Vysoké učení technické v Brně
Fakulta informačních technologií

Dokumentace k projektu do předmětů IFJ a IAL

Překladač jazyka IFJ18

Tým 068 – varianta II. rozdělení

Peter Krutý	(xkruty00)	25%
Bořek Reich	(xreich06)	25%
Martin Chládek	(xchlad16)	25%
Michal Krůl	(xkrulm00)	25%

5. prosince 2018

Obsah

1 Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načte zdrojový kod zapsaný ve zdrojovém jazyce IFJ18, který je zjednodušenou podmnožinou jazyka Ruby 2.0 a přeloží jej do cílového jazyka IFJcode18 (mezikód). Program funguje jako aplikace, která načítá řídící program ze standartního vstupu a generuje výsledný mezikód na standartní výstup. V případě chyby bude na standartní výstup vrácen náležitý chybový kód.

2 Návrh a implementace

Námi vytvořený program je sestaven z jednotlivých souborů, které jsme v průběhu několika týdnů implementovali. Tato kapitola obsahuje stručný popis dílčích částí a způsob vzájemné kooperace mezi nimi.

2.1 Lexikální analýza

Funkce provádějící lexikální analýzu (LA) programu je implementována v souboru `scanner.c`. LA je implementována jako konečný automat podle diagramu. Naše implementace konečného automatu využívá cyklicky se opakující příkaz `switch`, kde každá větev odpovídá jednomu stavu automatu. Cyklus, který je nastaven jako nekonečný, se provádí dokud stav automatu není konečný, v našem případě tedy dokud není nalezen token odpovídající danému lexému, a poté končí.

Hlavní funkcí LA a zároveň jedinou funkcí, která je využívána v jiných částech programu je funkce `get_token`. Pro načítání vstupních lexémů využívá LA operace s dynamicky alokovanými řetězci, které jsou naimplementované v souboru `str.c` a blíže popsané v kapitole 3.3. Využívaný řetězec s názvem buffer je před začátkem LA inicializován a po nalezení každého tokenu a předání dál vymazáván.

Token je definován strukturou, která má dva parametry, a to typ tokenu (může být identifikátor, klíčové slovo, číslo, řetězece, aritmetické a logické znaky apod.) a atribut, který je využíván pouze v případě že je nutné uložit kromě typu lexému i nějakou jeho hodnotu (například text řetězce apod.). Token je předáván funkci odkazem a poté zpracováván dále jinými částmi programu.

Pro zpracování escape sekvencí uvnitř řetězce je vytvořeno pomocné pole, implicitně vynulované a postupně naplňované podle toho jak načítáme znaky. Při načtení celé sekvence je vstup převeden do ASCII podoby.

Funkce `get_token` využívá celou řadu pomocných funkcí, například pro rozpoznání klíčového slova jazyka, předdefinované funkce anebo operátoru. K tomu jsou využívána předdefinovaná pole, kde je načtený prvek porovnáván se všemi prvky těchto polí a je hledána shoda.

2.2 Syntaktická analýza konstrukcí

Syntaktický analýzator (anglicky parser) je komponenta prekladače, která má na starosti proces analýzy tokenů vstupního programu. V syntaxí řízeném překladě je považovaný za mozek překladače a jeho návrh a implementace jsou náročné.

Před samotnou implementací byl potřebný návrh LL-gramatiky, kde jsme narazili na prekážky, které nám bránili ke korektnímu a úplnému popisu programovacího jazyka IFJ18. Po dokončení LL-gramatiky, která řídí analýzu, jsme odvodili LL-tabulku, kde je popsané volání pravidla v závislosti na terminálu (vstupním tokenu) a neterminálu (např. `<prog>`).

