Dokumentácia k projektu IFJ a IAL

Implementácia prekladača imperatívneho jazyka IFJ19

Tím 127, varianta I

Jiří Žák (xzakji02) 25%

Ivan Halomi (xhalom00) 25%

Adam Ševčík (xsevci64) 25%

Martin Hiner (xhiner00) 25%

10. 12. 2019

1. Úvod

Cieľom tohto projektu bolo vytvoriť prekladač, ktorý načíta zdrojový kód v IFJ19, čo je zjednodušená podmnožina jazyka Python 3, a prekladá ho do IFJcode19 kód. Táto dokumentácia obsahuje popis jednotlivých častí prekladača a ich implementačné detaily, použité špeciálne techniky, popis práce v tíme a dodatočné materiály ako diagramy a tabuľky.

2. Návrh a implementácia

Jednotlivé modely boli paralelne vyvíjané každým členom tímu, podľa rozdelenia.

2.1. Lexikálny analyzátor

Implementácia lexikálneho analyzátora sa nachádza v súbore `scanner.c` a využíva pomocnú knihovňu `string.c` pre jednoduchšiu prácu s reťazcami.

Hlavná funkcia lexikálneho analyzátora je `get_token()`, ktorá je implementácia konečného automatu podľa diagramu vytvoreného na základe zadania (viď nižšie). Funkcia vracia ukazateľ na objekt typu `tToken`, ktorý reprezentuje jednu lexému zo zadaného kódu. Sú v ňom uložené informácie ako riadok, na ktorom sa token nachádza, obsah a typ tokenu, prípadne podtyp, podla požiadaviek ostatných modulov. Jednotlivé stavy automatu sa nachádzajú v type `enum tState`, ktorý sa ale pre jednoduchosť zároveň využíva ako označenie typov a podtypov tokenov. Funkcia je implementovaná ako nekonečný switch, ktorý znak po znaku prechádza štandardný vstup a podla zadaných pravidiel ho delí na jednotlivé tokeny.

Jazyk IFJ19 využíva odsadenie riadkov na zoskupenie príkazov do sekvencie, preto bolo pre kontrolu nutné využiť pomocný zásobník typu LIFO. Zmena odsadenia sa prejaví odoslaním tokenov `sIndent` alebo s `sDedent`, ktoré nahradzujú zložené zátvorky známe z jazyka C.

V súbore sa ešte nachádzajú pomocné funkcie `init_token()` na inicializáciu tokenu, `unget_token()`, ktorou riešime konflikt pravidiel v LL gramatike, a `assign_type()` na zistenie, či sa je v tokene uložené kľúčové slovo alebo identifikátor. Taktiež sa tu nachádzajú pomocné funkcie pre prácu so zásobníkom.

2.2.Syntaktický analyzátor

Syntaktická analýza prebieha v súboroch `parser.c` a `exprParser.c`. Syntaktický analyzátor využíva funkciu `get_token()` na načítanie tokenov a ich následnú kontrolu. Syntaktickú analýzu máme rozdelenú na precedenčnú analýzu a rekurzívny zostup. Rekurzívny zostup sa odohráva vo forme nekonečného cyklu v funkcii `doParse()`, ktorá podľa potreby volá rekurzívne ďalšie funkcie v súbore. Táto kontrola prebieha pomocou LL gramatiky. Precedenčná analýza je volaná ak treba spracovať výraz a je volaná zo súboru `exprParser.c` funkciou `exprParsing()` a funguje na základe tabuľky precedenčnej analýzy (viď nižšie).

Analýza spracováva tokeny ktoré sa následne uložia na LIFO zásobník, kým analýza nenájde možnosť aplikovať pravidlo, pri ktorom zároveň prebieha sémantická kontrola a prípadné pretypovanie. A robí tak kým nenarazí v tabuľke na exit_parse. Na zásobníku zostáva len znak $ a token podla ktorého sa zistí typ výsledku a následne uloží do tabuľky symbolov.

2.3.Sémantický analyzátor

Úlohou sémantického analyzátoru je kontrola, či sú operandy správneho typu a teda či sa dá vykonať aritmetmeticko-logická operácia. V našom projekte je sémantický analyzátor implementovaný v súbore `exprParser.c` a `parser.c`.

