VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

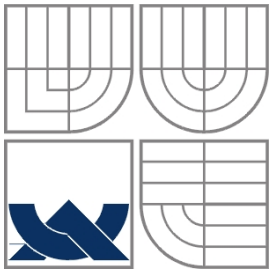
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

POKROČILÉ ZOTAVENÍ Z CHYB BĚHEM SYNTAKTICKÉ ANALÝZY SHORA DOLŮ

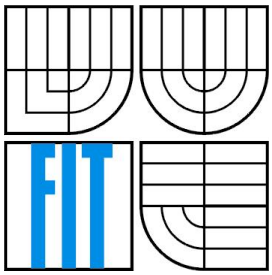
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Alena Obluková
AUTHOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

POKROČILÉ ZOTAVENÍ Z CHYB BĚHEM SYNTAKTICKÉ ANALÝZY SHORA DOLŮ

ADVANCED ERROR RECOVERY DURING TOP-DOWN PARSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
Alena Obluková

VEDOUCÍ PRÁCE
SUPERVISOR
Prof. RNDr. Alexander Meduna, CSc.

BRNO 2015

Abstrakt

Syntaktický analyzátor je jednou z nejdůležitějších částí překladače při často používaném přístupu syntaxí řízený překlad. Při tomto přístupu řídí syntaktický analyzátor sémantické akce a generování abstraktního syntaktického stromu. Jestliže je na vstupu chyba, syntaktický analyzátor nemůže pokračovat a celý překlad musí skončit. Proto je nesmírně důležité, aby syntaktický analyzátor byl schopen se zotavit z chyb, tedy aby i po nalezení chyby byl schopen dále pokračovat.
V této bakalářské práci jsou popsány metody zotavení z chyb, podrobně je popsána metoda Kontext zotavení z chyb pomocí pokračování, konkrétně pokračování u LL syntaktického analyzátoru. Přestože tato metoda není příliš známá, není příliš složitá na vysvětlení a na implementaci. Může tedy být snadno použita při výuce pro demonstraci zotavení z chyb při syntaktické analýze shora dolů.

Abstract

Parser is one of the most important parts in compiler since Syntax-Directed Translation is often used. This approach means that parser controls semantic actions and generation of syntax tree. When the input contains an error, parser cannot continue and the whole compiler has to stop. Therefore, it is important to have parser with error recovery, so when error occurs parser is able to continue.
There are several error-recovery strategies and methods. In this paper is described Acceptable-sets derived from continuations specifically continuation in LL parsers. However it is not so well known method it is easy to explain and to implement. It can be used in the lesson to demonstrate error recovery in top-down parser.

Klíčová slova

syntaktická analýza, LL gramatika, zotavení z chyb, bezkontextové gramatiky

Keywords

syntax analysis, LL grammar, error recovery, Context-free grammar

Citace

Obluková Alena:, Pokročilé zotavení z chyb během syntaktické analýzy shora dolů, Brno, FIT VUT v Brně, 2015

Pokročilé zotavení z chyb během syntaktické analýzy shora dolů

Prohlášení

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

.....
Alena Obluková
Datum 17. 5. 2015

Poděkování

Ráda bych poděkovala panu Prof. RNDr. Alexandru Medunovi, CSc., který mě po celou dobu bakalářské práce pomáhal a vedl.

Obsah

1 Úvod

Mezi odvětví informatiky patří formální jazyky a překladače. Jsou její nezbytnou součástí. Jazyky se mění a jsou těžko uchopitelné. Formální jazyky jsou tu proto, aby tento problém odstranily. Pomocí nich je možné popsat programovací jazyky, čehož se v praxi využívá.

Bez překladačů by nemohla informatika existovat v takové podobě, v jaké ji známe dnes. Počítač současné doby je schopen vykonávat strojové instrukce. Tyto strojové instrukce tvoří jeho instrukční soubor, specifický pro použitý procesor, případně mikroprocesor. Počítač by si neuměl poradit s programovacími jazyky jako je Pascal či C, nebýt překladačů či interpretů. Překladač má za úkol přeložit, jinými slovy přepsat program v daném programovacím jazyce do strojových instrukcí, se kterými počítač již umí zacházet.

Postup překladače, který má ze zdrojového programu vytvořit ekvivalentní výstupní program, se skládá z několika logických fází. První je lexikální analyzátor. Lexikální analyzátor má za úkol číst vstupní program a rozdělovat ho na lexikální jednotky, tzv. tokeny. Tokenem může být identifikátor, číslo nebo klíčové slovo. Posloupnost tokenů se předkládá syntaktickému analyzátoru, někdy překládanému jako parser. Tato bakalářská práce se soustředí na syntaktický analyzátor a syntaktickou analýzu- dále jen analyzátor a analýza. Syntaktický analyzátor dostává na vstup řetězec tokenů z lexikálního analyzátoru. Jeho výstupem je derivační strom. Derivační strom je tvořen podle gramatických pravidel. Pokud je program napsán syntakticky správně, znamená to, že se podařilo z pravidel úspěšně vytvořit derivační strom. Jestliže se derivační strom vytvořit nepodařilo, syntaktický analyzátor bez schopnosti zotavení z chyb končí. Tím končí i překlad. Syntaktický analyzátor je srdcem překladače. To on spouští lexikální analyzátor s požadavkem o další token. To on dává povel sémantickému analyzátoru, aby zpracoval derivační strom. Sémantický analyzátor má za úkol zkontrolovat, zda je program napsán sémanticky správně. Jinými slovy provádí například kontrolu typů a deklaraci proměnných. Jeho výstupem je abstraktní syntaktický strom. Po sémantickém analyzátoru přichází na řadu generátor vnitřního kódu, který z abstraktního stromu vytvoří vnitřní reprezentaci kódu nazývanou vnitřní kód. Vnitřní kód je většinou 3-adresný. Další složkou je optimalizace kódu a s ní související generátor cílového kódu, který z optimalizovaného kódu udělá cílový kód.

Syntaktický analyzátor, kterým se tato práce zabývá, může postupovat při své práci shora dolů nebo zdola nahoru. Při postupu zdola nahoru postupuje syntaktický analyzátor od vstupního řetězce směrem k počátečnímu, startovnímu symbolu. Tento přístup se používá, pokud dvě nebo více pravidel mají stejnou pravou stranu nebo jedná-li se o nejednoznačnou gramatiku. Mezi syntaktické analyzátory pracující zdola nahoru patří precedenční syntaktický analyzátor, který se implementuje jednoduše, ale je slabý. Dále pak LR analyzátor, ten je silnější, ale hůře implementovatelný. Typickým příkladem, kdy se používá postup zdola nahoru, je při matematických výrazech, ve kterých některý operátor (například součin) má přednost před jiným (například před sčítání). Druhý způsob postupu je syntaktická analýza shora dolů. Tento postup je implementován pomocí LL analyzátoru. LL v názvu značí "*left to right, left most*". LL analyzátor může být implementován rekurzivním sestupem nebo prediktivní syntaktickou analýzou. Rekurzivní sestup je v praxi běžný, nicméně prediktivní syntaktická analýza je názornější. Tato bakalářská práce má představit metody zotavování z chyb při syntaktické analýze shora dolů. Aby byly metody na první pohled zřejmé a pochopitelné, používá se v této práci učebnicová metoda prediktivní syntaktické analýzy. Výsledkem prediktivní syntaktické analýzy je levý rozbor. Pojmem levý rozbor je se rozumí posloupnost pravidel, která je použita v nejlevější derivaci pro vstupní řetězec.

Chyby se dělí do 4 základních skupin. Lexikální, sémantické, logické a syntaktické chyby. Lexikální chyby jsou odhaleny lexikálním analyzátorem. Mezi lexikální chyby patří chybně napsaný řetězec, nevalidní číslo a chyby způsobené překlepy. Další skupinou jsou sémantické, neboli

významové chyby. Tyto chyby jsou odhaleny sémantickým analyzátorem. Příkladem může být porovnávání nekompatibilních proměnných a chybějící deklarace proměnných. Třetí skupinu tvoří nejzáludnější chyby, chyby logické. Logická chyba nebývá odhalena překladačem. Logické chyby jsou takové chyby, které způsobují, že program lze přeložit, program pracuje, ale nefunguje podle představ programátora. Typickými chybami je například záměna porovnání za přiřazení v podmínce, nekonečný cyklus. Poslední skupinou jsou chyby syntaktické. Tato práce se zabývá pouze chybami syntaktickými a proto nadále slovem chyba je rozuměna chyba syntaktická. Příkladem syntaktické chyby je vynechaný středník, chybějící či přebývající operátor. Jak již bylo zmíněno, pokud syntaktický analyzátor bez zotavení narazí na chybu, končí překlad. Toto je velice nešťastné, neboť se nenajdou zbývající chyby v programu. Aby se nalezlo co nejvíce chyb v programu, čímž by se pomohlo autorovi programu, je nutné implementovat zotavení z chyb.

Tato práce se zabývá metodami zotavování z chyb při syntaktické analýze shora dolů. Porovná a představí metody spadající do pěti tříd. Dělení tříd se liší podle literatury. V této práci je zvoleno nejpodrobnější dělení dle [3] zkombinované s dělením v knize v tzv. Dračí knize [1]. Metody jsou děleny podle kontextu, tedy podle toho, o jak velké okolí se při zotavování opírají. Jsou to: Metody globálního zotavení z chyb (*Global error handling methods*), Zotavení na úrovni fází (*Phase-level recovery*), Metody lokálního zotavení z chyb (*Local error handling methods*), Suffixové metody (*Suffix methods*) a Ad Hoc metody (*Ad Hoc methods*). Nejpodrobněji jsou rozebrány Metody lokálního zotavení z chyb.

