

Dokumentace k projektu

Interpret jazyka IFJ15

Tým 044, Varianta b/2/II  
Rozšíření SIMPLE

25. listopadu 2015

Dušan Valecký (vedoucí) xvalec00 xx%

Jakub Vitásek xvitas02 xx%

Juraj Vida xvidaj00 xx%

Jaroslav Vystavěl xvysta02 xx%

Marek Vyroubal xvyrou05 00%

Obsah

1. Úvod
2. Struktura projektu
   1. Lexikální analyzátor
   2. Syntaktický analyzátor
   3. Precedenční analýza výrazů
   4. Interpret
3. Řešení vybraných algoritmů a datových struktur
   1. Boyer-Mooreův algoritmus
   2. Heap sort
   3. Tabulka s rozptýlenými položkami
4. Práce v týmu
   1. Příprava a plánování vývoje
   2. Verzovací systém
   3. Rozdělení práce v týmu a rozdělení bodů
5. Závěr
6. Zdroje
7. Úvod

Tato dokumentace popisuje implementaci interpretu imperativního jazyka IFJ15, který je zjednodušenou podmnožinou jazyka C++11. Struktura dokumentace zpočátku následuje strukturu projektu a rozdělení na jeho spolupracující části. Každá podkapitola popisuje způsob implementace dané problematiky a řešení specifických problémů. Dále dokumentace popisuje způsob řešení algoritmů z pohledu předmětu IAL. V druhé části dokumentace je reflektována práce v týmu, její nedílné součásti, použití podpůrných technologií a popis rozdělení bodů.

Poslední část obsahuje diagram konečného automatu využitého lexikálním analyzátorem {1}, LL gramatiku {2} a precedenční tabulku {3} využitou syntaktickou analýzou.

1. Struktura projektu

2.1 Lexikální analyzátor

Lexikální analýza je provedena implementací konečného automatu, který využívá pomocné stavy (např. při načítání čísla). Tento konečný automat rozděluje zdrojový program na lexémy – tedy identifikátory, čísla, operátory a další. Ty jsou ve výsledku reprezentovány tokeny. Token je přitom reprezentován abstraktní datovou strukturou obsahující jeho typ a řádek, na kterém se ve zdrojovém kódu nachází.

Je-li daný lexém identifikátor, číslo nebo řetězec, lexikální analyzátor vrací i jeho název, respektive hodnotu. Po načtení identifikátoru proběhne kontrola, zda jeho název není totožný s některým z klíčových slov, jelikož pro klíčová slova byly vytvořeny speciální typy tokenů. Komentáře a bílé znaky jsou lexikálním analyzátorem ignorovány. V případě chybné struktury aktuálního lexému program končí s odpovídající návratovou hodnotou.

**2.2 Syntaktický analyzátor**

Pro syntaktickou kontrolu byla zvolena metoda rekurzivního sestupu. Aby byla implementace co nejvíce bezproblémová, bylo podstatné převést prvotní návrh gramatiky na LL gramatiku, jejíž pravidla byla využita k zachycení syntaktických chyb. K jejich identifikaci byla vytvořena enumerace, do níž každá chybová hláška vrací index. Podle něj se poté formátuje výpis chyby a samozřejmě i návratová hodnota celého interpretu.

Sémantická kontrola v rámci rekurzivního sestupe je nicméně implementačně zajímavější, než samotná syntaktická analýza. Pro ukládání informací o proměnných, funkcích a formálních parametrech funkcí byla využita tabulka s rozptýlenými položkami. Proměnné, na které se během rekurzivního průchodu narazí, se ukládají primárně do globální tabulky symbolů vyhrazené pro proměnné. Tato tabulka obsahuje několik důležitých položek, které jsou podstatné pro sémantickou kontrolu. Je nimi zejména varType, do níž se ukládá načtený typ, a scope, jež obsahuje momentální rámec definice/deklarace proměnné.

