

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace do předmětů IFJ a IAL

Implementace překladače imperativního jazyka
IFJ18

Tým 032, varianta II

Řešitelé: Jakub Frejlich (xfrej100) – vedoucí, 25%
Kateřina Fořtová (xforto00), 25%
Tibor Škvrnda (xskvrn00), 25%
Lukáš Licek (xlícek01), 25%

Obsah

1	Úvod	2
2	Implementace	2
2.1	Lexikální analyzátor	2
2.2	Syntaktický a sémantický analyzátor, tabulka symbolů	2
2.3	Generátor cílového kódu	3
2.4	Testování	4
3	Vybrané algoritmy použité při řešení	4
3.1	Tabulka symbolů	4
3.2	Zásobník	4
3.3	Dynamický řetězec	4
4	Práce v týmu	5
5	Závěr	5
6	Přílohy	6
6.1	Graf konečného automatu lexikálního analyzátoru	6
6.2	LL-gramatika	7
6.3	LL-tabulka	8
6.4	Precedenční tabulka	9

1 Úvod

Dokumentace popisuje implementaci překladače imperativního jazyka IFJ18 – společný projekt do předmětů Formální jazyky a překladače (IFJ) a Algoritmy (IAL). Jazyk IFJ18 je zjednodušenou podmonžinou jazyka Ruby 2.0. Implementace se skládá z těchto částí

- Lexikální analyzátor (tzv. scanner)
- Syntaktický analyzátor (tzv. parser)
- Sémantický analyzátor
- Tabulka symbolů
- Generátor cílového kódu
- Testování

V dokumentaci mimo implementace budeme popisovat i vybrané použité algoritmy a práci v týmu. Závěr dokumentace je věnovaný přílohám – grafu konečného automatu lexikálního analyzátoru, LL-gramaticy, LL-tabulce a precedenční tabulce. Dokumentace byla vysázena v \LaTeX u.

2 Implementace

2.1 Lexikální analyzátor

Hlavním úkolem lexikálního analyzátoru je analyzovat vstupní řetězec a získávat jednotlivé lexémy - základní jednotky jazyka. Určuje o jaký typ lexému se jedná, předává jim informaci (např. jedná se o celé číslo, řetězec, komentář, atd...) a ve formě tokenu je posílá dál syntaktickému analyzátoru.

Lexikální analyzátor je implementován pomocí konečného automatu v modulu `scanner.c`. Při programování v jazyce C jsme využili konstrukci `switch` v nekonečném cyklu `while(1)`. Vždy se snažíme najít nejdelší možný lexém. Důležitá je zde funkce `getToken`, která načte právě jeden lexém a vrátí ho skrze ukazatel na strukturu `tToken`. Jakmile narazíme na znak, který pro daný stav není přípustný, vrátíme ho na `stdin`, uložíme načtený řetězec do tokenu, původní řetězec uvolníme a funkci ukončíme s návratovou hodnotou `LEX_CORRECT(0)`. Lexikální analyzátor využívá dynamického řetězce, který je implementován v modulu `dstring.c`.

Největším problémem při řešení lexikálního analyzátoru byla problematika rozpoznávání blokových komentářů.

2.2 Syntaktický a sémantický analyzátor, tabulka symbolů

Syntaktický a sémantický analyzátor je implementován ve dvou modulech – modul `parser.c` zpracovává syntaktickou a sémantickou analýzu bez výrazů, které jsou implementovány v modulu `expressions.c`.

Syntaktická analýza je založena na LL-pravidlech LL-gramatiky, je implementována ve formě rekurzivního sestupu. Syntaktická analýza využívá tabulku symbolů, do které si ukládáme ID a parametry funkcí. Více o tabulce symbolů je popsáno v kapitole 3.1.

Zpracování výrazů je implementováno pomocí precedenční analýzy. Při implementaci tabulky jsme využili dvourozměrné pole s hodnotami podle znaků a precedenční tabulku si pro své potřeby v kódu zjednodušili

v případech `+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=` a `==` `!=`. Jako pomocná datová struktura nám v `expressions.c` sloužil zásobník.

