Implementace překladače imperativního jazyka IFJ18

Tým 046, varianta II

Ondřej Dacer	xdacer00	25 %
Roland Žitný	xzitny01	25 %
Lubomír Švehla	xsvehl09	25 %
Jakub Novotný	xnovot11	25 %

Obsah

Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načte zdrojový kód v jazyku IFJ18 ze standardního vstupu, jenž je zjednodušenou podmnožinou jazyce Ruby 2.0, a přeloží jej do cílového jazyka IFJcode18 na standardní výstup. Pokud došlo k chybě, vrací se příslušná návratová hodnota dle zadání, jinak se vrací návratová hodnota 0. Jedná se o konzolovou aplikaci bez grafického rozhraní.

1 Návrh a implementace

Projekt je sestaven z několika částí, které spolu komunikují a které popisujeme v této kapitole.

1.1 Tabulka s rozptýlenými položkami (`symtable.c`)

Implementovali jsme tabulku s rozptýlenými položkami dle varianty II. Tabulka využívá explicitně zřetězená synonyma, takže počet položek je neomezený, pro jejich zřetězení jsme využili lineárně vázané seznamy. Jako mapovací funkci jsme využili algoritmus SDBM, který je podle zdrojů [1][2] ve všeobecnosti jedním z nejlepších algoritmů v distribuci položek. Pro výpočet distribucí využívá magickou konstantu 65599 (`hash[i] = hash[i-1] * 65599 + str[i]`), která byla vybrána na základě různých experimentů. Každá položka tabulky obsahuje svůj unikátní klíč, datový typ, elementární typ, který udává, zda se jedná o parametr, proměnnou nebo funkci, svou hodnotu (`int, float, char`) a ukazatel na následovníka v seznamu (synonymum). Následně jsme navrhli funkce potřebné pro práci s tabulkou, a sice inicializaci tabulky (`st_init`), přidání nové položky (`st_insert`), vyhledání položky (`st_search`) a uvolněni tabulky (`st_free`).

1.2 Lexikální analýza (`scanner.c`)

Scanner používaný jako lexikální analyzátor jsme implementovali jako deterministický konečný automat. Je složen z jednoho nekonečně opakujícího se cyklu `while`. Uvnitř cyklu se nachází jeden velký `switch`. V každém případě jsme řešili jiný stav automatu. Scanner načítá znak po znaku a pokud je třeba, ukládá je do bufferu (`attr`). Na základě posloupnosti znaků následně rozpozná token na vstupu anebo znaky ignoruje (v případe bílých znaků a komentářů). Takto přečtené tokeny se během prvního průchodu ukládají do seznamu tokenů. Jako nejtěžší část tvorby scanneru hodnotíme správné rozpoznání a ukončení blokových komentářů.

1.3 Syntaktická analýza (`parser.c`)

Hlavní částí celého překladače je syntaktická analýza, která se řídí metodou rekurzivního sestupu a LL – gramatikou. Naše syntaktická analýza je implementovaná pomocí dvou průchodů. První průchod se stará o uložení všech definicí funkcí do globální tabulky symbolů a o odhalení lexikálních a některých syntaktických chyb. Druhý průchod slouží ke generování kódu cílového jazyka a pro odhalení ostatních chyb (např. syntaktické chyby nebo chyby v počtu parametrů při volání funkcí).

1.4 Sémantická analýza (`expression_parser.c`)

Precedenční analýza je volána jako funkce `expression` ze souboru `parser.c`, ale je implementována v samostatném souboru `expression_parser.c`. Veškeré načtené symboly jsou ukládány do zásobníku symbolů implementovaném v souboru `symstack.c`.

Precedenční analýza je řízena precedenční tabulkou. Znaky se stejnou prioritou jsou sdružené do jednoho řádku/sloupce tabulky. Řádek `rel` představuje relační operátory, řádek `eq` představuje ==, nebo != a řádek `op` představuje literály, `nil` nebo identifikátor. Symbolem $ jsou zobrazeny symboly, které nejsou validní pro výraz. Všechny tyto znaky považujeme za terminály.

