FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Dokumentácia – IFJ 2018

Tím 40, varianta II

Adam Hostin	xhosti02	25 %
Sabína Gregušová	xgregu02	25 %
Dominik Peza	**xpezad00**	**25 %**
Adrián Tulušák	xtulus00	25 %

Obsah

1 Úvod

Naším cieľom je implementovať prekladač imperatívneho jazyka IFJ18 do predmetov IFJ a IAL v jazyku C. Hlavnou náplňou práce bola implementácia: lexikálneho analyzátora, parsera (syntaktická a sématická analýza) a generátora inštrukcií.

2 Lexikálna analýza

Na začiatku sme implementovali lexikálny analyzátor v súbore `lexer.c`, ktorého základom je deterministický konečný automat (ďalej iba DKA). Hlavnou funkciou v tomto súbore je `get_next_token`, ktorá číta jednotlivé znaky a pomocou príkazu `switch` prechádza do nasledujúcich stavov podľa DKA až kým nevyhodnotí lexikálne správny token, inak vracia `ER_LEX`. Lexikálny analyzátor musí mať nadstavený vstupný súbor, ktorý obsahuje program napísaný v jazyku IFJ18. Pri úspešnom vyhodnotení tokenu sa správne uvoľní všetká alokovaná pamäť. Matematické a relačné operátory sú vyhodnotené vcelku rýchlo a jednoducho, identifikátory, reťazce a čísla vyžadujú viacej prechodov a používajú dynamický reťazec, o ktorom ďalej pojednáva sekcia Spracovanie reťazcov. Pri identifikátore sa kontrolujú povolené znaky na základe pozície v reťazci a na záver sa identifikátor porovná so všetkými kľúčovými slovami, ak sa nájde zhoda, je to kľúčové slovo, inak je to identifikátor. Reťazce sú ohraničené dvojitými úvodzovkami (") a môžu obsahovať escape sequence. Pre tento prípad existuje špeciálny stav `STATE_BACKSLASH_LITERAL`, do ktorého sa prechádza pri prečítaní znaku \ a čaká sa na skratku escape sequence, ako napríklad t, s alebo n.

2.1 Štruktúra Token_t

Pre jednoduchšiu prácu s tokenmi sme použili štruktúru `Token_t`, ktorá obsahovala:

- **union Token_attr**
- **struct Token_type**

Union Token_attr obsahuje možné atribúty tokenu, konkrétne to sú: string, integer, flt a keyword.
Struct Token_type obsahuje typy tokenov, konkrétne to sú: `EOF`, `EOL`, identifikátor, kľúčové slovo, relačné a matematické operátory, ľava a pravá zátvorka, čiarka, komentár, int, float a string.

2.2 Spracovanie reťazcov

Pre jednoduchšie spracovanie reťazcov sme sa rozhodli implementovať súbor `dynamic_string.c`. Jeho súčasťou je aj štruktúra `string_t`, ktorá obsahuje samotný ukazateľ na dynamický reťazec, súčastnú veľkosť reťazca a celkovú veľkosť bufferu. Na začiatku je alokovaný reťazec s veľkosťou 10 a pri každom pridaní znaku sa kontroluje, či je ešte v reťazci miesto. Keď sa blížime k zaplneniu reťazca, funkcia `check_empty_bites` zväčší veľkosť buffera o 5, čím zaistí adekvátnu veľkosť pre reťazec. Všetky alokácie pamäte sú kontrolované a ich zlyhanie je adekvátne ošetrené vrátením internej chyby `ER_INTERNAL`.

3 Syntaktická analýza

3.1 Precedenčná analýza výrazov

Syntaktická analýza výrazov je implementovaná v súbore `expression.c` pomocou precedenčnej tabuľky. Hlavné telo `expression.c` pozostáva z funkcie `handle_expression`, ktorá postupne spracováva jednotlivé tokeny alebo symboly a vyhodnocuje ich syntaktickú správnosť. Keďže nie je vždy úplne jednoznačne možné určiť, či nasledujúci token bude súšasťou výrazu alebo nie, rozhodli sme sa implementovať jednosmerne viazaný zoznam s pracovným názvom `buffer`, kam sa postupne ukladajú tokeny vždy na koniec zoznamu.

