Vysoké Učení Technické v Brně

Fakulta Informačních Technologií

Dokumentace projektu do předmětů IFJ a IAL

Interpret jazyka IFJ15

Tým 78, varianta b/3/II

20%	xjanou06	Janoušek Lukáš (vedoucí)
20%	xsysel07	Sysel Josef
20%	xmahne00	Mahnert Jakub
20%	xpotoc04	Potoček Patrik
20%	xmalin26	Malina Peter

OBSAH

1. Úvod

Tato dokumentace popisuje implementaci interpretu imperativního jazyka IFJ15.
Jedná se o projekt do předmětu Formální jazyky a překladače(IFJ) a předmětu Algoritmy(IAL). IFJ15 je podmnožinou jazyka C++. Jde pouze o velmi zjednodušenou verzi tohoto staticky typovaného multiparadigmativního jazyka nabízejícího základní odvozování datových typů.

Byla vybrána varianta řešení b/3/II, jenž specifikuje použití vyhledávacího algoritmu Boyer-Mooreův algoritmus, řadícího algoritmu Shell sort a implementaci tabulky symbolů pomocí tabulky s rozptýlenými položkami.

2. Struktura projektu

Projekt lze rozdělit na tři hlavní části, z nichž každé bude věnována jedna samostatná podkapitola.

První z nich bude následovat lexikální analyzátor, který je vstupním blokem pro načítání zdrojového kódu. Jádro celého interpretu tvoří syntaktický analyzátor (Parser), překladač zdrojového kódu. Konečným blokem nazýváme Interpret, který posloupnost instrukcí vykoná.

2.1 Lexikální analýza

Na pomyslné první místo řetězu zpracování vstupního programu patří lexikální analyzátor (scanner). Práce lexikálního analyzátoru spočívá v rozdělení vstupní posloupnosti znaků (abecedy) na tzv. lexémy. Tuto hlavní činnost provádí na základě lexikálních pravidel jazyka. V našem případě souborem pravidel pro jazyk IFJ15.

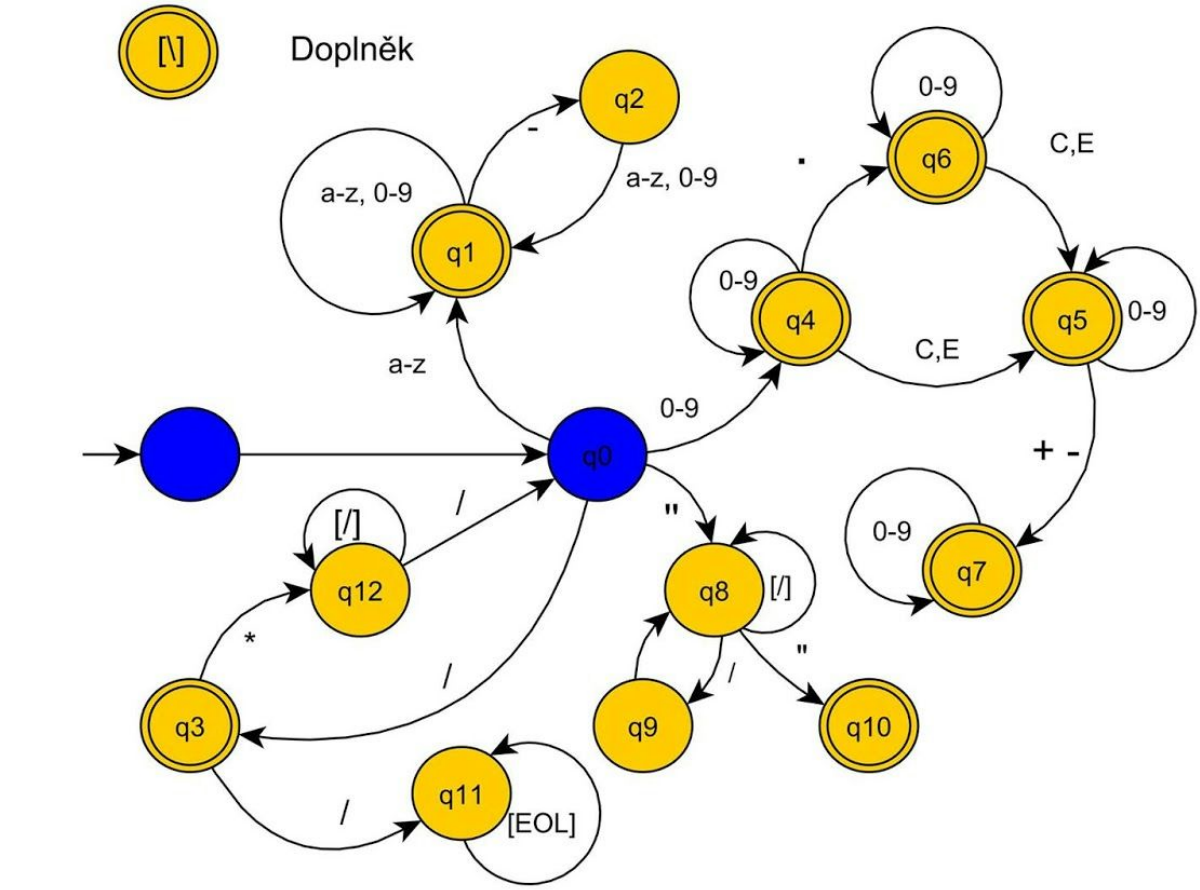
Lexikální analyzátor je implementován v souborech scanner.c a scanner.h. Veškerá komunikace s okolím probíhá pomocí funkce getToken(), která je součástí parseru a postupně načítá znaky ze vstupního souboru.