U množství možných implementací založených na LL-gramatice jsme se rozhodli pro použití syntaktické analýzy shora dolů. Konkrétní metody rekurzivního sestupu jsme implementovali v modulu `parser.c`. Rozhraní funkcí a deklaraci potřebných datových typů pro analýzu se nachází v hlavičkovém souboru `parser.h`. Implementaci jsme zvolili tak, aby každému neterminálu (např. `<prog>`) odpovídala jedna funkce. Funkce se do sebe navzájem vnořují a rekurzivně volají, přičemž si předávají ukazatel na strukturu `par_data`, kde se ukládají všechny potřebné data pro analýzu. V jistých případech syntaktické analýzy dochází k nejednoznačnosti použití pravidla (zvýrazněné červěně v LL-tabulce) a tak jsme museli přistoupit k odlišnému řešení. Řešení spočívá ve volání a ukládání extra tokenu na základě kterého se identifikuje které pravidlo bude volané.

Při implementaci jsme se snažili dělat věci co nejjednodušší a tak jsme si definovali následující makra:

- `GET_TOKEN()` – makro si zavolá token od scanneru, který uloží jako aktuální token ve struktuře `par_data`
- `GET_EXTRA_TOKEN()` – makro si zavolá extra token od scanneru (využití při nejednoznačnosti použití pravidla), který si uloží jako extra token ve sktruktuře `par_data`
- `CHECK_TOKEN(_type_)` – makro zkontroluje typ aktuálního tokenu na základě parametru `_type_`
- `CHECK_EXTRA_TOKEN(_type_)` – makro zkontroluje typ extra tokenu na základě parametru `_type_`
- `CALL_RULE(rule)` – makro zavolá další pravidlo na základě parametru `rule`
- `CHOOSE_SYMTAB(_symtabMain_, _symtabLocal_)` – makro vybere se kterou lokání tabulkou symbolů se bude pracovat podle toho, zda se momentálně analýza nachází v `main` anebo v definici funkce

2.3 Syntaktická analýza výrazů

Precedenční syntaktická analýza pro zpracování výrazů je definována v souboru `expression.c`. Jejím základem je precedenční tabulka. Díky stejné asociativitě a prioritě operátorů + a −, ∗ a / mohla být tabulka navrhnuta značně zjednodušeně. Stejnou prioritu mají také všechny relační operátory a všechny identifikátory. Relační operátory jsou v tabulce označeny **r** a identifikátory **i**. Hlavní funkcí SA pro výraz a zároveň jedinou funkcí volanou v jiných částech programu je funkce `expression`.

Funkce `expression` je volána poté co byl již načten minimálně jeden token. Ten je předáván funkci odkazem jako součást struktury `par_data`. Existuje možnost, že byly před voláním funkce načteny již tokeny dva, druhý token je předáván stejným způsobem v té samé struktuře.

Implementace pomocí cyklického provádění funkce `get_token` (kapitola 2.1) načítá tokeny pro jejichž uložení využívá zásobník `tStack` implementovaný v souboru `stack.c` a popsaný v kapitole 3.4. Výsledný derivační strom se poté vytváří podle pravidel precedenční syntaktické analýzy za využití precedenční tabulky.

Program využívá řadu pomocných funkcí, například pro rozpoznání pravidla které popisuje právě načtený výraz a nebo funkci, která dle daného pravidla provádí syntaktickou kontrolu daného podvýrazu a při úspěchu vrací předpokládaný výsledný datový typ pro další kontrolu v rámci derivačního stromu. Náš návrh syntaktické analýzy je úzce provázaný s generováním kódu a tedy funkce z generátoru jsou volány v souboru `expression.c`.

2.4 Sémantická analýza

Sémantická analýza je přidružená k syntaktické analýze. Pravidla implementovaná rekurzivním sestupem jsou doplněna o sémantické kontroly, při kterých jsou využívány tabulky symbolů uložené ve struktuře `par_data`. Sémantické kontroly spočívají v kontrole, zda byla daná proměnná/funkce již definována, případně v kontrole datových typů ve výrazech.