Ak parser vyhodnotí, že dané tokeny nie sú súšasťou výrazu, vyčistí buffer a a syntaktická analýza pokračuje v parseri. V opačnom prípade je zavolaná funkcia `handle_expression`, ktorá pri spracovaní používa tokeny v bufferi, až kým nie je buffer prázdny a ďalej si žiada tokeny pomocou funkcie `get_next_token`. Z každého

tokenu si vytvoríme symbol na základe jeho typu, ktorý si ďalej ukladáme na zásobník. Spracovávanie symbolov je naprogramované na základe algoritmu uvedeného v prezentácii [1]. Symboly majú svoj status, ktorý sa kopíruje aj pri redukcii danej časti výrazu. Časti výrazu sa postupne dávajú na zásobník generátora a akonáhle už raz bol daný výraz daný do generátora, jeho status sa mení na `ON_GENERATOR_STACK` a pri ďalšej redukcii už nebude znova pridaný na zásobník generátora.

Pri spracovaní výrazov máme flag `return_code`, ktorý môže nadobúdať hodnotu:

- **EXPRESSION_OK** - výraz má správnu syntax aj sématiku
- **SYNTACTICAL_ERRORS** - výraz má syntaktickú chybu
- **UNDEFINED_ID_EXPRESSION** - výraz má sématickú chybu
- **ER_INTERNAL** - interná chyba (neúspešný malloc, neúspešné pridanie do zásobníku a pod.)

Pri kontrole jednotlivých symbolov na zásobníku sa zároveň kontroluje aj tento flag, a jeho nadstavenie na čokoľvek iné ako `EXPRESSION_OK` vedie na ukončenie spracovávania výrazov s adekvátnym návratovým kódom.

3.2 Sématická analýza

Sématickú analýzu sme implementovali ako tabuľku s rozptýlenými položkami. Synonymá sú v tabuľke zoradené explicitne, čo zabezpečuje tereticky neobmedzený počet položiek uchovávateľných tabuľkou symbolov. Synonymá sú zreťazené v jednosmerne viazaných zoznamoch. Veľkosť mapovacieho poľa sme vyberali tak, aby bola rovná prvočíslu. Naša tabuľka má veľkosť 6421. Očakávame, že naplnenie tabuľky nepresiahne 75%. Mapovaciu funkciu sme prevzali z druhej domácej úlohy z IAL, lebo nám pripadala efektívna a ľahko pochopiteľná.

Funkcia spočítava ASCII hodnotu jednotlivých znakov kľúča a nakoniec vracia modulo veľkosti tabuľky z daného súčtu. Každá položka tabuľky obsahuje svoj vlastný unikátny kľúč, ktorý sa ukladá v podobe reťazca. Kľúče značia identifikátory funkcii a premenných. Každá položka taktiež obsahuje svoj typ, značiaci či sa jedná o premennú alebo funkciu, boolovskú hodnotu, určujúca či bola položka definovaná, ukazateľ na ďalší prvok v zozname a integer značiaci počet parametrov v prípade, že sa jedná o funkciu. Položky sa ukladajú do 2 idetických tabuliek s rozptýlenými položkami v závislosti od toho, či sa jedná o lokálne alebo globálne premenné. Všetky funkcie sa ukladajú do globálnej tabuľky symbolov. Implementovali sme taktiež niekoľko funkcii zabezpečujúce pohodlnú prácu s tabuľkou. Funkcie zabezpečujú inicializáciu, výpis chybových hlásení, pridávanie prvkov, vyhľadávanie v tabuľke, kontrolu jednotlivých atribútov položiek a čistenie tabuľky.

