



Dokumentácia projektu z predmetov IFJ a IAL
Implementácia prekladača imperatívneho jazyka
IFJ18

Tým 011, varianta I

5. december 2018

Filip Bali	(xbalif00)	27%
Natália Holková	(xholko02)	27%
Matej Novák	(xnovak2f)	20%
Albert Szöllösi	(xszoll02)	26%

1 Lexikálna analýza

Základom lexikálnej analýzy je konečný stavový automat, ktorý podľa prijatých znakov rozhoduje, o akú lexikálnu jednotku sa jedná. Túto jednotku označujeme ako token a je definovaná v súbore `scanner.h` ako štruktúra `Token`. Táto štruktúra sa skladá z dvoch položiek: `Token_Type type` a `char *attribute`. `Token_Type` je iba iné označenie pre vymenovanie všetkých možných druhov tokenov. Položka `attribute` je reťazec predstavujúci hodnotu, ktorú je pri niektorých druhov tokenov potrebné uchovať, napríklad prípade tokenu typu `IDENTIFIER` samotný identifikátor.

V súvislosti s lexikálnou analýzou sme nadefinovali viacero pomocných funkcií, avšak jadro tvorí funkcia `get_next_token`, ktorá určí a vráti na nasledujúci token. Práve v tejto funkcii je ako jeden veľký switch realizovaný nami navrhnutý konečný automat. Switch sa nachádza vnútri cyklu, ktorý funkciou `getc()` načítava po znaku zo vstupu. Z cyklu sa vyjde až pri načítaní EOF alebo v prípade nastavenia tokenu.

Scanner využíva štruktúru dynamického reťazca (štruktúra `Tstring`). Počiatočná dĺžka reťazca sa alokuje pri jeho inicializácii. Ak dôjde k presiahnutiu tejto dĺžky, maximálna možná dĺžka sa zvýši pomocou funkcie `realloc`.

Kvôli zjednodušeniu neskoršej práce s literálmi typu `float` pri generovaní inštrukcií je priamo v scanneri realizovaný prevod rôznych pôvolených tvarov datového typu `float` na jednotný tvar, ktorý je akceptovaný interpretom. Tento tvar je potom uložený ako atribút tokenu.

2 Tabuľky symbolov

Tabuľku symbolov sme implementovali ako binárny vyhľadávací strom. Rozlišovali sme dva rôzne typy tabuľky symbolov: globálnu, kde sme do jednotlivých uzlov stromu ukladali potrebné informácie o definovaných funkciách a lokálnu, kde sme naopak vkladali premenné, ktoré sa vyskytovali v danej funkcii.

Pretože jazyk zadaný jazyk nepodporoval globálne premenné, premenné vytvorené v hlavnom tele programu sa priradzovali do špeciálnej lokálnej tabuľky, ktorá prislúchala natvrdo vytvorenému uzlu s identifikátorom `main@function` v globálnej tabuľke symbolov.

Globálna a lokálna tabuľka symbolov boli implementované ako dve rozdielne štruktúry, pretože sa líšili v tvare dát, ktoré boli potrebné uchovávať v uzloch stromov. Týmto prístupom sa síce výrazne zvýšila veľkosť zdrojového súboru implementujúceho tieto štruktúry, ale v konečnom dôsledku sa nám lepšie s nimi pracovalo vďaka zvýšenej zrozumiteľnosti.

Uzly v globálnej tabuľke symbolov majú uchovávať názov funkcie a v štruktúre `tDataNodeGlobal` informáciu, či bola funkcia už definovaná, počet parametrov funkcie, lineárny zoznam s identifikátormi parametrov a odkaz na lokálnu tabuľku symbolov tej funkcie.

Lokálna tabuľka symbolov v jednotlivých uzloch naopak uchovávala názov premennej a v štruktúre `tDataNodeLocal` informácie o aktuálnom datovom type premennej a či už bola definovaná.

Binárny strom sme implementovali rekurzívnym prístupom a pri jeho programovaní sme sa inšpirovali vlastným riešením druhej domácej úlohy z predmetu IAL. Pre oba typy tabuliek sme vytvorili základné funkcie `init` – inicializácia stromu, `insert` – vloženie nového uzlu, `search` – vyhľadanie uzlu na základe identifikátora, `delete` s pomocou funkciou `replace_by_rightmost` – vymazanie uzlu a `dispose` – uvoľnenie celej tabuľky. Tieto základné funkcie sa využívali v špecializovaných funkciách pre kontrétny typ tabuľky symbolov. V porovnaní so základnými funkciami boli špeciálne výrazne jednoduchšie na používanie.

