Provozně ekonomická fakulta

Dokumentácia k projektu “*tpj-parser*” do predmetu TPJ - Teorie programovacích jazyků

Autori:	Bc. Tibor Dudlák	xdudlak
	Bc. Jan Hrbotický	xhrboti1
	Bc. Radovan Sroka	xsroka

15. Jan 2019

Obsah

Úvod

Tento dokument vznikol ako dokumentácia k tímovému projektu tpj-parser dostupného na GitHub-e [1] predmetu TPJ - Teorie programovacích jazyků.

Jazyk

Tento projekt implementuje interpreter pre jazyk podobný jazyku C++ s určitými obmedzeniami. Jazyk podporuje deklaráciu celočíselných premenných (int), binárnych premenných (bool), premenných držiacich hodnoty čísel s plavajúcou desatinou čiarkou (float), premenných pre reťazce znakov (string) a definíciu funkcií ktoré majú návratovú hodnotu (int, bool, float, string, void - bez návratovej hodnoty). Interpreter taktiež podporuje základné programové konštrukcie cyklov (while) a vetvenia (if - else).

V prvej kapitole *Lexikálna analýza* je opísaná implementácia scanneru tokenov pre jazyk pre jazyk ktorý je opísaný gramatikou v druhej kapitole *Syntaktická a sémantická analýza* spolu s implementovanými detailmi. Tretia kapitola *Interpreter* je venovaná samotnému vykonávaniu kódu a návrhu inštrukčnej sady projektu.

Lexikálna analýza

Autor: Jan Hrbotický (xhrbot1)

Lexikální analýza pomocí tzv. scanneru načte ze vstupu program, který zpracuje. Na výstupu je poté posloupnost tzv. tokenů, což je objekt reprezentující jeden lexém programu a informace o něm.

Štruktúra

- Lex.cpp – zdrojové kódy scanneru
- Lex.hpp – hlavičkový soubor obsahující deklarace metod a enum všech možných stavů konečného automatu
- Token.cpp – zdrojové kódy objektu typu token
- Token.hpp – hlavičkový soubor obsahující definici objektu a enum všechny možných typů tokenů

Princíp funkčnosti

Princip funkčnosti je popsán pouze okrajově, veškeré informace o metodách a funkčnosti lexikální analýzy naleznete ve zdrojových souborech.

Inicializace objektu lexikálního scanneru

Objekt má jediný parametr. Tím je vstupní program ve formě reference na objekt std::istream. Po načtení vstupu je nastaven počáteční stav konečného automatu a inicializován tzv. tokenpool pro objekty typu token. Tento přístup nám umožňuje snadnou správu paměti. Všechny inicializované objekty jsou totiž řízeny odjinud a v programu se pracuje pouze s referencemi na ně. Po ukončení chodu programu je pak veškeré alokovaná paměť automaticky uvolněna.

Získání tokenu

Hlavní činnost scanneru, tedy získání tokenu, je vyvolána metodou Token& getToken();. Ta se dle stavu zásobníku vrácených tokenů rozhodne, zda-li vrátí navrácený objekt ze zásobníku zavoláním Token& Lex::getTokenFromStack() – to se děje v případě, že je zásobník neprázdný, anebo se načte ze vstupu za pomocí metody Token& Lex::getTokenFromFile().

Načtení ze zásobníku

Metoda v tomto případě pouze využije metody pop, tedy vrátí referenci na daný objekt a odstraní ho z vrcholu zásobníku.

Načtení ze vstupu

Nejprve se při čtení nového tokenu inicializují prvky, se kterými se posléze pracuje. Tím je objekt tokenu, který je získán s tokenpoolu a znak, do kterého se bude ukládat aktuální vstup.
Samotné zpracování je implementováno za pomocí konečného automatu, který po znacích čte ze vstupu a dle svého, s časem se měnícího, stavu tyto posloupnosti znaků zpracovává. Bližší informace o fungování konečného automatu naleznete níže v grafu. Struktura tokenu vlastní text, id a typ. Metody, které objekt obsahuje jsou pak primárně pouze settery a gettery.

Dalším zajímavým aspektem jsou rezervovaná slova. Implementace spočívá pouze v tom, že má-li být vrácen objekt typu identifier, je nad ním volána metoda void keywordCheck(Token& token), která pouze zkontroluje, zdali není text (tedy textový obsah objektu) shodný s některým z rezervovaných slov. K tomu je využita tzv. mapa stringu na enum typu tokenu. Nalezne-li se takový index (tzn. že token je rezervované slovo), změní se mu jeho typ a je úspěšně navrácen. V opačném případě se jedná o identifikátor a objekt projde metodou beze změny.