2.5 Generování cílového kódu

Generování cílového kódu je finální fáze překladu vstupního programu v jazyce IFJ18. Cílovým kódem je v našem případě tříadresný mezikód IFJcode18, ten je průběžně ukládán do dynamického řetězce a posléze generován na standardní výstup programu.

2.5.1 Rozhraní generátoru kódu

Generování cílového kódu je zajištěno modulem `generator.c`. V tomto souboru jsou obsaženy všechny funkce potřebné ke generování mezikódu.

2.5.2 Generování základní kostry programu

Před začátkem generování je potřeba přiřadit generátoru patřičné prostředky pomocí funkce `init_generator()` a po skončení tyto prostředky opět uvolnit (funkce `clear_generator()`). První funkcí generující mezikód je `file_header()`, ta vytvoří hlavičku mezikódu, základní rámec, definuje a inicializuje pomocné globální proměnné. Následuje samotné tělo programu, které je vytvářeno ostatními funkcemi z `generator.c`. Poslední volanou funkcí je `file_end()`. Ta má za úkol rušení základního rámce programu a vytváření běhové chyby.

2.5.3 Generování funkcí

Definice funkcí

Základními funkcemi, které jsou potřeba pro vytvoření definice funkce v IFJcode18 jsou `func_start()` a `func_end()`.

První zmíněná vytvoří návěští podle identifikátoru funkce, přesune rámec TF vytvořený před voláním funkce na zásobník rámců, definuje a inicializuje proměnnou pro návratovou hodnotu a pokud má funkce parametry, tak definuje proměnné jim odpovídající a přiřadí do nich pomocné proměnné inicializované před voláním funkce.

Funkce `func_end()` vygeneruje ve spolupráci se syntaktickou analýzou kód pro správnou návratovou hodnotu funkce, dále pak přesun rámce používaného funkcí do dočasného (TF) a vrácení se zpět na místo volání funkce.

Jelikož IFJ18 nemá klasický `main`, je před každou definicí funkce zařazen nepodmíněný skok na odpovídající návěští, které je umístěné na konci definice. Tato skutečnost zabezpečí správné vykonání programu tím, že při sekvenčním provádění kódu odpovídajícího main bude definice funkce vždy přeskočena.

Volání funkcí

Funkce `func_param_pass_prep()`, `generator_func_call()`, `func_param_passing()` a `func_return_value_after_called_func()` jsou nezbytné pro generování volání funkcí.

- `func_param_pass_prep()` – funkce pro vytvoření nového datového rámce pro funkci
- `func_param_passing()` – na datovém rámci vytvořeném funkcí `func_param_pass_prep()` definuje pomocné proměnné pro předání parametrů funkci a přiřadí jim požadovanou hodnotu
- `generator_func_call()` – funkce generující samotné volání funkce pomocí instrukce `CALL`
- `func_return_value_after_called_func()` – vytvoří mezikód pro přiřazení návratové hodnoty funkce do požadované proměnné, v případě, že má být návratová hodnota uložena

2.5.4 Generování výrazů

Jednotlivé termy jsou ukládány na datový zásobník a při zavolání funkce `stack_operations()` nebo `if_func_start()` je nejprve generován kód pro provedení kontroly datových typů a posléze i požadované operace.

Kontrola datových typů probíhá za běhu výsledného programu. Pokud jsou rozpoznány nesprávné datové typy operandů instrukce, vykonávání programu bude přerušeno s chybou 4 - sémantická/běhová chyba, nebo 9 - běhová chyba dělení nulou.

V případě zavolání funkce `if_func_start()` je výsledkem provedení kódu pravdivostní hodnota na vrcholu zásobníku, která je dále použita při vyhodnocování podmíněných výrazů. V případě `stack_operations()` je na vrchol zásobníku vložena výsledná hodnota výrazu.

2.5.5 Generování návěští

Náveští jsou generována pomocí několika funkcí. Nejzákladnější je `label_func()`, ta vygeneruje návěští podle zadaného řetězce. Dále může být použita funkce `label_id()`, která na základě dvou zadaných celočíselných hodnot vygeneruje název, který může být použit např. v podmíněných nebo nepodmíněných skocích.

