VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

GRAMATICKÉ SYSTÉMY APLIKOVANÉ V PŘEKLADAČÍCH

GRAMMAR SYSTEMS APPLIED TO COMPILATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAKUB REŠ**
AUTHOR

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Reš Jakub**
Program: Informační technologie
Název: **Gramatické systémy aplikované v překladačích**
Grammar Systems Applied to Compilation
Kategorie: Překladače

Zadání:

1. Dle instrukcí vedoucího se seznamte s gramatickými systémy a vybranými metodami, které jsou použity při konstrukci kompilátorů, např. pro syntaxí řízený překlad.
2. Zaveďte nové metody, které jsou založeny na gramatických systémech. Tyto metody budou založeny na kombinaci několika dílčích metod z bodu 1.
3. Studujte vlastnosti metod z bodu 2 dle instrukcí vedoucího. Porovnejte jejich vlastnosti s klasickými metodami.
4. Aplikujte metody z bodu 2 v kompilátorech. Implementujte a testujte je.
5. Zhodnoťte dosažené výsledky. Diskutujte další vývoj projektu.

Literatura:

- Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1-3, Springer, 1997, ISBN 3-540-60649-1
- Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition), Pearson Education, 2006, ISBN 0-321-48681-1
- Meduna, A.: Automata and Languages, Springer, London, 2000, ISBN 978-1-85233-074-3
- Meduna, A.: Elements of Compiler Design, New York, Taylor & Francis, 2008

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a část 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Meduna Alexander, prof. RNDr., CSc.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 27. října 2020

Abstrakt

Cílem této práce je navrhnout metodu konstrukce kompilátoru založenou na konceptu gramatických systémů, která umožní uživateli libovolně kombinovat dílčí existující konstrukční metody.

Řešení zvoleného problému spočívá ve využití kooperujících distribuovaných gramatických systémů. Z těch byl převzán princip členění na komponenty, sekvenčně spolupracující na společné výsledné větě dle derivačního módu.

Vzniká nám tedy konstrukce menších, úzce specifikovaných jednotek, schopná vzájemně předávat řízení a společně tak analyzovat složité jazyky. Každá z těchto komponent využívá jedné z existujících metod analýzy a její libovolné implementace.

Přínosem této práce je navržení a ukázání užití principu gramatických systémů, které nám umožňuje konstruovat kompilátor užitím libovolných metod a zároveň, díky gramatickým systémům, zvýšit jeho celkovou generativní sílu.

Abstract

The main goal of this work is to design a method of constructing a compiler based on grammar systems that allows it's user to by able to combine any existing construction methods.

Solution of this problem lies in utilization of cooperating distributed grammar systems. The principle of dividing compiler into sequentially cooperating components was used by this thesis.

So we have a construction of smaller, narrowly specified units that are able to pass control to each other and together analyze complex languages. Each of these components is using one of the existing methods of analysis and any way of it's implementation.

Benefit of this thesis is the design of construction method using principle of grammar systems, that allows us to use combination of any existing methods and brings overall higher generative power, and showing a possible way of using this method.

Klíčová slova

překladač, konstrukce překladače, syntaktická analýza, gramatické systémy, metody syntaktické analýzy

Keywords

compiler, compiler structure, syntax analysis, grammar systems, methods of syntax analysis

Citace

REŠ, Jakub. *Gramatické systémy aplikované v překladačích.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexander Meduna, CSc.

Gramatické systémy aplikované v překladačích

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. RNDr. Alexandera Meduny, CSc. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jakub Reš
26. dubna 2021

Poděkování

Děkuji panu prof. RNDr. Alexanderu Medunovi, CSc. za vedení a odborné konzultace této práce, cenné rady a kritiku. Dále bych rád poděkoval doc. PaeDr. Jiřímu Rychterovi, Ph.D. za podporu při tvorbě této práce a Mgr. Jiřině Rešové za připomínky ohledně jazykové stránky.

Obsah

Kapitola 1

Úvod

Jazyk je nedílnou součástí každodenního života lidí. Napříč historií vždy primárně sloužil na poli mezilidské komunikace. Jeho původ lze datovat do stejného období, jako vznik samotné lidské rasy. Tehdejší forma jazyka byla značně omezena. Skládala se z primitivních skřeků a gest. Rychle rozvíjejícím se zástupcům, již tehdy vysoce inteligentní rasy, velice brzy vznikla potřeba přenášet komunikací větší množství informací, ale především tyto cenné informace uchovávat. Během let tak vzniká i třetí forma jazyka, a to psaná. Až do nedávné doby sloužil jazyk pro komunikaci mezi lidmi. Během posledních, přibližně stopadesáti let, tedy v období prudkého rozvoje techniky, strojů a především počítačů, se objevila potřeba sdílet informace s novými subjekty, stroji. Vznikl tak koncept jazyka i pro komunikaci mezi člověkem a strojem.

Komunikační schopnosti obou stran konverzace se ovšem již od prvopočátku značně liší. Člověk je přizpůsoben pro komunikaci za použití složitého kombinování znaků do smysluplných slov konkrétního jazyka a ta následně do vět. Člověk se tomu tak naučil během svého života ať už intuitivně, pozorováním svého okolí, či učením se daného jazyka ve škole. Stroj, na druhou stranu, je neživým objektem postrádajícím inteligenci a schopnost samovolné komunikace. Byl lidmi sestrojen za jediným účelem. V rámci komunikace přijmout informace ve vhodné formě a na jejich základě vykonávat práci. Nejsložitějšími, a v dnešní době hlavními na poli strojů, jsou počítače.

Počítač operuje se dvěma logickými hodnotami, pravdou a nepravdou. Tyto hodnoty jsou symbolizovány čísly 1 a 0. Požadovaná forma poskytovaných informací je tudíž právě v této podobě. U prvních počítačů, vzhledem ke složitosti tehdejších programů, nečinila komunikace pomocí jedniček a nul nepřekonatelný problém, byla ovšem i tak velice náročná a často během ní docházelo k chybám.

Z tohoto důvodu vznikly tzv. překladače. První překladače se objevily na počátku 50. let 20. století. Překladač je program, který má za úkol transformovat větu jednoho jazyka do druhého. Vytvoření překladačů tak dalo vzniku prvním programovacím jazykům. Při překladu musí být vždy zachována ekvivalence funkcionality věty. To znamená, že věta cílového jazyka musí mít vždy stejný význam, jako věta původní. Za tímto účelem je třeba, aby byl překladač schopen rozeznat větnou skladbu vstupního jazyka. K tomu dochází za pomoci gramatiky.

Častým přístupem, a dále v této práci využívaným k procesu překladu jazyka, je syntaxí řízený překlad. Neformálně řečeno, jedná se o kontrolu skladby věty vstupního jazyka, její porozumění a následné sestrojení významově ekvivalentní věty ve výstupním jazyce.

Překladače, konkrétně jejich části pro kontrolu syntaxe jazyka, jsou typicky konstruovány jednou z následujících metod. První metodou je tzv. metoda shora dolů. Tato metoda,

abstraktně řečeno, rozvíjí větu jako celek na dílčí podčásti a dále na samotná slova. Odtud název shora dolů. Kontrastem této metody je tzv. metoda zdola nahoru. Pracuje přesně opačným způsobem. Pohlíží na větu jako na jednotlivá slova, ta následně slučuje dle gramatických pravidel do celků a ty nakonec do celé věty. Každý z těchto způsobů nese své výhody i nevýhody a každý se může hodit v různých situacích či při překladech různých jazyků, ba dokonce pouhých částí jazyka.

Do nedávné doby byl přístup ke konstrukci kompilátoru (překladače) z velké části za jedno s myšlenkou, že překladač funguje jako samostatná, soustředěná jednotka. Pracovaly samostatně, konstruované jednou metodou a řídící se jednou gramatikou.

V 80. letech minulého století se ale začala rapidně rozvíjet myšlenka distribuce prostředků překladu. S velkým potenciálem přišel i rozvoj výzkumu této oblasti. Výsledkem byl návrh rozčlenění gramatik do menších, vzájemně spolupracujících celků, a tím zvýšení jejich generativní síly. Dnes známé jako gramatické systémy.

Cílem této práce je návrh metody využití principu gramatických systémů, tedy vzájemná spolupráce menších jednotek, k sestavení překladače za využití libovolné kombinace existujících metod konstrukce. Práce tak dále navrhuje rozšíření možnosti informačně obsáhlé a snadné komunikace mezi člověkem a strojem.

1.1 Předpoklady znalostí čtenáře

Práce předpokládá, že čtenář je obeznámen se základy z teorie automatů a formálních jazyků. Jsou zde využívány základní pojmy, které lze, včetně definic, nalézt v anglicky psaných dílech Meduny [3] nebo Hopcrofta, Motwaniho a Ullmana [1]. Z řad česky psaných děl je možno například využít Černé, Křetínského a Kučerovu publikaci [13]. V práci jsou též využívány obecně uznávané konvence značení a zápisu, které lze také nastudovat z výše uvedených publikací.

Kapitola 2

Konstrukce kompilátoru

V této kapitole jsou uvedeny základní informace ohledně stavby a konstrukce kompilátoru. Nejdříve je uvedena a popsána struktura překladače, typické logické části a jejich fungování. Dále je sekce zaměřená na syntaxí řízený překlad.

2.1 Struktura překladače

Následuje sekce o složení překladače. Jako první bude popsán překladač jako celek a jeho obecné fungování. Dále jsou uvedeny typické části překladače, jejich účel a principy práce.

Přehled

Kompilátor je program, který překládá vstupní jazyk na výstupní. Oba tyto jazyky, tedy vstup a výstup překladače, jsou funkčně ekvivalentní. Proces překladu lze rozložit do logických celků. Tyto celky pak obsahují jednotlivé kroky překladu, které mají na starosti části kompilátoru tvořící logické schéma překladače.

Logickými celky jsou: lexikální analýza, syntaktická analýza a generování kódu. Každá tato fáze přetváří, rovněž by se dalo říci, že překládá svůj vstupní jazyk na výstupní dle požadavku následující fáze. Během lexikální analýzy dochází ke čtení, analýze a překladu zdrojového kódu na posloupnost lexému. Ta je dále vhodnou formou zasílána jako vstup syntaktické analýze, která přicházející řetězec rovněž analyzuje a překládá na derivační strom. Generátor kódu následně přijme tento abstraktní strom a na jeho základě vytváří cílový program.

Jednotlivé kroky překladu zaopatřují specializované části. Každá tato část provádí jednu souvislou aktivitu. Části překladače jsou podrobně popsány dále v této sekci. Následuje seznam a stručný popis:

1. Lexikální analýza

   Lexikální analyzátor – kontroluje lexikální správnost vstupního programu, generuje posloupnost tokenů

2. Syntaktická analýza

   Syntaktický analyzátor – přijímá posloupnost tokenů, kontroluje syntaktickou stránku programu a generuje abstraktní syntaktický strom

   Sémantický analyzátor – pracuje po boku syntaktického analyzátoru, kontroluje sémantiku vstupního jazyka

3. Generátor kódu

   Generátor vnitřního kódu – přijímá abstraktní strom a generuje vnitřní kód překladače

   Optimalizátor – přijímá vnitřní kód a optimalizuje jej

   Generátor cílového kódu – přijímá optimalizovaný vnitřní kód a na jeho základě generuje cílový program

Lexikální analyzátor

Lexikální analyzátor je první jednotkou kompilátoru. Vstupem této části je zdrojový kód ve formě řetězce znaků. Úkolem lexikálního analyzátoru je v posloupnosti rozeznat a klasifikovat lexémy a následně odeslat formou tokenů syntaktickému analyzátoru. Další důležitou aktivitou je odstranění komentářů a prázdných míst ve zdrojovém kódu.

Každý lexém vstupního programovacího jazyka je typicky možné popsat regulárním výrazem. Na základě těchto výrazů jsou sestaveny a spojeny konečné automaty, rozšířené o výstupní pásku. Tento automat pracuje na základě algoritmu **Určení typu lexému**[5]:

Vstup: DKA **M** rozpoznávající lexémy
Výstup: typ lexému
Metoda:
while a je další znak ze zdrojového programu **and M** může udělat přechod se symbolem **a do**
  přečti **a**
  udělej přechod v **M** pro symbol **a**
  if M je v koncovém stavu **then**
    urči typ lexému, který koresponduje danému koncovému stavu
  else
    zahlas lexikální chybu (napiš zprávu atd.)
  end if
end while

Takto určené typy lexémů jsou na požádání zakódovány a zasílány syntaktickému analyzátoru ve formě tzv. *tokenů*. Tokeny jsou jednotnou reprezentací lexémů. Typicky mají obecný tvar: [*typ*, *atribut*]. Typ udává základní charakteristiku tokenu a atribut tento typ specifikuje. Atributy tokenů se běžně liší ([int, 42], [float, 42.5], [:, <nic>]).

Lexikální analyzátor též běžně pracuje s tabulkou symbolů. Nově nalezené identifikátory jsou zaneseny do tabulky symbolů a v atributu tokenu je zaslán pouze jejich odkaz.

