Vysoké učení technické v Brně

Fakulta informačních technologií

Dokumentace k projektu do předmětů IFJ a IAL
Implementace interpretu imperativního jazyka IFJ15

Tým 050, varianta a/1/II

Vedoucí:

Miroslav Karásek (xkaras31), 25%

Spoluautoři:

Libor Janíček (xjanic21), 25%
Jakub Kolb (xkolbj00), 25%
Karolína Klepáčková (xklepa04), 25%
Tomáš Kobylák (xkobyl01), 0%

2. prosince 2015

Obsah

1. Úvod

Tato dokumentace popisuje vývoj a implementaci interpretu pro imperativní jazyk IFJ15. Dokumentace je členěna na kapitoly a přílohy, které přibližují způsob implementace jednotlivých částí interpretu i jeho kompletaci. Rozebereme zde podstatné části našeho interpretu a s nimi také specifické metody a algoritmy.
Přílohy obsahují diagram konečného automatu, který je specifický pro lexikální analyzátor, dále LL-gramatiku a precedenční tabulku, které jsou jádrem našeho syntaktického analyzátoru.
V závěru je zhodnocení celého projektu s ohledem na odvedenou práci, kvalitu implementace a přínos celému týmu.

2. Rozdělení práce mezi členy týmu

Práci na interpretu jsme si rozdělili rovnoměrně mezi všechny členy týmu s výjimkou Tomáše Kobyláka, který se z důvodu dlouhodobé nemoci na projektu nepodílel. Rozdělení bylo následovné:
Miroslav Karásek - parser
Karolína Klepáčková - scanner, instrukční páska
Libor Janíček - interpret
Jakub Kolb - Quick sort, Knuth-Morris-Prattův algoritmus

3. Vývojový cyklus

3.1. Komunikace týmu

Komunikace v týmu probíhala z velké části pomocí společného chatu, popřípadě osobně ve škole nebo při týmových schůzkách. Scházeli jsme se jedenkrát v rozmezí jednoho až dvou týdnů v době, kdy měli všichni členové týmu čas.

3.2. Správa kódu

Pro správu kódu jsme využili verzovací program git. Aby měli všichni členové týmu ke kódu neustále přístup, použili jsme pro uložení repozitáře hosting http://bitbucket.com. Ten jsme vybrali proto, že umožňoval zdarma soukromý repozitář, tím pádem se k němu mohli dostat jen členové našeho týmu.

4. Návrh a implementace interpretu

4.1. Scanner

Scanner je celkově implementován jako konečný automat s 23 stavy bez epsilon přechodů. Princip jeho funkčnosti spočívá v načítání jednotlivých znaků ze zdrojového kódu do jednotlivých tokenů, do té doby než přijde nějaký nepřípustný znak. Tokeny jsou zde reprezentovány jako abstraktní datový typ, který obsahuje informaci o typu tokenu, jeho hodnotě, řádku a sloupci kde se v textu nachází. Hodnota tokenu je přiřazována pouze v případě řetězců, čísel, identifikátorů, rezervovaných slov a klíčových slov, v ostatních případech zůstává prázdná.
Obsahuje také kontrolu jednotlivých načtených identifikátorů, zda se jedná o klíčová nebo rezervovaná slova. Pokud tomu tak je tak se tokenu přidává konkrétní typ, který určuje, o jaké slovo se jedná. Tato kontrola probíhá porovnáním právě načteného identifikátoru s tabulkou rezervovaných a klíčových slov, která je implementována jako pole řetězců.

4.2. Parser

V našem případě parser obsahuje syntaktickou analýzu, sémantickou analýzu a zároveň generování kódu. Syntaxe je řízena LL-gramatikou, která je uvedena níže. Implementována je pomocí rekurzivního sestupu. Pro zpracování výrazů je použita precedenční syntaktická analýza. Ta je řízena tabulkou, která je rovněž uvedena níže. Dále provádí parser sémantické kontroly jako je kontrola datových typů a podobně. Pokud jsou všechny kontroly úspěšné, je vygenerován tříadresný kód. Parser vkládá symboly (názvy funkcí a proměnných) do tabulky symbolů. Vytváří se jedna globální tabulka symbolů pro uložení identifikátorů funkcí a dále se vytváří samostatná tabulka pro každý blok programu. Tabulky musejí být provázány tak, aby bylo možné vyhledat i symboly z nadřazeného jmenného prostoru.