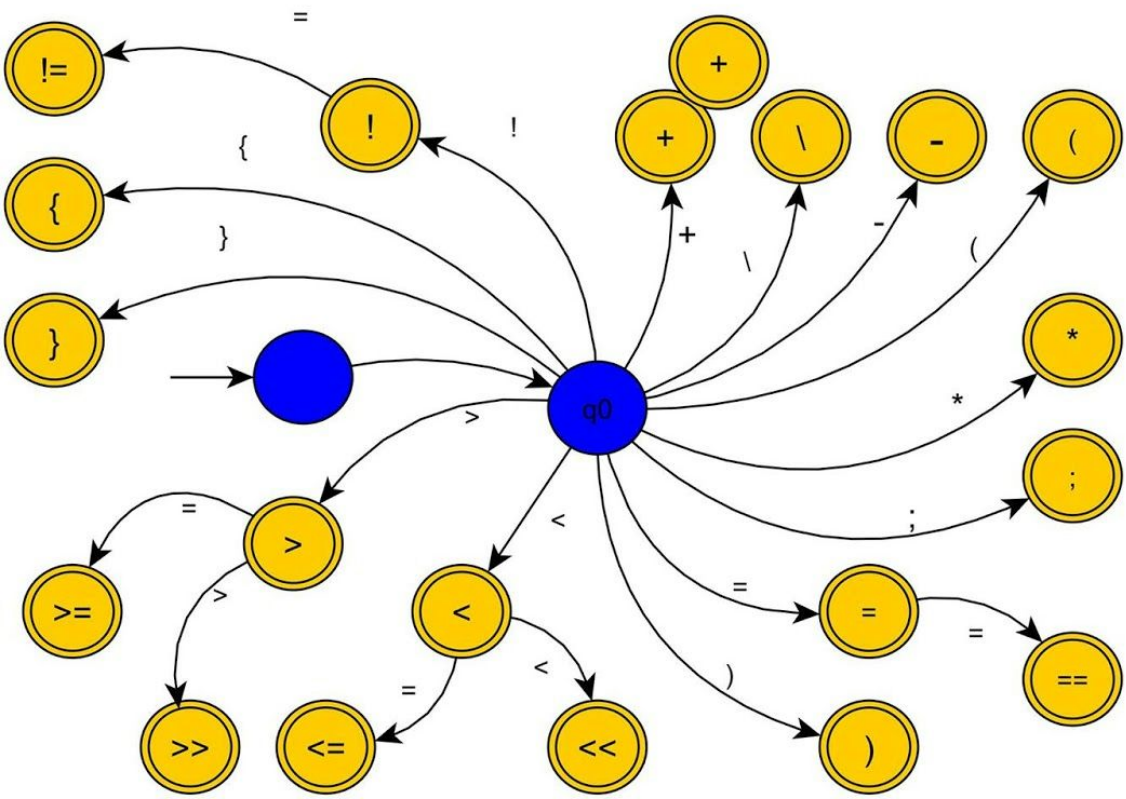
Lexémy jsou prakticky reprezentovány tokeny, což jsou výstupy lex. analyzátoru, o které si postupně žádá syntaktický analyzátor a tím ho i řídí. Spolu se samotným lexémem předá i načtenou hodnotu a číslo řádku, na kterém se vyskytoval. Pokud by
tento lexém způsobil chybu překladu, zobrazí uživateli chybovou hlášku.