Syntaktický analyzátor

Syntaktický analyzátor je druhou jednotkou překladače. U syntaxí řízeného překladu, popsaného dále v sekci 2.2, se jedná o hlavní řídící orgán kompilátoru. Syntaktický analyzátor zasílá signály ostatním jednotkám a ty na jeho požádání vykonávají kroky své práce. V této jednotce probíhá syntaktická analýza vstupního řetězce (zde již posloupnosti tokenů). Výstupem syntaktického analyzátoru je simulace konstrukce derivačního stromu. V praxi je možné tento výstup označit jako posloupnost čísel použitých pravidel gramatiky. Důležitou definicí je definice samotné syntaktické analýzy (převzané z opory předmětu IFJ[4]):

Definice 2.1.1. Buď $G = (N, T, P, S)$ bezkontextová gramatika, která má n pravidel očíslovaných $1, \ldots, n$ a nechť $w \in L(G)$. Syntaktická analýza metodou shora dolů je proces,

který vede k nalezení posloupnosti čísel pravidel užitých při levé derivaci věty w. Syntaktická analýza metodou zdola nahoru je proces, který vede k nalezení obrácené posloupnosti čísel pravidel použitých při pravé derivaci věty w.

Metody syntaktické analýzy, zmíněné v definici výše, jsou dále podrobně popsány v kapitole 3.

Sémantický analyzátor

Sémantický analyzátor je jednotkou následující syntaktický analyzátor. Vstupem je jí simulace konstrukce derivačního stromu, vygenerovaná předchozí částí. Hlavním úkolem je kontrola vstupního programu ze sémantické stránky. Patří sem například kontrola datových typů či deklarace proměnných. Sémantický analyzátor bývá často pomyslnou nadstavbou nad syntaktickým analyzátorem. V praxi je typicky spojen s předchozí jednotkou v tzv. *parser*. Výstupem této části překladače je abstraktní syntaktický strom.

Za účelem jednoduchého přeložení abstraktního syntaktického stromu dále je žádoucí tento strom převést z infixové do postfixové polské notace. V případě analýzy metodou zdola nahoru lze přímo sémantickým analyzátorem generovat 3-adresný kód.

Generátor vnitřního kódu

Generátor vnitřního kódu je jednotka překladače zajišťující překlad abstraktního syntaktického stromu, vytvořeného během syntaktické analýzy, na jeho reprezentaci ve *vnitřním kódu* kompilátoru. V praxi je ovšem často tato část spojena se syntaktickým/sémantickým analyzátorem. Společně tak tvoří modul pro příjem posloupnosti tokenů, jejich syntaktickou a sémantickou kontrolu a následný překlad do vnitřního kódu.

Hlavními důvody tohoto mezikroku překladu jsou:

1. Nezávislost vnitřního kódu a tudíž i majoritní části překladače na strojovém jazyce, čímž lze kompilátor snadno přenášet na různé typy počítačů[4]
2. Přehledná implementace překladače (přímé generování cílového kódu by mohlo být složité a neprůhledné)[10]
3. Možnost úpravy vnitřního kódu s cílem optimalizovat výsledný program[4]

Generování vnitřního kódu je typicky prováděno třemi způsoby[9]. Prvním je překlad pomocí dvou gramatik. Tento způsob je implementován vložením překladové gramatiky za syntaktickou analýzu, tedy pracuje na základě překladu levého rozboru na výstupní řetězec.

Druhou možností je překlad pomocí jedné gramatiky se dvěma pravými stranami. Tato metoda spočívá v provádění dvou pravých stran pravidla při syntaktické analýze. První část je využívána pro kontrolu vstupu a druhá pro generování výstupního řetězce.

Třetí možností je přímé generování 3-adresného kódu. Tento postup lze využít při konstrukci syntaktického analyzátoru metodou zdola nahoru 3.2, při níž je možné 3-adresný kód přímo generovat na základě využívaných pravidel.

Optimalizátor

Optimalizátor je částí překladače, která není ke správnému fungování nutně zapotřebí. Slouží totiž k transformaci vnitřního kódu na funkčně ekvivalentní, ale efektivnější podobu.

Vnitřní kód je, za účelem optimalizace, rozdělen do tzv. *základních bloků*[6]. Existují dva základní druhy optimalizací[6]:

1. Lokální optimalizace vs. globální optimalizace

   Lokální – v rámci základního bloku

   Globální – napříč více bloky

2. Rychlost vs. velikost cílového programu

Generátor cílového kódu

Generátor cílového kódu je poslední etapou v procesu překladu. Dochází zde k překládání vnitřního kódu kompilátoru na cílový program zapsaný ve strojovém jazyce počítače, nebo v jazyce symbolických adres. Tato část překladače je tudíž závislá na konkrétním počítači, na který je kompilátor navržen.

Cílový program je z vnitřního kódu, zapsaného formou trojic, generovaný na základě následujícího algoritmu[4]:

Generování cílového programu pro trojice
Vstup: Vnitřní forma programu zapsaná pomocí trojic.
Výstup: Cílový program
Metoda:
(I) přečti trojici – pokud jsme již přečetli všechny trojice (tj. celý program ve vnitřní formě), pak je na výstupu vytvořen cílový program a generování ukončíme;
(II) podle operátora trojice vyber příslušnou tabulku. Nalezni v této tabulce položku určenou operandy trojice. Tyto operandy jsou uvedeny buď přímo v trojici a nebo – v případě, že ve zpracované trojici je použit výsledek některé předcházející trojice – na vrcholu zásobníku;
(III) jestliže je některý z operandů v zásobníku uložen ve střádači (kromě operandů, jež vstupují do právě generovaných instrukcí), ulož střádač (tj. proveď : STORE Wi; i := i + 1) a v zásobníku nahraď S označením proměnné;
(IV) vygeneruj instrukce z tabulky (viz (2));
(V) zruš v zásobníku použité operandy a na vrchol zapiš adresu, kam byl uložen výsledek trojice;
(VI) jdi na (I)

2.2 Syntaxí řízený překlad

Tato sekce popisuje princip syntaxí řízeného překladu. V počátku jsou rozebrána překladová schémata. Následně je navázáno definicí samotného syntaxí řízeného překladu. Na konci je uveden algoritmus transformace derivačního stromu, převzatý z opory předmětu IFJ[4].

Hlavní myšlenkou syntaxí řízeného překladu je povýšení syntaktické analýzy na řídící fázi procesu překladu. V rámci povýšení je k syntaktické analýze též typicky přiřazena sémantická analýza a často také generování vnitřního kódu. Fáze syntaktické analýzy tímto nabývá funkcionality překladu vstupního řetězce na vnitřní kód kompilátoru. Za účelem formálního popisu této akce jsou zavedeny tzv. *syntaxí řízená překladová schémata*[4]:

Definice 2.2.1. Syntaxí řízené překladové schéma je pětice

$$T = (N, V_1, V_0, R, S),$$

kde

- N je abeceda neterminálů,
- V_1 je vstupní abeceda,
- V_0 je výstupní abeceda,
- R je konečná množina pravidel tvaru $A \to x, y$, kde $A \in N, x \in (N \cup V_1)^*, y \in (N \cup V_0)^*$ a přitom neterminály z y jsou permutací neterminálů z x,
- S je počáteční symbol, $S \in N$.

Za pomoci takto definovaných schémat je dále možno formálně zapsat i samotný syntaxí řízený překlad[4]:

Definice 2.2.2. Buď $T = (N, V_1, V_0, R, S)$ syntaxí řízené překladové schéma. Pak $P(T)$ se nazývá syntaxí řízený překlad. Gramatika

$$G_1 = (N, V_1, R_1, S),$$

kde

$$R_1 = A \to x : A \to x, y \in R$$

se nazývá vstupní gramatika schématu T a gramatika

$$G_0 = (N, V_0, R_0, S),$$

kde

$$R_0 = A \to y : A \to x, y \in R$$

se nazývá výstupní gramatika schématu T.

Syntaxí řízený překlad je možné obecně popsat jako transformaci derivačního stromu vstupní gramatiky na derivační strom výstupní gramatiky dle níže uvedeného algoritmu *transformace derivačního stromu*[4]:

Vstup: Syntaxí řízené překladové schéma $T = (N, V_1, V_0, R, S)$ se vstupní gramatikou $G_1 = (N, V_1, R_1, S)$ výstupní gramatikou $GO = (N, V_0, R_0, S)$, a derivační strom D_x věty $x \in V^*$ v G_1.
Výstup: Derivační strom D_y věty $y \in V^*$ ve výstupní gramatice G_0 tak, že $(x, y) \in P(T)$.

Metoda:
(1) Aplikuj (2) na kořen D_x.
(2) Nechť tento krok je aplikován na uzel u, jenž není koncový a nechť $u_1, u_2, \ldots, u_n$ jsou přímí následovníci uzlu u, jejichž ohodnocení (zleva doprava) tvoří řetěz x a buď $A \to x, y \in R$.
(a) Odstraníme z D_x všechny koncové uzly;
(b) Se zbývajícími uzly provedeme permutaci včetně příslušných podstromů tak, aby jejich pořadí odpovídalo pořadí neterminálů v y;
(c) připojíme nové koncové uzly tak, aby ohodnocení všech přímých následovníků uzlu u tvořilo řetěz y;
(d) Krok (2) aplikujeme na všechny přímé následovníky uzlu u, které nejsou koncové uzly.
(3) Výsledný strom je D_y.

Kapitola 3

Metody konstrukce kompilátoru

Tato kapitola pojednává o současném stavu v oblasti konstrukce kompilátoru. Jsou zde ukázány dvě metody konstrukce kompilátoru. První metodou je metoda *shora dolů*. V další části následuje metoda *zdola nahoru*. U obou těchto metod je popsán jejich princip a možné způsoby implementace. Informace obsažené v této kapitole jsou výtažkem informací z dokumentů opora předmětu IFJ[4] a patřičné prezentace předmětu IFJ[7][8] a inspirována publikací [12].

3.1 Metoda shora dolů

V následující sekci je popsána metoda *shora dolů* syntaktické analýzy. První část se věnuje teoretickému pricipu fungování a analýze založené na LL tabulce. Druhá část sekce je věnována dvěma přístupům k implementaci – rekurzivnímu sestupu a prediktivní analýze.

Princip

Princip analýzy metodou shora dolů je možné pozorovat na derivačním stromě věty. Při analýze je totiž postupováno od kořene stromu k jeho listům – odtud název *shora dolů*. Při tomto postupu vzniká použitím pravidel gramatiky levý rozbor věty. Tuto skutečnost lze ukázat na jednoduchém příkladu (inspirováno příkladem v opoře předmětu IFJ[4]):

Mějme bezkontextovou gramatiku

$$G = (\{A, B, C\}, \{i, (,), +, /\}, P, A)$$

$$P = \{1 : A \to B/C,\ 2 : A \to C,\ 3 : B \to (A),\ 4 : B \to i,\ 5 : C \to C + B,\ 6 : C \to B\}$$

Větu $(i + i)/i$ je následovným způsobem možné v gramatice G dosáhnout za pomoci levé derivace:

$$\begin{array}{lll} A & \to B/C & 1 \\ & \to (A)/C & 3 \\ & \to (C)/C & 2 \\ & \to (C + B)/C & 5 \\ & \to (B + B)/C & 6 \\ & \to (i + B)/C & 4 \\ & \to (i + i)/C & 4 \\ & \to (i + i)/B & 6 \\ & \to (i + i)/i & 4 \end{array}$$

Levým rozborem této věty je tedy 1 3 2 5 6 4 4 6 4.

Způsob uvedený výše může ovšem být ve spoustě případech nedeterministický, tudíž nevhodný pro praktické použití. Za tímto účelem vnikla tzv. *LL analýza*. Pod pojmem LL analýza se rozumí syntaktická analýza, metodou shora dolů, založená na LL tabulce.

Principem této analýzy je sestrojení tabulky, na základě LL gramatiky (blíže popsané v opoře předmětu IFJ[4] či prezentacích předmětu IFJ[7]), obsahující informace o neterminálech, terminálech a pravidlech. Obsah tabulky říká, jaké pravidlo je zapotřebí použít pro přepsání aktuálního neterminálu za účelem zpracování konkrétního terminálu.

Implementace

V praxi jsou k implementaci syntaktického analyzátoru, metodou shora dolů, typicky dva přístupy. Prvním z nich je *rekurzivní sestup*.

V tomto přístupu je každý neterminál reprezentován vlastní procedurou. Tyto procedury jsou na základě informace o nadcházejících tokenech a LL tabulce rekurzivně volány.

Tento přístup je prezentován na následujícím příkladu, převzatém z prezentace předmětu IFJ[7]:

LL tabulka příkladu:

	i	+	*	(	)	$
E	1			1		
E’		2			3	3
T	4			4		
T’		6	5		6	6
F	8			7		

Dále existují procedury `GetNextToken` a dílčí pro každý neterminál, obsahující simulaci pravidel na základě tabulky a následujícího tokenu. Příklad těla funkce neterminálu `E` je následující:

```
function E: boolean;
begin
    E := false;
    if token in ['i', '('] then
        { simulace pravidla 1: E → TE' }
        E := T and E1;
end;
```

kde `T` a `E1` jsou opět metody pro simulaci rozvoje neterminálů T a E’.

Při samotné analýze jsou volány jednotlivé funkce, tvořící tak strom volání, identický s derivačním stromem.

