Vysoké učení technické v Brně
Fakulta informačních technologií

Formální jazyky a překladače

Dokumentace ke skupinovému projektu

Tým 47, varianta I

Jakub Zich (vedoucí) - 25% *xzichj00*
Patrícia Hudecová - 25% *xhudec30*
Ondřej Marek - 25% *xmarek67*
Tomáš Willaschek - 25% *xwilla00*

1 Struktura projektu

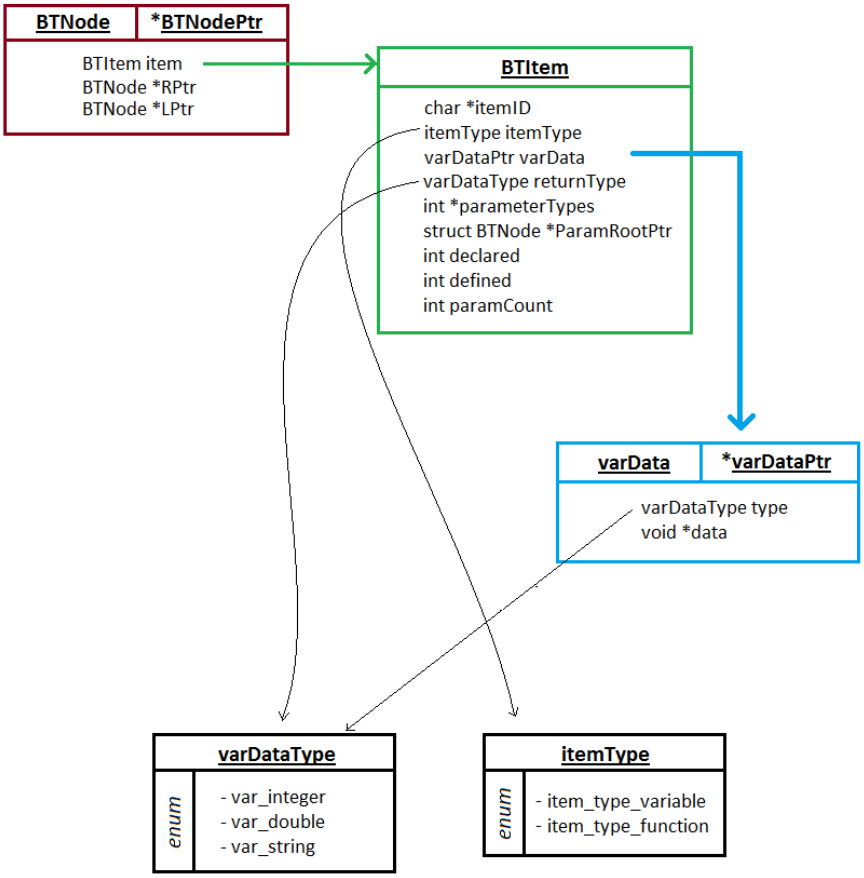
1.1 Lexikální analyzátor

Lexikálny analyzátor (scanner) načítava postupnosť znakov (lexém) zo štandardného vstupu a spracováva ich po jednom pomocou konečného automatu. Konečný automat je naimplementovaný pomocou príkazu switch (v jazyku C), ktorý je súčasťou funkcie getNextToken.

Komentáre a biele znaky sú spracovávané, ale nikam sa neukladajú po ich spracovaní scanner pokračuje v spracovávaní ďalších znakov. Do štruktúry sa ukladajú identifikátory, kľúčové slova, rezervované kľúčové slová, čísla (celé, desatinné a v exponenciálnom tvare), reťazcové literály, operátory a niektoré znaky ako napríklad zátvorky a podobne. Tokeny sú na ďalšie spracovanie posielané do syntaktického analyzátora.

1.2 Tabulka symbolů

Tabulka symbolů je implementována v podobě binárního stromu. Funkce pro vyhledáváni jsou programovány iterativně, protože rekurzivní přístup by byl při větším počtu položek náročný na paměť.



1.3 Syntaktický analyzátor

Syntaktický analyzátor (dále jen SA) pracuje na základě trojrozměrného pole, které reprezentuje LL-gramatiku[1]. První rozměr jsou jednotlivé neterminály, druhý všechny možné pravidla pro jeden

neterminál a třetí je jedno pravidlo vložené do pole.

Tyto pravidla se aplikují od zadu na zásobník. Aplikace je prováděna dle precedenční analýzy shora dolů - neterminál na vrcholu zásobníku se rozloží podle terminálu na vstupu.

SA získává tokeny (terminály) voláním lexikálního analyzátoru. Token se následně porovná s vrcholem zásobníku a když se čísla nerovnají, jedná se o syntaktickou chybu.

