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 překladače 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.

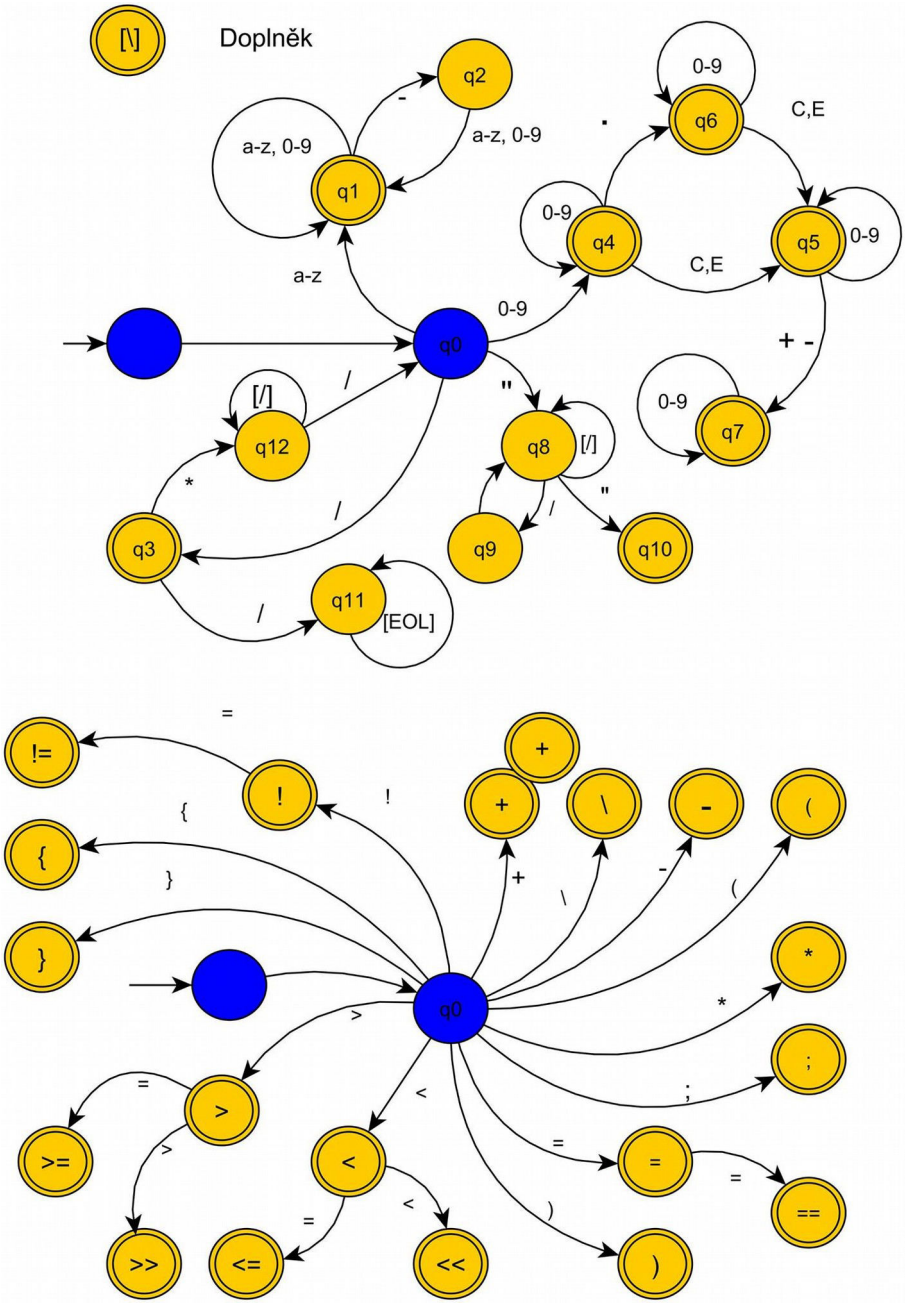
Lexikální analyzátor je implementován v souborech scanner.c a scanner.h. Veškerá komunikace s okolím probíhá pomocí funkce get_token(), která je součástí syntaktického analyzátoru a zařizuje postupné načítání znaků ze vstupního souboru.