Druhým častým přístupem k implementaci je *prediktivní analýza*. Tento přístup využívá LL tabulky a zásobníku. Během analýzy jsou na zásobník ukládány pravé strany pravidel v obráceném pořadí, tedy na vrcholu zásobníku se vždy nachází (ne)terminál výstupního řetězce nejvíce vlevo. Tento zásobník je po dobu analýzy porovnáván se vstupním řetězcem a dále rozvíjen či odstraňován dle pravidel LL tabulky. Celý algoritmus, převzaný z opory předmětu IFJ[4], zní následovně:

Vstup: LL-tabulka pro $G = (N, T, P, S)$; $x \in T^*$
Výstup: Levý rozbor pro x, pokud $x \in L(G)$ jinak chyba
Metoda:

push($) & push(S) na zásobník
repeat
  nechť **X** je vrchol zásobníku a **a** aktuální token
  case **X** of:
  X = $:
  if a = $ **then**
    úspěch
  else
    chyba
  end if
  X $\in T$:
  if X = **a then**
    pop(**X**) & přečti další **a** ze vstupního řetězce
  else
    chyba
  end if
  $X \in N$:
  if r : **X** $\rightarrow$ **x** $\in$ LL-tabulka[**X**, **a**] **then**
    zaměň na vrcholu zásobníku **X** za reversal(**x**) & zapiš **r** na výstup
  else
    chyba
  end if
until úspěch **OR** chyba

3.2 Metoda zdola nahoru

Tato sekce popisuje metodu zdola nahoru syntaktické analýzy. První část je věnována teoretickému pricipu fungování. Druhá část sekce je věnována jednomu z možných přístupů k implementaci – precedenční analýze.

Princip

Princip metody zdola nahoru syntaktické analýzy je opět, obdobně jako u metody popsané v předcházející sekci, možné pozorovat na derivačním stromě. Analýza touto metodou postupuje ve stromě, dle jména metody, *zdola nahoru*, tedy od listů ke kořeni. Provedením analýzy vzniká pravý rozbor věty. Princip je znovu ukázán na příkladu inspirovaném příklady z opory předmětu IFJ[4]:

Mějme bezkontextovou gramatiku

$$G = (\{A, B, C\}, \{i, (,), +, /\}, P, A)$$

$$P = \{1 : A \rightarrow B/C,\ 2 : A \rightarrow C,\ 3 : B \rightarrow (A),\ 4 : B \rightarrow i,\ 5 : C \rightarrow C + B,\ 6 : C \rightarrow B\}$$

Větu $i + (i/i)$ je následovným způsobem možné v gramatice G dosáhnout za pomoci pravé derivace:

$$
\begin{array}{lll}
A & \rightarrow C & 2 \\
& \rightarrow C + B & 5 \\
& \rightarrow C + (A) & 3 \\
& \rightarrow C + (B/C) & 1 \\
& \rightarrow C + (B/B) & 6 \\
& \rightarrow C + (B/i) & 4 \\
& \rightarrow C + (i/i) & 4 \\
& \rightarrow B + (i/i) & 6 \\
& \rightarrow i + (i/i) & 4
\end{array}
$$

Pravým rozborem této věty je tedy 4 6 4 4 6 1 3 5 2. Lze pozorovat, že pravým rozborem je **obrácená** posloupnost použitých pravidel.

Implementace

Dále je popsán jeden z možných přístupů k implementaci syntaktické analýzy metodou zdola nahoru. Tímto přístupem je precedenční analýza. Hlavními požadavky při použití jsou:

1. Neshodnost všech pravých stran pravidel gramatiky
2. Absence ε pravidel

Podstatou této metody je konstrukce a využití precedenční tabulky. Tato tabulka obsahuje symboly akcí, prováděných nad zásobníkem, dle aktuálního znaku na vstupní pásce a vrcholu zásobníku. Zmíněné akce jsou následující:

- redukce – v tabulce značena symbolem $>$
- posun – v tabulce značena symbolem $<$
- vložení – v tabulce značena symbolem $=$

Mimo zmíněné tabulka obsahuje prázdná pole. V případě pokusu o využití prázdného pole tabulky dochází k chybě analýzy. Kompletní význam těchto akcí je možno vidět v následujícím algoritmu precedeční analýzy, převzaného z prezentace předmětu IFJ[8]:

Vstup: Precedenční tabulka pro $G = (N, T, P, S)$; $x \in T^*$
Výstup: Pravý rozbor **x**, pokud $x \in L(G)$, jinak chyba
Metoda:
nechť funkce top vrací terminál na zásobníku nejblíže vrcholu
vlož $ na zásobník;
repeat
  nechť **a** = top & b = aktuální znak na vstupu,
  case Tabulka[a, b] of:
  = :
  push(b) & přečti další symbol b ze vstupu
  < :
  zaměň a za a< na zásobníku & push(b) & přečti další symbol b ze vstupu
  > :
  if <y je na vrcholu zásobníku **and** $r : A \rightarrow y \in P$ **then**
    zaměň <y za A & vypiš r na výstup

else
chyba
end if
prázdné políčko :
chyba
until b = $ **and** top = $
úspěch syntaktické analýzy

Důležitou součástí precedenční analýzy je precedeční tabulka. Tabulku je nutno ručně vyplnit. K jejímu sestrojení je třeba znát priority a asociativitu operátorů jazyka. Konstrukce této tabulky se řídí následujícími pravidly[8]:

1. Precedence operátorů

   Pokud má operátor i vyšší prioritu než operátor j

   $$op_i > op_j \text{ a } op_j < op_i$$

2. Asociativita (operátory se stejnou prioritou)

   Operátor i a operátor j jsou levě asociativní

   $$op_i > op_j \text{ a } op_j > op_i$$

   Operátor i a operátor j jsou pravě asociativní

   $$op_i < op_j \text{ a } op_j < op_i$$

3. Identifikátory

   Pokud $a \in T$ může být hned před id_i, potom $a < id_i$

   Pokud $a \in T$ může být hned za id_i, potom $id_i > a$

4. Závorky

   Pro jeden pár závorek platí $(=)$

   Nechť $a \in T - \{), \$\}$. Pak $(< a$

   Nechť $a \in T - \{(, \$\}$. Pak $a >)$

   Nechť $a \in T$ a a může být hned před (. Pak $a <$ (

   Nechť $a \in T$ a a může být hned za). Pak $) > a$

5. Znak konce řetězce

   Operátor i je libovolný operátor

   $$\$ < op_i \text{ a } op_i > \$$$

Nevyplněný zbytek tabulky zůstává prázdný a značí tak nepovolené kombinace ústící v syntaktickou chybu.

Kapitola 4

Kooperující distribuované gramatické systémy

V následující sekci jsou stručně popsány kooperující distribuované gramatické systémy (též *CDGS* z anglického *Cooperating distributed grammar system*). Obsah této sekce je převzat z prezentace [11].

4.1 Definice

Definice 4.1.1. Kooperující distribuovaný gramatický systém stupně n, $n \geq 1$, je konstrukce:

$$\Gamma = (N, T, S, P_1, \ldots, P_n)$$

kde:

- N je množina neterminálů
- T je množina terminálů
- S je počáteční neterminál
- P_i je konečná množina bezkontextových pravidel, nazývaná **komponentou** Γ, pro každé $i \in \{1, \ldots, n\}$

Notace.

- $G_i = (N, T, P_i, S)$ nazýváme *í*-tou gramatikou
- $y \not\Rightarrow z$ značí skutečnost, že neexistuje posloupnost derivačních kroků z řetězce y do řetězce z

Výše uvedená definice zavádí pojem *komponenta* gramatického systému. Na tyto komponenty je možné nahlížet jako na bezkontextové gramatiky. Komponenty ovšem sdílí množiny neterminálů, terminálů a mají jediný počáteční neterminál. Jednotlivé komponenty se tudíž odlišují pouze svými množinami pravidel.

Hlavní myšlenkou kooperujících distribuovaných gramatických systémů je spolupráce dílčích komponent na společném cíli. Společným cílem je zde myšlena výstupní věta jazyka. Spolupráce komponent probíhá sériově. Na počátky analýzy je vybrána vhodná komponenta, které je předána priorita. Zbylé komponenty vyčkávají v neaktivním stavu. Právě

aktivní komponenta provádí analýzu až do chvíle, kdy splní podmínku *derivačního módu*4.2. V tuto chvíli se zvolená komponenta stává neaktivní a priorita je přidělena následujícímu vhodnému kandidátovi.

4.2 Derivační módy

Derivační mód je hlavním prvkem komunikace komponent gramatického systému. V běžných CDGS je derivační mód sdílený celým systémem, tudíž se jím řídí všechny komponenty. Vedle CDGS dále existují například hybridní CDGS, jejichž předností je možnost výběru derivačního módu pro jednotlivé komponenty zvlášť. Tento mód udává podmínku práce komponenty. Základní derivační módy lze vyjádřit následující množinou:

$$D = \{*, t\} \cup \{\leq k, = k, \geq k : k = 1, 2, \ldots\}$$

Dále jsou popsány jednotlivé derivační módy. Jejich princip je také ukázán na příkladech.

Ukončující derivační mód

Definice 4.2.1. Pro každé $i = 1, \ldots, n$, je ukončující derivace i-tou komponentou

$$x \;{}_i \Rightarrow^t y$$

tehdy a jen tehdy

1. $x \Rightarrow^* y$ náleží do i-té gramatiky a
2. $y \not\Rightarrow z$ pro všechna $z \in (N \cup T)^*$

Ukončující derivační mód, z anglického *terminating mode*, je značen písmenem t. Přikazuje komponentám gramatického systému pracovat až do chvíle, kdy není možné provést žádnou další derivaci užitím pravidel aktivní komponenty. Tato funkcionalita je nejlépe vidět na příkladu:

Mějme kooperující distribuovaný gramatický systém:

$$\Gamma = (\{S, A\}, \{a\}, S, P_1, P_2, P_3)$$

Představený systém, jak je možné na první pohled vidět, obsahuje tři komponenty. Tyto komponenty budou průběžně adresovány svým číselným označením. Pravidla definující komponenty jsou následovná:

$$P_1 = \{S \to AA\}, P_2 = \{A \to S\}, P_3 = \{A \to a\}$$

Z pravidel je očividné, že první aktivní komponentou systému bude vždy komponenta 1. Ostatní komponenty neobsahují počáteční symbol. Za použití ukončujícího derivačního módu provede komponenta 1 pouze jeden derivační krok a ukončuje svoji práci. Následují komponenty 2 a 3 na které je dle potřeby vhodně přepnuto. Ty operují stejným způsobem jako komponenta 1. Výsledný jazyk generovaný tímto gramatickým systémem za použití ukončujícího módu je následovný:

$$L_t(\Gamma) = \{a^{2^n} : n \geq 1\}$$

k derivačních kroků v G_i

Definice 4.2.2. Pro každé $i = 1, \ldots, n$, je k-tá derivace i-tou komponentou

$$x \ {}_i \Rightarrow^{=k} y$$

tehdy a jen tehdy, náleží-li $x \Rightarrow^k y$ do i-té gramatiky

Derivační mód k kroků je typicky značen symboly $= k$. K kroků umožňuje komponentám předat řízení pouze ve chvíli, kdy provedly přesně k derivací užitím svých pravidel. V případě užití počtu pravidel nerovnajícímu se k, dochází k chybě analýzy jazyka.

Maximálně k derivačních kroků v G_i

Definice 4.2.3. Pro každé $i = 1, \ldots, n$, je k-tá a nižší derivace i-tou komponentou

$$x \ {}_i \Rightarrow^{\leq k} y$$

tehdy a jen tehdy, náleží-li $x \Rightarrow^j y$ do i-té gramatiky pro nějaká $j \leq k$

Derivační mód maximálně k kroků je značen symboly $\leq k$. Maximálně k kroků umožňuje komponentám předat řízení pouze ve chvíli, kdy provedly k nebo méně derivací užitím svých pravidel. V případě užití více než k pravidel, dochází k chybě analýzy jazyka.

Minimálně k derivačních kroků v G_i

Definice 4.2.4. Pro každé $i = 1, \ldots, n$, je k-tá a vyšší derivace i-tou komponentou

$$x \ {}_i \Rightarrow^{\geq k} y$$

tehdy a jen tehdy, náleží-li $x \Rightarrow^j y$ do i-té gramatiky pro nějaká $j \geq k$

Derivační mód minimálně k kroků je značen symboly $\geq k$. Minimálně k kroků umožňuje komponentám předat řízení pouze ve chvíli, kdy provedly k a více derivací užitím svých pravidel. V případě užití méně než k pravidel, dochází k chybě analýzy jazyka.

Tři výše uvedené módy lze přiblížit následujícím příkladem. Mějme kooperující distribuovaný gramatický systém:

$$\Gamma = (\{S, A, A', B, B'\}, \{a, b, c\}, S, P_1, P_2)$$

Systém obsahuje tentokrát pouze dvě komponenty, které budou opět adresovány svým číselným označením. Množiny pravidel těchto komponent jsou definovány následovně:

$$P_1 = \{S \to S, S \to AB, A' \to A, B' \to B\}$$

$$P_2 = \{A \to aA'b, B \to B'c, A \to ab, B \to c\}$$

První, do očí bijící, abnormalita ve výše představených pravidlech je první pravidlo komponenty 1 $S \to S$. Na první pohled se tento derivační krok může zdát zbytečný. Existuje z jediného důvodu, a to pro možné použití derivačního módu $= k$, kde $k = 2$, popřípadě v tomto příkladě ekvivalentnímu $\geq k$, kde $k = 2$.