2.6 Kooperace mezi moduly

2.6.1 Lexikální analýza ↔ Syntaktická analýza

Syntaktická analýza volá funkci lexikálního analyzátoru `get_token` implementovanou ve `scanner.c`. Tato funkce vrátí token s jeho příslušným typem a atributem ve formě struktury `tToken`. Sled volaných tokenů je následně analyzován SA.

2.6.2 SA konstrukcí ↔ SA výrazů

Jedinou funkcí, kterou SA konstrukcí volá z SA výrazů je funkce `expression`. Syntaktická analýza konstrukcí jí při volání předává odkaz na strukturu `par_data`. Po vykonání syntaktické analýzy pro daný výraz je do `par_data` uložen poslední načtený token a funkcí `expression` je vrácena konstanta `COMP_SUCC` značící úspěch nebo příslušný chybový kód.

2.6.3 SA konstrukcí ↔ Generátor cílového kódu

Syntaktická analýza konstrukcí spolupracuje s generátorem cílového kódu voláním funkcí implementovaných v `generator.c`. Všechny funkce generují mezikód IFJcode18 na základě údajů, které jsou jim poskytnuty z `parser.c`.

2.6.4 SA výrazů ↔ Generátor cílového kódu

Syntaktická analýza výrazů a generování cílového kódu spolupracují na úrovni volání funkcí, kdy SA výrazů volá funkce, jejichž rozhraní je uvedeno v `generator.h`. Zejména se jedná o funkce `generate_stack_push`, `stack_operations` a `if_func_start`.

První z uvedených zajišťuje generování ukládání identifikátoru na zásobník, kdy předáván je typ tokenu a jeho atribut. Funkce `stack_operations` zajišťuje generování tříadresného kódu podvýrazů pro aritmetické operace podle pravidel, které SA výrazů předává generátoru ve formě výčtového typu. Poslední funkce, která je volána syntaktickou analýzou výrazů z `generator.h` je `if_func_start`, ta zajišťuje zpracování a generování výrazů podle pravidel, které obsahují relační operátor.

Všechny ze zmíněných funkcí mají jeden společný parametr a tím je pomocná pravdivostní hodnota `in_while`, která je uložena ve struktuře `par_data` a identifikuje, zda se generování nachazí v cyklu `while` nebo v podmínce `if`.

3 Speciální algortimy a datové struktury

Pro úspěšné vyřešení projektu jsme naimplementovali několik následujících speciálních datových struktur.

3.1 Datová struktura par_data

Datová struktura `par_data` slouží k uchovávání všech informací, které jsou potřebné při syntaktické a sémantické analýze. Jednotlivé funkce reprezentující neterminály (např. `<prog>`) si předávají její ukazatel jako jediný parametr. Tento fakt zajišťuje, že se data při zanořování, rekurzívním volání a ukončení funkce neztrácejí. Samotná struktura `par_data` je deklarována v souboru `parser.h` a inicializována je v `parser.c`. Při procesu překladu zdrojového programu ji využívají moduly `parser.c`, `expression.c` a `generator.c`. Její obsah je následující:

- `tSymtab glob_table` – globální tabulka symbolů pro ukládání funkcí
- `tSymtab loc_table` – lokální tabulka symbolů pro definice funkcí
- `tSymtab main_table` – lokální tabulka symbolů pro main
- `tToken token` – aktuálně načtený terminál (token) vstupního programu