Nezbytně důležitý úkol lexikálního analyzátoru je také odstranění veškerých komentářů a bílých znaků, ze vstupního zdrojového programu.

K návrhu lexikálního analyzátoru jsme použili konečný deterministický automat.

Návrhem toho speciálního automatu jsme schopni přesně rozpoznat množiny slov. Aktuální stav konečného automatu vždy vyjadřuje, jaký konkrétní lexém byl detekován.
Přesný popis tohoto konečného automatu je na obr. 1.


[\]
Doplněk
q0
q1
q2
q3
q4
q5
q6
q7
q8
q9
q10
q11
q12
a-z, 0-9
-
a-z
0-9
.
C,E
+ -
[/]
/
*
"
[EOL]


q0
!=
=
!
{
}
+
\
-
(
*
;
>
>=
>>
<
<=
<<
=
==
)

2.2 Syntaktická a sémantická analýza

Syntaktický analyzátor, neboli parser, tvoří jádro celého překladače. Tento analyzátor se stará o přeložení zdrojového kódu v jazyce IFJ15 na posloupnost pseudoinstrukcí. Hlavním cílem je kontrola syntaktické korektnosti vstupního programu. Syntakticky správná konstrukce, je definována pomocí LL gramatiky, což je předem definovaná formální gramatika.

Komunikace s lexikálním analyzátorem probíhá pomocí metody getToken(), která je volána vždy, když syn. analyzátor požaduje další token. Syntaktický analyzátor dále komunikuje se sémantickým voláním požadovaných sémantických akcí.

Syntaktický analyzátor tokeny transformuje do speciálně navržené datové struktury, které říkáme derivační strom. Pokud se derivační strom nepodaří sestrojit, program obsahuje syntaktickou chybu. Náš analyzátor pracuje na principu rekurzivního sestupu, který vychází z LL tabulky vytvořené z LL gramatiky.

program_body -> function_definition zbytek_tela
program_body_remainder -> EOF
program_body_remainder -> function_definition zbytek_tela

function_definition -> data_type IDENTIFIER LEFT_PARENTHESIS function_arguments
RIGHT_PARENTHESIS LEFT_BRACE program_block RIGHT_BRACE
data_type -> KEYWORD_INT
data_type -> KEYWORD_STRING
data_type -> KEYWORD_DOUBLE
data_type -> KEYWORD_AUTO
function_arguments -> NULL
function_arguments -> data_type IDENTIFIER function_argument_pump
function_argument_pump -> NULL
function_argument_pump -> COMMA data_type IDENTIFIER function_argument_pump

program_block -> NULL
program_block -> statement program_block

statement -> var_creation SEMICOLON
statement -> if
statement -> return
statement -> cout
statement -> cin
statement -> for
statement -> generic_id SEMICOLON
statement -> LEFT_BRACE program_block RIGHT_BRACE
statement -> expression

