

IFJ projekt 2022

Tým xnovot2j varianta BVS

Václav Berný xberny00 (20)

Tomáš Drdla xdrdla04 (35)

Libor Novotný xnovot2j (25)

Dalibor Peñas xpenas02 (20)

Bez rozšíření

Obsah

Obsah	2
Rozdělení práce.....	3
Úvod.....	4
Struktury	4
Dynamické pole	4
Zásobník.....	4
Memory tracker.....	4
Tabulka symbolů.....	4
Makefile	4
Lexikální analýza (scanner)	5
Syntaktická a sémantická analýza (parser)	6
LL gramatika.....	6
Neterminály.....	6
Terminály.....	6
Počáteční symbol	6
Popis pravidel.....	6
LL tabulka.....	8
Precedenční analýza pro zpracování výrazů	9
Precedenční tabulka	9
Generování kódu.....	10

Rozdělení práce

Lexikální analýza – Berný, Peñas

Syntaktická analýza – Drdla, Novotný

Sémantická analýza – Drdla

LL gramatika – Novotný

LL tabulka – Berný

Precedenční tabulka – Novotný

Tabulka symbolů – Novotný

Vestavěné funkce – Novotný

Generování výstupního kódu – Novotný, Peñas

Úvod

Cílem projektu bylo vytvořit v jazyce C překladač z jazyku IFJ22 do cílového jazyka IFJcode22.

Struktury

Dynamické pole

Soubory: dArr.c, dArr.h

Přidává struktury pro dynamické pole znaků (dArrC) a void (dArrV). V obou je zapsána jejich velikost a počet prvků. To umožňuje pomocí jednoduchých funkcí přidávat a odebírat prvky bez starosti o velikost dané paměti

Zásobník

Soubory: stack.c, stack.h

Klasický zásobník upravený pro potřeby syntaktické analýzy. Struktura sElem pomáhá tvořit výrazy v postfixové notaci při syntaktické analýze.

Memory tracker

Soubory: memory.c, memory.h

Uchovává odkazy ke všem alokovaným blokům paměti. Lze je potom jednoduše uvolnit funkcí purge.

Tabulka symbolů

Soubory: symtable.c, symtable.h

Pro naši implementaci jsme použili variantu využívající binární vyhledávací strom, kterou jsme si vybrali při přihlašování k projektu. Jako klíče nám posloužily názvy proměnných. Pro každý rámec (hlavní tělo nebo funkce) je pak vždy vygenerován samostatný binární vyhledávací strom, přičemž odkazy na kořenové uzly jsou ukládány v dynamickém poli dArrV.

Uzel

Soubory: node.h

Tento soubor obsahuje seznam typů uzlu a samotný uzel určený pro abstraktní syntaktický strom. Nachází se v samostatném souboru, aby se dal jednoduše použít i v jiných částech programu.

Main

Soubory: main.c

V tomto souboru se nachází funkce main celého překladače.

Makefile

Soubory: Makefile

Slouží k přeložení programu pomocí příkazu make.

Lexikální analýza (scanner)

Soubory: lex.c, lex.h

Náš překladač je lineární, to znamená, že parser zavolá hlavní funkci scanneru (getLexeme) pokaždé, když potřebuje do syntaktického stromu zanést další uzel.

Ze zadání jsme zjistili, že existují 3 typy různých lexémů (naše pojmenování tokenů)

- 1) Jednoznakové reprezentované přechodným znakem automatu.
- 2) Řetězce znaků, které začínají přechodným znakem.
- 3) Řetězce znaků, které jsou:
 - a. Klíčové slovo
 - b. Název funkce
 - c. Číslo
 - d. Řetězec

Každý z nich je speciálně ošetřen. První typ je přímo zkoumaný znak, ale druhý typ potřebuje více pozornosti. Například = je lexém prvního typu a zároveň může být součástí lexému druhého typu ==. Každý z těchto lexémů je samostatně ošetřen. Pro čitelnost funkce getLexeme jsme se rozhodli třetí typ lexémů rozpoznávat až ve funkci createLexeme. V ní se rozhodne, zda se jedná o některý z podtypů (a. b. nebo c. řetězce se hledají už v getLexeme), či lexikální chybu ve vstupním programu.