Formální jazyky využívají gramatiky. Gramatiky mohou být jednoduché, například jazyk Pascal lze popsat pomocí zhruba 55 pravidel. Na popsání jazyka C++ je však potřeba pravidel alespoň desetinásobek. V úvodu je definována gramatika, která napříč prací pomáhá k porozumění jednotlivých metod. Gramatika není Turingovsky kompletní. Chybí základní programovací konstrukce, jako je podmínka a smyčka. K jejímu popsání je potřeba pouze 15 pravidel v rozvité formě. V praxi by vypadala pravidla jistě jinak, zde jsou však schválně upravena, aby se daly ukázat typické chyby (například vynechaná závorka či chybějící středník). Gramatika je inspirována jazykem Pascal.

V rámci bakalářské je implementován syntaktický analyzátor pracující shora dolů. Tento syntaktický analyzátor využívá zotavení z chyb pomocí metody Kontex tvořený pokračovací metodou - Pokračování u LL syntaktického analyzátoru. Tato metoda je zajímavá tím, že používá dvě gramatiky. Kromě obyčejné gramatiky používá i pokračovací gramatiku.

Bakalářská práce je rozdělena do několika kapitol.V úvodní **kapitole 2** jsou vysvětleny základní definice, jako například pojmy abeceda, jazyk, derivace a gramatika.

Kapitola 3 má za úkol popsat syntaktickou analýzu. Na začátku této kapitoly je v příkladu 2 uvedena jednoduchá gramatika. Tato gramatika se používá v celé bakalářské práci. Kapitola se zaměřuje na syntaktickou analýzu shora dolů a na tvorbu LL tabulky. S tvorbou LL tabulky souvisí pojmy jako je množina Empty, First, Follow a Predict. Pro tvorbu LL tabulky je použita již zmíněná gramatika z příkladu 2. LL tabulka vytvořená na konci této kapitoly je používána dále v popisech jednotlivých metod.

Kapitola 4 otevírá téma zotavování z chyb při syntaktické analýze shora dolů. Na začátku kapitoly jsou rozděleny typy chyb, jsou vypsány hlavní důvody, proč je nutné se z chyb umět zotavit. Nadále kapitola pokračuje základním dělením metod. Nejobsáhlejší je třída metod lokálního zotavení z chyb. Tato třída obsahuje následující metody: Panický mód (*Panic-mode recovery*), Follow set zotavení z chyb (*Follow set error recovery*), Metoda zotavování z chyb pomocí vkládání symbolů (*Insertion-only error correction*), Metoda lokálního zotavování z chyb na základě nejnižší ceny (*Locally least-cost error recovery*) a Kontext tvořený pokračovací metodou (*Acceptable-set derived from continuation*). Poslední jmenovaná metoda je podrobněji rozebrána - je popsána pokračovací gramatika (*continuation grammar*) a Pokračování u LL syntaktického analyzátoru (*Continuation in LL parser*).

V **kapitole 5** je popsána implementace zotavení z chyb pomocí metody Kontex tvořený pokračovací metodou.

Závěrečná **kapitola 6** shrnuje celé téma. Jsou v ní uvedeny vyvozené závěry a návrhy na další rozšíření této bakalářské práce. Mezi návrhy patří například grafické zpracování implementované metody zotavení z chyb či implementace metody pomocí LR analyzátoru.

2 Základní definice a pojmy

Jako každá vědní disciplína, i formální jazyky mají své základní pojmy, termíny. V této kapitole jsou popsány definice, na jejichž základě bude práce vytvářet další definice. Veškeré definice jsou přejaty ze zdroje [5], [6]. Základním pojmem je jazyk, k jehož popsání je potřeba znát pojem abeceda, symbol, řetězec. Dále je zde definován termín bezkontextová gramatika a s tím související pojem derivace.

2.1 Abeceda, řetězec, jazyk

Definice 1 (abeceda)
Abeceda je konečná, neprázdná množina elementů, které nazýváme symboly.

Symboly lze spojovat za sebe. Vznikají tak *řetězce*.

Definice 2 (řetězec)
Nechť Σ je abeceda.

- ε je řetězec nad abecedou
- pokud *x* je řetězec nad Σ a $a \in \Sigma$, potom *xa* je řetězec nad abecedou

ε značí tzv. prázdný řetězec = neobsahuje žádný symbol.

Σ^* značí množinu všech řetězců nad abecedou Σ, $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Definice 3 (délka řetězce)
Nechť x je řetězec nad abecedou Σ. Délka řetězce x, |x|, je definována:

- pokud $x = \varepsilon$, pak $|x| = 0$
- pokud $x = a_1 \ldots a_n$, pak $|x| = n$ pro $n \geq 1$ a $a_i \in \Sigma$ pro všechna $i = 1, \ldots, n$

Definice 4 (konkatenace řetězců)
Nechť *x* a *y* jsou dva řetězce nad abecedou Σ. Konkatenace *x* a *y* je řetězec *xy*.

Definice 5 (mocnina řetězce)
Nechť *x* je řetězec nad abecedou Σ.Pro $i \geq 0$, i-tá mocnina řetězce *x*, *xi*, je definována:

- $x^0 = \varepsilon$
- pro $i \geq 1$: $x^i = xx^{i-1}$

Definice 6 (suffix řetězce)
Nechť *x* a *y* jsou dva řetězce nad abecedou Σ; *x* je sufixem *y*, pokud existuje řetězec *z* nad abecedou Σ, přičemž platí *zx* = *y*.

Definice 7 (podřetězec)
Nechť *x* a *y* jsou dva řetězce nad abecedou Σ. *x* je podřetězec *y*, pokud existují řetězce *z*, *z'* nad abecedou Σ přičemž platí *zxz'* = *y*.
Pokud $x \notin \{\varepsilon, y\}$, pak *x* je vlastní podřetězec řetězce *y*.

Je možné vytvořit nekonečně mnoho řetězců. Aby se s řetězci lépe pracovalo, spojují se do skupin. Řetězce tvořené stejnou abecedou pak tvoří *jazyk*. Jazyk je množinou řetězců nad danou abecedou.

Definice 8 (jazyk)
Nechť Σ* značí množinu všech řetězců nad Σ. Každá podmnožina $L \subseteq \Sigma^*$ je jazyk nad Σ.

Jazyk může být konečný či nekonečný. Nekonečný jazyk je tvořen nekonečným počtem řetězců.

2.2 Bezkontextová gramatika, derivace

Jazyk není formulován jen nahodilými symboly jdoucími za sebou. Jazyk je formulován gramatikou, gramatickými pravidly. Stejně jako český či anglický jazyk má svá pravidla, i programovací jazyk musí mít svou syntax a sémantiku. Gramatická pravidla programovacího jazyka nejsou zdaleka tak složitá, jako u běžné řeči, jinak by ho bylo velmi obtížné strojově zpracovat. Ve této práci je uvedená gramatika, která je tak jednoduchá, že není Turingovsky kompletní. Chybí základní programovací konstrukce, jako je podmínka, smyčka. Je však takto volena záměrně, aby se na ní daly demonstrovat jednotlivé zotavovací metody.

Příklad 1

1. <PROGRAM> → begin <BODY> end
2. <BODY> → ε

Gramatika uvedená v příkladu 1. je definována jako bezkontextová gramatika. Má celkem dva *neterminály*, tři *terminály*, dvě pravidla. Její *počáteční neterminál* je <PROGRAM>. Pravidla jsou pro jednoduchost označeny čísly. Neterminál je psán velkým písmem, uzavřen v závorkách: <PROGRAM>, <BODY>. Terminál je dále nederivovatelný. Je psán malým písmem. Zde mezi neterminály patří begin, end a ε (prázdný řetězec).Symbol → značí *derivační krok*. Pomocí prvního pravidla je nahrazen neterminál <PROGRAM> jeho pravou stranou begin <BODY> end. V jeho pravé straně jsou dva terminály a jeden neterminál.

Pravidla uvedená v příkladu 1. patří do gramatiky používané v této práci. Celá gramatika je uvedena v kapitole 3.

Definice 9 (bezkontextová gramatika)
Bezkontextová gramatika (BKG) 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 $A \rightarrow x$, kde $A \in N$, $x \in (N \cup T)^*$
- $S \in N$ je *počáteční neterminál*

Definice 10 (derivační krok)
Nechť G = (N, T, P, S) je BKG. Nechť $u, v \in (N \cup T)^*$ a $p=A \rightarrow x \in P$. Potom, uAv přímo derivuje uxv za použití p v G, zapsáno $uAv \Rightarrow uxv$ [p] nebo zjednodušeně $uAv \Rightarrow uxv$. Jestliže je provedeno několik derivačních kroků, provádíme derivaci. Jsou používána pravidla, která jsou v gramatice uvedena. Pořadí derivačních kroků závisí na metodě. Derivace končí, jakmile nelze použít žádné další pravidlo a derivovaný řetězec neobsahuje žádné neterminály. Větná forma obsahující pouze terminály se nazývá *věta*.

Definice 11 (nejlevější derivace)
Nechť G = (N, T, P, S) je BKG, nechť $u \in T^*$, $v \in (N \cup T)^*$, $p = A \rightarrow x \in P$ je pravidlo. Pak uAv přímo derivuje uxv za pomocí nejlevější derivace užitím pravidla p v G, zapsáno jako: $uAv \Rightarrow_{lm} uxv$ [p].

3 Syntaktická analýza

Syntaktickou analýzu provádí syntaktický analyzátor. V tradičním schématu se nachází mezi lexikálním analyzátorem a sémantickým analyzátorem Jeho vstupem je řetězec tokenů. Jeho výstupem je derivační strom.
Jeho cílem je zjistit, zda řetězec tokenů patří do daného jazyka, tedy zda tento řetězec reprezentuje správně syntakticky napsaný program. Jestliže je k danému řetězci nalezen derivační strom, program je správný. V opačném případě je na vstupu chyba. Derivační strom se tvoří na základě gramatických pravidel.
Existují dva přístupy:

- zdola nahoru
- shora dolů

Tato práce se zabývá syntaktickou analýzou shora dolů.
Při metodě shora dolů se používá LL tabulka. K jejímu sestrojení je potřeba množin Empty, First, Follow a Predict. Tyto množiny budou vysvětleny níže.
V této kapitole je využito definic ze zdrojů [5] a [6].