Při použití těchto derivačních módů dosáhneme následujícího. První aktivní jednotkou se stane komponenta 1. V případě, že by komponenta jako první využila svého druhého

pravidla, $S \rightarrow AB$, nebyla by schopna vykonat druhou, požadovanou, derivaci. Přichází tedy na scénu zmíněné první pravidlo, které zaručí úspěšné využití dvou derivačních kroků.

Následuje přepnutí na komponentu 2. Ta již má na výběr, rozhodne tedy dle analyzované vstupní věty, jaká pravidla využije a zdali opět přenechá prioritu komponentě 1. Tento proces proběhne, v závislosti na délce vstupní věty, n-krát.

Výsledný jazyk generovaný uvedeným systémem za použití derivačního módu $= k$ nebo $\geq k$, kde se v obou případech $k = 2$, je následující:

$$L_{=2}(\Gamma) = L_{\geq 2}(\Gamma) = \{a^n b^n c^n : n \geq 1\}$$

V případě, kdy je $k \geq 3$, je generovaný jazyk prázdnou množinou. Obdobnou analýzou průchodu můžeme zjistit, že systém není schopen za žádných okolností vykonat v druhém kroku, druhou komponentou tři a více derivačních kroků.

Derivační mód $*$

Derivační mód $*$ je značen symbolem $*$. Tento derivační mód dovoluje komponentám gramatického systému pracovat tak dlouho, jak samy uznají za vhodné. V libovolnou chvíli mohou předat řízení příhodným komponentám pro pokračování v analýze.

4.3 Generovaný jazyk

Jak již bylo dříve naznačeno, zápis generovaného jazyka pomocí CDGS má formu $L_f(\Gamma), f \in D$. K jeho formálnímu, úplnému popisu je však zapotřebí nejdříve definovat tzv. *množinu možných derivací*:

$$F(G_j, u, f) = \{v : u \ _j \Rightarrow^f v\}, \ kde \ j \in \{1, \ldots, n\}, f \in D, u \in (N \cup T)^*$$

Jedná se o množinu všech možných podob větné skladby po provedení posloupnosti derivačních kroků, dle derivačního módu f v gramatice G_j nad řetězcem u. Díky této definici můžeme následovně formálně popsat generovaný jazyk:

$$\begin{aligned} L_f(\Gamma) = \{ \quad & w \in T^* : v_0, v_1, ..., v_m \text{ takové, že} \\ & v_i \in F(G_{j_i}, v_{i-1}, f), i = 1, ..., m, j_i \in \{1, ..., n\}, \\ & v_0 = S, v_m = w, \text{ pro nějaká } m \geq 1 \ \} \end{aligned}$$

V zápisu výše se objevují části analýzy $v_0, \ldots, v_m$. Tyto části lze chápat jako stavy generované věty mezi přepínáním komponent. Před první aktivací komponenty je stav věty S, neboli počáteční neterminál. Následuje posloupnost polotovarů věty, které náleží patřičným množinám možných derivací. V případě úspěšné analýzy je věta po poslední deaktivaci komponenty složena z terminálů a je výsledkem práce gramatického systému. Tato věta spadá do generovaného jazyka.

4.4 Generativní síla

V otázce generativní síly gramatických systémů je vhodno si nejdříve zavést hierarchii rodin jazyků, vůči kterým můžeme generativní sílu porovnávat, a patřičný zápis. Jako první si představíme Chomského hierarchii rodin jazyků [11]:

$$REG \subset LIN \subset CF \subset CS \subset RE \subset ALL$$

kde:

- REG je rodina regulárních jazyků
- LIN je rodina lineárních jazyků
- CF je rodina bezkontextových jazyků
- CS je rodina kontextových jazyků
- RE je rodina rekurzivně spočitatelných jazyků
- ALL je rodina všech jazyků

Dále je zapotřebí uvést notaci rodin CD jazyků. Následující notace byla inspirována notací [11]. Rodina CD jazyků je značena jako $CD_x^y(f)$, kde $f \in D$ je použitý derivační mód. Dále y značí použití ε pravidel. Hlavním znakem je x, které udává maximální počet komponent gramatického systému. Následující relace popisují generativní sílu CDGS [11]:

- $CD_\infty^y(f) = CF$, pro všechna $f \in \{= 1, \geq 1, *\} \cup \{\leq k \ : \ k \geq 1\}$
- $CF = CD_1^y(f) \subset CD_2^y(f) \subseteq CD_r^y(f) \subseteq CD_\infty^y(f)$, pro všechna $f \in \{= k, \geq k \ : \ k \geq 2\}, r \geq 3$
- $CD_r^y(\geq k) \subseteq CD_r^y(\geq k+1)$
- $CD_\infty^y(\geq) \subseteq CD_\infty^y(=)$
- $CF = CD_1^y(t) = CD_2^y(t) \subset CD_3^y(t) = CD_\infty^y(t)$

Kapitola 5

Zavedení nové metody konstrukce

V této kapitole se budeme podrobně věnovat návrhu systému pro komunikaci gramatik. V začátcích pojednáváme o základním principu, ukázaném na jednoduchém příkladu a formální definici konstrukce. Následně popisujeme komunikační protokol a v závěru ukazujeme generovaný jazyk. Kapitola je zakončena příkladem ukazujícím konkrétní použití této konstrukce.

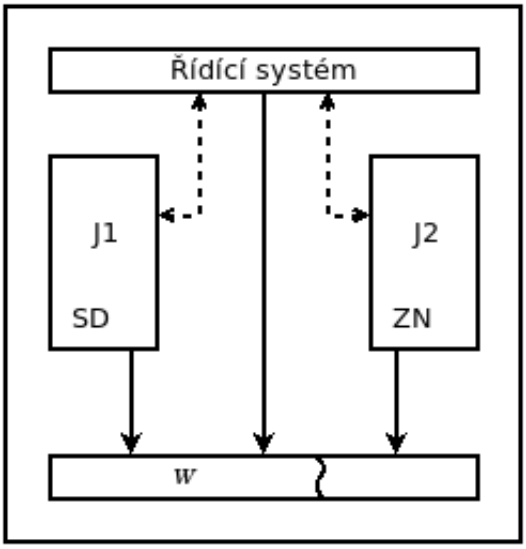
5.1 Princip

Hlavní myšlenkou návrhu je vytvoření a formální definice konstrukce, v rámci které budou schopny spolu komunikovat dílčí jednotky kompilátoru. Ty operují výhradně na základě bezkontextových gramatik. Každá jednotka postupuje dle jedné z používaných metod uvedených v sekci 3.

Představme si, pro ilustraci, následující situaci. Jednotka č. 1 kompilátoru (dále pro jednoduchost *jednotka 1*) generuje blíže nespecifikovaný jazyk metodou rekurzivního sestupu (shora dolů). Ovšem po vykonání sekvence derivačních kroků narazí na část jazyka, kterou není schopna vygenerovat. Jednotka 1 se tedy tímto dostává do koncového stavu a na scénu přichází jednotka č. 2 kompilátoru (dále pro jednoduchost *jednotka 2*). Ta, na rozdíl od jednotky 1, provádí precedenční analýzu (zdola nahoru) a je, shodou okolností, vhodným kandidátem pro pokračování v generování našeho smyšleného jazyka.

Díky použití námi navržené konstrukce si je jednotka 1 vědoma této skutečnosti. Vygenerovala tudíž v průběhu analýzy značku pro signalizaci počátku analýzy jednotky 2, kterou jí poskytl řídící systém. Při přechodu jednotky 1 do koncového stavu je řídící systém uvědoměn, ten následně prohlédne generovaný řetězec a v případě nalezení počáteční značky předá řízení jednotce, které značka náleží. V ilustračním příkladě tedy našel signalizaci pro předání řízení jednotce 2. Jednotka 2 následně pokračuje precedenční analýzou, během které je schopna analogicky předat řízení zpět jednotce 1.

Řídící systém, jak je uvedeno v krátké ukázce, je určen ke sjednocení samostatných gramatických systémů, zajištění nutných prostředků pro komunikaci a dohledu nad rozdělením práce.



Obrázek 5.1: Blokové schéma parseru kompilátoru

5.2 Definice konstrukce řídícího systému

V této sekci si představíme a blíže popíšeme formální definici řídícího systému pro komunikaci mezi jednotkami kompilátoru.

Návrh je založen, jak bylo již dříve avizováno, na kooperujících distribuovaných gramatických systémech (dále pouze CDGS) [11]. Tyto gramatické systémy byly vybrány z důvodu blízké vnitřní podobnosti námi řešeného problému. Popis následující konstrukce vychází z Γ CDGS 4.1.

Systém komunikujících gramatických systémů značíme Γ. Jedná se o konstrukci:

$$\Gamma = (\Lambda, G_1, \ldots, G_n),\ pro\ n \geq 1$$

Γ obsahující $G_1, \ldots, G_n$ též nazýváme systémem komunikujících gramatických systémů *stupně n*. Pro ten platí, že $G_1, \ldots, G_n$ jsou bezkontextové gramatiky, které též nazýváme *komponenty* Γ. G_i, také označujeme jako *i-tá gramatika*, je daná předpisem $G_i = (T_i, N_i, P_i, S_i)$. Dle těchto gramatik se následně řídí jednotlivé části kompilátoru během překladu vstupního jazyka.

Mimo samotné komponenty obsahuje řídící systém množinu Λ.

$$\Lambda = \{S_1, ..., S_n\}$$

Ta obsahuje, za účelem umožnění jednoduché komunikace, počáteční neterminály $S_1, \ldots, S_n$ gramatik řídícího systému, využívané jako "signalizační značky"pro určení počátku analýzy právě probuzené jednotky. Pro zachování determinismu je zapotřebí, aby Λ neobsahovala duplikátní prvky. Tedy je třeba zachovat unikátnost počátečních neterminálů gramatik.

Předpis konstrukce hlavního systému z ilustračního příkladu sekce 5.1 by vypadal následovně:

$$\Gamma = (\{S_1, S_2\}, G_1, G_2)$$

Jednotka 1, pracující metodou shora dolů, využívá k překladu gramatiky G_1 s počátečním neterminálem S_1. Jednotka 2, pracující metodou zdola nahoru, obdobně G_2 / S_2.

Celkové shrnutí úpravy konstrukce CDGS pro aplikaci při konstrukci kompilátoru lze vyjádřit následující definicí:

Definice 5.2.1. Řídící systém jednotek kompilátoru stupně $n, n \geq 1$ je konstrukce:

$$\Gamma = (\Lambda, G_1, \ldots, G_n), \ pro\ n \geq 1$$

kde:

- $\Lambda = \{S_1, ..., S_n\}$ je množina počátečních neterminálů gramatik
- $G_1, \ldots, G_n$ pro $n \geq 1$ jsou gramatiky jednotek kompilátoru, též nazývané jako komponenty Γ

5.3 Derivační módy

V této sekci si představíme hlavní prvek komunikace jednotlivých částí kompilátoru. Derivační mód, jak byl převzán z CDGS, je formou omezení generování výstupního jazyka. Specifikuje, jak lze provádět derivační kroky v rámci gramatikého systému.

Vzhledem k obecnosti definicí derivačních módu nebylo zapotřebí změn, byly tedy převzaty v následující podobě z CDGS.

Ukončující derivační mód

Pro každé $i = 1, \ldots, n$, je ukončující derivace *i*-tou komponentou

$$x \ _i \Rightarrow^t y$$

tehdy a jen tehdy

1. $x \Rightarrow^* y$ náleží do *i*-té gramatiky a
2. $y \not\Rightarrow z$ pro všechna $z \in (N \cup T)^*$

k derivačních kroků v G_i

$$x \ _i \Rightarrow^{=k} y$$

tehdy a jen tehdy, náleží-li $x \Rightarrow^k y$ do *i*-té gramatiky

Maximálně k derivačních kroků v G_i

$$x \ _i \Rightarrow^{\leq k} y$$

tehdy a jen tehdy, náleží-li $x \Rightarrow^j y$ do *i*-té gramatiky pro nějaká $j \leq k$

Minimálně k derivačních kroků v G_i

$$x \ _i \Rightarrow^{\geq k} y$$

tehdy a jen tehdy, náleží-li $x \Rightarrow^j y$ do *i*-té gramatiky pro nějaká $j \geq k$

Množina derivačních módů

$$D = \{*, t\} \cup \{\leq k, = k, \geq k : k = 1, 2, \ldots\}$$

5.4 Komunikační protokol

Dále je blíže popsán protokol, specifikující komunikaci komponent řídícího systému Γ, popsaného v sekci 5.2. Je zde definováno, kdy a za jakých podmínek ke komunikaci dochází. Následně je ukázáno, jakým způsobem je určeno, která jednotka kompilátoru má být probuzena a které je předáno řízení.