2.4.Generovanie kódu

Generovanie kódu je implementované v súbore `instruction-list.c`. V parseri sa postupne vytvárajú inštrukcie na generovanie kódu, ktoré sa ukladajú do jedného z dvoch dvojsmerne viazaných zoznamov typu `tDLListInst`. V prvom zozname sú uložené definície funkcií, v druhom vytvorené inštrukcie. Ak počas behu prekladača nebola zistená žiadna z chýb, vypíše funkcia `instructionPrinter()` všetky uložené inštrukcie na štandardný výstup.

Ďalej sa tiež v tomto súbore nachádzajú implementácie vstavaných funkcií.

2.5.Makefile

Požiadavky projektu zahŕňali súbor Makefile, ktorý prekladá projekt príkazom make. V súbore sú nastavené pravidlá, podla ktorých sa má projekt prekladať. Projekt se prekladá prekladačom `gcc` s nastavenými príznakmi. Zo súborov s príponou `.c` sa vytvoria objektové súbory s príponou `.o` a z tých sa potom vytvorí jeden spustiteľný súbor s názvom main.

Tiež sme použili Makefile na testovanie a odstránenie dočasných súborov.

3. Použité špeciálne techniky a algoritmy

Zaviedli sme niekoľko špeciálnych dátových štruktúr pre projekt.

3.1. Tabuľka s použitím binárneho vyhľadávacieho stromu

Vytvorili sme tabuľku s použitím binárneho vyhľadávacieho stromu, ktorý slúži ako tabuľka symbolov, čo súvisí s predmetom IAL, kde sme sa o tom učili. Implementovali sme potrebné funkcie na prácu s touto tabuľkou ako inicializáciu, pridanie novej položky, odstránenie konkrétnej položky, vyhľadávanie a odstránenie celej tabuľky z pamäte.

3.2.ADT String

Pre jednoduchšiu prácu s reťazcami sme implementovali abstraktnú dátovú štruktúru string v zdrojovom súbore `string.c`. V štruktúre je uložená dĺžka reťazca, alokovaná dĺžka reťazca a ukazateľ na reťazec. Funkcie sprostredkovávajú inicializáciu a uvoľnenie pamäti, pridanie jedného znaku, pridanie reťazca a porovnanie dvoch reťazcov.

3.3. LIFO zásobník

Syntaktická analýza vyžaduje pre svoj beh uloženie tokenov. Z tohto dôvodu sme implementovali zásobník typu Last In – First Out. Implementácia sa nachádza v súbore `lifo.c` a okrem zásobníku definuje novú štruktúru `tRedukToken`, ktorá viac vyhovuje potrebám syntaktickej a sémantickej analýzy než pôvodný token získaný z lexikálnej analýzy.

3.4. Dvojsmerne viazaný zoznam

S cieľom usporiadaného ukladania inštrukcií, pre neskoršie spracovanie, bola využitá abstraktná dátová štruktúra Dvojsmerne viazaný zoznam, ktorá bola bližšie preberaná v predmete IAL. Tento typ zoznamu nám poskytoval obojsmerný prechod zoznamom a možnosť pristupovania k prvému a poslednému prvku. Štruktúra bola implementovaná priamo v súbore `instruction-list.c` spoločne s potrebnými funkciami.

4. Tímová práca

Najprv sme rozdelili projekt podľa častí a potom sme postupne na ňom pracovali. Každý člen tímu pracoval samostatne na svojej časti, s prípadnou pomocou druhých členov.

5. Spôsob vývoja

5.1. Git

Ako verzovací systém sme použili git a ako vzdialený repozitár službu GitHub. Každý začínal vo svojej vlastnej vetve, kde vyvíjal svoju časť zadania. Neskôr sme všetky časti spojili do hlavnej vetvy, kde sme pokračovali vo vývoji už aspoň čiastočne funkčných modulov.

5.2. Komunikácia

Komunikácia bola väčšinou vykonávaná osobne alebo prostredníctvom služby Messenger. V neskorších etapách prevládala osobná komunikácia za prítomnosti celého tímu.