3.1 LL gramatika

Aby mohla být syntaktická analýza prováděna, musí být její postup deterministický. K tomu slouží LL gramatika. LL v názvu značí "*left to right, left most*".
Obecně jsou bezkontextové gramatiky silnější než LL gramatiky. Ne všechny bezkontextové gramatiky lze převést na LL gramatiky. Převod se provádí pomocí faktorizace(vytýkání) a odstranění levé rekurze.
LL gramatiku můžeme rozdělit na:

- silnou LL gramatiku - s epsilon pravidly.
- slabou LL gramatiku - bez epsilon pravidel

Definice 1 (LL gramatika)
Nechť G = (N, T, P, S) je BKG bez ε-pravidel. G je LL gramatika, pokud pro každé $a \in T$ a $A \in N$ existuje maximálně jedno pravidlo $A \rightarrow X_1X_2 \ldots X_n \in P$ takové, že: $a \in \text{First}(X_1X_2 \ldots X_n)$

Definice 2 (LL gramatika s epsilon pravidly)
Nechť G = (N, T, P, S) je BKG. G je LL-gramatika, pokud pro každé $a \in T$ a každé $A \in N$ existuje maximálně jedno A-pravidlo tvaru $A \rightarrow X_1X_2 \ldots X_n \in P$ a platí: $a \in \text{Predict}(A \rightarrow X_1X_2 \ldots X_n)$

3.1.1 Množiny Empty, First, Follow a Predict

K sestrojení LL tabulky, na jejímž základě syntaktická analýza shora dolů probíhá, je potřeba výpočet množin Empty, First, Follow a Predict.V příkladu č. 2 jsou napsána všechna pravidla, která se v této práci používají. Na nich je demonstrována tvorba výše zmíněných množin, později i vytvoření samotné tabulky.

Příklad 2

1. <PROGRAM> ⟶ begin <BODY> end
2. <BODY> ⟶ ε

3. <BODY> ⟶ <STATEMENT> ; <BODY>
4. <STATEMENT> ⟶ read id
5. <STATEMENT> ⟶ write <TYPE>
6. <TYPE> ⟶ <TERM>
7. <TYPE> ⟶ string
8. <TERM> ⟶ int
9. <TERM> ⟶ id
10. <STATEMENT> ⟶ id = <EXPRESSION>
11. <EXPRESSION> ⟶ <TERM> <EXP>
12. <EXPRESSION> ⟶ (<TERM> <EXP>)
13. <EXP> ⟶ + <EXPRESSION>
14. <EXP> ⟶ − <EXPRESSION>
15. <EXP> ⟶ ε

V literatuře se často jako speciální ukončovací terminál uvádí symbol $, který se přímo do gramatiky nepíše, je implicitní. Zde je však ukončovacím terminálem terminál *end*, který je v gramatice uveden explicitně.

Definice 3 (množina Empty)
Nechť G = (N, T, P, S) je BKG.

- Empty(x) = {ε} if $x \Rightarrow^* \varepsilon$
- Empty(x) = ∅, kde $x \in (N \cup T)^*$

[1]

Empty(begin) = ∅	Empty(int) = ∅	Empty(PROGRAM) = ∅
Empty(end) = ∅	Empty(=) = ∅	Empty(BODY) = {ε}
Empty(;) = ∅	Empty(( ) = ∅	Empty(STATEMENT) = ∅
Empty(read) = ∅	Empty( )) = ∅	Empty(TYPE) = ∅
Empty(id) = ∅	Empty(+) = ∅	Empty(TERM) = ∅
Empty(write) = ∅	Empty(−) = ∅	Empty(EXPRESSION) = ∅
Empty(string) = ∅		Empty(EXP) = {ε}

Definice 4 (množina FIRST)
Nechť G = (N, T, P, S) je BKG. Pro každé $x \in (N \cup T)^*$ je definováno First(x) jako: First(x)={a: $a \in T$, $x \Rightarrow^* ay$; $y \in (N \cup T)^*$ }

First(begin) = {begin}	First(int) = {int}	First(PROGRAM) = {begin}
First(end) = {end}	First(=) = {=}	First(BODY) = {read, write, id}
First(;) = { ; }	First(( ) = { (}	First(STATEMENT) = {read, write, id}
First(read) = {read}	First( )) = {) }	First(TYPE) = {int, id, string}
First(id) = {id}	First(+) = {+}	First(TERM) = {int, id}
First(write) = {write}	First(−) = {−}	First(EXPRESSION) = { (, int, id}
First(string) = {string}		First(EXP) = {+, −}

[1] pozn: Empty(*(*) a Empty(*(*) je totožné, mezery jsou vloženy pouze pro lepší čitelnost. Stejně tak Empty(*;*) a Empty(*)*). Mezery jsou uměle vloženy i při výpisu množin First, Follow a Predict.

Definice 5 (množina Follow)
Nechť G = (N, T, P, S) je BKG. Pro všechna $A \in$ N definujeme množinu Follow(A): Follow(A)={a: $a \in$ T, S⇒* $xAay$, $x, y \in$ (N∪T)*} ∪ {$: S⇒* xA, $x \in$ (N∪T)*} [2]

Follow(PROGRAM) = {end}
Follow(BODY) = {end}
Follow(STATEMENT) = { ; }
Follow(TYPE) = { ; }
Follow(TERM) = { +, −, ;,)}
Follow(EXPRESSION) = {), ; }
Follow(EXP) = {), ; }

Definice 6 (množina Predict)
Nechť G = (N, T, P, S) je BKG. Pro každé $A \rightarrow x \in$ P definujeme množinu Predict($A \rightarrow x$) jako:

- pokud Empty(x) = {ε} potom: Predict($A \rightarrow x$) = First(x) ∪ Follow(A)
- jinak pokud Empty(x) = ∅ potom: Predict($A \rightarrow x$) = First(x)

<PROGRAM> ⟶ begin <BODY> end	{begin}
<BODY> ⟶ ε	{end}
<BODY> ⟶ <STATEMENT> ; <BODY>	{read, write, id}
<STATEMENT> ⟶ read id	{read}
<STATEMENT> ⟶ write <TYPE>	{write}
<TYPE> ⟶ <TERM>	{int, id}
<TYPE> ⟶ string	{string}
<TERM> ⟶ int	{int}
<TERM> ⟶ id	{id}
<STATEMENT> ⟶ id = <EXPRESSION>	{id}
<EXPRESSION> ⟶ <TERM> <EXP>	{int, id}
<EXPRESSION> ⟶ (<TERM> <EXP>)	{ (}
<EXP> ⟶ + <EXPRESSION>	{+}
<EXP> ⟶ − <EXPRESSION>	{−}
<EXP> ⟶ ε	{), ; }

3.1.2 Tvorba LL-tabulky

LL-tabulka obsahuje neterminály v levém sloupci a terminály v prvním řádku. V buňkách tabulky jsou čísla odpovídající jednotlivým pravidlům nebo samotná pravidla. V každém poli musí být právě jedno pravidlo nebo je prázdné. Při syntaktické analýze odpovídá prázdné pole chybě.

Λ	a	…
A	λ(A, a)	

[2] Pravidla jsou pro názornost napsána tak, že znak konce řetězce nepředstavuje znak *$*, ale terminál *end*, který je explicitně v pravidle číslo 1. uveden.

…		

$\lambda(A, a) = A \rightarrow X_1X_2 \ldots X_n \in$ P pokud $a \in$ Predict($A \rightarrow X_1X_2 \ldots X_n$); jinak $\lambda(A, a)$ je prázdné.

Příklad 3
Výsledná LL tabulka vypadá takto:

	begin	end	;	read	id	write	string	int	=	+	−	)	(
PROGRAM	1												
BODY		2		3	3	3							
STATEMENT				4	10	5							
TYPE					6		7	6					
TERM					9			8					
EXPRESSION					11			11					12
EXP			15							13	14	15	

Bílá místa znázorňují chybu. Pro větší přehlednost jsou v tabulce uvedena pouze čísla pravidel.

4 Zotavení z chyb

Tato kapitola se věnuje samotnému zotavování z chyb při syntaktické analýze shora dolů. U metod je krátký popis, shrnutí jejich výhod a nevýhod, případně porovnání s jinými metodami. Pro názornost je u některých metod uveden i příklad. Příklady vychází z gramatických pravidel definovaných v kapitole 3. Některé metody využívají množin First a Follow souvisejících s gramatikou v kapitole 3 a taktéž LL tabulku z nich vytvořenou.

Některé metody jsou vhodné pouze pro syntaktickou analýzu zdola nahoru, proto jsou zde zmíněny pouze krátce. V této kapitole je čerpáno ze zdrojů [1], [2], [3] a [4].Rozdělení do jednotlivých kategorií je podle knih [1], [3], implementovaná metoda, kontext tvořený pokračovací metodou, čerpá ze zdrojů [3] a [4].

4.1 Úvod do zotavení z chyb

Každý korektně napsaný syntaktický analyzátor umí detekovat chybu - ta nastává v případě, kdy nelze použít žádné pravidlo. Z toho plyne, že každý korektně napsaný syntaktický analyzátor má funkci detekce chyb (*error detection*). Syntaktický analyzátor má velký problém zjistit, kde chyba nastala. Lépe řečeno, je to téměř nemožné. Některé syntaktické analyzátory umí detekovat chybu u prvního symbolu na vstupu, který způsobí, že daný prefix nemůže být začátkem řetězce daného jazyka. Tuto vlastnost však mají pouze gramatiky bez epsilon pravidel. Přesto si však autor syntaktického analyzátoru nemůže být jistý, že chyba vznikla opravdu v detekovaném místě. Chyba z pohledu programátora mohla vzniknout daleko dříve.

Opravením však není možné chápat modifikaci vstupu tak, jak ho zamýšlel programátor. Tudíž výsledkem je sice úprava vstupu tak, aby odpovídal řetězci přijímaného jazyka, výsledek se však nemusí shodovat s úmysly programátora.