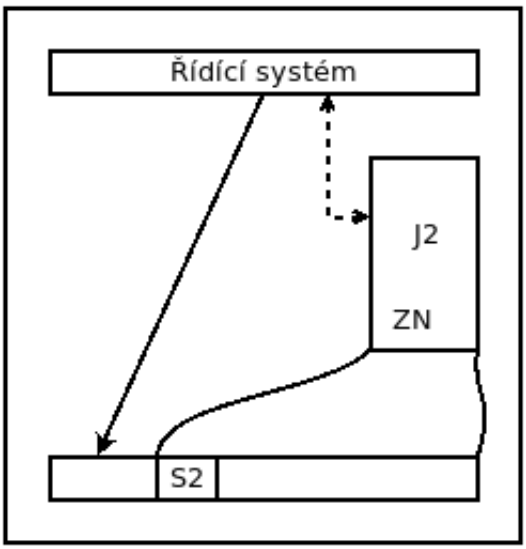
Kompilátor se dle navržené konstrukce skládá z více menších částí, zodpovědných za překlad jim určené části jazyka. Vzhledem k faktu, že tyto dílčí jednotky pracují na základě libovolné metody a gramatiky, nemusí být schopny zpracovat kompletní vstupní řetězec slov. Z tohoto důvodu bylo třeba navrhnout protokol, který nám umožní komunikaci mezi částmi překladače a nadřazeným řídícím systémem. Konkrétně se jedná o akce signalizace ukončení analýzy ze strany jednotky a následné probuzení jednotky řídícím systémem.

Akce signalizace ukončení analýzy

Jak již bylo zmíněno, tento signál přichází řídícímu systému ze strany jednotky. Ta tímto signálem dává najevo, že dokončila svoji část analýzy, tedy dosáhla jednoho z koncových stavů, a je připravena předat řízení.

Řečený koncový stav se určuje dle tzv. *derivačního módu*, popsaného v sekci 5.3. Derivační mód ze značné části určuje podobu výsledného jazyka, neboť ovlivňuje právě rozsah analýzy jednotlivých gramatik.

Samotný signál ale ke správnému předání řízení nestačí. Jestliže si je jednotka kompilátoru vědoma, že se v analyzované části nachází úsek jazyka jí nenáležící, vygeneruje na daném místě počáteční neterminál příslušné gramatiky. Tím je zajištěno předání řízení správné jednotce.

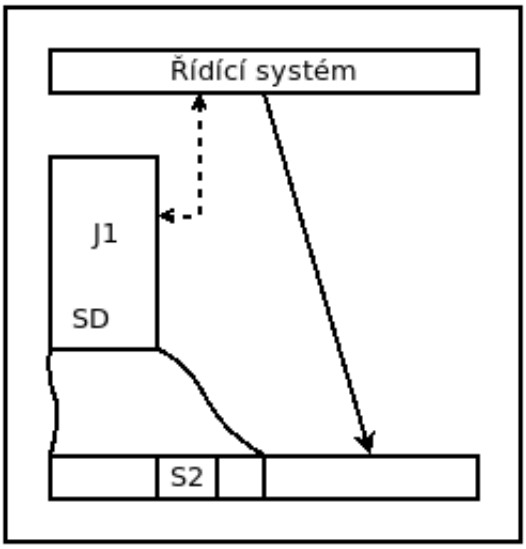


Obrázek 5.2: Ilustrace situace ukončení analýzy jednotkou 1

Akce probuzení jednotky

Jedná se o signál řídícího systému jednotce. Signál s sebou vždy nese informaci o počátečním bodě analýzy.

Před vysláním signálu probuzení provede řídící systém průzkum vygenerované pásky symbolů. Průzkum je prováděn zleva doprava, tedy od začátku vstupního jazyka ke konci. V případě, že je během tohoto skenování nalezen počáteční neterminál některé z komponent, je probíhající prohledávání ukončeno a je vyslán signál probuzení s informací o pozici nalezeného symbolu.



Obrázek 5.3: Ilustrace situace počátku analýzy jednotkou 2

5.5 Generovaný jazyk

Tato sekce podrobně popisuje výsledný generovaný jazyk pomocí kompilátoru konstruovaného metodou představenou výše. Je předpokládáno, že generativní síla této metody konstrukce je rovna generativní síle gramatických systémů CD.

V první části sekce si popíšeme pomocnou množinu potřebnou pro jednodušší zápis výsledného jazyka. Následuje popis jednotlivých částí zápisu a nakonec ucelená formule.

Množina možných derivací

Tato množina nám pomáhá definovat platné sekvence derivačních kroků v rámci jedné jednotky kompilátoru. Množina možných derivací je závislá na třech vstupních proměnných: gramatice G_j, počátečnímu stavu pásky symbolů u a derivačnímu módu f. Následující klauzule představuje obecný zápis této množiny pro použití v námi navržené metodě konstrukce.

$$F(G_j, u, f) = \{v : u \Rightarrow_j^f v\}, j \in \{1, ..., n\}, f \in D, u \in (T^* S_j T^*)^*, v \in ((T^* S T^*)^* \cup T^*), S \in \Lambda$$

Hlavní úpravou oproti CDGS jsou strany derivačního pravidla. Levá strana, u, je posloupností terminálů a počátečních neterminálů gramatik. Ta může přeměněna na posloupnost terminálů a počátečních neterminálů gramatik nebo na posloupnost pouze terminálů.

Jazyk

Výsledný jazyk w je posloupností terminálů jednotlivých komponent řídícího systému. Postup generování lze rozdělit do kroků, symbolizujících provádění analýzy dílčími jednotkami.

Každá část kompilátoru přijme pásku symbolů a dle své gramatiky jí pozmění, čímž vytvoří zmíněný krok. Tyto kroky značíme jako v.

$$w \in T^* : v_0, v_1, ..., v_m$$

Tyto kroky musí splňovat podmínky provedení derivace, specifikované výše. Tudíž musí náležet do množiny možných derivací.

$$v_i \in F(G_{j_i}, v_{i-1}, f), i = 1, ..., m, j_i \in \{1, ..., n\},$$

První krok při generování jazyka, v_0, je vždy stejný. Musí být roven počátečnímu neterminálu jedné z gramatik, aby bylo možno započít analýzu. Konečný jazyk je výsledkem zpracování poslední jednotky, tedy kroku v_m.

$$v_0 = S, S \in \Lambda, v_m = w, \text{ pro nějaká } m \geq 1 \ \}$$

Ucelená formule

Zde je uvedena ucelená formule generovaného jazyka.

$$\begin{aligned} L_f(\Gamma) = \{ \quad & w \in T^* : v_0, v_1, ..., v_m \text{ takové, že} \\ & v_i \in F(G_{j_i}, v_{i-1}, f), i = 1, ..., m, j_i \in \{1, ..., n\}, \\ & v_0 = S, S \in \Lambda, v_m = w, \text{ pro nějaká } m \geq 1 \ \} \end{aligned}$$

Kapitola 6

Ukázkový příklad

V této kapitole si předvedeme praktický příklad možného využití nové metody konstrukce kompilátoru představené a definované v předchozí kapitole. Jedná se o jazyk IFJ19, syntaxí podobný široce rozšířenému Python. Bližší specifikace vstupního jazyka je dostupná zde [2]. Příklad je zaměřen na oblasti jazyka, ve kterých je nutno přepnout mezi jednotkami, které pracují rozdílnými metodami. Konkrétně se v rámci IFJ19 jedná o struktury obsahující výrazy, jež jsou analyzovány metodou zdola nahoru.

6.1 Gramatiky ukázkového příkladu

Následující sekce obsahuje definice řídícího systému a gramatiky jednotlivých částí kompilátoru. Gramatika G_0 slouží k analýze výrazů a pracuje metodou zdola nahoru. Ostatní, tedy $G_1, \ldots, G_8$ jsou gramatiky pro zpracování zbytku jazyka, pracující metodou shora dolů. V případě použití ukončujícího derivačního módu dostáváme pomocí následujícího systému jednotek jazyk IFJ19.

$$L_t(\Gamma) \approx IFJ19$$

$$
\begin{aligned}
\Gamma = &\quad (\{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8\}, G_0, \ldots, G_8) \\
G_0 = &\quad (\{S_0\}, \{ID, int, float, None, +, -, *, /, //, (,), <, >, <=, >=, ==, !=\}, P_0, S_0) \\
G_1 = &\quad (\{S_1, LINE\}, \{EOF, ID, def, if, while, pass, EOL\}, P_1, S_1) \\
G_2 = &\quad (\{S_2\}, \{=, EOL, (\}, P_2, S_2) \\
G_3 = &\quad (\{S_3, ARG\ REP, ARG\}, \{), \ , \ , ID, int, float, str\}, P_3, S_3) \\
G_4 = &\quad (\{S_4, FCTION\ END, RET\ VAL\}, \{ID, (, :, EOL, INDENT, DEDENT, return\}, P_4, S_4) \\
G_5 = &\quad (\{S_5, ARG\ REP, ARG\}, \{), \ , \ , ID\}, P_5, S_5) \\
G_6 = &\quad (\{S_6, LINE\}, \{ID, if, while, pass, EOL\}, P_6, S_6) \\
G_7 = &\quad (\{S_7\}, \{:, EOL, INDENT, DEDENT, else\}, P_7, S_7) \\
G_8 = &\quad (\{S_8\}, \{:, EOL, INDENT, DEDENT\}, P_8, S_8)
\end{aligned}
$$

P0

1	$< S0 >$	$\rightarrow$	ID
2	$< S0 >$	$\rightarrow$	int
3	$< S0 >$	$\rightarrow$	float
4	$< S0 >$	$\rightarrow$	None
5	$< S0 >$	$\rightarrow$	$< S0 > + < S0 >$
6	$< S0 >$	$\rightarrow$	$< S0 > - < S0 >$
7	$< S0 >$	$\rightarrow$	$< S0 > * < S0 >$
8	$< S0 >$	$\rightarrow$	$< S0 > / < S0 >$
9	$< S0 >$	$\rightarrow$	$< S0 > // < S0 >$
10	$< S0 >$	$\rightarrow$	$(< S0 >)$
11	$< S0 >$	$\rightarrow$	$< S0 > < < S0 >$
12	$< S0 >$	$\rightarrow$	$< S0 > > < S0 >$
13	$< S0 >$	$\rightarrow$	$< S0 > <= < S0 >$
14	$< S0 >$	$\rightarrow$	$< S0 > >= < S0 >$
15	$< S0 >$	$\rightarrow$	$< S0 > == < S0 >$
16	$< S0 >$	$\rightarrow$	$< S0 > != < S0 >$

P1

1	$< S1 >$	$\rightarrow$	$< LINE > < S1 >$
2	$< S1 >$	$\rightarrow$	$< LINE >$
3	$< S1 >$	$\rightarrow$	EOF
4	$< LINE >$	$\rightarrow$	ID $< S2 >$
5	$< LINE >$	$\rightarrow$	def $< S4 >$
6	$< LINE >$	$\rightarrow$	if $< S7 >$
7	$< LINE >$	$\rightarrow$	while $< S8 >$
8	$< LINE >$	$\rightarrow$	pass EOL
9	$< LINE >$	$\rightarrow$	$< S0 >$ EOL

P2

1	$< S2 >$	$\rightarrow$	$= < S0 >$ EOL
2	$< S2 >$	$\rightarrow$	$(< S3 >$ EOL

P3

1	$< S3 >$	$\rightarrow$	)
2	$< S3 >$	$\rightarrow$	$< ARG > < ARG\ REP >$
3	$< ARG\ REP >$	$\rightarrow$	)
4	$< ARG\ REP >$	$\rightarrow$	$, < ARG > < ARG\ REP >$
5	$< ARG >$	$\rightarrow$	ID
6	$< ARG >$	$\rightarrow$	int
7	$< ARG >$	$\rightarrow$	float
8	$< ARG >$	$\rightarrow$	str

P4

1	$< S4 >$	$\rightarrow$ ID ($< S5 >$: EOL INDENT $< S6 > < FCTION\ END >$
2	$< FCTION\ END >$	$\rightarrow$ DEDENT
3	$< FCTION\ END >$	$\rightarrow$ return $< RET\ VAL >$
4	$< RET\ VAL >$	$\rightarrow$ $< S0 >$ EOL DEDENT
5	$< RET\ VAL >$	$\rightarrow$ EOL DEDENT

P5

1	$< S5 >$	$\rightarrow$)
2	$< S5 >$	$\rightarrow$ $< ARG > < ARG\ REP >$
3	$< ARG\ REP >$	$\rightarrow$)
4	$< ARG\ REP >$	$\rightarrow$, $< ARG > < ARG\ REP >$
5	$< ARG >$	$\rightarrow$ ID

P6

1	$< S6 >$	$\rightarrow$ $< LINE > < S6 >$
2	$< S6 >$	$\rightarrow$ $< LINE >$
3	$< LINE >$	$\rightarrow$ ID $< S2 >$
4	$< LINE >$	$\rightarrow$ if $< S7 >$
5	$< LINE >$	$\rightarrow$ while $< S8 >$
6	$< LINE >$	$\rightarrow$ pass EOL
7	$< LINE >$	$\rightarrow$ $< S0 >$ EOL

P7

1	$< S7 >$	$\rightarrow$ $< S0 >$: EOL INDENT $< S6 >$ DEDENT else : EOL INDENT $< S6 >$ DEDENT

P8

1	$< S8 >$	$\rightarrow$ $< S0 >$: EOL INDENT $< S6 >$ DEDENT

6.2 Překlad jazyka

Následuje praktický příklad překladu jazyka kompilátorem konstruovaným navrženou metodou. Příklad bude využívat následujícího úseku kódu.

```
if ((a + b) * 25) <= 10:
    a = a + 1
    foo()
else:
    print("Hello World!")
```

Překlad vždy začíná s počátečním neterminálem `<S1>`. Jednotka č. 1, využívající gramatiky G_1, tento symbol přepíše neterminálem `<LINE>` pomocí pravidla 2. Následně poté za pomoci pravidla 6 nahrazuje posloupností:

```
if <S7>
```

Touto derivací se jednotka 1 dostává do koncového stavu a odevzdává řízení. Řídící systém skenuje pásku symbolů a nachází neterminál `<S7>`. Probouzí tímto jednotku č. 7, která pokračuje v analýze kódu.