Na začátku funkce `expression` se sleduje vrchní teminál. Ten pak spolu s načteným tokenem slouží jako indexy pro vyhledávání v precedenční tabulce. Následně se provede jednoduchá kontrola dělení nulou, která

je implementována funkcií `computeForZero`. Po kontrole dělení nulou se provede vyhledání v precedenční tabulce. Na základě vybraného pravidla se vykonávají následujíci akce. V případě redukce se sleduje, jestli redukujeme výraz s validním počtem operandů (ten je pro náš soubor pravidel vždy buď 1, nebo 3). Následně na základě toho, jaké mají operandy hodnoty (ty odpovídají hodnotám tokenů načtených lexikální analýzou), zjistíme, jestli je použité pravidlo validní a o jaké pravidlo se jedná. Další krok je kontrola použití správných typů operandů pro dané pravidlo a případné přetypování. Následuje už jen popnutí původních znaků ze zásobníku symbolů a pushnutí nového neterminálu s příslušným datovým typem.

V závěru funkce `expression` pak ještě proběhne kontrola validity následujícího symbolu. Ten je pro matematický výraz vždy `EOL` nebo `EOF` a pro booleovský výraz může mít taky hodnotu `)`, `then` nebo `do`.

1.5 Generování cílového kódu (`code_generator.c`)

Generování cílového kódu IFJcode18 se vykonává v průběhu druhého průchodu, kdy se jednotlivé instrukce vkládají do seznamu všech instrukcí. Takto sestavený seznam nám umožňuje případné přeskládání jednotlivých instrukcí. Specifický případ takového využití je deklarace proměnné v těle cyklu `while`, kdy její deklaraci přesuneme před jeho začátek. Tento seznam je následně vypsán na standardní výstup, pokud se během překladu nevyskytne nějaká chyba.

1.5.1 Začátek generování

Jedna z prvních instrukcí je generování hlavičky mezikódu `.IFJcode18` a generování definicí vestavěných funkcí. Toto generování probíhá na začátku druhého průchodu zdrojového kódu.

1.5.2 Generování hlavního kódu

Generování hlavního kódu probíhá v průběhu celého druhého průchodu. Syntaktická analýza a sémantická analýza volá instrukce generátoru kódu, když vyhodnotí, o jaký příkaz se jedná.

2 Speciální datové struktury

2.1 Seznam tokenů (`token_list.c`)

Během prvního průchodu načítáme jednotlivé tokeny ze standardního vstupu pomocí funkce `getNextToken` implementované v souboru `scanner.c` a ihned je zapisujeme do seznamu tokenů implemenetovaném v souboru `token_list.c`. Z tohoto seznamu čteme během druhého průchodu, neboť během překladu je zakázané modifikovat jakýkoliv soubor, jak je řečeno v zadání.

2.2 Struktura `string` (`str.c`)

Při tvorbě struktury `string` pro ukládání bufferu aktuálního tokenu jsme vycházeli z již vytvořené struktury a funkcí pro práci s ní v materiálech předmětu IFJ, a sice z archivu „Zjednodušená implementace interpretu jednoduchého jazyka“.

2.3 Struktura `symstack` (`symstack.c`)

Zásobník symbolů je implementován v souboru `symstack.c` a je využíván při precedenční analýze. Přistupujeme do něj ze souborů `parser.c` a `expression_parser.c`. Zásobník podporuje standardní zásobníkové operace a zároveň rozšiřující operaci na vkládání do zásobníku mimo jeho vrchol a operaci na získaní vrchního terminálu.

Položka zásobníku je typu struktury, která obsahuje druh symbolu, který je reprezentován hodnotou načtenou při lexikální analýze. Dále obsahuje datový typ, který, pokud symbol není literál, nebo id, je vždy typu `nil`. Posledním prvkem struktury je ukazatel na další prvek struktury.

3 Práce v týmu

3.1 Způsob práce v týmu

Na projektu jsme začali pracovat prakticky od jeho zadání. Postupně jsme si dle náročnosti rozdělovali práci mezi jednotlivce či dvojice. Pro zjednodušení práce jsme využívali vzdálený repozitář na webovém serveru `www.bitbucket.org`, který využívá verzovací systém Git.

3.2 Komunikace

Mezi sebou jsme komunikovali především osobně a to v prostorách seminárních místností školní knihovny jednou týdně a v pozdější fázi projektu i několikrát týdně. Mimo to jsme byli neustále v kontaktu skrze sociální sítě, především přes Skype.

3.3 Rozdělení práce

Každý z nás vynaložil na projekt maximální úsilí a přispěl k jeho vytvoření prakticky rovným dílem, proto má každý člen našeho týmu 25% podíl na tvorbě výsledného překladače imperativního jazyka IFJ18.