Na zotavování z chyb je možno pohlížet ze dvou úhlů. Syntaktický analyzátor nemůže chybu nikdy opravit objektivně správně, proto by neměl plýtvat strojovým časem na její co nejlepší opravení. Zároveň je program se syntaktickou chybou vždy špatný a již nepřeložitelný, proto má být zotavení co nejjednodušší, se snahou zjistit co nejvíce syntaktických chyb. Druhý pohled říká, že syntaktický analyzátor by měl chybu opravit co nejlépe, aby autorovi programu co nejvíce pomohl a co nejvíce problémů za něj vyřešil.

Metody se snaží o skloubení těchto protichůdných požadavků.

4.1.1 Cíl zotavení z chyb

Syntaktický analyzátor má za úkol zpracovat vstup - řetězec tokenů od lexikálního analyzátoru. Jeho výstupem je derivační strom. Jestliže je k danému řetězci tokenů nalezen derivační strom, program je správný, jinak ne. Jestliže je v programu syntaktická chyba, překladač bez zotavení skončí. To však není příliš vhodné, neodhalí se totiž další chyby v programu. Aby bylo možné pokračovat v syntaktické analýze, přestože byla nalezena chyba, je nutné implementovat zotavení z chyb.

Cíle zotavení jsou:

1. nalézt chybu co nejrychleji v místě, kde skutečně nastala, bez načítání dalších symbolů
2. zotavit se co nejrychleji
3. díky zotavení nepřeskočit další chyby
4. nehlásit neexistující chyby

Cíle nejsou stoprocentně splnitelné. Je však snaha se jim co nejvíce přiblížit.

4.1.2 Typy chyb

Existují dva typy chyb. Chyby, které zabrání překladači v nějaké fázi překladu pokračovat. Ty se dají snadno odhalit a programátorovi může překladač říci, v jakém místě chybu odhalil a dokonce mu může navrhnout opravu. Do této kategorie patří:

- **lexikální chyby** překlep - například záměna velkého písmene za malé
- **syntaktické chyby** neočekávaný token či nenalezení pravidla - například přebývající aritmetický operátor, chybějící středník
- **sémantické chyby** – špatné porovnání typů - například do proměnné typu integer přiřazení desetinného čísla

Druhý typ chyb je zákeřnější, protože se nejedná o chyby, které by mohl překladač jednoduše opravit. Program je funkční, ale nevyjadřuje to, co autor zamýšlel. Na některé z častých chyb může překladač programátora upozornit - například záměna "=" a "==" v if podmínce. Tyto chyby se nazývají **chyby logické**.

4.2 Druhy zotavení

Zotavovaní z chyb při syntaktické analýze shora dolů je více druhů. Dělí se podle úrovní, na jakých k chybám přistupuji. Dělení není přesné, úrovně se překrývají a různé zdroje se liší.Pokud používají globální kontext, tedy celý vstup, jedná se o metody globálního zotavení z chyb. V případě použití části kontextu v okolí se jedná o zotavení na úrovní fází. Pokud používají pouze aktuální stav překladače a aktuální vstupní symbol, jedná se o metody lokálního zotavení z chyb. Jestliže nepoužijí žádný kontext, jedná se o sufixové metody. Poslední skupinou jsou ad hoc metody, které chyby neopravují za běhu překladu.

4.3 Metody globálního zotavení z chyb

Metody globálního zotavení berou v potaz celý vstup.
Nejznámější metodou je metoda co nejmenšího počtu oprav chyb.

4.3.1 Metoda co nejmenšího počtu oprav chyb

Tato metoda se snaží o co nejméně oprav či změn ve vstupu. Vynechání, přidání či přepsání symbolu je většinou považováno za jednu opravu (*one edit operation*). Minimální či maximální počet oprav může být jednoduše vypočítán, je odvozen z délky řetězce. V principu je nutné vypočítat všechny varianty oprav, které budou v limitu, a vybrat opravení, které bude nejlevnější, tedy bude zahrnovat nejméně oprav či změn ve vstupu.

Metody globálního zotavení z chyb jsou obecně příliš náročné na čas i na paměť, takže se používají jen v obecných syntaktický analyzátorech (*general parsers*), které jsou sami o sobě náročné a používají se většinou pouze na teoretické úrovni.

4.4 Metody zotavení na úrovni fázi

Metody zotavení na úrovni fází se oproti globálnímu zotavení používají i v praxi. Metody zotavení na úrovni fází se využívají jak při postupu zdola nahoru, tak shora dolů.

Při postupu zdola nahoru je princip těchto metod následující. Při zotavování se analyzátor podívá na okolí chyby, což je zpravidla část zásobníku. Analyzátor nahradí část vstupu tak, aby mohl překlad dál pokračovat. Typicky se jedná o nahrazení čárky středníkem, vložení chybějícího středníku či vymazání přebývajícího středníku.

Výhodou této metody je, že dokáže opravit jakýkoliv chybný vstup.

Při překladu shora dolů je tato metoda implementována následovně. Nalezení chyby můžeme nazvat situaci, kdy analyzátor nemá žádné pravidlo, které by použil, respektive na místě v tabulce, kde pravidlo očekává, žádné pravidlo není (pro názornost - místo je prázdné, bílé).Princip této metody spočívá v tom, že prázdná místa v tabulce jsou nahrazeny odkazem na opravné rutiny. Ty mají za účel změnit, odebrat či přidat symbol na začátku vstupu tak, aby mohl analyzátor pokračovat. Můžou také odebrat ze zásobníku. Zároveň také vyvolají chybovou hlášku, která programátora na upozorní, že vznikla chyba a že program bude nepřeložitelný.

Nevýhodou těchto metod je, že si autor syntaktického analyzátoru musí dát pozor, aby se program nedostal do nekonečné smyčky. Obranou je kontrola, zda rutina odstranila symbol ze vstupu (či zkrátila zásobník, jestliže vstup již skončil). Hlavní nevýhodou je situace, kdy je výskyt skutečné chyby před momentem detekce chyby, tudíž nikdy nelze napravit chybu samotnou.

4.5 Ad Hoc metody

Ad Hoc metody patří mezi nejjednodušší metody. Metody patřící do této skupiny nejsou ovlivněny žádným kontextem. Gramatiky jsou upraveny předem, takže syntaktický analyzátor chybu v pravém smyslu nepozná. Pouze použitím již předem napsaných chybných konstrukcí se vyvolá chybová hláška. Ad Hoc metody jsou vždy pouze tak dobré, jak dobrý je programátor překladače, respektive jak dobře odhadne programátor výskyty chyb. Patří sem například metoda Tvoření chyb (*Error Production*).

4.5.1 Tvoření chyb

Metoda tvoření chyb patří mezi časté metody zotavení. Nepoužívá se často samotná, ale v kombinaci s jinými metodami. Tato metoda je jednoduchá a rychlá.
Syntaktický analyzátor je řízen předem napsanými pravidly. Princip metody tvoření chyb spočívá v tom, že správná pravidla rozšířím o tzv. chybová pravidla. Chybové pravidlo vyvolá chybovou hlášku, syntaktická analýza nebude provedena korektně, ale umožní to syntaktickému analyzátoru pokračovat, neboť syntaktický analyzátor bude vstup vnímat jako bezchybný.

Výhodou této metody je, že uživateli obdrží chybovou hláškou s poměrně přesnou informací, o jakou chybu se jedná, respektive jak přesně byla opravena (například chybí operátor, vložen operátor “plus”).

Nevýhodou této metody je, že odhalí pouze předem napsaná pravidla. Pokud by měla metoda obsahovat všechny možnosti chyb, gramatických pravidel by bylo potenciálně nekonečné množství.

Příklad 4
Nechť existují následující pravidla:

1. <PROGRAM> ⟶ begin <BODY> end
2. <BODY> ⟶ ε
3. <BODY> ⟶ <STATEMENT> ; <BODY>
4. <STATEMENT> ⟶ read id
5. <STATEMENT> ⟶ id = <EXPRESSION>
6. <EXPRESSION> ⟶ <TERM> <EXP>

7. <EXP> ⟶ + <EXPRESSION>

Předpokládejme, že častou chybou programátorů je nahrazení znaku “;” znakem “,” , vynechání operátoru a podobně.

```
1. begin
2.   read a,
3.   a = a 5;
4. end
```

Ke gramatice přidáme tato pravidla:

<BODY> ⟶ <STATEMENT>, <BODY> //nahrazení znaku “;” znakem “,”
<EXP> ⟶ <EXPRESSION> // vynechání operátoru.

Po přidání těchto pravidel syntaktický analyzátor nepozná, že je v kódu chyba. Nicméně tato pravidla vyvolají příslušnou sémantickou akci, vyvolají chybovou hlášku.

4.5.2 Doplnění LL tabulky

Další jednoduchou metodou je zaplnit bílá místa v LL tabulce, jež představují chyby. Na bílá místa se jednoduše vepíší existující pravidla. Jaká pravidla vepsat na která místa záleží na programátorovi. Použitím vepsaného pravidla se vyvolá chybová hláška, program nelze přeložit. Oproti metodě tvoření chyb nemůže nastat stav, že by se dostal syntaktický analyzátor do chybového stavu, tedy nevěděl, jak pokračovat.Tato metoda je velice jednoduchá, ale není příliš efektivní.

Příklad 5
V příkladu je vyříznuta pouze část tabulky. Čísly jsou napsána doplněná pravidla.

	begin	end	;	read	id
PROGRAM	begin <BODY> end	1	1	1	1
BODY	3	ε	2	<STATEMENT> ; <BODY>	<STATEMENT> ; <BODY>

4.6 Sufix analýza

Většina klasických zotavovacích metod funguje na následujícím principu.V případě chyby se změní stav syntaktického analyzátoru pomocí heuristiky, která nám zvolí jednu z mnoha variant změny. Heuristiky jsou různě složité a lépe či hůře napraví chybu. Nikdy si však autor syntaktického analyzátoru nemůže být jistý, že heuristikou vybraná varianta je ta správná. Výběrem špatné varianty je možné zapříčinit vznik dalších chyb.

