčiněna odstraněním pravidla $< condition > \rightarrow (< condition >)$. Pokud by pravidlo nebylo odstraněno (a byla by aktualizována LR tabulka), je dost možné, že by metoda dokázala opravit i tuto skupinu chyb. Další alternativou je spojení s Ad-hoc metodou. Ta by však měla omezenější množství rutin a spouštěla by se pouze v případě, kdyby Alanova metoda nebyla schopná zotavení.

Panický mód s množinou Follow dopadl ve výsledku o něco hůře než Alanova metoda. Opět zde máme problém se závorkami a největším omezením metody je závislost na předešlém kontextu programu. Zjednodušeně řečeno, čím dál od začátku programu se chyba vyskytuje, tím je větší pravděpodobnost, že metoda dokáže úspěšně provést zotavení.

Nejhůře pak dopadl Panický mód s množinou First, což jsme však predikovali již v teoretické části, především kvůli větší vhodnosti pro LL syntaktickou analýzu. Přesto v některých případech příjemně překvapila a v jednom byla dokonce ještě rychlejší než Ad-hoc metoda.

Pokud bychom chtěli implementovat rozsáhlejší překladač s větším množství pravidel, je rozumné (a v praxi běžné) uvažovat spojení více metod zotavení. Autor překladače zde může zvolit citlivé varianty, které povedou k co nejpřesnější detekci příčiny chyby a co možná nejméně násilné cestě k zotavení. Z metod, které byly implementovány v této práci se pro LR analýzu jeví nejpříznivější kombinace Alanovy a Ad-hoc metody, zajímavých výsledků by mohla dosáhnout i kombinace Panického módu s množinou Follow a Ad-hoc metody. Panický mód s množinou First by bylo zajímavé otestovat na LL parseru, avšak v případě LR gramatiky zřejmě není příliš praktický.

V této práci by autorka ráda vyzvedla úspěšnou implementaci čtyř zástupců metod zotavení, přičemž originální Alanova metoda dopadla ve srovnání s již dříve definovanými velmi dobře. Nepříliš vhodnou redukcí pravidel byla sice omezena síla jednotlivých metod, avšak to ukázalo až konečné testování. Oprava by pak znamenala znovu vytvoření LR tabulky a potřebu změnit Ad-hoc metodu, což může být úkol budoucna, či pro jiného autora.

Webové rozhraní má uspokojivou podobu, která by však v rukou zkušenějšího návrháře mohla vypadat o něco více profesionálně. Autorka je však spokojená s podobou reprezentovaných informací.

Většinou je také zvykem na závěr navrhnout případná vylepšení či rozšíření, která by mohla být vykonána v budoucnu, ať už autorem samotným, tak někým jiným. V tomto případě bychom mohli navrhnout a implementaci více metod, nebo změnit webového rozhraní, které by mohlo být ještě více interaktivní. Parser by mohl být například krokovatelný a mohla by zde být i volba návratu o několik kroků zpět. Uživatel by tedy byl schopen ještě lépe vidět proces analýzy, tak jak jdou jednotlivé úkony za sebou.

Další možností, v rámci myšlenky o minimalismu, kterou jsme uvedli v úvodu, vytvořit překladač, který by se podobal *Unix shellu*. Ovládal by se pomocí příkazové řádky, kdy by přijal vstupní program v podobě řetězce (nebo také v podobě textového souboru) a pomocí přepínačů by uživatel zadal, zda chce pouze analýzu lexikální, nebo i syntaktickou, jakou metodu zotavování chce použít, případně zda chce výstup zobrazit v konzoly, či uložit do výstupního souboru. Zobrazit všechny informace v tomto prostředí by byla větší výzva, ale výsledek by mohl být zajímavý a pro některé uživatele příjemnější, než užívaní webových stránek.