V prípade globálnej tabuľky sme využívali funkcie: `function_add_param_id_to_list` (vloženie identifikátora parametru do zoznamu), `function_get_number_params`, `function_increase_number_param`, `function_set_number_params` (práca s počtom parametrov funkcie), `function_set_defined` (nasta-

venie/vytvorenie uzlu s funkciou ako definovanej), `get_function_node` (získanie uzla s premennou) a `set_function_table` (nastavenie odkazu na lokálnu tabuľku).

Pri lokálnej tabuľke to boli zasa: `variable_set_defined` (nastavenie premennej ako definovanej), `variable_set_type` (nastavenie typu premennej), `variable_get_type` (získanie typu premennej) a `get_variable_node` (získanie uzla s premennou).

3 Syntaktická analýza

Syntaktická analýza sa, až na analýzu výrazov, riadi LL-gramatikou a je implementovaná metódou rekurzívneho zostupu podľa LL-tabuľky.

Pre každý neterminál je vytvorená vlastná funkcia, ktorá kontroluje pravidlá, ktoré sa v danom stave môžu uplatniť.

Syntaktický analyzátor postupne, pomocou funkcie `get_next_token`, načítava tokeny (získané lexicálnym analyzátorom zo vstupného kódu) do parametru token, ktorý je typu ukazovateľ na Token. Každá funkcia hľadá pravidlo, ktoré sa s daným tokenom na vstupe dá uplatniť. V prípade, že sa v aktuálnom stave (funkcii) pre daný token nenachádza uplatniteľné pravidlo, nastáva syntaktická chyba a program končí s návratovou hodnotou 2 (`ERR_SYNTAX`). V opačnom prípade sa podľa LL-tabuľky uplatní dané pravidlo. Program sa takto postupne rozdeľuje na tzv. statementy, ktoré musia byť oddelené znakom konca riadku (Pravidlo 2 v LL-gramatike). V prípade, že sa postupným uplatnením pravidiel prejde do koncového stavu, znamená to, že statement je syntakticky správny, daná funkcia vráti návratovú hodnotu 0 (`ERR_OK`) a syntaktická analýza pokračuje na ďalší statement.

Ak syntaktický analyzátor narazí na výraz, odovzdá riadenie analyzátoru výrazov. Ak je výraz v poriadku, analýza pokračuje ďalej. Ak narazí na token `EOF` a všetky statementy sú ukončené, znamená to, že prekladaný kód je syntakticky správny a syntaktická analýza vráti návratovú hodnotu 0 (`ERR_OK`).

4 Parser výrazov

Pri volaní, výrazový parser dostáva v parametri posledný načítaný token z hlavného parseru programu. Následne výrazový parser spracováva všetky tokeny pomocou volania funkcie `get_next_token` až pokým získaný token nie je buď kľúčové slovo `do` alebo `then` prípadne pokým nie je koniec riadku (`EOL`) alebo koniec súboru (`EOF`). Takto získané tokeny postupne počas načítavania uklada do dvojsmerného zoznamu `ExprArray`. Počas načítavania sú vyhodnocované návratové hodnoty zo scanneru v prípade že ohlásí chybu.

Následne je volaná funkcia `MainSyntaxCheck` kde sú alokované a uvoľnené potrebné zdroje k ďalšej práci s výrazom.

Prvá kontrola výrazu sa vykonáva vo funkcii `FindRule` ktorá je volaná vo funkcii `MainSyntaxCheck`. `FindRule` zisťuje podľa syntaktickej tabuľky či je možné aby tokeny nasledovali v takom poradí ako boli prijaté do scanneru. Zároveň sa počíta počet jednotlivých typov tokenov, ktorý je využitý v ďalšej syntaktickej kontrole kde sa zamedzuje jednotlivým syntaktickým chybám ktoré nie sú možné vyzistiť iba s poradia jednotlivých primaných tokenov. Ako príklad môže poslúžiť počet pravých a ľavých zátvoriek vo výraze a ich správne použitie, čiže nemôže nasledovať pravá zátvorka skôr ak nemá svoj pár k ľavej zátvorke a podobne.

Ak syntaktická kontrola je neúspešná, to znamená bola vygenerovaná syntaktická chyba, ďalšie úlohy výrazového parseru sú zrušené a príslušný chybový kód je vrátený hlavnému parseru programu.