Jiný přístup ke zotavování z chyb popsal Richter. Autor argumentuje, že by se při zotavování z chyb nemělo pokoušet o opravu chyby, neb není možné zjisti, zda byla opravena správně. Stejně tak není moudré měnit stav analyzátoru. Jediné, co lze předpokládat je, že celý zbytek vstupu, tedy vstup za chybou, včetně chyby samotné, je sufix řetězce nějakého jazyka. Množina řetězců, které tvoří sufix řetězce nějakého jazyka, je jazyk sám. Jazyk tvořený sufixy je sufixový jazyk, je tedy možné pro něj

vytvořit i tzv. sufixovou gramatiku.Princip metody je následující. Jakmile je nalezena chyba, chybný symbol je přeskočen a zbývající vstup je přiveden na vstup sufix analyzátoru, postaveném na sufix gramatice. Jestliže je opět nalezena chyba, chyba je přeskočena, sufix analyzátor je restartován a je připraven přijmout další sufix.Tato metoda má některé výhody:

- tato metoda hlásí každou chybu pouze 1x
- jakmile je nahlášena chyba, analyzátor je restartován do původního stavu; proto zde nejsou žádné falešné chyby
- nic, kromě chyby, není přeskočeno

Hlavní nevýhodou je, že tato metoda funguje pouze nad sufixovými gramatikami. Aby byla analýza efektivní, je potřeba sestrojit efektivní sufix analyzátor. Sufix gramatiky ale nejsou vhodné pro deterministické metody, jako je LL či LR. Ve skutečnosti, zkonstruováním sufix gramatiky je téměř jistě dosaženo nejednoznačné gramatiky. Další, méně podstatnou nevýhodou je fakt, že ne všechny chyby budou nahlášeny. To však není důležité v interaktivních prostředích, kde je důležitější, aby žádná chyba nebyla hlášena vícekrát.

V případě chyby analyzátor nedokáže vytvořit smysluplný derivační strom, což může i nemusí být považováno za nevýhodu.

4.7 Metody lokálního zotavení z chyb

Jakmile analyzátor narazí na chybu, vypočítá se ze stavu tzv. množina Kontext (acceptable-set). Symboly na vstupu jsou přeskakovány do té doby, než se na vstupu objeví symbol z množiny Kontext a stav analyzátoru se přizpůsobí tak, aby mohl být tento symbol přijat.

Metody patřící do této skupiny se liší ve způsobu výpočtu množiny Kontextu a způsobem, jak se stav analyzátoru přizpůsobí.

4.7.1 Panický mód

Dle literatury se jedná o nejjednodušší zotavovací metodu, která se dá v praxi použít.

Množina Kontext je vytvořena autorem analyzátoru a je neměnná po celou dobu analýzy. Symboly z množiny obvykle indikují konec syntaktické konstrukce (například v jazyce C by to mohl být středník).

Při nalezení chyby se přeskakují symboly, dokud se na vstupu neobjeví symbol z množiny Kontext. Analyzátor se poté přenese do stavu, kdy je symbol akceptovatelný - u LL analyzátoru to spočívá ve vymazání několika symbolů ze vstupu.

Jeho velkou výhodou je jednoduchá implementace.

Nevýhodou je, že mnoho chyb může zůstat neobjeveno, neboť se může během přenesení do akceptovatelného stavu přeskočit velká část vstupu.

4.7.2 Follow set zotavení z chyb

Tato metoda se používá se v LL analyzátorech a je jedna z nejznámějších učebnicových metod pro zotavení z chyb při analýze shora dolů.

Princip metody je následující. Chyba vzniká, když není nalezeno pravidlo pro neterminál A. Pokud pravidlo neexistuje, syntaktický analyzátor přeskakuje vstupní symboly do té doby, kdy symbol na vstupu nenáleží do množiny Follow(A).

Existují dvě varianty dalšího postupu.

Odstraní se nezpracované pravé části nonterminálu A a syntaktický analyzátor dále pokračuje. Není ale možné zaručit, že vstupní symbol může následovat za A v daném kontextu.

Druhou variantou je použití pouze té části Follow(A), která může následovat za A v daném kontextu. Tato varianta je ale složitá.

V praxi se při rekurzivním sestupu nepoužívá pouze Follow(A). Obecně je během analýzy více jak jeden aktivní nonterminál. Proto se bude vstupní symbol, který se již nepřeskočí, vybírat z množiny vzniklé sjednocením buď množin Follow všech nonterminálů a nebo sjednocením všech nonterminálů těch částí, jenž mohou následovat nonterminál v daném kontextu.

Varianty metody

Do množiny kontextu se může mimo množiny Follow(A) přidat i množina First(A).
Pokud může nonterminál generovat prázdný řetězec, poté je možné derivaci na prázdný řetězec použít jako implicitní. V tom případě se sice může oddálit odhalení některých chyb, není možné však žádnou chybu přeskočit. Tento způsob redukuje počet nonterminálů, které je nutno brát v potaz při zotavování z chyb.

Tuto metodu lze i podstatně zjednodušit. Pokud terminál na vrcholu zásobníku není možné spárovat se vstupem, odstraní se tento terminál ze zásobníku, vypíše se chybová zpráva a pokračuje se v analýze.

Na příkladu je ukázán nerekurzivní sestup této zotavovací metody.

Příklad 6
Pro každý nonterminál A se vypočte jeho množina Kontext, obsahující množinu Follow(A). LL tabulka obsahuje slovo "sync" indikující synchronizační tokeny z množiny Kontext. Jestliže analyzátor hledá pravidlo M[A, a] a nalézá v tabulce bílé místo, vstupní symbol a je smazán. Jestliže je v tabulce "sync", analyzátor vyjme nonterminál z vrcholu zásobníku a pokračuje v analýze. Jestliže symbol na vrcholu zásobníku neodpovídá vstupnímu symbolu, je symbol z vrcholu zásobníku vyňat.

	begin	end	;	read	id	write	string	int	=	+	−	)	(
PROGRAM	1	sync											
BODY		2		3	3	3							
STATEMENT			sync	4	10	5							
TYPE			sync		6		7	6					
TERM			sync		9			8		sync	sync	sync	
EXPRESSION			sync		11			11				sync	12
EXP			15							13	14	15	

Je uvažován chybný vstup:

```
1.begin
2.  write "Hello world"
3.  write ;
4.end
```

Je vidět, že na řádku č. 2 chybí středník. Na řádku č. 3 chybí identifikátor. Syntaktický analyzátor dostane od lexikálního analyzátoru tokeny *begin*, *write*, *string*, *write*, *;*, *end*, místo korektního vstupu *begin*, *write*, *string*, *;, write*, *string/int/id*, *;*, *end.*

V následující tabulce je ukázka zotavení z chyby.

zásobník	vstup	pravidlo
<PROGRAM>	begin write string write ; end	1. <PROGRAM> ⟶ begin <BODY> end
end <BODY> **begin**	**begin** write string write ; end	
end <BODY>	write string write ; end	3. <BODY> ⟶ <STATEMENT> ; <BODY>
end <BODY> ; <STATEMENT>	write string write ; end	5. <STATEMENT> ⟶ write <TYPE>
end <BODY> ; <TYPE> **write**	**write** string write ; end	
end <BODY> ; <TYPE>	string write ; end	7. <TYPE> ⟶ string
end <BODY> ; **string**	**string** write ; end	
end <BODY> ;	write ; end	**CHYBA** - terminál je z vrcholu zásobníku vyňat
end <BODY>	write ; end	3. <BODY> ⟶ <STATEMENT> ; <BODY>
end <BODY> ; <STATEMENT>	write ; end	5. <STATEMENT> ⟶ write <TYPE>
end <BODY> ; <TYPE> **write**	write ; end	
end <BODY> ; <TYPE>	; end	**sync** - nonterminál je z vrcholu zásobníku vyňat
end <BODY> **;**	**;** end	
end <BODY>	end	2. <BODY> ⟶ ε
end	end	úspěšný konec

4.7.3 Kontext tvořený pokračovací metodou

Tato metoda zotavení z chyb je implementována.

Zajímavostí této metody je, že zotavení z chyby syntaktického analyzátoru, tedy změna zásobníku či vstupu, nenastává ihned při nalezení chyby. Jestliže je nalezena chyba, syntaktický analyzátor pokračuje v analýze, ovšem s jinou, pokračovací gramatikou.

Syntaktický analyzátor detekuje chybu ve vstupu poté, co zpracoval prefix *u*. Prefix *u* je část řetězce jazyka. Z toho vyplývá, že musí existovat i pokračování (continuation) tohoto řetězce.

Pokračování je konečný řetězec *w* takový, že *uw* tvoří řetězec jazyka. Jestliže lze spočítat pokračování, je možné při nalezení chyby postupovat dle následujícího algoritmu.

1. Je determinováno pokračování *w* řetězce *u*.
2. Pro všechny prefixy *w'* z *w* se vypočítá množina terminálních symbolů, které budou akceptovatelné analyzátorem poté, co zpracuje *w'*, a vytvoří se sjednocení těchto množin. Tato množina je hledaný Kontext. Pokud symbol *a* je obsažen v této množině Kontext, pak *uw'a* je prefix řetězce v daném jazyce.
3. Dokud není nalezen symbol z množiny Kontext, přeskakují se všechny symboly na vstupu. Výsledkem může být i přeskočení všeho, až do koncového znaku (v případě gramatiky uvedené v této práci je koncový znakem symbol *end*).
4. Vloží se nejkratší prefix *w* tak, aby symbol na vstupu byl akceptovatelný. Pokud bylo vymazáno vše až do koncového symbolu, vloží se celý řetězec *w*.
5. Syntaktický analyzátor vypíše chybu se sdělením, jaké znaky byly přeskočeny a jaké byly vloženy.
6. Syntaktický analyzátor je následně restartován do stavu, kde byla chyba detekována a pokračuje v analýze, počínaje vloženým symbolem, jako by žádná chyba nenastala.