„Dokonalosti není dosaženo tehdy, když už není co přidat, ale tehdy, když už nemůžete nic odebrat.“ Dokonalosti zřejmě nebude dosaženo nikdy, ale autorka pevně věří, že i kdyby její práce využil jediný student, který by díky ní získal nové poznatky a vědomosti, veškerá práce by se vyplatila. Protože nejde o dokonalost, jde o sdílení poznatků a vědomostí.

Literatura

[1] Aho, A. V.: *Compilers: principles, techniques & tools*. Boston: Addison Wesley, 2007, ISBN 0-321-48681-1.

[2] Burke, M. G. a Fisher, G. A.: *A practical method for LR and LL syntactic error diagnosis and recovery. ACM Transactions on Programming Languages and Systems (TOPLAS)*, ročník 9, č. 2, April 1987: s. 164–197.

[3] Graham, S. L. a Rhodes, S. P.: *Practical syntactic error recovery. Communications of the ACM*, ročník 18, č. 11, November 1975: s. 639–65

[4] Grune, D.: *Parsing techniques: a practical guide*. New York: Springer, 2008, ISBN 978-0-387-20248-8.

[5] Johnson, S. C.: *Yacc: Yet Another Compiler-Compiler*. `http://dinosaur.compilertools.net/yacc/`, 2006-03-29 [cit. 2016-05-20].

[6] Lesk, M. E. a Schmidt, E.: *Lex - A Lexical Analyzer Generator*. `http://dinosaur.compilertools.net/lex/`, 2006-03-29 [cit. 2016-03-21].

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

[8] Meduna, A.: *Formal languages and computation: models and their applications*. CRC Press, 2014, ISBN 978-1-4665-1345-7.

[9] Meduna, A.: *Materiály do předmětu Formální jazyky a překladače*. `https://www.fit.vutbr.cz/study/courses/IFJ/public/materials/`, 2015-09-13 [cit. 2015-01-10].

[10] Meduna, A.: *Materiály do předmětu Výstavba překladačů (v angličtině)*. `https://www.fit.vutbr.cz/study/courses/VYPe/public/materials/`, 2015-09-13 [cit. 2015-01-10].

[11] Milan Češka, Tomáš Hruška, Miroslav Beneš: *Překladače*. Auerbach Publications, 1993, ISBN 978-80-214-0491-5.

Přílohy

Seznam příloh

A Obsah CD **36**

A.1 Technická zpráva 36

A.2 Projekt Alan 36

A.3 Archiv ukázkových programů 36

B Užitečné informace o Projektu Alan **37**

B.1 Adresářová struktura projektu 37

B.2 LR tabulka 38

B.3 LR tabulka pro Ad-hoc metodu 39

B.4 Zprovoznění Projektu Alan 40

Příloha A

Obsah CD

A.1 Technická zpráva

Složka `documentation` obsahuje technickou zprávu ve formátu `.pdf` a podsložku `src` se zdrojovými soubory pro sestavení technické zprávy.

A.2 Projekt Alan

Kompletní implementace projektu. Obsah složky `alan_project` a adresářová struktura jsou popsány v [B.1]. Pro sestavění postupujte dle návodu [B.4].

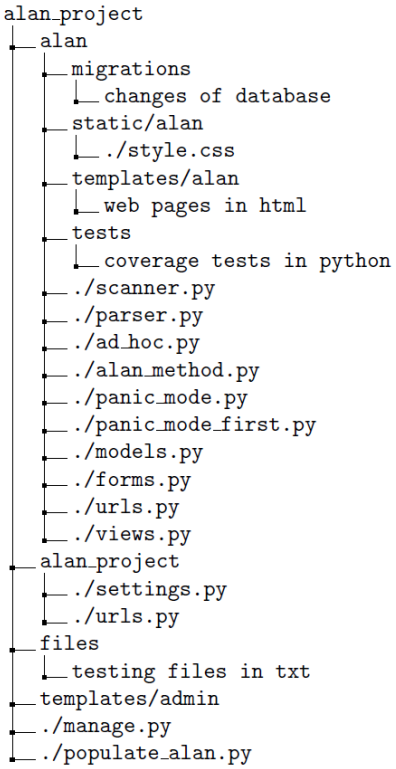
A.3 Archiv ukázkových programů

Ve složce `alanfiles.zip` naleznete soubory, které jsou využívány i pro testovací účely aplikace. Pro lepší přehlednost mají názvy souborů předpony, pokud obsahují nějakou chybu: *.le* v případě lexikální chyby, *.se* s chybou syntaktickou.