Hlavním úkolem syntaktického analyzátoru je kontrolovat syntaktickou správnost kódu a komunikovat s lexikálním analyzátozem, od kterého si parser žádá nové tokeny pro následné zpracování. Sémantický analyzátor dále kontroluje sémantickou správnost kódu. Nutno však podotknout, že syntaktická analýza, sémantická analýza a generování cílového kódu se prolínají.

Syntaktická a sémantická analýza začínají funkcí `analysis`. Tato funkce si od lexikální analýzy vyžádá první token a inicializuje strukturu `aData`, která nese důležité součásti syntaktické a sémantické analýzy jako tabulky symbolů, tabulku funkcí a důležité flagy. Do tabulky funkcí jsou rovněž předem přidány vestavěné funkce. Dále je pak volána funkce `ListUntilToken` s důležitým parametrem `ender`, který určuje ukončující typ lexému, funkce je proto poprvé volána s typem enderu `LEX_EOF`. Tato funkce očekává svůj ukončovač (`ender`) nebo konec řádku, potom je tato funkce volána znova rekurzivně pro zpracování dalšího řádku a nebo očekává `Item` (`if-else` konstrukci, `while` cyklus, definici funkce, atd...) a po zpracování itemu je tato funkce opět rekurzivně volána na zpracování dalšího řádku. Na zpracování jednotlivých konstrukcí jazyka IFJ18 jsou z funkce `Item` volány další funkce jako například `Term`, `FirstParameter` a `ParametersUntilToken` (pro zpracování jednotlivých parametrů funkce), `FunctionCall` (pro zpracování volání funkce), `VariableAssign` (pro zpracování přiřazení do proměnné) a `IdSwitch` (pro komunikaci se sémantickou analýzou ohledně rozpoznání ID proměnné a ID funkce). Výrazy jsou zpracovávány funkcí `Expression` ze samostatného modulu.

Velkým problémem v rámci syntaktické a sémantické analýzy byla skutečnost, že volání funkce (pokud se jednalo o volání z definice jiné funkce) mohlo být umístěno lexikálně před svou vlastní definicí. Pro tento případ jsme museli implementovat vlastní dodatečné řešení. Pokud se nacházíme v definici funkce a je zde použito ID takovým způsobem, že se buď jedná o volání nedefinované funkce a nebo o výraz skládající se z jedné nedefinované proměnné, nemůžeme hned s jistotou ukončit analýzu sémantickou chybou. Přistoupíme k tomuto identifikátoru jako k volání funkce, vložíme tuto funkci do tabulky funkcí, rovněž do tabulky funkcí vložíme počet jejich parametrů a nastavíme flag `firstCalled` na hodnotu `true`. Funkce se tímto již tváří jako definovaná, abychom v rámci dalších volání nemuseli ukončovat analýzu se sémantickou chybou. Jakmile narazíme na její skutečnou definici, spočítáme její počet parametrů, porovnáme s původním voláním a pokud vše souhlasí, flag `firstCalled` nastavíme na `false`. Funkce je tímto korektně definována a dál se již může volat i z hlavního těla programu. Těsně před koncem syntaktické-sémantické analýzy zkontrolujeme celou tabulku funkcí. Pokud se zde nachází nějaká funkce s flagem `firstCalled` nastaveným na `true`, můžeme ukončit analýzu se sémantickou chybou.

2.3 Generátor cílového kódu

Úkolem generátoru kódu, který je implementován v modulech `generate.c` a `stack.c`, je přeložit zpracovávané informace do cílového jazyka – IFJcode18.

Skládá se z funkcí obsahující tisk předpřipravených šablon, které jsou pojmenovány dle větví, ze kterých jsou volány z `parser.h` (například `gen_While()`, ...). Protože se jedná o syntaxi řízený překlad, kód ukládá do datových struktur `dString` a v případě správné syntaktické a sémantické analýzy pak uložený kód tiskne na výstup.

