Dokumentace společného projektu pro IFJ a IAL

Tým číslo 041, varianta I

Bez rozšíření

Šíma Vojtěch xsimav01 32%
Fabián Michal xfabia13 15%
Čábela Radek xcabel04 33%
Poposki Vasil xpopos00 20%

prosinec 2020

Obsah

Seznam tabulek

Obrázky

1 Lexikální analýza

Celý proces činnosti filtru zahájí skener. Jeho funkcionalita je následující: analyzuje vstupní kód znak po znaku a seskupuje jej do tokenů podle konečného automatu na obrázku níže. Tedy při startu programu se vždy přečte celý vstupní kód a zkontroluje jeho lexikální správnost rozpoznáváním lexémů, a až poté se zavolá parser a předá se mu zřetězený seznam tokenů. **Zřetězený seznam** je definovaný právě v hlavičce souboru **scanner.h**. Prvky seznamu jsou tokeny, které jsou deklarované také v tomto souboru jako enumerizace prvků, v níž název prvku je dán formulou `T_NÁZEV`.
V hlavičkovém souboru skeneru jsou dále definovány stavy jako datový typ enum (pojmenované podle formule `STATE_NÁZEV`) využité v hlavní funkci `scannerDKA`. Tato funkce je jako stěžejní bod konečného automatu lexikální analýzy volána vyšší funkcí opakovaně, aby načetla znak po znaku. Funkce v případě úspěchu vytvoří token a v opačném případě, kdy narazí na chybu (neplatný znak na vstupu pro daný stav), přeruší nadřazenou tvorbu zřetězeného seznamu a skener skončí s chybou.
Konečný automat je navržen tak, aby ignoroval **mezery a komentáře**, nikoli odřádkování, to je vráceno pomocí tokenu `T_EOL`.

2 Syntaktická analýza

Syntaktická analýza je dalším velkým segmentem celého projektu, její kód se nachází v souboru **parser.c**.
Základem syntaktické analýzy je LL gramatika, pro jako první byly vytvořeny pravidlo podle zadání projektu. Podle gramatiky je následně sestrojena LL tabulka, kterou se analýza řídí. Jednotlivá pravidla (netermy) mají vytvořeny vlastní funkce (jednodušší pravidla jsou vloženy do jiných nadřazených pravidel) a v nich se dále provádí pomocí LL tabulky rekurzivní sestup.
Pracujeme zde s obousměrným vázaným seznamem, do kterého lexikální analyzátor uložil jednotlivé lexémy (tokeny) ze vstupního souboru. Vždy v jednotlivých funkcích kontrolujeme, zda aktuální token odpovídá aktuálnímu pravidlu. V případě, že ano, postupujeme hlouběji do funkce, případně voláme jinou odpovídající funkci. V opačném případě přepíšeme typ aktuálního tokenu na `T_UNKNOWN`, do jeho datové části uložíme typ chyby - `ERR_SYNTAX` a následně tento token vrátíme pomocí příkazu RETURN výše v rekurzivním sestupu.
Pokud projdeme celým kódem až k tokenu `T_EOF`, který se na daném místě kódu může nacházet (byla funkce main(), správné uzávorkování funkcí apod.), ukládáme do datové části `T_UNKNOWN` hodnotu `ERR_OK` a následuje return. Na nejvyšší úrovni pak rozhodujeme podle datové části tokenu `T_UNKNOWN`, jakou chybovou hlášku budeme předávat do funkce main.

2.1 Parsování výrazů

Dle pravidel LL gramatiky se v určitých místech kódu parser dostane na místo, kde by se měl vyskytovat výraz. V naší implementaci se právě v tomto místě parser odkáže na funkci, která provede **precedenční** syntaktickou **analýzu zdola nahoru**. Mimo jiné vygeneruje mezikód pro aritmetické operace.
Funkce pracuje principem precedenční analýzy zdola nahoru. Důležité je rozlišení vstupních tokenů, zda-li jsou platné. Podle vytvořené tabulky se v další funkci identifikují. Následně je zde prováděna podoba algoritmu probíraného na přednášce IFJ. Získá se vstupní token na vrcholu zásobníku a porovná se s novým, z jejich vztahu se získá neterminál "$<$", "$>$", "$=$", nebo chybový " ". Pro každou relaci se následně provede odpovídající činnost, z nichž nejkomplexnější se provadí pro uzavírající neterminál. Zde se nad rámec činnosti uváděného algoritmu například provádí ještě sémantická kontrola pro identifikátory na vstupu a nebo se generuje mezikód aritmetických operací.
Parser je zpětně informován o výskytu porovnání a nebo chybě ve výrazu.