- `tToken extra_token` – extra načtený terminál (token) vstupního programu využíván na rozhodnutí při nejednoznačnosti použití pravidla
- `tToken return_id_var_tok` – uložený terminál (token) proměnné, do kterého má být přiřazena návratová hodnota funkce
- `int label_i` – index (počítadlo) návěští if / while
- `int label_deep` – index (počítadlo) zanoření návěští if / while
- `int freturn_type` – typ návratu funkce (0 - if / while, 1 - proměnná, 2 - funkce, 3 - výraz)
- `char *freturn_id` – identifikátor v případě, že se jedná o typ návratu proměnná / funkce (jinak `NULL`)
- `bool extra_t_used` – příznak definující, zda byl načten extra terminál (token)
- `int control_par_num` – počítadlo parametrů funkce pro kontrolu při volání
- `bool in_function_def` – příznak definující, zda je analýza v definici funkce
- `bool in_main` – příznak definující, zda je analýza v main
- `bool in_var_init` – příznak definující, zda se analýza momentálně nachází v inicializaci proměnné
- `bool in_while` – příznak definující, zda je analýza ve while
- `bool default_func` – přepínač, který značí vestavěnou funkci
- `bool default_func_print` – přepínač pro rozpoznání vestavěné funkce print
- `bool rel_op` – přepínač pro rozpoznání podmínky s relačním opraátorem a bez něj
- `struct tab_item *act_f_id` – ukazatel na funkci v tabulce symbolů, která je aktuálně definována
- `struct tab_item *act_cf_id` – ukazatel na funkci v tabulce symbolů, která je aktuálně volána
- `struct tab_item *act_id` – ukazatel na položku v tabulce symbolů, která je aktuálně zpracovávána

3.2 Tabulka s rozptýlenými položkami (Hash table)

Tabulka symbolů je datová struktura, kterou používá překladač pro uložení informací o všech identifikátorech nalezených během překladu ve vstupním programu.

Implementací tabulky symbolů je mnoho, v zadání byly na výběr 2 varianty (1. binární vyhledávací strom, 2. tabulka s rozptýlenými položkami). Rozhodli jsme se pro 2. variantu, kde jsme zvolili variantu s explicitně zřetězenými synonymi. Pro zřetězení jsme použili jednosměrně vázaný seznam, protože jsme potřebovali strukturu, která dokáže měnit svou velikost za běhu programu.

Jako mapovací funkce byla zvolena varianta Daniela J. Bernsteina, která splňovala všechny naše požadavky (determinismus, rychlost, ...). Pro velikost tabulky jsme zvolili číslo 5381, protože se jedná o prvočíslo, které je dlouhodobě považováno za nejoptimálnější a vykazuje nejmenší počet kolizí.

Nad tabulkou symbolů jsou implementované následující operace: inicializace TS, vyhledávání v TS, vkládání do TS, ověřování zda je každá funce v globální TS definovaná a uvolňování TS.

Tabulka symbolů a všechny používané funkce jsou deklarovány v souboru `symtable.h`, implementace v souboru `symtable.c`

3.3 Řetězec s dynamickým rozsahem

Pro potřeby programu, především v části lexikální analýzy, kde předem není známa velikost načítaného lexému, byla vytvořena knihovna funkcí, které pracují s dynamicky alokovanými řetězci, implementovaná v souboru `str.c`. Byla definována struktura `inf_string` s parametry: aktuální velikost řetězce, velikost alokovaného bloku a řetězec jako takový.

Při inicializaci se dynamicky alokuje místo pro řetězec odpovídající velikosti alokačního bloku. V případě, že by aktuální velikost nebyla dostatečná (například bychom chtěli do řetězce přidat znak který by znamenal přesáhnutí přidělené paměti), je místo pro řetězec zvětšeno o velikost alokačního bloku.

Do dynamicky alokovaného řetězce je možno přidávat znaky anebo jiné řetězce, je možno znaky mazat anebo vymazat celý řetězec.

3.4 Zásobník symbolů

Při syntaktické analýze výrazů používáme zásobník v podobě jednosměrně vázaného lineárního seznamu, který je naimplementován v souboru `stack.c`. Ten má své rozhraní ve `stack.h`. K práci se zásobníkem jsme implementovali základní operace jako `stack_push`, `stack_top`, `stack_pop` a `stack_free`.