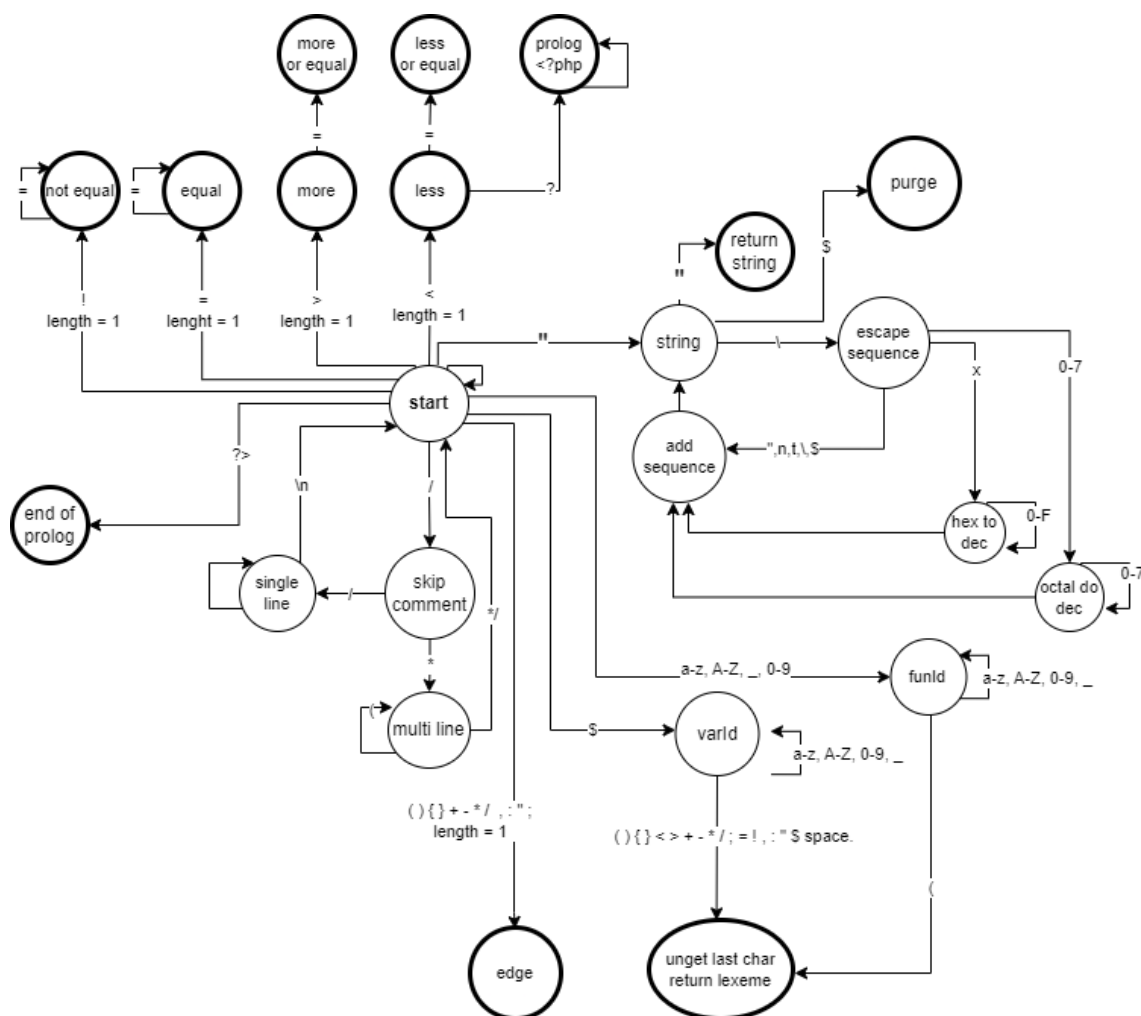


Diagram 1 Diagram konečného automatu specifikující lexikální analyzátor

Syntaktická a sémantická analýza (parser)