5.3. Rozdelenie práce

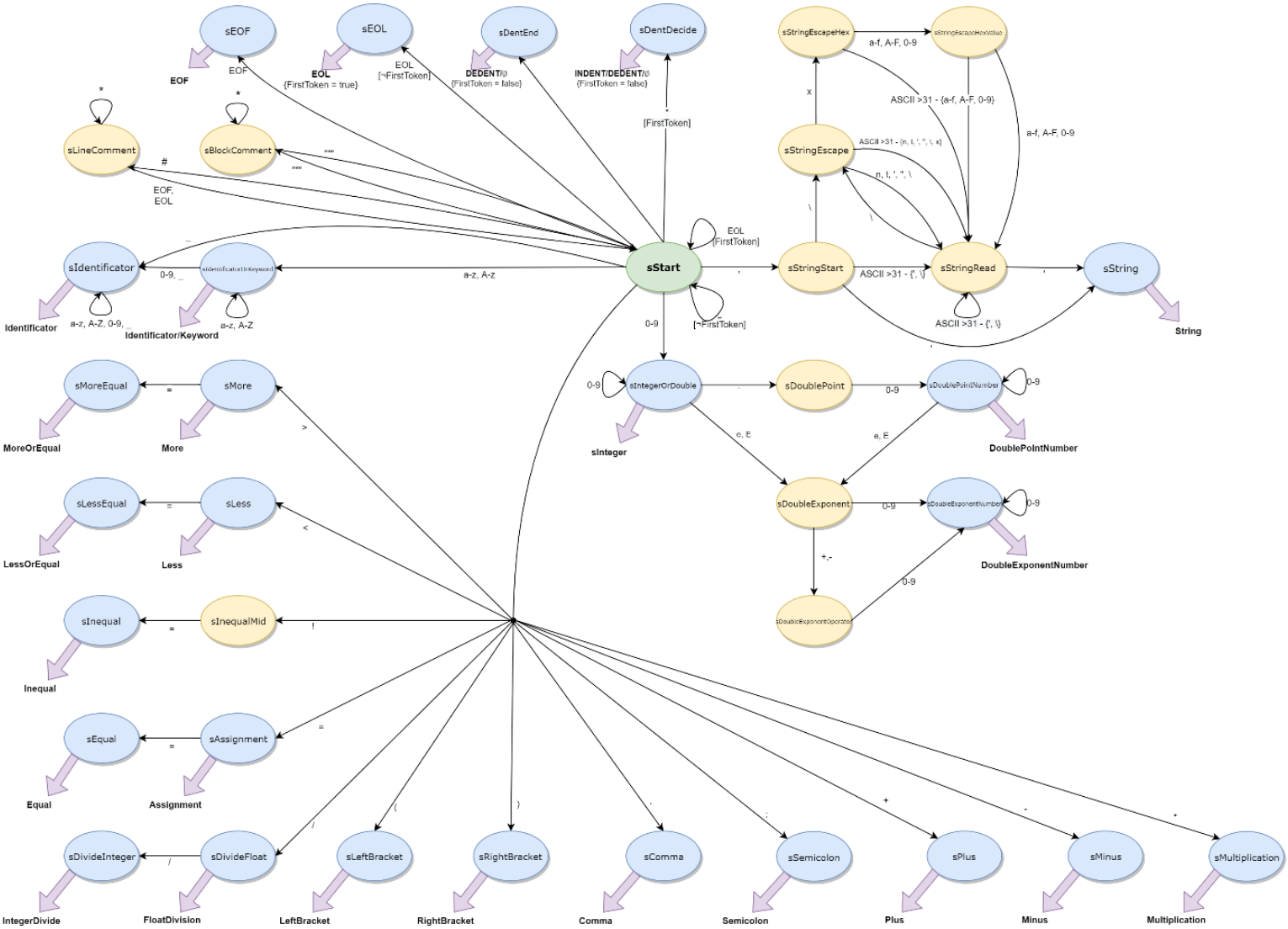
Rozdelili sme prácu rovnomerne s ohľadom na zložitosť. Takže každý člen tímu dostal hodnotenie 25%.