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 byl použit konečný deterministický automat.

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



2.2 Syntaktická analýza

Syntaktický analyzátor, neboli parser, tvoří jádro celého překladače. Tento analyzátor se stará o přeložení zdrojového kódu v jazyce IFJ15 na abstraktní syntaktický strom (dále jen AST). Hlavním cílem je kontrola syntaktické korektnosti vstupního programu. Syntakticky správná konstrukce, je definována pomocí LL gramatiky, což je předem definovaná formální gramatika. Více na obr.2.

Komunikace s lexikálním analyzátorem probíhá pomocí metody get_token(), která je volána vždy, když syn. analyzátor požaduje další token. Vstupem je tedy retězec tokenů, které mu průběžně zasílá lexikální analyzátor. V závislosti na přijímané posloupnosti tokenů je sestavován AST. Výstupem následně jsou informace o tom, zda je program syntakticky správně. Pokud se AST nepodaří sestrojit, program obsahuje syntaktickou chybu.

Náš analyzátor pracuje na principu rekurzivního sestupu, který vychází z LL tabulky vytvořené LL gramatikou. Každé ze slov této gramatiky je reprezentováno jednou metodou, jejímž úkolem je zpracovat dané slovo, pro které je určena.

Metody je dále možné postupným voláním řetězit v závislosti na očekávaném pořadí slov za sebou. Všechny volané metody odpovídající určitému slovu jsou zabaleny do speciálního makra, nesoucího název EXPECT. Učelem makra je zajistit kontrolu správnosti všech metod získaných voláním. Pouze případě validnosti všech součástí slova, můžeme říci, že syntaktická analýza proběhla korektně. Každá z volaných metod vytvoří svůj AST, který uloží do proměnné, přicházející jako parametr.

Na výstupu parseru se nachází kompletní AST, složený z jednotlivých AST od použitých metod. Celý výsledný obsah je uložen do sdílené datové struktury, který je dál předáván interpretu jako argument. Správnou komunikaci a přenos dat mezi bloky (parser a interpret) zajišťuje interface s názvem parse_run.

1: program_body → function_definition zbytek_tela
2: program_body_remainder → EOF
3: program_body_remainder → function_definition zbytek_tela
4: function_definition → data_type IDENTIFIER LEFT_PARENTHESIS function_arguments RIGHT_PARENTHESIS LEFT_BRACE program_block RIGHT_BRACE
5: data_type → KEYWORD_INT
6: data_type → KEYWORD_STRING
7: data_type → KEYWORD_DOUBLE
8: data_type → KEYWORD_AUTO
9: function_arguments → NULL
10: function_arguments → data_type IDENTIFIER function_argument_pump
11: function_argument_pump → NULL
12: function_argument_pump → COMMA data_type IDENTIFIER function_argument_pump
13: program_block → NULL
14: program_block → statement program_block
15: statement → var_creation SEMICOLON
16: statement → if
17: statement → return
18: statement → cout
19: statement → cin
20: statement → for
21: statement → generic_id SEMICOLON
22: statement → LEFT_BRACE program_block RIGHT_BRACE
23: statement → expression
24: var_creation → data_type IDENTIFIER assign
25: assign → NULL
26: assign → ASSIGN EXPRESSION
27: if → IF LEFT_PARENTHESIS EXPRESSION RIGHT_PARENTHESIS LEFT_BRACE program_block RIGHT_BRACE
28: return → RETURN EXPRESSION SEMICOLON
29: cout → COUT OUTPUT_OPERATOR EXPRESSION cout_pump
30: cout_pump → SEMICOLON
31: cout_pump → OUTPUT_OPERATOR EXPRESSION cout_pump
32: cin → CIN INPUT_OPERATOR IDENTIFIER cin_pump
33: cin_pump → SEMICOLON
34: cin_pump → INPUT_OPERATOR IDENTIFIER
35: for → FOR LEFT_PARENTHESIS for_first_field SEMICOLON EXPRESSION SEMICOLON generic_id RIGHT_PARENTHESIS LEFT_BRACE program_block RIGHT_BRACE
36: for_first_field → var_creation
37: for_first_field → IDENTIFIER ASSIGN EXPRESSION
38: generic_id → IDENTIFIER assign
39: generic_id → IDENTIFIER LEFT_PARENTHESIS function_call_arguments RIGHT_PARENTHESIS
40: function_call_arguments → IDENTIFIER function_call_argument_pump