Soubory: parse.c, parse.h

LL gramatika

Neterminály

$N = \{ \langle \text{program} \rangle, \langle \text{st-and-func-list} \rangle, \langle \text{st-list} \rangle, \langle \text{stat} \rangle, \langle \text{func-def} \rangle, \langle \text{end-token} \rangle, \langle \text{param-list} \rangle, \langle \text{param-list}' \rangle, \langle \text{data-type} \rangle, \langle \text{return-stat} \rangle, \langle \text{ret-expr} \rangle, \langle \text{input-param-list} \rangle, \langle \text{input-param-list}' \rangle, \langle \text{term} \rangle, \langle \text{expr} \rangle, \langle \text{expr-term} \rangle \}$

Terminály

$T = \{ \text{prolog}, \text{?}, \text{function}, \text{func-id}, \text{id}, \text{()}, \text{\{ }}, \text{\} }, \text{:}, \text{;}, \text{=}, \text{return}, \text{string}, \text{int}, \text{float}, \text{str-val}, \text{int-val}, \text{float-val}, \text{null}, \text{if}, \text{else}, \text{while} \}$

Počáteční symbol

$\langle \text{program} \rangle$

Popis pravidel

Pravidla 1 – 7: obecná struktura programu

4) Pravidlo existuje, protože return může být kdekoliv v programu.

Pravidla 8 – 26: gramatika pro funkce a volání funkcí

9) Neterminál vytvořený pro odstranění levé rekurze označujeme apostrofem.

20) Pokud má return návratovou hodnotu volání funkce, lze ji zpracovat rekurzivním sestupem.

21) Když za příkazem return následuje term, tak to znamená, že se jedná o výraz a je tedy potřeba zastavit rekurzivní sestup a pokračovat precedenční analýzou.

Pravidla 32 – 41: gramatika pro příkazy

33) a 34) V těchto pravidlech je po levé závorce jednoznačně výraz, takže může hned zavolat precedenční analýzu výrazu.

37) Zde se nenachází $\langle \text{term} \rangle$, protože v běžném výrazu se nemůže vyskytnout null a je třeba předat řízení precedenční analýze.

1) <program>	-> prolog <st-and-func-list> <end-token>
2) <st-and-func-list>	-> <func-def> <st-and-func-list>
3) <st-and-func-list>	-> <stat> <st-and-func-list>
4) <st-and-func-list>	-> return <ret-expr> ;
5) <st-and-func-list>	-> ϵ
6) <end-token>	-> ?>
7) <end-token>	-> ϵ
8) <func-def>	-> function func-id (<param-list>) : <data-type> { <st-list> <return-stat> }
9) <param-list>	-> <data-type> id <param-list'>
10) <param-list>	-> ϵ
11) <param-list'>	-> , <data-type> id <param-list'>
12) <param-list'>	-> ϵ
13) <data-type>	-> string
14) <data-type>	-> int
15) <data-type>	-> float
16) <st-list>	-> <stat> <st-list>
17) <st-list>	-> ϵ
18) <return-stat>	-> return <ret-expr> ;
19) <return-stat>	-> ϵ
20) <ret-expr>	-> func-id (<input-param-list>)
21) <ret-expr>	-> <term>
22) <ret-expr>	-> ϵ
23) <input-param-list>	-> <term> <input-param-list'>
24) <input-param-list>	-> ϵ
25) <input-param-list'>	-> , <term> <input-param-list'>
26) <input-param-list'>	-> ϵ
27) <term>	-> str-val
28) <term>	-> int-val
29) <term>	-> float-val
30) <term>	-> null
31) <term>	-> id
32) <stat>	-> id = <expr> ;
33) <stat>	-> if () { <st-list> } else { <st-list> }
34) <stat>	-> while () { <st-list> }
35) <stat>	-> func-id (<input-param-list>) ;
36) <expr>	-> func-id (<input-param-list>)
37) <expr>	-> <expr-term>
38) <expr-term>	-> str-val
39) <expr-term>	-> int-val
40) <expr-term>	-> float-val
41) <expr-term>	-> id

