Dokumentace k projektu IFJ
Překladač imperativního jazyka IFJ18

Tým 025, varianta I

1. prosince 2018

Křehlík Petr (vedoucí)	xkrehl04	25%
Klobušický Martin	xklobu03	25%
Fučík Pavel	xfucik08	25%
Mikeš Tibor	xmikes12	25%

1 Úvod

Cílem projektu bylo vytvořit funkční překladač z imperativního jazyka IFJ18 na mezikód IFJcode18. Projekt je implementovaný v programovacím jazyku C.

2 Lexikální analyzátor (scanner)

Scanner pracuje jako konečný automat. Načítá postupně znaky a podle toho, jaký znak načetl a v jakém stavu se nachází, přechází do dalších stavů. Pokud je v koncovém stavu a dalším znakem nemůže pokračovat, vytvoří a předá parseru token. Token je struktura nesoucí veškeré potřebné informace, jako např.: jméno identifikátoru, typ tokenu, typ klíčového slova, atd. Jelikož parser provádí dvouprůchodovou analýzu, tokeny si při prvním průchodu ukládáme do jednosměrně vázaného seznamu, pro pozdější použití. Zároveň se také provádí lexikální analýza.

3 Syntaktická a sémantická analýza (parser)

Parser načítá postupně tokeny, které dodává scanner po zavolání funkce `get_next_token`. Je navržený jako dvouprůchodový.

Při prvním průchodu se hledají definice funkcí a zapisují se do globální tabulky symbolů. Nepostupuje podle LL pravidel, prochází tokeny a když najde definici funkce tak zapíše do tabulky její název a jména a počet parametrů. Následně hledá konec funkce. Již v této fázi je možné odhalit syntaktické chyby při zápisu funkcí.

Druhý průchod používá LL pravidla pro rekurzivní sestup. Postupně prochází rekurzivně kód a kontroluje, jestli je vše syntakticky a sémanticky správně. Pokud narazí na výraz tak předá řízení precedenční analýze, která vyhodnotí syntaktickou správnost výrazu a případně vrátí chybový kód. Pokud nalezne chybu, je průchod ukončen a vrací se chybový návratový kód. Syntaktické a sémantické chyby se kontrolují zároveň.

Pro sémantickou kontrolu se používají dvě tabulky symbolů. První, globální tabulka, obsahuje definice funkcí z prvního průchodu parseru a při druhém se do ní ukládají definice proměnných v hlavním těle programu. Druhá, lokální, tabulka se používá pro funkce. Pokud se při druhém průchodu najde definice funkce, je tabulka vymazána a naplněna definicemi parametrů funkce. Při průchodu funkcí se pracuje pouze s lokální tabulkou. Výjimka

nastává při nalezení identifikátoru. V tomto případě se musí ověřit v globální tabulce symbolů, zda se nejedná o volání funkce.

4 Precedenční analýza výrazů

4.1 Datové struktury pro analýzu výrazů

K implementaci precedenční analýzy byl zapotřebí zásobník symbolů. Tato struktura se nazývá `esym_stack` a její položky jsou typu `expr_symbol`. Struktura `expr_symbol` obsahuje informace o typu (identifikátor, operátor, neterminál, speciální znaky < a >, ...), případně i hodnotu, např. název identifikátoru nebo hodnotu konstanty. `esym_stack` nabízí kromě běžných funkcí nad zásobníkem i vložení za poslední terminál a vrácení posledního terminálu.

4.2 Algortimus pro analýzu výrazů

Algoritmus pro samotnou analýzu je stejný, jaký byl prezentován na přednášce. K načítání dalších tokenů je použita funkce `get_next_token()` a načtené tokeny jsou pak konvertovány na typ `expr_symbol` funkcí `expr_symbol_from_token()`. Následující symbol výrazu a terminál nejblíže vrcholu zásobníku jsou konvertovány na typ `prec_table_input` funkcí `pti_from_esym()`. Poté jsou použity jako indexy precedenční tabulky, která rozhoduje o jejich prioritě/validitě. V závislosti na jejich prioritě může být na zásobník přidán buď samotný přečtený symbol, speciální symbol < společně s přečteným symbolem, nebo může být redukována část zásobníku. Při redukování dochází k volání generátoru kódu, který vytváří zásobníkové instrukce pro zpracování výrazu.