41: function_call_argument_pump → NULL
42: function_call_argument_pump → COMMA IDENTIFIER function_call_argument_pump

2.3 Interpret

Interpret je poslední fáze zpracování zdrojového programu. Na vstup interpretu přichází pomocí volaného argumentu abstraktní syntaktický strom, vygenerovaný po úspěšném skončení syntaktické analýzy. Dalším vstupem je samotný vstup zdrojového programu.

Prvním úkolem interpretu je traverzování vstupního AST. ...

Kontroly, které nevykonal syntaktický analyzátor má na starosti sémantická analýza.

Finálním úkolem interpretu je samotná interpretace již kompletně připraveného vstupu.

Shunting Yard
Algoritmus čte předáváné tokeny znak po znaku a jednotlivé operandy a operátory vkládá do pomocného zásobníku, nebo výstupní fronty. Při zpracování dochází ke konverzi tokenů na uzly AST. Fronta bude po ukončení algoritmu obsahovat přímo AST.

3. Implementace vybraných algoritmů

3.1 Boyer-Mooreův algoritmus

Algoritmus se používá ve vyhledáváni podřetězce v řetězci u vestavěné funkce find(řetězec, hledaný_podřetězec). 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. 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

S tabulkou symbolů pracuje interpret pomocí listů v AST definovaných jako var, create a nebo assign. Jde o zásobník, ve kterém se uchovávají hash table proměnných. ...

4. Vývoj

4.1 Rozdělení činnosti

Již na první týmové schůzce bylo navrženo základní rozdělení práce na projektu. Během vývoje se o rozdělení činností dále diskutovalo a jednotlivé činnosti, zahrnující i konkrétní úkoly byli doplňovány.

Janoušek Lukáš – Lexikální analyzátor, prezentace, dokumentace.
Sysel Josef – Lexikální analyzátor, dokumentace, vestavěné funkce.
Mahnert Jakub – Syntaktický analyzátor, testování.
Potoček Patrik – Interpret, testování.
Malina Peter – Interpret, testování.

4.2 Použité prostředky

Při vývoji je nezbytně nutné urdžovat aktivní komunikaci mezi jednotlivými členy. Hlavním komunikačním prostředkem se stala sociální síť Facebook, která nabízí možnosti soukromého sdílení a uchování informací s dostupným online chatem. Mimo jiné probíhalo každý týden osobní setkání.

Pro společný vývoj programové části byl využit verzovací systém Git, při čemž jako hosting pro náš repozitář jsme použili službu GitHub.

4.3 Vývojové metodiky

Od počátku vývoje bylo rozumné určit metodiku vývoje, kterou se bude tým programátorů snažit do maximální míry naplňovat.

Náš tým vychází z úsudku, že při psaní kódu mohou vzniknout velmi často chyby, a dále z úvahy, že některé myšlenky je vhodné si ověřit testováním, než budou zahrnuty do projektu. Dále jsme počítali od začátku projektu s častými změnami ve specifikaci, nebo s postupným procesem chápání projektu jako celku.

Proto náš tým zvolil iterační způsob vývoje. Tato metoda využívá postup: Návrh (analýza)→programování→testování. Vývoj byl rozdělen na krátké iterace, které byli přizpůsobeny rozsahu konkrétní implementace. Jednou za týden proběhla schůza všech členů skupiny, kde byla probírána týmová strategie, technické otázky a jednotlivé úkoly na další setkání. Lepší orientaci v kódu zajišťoval důraz na psaní komentářů a povinný popis provedených změn ve sdíleném repozitáři.

5. Závěr

5.1 Shrnutí

5.2 Použitá literatura

- přednášky a studijní opora předmětu IFJ
- přednášky a studijní opora předmětu IAL

5.3 Metriky kódu