Ak je syntaktická kontrola úspešná, potom je vo funkcii `MainSyntaxCheck` volaná funkcia `ParseToPostfix` kde je výraz spracovaný do postfixovej notácie a uložený v presnom poradí do výsledného zásobníka. Následne je vo `MainSyntaxCheck` volaná posledná funkcia `EvaluateFromPostfix` kde je vykonané

spracovanie výrazu pomocou precedentnej tabuľky kde sa usporiada výraz podľa priorít v precedentnej tabuľke tak aby mohol byť korekne vyhodnotený a takto usporiadaný výraz je uložený do výsledného zásobníka.

Pre samotné vyhodnotenie jednotlivých operácií si funkcia `EvaluateFromPostfix` vola funkciu `EvaluateNow` kde sú vykonané jednotlivé operácie, respektíve generovanie medzikódu.

5 Sémantická analýza

Sémantická analýza bola riešená priamo v súbore `parser.c` a `expression_parser.c`. V nich sa na príslušných miestach vykonávali vhodné kontroly a sémantické akcie. Vo vlastnom súbore `semantic_analysis` sú implementované funkcie, ktoré pomáhajú pri sémantickej analýze. Celá sémantická analýza veľmi úzko spolupracuje s tabuľkami symbolov.

V hlavnom parseri sa kontroluje možná redefinícia pri definovaní novej funkcie, ktorá sa riešila prehľadávaním globálnej tabuľky symbolov pred pridaním nového uzla. Pri volaní funkcie sa kontroluje, či bola predtým definovaná a v prípade, že áno sa taktiež dáva pozor na správny počet argumentov pri volaní. Pokiaľ je argumentom premenná, musí byť vopred definovaná.

Hlavnou úlohou sémantickej analýzy v parseri výrazov je skontrolovať typovú kompatibilitu výrazov a odhaliť nedefinované premenné. Toto celé sa vykonáva v súbore `expression_parser.c` vo funkcii `EvaluateNow`. Spracovávajú sa vždy dva operandy a operátor v posfixovej notácii.

Kompatibilita typov sa rieši buď pomocou funkcie `arithmetic_check_compatibility` alebo `comparison_check_compatibility` v závislosti na type operátora.

6 Generovanie kódu

Vygenerované inštrukcie sme sa rozhodli ukladať vo forme jednosmerného zoznamu s ukazovateľmi na prvý, aktívny a posledný prvok. Tú metódu sme zvolili najmä kvôli generovaniu definícií funkcií a cyklov, kde bolo potrebné niekedy vkladať inštrukcie na iné miesto, než za poslednú inštrukciu. Samotná inštrukcia je reprezentovaná štruktúrou `tInstr`, ktorá sa skladá z typu inštrukcie `tInstruction_type` a až troch ďalších adries vo forme reťazcov.

Na začiatku generovania inštrukcií sa inicializuje zoznam a štruktúra na zapisovanie aktuálnej inštrukcia. Vygeneruje sa hlavička kódu, skok na návestie `$main` a vlastný rámec pre `main`.

Pri definícii užívateľskej funkcie musí najskôr dôjsť k presunu aktívneho prvku zoznamu na začiatok a ďalší posun až za skok do `$main`. Po vytvorení nového rámca a načítaní parametrov zo zásobníka sa postupuje rovnako ako pri generovaní v tele funkcie. Pri konci definície sa uloží návratová hodnota na zásobník, ukončí sa aktuálny rámec, vygeneruje sa `RETURN` a opäť sa zapisujú inštrukcie na koniec zoznamu.

Pri generovaní vetvenia `if-else` sa pri spracovaní riadku zdrojového kódu s `if <expression> then` vygeneruje skok do návestia `$else`, ak nie je výraz vyhodnotený ako `true`. Po skončení príkazov vo vetve `if` sa vygeneruje skok až za koniec spracovania vetvenia a označenie návestia `$else`.

Najväčší problém pri generovaní inštrukcií predstavovalo generovanie cyklu, pretože v prípade inštrukcie `DEFVAR` v tele cyklu by dochádzalo k redefinícii premennej. Toto sme vyriešili odchyťovaním definícií v cykle do samotného zoznamu inštrukcií. Po skončení príkazov patriacich do cyklu sa spätne vložili odchytené deklarácie pred návestie označujúce cyklus.