Vrácení tokenu

Jednou z požadovaných vlastností byla také implementace tzv. unget tokenu. Tedy proceduře, při které je ze syntaktické analýzy navrácen token zpět. Implementace volně přístupné metody void Lex::ungetToken(Token& t) je pak triviální. Pouze vloží referenci objektu předaného v argumentu na vrchol zásobníku. V tomto momentě je tedy stack neprázdný a při další žádosti o token bude vrácen právě tento objekt.

Podpora jednotkových testů

Z důvodu nutnosti otestovat jednotlivé stavy konečného automatu bylo třeba doimplementovat některé pomocné nástroje. Primárně tedy převod ENUM hodnot na string. To se děje opět za pomocí mapy, tentokráte ovšem ENUM na string. Díky tomuto lze na výpis propagovat lépe čitelné hodnoty, díky kterým je ladění mnohem snazší.

Samotné jednotkové testy můžete nalézt ve složce /tests. Jsou nazvány jako lex_testX, kde X je číslo označující daný test. Na prvních řádcích je vždy popsáno, na co se zaměřuje.

Syntaktická a sémantická analýza

Autor: Tibor Dudlák (xdudlak). Radovan Sroka (xsroka)

Štruktúra

- syntax
    - Syntax.cpp
    - Syntax.hpp
- semantic
    - Semantic.cpp
    - Semantic.hpp
    - SymbolTable.cpp
    - SymbolTable.hpp
    - SymbolTableItem.cpp
    - SymbolTableItem.hpp

Princíp funkčnosti

Syntaktická analýza je implementovaná rekurzívne zanorením sa do pravidiel gramatiky jazyka a postupným vynorovaním pri úspešnom rozbalení pravila. Je implementovaná v metóde parseSyntax(int, int); súboru **Syntax.cpp** v podadresári ***syntax***. Tabuľka tejto gramatiky je priložená v tejto kapitole a jej odseku: *Pravidlá gramatiky jazyka akceptovaného interpreterom tpj-parser*.

Spolu s kontrolovaním syntaxe vďaka zostrojeným pravidlám je za behu kontrolovaná aj sémantika pomocou objektu SymbolTable implementovaného v podadresári ***semantic*** v súboroch **SymbolTable.cpp** a **SymbolTableItem.cpp** sa kontroluje výskyt premenných v rámci kontextu v ktorom sa nachádzame, či už je to globálny alebo samostatný pre každé telo funkcie.

Pravidlo EXPRESSION odkazuje na metódu ParseExpression(); ktorá využíva vyhodnocovanie výrazov pomocou precedenčnej tabuľky ktorej autorom je kolega Radovan Sroka.

Trieda Syntax umožňuje samostatnu kontrolu syntaxe s možnosťou "zapnutia" sémantickej nastavením privátnej premennej _semanticsCheck zavolaním metódy setSemanticsCheck(false); s parametrom "false". Týmto sa zaistilo možnosť testovania syntaxe oddelene od sémantiky.

Špecialitou tejto implementácie je možnosť prepnutia kompatibility na klasické definovanie funkcií v tele programu alebo používania ako in-line interpretera s in-line funkciami. Táto funkcionalita sa prepína pomocou parametru nasledovne:

- parseSyntax(INLINE_START, INLINE_START); - interpreter s in line definíciami podobne ako Python
- parseSyntax(START, START); - klasické C/C++ definície

pričom INLINE_START a START sú štartovacie stavy pre jazyk prijímaný interpreterom.

Pravidlá gramatiky jazyka akceptovaného interpreterom tpj-parser

RULES		
START	⇒	FUNCTIONS
INLINE_START	⇒	ε
	⇒	IFSTMNT CONT_INLINE
	⇒	WHILESTMT CONT_INLINE
	⇒	STATEMENTS CONT_INLINE
	⇒	FUNCDECL_INLINE CONT_INLINE
CONT_INLINE	⇒	ε
	⇒	**INLINE_START CONT_INLINE**
FUNCDECL_INLINE	⇒	**FUNCDECL**
FUNCTIONS_CONT	⇒	FUNCDECL
FUNCTIONS	⇒	FUNCDECL FUNCTIONS_CONT
	⇒	ε
FUNCDECL	⇒	HEAD TAIL
TAIL	⇒	{ BODY }
HEAD	⇒	DECL (ARGUMENTS)
DECL	⇒	TYPE NAME
ARGUMENTS	⇒	ε
	⇒	DECL ARGUMENTS_N
ARGUMENTS_N	⇒	ε
	⇒	, DECL ARGUMENTS_N
BODY	⇒	STATEMENT CONTSTMNT
CONTSTMNT	⇒	ε
	⇒	BODY
STATEMENT	⇒	STATEMENTS
	⇒	IFSTMNT
	⇒	WHILESTMT