Položka ukládaná na zásobník je strukturou, která obsahuje informace získané ze surového tokenu načteného lexikálním analyzátorem – typ tokenu ve formě výčtového typu a atribut tokenu. Dále pak je struktura doplněna o pravdivostní hodnotu, zda token je ukončujícím znakem výrazu (značí se pomocí `$`) a ukazatel na další položku v zásobníku.

4 Práce v týmu

4.1 Způsob práce v týmu

Naši práci na projektu jsme započali začátkem října. Hlavní fáze projektu jsme si dle doporučení rozčlenili a náš vedoucí rovnoměrně rozdělil práci podle obtížnosti a časové náročnosti. Toto rozdělení se ale během vypracovávání projektu přirozeně pozměnilo, abychom docílili optimálního výsledku. Na dílčích částech projektu pracovali většinou dva studenti, případné složitější problémy však byly řešeny kolektivně v rámci celého týmu.

4.1.1 Použitý verzovací systém

Pro zaznamenávání a uchovávání souborů v průběhu času jsme používali verzovací systém Git. Tento systém nám umožnil pracovat na jednotlivých částech paralelně v tzv. větvích, kde do hlavní vývojové větve jsme nahrávali pouze přeložitelné soubory. Jako vzdálený repozitář jsme používali GitHub.

4.1.2 Komunikace v týmu

Komunikace v týmu probíhala několika různými způsoby. Pořádali jsme pravidelné setkání v univerzitní knihovně, kde se setkávali dva studenti pracující společně na dílčí části projektu, nebo celý tým. Jedním z hlavních komunikačních kanálů prostřednictvím internetu byla pro nás aplikace Discord, kterou jsme používali jak pro přenos zpráv, tak i skupinový videochat a sdílení obrazovky. Ke komunikaci jsme dále využívali Facebook Messenger, kde docházelo ke komunikaci mezi jednotlivci i hromadně v týmu.

4.2 Rozdělení práce mezi členy

Jednotlivé úkoly jsme si v týmu rozdělili takovým způsobem, abychom všichni vynaložili stejné úsilí. To je důvod, proč každý z týmu dostal procentuální hodnocení 25 %. Následující seznam shrnuje rozdělení práce v týmu mezi jednotlivými členy.

- **Peter Krutý**: vedoucí týmu, rozdělení jednotlivých fází, kontrola, syntaktický analyzátor pro řídící konstrukce, sémantická analýza, testování
- **Bořek Reich**: lexikální analyzátor, generátor cílového kódu, testování
- **Martin Chládek**: sémantická analýza, syntaktická analyza výrazů, testování, dokumentace
- **Michal Krůl**: lexikální analyzátor, syntaktická analýza výrazů, testování

5 Závěr

Cílem projektu bylo ověřit naše znalosti nabyté v předmětech IFJ a IAL, vyzkoušet naši spolupráci v týmu a získat další zkušenosti v oblasti programování.

Tým jsme měli sestavený pár dní po zvěřejnění zadání a společně jsme hodlali čelit této výzvě. Již od začátku jsme věděli, že projekt nesmíme podcenit a musíme na něm začít pracovat včas. Přestože nás projekt svou složitostí překvapil, díky týmové práci, kdy jsme si vzájemně pomáhali s pochopením problematiky, se nám podařilo funkční překladač jazyka IFJ18 sestavit.

Díky tomuto projektu jsme si prakticky osvojili praktiky probírané v předmětech IAL a IFJ. Do projektu jsme investovali spoustu času a sil a snažili jsme se ho vyřešit nejlépe jak umíme, ale přáli bychom si, abychom měli více času a mohli naimplementovat i nějaké z nabízených rozšíření.