Příloha B

Užitečné informace o Projektu Alan

B.1 Adresářová struktura projektu



Obrázek B.1: Adresářová struktura projektu.

B.2 LR tabulka

Stav	;	i	#	r	+	-	*	/	&	|	!	(	)	$
0		s9	s10								s4	s8		
1														acc
2	s11			s12					s13	s14				r2
3	r6			r6					r6	r6				r6
4		s9	s10								s4	s8		
5	r8			r8	s16	s17			r8	r8				r8
6	r11			r11	r11	r11	s18	s19	r11	r11			r11	r11
7	r14			r14	r14	r14	r14	r14	r14	r14			r14	r14
8		s9	s10									s8		
9	r16			r16	r16	r16	r16	r16	r16	r16			r16	r16
10	r17			r17	r17	r17	r17	r17	r17	r17			r17	r17
11		s9	s10								s4	s8		
12		s9	s10								s4	s8		
13		s9	s10								s4	s8		
14		s9	s10								s4	s8		
15	r7			r7					r7	r7				r7
16		s9	s10									s8		
17		s9	s10									s8		
18		s9	s10									s8		
19		s9	s10									s8		
20					s16	s17							s29	
21														r1
22	r3			r3					r3	r3				r3
23	r4			r4					r4	r4				r4
24	r5			r5					r5	r5				r5
25	r9			r9	r9	r9	s18	s19	r9	r9			r9	r9
26	r10			r10	r10	r10	s18	s19	r10	r10			r10	r10
27	r12			r12	r12	r12	r12	r12	r12	r12			r12	r12
28	r13			r13	r13	r13	r13	r13	r13	r13			r13	r13
29	r15			r15	r15	r15	r15	r15	r15	r15			r15	r15

Stav	<statement_list>	<statement>	<condition>	<expression>	<term>	<factor>
0	1	2	3	5	6	7
1						
2						
3						
4			15	5	6	7
5						
6						
7						
8				20	6	7
9						
10						
11	21	2	3	5	6	7
12			22	5	6	7
13			23	5	6	7
14			24	5	6	7
15						
16					25	7
17					26	7
18						27
19						28
20						
21						
22						
23						
24						
25						
26						
27						
28						
29						

Obrázek B.2: LR tabulka zjednodušené gramatiky FUN.