Tato konkrétní SA musí rozpoznávat, zda je daný výraz funkce, či skutečný výraz. Proto je objem praviděl poněkud rozsáhlejší.

Kdykoli, když mý být zpracován výraz, volá se spaciální funkce obsažaná přímo v SA, která konvertuje jednotlivé tokeny na dohodnutý string, který následně pošle precedenční analýze ke skontrolování. Kód se ukládá token po tokenu do struktury, která následně slouží pro kontrolu sémantiky a generování kódu. Při získání tokenu EOL se zkontroluje sémantika aktuálního řádku, a pokud je správná, spustí se generování kódu.

1. **S** → declare function **Hlavička** eol **S**
2. **S** → function **Hlavička** eol **Tělo_fce** function eol **S**
3. **S** → scope eol **Tělo_scope** scope
4. **Hlavička** → id (**Parametry** as **Datový_typ**
5. **Parametry** → id as **Datový_typ Parametr_next**
6. **Parametry** →)
7. **Parametr_next** → , **Parametry**
8. **Parametr_next** →)
9. **Tělo_fce** → dim id as **Datový_typ Přiřazení Tělo_fce**
10. **Tělo_fce** → if **Výraz_if** then eol **Tělo_if1** if eol **Tělo_fce**
11. **Tělo_fce** → do while **Výraz** eol **Tělo_while** eol **Tělo_fce**
12. **Tělo_fce** → id = **Přiřazení2 Tělo_fce**
13. **Tělo_fce** → input id eol **Tělo_fce**
14. **Tělo_fce** → print **Print_param Tělo_fce**
15. **Tělo_fce** → return **Přiřazení2 Tělo_fce**
16. **Tělo_fce** → end
17. **Tělo_if1** → dim id as **Datový_typ Přiřazení Tělo_if1**
18. **Tělo_if1** → if **Výraz_if** then eol **Tělo_if1** if eol **Tělo_if1**
19. **Tělo_if1** → do while **Výraz** eol **Tělo_while** eol **Tělo_if1**
20. **Tělo_if1** → id = **Přiřazení2 Tělo_if1**
21. **Tělo_if1** → input id eol **Tělo_if1**
22. **Tělo_if1** → print **Print_param Tělo_if1**
23. **Tělo_if1** → return **Přiřazení2 Tělo_if1**
24. **Tělo_if1** → else eol **Tělo_if2**
25. **Tělo_if2** → dim id as **Datový_typ Přiřazení Tělo_if2**
26. **Tělo_if2** → if **Výraz_if** then eol **Tělo_if1** if eol **Tělo_if2**
27. **Tělo_if2** → do while **Výraz** eol **Tělo_while** eol **Tělo_if2**
28. **Tělo_if2** → id = **Přiřazení2 Tělo_if2**
29. **Tělo_if2** → input id eol **Tělo_if2**
30. **Tělo_if2** → print **Print_param Tělo_if2**
31. **Tělo_if2** → return **Přiřazení2 Tělo_if2**
32. **Tělo_if2** → end
33. **Tělo_while** → dim id as **Datový_typ Přiřazení Tělo_while**
34. **Tělo_while** → if **Výraz_if** then eol **Tělo_if1** if eol **Tělo_while**
35. **Tělo_while** → do while **Výraz** eol **Tělo_while** eol **Tělo_while**
36. **Tělo_while** → id = **Přiřazení2 Tělo_while**
37. **Tělo_while** → input id eol **Tělo_while**
38. **Tělo_while** → print **Print_param Tělo_while**
39. **Tělo_while** → return **Přiřazení2 Tělo_while**
40. **Tělo_while** → loop
41. **Tělo_scope** → dim id as **Datový_typ Přiřazení Tělo_scope**
42. **Tělo_scope** → if **Výraz_if** then eol **Scope_if1** if eol *_Tělo_scope*
43. **Tělo_scope** → do while **Výraz** eol **Scope_while** eol **Tělo_scope**
44. **Tělo_scope** → id = **Přiřazení2 Tělo_scope**
45. **Tělo_scope** → input id eol **Tělo_scope**
46. **Tělo_scope** → print **Print_param Tělo_scope**
47. **Tělo_scope** → end
48. **Scope_while** → dim id as **Datový_typ Přiřazení Scope_while**
49. **Scope_while** → if **Výraz_if** then eol **Scope_if1** if eol **Scope_while**
50. **Scope_while** → do while **Výraz** eol **Scope_while** eol **Scope_while**
51. **Scope_while** → id = **Přiřazení2 Scope_while**
52. **Scope_while** → input id eol **Scope_while**
53. **Scope_while** → print **Print_param Scope_while**
54. **Scope_while** → loop
55. **Scope_if1** → dim id as **Datový_typ Přiřazení Scope_if1**
56. **Scope_if1** → if **Výraz_if** then eol **Scope_if1** if eol **Scope_if1**
57. **Scope_if1** → do while **Výraz** eol **Scope_while** eol **Scope_if1**
58. **Scope_if1** → id = **Přiřazení2 Scope_if1**
59. **Scope_if1** → input id eol **Scope_if1**
60. **Scope_if1** → print **Print_param Scope_if1**
61. **Scope_if1** → else eol **Scope_if2**
62. **Scope_if2** → dim id as **Datový_typ Přiřazení Scope_if2**
63. **Scope_if2** → if **Výraz_if** then eol **Scope_if1** if eol **Scope_if2**
64. **Scope_if2** → do while **Výraz** eol **Scope_while** eol **Scope_if2**
65. **Scope_if2** → id = **Přiřazení2 Scope_if2**
66. **Scope_if2** → input id eol **Scope_if2**
67. **Scope_if2** → print **Print_param Scope_if2**
68. **Scope_if2** → end
69. **Přiřazení** → = **Přiřazení2**
70. **Přiřazení** → eol
71. **Přiřazení2** → id **Id_as**
72. **Přiřazení2** → valueOfX **Id_as**
73. **Přiřazení2** → 299
74. **Id_as** → (**Parametr_fce**
75. **Id_as** → eol
76. **Id_as** → 298
77. **Parametr_fce** → id **Parametr_fce_next**
78. **Parametr_fce** → valueOfX **Parametr_fce_next**
79. **Parametr_fce** →) eol
80. **Parametr_fce_next** → , **Parametr_funkce**
81. **Parametr_fce_next** →) eol
82. **Výraz** → id **Id_as**
83. **Výraz** → 298
84. **Výraz_if** → id **Id_as**
85. **Výraz_if** → 296
86. **Datový_typ** → integer
87. **Datový_typ** → double
88. **Datový_typ** → string
89. **Print_param** → eol
90. **Print_param** → 297 ; **Print_param**
91. **Speciální funkce**