Rámce jsou další velice podstatnou součástí sémantické analýzy. Pro každý blok, který definuje pravidlo COMM\_SEQ, je inkrementována celočíselná globální proměnná. Na konci tohoto pravidla je opět dekrementována, obsahuje tudíž vždy aktuální rámec. Při kontrole redefinice/redeklarace proměnné pak lze tohoto rámce využít. Pro zjednodušení vyhledávání položek v tabulce symbolů podle rámce byla vytvořena funkce htReadScope s parametrem navíc, specifikujícím hledaný rámec. Rovněž při každém dalším rámci je aktuální tabulka symbolů přidána na vrchol zásobníku tabulek symbolů, přičemž po průchodu rámcem je opět ze zásobníku odstraněna.

Poslední důležitou částí sémantické kontroly je kontrola, zda při volání definované funkce byl vložen správný počet parametrů a správný datový typ pro každý z nich. Kontroly počtu parametrů bylo docíleno pomocí využití synonym v tabulce symbolů, kdy pro každý parametr funkce byla vytvořena v tabulce parametrů položka s klíčem (neboli pomyslnou maskou), s tím, že každá položka obsahuje typ parametru a jeho pořadí. Poté už při volání funkce stačí jen každý vložený parametr porovnat záznam v tabulce parametrů, jenž obsahuje pořadí souhlasící s momentálním pořadím vložených parametrů. Opět byla pro zjednodušení vyhledávání podle pořadí implementována nová vyhledávací funkce htReadOrder, jež navíc přijímá vyhledávané pořadí položky.

**2.3 Precedenční analýza výrazů**

Precedenční syntaktická analýza (dále pouze PSA) slouží k vyhodnocování výrazů. Řízení PSA rekurzivní sestup předává v případě, že narazí na pravidlo EXPR. Hlavním algoritmem PSA je cyklus do while, v němž jsou postupně vyhodnocována a redukována pravidla získaná z precedenční tabulky. PSA po každé redukci vygeneruje patřičné instrukce a uloží je na instrukční pásku (s tím, že RS předá zpět informaci o tom, že vše proběhlo v pořádku), nebo vrátí RS patřičnou chybu.

Ke své práci potřebuje precedenční tabulku, která obsahuje pravidla pro výrazy a jejich redukci. Dále je nutná implementace zásobníku a přístup k tabulkám symbolů vytvořených v rámci RS. Předání tabulek je pro jednoduchost řešeno globální deklarací v hlavičkovém souboru parseru. Rovněž je nutný přístup k instrukční pásce, na kterou jsou zapisovány vygenerované instrukce. Ta je poté předána interpretu k provedení.

**2.4 Interpret**

...

1. Řešení vybraných algoritmů a datových struktur

**3.1 Boyer-Mooreův algoritmus**

Boyer-Mooreův algoritmus (dále BMA) byl implementován pro využití v interpretaci volání vestavěné funkce find. BMA nabízí několik typů heuristiky, díky nimž je možné najít shodu mezi řetězcem a podřetězcem. Dle vybrané heuristiky je pak algoritmus například optimálnější, jelikož dokáže specifické znaky ignorovat, a díky tomu prohledat řetězec rychleji. Naše implementace BMA vychází z heuristiky v opoře předmětu IAL (zdroj [2]).

**3.2 Heap Sort**

Řazení hromadou využívá hromady jako struktury stromového typu, přesněji binárního stromu. Otcovský uzel musí mít vždy větší hodnotu než uzly synovské, a pokud je toto pravdilo porušeno, je nutné provést takzvanou rekonstrukci hromady. Naše implementace využívá funkce siftDown, která postupnými výměnami dostane do kořene prvek splňující zmiňované pravidlo hromady.

Tedy v případě, že má hromada v kořeni prvek s nejnižší hodnotou, postupně se odebírá prvek z kořene a ukládá se do výstupního pole. Po každém odebrání je nutné do kořene vložit hodnotu nejnižšího a nejpravějšího uzlu a posléze opět zavolat funkci siftDown.

**3.3 Tabulka s rozptýlenými položkami**

Tabulka symbolů je implementována formou tabulky s rozptýlenými položkami, podporující explicitní zřetězení synonym. Tato tabulka má podobu pole, v němž jsou uloženy ukazatele na záznamy synonym.