4.3 Precedenční tabulka

	+ -	*/	(	)	**i**	**r**	**$**
+ -	>	<	<	>	<	>	>
*/	>	>	<	>	<	>	>
(	<	<	<	=	<	<	x
)	>	>	x	>	x	>	>
i	>	>	x	>	x	>	>
r	<	<	<	>	<	x	>
$	<	<	<	x	<	<	x

5 Generování kódu

5.1 Datová struktura pro ukládání výsledného kódu

Generátor kódu používá pro ukládání výsledného kódu strukturu `dstring`, která funguje jako dynamicky se rozšiřující řetězec znaků. Pokud by vložením řetězce došlo k překročení aktuální kapacity struktury, je kapacita rozšířena o konstantní velikost.

5.2 Generování kódu z parseru a předávání informací

Výsledný kód je generován v průběhu druhého průchodu syntaktické analýzy. Ke generování dochází voláním funkcí deklarovaných v souboru `code_gen.h`, které kód vkládají do datové struktury `dstring`. Pro předávání informací o symbolech a konstantách je použita struktura `symbol_t`. Pokud je potřeba pouze názvu funkce/proměnné, může být použit i obyčejný řetězec znaků.

5.3 Definice proměnných ve smyčkách

Proměnná nemůže být v `IFJcode18` vícekrát definována, což mohlo způsobit problémy při generování cyklů. Před vstupem do cyklu proto parser vytvoří dočasný `dstring`, do kterého ukládá generovaný kód pro všechny příkazy cyklu kromě definic proměnných. Definice proměnných vkládá do hlavního `dstring` , čímž zajišťuje, že se definice nachází před cyklem. Po nalezení konce cyklu připojí parser dočasný *dstring* obsahující tělo cyklu na konec hlavního `dstringu`.

5.4 Generování funkcí

Před voláním funkce je vytvořen nový dočasný rámec, na který jsou ukládány argumenty funkce pod názvem určeným při definici funkce. Návratová hodnota funkce je ukládána do proměnné `$expr_result` nacházející se v globálním rámci.

5.5 Definice funkcí uvnitř hlavního těla programu

Každému tělu funkce v generovaném kódu předchází instrukce skoku za konec těla této funkce. Tím je zajištěno, že definice funkcí nebudou ovlivňovat hlavní tělo programu.

5.6 Generování funkce print

Funkce `print` se od ostatních funkcí liší libovolným (nenulovým) počtem parametrů, proto je její generování řešeno odlišně. Parser pro každý argument předaný této funkci volá generátor, který každým voláním vytvoří instrukci `WRITE` vypisující daný argument. Nedochází tedy k vytváření rámce, předávání parametrů, skoku, a ani jiným podobným úkonům společným pro volání funkce.

5.7 Předávání hodnoty výrazu

Hodnoty výrazů jsou stejně jako návratové hodnoty funkcí ukládány do proměnné `$expr_result` nacházející se v globálním rámci.

5.8 Generování návěští pro výrazy a řídicí struktury

Součástí kódu pro provádění operací nad datovým zásobníkem, jako je například sčítání, jsou kontroly typů/hodnot operandů. Tyto kontroly používají podmíněný skok na návěští, které jsou automaticky generovány. K tomu jsou použity čítače `expression_id` a `subexpression_id`, kde `expression_id` značí identifikátor výrazu a `subexpression_id` značí identifikátor jednotlivých operací výrazu. Návěští řídicích struktur také vyžadují unikátní identifikátory, které jsou určeny pomocí čítače `unique_id`.

5.9 Implicitní konverze

Pro implicitní konverze operandů byla vytvořena vestavěná funkce `conv_ops`, která vyjme poslední dva prvky datového zásobníku a na základě jejich typů je (pokud je to možné) konvertuje a pak vrátí v `$expr_result` číslo informující o typu konvertovaných operandů. Tato funkce je volána před každým provedením aritmetické/logické operace nad zásobníkem. Pokud jsou operandy nekompatibilní a prováděná operace není porovnání na rovnost/nerovnost, je vygenerována instrukce ukončující program.

