Projektová dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ18

Tým 069, variant I

27. november 2018

Autori:

Tomáš Sasák	**xsasak01**	25%
Tomáš Venkrbec	xvenkr01	25%
Martin Krajči	xkrajc21	25%
Natália Dižová	xdizov00	25%

1 Úvod

Cieľom dokumentácie je objasniť postup pri vypracovaní skupinového projektu do predmetu IFJ a IAL. Výsledný vypracovaný projekt pracuje so zdrojovým kódom v zdrojovom jazyku IFJ18, ktorý načíta zo štandardného vstupu a preloží ho do cieľového jazyka IFJcode18, teda generuje výsledný medzikód na štandardný výstup.

V tejto dokumentácii nájdete postup pri riešení tohto projektu a popis implementácie jeho základných častí.

2 Návrh a implementácia

2.1 Lexikálna analýza

Pre umožnenie fungovania ostatných častí, je nutné vždy najskôr spustiť lexikálnu analýzu. Tá je implementovaná ako konečný automat. Počas lexikálnej analýzy sú vždy vo funkcii `get_token` postupne načítané jednotlivé znaky zo vstupného súboru, do tej doby, pokiaľ automat nenarazí na reťazec, ktorý reprezentuje identifikátor, reťazcový, desatinný alebo celočíselný literál, operátor, podporovaný špeciálny znak, koniec riadka alebo koniec súboru. Tieto dáta sú predávané syntaktickej analýze v štruktúre `tToken`, ktorá obsahuje typ tokenu a úniu, v ktorej je podľa typu tokenu uložený buď dynamický string (pre reťazcové literály a identifikátory), alebo hodnota dátového typu integer alebo float pre číselné literály. Ako samotná štruktúra, tak aj všetky typy prijímaných tokenov sú definované v súbore `scanner.h`.

(implementácia v súbore `scanner.c`)

2.2 Syntaktická analýza

Z dvoch možností implementácie syntaktickej analýzy, sme si vybrali rekurzívny zostup. Tento spôsob nám prišiel jednoduchší na implementáciu a predstavenie, ale toto malo aj druhý následok a to bolo zložité debugovanie problémov. V súbore `parser.h`, je vytvorená štruktúra `tPData`, ktorá predstavuje dáta, ktoré používa syntaktická a aj sémantická analýza pre správny chod, analýzu a spracovanie užívateľského programu. Táto štruktúra sa skladá z nasledujúcich položiek, `tToken token` značí aktuálny načítaný token lexikálnou analýzou, `tSymTable *local` ukazatel na lokálnu tabulku symbolov, `tSymTable *global` ukazatel na globálnu tabulku symbolov, `tIList *instrs` ukazatel na aktuálne vygenerovaný vnútorný kód, `int scopes` počítadlo zanorení, `bool inDefinition` značí stav keď analýza prebieha pre definíciu funkcie, `int currentLine` počítadlo aktuálneho riadku na ktorej sa analýza nachádza (pri chybe, oznamuje užívatelovi, na ktorom riadku sa chyba nachádza). Syntaktická analýza pracuje, presne podľa pravidiel LL-gramatiky. Syntaktická analýza žiada lexikálnu analýzu o token, pomocou makra `GET_TOKEN`.

(implementácia v súbore `parser.c`)

2.2.1 Syntaktická analýza matematických výrazov

Matematické výrazy sa spracujú pomocou volania funkcie `pars_expression`. Táto funkcia pracuje so zásobníkom v štruktúre `tPData`. Na základe precedenčnej tabuľky rozhodne, ktorú operáciu pre spracovanie výrazu vykoná.

Operácia Shift je implementovaná vo funkcii `opt_switch`. Táto funkcia pridá na zásobník zarážku a aktuálny token, a načíta ďalší token pomocou makra `GET_TOKEN`.

Operácia Reduce, implementovaná vo funkcii `opt_reduce`, používa vlastný pomocný zásobník na ukladanie (pod)výrazu od aktuálneho tokenu až po zarážku v hlavnom zásobníku. Tento výraz je nasledovne spracovaný a zredukovaný na základe nasledujúcich pravidiel:

```
E->i                E->E==E
E->(E)              E->E!=E
E->E+E              E->E>E
E->E-E              E->E<E
E->E*E              E->E>=E
E->E/E              E->E<=E
```

Operácia Equal je implementovaná pomocou funkcie `opt_eq`. Táto operácia sa vykoná len v prípade, že sa vo výraze nachádzajú zátvorky. Funkcia zredukuje všetko medzi zátvorkami a načíta ďalší token pomocou makra `GET_TOKEN`.