Každému symbolu je vygenerován klíč pomocí mapovací funkce, podle kterého je následně možné symbol vyhledat. Pro vkládání prvků do tabulky byla implementována funkce htInsert a htInsertData, přičemž první z nich obsah přepíše a druhá vytvoří nové synonymum.

1. Práce v týmu

4.1 Příprava a plán vývoje

Zezačátku vývoje byla každému členovi týmu určena jeho role a přidělen úkol. Při rozhodování, komu jakou část přidělit, bylo vzato v potaz základní rozložení složitosti a také schopnosti daného jedince. Časové rámce pro každou část byly stanoveny pouze velice abstraktně a spoléhalo se spíše na svědomitý přístup každého člena týmu.

**4.2 Verzovací systém**

Pro jednoduchou a příjemnou kontrolu verzí a možnost rollbacku v případě nalezení chyby byla využita technologie Git s frontendem na stránce Bitbucket, jelikož oproti Githubu nabízí nezpoplatněný privátní repozitář. Během vývoje přišla vhod možnost vytvoření větve (branch), aby mohl probíhat souvislý vývoj s různým kódem, a poté spojení těchto dvou částí do jedné. Pro ověřování důvodu výskytu chyb byl hojně zadáván i příkaz checkout, který dočasně načte soubory z některého ze starších commitů. To nám dalo možnost se takřka "vrátit zpět v čase" a zkompilovat projekt se starším kódem, nad kterým jsme pak spustili testovací sadu a zjistili, co dříve fungovalo. Další a poslední výhodou je i prokazatelnost odvedené práce, díky níž můžeme jednoznačně dokázat, kdo odvedl jakou práci, případně i její množství a významnost.

**4.3 Rozdělení práce v týmu a rozdělení bodů**

Implementaci projektu započal kolega Vida, a to naprogramováním lexikálního analyzátoru. Dále se podílel na vytvoření prvotní kostry projektu a gramatiky, návrhu instrukční sady, generování instrukcí a také rozsáhlou pomocí při debuggingu.

Kolega Valecký přispěl tvorbou gramatiky, LL tabulky a pomocí se začátkem rekurzivního sestupu, jeho hlavní doménou však byla PSA. Dále byl jmenován vedoucím týmu, takže řešil i většinu organizačních záležitostí.

Jakub Vitásek dostal na starost rekurzivní sestup (dále pouze RS) a potažmo veškerou syntaktickou analýzu v rámci RS, celou sémantickou analýzu s využitím tabulky symbolů, programování vestavěných funkcí jazyka IFJ15 a dalé testovací skripty, založení a zaučení do verzovacího systému, pomoc s řízením týmu a obecný debugging.

Zanedbání časného započetí práce se stalo osudným kolegovi Vyroubalovi, který za celou dobu implementace poskytl jen zanedbatelné množství využitelného kódu a na vývoji či konzultacích se takřka vůbec nepodílel. Tím se celý tým dostal do časového skluzu. Proto jsme byli nuceni změnit rozložení bodů.