- 297, 296, 298, 299
  - 298 → id je ve struktuře kódu
- výrazy se pohlcují až po oddělovače *then*, *eol* a *;*
- Výraz se sjednotí, upraví a pošle precedenční analýze

Obrázek 1: LL-gramatika

	D	F	S	id	)	,	dim	if	do	I	P	R	end	E	L	=	eol	V	int	str	dbl	(
S	1	2	3																			
Hlavička				4																		
Parametry				5	6																	
Parametr_next					8	7																
Tělo_fce				12			9	10	11	13	14	15	16									
Tělo_if1				20			17	18	19	21	22	23		24								
Tělo_if2				28			25	26	27	29	30	31	32									
Tělo_while				36			33	34	35	37	38	39			40							
Tělo_scope				44			41	42	43	45	46		47									
Scope_while				51			48	49	50	52	53				54							
Scope_if1				58			55	56	57	59	60			61								
Scope_if2				65			62	63	64	66	67		68									
Přiřazení																69	70					
Přiřazení2				71														72				
Id_as																	75					74
Parametr_fce				77	79													78				
Parametr_fce_next					81	80																
Výraz				82																		
Výraz_if				84																		
Datový_typ																			86	87	88	
Print_param																	89					

D = declare — F = function — S = scope — V = valueOfX — I = input — P = print — R = return — E = else — L = loop

Tabulka 1: LL tabulka

1.4 Precedenční analýza

Precedenční analýza je volaná ze syntaktické. Pokud syntaktická analýza najde v kódu výraz, zavolá precedenční, která ho zkontroluje a vrátí syntaktické příslušnou návratovou hodnotu podle toho, zda je výraz syntakticky správně, nebo ne.

Naše verze precedenční analýzy nepracuje s jednotlivými tokeny, které posílá lexikální analyzátor, ale pouze s textovým řetězcem. Všechny identifikátory, ať už se v kódu jmenují jakkoli, jsou poslány precedenčnímu analyzátoru jako i. Tedy syntaktický analyzátor, který má přístup k tokenům, a pozná, kde je začátek a konec výrazu, načte příslušné tokeny, vytvoří z nich string, kde změní jména proměnných na i, a pošle string ke kontrole precedenčnímu. Precedenční vrátí pouze 0, 2 nebo 99, podle toho, zda se někde vyskytne chyba, nebo ne, a syntaktický pak pracuje dále s tokeny, které má už načtené.

Výrazy jsou kontrolovány pomocí precedenční tabulky. V naší verzi precedenčního analyzátoru jsou použity tabulky dvě, jedna hlavní, která kontroluje, jestli po určitém znaku může přijít v pořadí další, a druhá, která porovnává operátory podle priority, či asociativity. Také je použita pomocná datová struktura Zásobník. Obě tabulky by se daly spojit do jedné, v podstatě se stejným výsledkem, ale tato možnost také funguje. Zde přiložená tabulka je jen jedna, spojená dohromady, pro větší přehlednost.