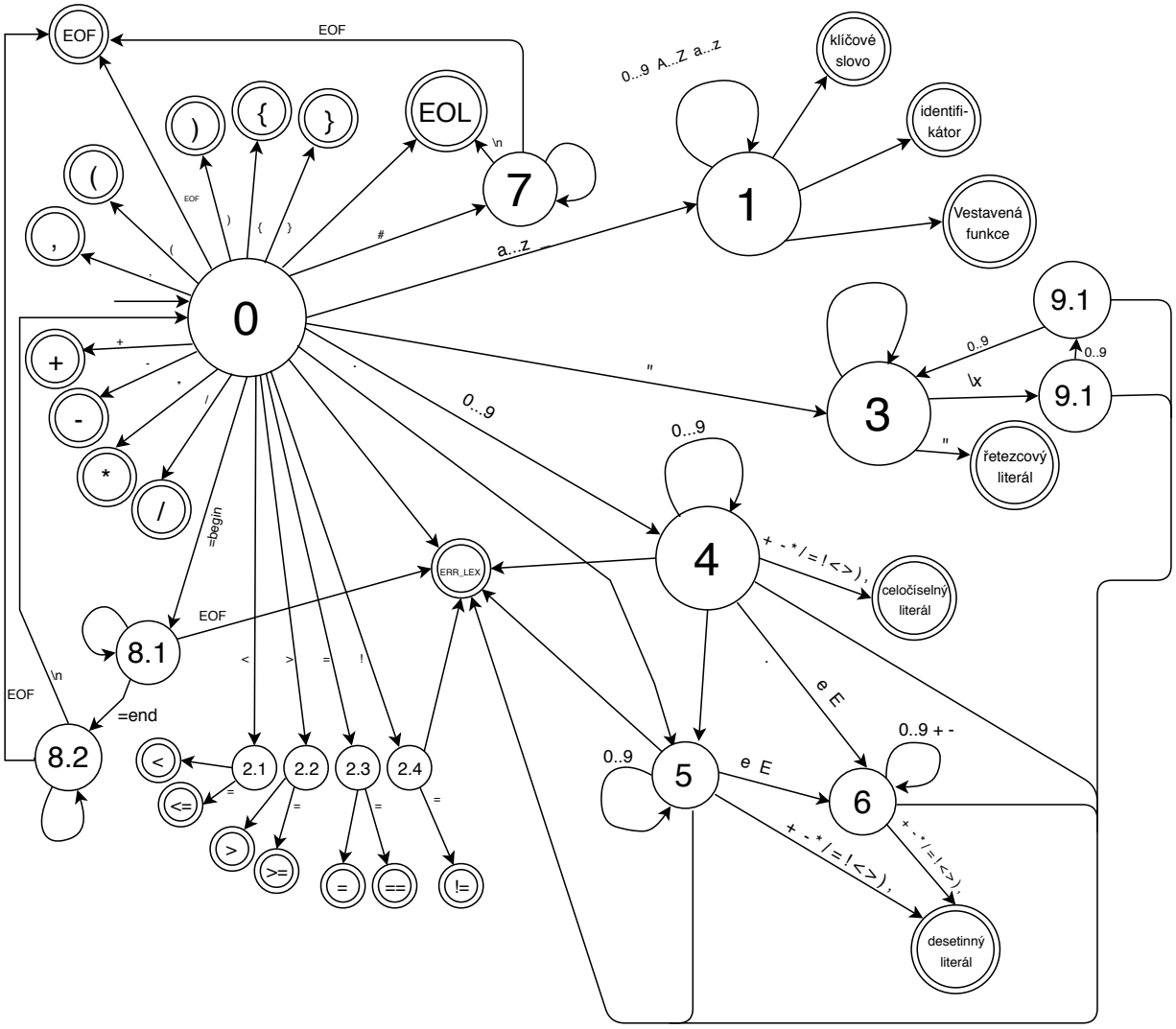
Člen	Úlohy
Ondřej Dacer	vedení týmu, organizace práce, lexikální analyzátor, syntaktický analyzátor, dokumentace
Roland Žitný	tabulka symbolů, generátor cílového kódu, testování, dokumentace
Ľubomír Švehla	tabulka symbolů, zásobník symbolů, sémantický analyzátor, testování, dokumentace
Jakub Novotný	lexikální analyzátor, testování, dokumentace

Tabulka 1: Rozdělení práce

Závěr

Na projektu jsme začali poctivě pracovat od začátku semestru, ihned po složení týmu. Udělali jsme si předběžný časový plán, který se nám jistou dobu dařilo plnit. Vzhledem k tomu, že jsme mírně podcenili složitost problematiky a nedostatečně jsme si ji předem nastudovali, jsme však začali po pár týdnech mírně zaostávat za naším plánem. V posledních dvou týdnech jsme se dostali k samotnému testování, které nám pomohlo do detailů pochopit zbývající části. Projekt nás mimo problematiky IFJ a IAL taky naučil, jak správně komunikovat a přidělovat práci v menším týmu.