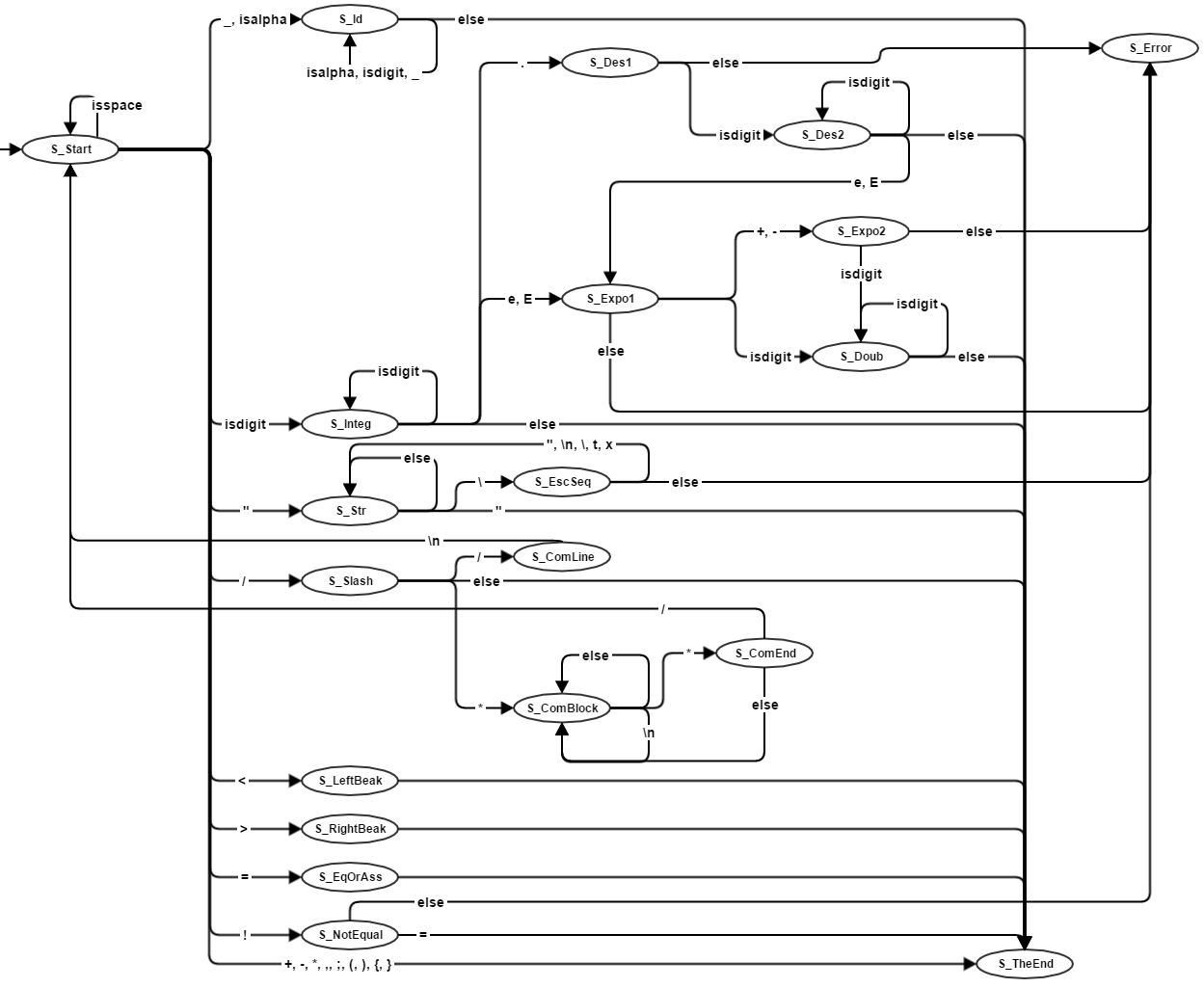
1. Závěr

Tento projekt byl jednoznačně výzvou pro celý tým, jelikož dosahuje v rámci studia zatím nevídané komplexnosti a rozsahu. Transformace teoretických znalostí z přednášek do samotné implementace je velkou praktickou zkušeností, tou však zároveň je i spolupráce v týmu, komunikace a koordinace.