3 Sémantická analýza

Sémantická analýza je založena na dvou dvou tabulkách, které jsou implementovány pomocí binárního vyhledávacího stromu, který byl tvořen na základě našeho dřívějšího domácího úkolu z předmětu IAL.
Jedna z tabulek (LocalTable) je pro proměnné, které se ukládájí na základě jména a aktuální hloubky zanoření v programu. Při výstupu ze zanoření o úroveň výše proběhne promazání tabulky a smazání hodnot, který byly na dané

úrovni definovány. K těmto úkonům používáme pomocných funkcní v souboru **symtable.c**, které nám například procházejí stromem a hledají položky, mažou požadované položky nebo vrací datový typ definované proměnné. Druhá tabulka slouží pro ukládání jmen funkcí a hodnot k nim příslušným (datové typy parametrů a návratových hodnota, zda byla funkce definována).
Hlavním úkolem sémantické analýzy je kontrola kompatibality datových typů dřívě definovaných proměnných a funkcí. Veškerá sémantická analýza probíhá v souboru **parser.c** (kromě sémantické akce ve výrazech, ta probíhá nezávisle a pouze dochází k vzájemnému porovnávní očekávaných hodnot s hodnotami reálnými.

4 Generování kódu

Generování kódu v jazyce IFJcode20 probíhá současně se syntaktickou a sémantickou analýzou. Cílový kód se vypisuje na standardní výstup. Generování kódu je implementováno v souborech **parser.c**, **exprBottomUp.c** a **codegen.c**. Soubor **codegen.c** obsahuje funkce pro generování vestavěných funkcí v jazyce IFJcode20, formátování datových typů a generování jednotlivých částí programu. Na začátku generování program vypisuje záhlaví mezikódu a vestavěné funkce. Poté se vypíše příkaz pro skok do funkce `$$main`, která je hlavní funkcí programu, ekvivalentně k jazyku IFJ20. Generování výrazů je implementováno v souboru **exprBottomUp.c**. Generování jednotlivých částí programu, které je implementováno v souboru **parser.c** obsahuje generování funkcí, navěští uvnitř funkce, proměnných a návratových typů a generování podmínek a cyklů.

5 Kompilace projektu

K finálnímu sestavení spustitelného souboru projektu `projIFJ20` byl vytvořen Makefile, který spustí open source nástroj **CMake** a sestaví ze všech zdrojových a hlavičkových souborů spustitelný soubor projektu.
K používání CMaku bylo rozhodnuto z hlediska flexibility a rychlosti práce.

6 Týmová práce na projektu

6.1 Rozdělení práce

Vojtěch Šíma	syntaktická analýza, tvorba gramatiky, sémanická analýza, generování kódu, dokumentace
Radek Čábela	tvorba automatu, lexikální analýza, parser výrazů, generování kódu, dokumentace
Vasil Poposki	generování kódu
Michal Fabián	tabulka symbolů, testování, dokumentace

6.2 Správa souborů projektu

Pro správu souborů jsme využívali verzovací nástroj Git. Službu GitHub jsme používali jakožto vzdálený repositář, který nám pomohl při vzájemné spolupráci na projektu.
Tato služba nám umožnila společnou práci nad jedním či více soubory v jednom okamžiku. Jednotlivé soubory jsme postupně ukládali do tzv. repositáře, což nám umožnilo pracovat s aktuálními soubory.

6.3 Komunikace

Pro komunikaci nám ve většině případech sloužil komunikační kanál Discord, který nám umožnil sdílení obrazovky a nahradil nám tak nutnost osobního kontaktu. Během práce na projektu jsme zde mohli sdílet své nápady a postupně je implementovat do projektu. Dále jsme využili platformu Facebook Messenger, kde probíhalo plánování schůzek.

6.4 Rozdělení bodů

Nerovnoměrné rozdělení bodů je způsobeno tím, že každý člen týmu odvedl na projektu jiné množství práce, které je ekvivaletní přiděleným procentům.