Počas spracovania celého výrazu sa zároveň pridávajú inštrukcie matematických operácií pre následné generovanie kódu.

(implementácia v súbore `expression.c`)

2.3 Sémantická analýza

Sémantická analýza je rozdelená na dve časti, časť sémantickej analýzy sa vykonáva počas syntaktickej analýzy a časť po jej skončení. Sémantická analýza rovnako ako syntaktická používa štruktúru `tPData`, v ktorej sa nachádzajú dáta dôležité pre sémantické kontroly. Ak počas syntaktickej analýzy narazíme na identifikátor, ktorý jazyk IFJ2018 nepozná, môže sa jednať o identifikátor premennej alebo názov funkcie. Ak by sa v parametroch funkcie vyskytol identifikátor, analýza musí skontrolovať, či sa jedná o premennú, ktorá už bola definovaná a ak nie, jedná sa o chybu. Ďalej musíme počítať s tým, že v tomto jazyku sa môžu definície funkcií vyskytovať až po ich volaní, takže ak analýza narazí na identifikátor, ktorý sa nezhoduje so žiadnou z už definovaných premenných a funkcií, musí to uložiť ako možné volanie funkcie a po skončení syntaktickej analýzy znovu skontrolovať, či v programe táto funkcia bola definovaná.

Ďalšia sémantická kontrola sa vykonáva za behu programu. Inštrukcie na túto kontrolu sa generujú v generátore troj-aresného kódu. Pri operáciách vo výrazoch musíme kontrolovať či sú obi dva operandy typu int alebo float. Ak je jeden operand typu int a druhý float alebo naopak, musíme ten operand, ktorý má typ int, pretypovať na float. Výnimku tvorí inštrukcia add a inštrukcie na porovnávanie, kde sa môžu vyskytnúť aj operandy typu string, pričom tento typ musia mať oba operandy, inak sa jedná o chybu. Kontrolovať sa musia aj parametre vstavaných funkcií, pričom sa typy parametrov líšia podľa danej funkcie.

(implementácia v súbore `parser.c`)

2.4 Generovanie cieľového kódu

Po skončení sémantickej a syntaktickej analýzy predá parser vnútorný kód prekladača generátoru výsledného kódu (`tIList *ilist`). Generátor sa správa nasledovne. Vygeneruje hlavičku (`generate_head`), ktorá musí byť v každom IFJcode18 výstupnom medzikóde, vygeneruje kód hlavného tela programu (`generate_main`), vygeneruje užívateľsky definované funkcie (`generate_fundef`) a nakoniec na základe požiadavky od užívateľa vygeneruje vstavané funkcie jazyka IFJ2018 (`generate_builtin`), generátor jednoducho prechádza cez jednosmerne viazaný zoznam. Pre všetky inštrukcie je vytvorená rozradzovacia funkcia (`generate_instruction`), v ktorej sa nachádza switch pre každé inštrukčné makro nachádzajúce sa v súbore `code_generator.h`. Z tejto rozdradzovacej funkcie, sa potom zavolá funkcia vytvorená pre každé makro. Pri generovaní zložitých blokov kódu (napr.: `if`, `while`, definícia funkcie), tieto funkcie prechádzajú samostatne tento zoznam inštrukcií (odkiaľ začína tento blok) a aby zabezpečili neduplicitu príkazov, nahrádzajú každú vygenerovanú inštrukciu inštrukčným makrom (`#define NOP 299`). Pri vnorených blokoch v cykle while, funkcia pre cyklus while, vygeneruje všetky vnútorné premenné prekladača pred začiatok cyklu, aby nedošlo ku redefinovaniu existujúcich premenných v interprete, čo by viedlo ku chybe. A tak isto vygeneruje užívatelské premenné, aby sa splnila požiadavka pre projekt, aby premenné definované vnútri blokov platili aj mimo nich. Podobne, pracuje aj blok if. Pri generovaní sme si museli dať pozor na unikátnosť návestí a rôznych vnútorných premenných prekladača, čo sme zabezpečili globálnou premennou `int uniqueCounter`, ktorá obsahuje vždy unikátne číslo pre inštrukciu a po využití sa vždy inkrementuje o 1. Pri spracovávaní výrazov sa vždy používa dátový zásobník interpretu. Vzhľadom na to, že tento jazyk je dynamicky typovaný, považovali sme za dosť náročné, priam až nemožné, robiť typové kontroly v priebehu sémantickej analýzy, preto všetky typové kontroly sú vykonávané počas behu interpretu. Táto taktika na druhú stranu má taktiež nevýhodu, výsledný medzikód je niekoľko násobne dlhší ako kód, ktorého dátové typy boli kontrolované sémantickou analýzou a znižuje efektivitu interpretovaného medzikódu. Po vygenerovaní výsledného medzikódu, generátor uvoľní využívanú pamäť a ukončí prácu.