Jednotka č. 7 vygeneruje následující řetězec za pomoci pravidla 1 své gramatiky.

```
if <S0>:
    <S6>
else:
    <S6>
```

Dostává se do koncového stavu a odevzdává řízení. Řídící systém opět prochází vygenerovaný řetězec a naráží na počáteční neterminál `<S0>`. Jednotka č. 0, které `<S0>` náleží, zpracovává výrazy, na rozdíl od předchozích jednotek, metodou zhora dolů. Jednotka výraz zpracuje, v průběhu ovšem negeneruje žádný počáteční neterminál jiné gramatiky.

```
if ((a + b) * 25) <= 10:
    <S6>
else:
    <S6>
```

Řídící systém tedy po obdržení signálu ukončení analýzy prochází pásku a naráží na počáteční neterminál vygenerovaný v předchozích krocích. Postup se identicky opakuje i pro jednotku č. 6. Ta provádí analýzu, při které opět generuje další neterminály jiných gramatik.

```
if ((a + b) * 25) <= 10:
    <LINE> <S6>
else:
    print<S2>
```

```
if ((a + b) * 25) <= 10:
    a<S2> <S6>
else:
    print(<S3>
```

```
if ((a + b) * 25) <= 10:
    a = <S0>
    <S6>
else:
    print(<ARG><ARG REP>
```

```
if ((a + b) * 25) <= 10:
    a = a + 1
    foo()
else:
    print("Hello World!")
```

V tuto chvíli řídící systém nenachází žádný počáteční neterminál na pásce symbolů, analýza je tímto úspěšně ukončena.

Kapitola 7

Implementace

Tato kapitola popisuje implementaci syntaktického analyzátoru (dále pouze *parser*) metodou navrženou v kapitole 5. Nejdříve bude popsána architektura parseru a implementované jednotky. V další části následuje popis vstupu, výstupu a s nimi spojeným generátorem tokenů (dále adresován jako *scanner*). V poslední části bude představený přijímaný jazyk a jeho rozšíření.

7.1 Použitá technologie

Praktická část práce byla napsána v programovacím jazyce C++. Při implementaci byly použity pouze standardní knihovny jazyka.

Projekt obsahuje Makefile pro jeho sestavení. Podporovány jsou makra `clean` pro odstranění vygenerovaných souborů a `pack` pro zabalení a komprimaci projektu do souboru formátu *zip*.

7.2 Architektura parseru

Následující sekce představuje architekturu implementovaného parseru. První část se zaměřuje na řídící systém, zprostředkovávající pomocná data a struktury pro jednotky. V další části je uvedena realizace základní jednotky, tedy rodičovské třídy používaných jednotek a v neposlední řadě je popsán postup v případě chyby analýzy.

Řídící systém

Hlavním stavebním kamenem kompilátoru, konstruovaného metodou navrženou v kapitole 5, je řídící systém. Umožňuje nejen předávání řízení mezi jednotkami, ale i zpřístupňuje komunikaci se scannerem a napomáhá při zotavení parseru ze stavu nalezení syntaktické chyby.

Řídící systém je realizován formou neměnných atributů jednotek. Při spuštění parseru je provedeno vytvoření a následná inicializace, při které je řídící systém jednotkám předán.

Obsahem řídícího systému jsou:

1. vektor odkazů na zavedené jednotky, sloužící pro předávání komunikace,
2. odkaz na modul scanneru, pomocí kterého mohou jednotky získávat vstupní data,
3. odkaz na sdílený prvek signalizující chybový stav parseru.

General unit

General unit je označení pro rodičovskou třídu všech zavedených jednotek. Tato třída implementuje společné atributy a funkce. Patří mezi ně například data řídícího systému, funkce pro inicializaci, funkce pro výpis výstupních dat nebo sada funkcí pro zpracování dat na vstupu.

Důležitou zavedenou funkcí je funkce `run()`. Ta je každou dílčí jednotkou, která z *general unit* dědí, přepsána a představuje její počáteční neterminál. Při předávání řízení je tudíž vždy volána právě tato metoda.

Dalšími metodami jsou `process_ID()`, `process_keyword(string kw)` a `process_special(bttoken_type_t token_type)`. Jedná se o sadu funkcí zpracovávající vstupní data poskytnutá scannerem. Samy tyto funkce obsahují mechanismus pro zotavení kompilátoru z chybového stavu, díky čemu není třeba dodatečného kódu v samotných dílčích jednotkách. Jejich jména napovídají účel:

- `process_ID()` zpracovává následující identifikátor,
- `process_keyword(string kw)` kontroluje následující klíčové slovo a porovnává jej s parametrem `kw`
- `process_special(bttoken_type_t token_type)` porovnává typ tokenu (7.3) příchozích dat.

Neméně důležitou funkcí, sdílenou jednotkami, je `set_err()`. Jedná se metodu, která přepíná systém jednotek do chybového stavu. Dochází k jejímu spuštění v případě nalezení syntaktické chyby vstupních dat. Chybový stav je blíže popsán dále v této sekci.

Zavedené jednotky

V parseru bylo implementováno celkem 17 jednotek. Tyto jednotky jsou rozlišovány číselným označením. Jednotky 1–16 využívají při analýze metodu shora dolů. Každá má přiřazenou patřičnou podmnožinu jazyka, typicky se jedná o jazykovou konstrukci, specifickou část příkazu nebo celý blok. Jednotky jsou navrženy jako LL(4) analyzátory a pracují na základě algoritmu rekurzivního sestupu, popsaného v sekci 3.1.

Jednotka 0 je navržena pro analýzu výrazů jazyka. Pracuje metodou zdola nahoru. Implementace byla provedena metodou precedenční analýzy, blíže popsané v sekci 3.2.

Mezi členy systému je přepínáno dle derivačního módu *. To znamená, že k přepínání mezi jednotkami dochází ve chvíli, kdy se aktivní jednotka sama rozhodne předat řízení jiné. K přepnutí dochází právě tehdy, kdy je vygenerován počáteční neterminál jedné z jednotek.

Samotné předávání řízení je implementováno jako volání funkcí, symbolizujících počáteční neterminál, jednotek poskytnutých řídícím systémem. Dochází tímto k rekurzivnímu zanořování i samotných jednotek.

Chybový stav

V případě nalezení syntakticky nesprávné konstrukce vstupního jazyka přechází celý systém do chybového stavu. Vzhledem k architektuře parseru nemusí být vždy schopna samotná jednotka, která chybu objevila, tento problém vyřešit. Bylo tedy nutné zavést tento stav globálně za pomoci řídícího systému.

Ve chvíli, kdy je nalezena syntaktická chyba, aktivní jednotka nastavuje chybový příznak. Následně provede odebrání posloupnosti příchozích dat až do prvního výskytu symbolu konce řádku. Od této chvíle pokračují jednotky v analýze s rozdílem, že vstupní data pro analýzu nejsou nadále kontrolována ani odebírána od scanneru až do chvíle, kdy poprvé jedna z jednotek zažádá o zpracování symbolu konce řádku. V této chvíli se systém zotavuje a vrací zpět do normálního režimu.

7.3 Vstup a výstup

Tato sekce popisuje vstup a výstup implementovaného parseru. První část popisuje, jak a v jakém formátu jsou poskytována vstupní data. V druhé části je popsán výstup parseru spolu s krátkým příkladem.

Vstup parseru

Modul syntaktické analýzy kompilátoru typicky přijímá data od jednotky lexikální analýzy. Tento koncept je v případě této práce zachován. Parser při své inicializaci očekává poskytnutí odkazu na strukturu, která bude schopna dle stanovených kritérií poskytovat dané služby. Data jsou přijímána jako tzv. *tokeny*.

Token obsahuje data popisující konkrétní lexém. Pro dostatečné popsání lexému bylo zapotřebí v tokenu nést informaci o typu a hodnotě.

Typ tokenu udává základní charakteristiku, např. `EOL` (značící přicházející znak konce řádku). Tento typ tokenu je jednoznačný, tudíž nenese žádnou informaci o hodnotě. Mezi nejednoznačné typy patří například `KEYWORD`. Tyto tokeny s sebou vždy nesou hodnotu potřebnou k jednozačné specifikaci popisovaného lexému.

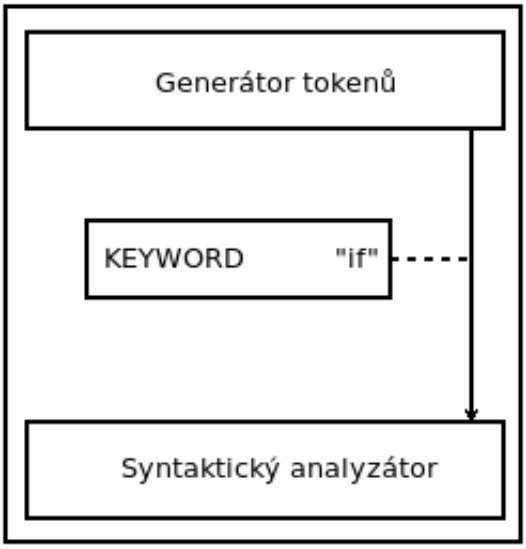
Typ tokenu	Typ hodnoty	Očekávaná hodnota
BTTG_KEYWORD	řetězec	klíčové slovo jazyka
BTTG_ID	řetězec	identifikátor
BTTG_STR	řetězec	řetězec znaků
BTTG_INT	celé číslo	celé číslo
BTTG_FLOAT	desetinné číslo	desetinné číslo

Tabulka 7.1: Nejednoznačné typy tokenů

Typ tokenu	Význam
BTTG_ASSIGN	=
BTTG_PAL	(
BTTG_PAR	)
BTTG_BRL	[
BTTG_BRR	]
BTTG_BCL	{
BTTG_BCR	}
BTTG_COMMA	,
BTTG_COLON	:
BTTG_DOT	.
BTTG_INDENT	odsazení
BTTG_DEDENT	zrušení odsazení
BTTG_EOL	znak konce řádku
BTTG_EOF	znak konce souboru
BTTG_EXP_PLUS	+
BTTG_EXP_MINUS	-
BTTG_EXP_MUL	*
BTTG_EXP_DIV	/
BTTG_EXP_DIVF	//
BTTG_EXP_EQ	==
BTTG_EXP_NEQ	!=
BTTG_EXP_LT	<
BTTG_EXP_GT	>
BTTG_EXP_LET	<=
BTTG_EXP_GET	>=

Tabulka 7.2: Jednoznačné typy tokenů

Z posloupnosti tokenů je poskytován ke zpracování vždy pouze první, nahlédnout je však v průběhu analýzy možno libovolný počet vpřed. Tato funkcionalita byla zavedena pro možnost navržení a zkonstruování LL(k) analyzátorů v jednotkách 1–16.



Obrázek 7.1: Vstupní data parseru

Výstup parseru

Jako výstup programu byla zvolena posloupnost použitých pravidel k analýze vstupní části jazyka. Program při využití pravidla gramatiky vytiskne na standardní výstup textové informace ve formátu:

```
Gramatika   Číslo pravidla  Pravidlo
```

Gramatiky jsou značeny písmenem `G` a odlišovány čísly patřičných jednotek, které je využívají. Pořadová čísla pravidel se vztahují vždy pouze na gramatiku, do které dané pravidlo náleží. Mohou se tedy ve výpisu vyskytovat stejná pořadová čísla s rozdílnými pravidly.

Následuje příklad výstupu programu pro triviální konstrukci if/else. V ukázce se nachází posloupnosti terminálů a neterminálů v použitých pravidlech. Ty je možné od sebe rozlišit pomocí ohraničení <>. Jeli název takto označen, jedná se o neterminál patřičné gramatiky.

```
if 1 == 1:
    pass
else:
    pass
```

```
G1:     1   <S1>        ->  <LINE> <S1>
G1:     5   <LINE>      ->  if <S7>
G7:     1   <S7>        ->  <S0> : EOL INDENT <S16> else : EOL INDENT <S16>
G0:     2   <S0>        ->  int
G0:     2   <S0>        ->  int
G0:     16  <S0>        ->  <S0> == <S0>
G16:    1   <S16>       ->  <LINE> <S16>
G16:    7   <LINE>      ->  pass EOL
G16:    2   <S16>       ->  DEDENT
G16:    1   <S16>       ->  <LINE> <S16>
G16:    7   <LINE>      ->  pass EOL
```

```
G16:    2   <S16>       ->  DEDENT
G1:     2   <S1>        ->  EOF
```

V příkladu si lze všimnout, že pravidla jsou vypisována v pořadí daném metodou analýzy jednotky. Jelikož jednotka 0, využívající gramatiku G0, pracuje metodou zdola nahoru, jsou i její pravidla vypisována v patřičném pořadí. Naopak je tomu u zbylých jednotek, které pracují metodou shora dolů.