V `expression.h` jsou informace ukládané na zásobníky operací a lexémů s jejich hodnotami, které pak po ukončení výrazů zpracuje funkce `gen_Expression()`. Pokud je zásobník operací prázdný, uloží hodnotu dle jejího správného typu, jinak se dostává do cyklu `while`, dokud zásobník operací (a tím pádem i hodnot) nevyprázdní.

2.4 Testování

Testování probíhalo pomocí programů, které posílaly na vstup naše definované vstupní kódy v jazyce IFJ18.

Testování lexikálního analyzátoru probíhalo pomocí automatického programu v jazyce C. Zkontroloval názvy stavů a správné typy tokenů. Vymysleli jsme testovací případy, které se snaží analyzovat různé vstupní kódy. Každý testovací případ má v sobě celou řadu různých lexémů a program očekává jejich přesnou posloupnost. Testování syntaktického a sémantického analyzátoru probíhalo pak pomocí automatického skriptu v shellu.

Testování bylo dále prováděno i prostřednictvím ručně psaných vstupů za běhu programu, které většinou přímo souviseli s aktuálně implementovaným problémem. Testování nám pak pomohlo v zkontrolování našeho projektu, stalo se nedílnou součástí finální kontroly.

3 Vybrané algoritmy použité při řešení

3.1 Tabulka symbolů

Tabulka symbolů je základní datovou strukturou využívanou syntaktickou analýzou. Je implementována ve formě tabulky s rozptylovací funkcí (tzv. hashovací tabulky) v modulu `syntable.c`. Je využito hashovací funkce `sdbm`, se kterou se pracovalo již v projektu do předmětu Jazyk C (IJC). V tabulce symbolů si ukládáme ID a parametry funkcí, data si neukládáme z důvodu toho, že pracujeme s dynamicky typovaným jazykem.

Mapovací funkce `sdbm` má předpis:

```
hash(i) = hash(i - 1) * 65599 + str[i];
```

Funkce je rychlá a má dobrý rozptyl. Magická konstanta 65599 byla vybrána zcela náhodně. [1]

3.2 Zásobník

Pomocnou abstraktní datovou strukturu zásobníku jsme využili při precedenční analýze výrazů. Prakticky jsme se při konstrukci řídili případem pseudokódu z přednášky.

3.3 Dynamický řetězec

Dynamický řetězec se nachází v modulu `dstring.c`. Je to struktura, která má v sobě dynamicky alokované pole charů, potom svoji velikost a svoji aktuální délku. K práci s dynamickým řetězcem nám slouží řada funkcí: `dStringInit` (inicializace dynamického řetězce), `dStringResize` (změna velikosti dynamického řetězce), `dStringAppend` (přidání symbolu na konec dynamického řetězce s případným rozšířením řetězce, pokud jeho velikost nedostačuje), `dStringCopy` (zkopírování dat z jednoho dynamického řetězce do druhého s případným rozšířením druhého řetězce), `dStringClear` (vymazání dat dynamického řetězce) a `dStringFree` (uvolnění paměti dynamického řetězce).

4 Práce v týmu

Na projektu jsme začali pracovat již na začátku října, věděli jsme, že není dobré ho nechávat na poslední chvíli. Při spolupráci jsme používali verzovací systém GitHub a pravidelně jsme se scházeli. Práci jsme si rozdělili následovně:

- Jakub Frejlich – lexikální analýza, tabulka symbolů, sémantická analýza, celková koordinace projektu a testování
- Kateřina Fořtová – lexikální analýza, návrh syntaktické analýzy, dokumentace, prezentace
- Tibor Škvrnda – generátor cílového kódu
- Lukáš Licek – návrh a implementace syntaktické analýzy

Zohledňovali jsme pokročilost programování v jazyce C, zájmy členů týmu a snažili se pomáhat ostatním v týmu, když bylo třeba.

5 Závěr

Závěrem dokumentace bychom chtěli poukázat na přínos, jaký pro nás projekt měl. Zdokonalili jsme si nejen programovací schopnosti, ale hlavně si zkusili spolupráci v týmu. Naučili jsme se společně řešit problémy a komunikovat.