LL tabulka

	prolog	return	ϵ	?>	function	func-id	id	,	string	int	float	str-val	int-val	float-val	null	if	while	\$	(	)	:	{	}	=	else
<program>	A																								
<st-and-func-list>		B	ϵ		C	E	E									K	K								
<end-token>			ϵ	X																					
<func-def>					D																				
<param-list>			ϵ						I	I	I														
<param-list'>			ϵ					, I																	
<data-type>									U	V	W														
<st-list>			ϵ			F	F									L	L								
<return-stat>		B	ϵ																						
<ret-expr>			ϵ			G	O					O	O	O	O										
<input-param-list>			ϵ				H					H	H	H	H										
<input-param-list'>			ϵ					, H																	
<term>							T					P	Q	R	Y										
<stat>						G ;	S									M	N								
<expr>						G	J					J	J	J											
<expr-term>							T					P	Q	R											

Tabulka 1: LL tabulka

A...prolog <st-and-func-list> <end-token>

B...return <ret-expr> ;

C...<func-def> <st-and-func-list>

D...function func-id (<param-list>) : <data-type> { <st-list> <return-stat> }

E...<stat> <st-and-func-list>

F...<stat> <st-list>

G...func-id (<input-param-list>)

H... <term> <input-param-list'>

I...<data-type> id <param-list'>

J...<expr-term>

K...<stat> <st-and-func-list>

L...<stat> <st-list>

M...if () { <st-list> } else { <st-list> }

N...while () { <st-list> }

O... <term>

P... str-val

Q...int-val

R...float-val

S...id = <expr> ;

T... id

U...string

V...int

W...float

X...?>

Y...null

ϵ ... ϵ

Precedenční analýza pro zpracování výrazů

Zpracování výrazů probíhá pomocí této metody syntaktické analýzy zdola nahoru, která vedle toho, že kontroluje, zdali je výraz syntakticky správný, také převádí výraz do postfixové notace. Postfixovou notaci jsme zvolili z důvodu snadného navázání na abstraktní syntaktický strom a později i snazšího generování kódu.

Algoritmus precedenční analýzy je implementovaný v souboru *parse.c* ve funkci *solveExpr()* a je založený na precedenční tabulce níže. V případě, že první terminál na vrcholu zásobníku má větší prioritu než aktuální lexém na vstupu, aplikuje se jedno z následujících pravidel:

1. $E \rightarrow E <op> E$
2. $E \rightarrow (E)$
3. $E \rightarrow i$

Značka $<op>$ značí operátor, „E“ výraz a „i“ konstantu nebo proměnnou. Pokud se žádné pravidlo nemohlo použít, končí analýza chybou. Výraz v postfixové notaci se pak vytvářel tak, že při použití 1. pravidla se do stromu vložil operátor a při použití 3. pravidla příslušná konstanta či proměnná.

Precedenční tabulka

top \ input	$*, /$	$+, -, .$	$<, >, <=, >=$	$==, !=$	$($	$)$	varld, lit, string	$\$$ (empty)
$*, /$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$<$
$+, -, .$	$<$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
$<, >, <=, >=$	$<$	$<$	$>$	$>$	$<$	$>$	$<$	$>$
$==, !=$	$<$	$<$	$<$	$>$	$<$	$>$	$<$	$>$
$($	$<$	$<$	$<$	$<$	$<$	$=$	$<$	
$)$	$>$	$>$	$>$	$>$		$>$		$>$
varld, lit, string	$>$	$>$	$>$	$>$		$>$		$>$
$\$$ (start)	$<$	$<$	$<$	$<$	$<$		$<$	

Tabulka 2: Precedenční tabulka

Generování kódu

Soubory: `genCode.c`, `genCode.h`, `parse.c`

Generování kódu probíhá během syntaktické analýzy, a to konkrétně vždy během řešení neterminálu `<stat>`. Syntaktický analyzátor tedy zavolá příslušnou funkci ze souboru *genCode.c* a jako parametr předá ukazatele na právě řešený uzel `<stat>`. Díky tomu má generátor přístup k podstromu daného příkazu, ze kterého si zjistí informace potřebné ke generování kódu.

Aby se zabránilo redefinici proměnných, nejprve se vždy zjistí, jestli se již nenachází v tabulce symbolů. Pokud ne, vloží se do tabulky symbolů a vytvoří se příslušná instrukce pro její definici. Pokud nalezena je, nic se neděje. Dále bylo potřeba zabránit vytváření návěští se shodnými jmény, k tomuto účelu byly zhotoveny čítače vytvoření určitého typu návěští a název návěští se pak generuje konkatencí názvu typu návěští a pořadí jeho vytvoření.