Pokračovací gramatika

Kontext tvořený pokračovací metodou má dva zásadní problémy. Jak vypočítat množinu Kontext bez toho, aby se prošlo všemi možnostmi? A jak vytvořit pokračování?

Cílem je vytvořit řetězec co nejrychleji, s co nejkratším krokem. Toto je možné, jestliže pro každý nonterminál je znám jeho nejrychlejší "konec", tedy jaké jeho pravé strany vedou k řetězci složeného pouze terminály s co nejmenším počtem kroků.

Je zajímavé, že je možné si vypočítat nejrychlejší "konce" předem. Jinými slovy, je možné předem zjistit pro každý symbol minimum kroků, které vedou k derivaci na terminál. Toto číslo můžeme nazvat *číslo kroků*. Terminální symboly mají číslo kroku 0, nonterminály mají zatím neznámé číslo kroků, které je implicitně nastaveno na nekonečno.

Postupně se projde každá pravá strana pravidla. Jestliže je počet kroků znám pro každého člena pravé strany, celá pravá strana má hodnotu počtů kroků rovnu součtu všech počtů kroků jednotlivých členů. Pomocný počet kroků levé strany pravidla je pak počet kroků celé pravé strany + 1. Pokud je toto číslo menší, než byl původní počet kroků nonterminálu, je nastaveno toto nové číslo jako aktuální počet kroků. Tento proces se opakuje, dokud se čísla počtu kroku nemění. Pokud je gramatika napsána správně, všechna čísla kroků jsou konečná.

Nyní je nutné vybrat takové pravidlo pro každý neterminál, jehož pravá strana má nejmenší počet kroků. Tato pravidla tvoří pokračovací gramatiku, ačkoli pravidla sama o sobě pravděpodobně netvoří korektní gramatiku.

Zde je ukázán výpočet pokračovací gramatiku pro gramatiku používanou v této práci. Je ukázáno pouze pár kroků a následně vypsána celá pokračovací gramatika.
V hranatých závorkách je za každým neterminálem uveden jeho počet kroků.[3]

<PROGRAM> **[∞]** ⟶ begin **[0]** <BODY> **[∞]** end **[0]**
<BODY> **[∞]** ⟶ ε **[0]**
|<STATEMENT> **[∞]** ; **[0]** <BODY> **[∞]**

[3] Je zde použita varianta zkráceného zápisu, kde <TERM>⟶ int a <TERM>⟶ id je zjednodušeno na <TERM> ⟶ int | id.

<STATEMENT> **[∞]** ⟶ read **[0]** id **[0]**
|write **[0]** <TYPE> **[∞]**
| id **[0]** = **[0]** <EXPRESSION> **[∞]**
<TYPE> **[∞]** ⟶ <TERM> **[∞]**
| string **[0]**
<TERM> **[∞]** ⟶ int **[0]**
| id **[0]**
<EXPRESSION> [∞] ⟶ <TERM> **[∞]** <EXP> **[∞]**
| (**[0]** <TERM> **[∞]** <EXP> **[∞]**) **[0]**
<EXP> **[∞]** ⟶ + **[0]** <EXPRESSION> **[∞]**
| − **[0]** <EXPRESSION> **[∞]**
| ε **[0]**

Je vidět, že pro neterminál *TERM* již lze určit počet kroků. Je to jeho pravá strana + 1, tedy 1. Dále také lze určit počet kroků pro neterminál *BODY*. Ten má počet kroků 1 (terminál eps **[0]** + 1). Stejně tak neterminál *EXP*, který má též počet kroků roven 1 (terminál eps **[0]** + 1). Taktéž neterminál *STATEMENT* a *TYPE*.

<PROGRAM> **[∞]** ⟶ begin **[0]** <BODY> **[∞]** end **[0]**
<BODY> **[1]** ⟶ ε **[0]**
|<STATEMENT> **[∞]** ; **[0]** <BODY> **[∞]**
<STATEMENT> **[1]** ⟶ read **[0]** id **[0]**
|write **[0]** <TYPE> **[∞]**
| id **[0]** = **[0]** <EXPRESSION> **[∞]**
<TYPE> **[1]** ⟶ <TERM> **[∞]**
| string **[0]**
<TERM> **[1]** ⟶ int **[0]**
| id **[0]**
<EXPRESSION> [∞] ⟶ <TERM> **[∞]** <EXP> **[∞]**
| (**[0]** <TERM> **[∞]** <EXP> **[∞]**) **[0]**
<EXP> **[∞]** ⟶ + **[0]** <EXPRESSION> **[∞]**
| − **[0]** <EXPRESSION> **[∞]**
| ε **[0]**

Dalšími kroky jsou vypočteny všechny neterminály. Pro každý neterminál se vybere takové pravidlo, jehož pravá strana má nejmenší počet kroků. Jestliže je počet kroků roven, musí se zahrnout pouze jedno pravidlo, vybrané autorem syntaktického analyzátoru. Zde jsou tedy vyškrtnuta pravidla <TERM> [1] ⟶ id [0] a <EXPRESSION> [3] ⟶ ([0] <TERM> [1] <EXP> [1]) [0]. Pouze jedno pravidlo se vybere z toho důvodu, že by syntaktický analyzátor nedokázal deterministicky rozhodnout, jak má nezpracovanou část pravidla derivovat (nemá žádnou LL tabulku, podle které by se mohl rozhodovat). Pokračovací gramatika vypadá následovně:

1. <PROGRAM> ⟶ begin <BODY> end
2. <BODY> ⟶ ε
3. <STATEMENT> ⟶ read id
4. <TYPE> ⟶ string
5. <TERM> ⟶ int

6. <EXPRESSION> ⟶ <TERM> <EXP>
7. <EXP> ⟶ ε

Pokračování u LL syntaktického analyzátoru

Postup při chybě u LL analyzátoru není složitý. Je nutné mít vyjádřenou pokračovací gramatiku.

1. **Zpracování pravidla**: Analyzátor se podívá na dosud nezpracovanou část pravidla, tedy první neterminál na vrcholu zásobníku. Tuto nezpracovanou část derivuje na koncový řetězec, používají se pouze pravidla z pokračovací gramatiky. Každý terminál, který se vygeneruje, se vloží do množiny Kontext. Navíc, pokaždé, když je neterminál přepsán pravou stranou pravidla Pokračovací gramatiky, přidá se do množiny Kontext i terminální symbol z množiny First daného neterminálu.
2. **Přeskočí se nepřijaté tokeny**: Nula či více symbolů na vstupu je přeskočeno. Přeskakuje se tak dlouho, dokud není na vstupu symbol z množiny Kontext. Koncový terminál, v ukázkové gramatice je to terminál end, je vždy přijatý. Proto se můžou přeskočit všechny symboly na vstupu až po zarážku tvořenou speciální ukončovací terminál *end*.
3. **Resynchronizace analyzátoru**: Analyzátor se pokusí pokračovat. Jestliže nelze pokračovat, platí:
   a. **na vrcholu zásobníku je neterminál**: Terminál se derivuje dle pokračovací gramatiky. Algoritmus se opakuje od kroku 1.
   b. **na vrcholu zásobníku je terminál**: Vloží očekávaný token. Při vložení se tokenu nastaví implicitní hodnota daná programátorem analyzátoru. Vložením dojde ke shodě s terminálem na zásobníku a ten se tudíž vyjme.

algoritmus pro resynchronizaci syntaktického analyzátoru [3]:

```
resynchronised = False;
while (not resynchronised)
     predicted = pop(Stack);
     if predicted is a terminal
          if predicted == input_token[index]
               index++;   //match!
               resynchronised = True;
          else  //predicted != input token
               insert a token;
               //same type with a representation (e.g.. 0, "")
               printf("token of %s type inserted \n", type);
     else  // predicted is a nonterminal
          use rule from continuation grammar;
          for each symbol S in right side of the rule reversed
               push(S, Stack);
```

Je nutné povšimnout si, že v algoritmu v bodě 2. se tokeny přeskakují. Je možné, že se přeskočí identifikátor, který již má hodnotu, má alokovanou paměť, je zapsán v tabulce symbolů. V kroku 3. se může stát, že je na vrcholu terminál, vloží se očekávaný token. Je možné, že vkládaný token je identifikátor. Vložený identifikátor však bude mít jinou hodnotu, než identifikátor před okamžikem přeskočený. Vložený identifikátor bude mít implicitní hodnotu danou programátorem – například err_id. Algoritmus, který by předcházel tomuto jevu, by však byl příliš kombinovaný a zdlouhavý. Jakmile je nalezena chyba, program na vstupu je chybný a nelze přeložit. Syntaktický analyzátor má za úkol se co nejrychleji zotavit a objevit ještě co nejvíce chyb. Pokud by programátor nechtěl přijít o hodnotu původně napsaného identifikátoru, měl by vstupní program napsat bez chyb.

Modifikace metody a její implementace

V rámci bakalářské práce je tato metoda implementována. V algoritmu pro resynchronizaci syntaktického analyzátoru je udělána malá modifikace. Symbol na vrcholu zásobníku se nevyjme, pouze se zkontroluje, zda je validní. Pokud validní není, k jeho vyjmutí dojít musí v rámci bodu 1 nebo bodu 3b, resynchronizace pokračuje. Pokud však validní je, vyjme se až v rámci analýzy zotaveného syntaktického analyzátoru.