6 Týmová práce

První týden po zveřejnění zadání jsme se společně sešli a rozdělili úkoly následovně:

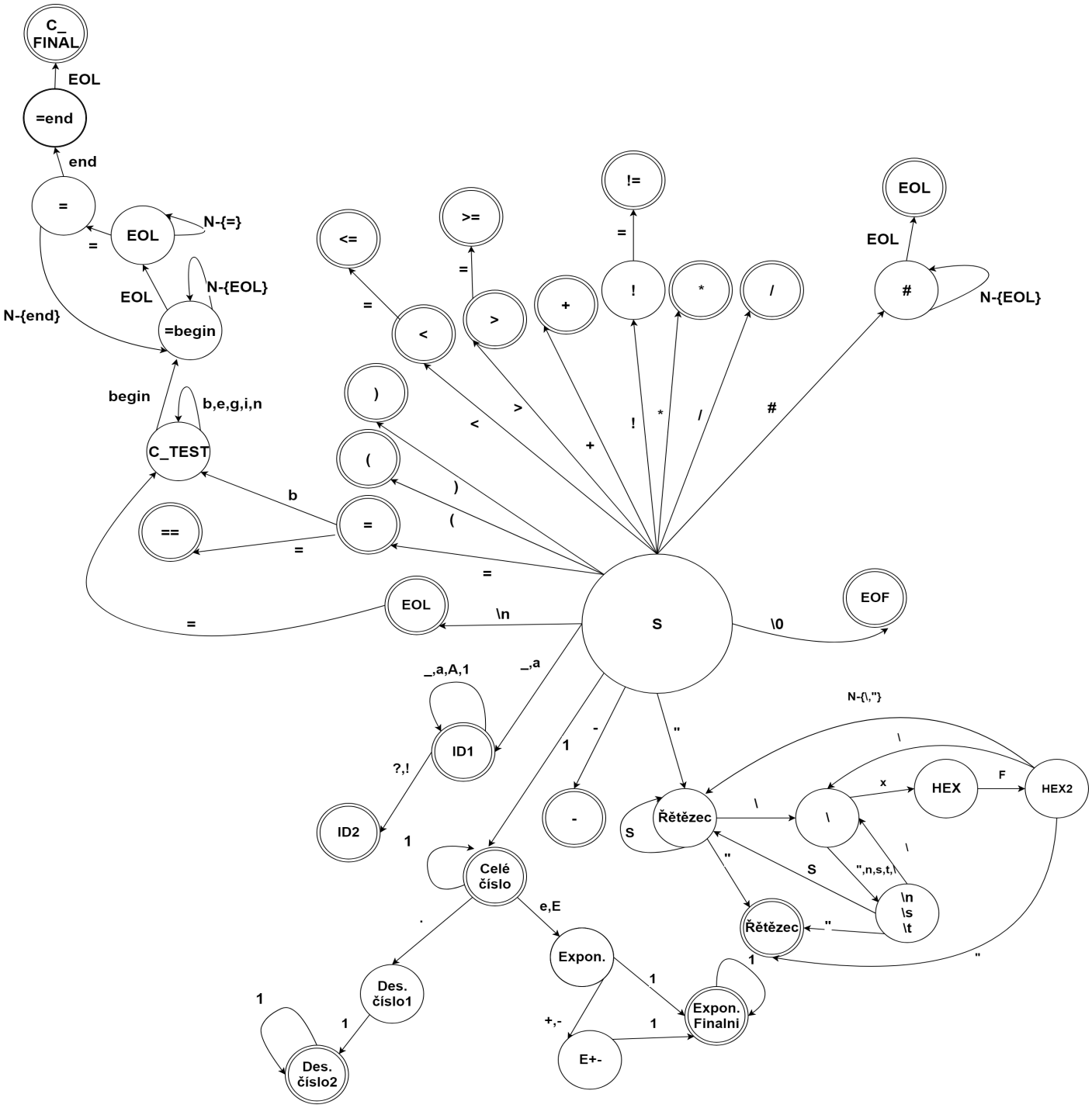
- Lexikální analyzátor (xklobu03,xfucik08)
- Syntaktický a sémantický analyzátor (xkrehl04)
- Precedenční syntaktická analýza (xmikes12)
- Generování kódu (xmikes12)
- Testování (xkrehl04,xmikes12,xklobu03)