Uživatelské rozhraní

Program obsahuje triviální textové uživatelské rozhraní. Dovoluje uživateli v parametrech spuštění programu specifikovat vstupní soubor s daty lexémů. Program je spouštěn následovně:

```
./main FILE
```

kde `FILE` je soubor se vzorkem dat. Podoba tohoto souboru je blíže specifikována v sekci 7.4. Projekt mimo jiné obsahuje i sadu připravených vstupních souborů pro možnost ukázky funkcionality. Tyto soubory jsou odlišitelné především podle svých přípon `.sample`.

7.4 Scanner

Následuje popis struktury pro generování posloupnosti tokenů, pro jednoduchost nazývané *scanner*. V první části jsou popsány základní požadavky parseru na fungování modulu scanneru. Následně je ukázáno, jak je implementováno načítání posloupnosti tokenů ze souboru a jeho formát. V poslední části je uvedeno komunikační rozhraní scanneru a funkce, které poskytuje.

Základní požadavky na posloupnost tokenů

Implementace parseru vyžaduje od zdroje tokenů specifické vlastnosti. Tyto vlastnosti lze shrnout následovně:

- scanner ignoruje komentáře a bílé znaky ve zdrojovém souboru,
- ke každému znaku odsazení musí být vygenerován znak zrušení odsazení,
- poslední neprázdný řádek, obsahující scannerem neignorovanou část jazyka, musí být zakončen znakem konce řádku,
- dokumentační řetězce jsou scannerem vnímány a odesílány jako řetězec znaků.

V současné podobě jsou ovšem tyto požadavky uspokojeny podstatou načítání tokenů. Data o tokenech jsou zadávána přímo uživatelem, který zodpovídá za jejich splnění. Splněny musí být i v případě chybné syntaxe vstupního jazyka.

Načítání posloupnosti tokenů

Vzhledem k podstatě této práce byl scanner navržen pouze jako generátor posloupnosti tokenů. Nedochází tedy k lexikální analýze, nýbrž pouze k načítání a zpřístupnění předem připravených tokenů. Tato data jsou uložena ve vstupním souboru s následujícím formátem:

```
Typ tokenu  Hodnota
```

V případě tokenů, které nevyžadují hodnotu, je tato položka vynechána. Příklad takového vstupního souboru obsahujícího data o části jazyka je následovný:

```
BTTG_KEYWORD    while
BTTG_INT        1
BTTG_EXP_EQ
BTTG_INT        1
BTTG_COLON
BTTG_EOL
BTTG_INDENT
BTTG_KEYWORD    pass
BTTG_EOL
BTTG_DEDENT
BTTG_EOF
```

Soubor je zpracován modulem scanneru, převeden na posloupnost tokenů a načten do interního vektoru. Z tohoto místa jsou tokeny poskytovány pomocí komunikačního rozhraní parseru.

Komunikační rozhraní scanneru

Scanner poskytuje komunikační rozhraní pro parser formou dvou funkcí. První z nich je `preview_token`. Umožňuje náhled na libovolný počet tokenů vpřed. Tato funkce přebírá parametrem index tokenu, na který chce parser nahlédnout. Lze tak nahlédnout pouze na následující tokeny, ne ovšem na předešlé. Tato funkce je parserem využívána především u předvídání existence výrazu na aktuálním řádku, a to s výhledem až na index 4. Token, na který je takto nahlédnuto, není ze vstupu odstraněn. Poskytnuta je pouze jeho kopie.

Druhou funkcí je `get_next_token`. Tato funkce vždy vrací následující token posloupnosti. Takto zpřístupněný token je okamžitě odstraněn z interního uložiště scanneru. Dochází tak ke zpracování a posunutí vstupní posloupnosti dále.

7.5 Přijímaný jazyk

Tato sekce představuje parserem přijímaný jazyk. První část stručně popisuje jazyk IFJ19, který je v rozšířené podobě podporován. V druhé části jsou uvedeny rozšíření tohoto jazyka.

IFJ19

Imperativní jazyk IFJ19 tvoří základ analyzátorem podporovaného jazyka. Jedná se o podmnožinu rozšířeného jazyka Python 3. Podporuje základní datové typy a konstrukce. Jazyk v základu nepodporuje třídní programování. Bližší specifikace je uvedena v zadání projektu předmětu IFJ roku 2019[2].

Rozšíření

Parser mimo původní jazyk IFJ19 podporuje i řadu rozšíření. Jedná se o struktury slovník a pole, převzaté z originálního Python 3, cyklus `do/while`, pro který byla inspirací jeho C++ podoba. Dále jsou zde konstrukce `try/except/finally` pro možnost zachycení výjimek a

klíčové slovo `raise` pro jejich vyvolání. V neposlední řadě jsou zde zavedeny i základy konstrukce třídy.

Slovník

Prvním podporovaným rozšířením je slovník. Jedná se o strukturu známou též pod pojmem *asociativní pole*. Dovoluje uživateli specifikovat podobu indexů jednotlivých položek. Deklarace slovníků má následující syntax:

```
ID = {<vyraz>:<vyraz>, ...}
```

Pro přístup k položkám již existujícího slovníku byl zaveden operátor `[]`. Ten je možné využít jak vně, tak uvnitř výrazů. Při indexování za pomoci operátoru **mimo výraz**, tedy například na levé straně přiřazení, není možné jako index struktury použít výraz, nýbrž pouze jednoduché datové položky (ID/int/float/str/None).

Jedná se o binární operátor. Povolené typy operandů, jak bylo naznačeno výše, se liší dle místa použití. Mimo výrazy je prvním operandem vždy identifikátor. Druhým mohou být zmíněné jednoduché datové položky. Uvnitř výrazů je prvním operandem opět pouze identifikátor, druhým operandem může ovšem být celý výraz.

V rámci výrazu vykazuje operátor `[]` vlastnosti závorek. Má tedy nejvyšší prioritu a jeho druhý operand je vždy zpracováván jako celek, první.

Následně je dán konkrétní příklad možného použití slovníku v praxi. Příklad obsahuje definici slovníku s dvěma položkami a dále přístup k prvků struktury vně a uvnitř výrazu za pomoci zavedeného operátoru `[]`.

```
a = {"b": "retezec", 1+1: 24}

if a[1+1] >= 10:
    a[2] = 0
else:
    pass
```

Pole

Pole jsou struktury značně podobné výše popsaným slovníkům. Jedná se o neasociativní, nehomogenní, seřazené posloupnosti položek. Indexy položek jsou automaticky odvozeny podle pořadí prvku. Syntax definice pole je následující:

```
ID = [<polozka>, ...]
```

kde *<polozka>* zastupuje jednoduché datové položky (ID, int, float, str, None). Ukládání výsledku výrazu při deklaraci pole není podporováno.

Obdobně jako u slovníků je k přístupu k prvkům využíván operátor `[]` v podobě, v jaké byl popsán výše.

Následuje konkrétní příklad možného použití polí. Příklad ukazuje definici pole o dvou hodnotách a následný přístup k prvkům uvnitř a vně výrazu.

```
a = ["b", 24]

while a[1] >= 10:
    a[1] = a[1] - 1
```

Do/while

Cyklus `do/while` není v jazyce Python 3 podporován. Jeho podoba je tedy založena za podobě tohoto cyklu v jazyce C++. Jedná se o cyklus s podmínkou na konci, vždy je tedy proveden alespoň jednou. Vzhledem k absenci této konstrukce v jazyce Python 3 byla jeho podoba přizpůsobena podobě ostatních. Blok je uvozen klíčovým slovem `do`. Následuje `:` a posloupnost příkazů. Mezi těmito příkazy se mohou vyskytovat i libovolné podporované konstrukce jazyka. Toto seskupení příkazů je ukončeno klíčovým slovem `while` následovaným výrazem.

Syntax tohoto rozšíření je tudíž následující:

```
do:
    <blok prikazu>
while <výraz>
```

Dále je uveden konkrétní příklad možného použití. Jedná se o výčet hodnot $0 \rightarrow 10$ a jejich vytisknutí.

```
a = 0
do:
    a = a + 1
    print(a)
while a < 10
```

Try/except/finally/raise

Jedná se o konstrukci, nacházející se v jazyce Python 3, pomocí které je uživatel schopen zachytit a zpracovat vyvolanou vyjímku. Konstrukce se skládá ze tří bloků, každý uvozený vlastním klíčovým slovem. Jsou to klíčová slova `try`, `except` a `finally`. Po každém tomto klíčovém slově následuje blok posloupnosti příkazů. Uvnitř těchto částí lze též použít libovolné podporované jazykové konstrukce. Není podporováno vynechání jednotlivých klíčových slov, je tedy nutno vždy uvést i nepoužívané části. Syntax je následující:

```
try:
    <blok prikazu>
except ID:
    <blok prikazu>
finally:
    <blok prikazu>
```

Za účelem možnosti vyvolat výjimku bylo přidáno klíčové slovo `raise`. Následně je očekávána hodnota vyvolané výjimky. Hodnotu lze zapsat formou výrazu. Tento příkaz lze použít i mimo konstrukci pro zachycení výjimek.

```
raise <vyraz>
```

Dále je uveden konkrétní příklad použití nově zavedeného `try/except/finally` společně s příkazem `raise`. V ukázce lze též vidět nepoužitý blok `finally`, jehož vynechání není podporováno.

```
try:
    a = foo()
    a = a + 10

    if a > 20:
        raise "Chybna hodnota"
    else:
        pass
except e:
    print(e)
finally:
    pass
```

Třídy

Posledním rozšířením jazyka je podpora základní konstrukce třídy. Syntax tohoto bloku jazyka je založena na její podobě v jazyce Python 3. Nově je zavedeno klíčové slovo `class`. Je podporováno vytvoření jednoduché třídy bez možnosti dědění. Každá takto nově vytvořená třída vyžaduje vlastní identifikátor. Těmto třídám lze zavádět atributy a metody. Po syntaktické stránce lze do těla zapsat libovolnou posloupnost příkazů či jiných jazykových konstrukcí. Je tudíž podporováno, mimo jiné, i zanořování tříd. Syntax zápisu konstrukce třídy je následující:

```
class ID:
    <blok prikazu>
```

kde *<blok prikazu>* představuje libovolnou posloupnost podporovaných příkazů či konstrukcí jazyka.

Za účelem přístupu k členům tříd byl přidán operátor `.` . Pomocí tohoto operátoru je možné zapsat přístup k jednotlivých atributům či metodám objektu. Operátor je možné použít pouze s operandy typu *identifikátor/ID* a to pouze vně výrazů. Použití tohoto operátoru je tedy omezeno na samostatné členy pravých stran přiřazení nebo či užití na celém vlastním řádku.

Následuje konkrétní příklad použití klíčového slova `class` pro definici nové třídy a jejího těla. Ukázka pokračuje vytvořením nové instance a přístupu k její metodě.

```
class a:
    name = "tridaA"

    def print_name(self):
        n = self.name
        print(n)

trida = a()
trida.print_name()
```

Kapitola 8

Závěr

První částí práce bylo studium struktury překladače. Jsou zde uvedeny hlavní logické kroky překladu, lexikální analýza, syntaktická analýza a generování cílového kódu. Dále jsou v rámci těchto logických kroků vytvářeny jednotlivé části překladače, které zajišťují konkrétní úlohy. První je lexikální analyzátor, kontrolující lexikální správnost vstupního programu a překlad na posloupnost tokenů. Dalším, k této práci nejdůležitějším a dále rozebíraným, je syntaktický analyzátor. Stará se o analýzu správné větné skladby vstupního jazyka dle gramatických pravidel. Třetí částí je sémantický analyzátor, který hodnotí, zdali dává vstupní věta smysl. Čtvrtou částí je generátor vnitřního kódu překladače. Zajišťuje vytvoření interpretace vstupního programu ve vnitřním kódu, který lze díky vhodné formě optimalizovat následující jednotkou optimalizace. Poslední částí je generátor cílového kódu, ten má nastarosti transformaci vnitřního kódu na cílový.

Druhá část práce popisuje syntaxí řízený překlad. Jedná se o překlad, při kterém je hlavní jednotkou překladače syntaktický analyzátor a na základě jeho akcí jsou prováděny dílčí procesy. V syntaxí řízeném překladu bývají typicky sloučeny jednotky syntaktické analýzy, sémantické analýzy a generování vnitřního kódu. Obecně lze tedy říci, že syntaxí řízený překlad je překlad z reprezentace vstupního programu posloupností tokenů na jeho funkcionálně ekvivalentní reprezentaci ve vnitřním kódu kompilátoru.