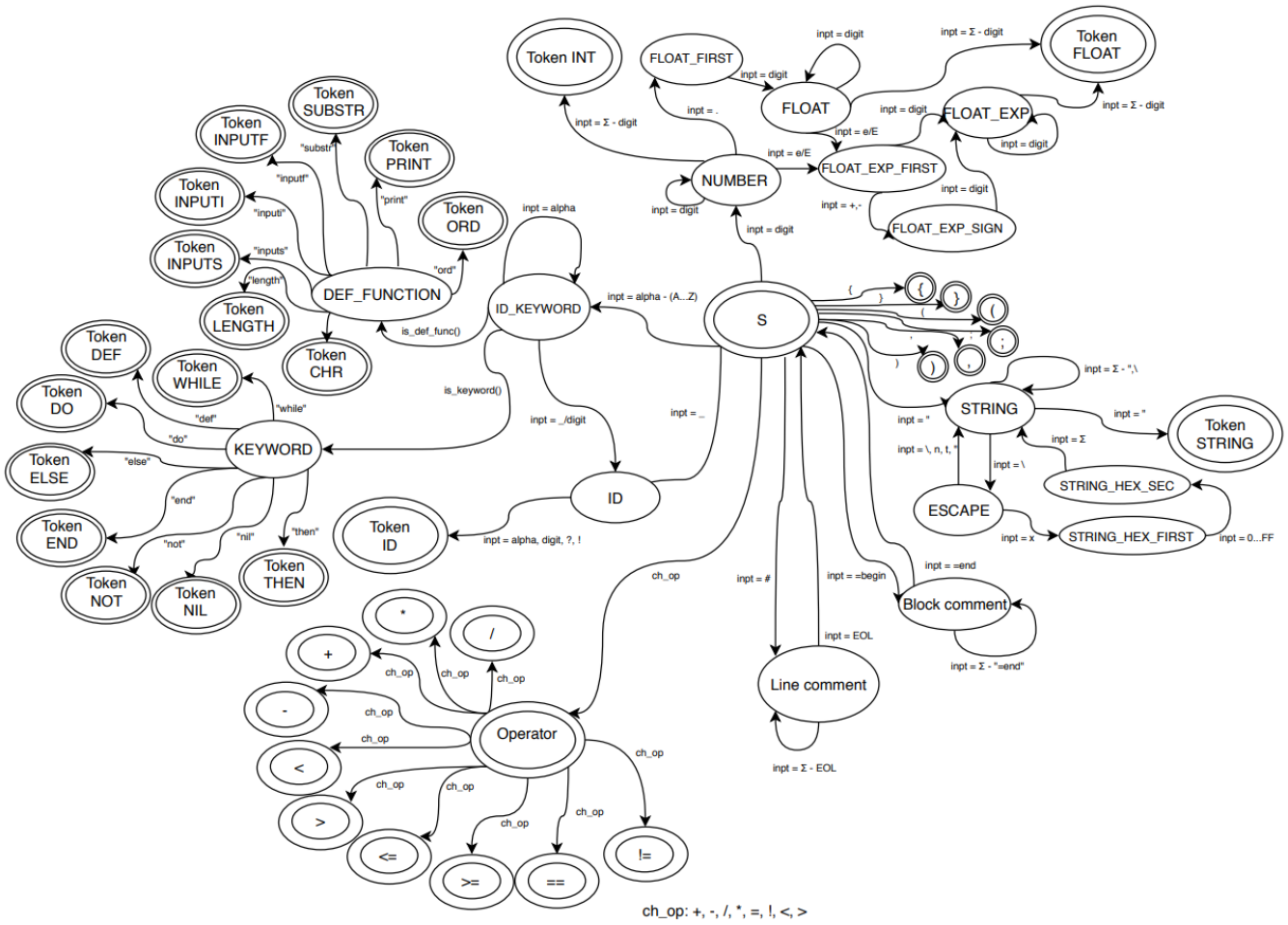
Použitá literatura

[1] HONZÍK, J. M.; BURGETOVÁ, I.; KŘENA, B.: Vyhledávací tabulky III. [online], VUT v Brně, prezentace předmětu IAL.
URL `<https://goo.gl/nS4R2U>`

[2] HONZÍK, J. M.; HRUŠKA, T.: *Vybrané kapitoly z programovacích technik.* VUT v Brně, třetí vydání, 1991, ISBN BN 80-214-034.