Reference

- [1] YIGIT, O. *Hash functions* [online]. Poslední změna 22. září 2003 [cit. 30. listopadu 2018]. Dostupné na: www.cse.yorku.ca/~oz/hash.html.

Dále nám byly při řešení projektu nápomocny přednášky a slidy z předmětů IFJ a IAL.

6.1 Graf konečného automatu lexikálneho analyzátoru



6.2 LL-gramatika

1: ANALYSIS → **ListUntilToken** (EOF)

ListUntilToken (ender)

2: ListUntilToken → ender

3: ListUntilToken → EOL **ListUntilToken**(ender)

4: ListUntilToken → **Item ListUntilToken**(ender)

5: Term → ID

6: Term → STRING

7: Term → INT

8: Term → DOUBLE

9: Term → NIL

FirstParameter (ender)

10: FirstParameter → ender

11: FirstParameter → **Term ParametersUntilToken**(ender)

ParametersUntilToken (ender)

12: ParametersUntilToken → ender

13: ParametersUntilToken → COMMA **Term ParametersUntilToken**(ender)

14: FunctionCall → L_BRACKET **FirstParameter**(R_BRACKET) EOL

15: FunctionCall → **FirstParameter**(EOL)

16: ITEM → DEF ID L_BRACKET **ParametersUntilToken**(R_BRACKET) EOL **ListUntilToken**(END) EOL

17: ITEM → IF **EXPRESSION** THEN EOL **ListUntilToken**(ELSE) EOL **ListUntilToken**(END) EOL

18: ITEM → WHILE **EXPRESSION** DO EOL **ListUntilToken**(END) EOL

19: ITEM → **IdSwitch**

20: ITEM → **EXPRESSION** EOL

21: IdSwitch → ϵ

22: IdSwitch → ID **FunctionCall**

23: IdSwitch → ID ASSIGN **VariableAssign**

24: IdSwitch → **EXPRESSION** EOL

25: VariableAssign → ID **FunctionCall**

26: VariableAssign → **EXPRESSION** EOL

6.3 LL-tabulka

	EOF	EOL	ID	STRING	INT	DOUBLE	NIL	COMMA	L_BRACKET	R_BRACKET	DEF	ID	END	IF	THEN	ELSE	WHILE	DO	ASSIGN	ϵ	EXPR.
ANALYSIS	2	3									16	22		17			18			21	20
ListUntilToken	2	3									16	22	2	17		2	18			21	20
Term			5	6	7	8	9														
FirstParameter		10	5	6	7	8	9			10											
ParametersUntilToken		12						13		12											
FunctionCall		10	5	6	7	8	9		14												
ITEM											16	22		17			18			21	20
IdSwitch												22								21	24
VariableAssign												25									26

22/23 – výpomoc sémantikou!

6.4 Precedenční tabulka

	+	-	*	/	<	>	<=	>=	==	!=	()	i	\$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<					>	>	<	>	<	>
>	<	<	<	<					>	>	<	>	<	>
<=	<	<	<	<					>	>	<	>	<	>
>=	<	<	<	<					>	>	<	>	<	>
==	<	<	<	<	<	<	<	<			<	>	<	>
!=	<	<	<	<	<	<	<	<			<	>	<	>
(<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
i	>	>	>	>	>	>	>	>	>	>		>		>
\$	<	<	<	<	<	<	<	<	<	<	<		<	

Komentář k precedenční tabulce:

- $E \rightarrow E + E$
- $E \rightarrow E - E$
- $E \rightarrow E * E$
- $E \rightarrow E / E$
- $E \rightarrow E < E$
- $E \rightarrow E > E$
- $E \rightarrow E \leq E$
- $E \rightarrow E \geq E$
- $E \rightarrow E == E$
- $E \rightarrow E != E$
- $E \rightarrow (E)$
- $E \rightarrow i$
- $E \rightarrow ()$

- \$ – dno zásobníku
- i – může být ID, INT, DOUBLE, STRING nebo NIL
- záhlaví - řádek – vstupní token
- záhlaví - sloupec – terminál na zásobníku