(implementácia v súbore `code_generator.c`)

3 Algoritmy a dátové štruktúry

3.1 Tabuľka symbolov

Pre náš projekt sme si vybrali variantu projektu s abstraktnou dátovou štruktúrou binárny vyhľadávací strom (ďalej už ako BST), s ktorou sme už boli oboznámený z predchádzajúcich predmetov na VUT FIT. Operácie nad týmto BST sme tvorili rekurzívne, pretože sú jednoduchšie na implementáciu a taktiež sme sa s ich implementáciou stretli pri druhom projekte z predmetu IAL. Presnejšie, v našom projekte je táto štruktúra pomenovaná `tNode`(predstavuje jeden uzol BST) a jej koreň je uložený v štruktúre `tSymTable`. Ďalší dôvod výberu BST bola dobrá asymptotická zložitosť, ktorá pri vyhľadávaní, vkladaní a mazaní naberá priemernú zložitosť O(log n) a v najhoršom prípade O(n).

V našom projekte, sa vyskytujú dve tabuľky symbolov, lokálna a globálna. Lokálna tabuľka symbolov slúži na uchovávanie premenných v danej funkcií alebo mimo funkcie. Globálna tabuľka slúži na uchovávanie funkcií v celom užívateľskom zdrojovom kóde. Presnejšie v súbore `parser.h`, v štruktúre `tPData pData` (dáta parsera) sa nachádzajú ukazovatele na tieto dve tabuľky `tSymTable *local` a `tSymTable *global`.
Obsah jedného uzla stromu sme postupom riešenia projektu upravovali. Jeden uzol sa skladá z `string id` názov symbolu (funkcie/premennej), `int paramsNum` počet parametrov funkcie (pri premennej má vždy hodnotu 0), `bool wasDefined` slúži pre splnenie požiadavku na rekurzívne volanie funkcií navzájom, kde volanie funkcie sa vyskytuje pred definíciou funkcie a `tNode *lptr` a `tNode *rptr,` ktoré slúžia ako ukazovateľ na ľavý a pravý uzol BST.

Nasledujúce operácie boli implementované pre prácu s tabuľkou symbolov, `init_table` nastaví ukazovateľ na koreň BST na `NULL`, `insert_var` vloží jeden symbol (presnejšie premennú) do danej tabuľky symbolov, `insert_fun` vloží jeden symbol (presnejšie funkciu) do tabuľky symbolov, `search_table` vyhľadá v danej tabuľke symbolov určitý symbol vyžiadaný užívateľom podľa názvu (`string`) a vráti jeho ukazovateľ do tabuľky symbolov, pri neúspešnom hľadaní vráti `NULL`, `free_symtable` uvoľní pamäť a danú tabulku symbolov.

(implementácia v súbore `symtable.c`)

3.2 Vnútorný kód prekladača

Aby sme mohli splniť požiadavky zadania a bez problémov generovať výsledný medzikód, rozhodli sme sa ukladať vnútorný kód prekladača. Tento vnútorný kód je ukladaný do abstraktnej dátovej štruktúry jednosmerne viazaný zoznam (ďalej už ako SLL). Tento list, abstraktne predstavuje "zparsovaný" užívateľský IFJ2018 kód. Hlava a aktívny prvok listu sa nachádza v štruktúre `tIList`. Jeden prvok SLL predstavuje štruktúra `tInst`, v tomto prvku sa nachádza kód (makro, int) inštrukcie, ukazovateľ na ďalšiu inštrukciu a ďalší SLL zoznam parametrov (tokenov) menom `tTList`. Tento list parametrov predstavuje parametre jednej inštrukcie. Táto štruktúra obsahuje token ktorý predstavuje parameter a ukazovateľ na ďalší parameter. Nasledujúce operácie boli implementované pre prácu s listom inštrukcií, `init_ilist` inicializuje list inštrukcií pre používanie, init_plist inicializuje list parametrov pre používanie, `insert_instr` slúži na vloženie jedného kódu inštrukcie hneď za aktívny prvok listu inštrukcií a nastavenie aktívneho prvku na danú inštrukciu, `insert_param` slúži na vloženie jedného parametra do aktívnej inštrukcie, `free_ilist` uvoľní pamäť a list inštrukcií spolu s parametrami inštrukcie, `free_plist` uvoľní pamäť a list parametrov jednej inštrukcie (volaná funkciou `free_ilist`).