4 Generovanie inštrukcií

5 Práca v tíme

Náš tím sme si zostavili pomerne skoro. Po rozdelení práce na menšie celky sme začali každý pracovať na časti pridelenej vedúcim tímu. Približne 3 týždne pred pokusným odovzdaním sme začali jednotlivé časti spájať do celku. Pokusné odovzdávanie sme využili, no výsledok nás nemilo prekvapil. Zistili sme, že hoci všetko fungovvalo pomerne správne, návratové hodnoty sme vždy ”natvrdo”vracali buď ako 0 alebo 1, čo sa aj prejavilo na celkovom percentuálnom hodnotení. Pri tejto príležitosti sme spravili riadnu revíziu kódu a poopravovali čo najviac chýb a nedostatkov.

5.1 Komunikácia

Už na začiatku sme sa dohodli na pravidelných týždenných stretnutiach, kde sme diskutovali o našej ďalšej práci na nadchádzajúci týždeň. Komunikovali sme najmä cez facebook a skype a osobné stretnutia boli veľmi príjemné.

5.2 Verzovanie

Pre správu projektu sme používali verzovací systém Git a vzdialený repozitár GitHub. Tento spôsob správy projektu nám umožňoval pracovať na viacerých častiach projektu súčasne. Po otestovaní jednotlivých podčastí sme súbory začali spájať do celku a každý mal možnosť testovať projekt už ako celok.

5.3 Hodnotenie

Celkovo hodnotíme tento projekt kladne, hoci na začiatku vyzeral pomerne zložito, postupne sme mali nápady ako dané problémy vyriešiť a nakoniec sme všetko stihli v časovom predstihu, takže sme mali možnosť odstraňovať chyby a testovať správnosť a určite tento projekt prispel k zlepšeniu našim programátorských schopností.

Použitá literatúra

[1] MEDUNA, A.; LUKÁŠ, R.: Kapitola VIII. Syntaktická analýza zdola nahoru, rev. 1. septembra 2018, [vid. 3. novembra 2018].

A Deterministický konečný automat

B LL-gramatika

1. <prog> -> DEF ID_FUNC (<params>) EOL <statement> END <prog>
2. <prog> -> EOL <prog>
3. <prog> -> EOF
4. <prog> -> <statement> <prog>
5. <statement> -> IF <expression> THEN EOL <statements> ELSE EOL <statements> END <p
6. <statement> -> WHILE <expression> DO EOL <statement> END EOL
7. <statement> -> <function> EOL
8. <statement> -> ID EOL
9. <statement> -> ID <declare> EOL
10. <statement> -> EOL <prog>
11. <declare> -> = <value>
12. <declare> -> = <expression>
13. <declare> -> = <function>
14. <declare> -> ε
15. <params> -> ID <param>
16. <params> -> ε
17. <param> , ID <param>
18. <param> -> ε
19. <argvs> -> <value> <arg>
20. <argvs> -> ε
21. <arg> -> , <value> <arg>
22. <arg> -> ε
23. <value> -> INT_VALUE
24. <value> -> FLOAT_VALUE
25. <value> -> STRING_VALUE
26. <value> -> ID
27. <function> -> PRINT (<argvs>) EOL
28. <function> -> LENGTH (<argvs>) EOL
29. <function> -> SUBSTR (<argvs>) EOL
30. <function> -> ORD (<argvs>) EOL
31. <function> -> CHR (<argvs>) EOL
32. <function> -> INPUTS EOL
33. <function> -> INPUTI EOL
34. <function> -> INPUTF EOL
35. <function> -> ID_FUNC (<argvs>)

C Precedenčná tabuľka

	+	-	*	/	(	)	i	R	$
+	>	<	<	<	<	>	<	>	>
-	>	>	<	<	<	>	<	>	>
*****	>	>	>	>	<	>	<	>	>
/	<	>	>	>	<	>	<	>	>
(	<	<	<	<	<	=	<	<	
)	>	>	>	>		>		>	>
i	>	>	>	>		>		>	>
R	<	<	<	<	<	>	<		>
$	<	<	<	<	<		<	<	