Výhodou tohoto přístupu je, že analyzátor co nejrychleji pokračuje v normálním módu. Vyjmutí terminálu ze zásobníku je provedeno na základě shody terminálu na zásobníku se symbolem na vstupu, případně nonterminál se převede na pravou stranu pomocí pravidel. To vše při běžném módu. Další velikou výhodou je fakt, že po zotavení nemůže analyzátor opět upadnout do chyby. Aplikací nemodifikovaného algoritmu může nastat situace, kdy analyzátor zahlásí validní zotavení z chyb. Po vyjmutí terminálu ze zásobníku či přepsání neterminálu na zásobníku, jenž se provede ještě v rámci zotavení, se může na vstupu opět objevit chyba. Jakmile se tedy syntaktický analyzátor zotaví, do chyby opět upadá. Pokud je algoritmus modifikovaný, tato situace nenastane. V modifikovaném algoritmu se syntaktický analyzátor podívá, zda je možné pokračovat v analýze. Pouze pokud to možné je, prohlásí se zotavení za úspěšné. Tato modifikace sice způsobí nárůst kódu, avšak vroste zapouzdřenost a ucelenost zotavení.

Příklad 6

```
1.begin
2.  read "Hello world";
3.end
```

Tento jednoduchý program by se zapsal jako následující posloupnost tokenů:
begin, *read*, *string*, *;*, *end*.

Na řádku č. 2 je chyba. Syntaktický analyzátor očekává na vstupu tokeny v pořadí *read id* a nebo *write string*. Kombinace *read string* je nepřípustná.V příkladu se pracuje s gramatikou užitou v této bakalářské práci a s LL tabulkou, v této bakalářské práci z dané gramatiky vytvořené.

zásobník	vstup	pravidlo
<PROGRAM>	begin read string;end	1. <PROGRAM> ⟶ begin <BODY> end
end <BODY> **begin**	**begin** read string;end	
end <BODY>	read string;end	3. <BODY> ⟶ <STATEMENT> ; <BODY>
end <BODY> ; <STATEMENT>	read string;end	4. <STATEMENT> ⟶ read id
end <BODY> ; id **read**	**read** string;end	
end <BODY> ; id	string;end	CHYBA

V tomto okamžiku se syntaktický analyzátor marně snažil porovnat terminál *id* a *string*. Musí přijít na řadu zotavování z chyb pomocí pokračovací gramatiky.

- Derivuje se první neterminál na vrcholu zásobníku, tedy neterminál *BODY*. Ten se dle pravidla <BODY> **[1]** ⟶ ε **[0]** přepíše na epsilon. Do množiny Kontext se vloží First(*BODY*), tedy terminály *read*, *write* a *id*.
- Přeskakují se všechny symboly na vstupu, dokud se nenalezne symbol z množiny Kontext nebo koncový terminál. Je tedy přeskočen symbol *string* a "*;*". Zásobník a vstup vypadá následovně:

zásobník	vstup
end ε ; id	end

- Nastává resynchronizace zásobníku. Analyzátor není resynchronizovaný. Na vrcholu zásobníku není neterminál, ale terminál. Terminál neodpovídá terminálu na vstupu. Na vstup se vloží očekávaný token, tedy *id*, s implicitní hodnotou err_id. Analyzátor zůstává v chybovém stavu. Opakuje se tento bod.
- Nastává resynchronizace zásobníku. Na vrcholu zásobníku je terminál. Analyzátor není resynchronizovaný, protože terminály *id* a *string* nejsou totožné. Dochází ke vložení terminálu "*;*". Opakuje se tento bod, analyzátor stále zůstává v chybovém stavu.
- Nastává resynchronizace zásobníku. Analyzátor stále není resynchronizovaný. Na vrcholu zásobníku je terminál. Terminál *end* na zásobníku terminálu *end* na vstupu. Analyzátor je synchronizován, může se vrátit do stavu, jako by chyba nenastala.

4.7.4 Metoda zotavování z chyb pomocí vkládání symbolů

Tato metoda je známá jako *FMQ error correction method*, jejími autory jsou Fisher, Milton a Quiring.V této metodě je množina Kontext tvořena všemi terminálními symboly.

Fisher, Milton a Quiring tvrdí, že nelze zjistit, co chtěl programátor přesně napsat. Proto by bylo chybné odstraňovat či měnit nějaký symbol. Proto se symboly již načtenými nebude manipulováno. Správný vstup je zajištěn pouze vložením symbolů kolem chyby. Aby se dala metoda použít, je nutné mít *insert-correptable* gramatiku, což je LL(1) gramatika s okamžitou detekcí chyb. To znamená že tato metoda je aplikovatelná na LL(1) gramatiku bez epsilon pravidel. Fisher, Tai and Milton dokazují, že tato metoda se dá aplikovat i na LL(1) gramatiky, kde pro každý neterminál, který derivuje epsilon platí, že tak dělá explicitně skrz epsilon pravidlo.V knize Parsing Techniques: A Practical Guide je psáno, že ne každá gramatika je gramatika s okamžitou detekcí chyb. Příkladem může být gramatika, která začíná s terminálem *program* a terminál *program* se již nikde jinde v pravidlech nevyskytuje. Potom, jestliže za sebou následují dva tokeny s terminálem *program*, tato metoda nemůže chybu napravit.Přesně taková gramatika se však v této bakalářské práci používá. Jen místo slova *program* užívá pojmenování *begin*.

4.7.5 Metoda lokálního zotavování z chyb na základě nejnižší ceny

Podobně jako metoda FMQ, tedy metoda zotavování z chyb pomocí vkládání symbolů, je tato metoda založena na technice zotavení se z chyby pomocí modifikace terminálního symbolu v místě detekce chyby. Metoda FMQ však pouze vkládá symboly. Metoda lokálního zotavování z chyb na základě nejnižší ceny opravuje chybu smazáním symbolu nebo vložením sekvence terminálů či nonterminálů nebo změněním symbolu. Metoda lokálního zotavování z chyb na základě nejnižší ceny bere v potaz pouze chybový stav jako takový a symbol následující. Oprava je udělána na základě nejnižší ceny: každý symbol má jistou cenu vložení, každý terminál, jenž byl vymazán, má svojí cenu, každá změna má svou cenu. Všechny ceny jsou nastaveny tvůrcem analyzátoru.

Její nevýhodou a zároveň výhodou je, že metoda bude vždy tak dobrá, jako tvůrce syntaktického analyzátoru.Tato metoda nezávisí na konkrétním způsobu analýzy, její implementace však již na způsobu analýzy závisí. Tato metoda byla implementována v LL, LR a Earley analyzátorech.

5 Implementace

V rámci bakalářské práce byl implementován syntaktický analyzátor pro analýzu shora dolů, se zotavením z chyb pomocí metody Kontext tvořený pokračovací metodou - pokračování u LL analyzátoru. Používá gramatiku uvedenou v této práci, stejně tak množinu First a LL tabulku uvedenou v kapitole 3 a pokračovací gramatiku z kapitoly 4. Ač se v praxi využívá více rekurzivní sestup jak prediktivní syntaktická analýza, implementovaný syntaktický analyzátor používá druhého přístupu s ohledem na větší názornost. Cílem bylo zejména ukázat rozdíl mezi syntaktickým analyzátorem se zotavením a bez zotavení, a také postup analyzátoru při zotavení.

Program je napsán v jazyce C, neb je tento jazyk je často považován za základní, učebnicový. Výsledkem je konzolová aplikace.

Syntaktický analyzátor očekává na vstupu dva povinné parametry: vstupní a výstupní soubor. Ve vstupním souboru jsou očekávány tokeny, tedy terminály, které by syntaktický analyzátor obdržel od lexikálního analyzátoru. Tokeny neobsahují žádnou hodnotu, která v tokenu byla lexikálním analyzátorem uložena. Hodnoty potřebuje až sémantický analyzátor, který však není součástí bakalářské práce. Do výstupního souboru se zapíše posloupnost pravidel, použitých při syntaktické analýze. Na standartní výstup vypisuje syntaktický analyzátor stav zásobníku. V případě chyby vypisuje, co při zotavení provádí: hledání prvního neterminálu na zásobníku, přepsání jeho pravou stranou, přeskakování symbolů ze vstupu a následná synchronizace analyzátoru do původního stavu.

Nepovinným parametrem je parametr -n. Jestliže je tento parametr zadán, proběhne syntaktická analýza bez zotavení.

Samozřejmostí je parametr -h či —help pro nápovědu.

Příklad 8

Ve vstupním souboru může být následující program:

```
1.begin
2.  id =  ;
3.end
```

tedy tato posloupnost tokenů:

```
T_begin T_id T_equal T_semicolon T_end
```

Jestliže není zapnuté zotavování z chyb, výsledek bude následující:

```
Parser is reading from soubory/fail2.txt and write to
soubory/out2.txt

           stack| input |rule
 14             |   1|r: 1
 0 15 1         |   1|POP
 0 15           |   5|r: 3
 0 15 2 16      |   5|r: 10
 0 15 2 19 6 5  |   5|POP
 0 15 2 19 6    |   6|POP
 0 15 2 19      |   2|error
```

```
Rule wasn't found, erroneous input
Finish
```

V souboru out1.txt bude následující posloupnost pravidel:

```
Used rules:
r1
r3
r10
ERROR
```

Jestliže však zotavení z chyb zapnuté je, výsledek bude následující:

```
~/Develop/bakalarka/bp/bp:  ./xobluk00 soubory/fail2.txt soubory/out2.txt
Parser is reading from soubory/fail2.txt and write to soubory/out2.txt

        stack           | input |rule
 14                     |   1   |r: 1
 0 15 1                 |   1   |POP
 0 15                   |   5   |r: 3
 0 15 2 16              |   5   |r: 10
 0 15 2 19 6 5          |   5   |POP
 0 15 2 19 6            |   6   |POP
 0 15 2 19              |   2   |error

Error recovery:
Nonterminal N_expression(19) is derived using continuation grammar
Replaced by its right-hand side: : N_exp(20)  N_term(18)
Nonterminal N_exp(20) is derived using continuation grammar
Replaced by its right-hand side: : T_epsilon(13)
Nonterminal N_term(18) is derived using continuation grammar
Replaced by its right-hand side: : T_int(8)
Inserted terminal:  T_int(8)
Inserted terminal:  T_semicolon(2)
Nonterminal N_body(15) is derived using continuation grammar
Replaced by its right-hand side: : T_epsilon(13)
Error recovery was successful, parsing continues

        stack           | input |rule
 0                      |   0   |POP

Finished successfully!:)
~/Develop/bakalarka/bp/bp:  cat soubory/out2.txt
Used rules:
r1
r3
r10

Error recovery:
r5 (continuation grammar)
r6 (continuation grammar)
r4 (continuation grammar)
r1 (continuation grammar)
Error recovery was successful, parsing continues
```

Úkolem této práce bylo zabývat se syntaktickým analyzátorem a nikoliv analyzátorem lexikálním. Aplikace předpokládá, že lexikální chyby jsou odhaleny, že každý program končí speciálním ukončovacím terminálem T_end. Jelikož v této práci lexikální analyzátor představuje člověk a ten není neomylný, probíhá kontrola, že terminály ve vstupním souboru jsou validní. Program si také umí

poradit se vstupem, který není ukončen speciálním ukončujícím terminálem T_end. Pokud se však budou na vstupu vyskytovat dva ukončující terminály T_end, syntaktický analyzátor tuto chybu neodhalí, respektive skončí u prvního terminálu. Aby dokázal tuto chybu odhalit, musel by plnit i funkci lexikálního analyzátoru a to není předmětem této bakalářské práce.