(implementácia v súbore `code_generator.c`)

3.3 Zásobník symbolov

Pre implementáciu precedenčnej syntaktickej analýzy na spracovanie výrazov, bolo nutné implementovať abstraktnú dátovú štruktúru zásobník, presnejšie zásobník symbolov. Tento zásobník je ale taktiež používaný pri syntaktickej a sémantickej analýze (napr.: kontrolovanie duplicity formálnych parametrov funkcie, ukladanie symbolu ľavej strany

priradenia). Zásobník predstavuje štruktúra `tStack`, ktorá obsahuje aktuálny vrchol zásobníka (symbol, token) a ukazovateľ na ďalšieho člena zásobníku. Pri inicializácií zásobníka je na jeho vrchole pridaný špeciálny symbol ktorý slúži, ako zarážka (presnejšie `#define EMPTY 999`), ktorá symbolizuje prázdny zásobník. Nad zásobníkom boli implementované tieto operácie, `init_stack` inicializuje zásobník pre používanie, `head_stack` vracia najvrchnejší symbol (token) uživateľovi, `clear_stack` vyčistí celý zásobník až po symbol EMPTY, `push_stack` vloží na vrchol zásobníku symbol, `free_stack` uvoľní pamäť a celý zásobník.

(implementácia v súbore `stack.c`)

3.4 Dynamický reťazec

Statické pole typu char bolo nevýhodné pre použitie, a tak pre správny chod prekladača bolo nutné implementovať štruktúru dynamický reťazec (ďalej už ako DS), ktorý bol použitý pre ukladanie mien symbolov. DS predstavuje v našom projekte štruktúru `string`. V tejto štruktúre sa nachádza, `char *str` ukazovateľ na pole znakov, ktoré predstavujú meno, `int length` predstavuje dĺžku tohto DS a `int allocSize` znamená koľko pamäte je alokovanej pre tento DS. Konštanta `#define STR_LEN_INC 128` predstavuje veľkosť o akú sa realokuje DS, ak nastane `length > allocSize` (užívateľ žiada viac miesta pre názov DS). Nasledujúce operácie boli implementované pre prácu s DS, `str_init` inicializuje DS pred prvým použitím, `str_add_char` pridá znak na koniec DS, `str_copy_string` skopíruje obsah jednej DS do druhej DS, `str_copy_const_string` skopíruje konštantné pole znakov do jednej DS, `str_cmp_string` porovná dve zadané DS a vráti užívateľovi výsledok porovnávania, `str_cmp_const_string` porovná obsah DS a konštantného pola znakov a vráti výsledok porovnávania, `str_free` uvoľní pamäť a DS.
(uložený v súbore `str.c`)

4 Práca v tíme

4.1 Spôsob práce a komunikácie v tíme

S vypracovaním projektu sme začali pomerne skoro a snažili sme sa vytvoriť si celkový pohľad na problematiku. Na jednotlivých úlohách sme pracovali jednotlivo alebo vo dvojiciach, pričom po vypracovaní každej časti sme sa s daným riešením oboznámili všetci a konzultovali sme spolu možné vylepšenia. Komunikácia v tíme prebiehala osobne na spoločných stretnutiach, prípadne sme sa kontaktovali online v našej skupinovej konverzácii.

4.2 Verzovací systém

Pre správu súborov sme použili verzovací systém Git. Zdrojové kódy sme mali uložené na vzdialenom repozitári webovej služby GitHub. Vďaka tomu sme mali všetci prístup vždy k najnovšej verzii nášho projektu. Ukladali sme si aj staré verzie vo vedľajších vetvách, čo sa nám vyplatilo v prípade, že sme sa museli k niečomu vrátiť, prípadne použiť niektorú staršiu verziu a pracovať s ňou naďalej ako s hlavnou pre vývoj nášho riešenia.

4.3 Rozdelenie práce

Prácu v tíme sme si rozdelili hneď ako sme si ujasnili celkovú štruktúru projektu. Snažili sme sa rozdeliť úlohy rovnomerne medzi všetkých členov tímu. Na oprave chýb a zdokonaľovaní kódu sme sa podieľali všetci.