STATEMENTS	⇒	STATEMENTBODY ;
STATEMENTBODY	⇒	VARIABLEDEF
	⇒	RETURNSTMT
	⇒	OPERATION
OPERATION	⇒	NAME OPERATION_KIND
OPERATION_KIND	⇒	FUNCTIONCALL
	⇒	ASSIGN
VARIABLEDEF	⇒	TYPE *NAME* ASSIGN VARIABLEDEF_N
VARIABLEDEF_N	⇒	ε
	⇒	, TYPE NAME ASSIGN VARIABLEDEF_N
ASSIGN	⇒	ε
	⇒	= EXPRESSION
IFSTMNT	⇒	if (EXPRESSION) { BODY } ELSEBODY
ELSEBODY	⇒	ε
	⇒	else { BODY }
WHILESTMT	⇒	while (EXPRESSION) { BODY }
RETURNSTMT	⇒	return EXPRESSION
TYPE	⇒	int
	⇒	string
	⇒	bool
	⇒	float
FUNCTIONCALL	⇒	(PARAMETERS)
PARAMETERS	⇒	ε
	⇒	EXPRESSION, PARAMETERS_N
PARAMETERS_N	⇒	ε
	⇒	EXPRESSION, PARAMETERS_N

Interpreter

Autor: Radovan Sroka (xsroka)

Princíp

Tento interpret funguje na jednoduchom princípe postupného vykonávania inštrukcii. Skladá sa z vektoru inštrukcii cez ktorý sa iteruje zásobníku nad ktorým sa vykonávajú operácie dane typom inštrukcii. Aktuálnu polohu vykonávanej inštrukcie nám udáva tzv. IP (instruction pointer) a začiatok aktuálneho scope udáva tzv. BP (base pointer).

Štruktúra

- Interpet.cpp – zdrojové kódy interpretu
- Interpret.hpp – hlavičkový súbor obsahujúci deklarácie metód

Implicitné pretypovania

- Interpret sa snaží za každú cenu previesť operandy tak aby ich dokázal nejak z operovať.
- Pre binárne operátory platí že sa vždy prevádza druhý operand podľa typu prvého. V prípade logických operátorov sa oba operandy prevádzajú na BOOL.
- Platí že výsledok vyhodnotenej podmienky pri IF a WHILE sa tiež prevádza na BOOL.
- Pri pretypovaní String-u na číslo sa prevedie iba tá začínajúca časť reťazca ktorú je možné previesť inak je výsledok 0.

Volanie funkcií

Volanie funkcií prebieha nasledovne:

1. CALL inštrukcia s adresou na funkciu ako operand
2. uložíme IP
3. uložíme staré BP
4. nastavíme nové BP
5. skočíme na funkciu

Návrat z funkcie:

1. RET inštrukcia
2. odmažeme zo zásobníka všetko až po BP
3. hodnota na vrchole zásobníka je staré BP tak ho obnovíme
4. obnovíme IP
5. zanecháme návratovú hodnotu ak funkcia niečo vracia

Vstavané funkcie

Tento interpreter ma hneď niekoľko vstavaných funkcií pripravených na použitie.

Cast-y

- int castF2I (float)
- int castB2I (bool)
- int castS2I (string)
- float castI2F (int)
- float castB2F (bool)
- float castS2F (string)

Print-y

- void printINT (int)
- void printFLOAT (float)
- void printBOOL (bool)
- void printSTRING (string)

Inštrukčná sada

- NOP – prázdna inštrukcia
- SUM – spočíta vrchol zásobníka
- SUB – odčíta vrchol zásobníka
- MUL – vynásobí vrchol zásobníka
- DIV – vydelí vrchol zásobníka
- NEG
    - neguje vrchol zásobníka
    - výsledok bool
- MINUS – unárne mínus nad vrcholom zásobníka
- OR

- AND
- EQ
- MORE
- LESS
- MOEQ
- LOEQ
- NEQ
    - všetky porovnávacie operácie nad vrcholom zásobníka
    - výsledok bool
- CALL
    - zavolá funkciu
    - skok, nastavenie BP/IP
- RET – návrat z funkcie
- PUSH – pridáva niečo na zásobník
- POP – maže vrchol zásobníka
- POPN – maže N prvkov zo zásobníka
- LOAD – načítanie premennej na vrchol zásobníka
- STORE – uloženie vrcholu zásobníka do premennej
- SAVEBP – uloženie base pointra na vrchol zásobníka
- RESTOREBP – nastavenie base pointra z hodnoty na vrchole zásobníka
- JUMP – skok na danú adresu
- JUMPIFNOTTRUE – skok podmienený vrcholom zásobníka
- DUP – duplikácia vrcholu zásobníka
- PRINT – výpis vrcholu zásobníka
- CASTINT – pretypovanie vrcholu zásobníka na INT
- CASTFLOAT – pretypovanie vrcholu zásobníka na FLOAT
- CASTBOOL – pretypovanie vrcholu zásobníka na BOOL
- CASTSTRING – pretypovanie vrcholu zásobníka na String

Referencie

[1] tpj-parser, [online]. [cit. 2019-01-15]. Dostupné z: https://github.com/radosroka/tpj-parser