[3] MELICHAR, B.; ČEŠKA, M.; JEŽEK, K.; aj.: *Konstrukce překladačů II. část.* ČVUT, první vydání, Listopad 1999, ISBN 80-01-02028-2.

A Diagram konečného automatu popisující lexikální analyzátor



Obrázek 1: Diagram konečného automatu popisující lexikální analyzátor

B LL - gramatika

1. <program> → "def" "ID" "(" <param> ")" "EOL" <stat> "end" "EOL" <program>
2. <program> → <stat> <program>
3. <program> → "EOL" <stat>
4. <program> → "EOF"

5. <param> → "ID" <param_list>
6. <param> → ε

7. <param_list> → "," "ID" <param_list>
8. <param_list> → ε

9. <stat> → "ID" <var_init> "EOL" <stat>
10. <stat> → <expression> "EOL" <stat>
11. <stat> → <func_call> "EOL" <stat>
12. <stat> → "if" <expression> "then" "EOL" <stat> "else" "EOL" <stat_list> "end" "EOL" <stat>
13. <stat> → "while" <expression> "do" "EOL" <stat> "end" "EOL" <stat>
14. <stat> → "EOL" <stat>
15. <stat> → ε

16. <var_init> → "=" <expression>
17. <var_init> → "=" <func_call>
18. <var_init> → ε

19. <func_call> → "ID" <arg>
20. <func_call> → "inputi" <arg>
21. <func_call> → "inputf" <arg>
22. <func_call> → "inputs" <arg>
23. <func_call> → "print" <arg>
24. <func_call> → "length" <arg>
25. <func_call> → "substr" <arg>
26. <func_call> → "ord" <arg>
27. <func_call> → "chr" <arg>

28. <arg> → "(" <value> <arg_list> ")"
29. <arg> → "(" ")"
30. <arg> → <value> <arg_list>
31. <arg> → ε

32. <arg_list> → "," <value> <arg_list>
33. <arg_list> → ε

34. <value> → "ID"
35. <value> → "DOUBLE_NUMBER"
36. <value> → "INTEGER_NUMBER"
37. <value> → "STRING"
38. <value> → "nil"

C LL - tabulka

	<program>	<param>	<param_list>	<stat>	<var_init>	<func_call>	<arg>	<arg_list>	<value>
"EOF"	4								
"EOL"	3			14					
ID	2	5		9/10/11		19	30		34
INT	2			10			30		35
FLOAT	2			10			30		36
STRING	2			10			30		37
"nil"	2			10			30		38
"def"	1								
"do"									
"else"									
"end"									
"if"	2			12					
"not"	2			13					
"then"									
"while"									
"+"									
"−"									
"∗"									
"/"									
"="					16/17				
"<"									
">"									
"<="									
">="									
"=="									
"! ="									
"."									
","			7					32	
";"									
"("	2						28/29		
")"									
"{"									
"}"									
"inputi"	2					20			
"inputf"	2					21			
"inputs"	2					22			
"print"	2					23			
"length"	2					24			
"substr"	2					25			
"ord"	2					26			
"chr"	2					27			
ε		6	8	15	18		31	33	

Tabulka 1: LL - tabulka použitá při syntaktické analýze

V buňkách vyznačených červenou barou nastala nejednoznačnost použití pravidla. Tento problém byl řešený načítáním dalšího extra tokenu na základě kterého se určilo, jaké pravidlo se použije.

D Precedenční tabulka

	+−	*/	**i**	**r**	(	)	**$**
+−	>	<	<	>	<	>	>
*/	>	>	<	>	<	>	>
i	>	>		>		>	>
r	<	<	<		<	>	>
(	<	<	<	<	<	=	
)	>	>		>		>	>
$	<	<	<	<	<		

Tabulka 2: Precedenční tabulka použitá při syntaktické analýze výrazů