Pravidelně každý týden jsme měli schůzky a probírali problémy při řešení. Pro online komunikaci jsme používali službu Discord a jako úložiště sdílený repozitář GitHub.

6.1 Testování

Použili jsme dva typy testování. Jako první jednotkové testy napsané v jazyku C pro testování funkčnosti samostatných částí a jako druhé regresní testování. K regresnímu testování jsme použili Shell skript, který projde všechny soubory (vstupní kód) ve složce s testy a vyhodnotí, jestli se očekávaný návratový kód (umístěný jako komentář na posledním řádku vstupního souboru) rovná skutečnému.

7 Návrh konečného automatu lexikální analýzy



7

Vysvětlivky:

N...Cokoliv
a...Malá písmena
A...Velká písmena
1...Číslice
S... Znaky pro řětězec, vše kromě ",\
F... Hexadecimální číslice

Stav automatu

Koncový stav automatu

8 Pravidla LL gramatiky

1.<prog> -> def func-id (<param-list-first>) eol <statement-list> end eol <prog>
2.<prog> -> <statement> <prog>
3.<prog> -> eol <prog>
4.<prog> -> eof
5.<statement-list> -> <statement> <statement-list>
6.<statement-list> -> eol <statement-list>
7.<statement-list> -> ϵ
8.<statement> -> id <id>
9.<statement> -> func-id (<arg-list-first>) eol
10.<statement> -> if <expression> then eol <statement-list> <else-end> eol
11.<statement> -> while <expression> do eol <statement-list> end eol
12.<statement> -> print(<print-arg-list-first>) eol
13.<statement> -> ϵ
14.<else-end> -> else eol <statement-list> end
15.<else-end> -> end
16.<id> -> = <id-assign>
17.<id> -> <expression> eol
18.<id> -> ϵ
19.<id-assign> -> <expression> eol
20.<id-assign> -> func-id(<arg-list-first>) eol
21.<print-arg-list-first> -> ϵ
22.<print-arg-list-first> -> id <param-list>
23.<print-arg-list> -> , id <param-list>
24.<print-arg-list> -> ϵ
25.<param-list-first> -> ϵ
26.<param-list-first> -> id <param-list>
27.<param-list> -> , id <param-list>
28.<param-list> -> ϵ
29.<arg-list-first> -> ϵ
30.<arg-list-first> -> arg <arg-list>
31.<arg-list> -> , arg <arg-list>
32.<arg-list> -> ϵ
33.<arg> -> id
34.<arg> -> int
35.<arg> -> float
36.<arg> -> string

	def	eol	eof	ε	id	func-id	if	while	print	else	end	=	,	arg	int	float	string
PROG	1	3	4														
STATEMENT-LIST		6		7													
STATEMENT				13	8	9	10	11	12								
ELSE-END										14	15						
ID												16					
ID-ASSIGN				18		20											
PRINT-ARG-LIST-FIRST				21	22												
PRINT-ARG-LIST				24								23					
PARAM-LIST-FIRST				25	26												
PARAM-LIST				28									27				
ARG-LIST-FIRST				29										30			
ARG-LIST				23									31				
ARG					33										34	35	36

9 Závěr

Projekt se nám podařilo po náročné práci zprovoznit. Velmi nám pomohlo vlastní testování a pokusná odevzdání. Jelikož jsme práci na projektu zahájili s předstihem, stihli jsme dokončit vše, co jsme dokončit plánovali. Práce na projektu byla zajímavá z hlediska rozsáhlosti a nutnosti pracovat v týmu. Přinesl nám mnoho zajímavých zkušeností do budoucna.