7 Pravidla LL gramatiky

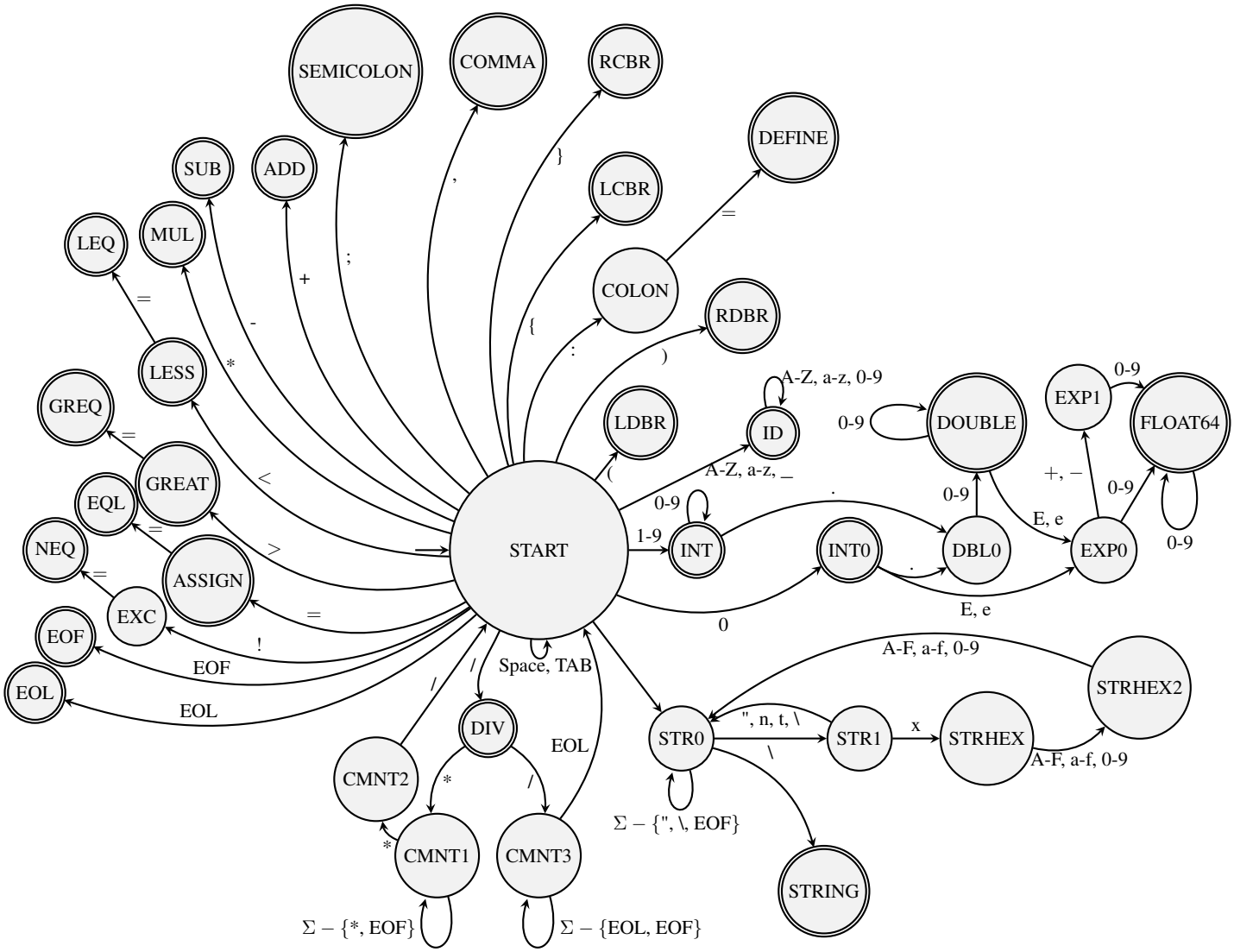
1. <prog> –> package main <body> EOF
2. <func> –> func id (<params>) (<navrattype_n>) { EOL <body> }
3. <params> –> id (<datatype>) (<params_n>)
4. <params> –> ε
5. <params> –> , id <datatype> (<params_n>)
6. <params_n> –> ε
7. <datatype> –> int
8. <datatype> –> float64
9. <datatype> –> string
10. <navrattype_n> –> <datatype> , <navrattyype_n>
11. <navrattype_n> –> ε
12. <paramscall> –> id <paramscall_n>
13. <paramscall_n> –> , id <paramscall_n>
14. <paramscall_n> –> ε
15. <id_n> –> , id tem <id_n>
16. <id_n> –> ε
17. <returndef> –> return<vyraz_n>
18. <if> –> if vyraz { EOL <body> } else { EOL <body> }
19. <if> –> for <fordef>; vyraz; <prikaz_prirazeni> { EOL <body> }
20. <fordef> –> id := vyraz
21. <fordef> –> ε
22. <prikaz_prirazeni> –> id <id_n>= <prirazeni>
23. <body> –> ε
24. <body> –> <if> <body>
25. <body> –> <func> <body>
26. <body> –> <for> <body>
27. <body> –> id <id_next> <body>
28. <body> –> <returndef> <body>
29. <body> –> <prirazeni_n> <body>
30. <body> –> EOL <body>
31. <id_next> –> := vyraz
32. <id_next> –> <id_n> = <prirazeni>
33. <prirazeni> –> vyraz <vyraz_n>
34. <prirazeni> –> id (<paramscall>)
35. <vyraz_n> –> , vyraz <vyraz_n>
36. <vyraz_n> –> ε
37. <prirazeni_n> –> _= vyraz

IFJ20	if	func	for	vyraz	id	return	–	EOL	EOF	)	,	int	float	string	=	: ,	}	:=	package
prog																			1
func		2																	
params					3					4									
params_										6	5								
datatype												7	8	9					
navrattype_										11		10	10	10					
paramscall					12														
paramscall_										14	13								
id_											15				16				
returndef						17													
if	18																		
for			19																
fordef					20											21			
prik_riraz					22														
body	24	25	26		27	28	29	30	23								23		
id_ext											32							31	
prirazeni				33	34														
vyraz_	36	36	36		36	36	36	36			35						36		
prirazeni_							37												

Tabulka 1: LL Tabulka

	$	/ *	– +	< > <= >= = == !=	(	)	n
$		<	<	<	<		<
/ *	>	<	>	>	<	>	<
– +	>	<	>	>	<	>	<
< > <= >= = == !=	>	<	<		<	>	<
(		<	<	<	<	=	<
)	>	>	>	>		>	
n	>	>	>	>		>	

Tabulka 2: Zjednodušená tabulka precedenční analýzy



Obrázek 1: DKA pro lexikální analýzu