var_creation -> data_type IDENTIFIER assign
assign -> NULL
assign -> ASSIGN EXPRESSION

```
if -> IF LEFT_PARENTHESIS EXPRESSION RIGHT_PARENTHESIS LEFT_BRACE program_block
RIGHT_BRACE
return -> RETURN EXPRESSION SEMICOLON

cout -> COUT OUTPUT_OPERATOR EXPRESSION cout_pump
cout_pump -> SEMICOLON
cout_pump -> OUTPUT_OPERATOR EXPRESSION cout_pump

cin -> CIN INPUT_OPERATOR IDENTIFIER cin_pump
cin_pump -> SEMICOLON
cin_pump -> INPUT_OPERATOR IDENTIFIER

for -> FOR LEFT_PARENTHESIS for_first_field SEMICOLON EXPRESSION SEMICOLON generic_id
RIGHT_PARENTHESIS LEFT_BRACE program_block RIGHT_BRACE
for_first_field -> var_creation
for_first_field -> IDENTIFIER ASSIGN EXPRESSION

generic_id -> IDENTIFIER assign
generic_id -> IDENTIFIER LEFT_PARENTHESIS function_call_arguments RIGHT_PARENTHESIS
function_call_arguments -> IDENTIFIER function_call_argument_pump
function_call_argument_pump -> NULL
function_call_argument_pump -> COMMA IDENTIFIER function_call_argument_pump
```

Sémantický analyzátor bezprostředně pracuje s výstupem syntaktické analýzy, kde se nachází binární strom. Hlavním úkolem je kontrolovat korektnost operací zapsaných ve zdrojovém programu. Sémantický analyzátor dále zohledňuje prioritu operací (např.: násobení má větší prioritu, než sčítání). Výsledek práce sémantického analyzátoru se nazývá abstraktní syntaktický strom.

Pro zpracování výrazu je použita precedenční syntaktická analýza řízená precedenční tabulkou, která udává prioritu a asociativitu všech operátorů.
V této tabulce jsou definována syntaktická pravidla pro všechny výrazy. Samotný algoritmus je pak tvořen cyklem, ve kterém jsou postupně vyhodnocována a redukována pravidla získaná z precedenční tabulky. Jako pomocná datová struktura zde slouží zásobník.

// TODO: precedencni tabulka

2.3 Interpret

Interpret je poslední fáze zpracování zdrojového programu. Na vstup interpretu přichází posloupnost instrukcí tříadresného kódu vygenerovaného po úspěšném skončení syntaktické a sémantické analýzy. Tříadresný kód je uložen v instrukčním listu, který je implementován pomocí abstraktního datového typu

jednosměrně vázaným lineárním seznamem. Dalším vstupem je samotný vstup zdrojového programu.

Interpret postupně prochází seznam instrukcí, implementovaných jako pole. Podle typu aktuální instrukce vykonává interpret odpovídající určité akce.

Při načtení instrukce značící definici funkce dojde k “přeskočení”. Pokud je načteno volání funkce, interpret vytvoří zásobník, na který uloží návratovou hodnotu, parametry funkce, adresu proměnné a místo pro zápis návratové hodnoty. Po úspěšném provedení zmiňovaných akcí je proveden skok na místo uložení definice volané funkce, kde je funkce standardně provedena. Nakonec musí interpret při návratu z funkce zapsat návratovou hodnotu a provést skok na adresu, kde musí pokračovat v generování programu. Vytvořený zásobník je opět uvolněn z paměti.

3. Implementace vybraných algoritmů

3.1 Boyer-Mooreův algoritmus

Algoritmus se používá ve vyhledáváni podřetězce v řetězci u vestavěné funkce find([retezec],[hledanyPodretezec]). Tento algoritmus spočívá ve vyhledávání shodných respektive neshodných znaků mezi řetězcem a podřetězcem. Boyer-Mooreův algoritmus umí „přeskočit“ některé části řetězce bez nutnosti jejich porovnání, což vede ke zvýšení rychlosti celého programu.

3.2 Shell sort

Pro náš překladač byl zvolen řadící algoritmus Shell sort. Využívá tzv. snižující se přírůstek. To znamená, že algoritmus neřadí prvky, které jsou přímo vedle sebe, ale prvky, mezi nimiž je určitá mezera (tj. první a pátý, pátý a devátý, druhý a šestý...). V každém kroku je pak mezera mezi prvky zmenšena. V okamžiku, kdy se velikost mezery sníží na 1, dojde k řazení sousedních prvků – algoritmus degeneruje na běžný insertion sort. Výhodou tohoto poněkud komplikovaného přístupu je, že jsou prvky vysokých a nízkých hodnot velmi rychle přemístěny na odpovídající stranu pole. Poslední iterace algoritmu (insertion sort) pak již přesouvá naprosté minimum prvků.

3.3 Tabulka symbolů