7 Rozdelenie úloh

Filip	parser výrazov, testy	27%
Natalia	lexikálna a sémantická analýza, tabuľka symbolov, generovanie kódu	27%
Matej	generovanie kódu, dokumentácia	20%
Albert	lexikálna a syntaktická analýza	26%

Tabuľka 1: Rozdelenie práce v tíme

Rozdelenie nie je rovnomerné z dôvodu neschopnosti člena tímu vykonať všetky úlohy, ktoré mu boli zverené.

A LL tabuľka

DEF	IF	ELSE	WHILE	END	EOL	EOF	ID	(,)	INTEGER	FLOAT	STRING	NIL	+	-	*	/	<	>	==	=
<prog>	1	1		1			1	1	1			1	1	1	1							
<stat_list>	2	2		2			3	2	2			2	2	2	2							
<stat>	4	5		6			7	8				8	8	8	8							
<params>							9				10											
<params_next>										11	12											
<if_stat_list>	13	14		13			13	13				13	13	13	13							
<nested_stat_list>	15			15	16		15	15				15	15	15	15							
<nested_stat>	17			18			20	21				21	21	21	21							
<arg_with_brackets>							22			23		22	22	22	22							
<arg_next_with_brackets>										24	25											
<arg_without_brackets>						27	26					26	26	26	26							
<arg_next_without_brackets>						29			28													
<after_id>						33	30	30				30	30	30	30	32	32	32	32	32	32	31
<after_func_call>							34	35				34	34	34	34							
<def_value>							37	36				36	36	36	36							
<value>							38					39	40	41	42							

B Precedenčná tabuľka

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	Operacia	UNARNE	NOT	*	/	div	mod	and	+	-	or	xor	==	<>	<	<=	>	>=	in	()	ID	function	array	,	\$
0	UNARNE	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
1	NOT	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
2	*	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
3	/	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
4	div	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
5	mod	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
6	and	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
7	+	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
8	-	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
9	or	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
10	xor	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
11	==	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	<	>
12	<>	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	<	>
13	<	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	<	>
14	<=	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	<	>
15	>	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	<	>
16	>=	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	<	>
17	in	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	<	>
18	(<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<	=	-
19)	-	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	-	>	-	-	-	-	>
20	ID	-	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	-	>	-	-	-	-	>
21	function	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	=	-	-	-	-	-	-
22	array	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	=	-	-	-	-	-	-
23	,	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<	=
24	\$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	-	<	<	<	<	-

D LL Gramatika

```
01: <prog> -> <stat_list> EOF
02: <stat_list> -> <stat> EOL <stat_list>
03: <stat_list> -> ε
04: <stat> -> DEF ID ( <params> ) EOL <nested_stat_list> END
05: <stat> -> IF <expression> THEN EOL <if_stat_list> ELSE <nested_stat_list> END
06: <stat> -> WHILE <expression> DO EOL <nested_stat_list> END
07: <stat> -> ID <after_id>
08: <stat> -> <expression>
09: <params> -> ID <params_next>
10: <params> -> ε
11: <params_next> -> , ID <params_next>
12: <params_next> -> ε
13: <if_stat_list> -> <nested_stat> EOL <if_stat_list>
14: <if_stat_list> -> ε
15: <nested_stat_list> -> <nested_stat> EOL <nested_stat_list>
16: <nested_stat_list> -> ε
17: <nested_stat> -> IF <expression> THEN EOL <if_stat_list> ELSE <nested_stat_list>
END
18: <nested_stat> -> WHILE <expression> DO EOL <nested_stat_list> END
19: <nested_stat> -> PRINT ( <value> <arg_next> )
20: <nested_stat> -> ID <after_id>
21: <nested_stat> -> <expression>
22: <arg_with_brackets> -> <value> <arg_next>
23: <arg_with_brackets> -> ε
24: <arg_next_with_brackets> -> , <value> <arg_next>
25: <arg_next_with_brackets> -> ε
26: <arg_without_brackets> -> <value> <arg_next>
27: <arg_without_brackets> -> ε
28: <arg_next_without_brackets> -> , <value> <arg_next>
29: <arg_next_without_brackets> -> ε
30: <after_id> -> <after_func_call>
31: <after_id> -> = <def_value>
32: <after_id> -> <expression>
33: <after_id> -> ε
34: <after_func_call> -> <arg_without_brackets>
35: <after_func_call> -> ( <arg_with_brackets> )
36: <def_value> -> <expression>
37: <def_value> -> ID <after_func_call>
38: <value> -> ID
39: <value> -> INTEGER
40: <value> -> FLOAT
41: <value> -> STRING
42: <value> -> NIL
```