5.1 Rozvržení aplikace

Aplikace se skládá z několika částí. Z části, ve které analyzátor zpracovává korektní vstup a z části, ve které analyzátor zpracovává vstup chybný. V modulu main.c lze nalézt definovaná pravidla gramatiky, pravidla pokračovací gramatiky a LL tabulku.

V modulu parsing.c probíhá syntaktická analýza. Probíhá pouze ze shora dolů. Princip spočívá v porovnání vrcholu zásobníku se vstupem. Je-li na vrcholu neterminál, derivuje se dle LL tabulky. Je-li na zásobníku terminál, snaží se shodnout s terminálem na vstupu. Jestliže není nalezeno v LL tabulce pravidlo nebo se terminál na vrcholu zásobníku neshodne s terminálem na vstupu, nastává chyba. Jestliže je zotavení z chyb povoleno, zavolá se funkce error_recovery z modulu error_recovery.c

Modul error_recovery.c obsahuje funkci error_recovery, která postupuje podle algoritmu uvedeného v kapitole 4 u popisované metody zotavení. Postupně tedy volá funkce na nalezení prvního neterminálu na zásobníku, jeho derivaci pomocí pravidel pokračovací gramatiky, dále přeskočení symbolů na vstupu, jenž nenáleží množině Kontext a konečně resynchronizaci zásobníku. V případě, že je na vrcholu terminál, spočívá resynchronizace v případném vložení hledaného terminálu na zásobník. Jestliže je na vrcholu neterminál, opakuje se algoritmus od prvního kroku.

5.2 Výstup aplikace

Aplikace tiskne na výstup probíhající proces analýzy. V případě, že na vstupu není chyba, ukazuje zásobník, vstup a pravidlo, které se použije. Do výstupního souboru tiskne použitá pravidla, tedy levý rozbor. Pokud chyba nastala, tiskne na standartní výstup podrobný postup zotavení. S ohledem na různá nastavení terminálu je v modulu print_me.c možné změnit šířku odsazení při tisku zásobníku, případně způsob výpisu názvu terminálů a nonterminálů.

6 Závěr

Formální jazyky a překladače jsou nezbytnou součástí informatiky. Bez překladačů by nebylo možné psát programy ve vyšších programovacích jazycích. Jelikož počítač nedokáže uchopit jiný než strojový kód, musí existovat překladač, který vyšší programovací jazyk (například jazyk C) do strojového přepíše. Překladač se skládá z několika částí. Z lexikálního analyzátoru, syntaktického analyzátoru, sémantického analyzátoru, generátoru vnitřního kódu, optimalizátoru vnitřního kódu a generátoru strojového kódu.

Člověk není neomylný a v programu dělá chyby. Chyby mohou vzniknout na několika úrovních. Jestliže se jedná o špatně napsaná slova a překlepy, jsou to lexikální chyby a odhalí je lexikální analyzátor. Pokud programátor používá například nedefinovanou proměnnou, či porovnává nekompatibilní čísla, jedná se o chybu sémantickou, tedy významovou, chybu odhalí sémantický analyzátor. Tato práce se zabývá chybami syntaktickými, tedy chybami, které objeví syntaktický analyzátor. Mezi typické syntaktické chyby patří vynechání středníku, přebývající závorka.

Odhalením syntaktické chyby je rozuměna situace, kdy syntaktický analyzátor neví, jak pokračovat v překladu. Je to proto, neboť nedokáže aplikovat žádné pravidlo z gramatiky na svůj vstup, tedy na posloupnost tokenů přicházející z lexikálního analyzátoru. Pokud syntaktický analyzátor nedokáže pokračovat, skončí překlad. To je nevhodné, protože se neodhalí další chyby v programu. Je tedy nutné vytvořit takový syntaktický analyzátor, který dokáže pokračovat v překladu, přestože narazí na chybu.

V kapitole 2 lze v příkladu 2 nalézt ukázkovou gramatiku. Gramatika není Turingovsky kompletní, slouží pouze pro lepší pochopení metod. V rámci této práce byly popsány metody zotavení při syntaktické analýze shora dolů. Je představeno několik metod spadajících do různých tříd. Rozdělení se v různých literaturách liší. V této bakalářské práci je dělení podle [1], [3]. Metody se dělí podle toho, s jakým okolím, tedy kontextem, pracují. Pokud používají globální kontext, tedy celý vstup, jedná se o metody globálního zotavení z chyb. V případě použití části kontextu v okolí se jedná o zotavení na úrovní fází. Pokud používají pouze aktuální stav překladače a aktuální vstupní symbol, jedná se o metody lokálního zotavení z chyb. Jestliže nepoužijí žádný kontext, jedná se o sufixové metody. Poslední skupinou jsou ad hoc metody, které nepracují s žádným kontextem, neboť chybám předchází ještě před překladem. Nejobsáhlejší třídou jsou metody lokálního zotavení z chyb. V rámci této třídy je popsána i implementovaná metoda a metoda Kontext tvořený pokračovací metodou.

Tato je neobvyklá v tom, že používá dvě gramatiky. Jednu gramatiku, která je definována na počátku a kterou je popsán jazyk. S touto gramatikou pracují všechny metody a podle této gramatiky probíhá syntaktická analýza. Jestliže je však nalezena chyba, použije se druhá, pokračovací gramatika. Pokračovací gramatika je gramatika tvořena pouze z některých pravidel gramatiky původní. Jejím cílem je změnit nejkratší cestou vstup tak, aby se dalo validně pokračovat dále s gramatikou původní. Tato metoda je podrobněji rozebrána v [3], [4]. Metodu lze použít pro LL i LR syntaktický analyzátor, implementace u LR syntaktického analyzátoru je však náročnější. Kouzlo metody spočívá v tom, že pokračovací gramatiku u LL syntaktického analyzátoru lze snadno vypočítat předem. Jak se počítá, je ukázáno v kapitole 4. Tato metoda lze aplikovat i na gramatiky s epsilon pravidly. Jak je psáno v úvody kapitoly 4, chybu není možné objektivně opravit, protože pouze autor programu ví, co chtěl doopravdy napsat. Výhoda této metody tkví v tom, že oproti jiným metodám se nesnaží chybu komplikovaně napravit, naopak se nejrychlejší cestou z chyby dostává. V rámci práce byl modifikován algoritmus, který se v literatuře pro zotavení z chyb při poslední fázi zotavení používá. Jsou zde podrobněji rozepsány důvody modifikace. Jeden z důvodů modifikace je, aby se analyzátor, který je nazýván zotaveným, nedostal ihned v prvním kroku při běhu v normálním módu opět do chybového stavu.

Cílem práce bylo zmínit metodu, která není v literatuře tak často zmiňována a přitom není příliš složitá na implementaci a nespotřebovává příliš strojového času. Zároveň dává programátorovi přesně najevo, kde našel syntaktický analyzátor chybu, jaké symboly odstranil a jaké naopak přidal. Díky snadné uchopitelnosti této metody může být bakalářská práce použita jako výukový materiál. Pomocí volitelného parametru aplikace lze jednoduše porovnat syntaktický analyzátor se zotavením z chyb a bez zotavení z chyb.

Tato bakalářská práce se dá dále rozšiřovat. Pěkným rozšířením by bylo implementovanou metodu zpracovat graficky. Tak by se dala využít i při výuce ještě lépe a názorněji. Dalším rozšířením by byla implementace více metod a jejich porovnání se zmiňovanou metodou. Metoda by se také dala implementovat v LR analyzátoru. Jak však již bylo řečeno, implementace v LR analyzátoru by nebyla tak snadná.

7 Literatura

[1] Aho, A. V., Sethi, R., Ullman, J. D.: *Compilers : principles, techniques, and tools*, Addison-Wesley, 2nd ed., 2007, ISBN: 0321486811

[2] Tremblay, J. P., Sorenson, P. G.: *The Theory and Practice of Compiler Writing*, Mcgraw-Hill College, 1985. ISBN 0070651612

[3] Grune, D., Jacobs, C. J.H.: *Parsing Techniques: A Practical Guide*, Springer; 2nd ed. 2008 edition, ISBN: 1441919015

[4] Grune, D., van Reeuwijk, K., Bal, E. H., Jacobs, C. J. H., Langendoen, K.: *Modern Compiler Design*, Springer; 2nd ed. 2012 edition, ISBN: 1461446988

[5] Meduna, A.: slidy k předmětu IFJ, [online] dostupné na URL **http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/**, poslední změna 15. září 2014

[6] Meduna, A.: *Formal Languages and Computation*, Taylor & Francis Informa plc, 2014, ISBN: 978-1-4665-1345-7

Příloha A

Přiložené CD

Obsah přiloženého CD

- text bakalářské práce
- zdrojové kódy syntaktického analyzátoru
- Makefile
- README
- vzorový vstupní soubor input_file.txt