Prácu sme si rozdelili nasledovne:
Tomáš Sasák: Syntaktická analýza, sémantická analýza, generátor kódu, tabuľka symbolov, dokumentácia
Tomáš Venkrbec: Lexikálna analýza, syntaktická analýza, testovanie, dokumentácia, kontrola kódu
Martin Krajči: Generátor kódu, syntaktická analýza, dokumentácia
Natália Dižová: Syntaktická analýza, lexikálna analýza, dokumentácia

5 Záver

Všeobecne projekt bol pre nás veľmi zaujímavý a prínosný. Zo začiatku, sme boli vystrašený zadaním, presnejšie jeho celým obsahom a množstvom požiadaviek. Bolo ťažké začať, ale postupom času a vďaka preberanej látke týkajúcej sa projektu, sme si problematiku mohli lepšie predstaviť a navrhnúť riešenie. Taktiež si myslíme, že projekt nám priniesol nielen poznatky z učiva týkajúceho sa predmetov IFJ a IAL, ale i zlepšenie ovládania vzdialeného repozitára a programovacie schopnosti. Na projekte sme začali robiť pomerne skoro a práca v tíme bola taktiež dobrá, pri tvorení projektu sme taktiež písali svoje vlastné testy a zverejňovali si ich na repozitár, po každej zmene sme testy púšťali.

6 Použitá literatúra a referencie

[1] Prezentácie z predmetu Formálne jazyky a prekladače

[2] Prezentácie z predmetu Algoritmy

[3] Článok ku syntax directed translation od portálu GeeksforGeeks

https://www.geeksforgeeks.org/compiler-design-syntax-directed-translation/

LL-Gramatika

1. <start> -> DEF ID OPEN_PARENTH <f_params> CLOSE_PARENTH END_OF_LINE <statement> END END_OF_LINE <start>
2. <start> -> <statement> <fin>
3. <statement> -> IF <expression> THEN END_OF_LINE <statement> ELSE END_OF_ LINE <statement> END END_OF_LINE <statement>
4. <statement> -> WHILE <expression> DO END_OF_LINE <statement> END END_OF_LINE <statement>
5. <statement> -> <expression> END_OF_LINE <statement>
6. <statement> -> <var> ASSIGNMENT <expression> END_OF_LINE <statement>
7. <statement> -> ID_F <params> END_OF_LINE <statement>
8. <statement> -> ID_F OPEN_PARENTH <params> CLOSE_PARENTH END_OF_LINE <statement>
9. <statement> -> ID <params> END_OF_LINE <statement>
10. <statement> -> ID OPEN_PARENTH <params> CLOSE_PARENTH END_OF_LINE <statement>
11. <statement> -> END_OF_LINE <statement>
12. <statement> -> ε
13. <fin> -> END_OF_FILE
14. <f_params> -> <var> <fn_param>
15. <f_params> -> ε
16. <fn_param> -> COMMA <var> <fn_param>
17. <fn_param> -> ε
18. <params> -> <var> <n_param>
19. <params> -> <c_val> <n_param>
20. <n_param> -> COMMA <var> <n_param>
21. <n_param> -> COMMA <c_val> <n_param>
22. <n_param> -> ε
23. <c_val> -> INTEGER
24. <c_val> -> FLOAT
25. <c_val> -> STRING
26. <c_val> -> NIL
27. <var> -> ID

Precedenčná tabuľka matematických výrazov

	+ -	*** /**	**RO**	**CO**	**(**	**)**	**i**	**$**
+ -	>	<	>	>	<	>	<	>
*** /**	>	>	>	>	<	>	<	>
RO	<	<	ERR	ERR	<	>	<	>
CO	<	<	ERR	ERR	<	>	<	>
(	<	<	<	<	<	=	<	ERR
)	>	>	>	>	ERR	>	ERR	>
i	>	>	>	>	ERR	>	ERR	>
$	<	<	<	<	<	ERR	<	OK

Skratka RO značí relačné operátory. CO značí porovnávacie operátory. Znamienko < značí operáciu Shift, znamienko > značí operáciu Reduce a znamienko = značí operáciu Equal. ERR v tabuľke značí syntaktickú chybu s návratovou hodnotou 2. OK v tabuľke znamená, že syntaktická analýza výrazu prebehla v poriadku, bez chýb.

Diagram konečného automatu