Třetí část byla věnována studiu základních metod konstrukce kompilátoru. První studovanou metodou je metoda shora dolů. Syntaktický analyzátor konstruovaný touto metodou provádí levou derivaci vstupní věty. Derivační strom je při této metodě procházen od kořene k listům. Jedním z nejznámějších přístupů, dále v práci používaný, je rekurzivní sestup. Druhou metodou, studovanou a použitou v této práci, je metoda zdola nahoru. Analýza použitím této metody provádí pravou derivaci vstupní věty a derivační strom je při této metodě procházen od listů ke kořeni.

Čtvrtou částí bylo studium gramatických systémů. Z řad gramatických systémů jsme vybrali kooperující distribuované gramatické systémy. CDGS byly vybrány pro svoji jednoduchost a tím pádem možnost prezentace aplikace principu gramatických systémů při konstrukci syntaktického analyzátoru užitím kombinace dílčích studovaných metod. Zvolené kooperující distribuované systémy pracují sekvečně. Jejich komponenty spolupracují na jedné výstupní větě dle komunikačního protokolu. Hlavním prvkem komunikace je derivační mód. V rámci práce byly představeny módy ukončující (*terminating*), $\leq k$, $\geq k$, $= k$ a derivační mód *.

V páté části byla úspěšně navržena metoda konstrukce syntaktického analyzátoru. Nová metoda vychází z principu gramatických systémů. Smyslem této metody je rozdělení syntaktického analyzátoru na samostatné jednotky. Všechny jednotky jsou zaštítěny řídícím

systémem. Tyto jednotky spolupracují na společné výstupní větě. Stejně jako u gramatických systémů jsou zde použity derivační módy pro komunikaci sériově pracujích částí. Na rozdíl od gramatických systémů jsou ale jednotky navzájem izolované a nesdílí své množiny terminálů a neterminálů se zbytkem. Za účelem možnosti komunikace, tedy předávání řízení dalším jednotkám, bylo zapotřebí zavedení zmíněného řídícího systému. Tento řídící systém poskytuje dílčím jednotkám nezbytné informace pro signalizaci aktivace cizích jednotek.

Dílčí jednotky kompilátoru, konstruovaného navrženou metodou, jsou nezávislé vůči ostatním, mohou tedy provádět libovolnou analýzu. To umožňuje kombinaci metod analýzy a jejich implementací. Vlastnosti navržené metody jsou tudíž kombinací vlastností studovaných metod.

Šestá část práce se věnuje implementaci metody navržené v předchozí části. V rámci práce byl, vzhledem k podstatě práce, implementován pouze syntaktický analyzátor a provizorní generátor posloupnosti tokenů. Zbylé jednotky překladače byly vynechány. Celkem byl syntaktický analyzátor rozdělen na sedmnáct jednotek. Důvodem počtu je snaha přidělit každé jednotce smysluplnou část jazyka a tak názorně ukázat možnosti spolupráce i v případě složitějších jazyků. Při implementaci komunikace byl využit derivační mód *, který byl vybrán jako nejvhodnější pro svoji přirozenost a intuitivnost.

Za účelem ukázky kombinace dílčích existujících metod je vstupní jazyk analyzován metodou shora dolů a jeho výrazy metodou zdola nahoru. Tuto skutečnost lze pozorovat na výstupu syntaktického analyzátoru. Výstupem implementovaného syntaktického analyzátoru je posloupnost gramatických pravidel použitých během analýzy. Pořadí vypsaných pravidel tedy záleží na metodě prováděné analýzy.

Práce nám ukázala nejen novou metodu konstrukce kompilátoru, ale také další směr možného výzkumu. Tato práce pokrývá pouhý základ gramatických systémů, na jejichž principu je založena. V tomto směru se skrývá dlouhá řada možných úprav a vylepšení.

Jedním z těchto vylepšení a vhodným tématem na rozšíření této práce, je použití hybridných CDGS na místo základních. Tento přístup by mimo jiné pravděpodobně přinesl i vyšší generativní sílu systému jako celku.

Generativní síla navržené metody nebyla v rámci práce zkoumána. Ačkoliv přepokládám převzetí generativní síly od rodičovských gramatických systémů, byl by vhodný dodatečný výzkum i v této oblasti.

Dalším možným vylepšením, kterým se práce nezabývá, je použití zbylých derivačních módů. S touto úpravou přichází ovšem i potřeba složitého návrhu jednotlivých gramatik a pravděpodobně odlišný přístup k implementaci systému.

Jako hlavní výhodu konstrukce kompilátoru navrženou metodou vidím široké možnosti individuálního řešení různorodých problémů při analýze jazyka. Metoda uživateli umožňuje nejen silné generativní schopnosti, dává ale také možnosti analyzovat podmnožiny jazyka preferovanými metodami a logicky tak členit výsledný systém. Navržená metoda tak může přijít vhod především u rozboru složitějších a rozsáhlejších jazyků.

Literatura

[1] Hopcroft, J. E., Motwani, R. a Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. 3rd edition. Pearson Education, 2006. ISBN 0321486811.

[2] Křivka, Z., Zobal, L. a Genčúrová Lubica. *ZADÁNÍ PROJEKTU Z PŘEDMĚTŮ IFJ A IAL* [online]. 2019 [cit. 2020-12-28]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIFJ-IT%2Fprojects%2FHistory%2Fifj2019.pdf&cid=13305.

[3] Meduna, A. *Automata and Languages : Theory and Applications*. 2000 ed. Springer London Ltd, Aug 2000. ISBN 9781852330743.

[4] Meduna, A. a Lukáš, R. *Opora IFJ* [online]. 2006 + revize 2009-2015 [cit. 2020-11-30]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIFJ-IT%2Ftexts%2F0poraIFJ.pdf&cid=12745.

[5] Meduna, A. a Lukáš, R. *Lexikální analýza* [online]. 2017 [cit. 2021-04-07]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj05-cz.pdf.

[6] Meduna, A. a Lukáš, R. *Optimalizace a generování cílového programu* [online]. 2017 [cit. 2021-04-07]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj10-cz.pdf.

[7] Meduna, A. a Lukáš, R. *Syntaktická analýza shora dolů* [online]. 2017 [cit. 2020-11-30]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj07-cz.pdf.

[8] Meduna, A. a Lukáš, R. *Syntaktická analýza zdola nahoru* [online]. 2017 [cit. 2020-11-30]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj08-cz.pdf.

[9] Meduna, A. a Lukáš, R. *Syntaxí řízený překlad a generování vnitřního kódu* [online]. 2017 [cit. 2021-04-07]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj09-cz.pdf.

[10] Meduna, A. a Lukáš, R. *Úvod do překladačů* [online]. 2017 [cit. 2021-04-07]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj02-cz.pdf.

[11] Meduna, A., Techet, J. a Masopust, T. *Cooperating Distributed Grammar Systems* [online]. 2007 [cit. 2020-11-30]. Dostupné z: http://www.fit.vutbr.cz/~meduna/work/lib/exe/fetch.php?media=lectures:phd:tid:frvs:09-cdgspres.pdf.

[12] Sippu, S. a Soisalon Soininen, E. *Parsing Theory.* 1st edition. Springer-Verlag, 1988. ISBN 978-3-642-64801-4.

[13] Černá, I., Křetínský, M. a Kučera, A. *Automaty a formální jazyky.* verze 1.3.1. 2002.

Příloha A

Použité gramatiky

Pravidla gramatik jednotek parseru

```
P0
1   <S0>        ->  ID
2   <S0>        ->  int
3   <S0>        ->  float
4   <S0>        ->  str
5   <S0>        ->  None
6   <S0>        ->  <S0> + <S0>
7   <S0>        ->  <S0> - <S0>
8   <S0>        ->  <S0> * <S0>
9   <S0>        ->  <S0> / <S0>
10  <S0>        ->  <S0> // <S0>
11  <S0>        ->  (<S0>)
12  <S0>        ->  <S0> < <S0>
13  <S0>        ->  <S0> > <S0>
14  <S0>        ->  <S0> <= <S0>
15  <S0>        ->  <S0> >= <S0>
16  <S0>        ->  <S0> == <S0>
17  <S0>        ->  <S0> != <S0>
18  <S0>        ->  ID[<S0>]

P1
1   <S1>        ->  <LINE> <S1>
2   <S1>        ->  EOF
3   <LINE>      ->  ID <S2>
4   <LINE>      ->  def <S4>
5   <LINE>      ->  if <S7>
6   <LINE>      ->  while <S8>
7   <LINE>      ->  pass EOL
8   <LINE>      ->  <S0> EOL
9   <LINE>      ->  do <S11>
10  <LINE>      ->  try <S12>
11  <LINE>      ->  raise <S0> EOL
12  <LINE>      ->  class <S13>
```

```
P2
1   <S2>         ->  = <S14>
2   <S2>         ->  ( <S3> EOL
3   <S2>         ->  [ <INDEX>
4   <S2>         ->  . ID ( <S3> EOL
5   <S2>         ->  EOL
6   <INDEX>      ->  ID ] = <S14>
7   <INDEX>      ->  int ] = <S14>
8   <INDEX>      ->  float ] = <S14>
9   <INDEX>      ->  str ] = <S14>
10  <INDEX>      ->  None ] = <S14>

P3
1   <S3>         ->  )
2   <S3>         ->  <ARG> <ARG REP>
3   <ARG REP>    ->  )
4   <ARG REP>    ->  , <ARG> <ARG REP>
5   <ARG>        ->  ID
6   <ARG>        ->  int
7   <ARG>        ->  float
8   <ARG>        ->  str
9   <ARG>        ->  None

P4
1   <S4>         ->  ID ( <S5> : EOL INDENT <S6> DEDENT

P5
1   <S5>         ->  )
2   <S5>         ->  ID <ARG REP>
3   <ARG REP>    ->  )
4   <ARG REP>    ->  , ID <ARG REP>

P6
1   <S6>         ->  <LINE> <S6>
2   <S6>         ->  DEDENT
3   <LINE>       ->  ID <S2>
4   <LINE>       ->  def <S4>
5   <LINE>       ->  if <S7>
6   <LINE>       ->  while <S8>
7   <LINE>       ->  pass EOL
8   <LINE>       ->  <S0> EOL
9   <LINE>       ->  do <S11>
10  <LINE>       ->  try <S12>
11  <LINE>       ->  raise <S0> EOL
12  <LINE>       ->  class <S13>
13  <LINE>       ->  return <RET VAL>
14  <RET VAL>    ->  <S0> EOL
```

```
15  <RET VAL>    ->  EOL

P7
1   <S7>         ->  <S0> : EOL INDENT <S16> else : EOL INDENT <S16>

P8
1   <S8>         ->  <S0> : EOL INDENT <S16>

P9
1   <S9>         ->  }
2   <S9>         ->  <S0> : <S0> <S15>

P10
1   <S10>        ->  ]
2   <S10>        ->  <ARG> <ARG REP>
3   <ARG REP>    ->  ]
4   <ARG REP>    ->  , <ARG> <ARG REP>
5   <ARG>        ->  ID
6   <ARG>        ->  int
7   <ARG>        ->  float
8   <ARG>        ->  str
9   <ARG>        ->  None

P11
1   <S11>        ->  : EOL INDENT <S16> while <S0> EOL

P12
1   <S12>        ->  : EOL INDENT <S16> except ID : EOL INDENT <S16>
                     finally : EOL INDENT <S16>

P13
1   <S13>        ->  ID : EOL INDENT <S16>

P14
1   <S14>        ->  <S0> EOL
2   <S14>        ->  ID <RIGTH ID>
3   <S14>        ->  { <S9> EOL
4   <S14>        ->  [ <S10> EOL
5   <RIGTH ID>   ->  . ID <OBJ END>
6   <RIGTH ID>   ->  ( <S3> EOL
7   <OBJ END>    ->  ( <S3> EOL
8   <OBJ END>    ->  EOL

P15
1   <S15>        ->  }
2   <S15>        ->  , <S0> : <S0> <S15>

P16
```

```
1   <S16>       ->  <LINE> <S16>
2   <S16>       ->  DEDENT
3   <LINE>      ->  ID <S2>
4   <LINE>      ->  def <S4>
5   <LINE>      ->  if <S7>
6   <LINE>      ->  while <S8>
7   <LINE>      ->  pass EOL
8   <LINE>      ->  <S0> EOL
9   <LINE>      ->  do <S11>
10  <LINE>      ->  try <S12>
11  <LINE>      ->  raise <S0> EOL
12  <LINE>      ->  class <S13>
```

Příloha B

Obsah přiloženého média

Přiložené médium obsahuje zdrojové texty programu v jazyce C++, manuál k přeložení a spuštění a zdrojové soubory zprávy v LaTexu včetně samotné práce ve formátu PDF. Médium obsahuje následující adresářovou strukturu:

- src – adresář se zdrojovými texty programu a manuálem README
  - bin – adresář pro přeložené spustitelné soubory
  - obj – adresář pro objektové soubory
  - parser – adresář se zdrojovými soubory spojenými s částí syntaktického analyzátoru
    - units – adresář se zdrojovými soubory spojenými s jednotkami parseru
  - samples – adresář s ukázkovými vstupními soubory
  - scanner – adresář se zdrojovými soubory spojenými s generátorem tokenů
  - utils – adresář se zdrojovými soubory pomocných funkcí a struktur
- thesis – adresář se soubory zprávy
  - src – adresář se zdrojovými soubory zprávy