B.3 LR tabulka pro Ad-hoc metodu

Stav	;	i	#	r	+	-	*	/	&	|	!	(	)	$
0	②	s9	s10	②	②	②	②	②	②	②	s4	s8	③	②
1	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	acc
2	s11	⑥	⑥	s12	⑥	⑥	⑥	⑥	s13	s14	⑥	⑥	⑥	r2
3	r6	⑥	⑥	r6	⑥	⑥	⑥	⑥	r6	r6	⑥	⑥	⑥	r6
4	③	s9	s10	③	③	③	③	③	③	③	s4	s8	②	②
5	r8	③	③	r8	s16	s17	③	③	r8	r8	③	③	③	r8
6	r11	⑥	⑥	r11	r11	r11	s18	s19	r11	r11	⑥	⑥	r11	r11
7	r14	⑥	⑥	r14	r14	r14	r14	r14	r14	r14	⑥	⑥	r14	r14
8	③	s9	s10	③	③	③	③	③	③	③	③	s8	②	②
9	r16	③	⑤	r16	r16	r16	r16	r16	r16	r16	③	③	r16	r16
10	r17	⑤	⑤	r17	r17	r17	r17	r17	r17	r17	③	③	r17	r17
11	③	s9	s10	③	③	③	③	③	③	③	s4	s8	③	①
12	②	s9	s10	③	②	②	②	②	②	②	s4	s8	③	②
13	②	s9	s10	②	②	②	②	②	③	②	s4	s8	③	②
14	②	s9	s10	②	②	②	②	②	②	③	s4	s8	③	②
15	r7	⑥	⑥	r7	⑥	⑥	⑥	⑥	r7	r7	⑥	⑥	⑥	r7
16	②	s9	s10	②	③	②	②	②	②	②	②	s8	③	②
17	②	s9	s10	②	②	③	②	②	②	②	②	s8	③	②
18	②	s9	s10	②	②	②	③	②	②	②	②	s8	③	②
19	②	s9	s10	②	②	②	②	③	②	②	②	s8	③	②
20	④	④	④	④	s16	s17	③	③	④	④	③	③	s29	④
21	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	r1
22	r3	⑥	⑥	r3	⑥	⑥	⑥	⑥	r3	r3	⑥	⑥	⑥	r3
23	r4	⑥	⑥	r4	⑥	⑥	⑥	⑥	r4	r4	⑥	⑥	⑥	r4
24	r5	⑥	⑥	r5	⑥	⑥	⑥	⑥	r5	r5	⑥	⑥	⑥	r5
25	r9	⑥	⑥	r9	r9	r9	s18	s19	r9	r9	⑥	⑥	r9	r9
26	r10	⑥	⑥	r10	r10	r10	s18	s19	r10	r10	⑥	⑥	r10	r10
27	r12	⑥	⑥	r12	r12	r12	r12	r12	r12	r12	⑥	⑥	r12	r12
28	r13	⑥	⑥	r13	r13	r13	r13	r13	r13	r13	⑥	⑥	r13	r13
29	r15	⑤	⑤	r15	r15	r15	r15	r15	r15	r15	③	③	r15	r15

Obrázek B.3: LR tabulka zjednodušené gramatiky FUN pro potřeby Ad-hoc metody.

B.4 Zprovoznění Projektu Alan

Součástí bakalářské práce je kompletní implementace na přiloženém CD disku [A], díky čemuž lze projekt zprovoznit na lokálním počítači. Je nezávislý na operačním systému a byl úspěšně spuštěn na Microsoft Windows 7 a 8, Red Hat Fedora 23 i na Apple iOS.

Následující příkazy se mohou jemně lišit dle operačního systému. Pro doplňující informace lze využít oficiální dokumentaci Django projektu.
Jako první je potřeba mít nainstalovány potřebné nástroje:

- Python3
- Python3-devel
- Pip (či jiný instalační nástroj)
- Django ve verzi 1.8 a vyšší

Pro instalaci Djanga lze využít výše zmíněný nástroj *pip*. Pro kontrolu, zda je framework řádně nainstalován stačí spustit přes terminál *python3* a napsat příkaz `import django`.

Projekt se nachází ve složce *alan_project*. Pro jeho nastavení a spuštění je potřeba být v terminálu ve uvnitř této složky, na stejné úrovni na jaké je umístěn klíčový soubor *manage.py*. Právě přes něj je projekt kontrolován.

Nejdříve nastavíme databázi. Dobrou zprávou je, že implicitní *SQLite* již byl instalován spolu s frameworkem. Nyní stačí pouze zadat následující příkazy pro vytvoření databáze:

```
python3 manage.py makemigrations
```

```
python3 manage.py migrate
```

Pokud vše proběhlo dobře, nemělo by se objevit žádné chybové hlášení. Nyní máme definovanou databázi, ale chybí nám obsah – data. Ty vložíme pomocí skriptu napsaného taktéž v Pythonu.

```
python3 populate_python.py
```

Nyní již zbývá pouze server spustit pomocí příkazu:

```
python3 manage.py runserver
```

Projekt by měl být v této chvíli dostupný na adrese:

http://127.0.0.1:8000/