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

1.5 Sémantická analýza

Sémantický analyzátor je podmnožinou syntaktického. Pracuje s jednotlivými řádky vstupního kódu, které jsou poskládány z tokenů předávaných lexikálním analyzátorem. Řádek může začínat slovy Declare, Function, Scope, End, Else, Loop, Return, Dim, Print, Input, If, Do, nebo ID některé proměnné, a končí znakem EOL.

Sémantický analyzátor přečte první token každého načteného řádku. Syntaktická kontrola řádků je prováděna před sémantickou, takže vychází z předpokladu, že jsou řádky syntakticky správně, tedy ví přesně, v jakém pořadí v řádku přicházejí další tokeny (proto stačí znát jen ten první). Pokud je vše tak, jak má být, vkládá na příslušných místech proměnné a funkce do tabulky symbolů. Pokud je některé proměnné přiřazena hodnota, pracuje s ní však generátor. Sémantický analyzátor ji do tabulky nevkládá.

Tabulka symbolů je tvořena následujícím způsobem: v hlavním binárním vyhledávacím stromě jsou vloženy všechny definované funkce. Scope je také vkládána pod jménem "@Scope". Před tím, než začneme sémantickou kontrolu, je nutno vložit do stromu ještě vestavěné funkce Lenght, SubStr, Asc a Chr. Každá funkce má svůj vlastní podstrom, kam se vkládají proměnné, definované v příslušné funkci. Pokud řádek začíná tokenem End, je podstrom příslušné funkce uvolněn. Po kontrole každého řádku vrací sémantický analyzátor náležitou hodnotu zpět syntaktickému, který dále načte další řádek atd.

1.6 Generátor kódu

Generátor, jako jeden ze základních částí překladače, je volán s každým zkontrolovaným řádkem vstupního kódu. Každý takový řádek je předgenerován do dočasné struktury - implementována seznamem. Vypsání na standartní výstup proběhne až po zkontrolování správnosti celého vstupního kódu.

1.6.1 Proměnné

Základní součástí programu jsou proměnné, které jsou implicitně inicializovány na počáteční hodnotu podle datového typu.

Přiřazení výsledku výrazu nebo funkce do proměnné probíhá vyjmutím připravené hodnoty z vrcholu datového zásobníku.

1.6.2 Výrazy

Vyhodnocení výrazů probíhá převedením na tvar s **postfixovou (polskou) notací**. Tuto variantu jsme zvolili kvůli následné implementaci výpočtů s využitím datového zásobníku.

Následně je výraz v tomto tvaru **optimalizován** předpočítáním konstant, čímž se u delších výpočtů značně zkrátí doba provádění programu. Optimalizace probíhá hledáním dvou sousedních konstant ve výrazu a použití příslušeného operátoru.

Optimalizovaný výraz je předán funkci pro zajištění nutných **konverzí datových typů** (ze zadání jazyka IFJ17 a zároveň správné interpretace kódu IFJcode17)
Výsledek výrazu je ponechán na vrcholu datového zásobníku pro další zpracování.

1.6.3 Podmínky a cykly

Základní konstrukce jazyka IFJ17 jako podmínky a cykly jsou generovány pomocí instrukcí podmíněných skoků využívajících datový zásobník.

Návěští se skládají z konstatní části popisující typ konstrukce a z části generované počítadlem.
S každou novou konstrukcí podmínky nebo cyklu se inkrementuje příslušné počítadlo.
Výsledné návěští vypadá např.: IF1, IFEND1 LOOP40, ...

1.6.4 Uživatelské funkce

Programátorem definované funkce se generují s nahlížením do tabulky symbolů, kde jsou nalezeny potřebné informace (návratový typ, datové typy parametrů, identifikátor funkce).

Případná návratová hodnota funkce je ponechána na vrcholu datového zásobníku pro další zpracování. Pokud funkce nevrací žádnou hodnotu, je použita výchozí hodnota podle datového typu. Při volání funkce Je vytvořen nový rámec a přesunut na vrchol zásobníku rámců. Uklizení rámce z vrcholu zásobníku probíha po návratu z funkce.

1.6.5 Vestavěné funkce

Překladač podporuje 4 vestavěné funkce. Tyto funkce jsou vygenerovány v záhlaví každého kódu IFJcode17 se stejným formátem jako uživatelem definované funkce.

1.7 Tabulka symbolů

Tabulka symbolů je implementována v podobě binárního stromu. Funkce pro vyhledáváni jsou programovány iterativně, protože rekurzivní přístup by byl při větším počtu položek náročný na paměť.