4 Diagram konečného automatu specifikujícího lexikální analyzátor



Legenda :
0 – Počáteční stav automatu
1 – Identifikátory, klíčová slova, vestavěné funkce
2 – Dvou znakové operátory
3 – Řetězové literály
4 – Celočíselné literály
5 – Desetinné literály
6 – Desetinné literálý s exponentem
7 – Jednořádkový komentář
8 – Blokový komentář
9 – Převod z hexadecimálních čísel uvnitř řetězcového literálu

Obrázek 1: Diagram konečného automatu

5 LL – gramatika

1. <prog> -> <stat_list> EOF
2. <stat_list> -> <stat> EOL <stat_list>
3. <stat_list> -> ε
4. <stat> -> DEF ID (<params>) EOL <function_body> END
5. <stat> -> IF <expression> THEN EOL <condition_body>
   ELSE <condition_body> END
6. <stat> -> WHILE <expression> DO EOL <cycle_body> END
7. <stat> -> PRINT (<arg>)
8. <stat> -> ID <assign>
9. <stat> -> <func_call>
10. <stat> -> <expression>
11. <params> -> ID <param_next>
12. <params> -> ε
13. <param_next> -> , ID <param_next>
14. <param_next> -> ε
15. <condition_body> -> <inside_stat> EOL <condition_body>
16. <condition_body> -> ε
17. <cycle_body> -> <inside_stat> EOL <cycle_body>
18. <cycle_body> -> ε
19. <function_body> -> <inside_stat> EOL <function_body>
20. <function_body> -> ε
21. <inside_stat> -> IF <expression> THEN EOL <condition_body>
    ELSE <condition_body> END
22. <inside_stat> -> WHILE <expression> DO EOL <cycle_body> END
23. <inside_stat> -> PRINT (<params>)
24. <inside_stat> -> <func_call>)
25. <inside_stat> -> <expression>
26. <def_value> -> <expression>
27. <def_value> -> <func_call>
28. <assign> -> = <def_value>
29. <assign> -> ε
30. <func_call> -> ID (<arg>)
31. <func_call> -> ID <arg>
32. <arg> -> <value> <arg_next>
33. <arg> -> ε
34. <arg_next> -> , <value> <arg_next>
35. <arg_next> -> ε
36. <value> -> ID
37. <value> -> INT_VALUE
38. <value> -> DOUBLE_VALUE
39. <value> -> STRING_VALUE
40. <value> -> NIL

Tabulka 2: LL – gramatika

6 LL – tabulka

	DEF	IF	WHILE	PRINT	ID	,	=	INT_VALUE	DOUBLE_VALUE	STRING_VALUE	NIL	$
<prog>	1	1	1	1	1							1
<stat_list>	2	2	2	2	2							3
<stat>	4	5	6	7	8							
<params>					11							12
<param_next>						13						14
<condition_body>		15	15	15	15							16
<cycle_body>		17	17	17	17							18
<function_body>		19	19	19	19							20
<inside_stat>		21	22	23	24							
<def_value>					27							
<assign>							28					29
<func_call>					30,31							
<arg>					32			32	32	32	32	33
<arg_next>						34						35
<value>					36			37	38	39	40	

Tabulka 3: LL – tabulka

7 Precedenční tabulka

	*/	+-	rel	eq	(	)	op	$
*/	>	>	>	>	<	>	<	>
+-	<	>	>	>	<	>	<	>
rel	<	<		>	<	>	<	>
eq	<	<	<		<	>	<	>
(	<	<	<	<	<	=	<	
)	>	>	>	>		>		>
op	>	>	>	>		>		>
$	<	<	<	<	<		<	

Tabulka 4: Precedenční tabulka

Použité zdroje

[1] Hash Functions. [online].
URL http://www.cse.yorku.ca/~oz/hash.html

[2] hashfunc.c. 2009, [online].
URL https://github.com/google/crush-tools/blob/master/src/libcrush/hashfuncs.c