1. Zdroje

[1] Opora IFJ, [2] Opora IAL

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **+** | **-** | **\*** | **/** | **==** | **!=** | **<=** | **>=** | **>** | **<** | **string** | **double** | **int** | **id** | **f** | **(** | **)** | **,** | **$** |
| **+** | > | > | < | < | > | > | > | > | > | > |  | < | < | < | < | < | > | > | > |
| **-** | > | > | < | < | > | > | > | > | > | > |  | < | < | < | < | < | > | > | > |
| **\*** | > | > | > | > | > | > | > | > | > | > |  | < | < | < | < | < | > | > | > |
| **/** | > | > | > | > | > | > | > | > | > | > |  | < | < | < | < | < | > | > | > |
| **==** | < | < | < | < | > | > | > | > | > | > | < | < | < | < | < | < | > | > | > |
| **!=** | < | < | < | < | > | > | > | > | > | > | < | < | < | < | < | < | > | > | > |
| **<=** | < | < | < | < | > | > | > | > | > | > | < | < | < | < | < | < | > | > | > |
| **>=** | < | < | < | < | > | > | > | > | > | > | < | < | < | < | < | < | > | > | > |
| **>** | < | < | < | < | > | > | > | > | > | > | < | < | < | < | < | < | > | > | > |
| **<** | < | < | < | < | > | > | > | > | > | > | < | < | < | < | < | < | > | > | > |
| **string** |  |  |  |  | > | > | > | > | > | > |  |  |  |  |  |  | > | > | > |
| **double** | > | > | > | > | > | > | > | > | > | > |  |  |  |  |  |  | > | > | > |
| **int** | > | > | > | > | > | > | > | > | > | > |  |  |  |  |  |  | > | > | > |
| **id** | > | > | > | > | > | > | > | > | > | > |  |  |  |  |  |  | > | > | > |
| **f** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | = |  |  |  |
| **(** | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | = | = |  |
| **)** | > | > | > | > | > | > | > | > | > | > |  |  |  |  |  |  | > | > | > |
| **,** | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | = | = |  |
| **$** | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < |  |  |

{1} Konečny automat

{2} LL gramatika

1. <PROGRAM> -> <FUNC\_N>
2. <FUNC\_N> -> <FUNC> <FUNC\_N>
3. <FUNC\_N> -> E
4. <VAR\_DEF> -> <TYPE> id <INIT>;
5. <VAR\_DEF> -> auto id <INIT>;
6. <INIT> -> = <EXPR>
7. <INIT> -> E
8. <TYPE> -> int
9. <TYPE> -> double
10. <TYPE> -> string
11. <FUNC> -> <TYPE> id <PAR\_DEF\_LIST> <DEC\_OR\_DEF>
12. <DEC\_OR\_DEF> -> <COMM\_SEQ>
13. <DEC\_OR\_DEF> -> ;
14. <PAR\_DEF\_LIST> -> ( <PARAMS> )
15. <PARAMS> -> <TYPE> id <PARAMS\_N>
16. <PARAMS> -> E
17. <PARAMS\_N> -> , <TYPE> id <PARAMS\_N>
18. <PARAMS\_N> -> E
19. <COMM\_SEQ> -> { <STMT\_LIST> }
20. <STMT\_LIST> -> <STMT> <STMT\_LIST>
21. <STMT\_LIST> -> E
22. <STMT> -> if( <EXPR> ) <COMM\_SEQ> <IF\_N>
23. <STMT> -> for( <VAR\_DEF> <EXPR>; <ASSIGN> ) <COMM\_SEQ>
24. <STMT> -> <COMM\_SEQ>
25. <STMT> -> <VAR\_DEF>
26. <STMT> -> cin >> id <CIN\_ID\_N>;
27. <STMT> -> cout << <COUT\_TERM>;
28. <STMT> -> <RETURN>
29. <STMT> -> id <CALL\_ASSIGN>
30. <CALL\_ASSIGN> -> = <EXPR>;
31. <CALL\_ASSIGN> -> (<terms>);
32. <TERMS> -> id <TERMS\_N>
33. <TERMS> -> E
34. <TERMS\_N> -> , id <TERMS\_N>
35. <TERMS\_N> -> E
36. <ASSIGN> -> id = <EXPR>
37. <CIN\_ID\_N> -> >> id <CIN\_ID\_N>
38. <CIN\_ID\_N> -> E
39. <COUT\_TERM> -> id <COUT\_TERM\_N>
40. <COUT\_TERM\_N> -> << <COUT\_TERM>
41. <COUT\_TERM\_N> -> E
42. <RETURN> -> return <EXPR>;
43. <IF\_N> -> else <COMM\_SEQ>
44. <IF\_N> -> E