Jiří Žák - syntaktický a sémantický analyzátor, testovanie, dokumentácia
Martin Hiner - lexikálny analyzátor, testovanie, dokumentácia
Ivan Halomi - syntaktický a sémantický analyzátor, testovanie
Adam Ševčík - generovanie kódu, testovanie

5. Záver

Projekt bol pre nás určite zatiaľ najnáročnejší za našu dobu pôsobenia na tejto fakulte, a to nielen z hľadiska implementácie ale tiež komunikácie v tíme a dodržania zadania. Počas vývoja sme sa stretli s mnohými prekážkami, ktoré sme museli riešiť ladením, testovaním alebo konzultáciou s iným tímom. Dokončením tohto projektu sme si nielen utvrdili naše znalosti z predmetov IFJ a IAL ale tiež získali skúsenosti s prácou v tíme.

Diagram konečného automatu lexikálnej analýzy:



Tabuľka precedenčnej analýzy:

	+	-	*	/	//	=	>	<	>=	=<	><	(	)	EOL	V>R
+	>	>	<	<	<	>	>	>	>	>	>	<	>	>	<
-	>	>	<	<	<	>	>	>	>	>	>	<	>	>	<
*	>	>	>	>	>	>	>	>	>	>	>	<	>	>	<
/	>	>	>	>	>	>	>	>	>	>	>	<	>	>	<
//	>	>	>	>	>	>	>	>	>	>	>	<	>	>	<
=	<	<	<	<	<	>	<	<	<	<	>	<	>	>	<
>	<	<	<	<	<	>	>	>	>	>	>	<	>	>	<
<	<	<	<	<	<	>	>	>	>	>	>	<	>	>	<
>=	<	<	<	<	<	>	>	>	>	>	>	<	>	>	<
=<	<	<	<	<	<	>	>	>	>	>	>	<	>	>	<
><	<	<	<	<	<	>	>	>	>	>	>	<	>	>	<
(	<	<	<	<	<	<	<	<	<	<	<	<	=	>	<
)	>	>	>	>	>	>	>	>	>	>	>	error	>	>	error
EOL	<	<	<	<	<	<	<	<	<	<	<	<	error	exit	<
V>R	>	>	>	>	>	>	>	>	>	>	>	error	>	>	error

10. 12. 2019

LL gramatika:

```
{
  1: DEF→def id ( PARAMS ) : seol sindent BODY sdedent BODY_N,
  2: PARAMS→id PARAMS_N ,
  3: PARAMS_N→, id PARAMS_N ,
  4: PARAMS_N→ε ,
  5: BODY_N→ε ,
  6: BODY→ASSIGN BODY_N,
  7: ASSIGN→id = EXPR ,
  8: BODY→if EXPR : seol sindent BODY sdedent else : seol sindent BODY sdedent BODY_N,
  9: BODY→while EXPR : seol sindent BODY sdedent BODY_N,
  10: ASSIGN→id = func ( PARAMS ) ,
  11: BODY→pass ,
  12: BODY→return EXPR ,
  13: EXPR→vyraz ,
  14: EXPR→ε ,
  15: EXPR→id ,
  16: EXPR→[EXPR_N] ,
  17: EXPR_N→,EXPR_N ,
  18: EXPR_N→ε ,
  19: EXPR_N→EXPR
  20: BODY_N→pass ,
  21: BODY_N→return EXPR
  22: BODY_N→if EXPR : seol sindent BODY sdedent else : seol sindent BODY sdedent BODY_N,
  23: BODY_N→while EXPR : seol sindent BODY sdedent BODY_N,
  24: BODY_N→ASSIGN BODY_N,
  25: PARAMS→ε
}
```

LL tabuľka:

def	id	(	)	:	seol	sindent	sdedent	,	=	if	else	while	func	pass	return	vyraz	[EXPR_N]	,EXPR_N	$
1																			
	2		25																
	6									8		9		11	12				
	24						5			22		23		20	21				5
			4					3											
	7,10																		
	14,15			14			14			14		14		14	14	13	16		14
	19															19	19	17	