4.3. Interpret

Vstupem interpretu je blok funkce main. Blok je datová struktura reprezentující ucelenou část programu. Obsahuje instrukční pásku a tabulku symbolů daného celku. Tuto ucelenou část zdrojového kódu rozpozná podle dvojic složených závorek. Díky tomu nejsou potřeba žádná navěští, neboť pokud je podmínka splněna, tak se provede první blok, jinak se provede blok druhý. Podobně funguje cyklus for. Zde se jeden blok opakuje, dokud platí podmínka. Tyto bloky jsou již vyplněny od parseru a interpret pouze provede jejich kód. Protože je umožněno rekurzivní volání funkcí, musí interpret zavádět tzv. rámce.

Rámec je pole pro uložení hodnot proměnných. Jeden rámec náleží právě jediné instanci dané funkce i všem jejím podblokům. Tabulky symbolů obsahují jen index do tohoto pole. Díky tomu může existovat více instancí funkcí. Při opětovném volání se jen alokuje nový rámec.

5. Řešení algoritmů pro IAL

5.1. Knuth-Morris-Prattův algoritmus

Tento algoritmus zrychluje hledání podřetězce v řetězci. Má lineární časovou složitost $O(m+n)$ a je význačný tím, že si zachovává minimální počet porovnání. To znamená, že neporovnává stejné znaky v případě, že už byly porovnány a není možné, aby byly součástí podřetězce, který je prefixem hledaného řetězce.

5.2. Quick sort

Na tento projekt jsme použili třídící algoritmus Quick Sort. Jak už nám název napoví, jedná se o rychlý řadící algoritmus, který funguje na principu "Divide et Impera" (rozděl a panuj). Má asymptotickou složitost $O(n^2)$.

Princip třídění spočívá v tom, že si zvolíme libovolný prvek v poli (pivot). Zvolit ho můžeme náhodně, matematickým výpočtem nebo pevně. My jsme si pivot vybrali pevně. Následně můžeme pole přeházet tak, aby na jedné straně byly větší prvky než pivot, na druhé straně menší a pivot byl umístěn mezi těmito částmi. Tento postup opakujeme pro obě rozdělené části, kde si zase zvolíme pivot. Opakujeme do té doby, dokud nejsou všechny prvky roztřízeny.

5.3. Tabulka s rozptýlenými položkami

Tabulka s rozptýlenými položkami se využívá k uložení symbolů. Umožňuje rychlé vyhledávání. Tabulka funguje tak, že se klíč pomocí hashovací funkce transformuje na číselnou hodnot, která se využije jako index v poli. Protože může docházet ke kolizím, nejsou data uložena přímo v poli, ale je využito tzv. explicitní zřetězení. To znamená, že položka v poli je začátek jednosměrně vázaného seznamu. Vyhledávání tedy probíhá tak, že nejprve určíme index do pole a poté prohledáváme lineární seznam již za použití klasického porovnávání řetězců.

6. Testování

Testování jsme považovali za velmi důležitou část celého procesu, a proto jsme jí věnovali náležitou pozornost. Každý člen týmu měl svůj vlastní test, který uplatňoval při kontrole správnosti jeho části. Tento přístup se nám vyplatil, neboť jsme při kompletaci celého interpretu nemuseli hledat chyby chodu jednotlivých částí. Jediné, co jsme testovali po kompletaci, byly vstupy a výstupy jednotlivých částí, které se zdály nesprávné. Testování celku dělal převážně vedoucí a poté se odkazoval na konkrétního člena, který nesprávně pracující část implementoval.

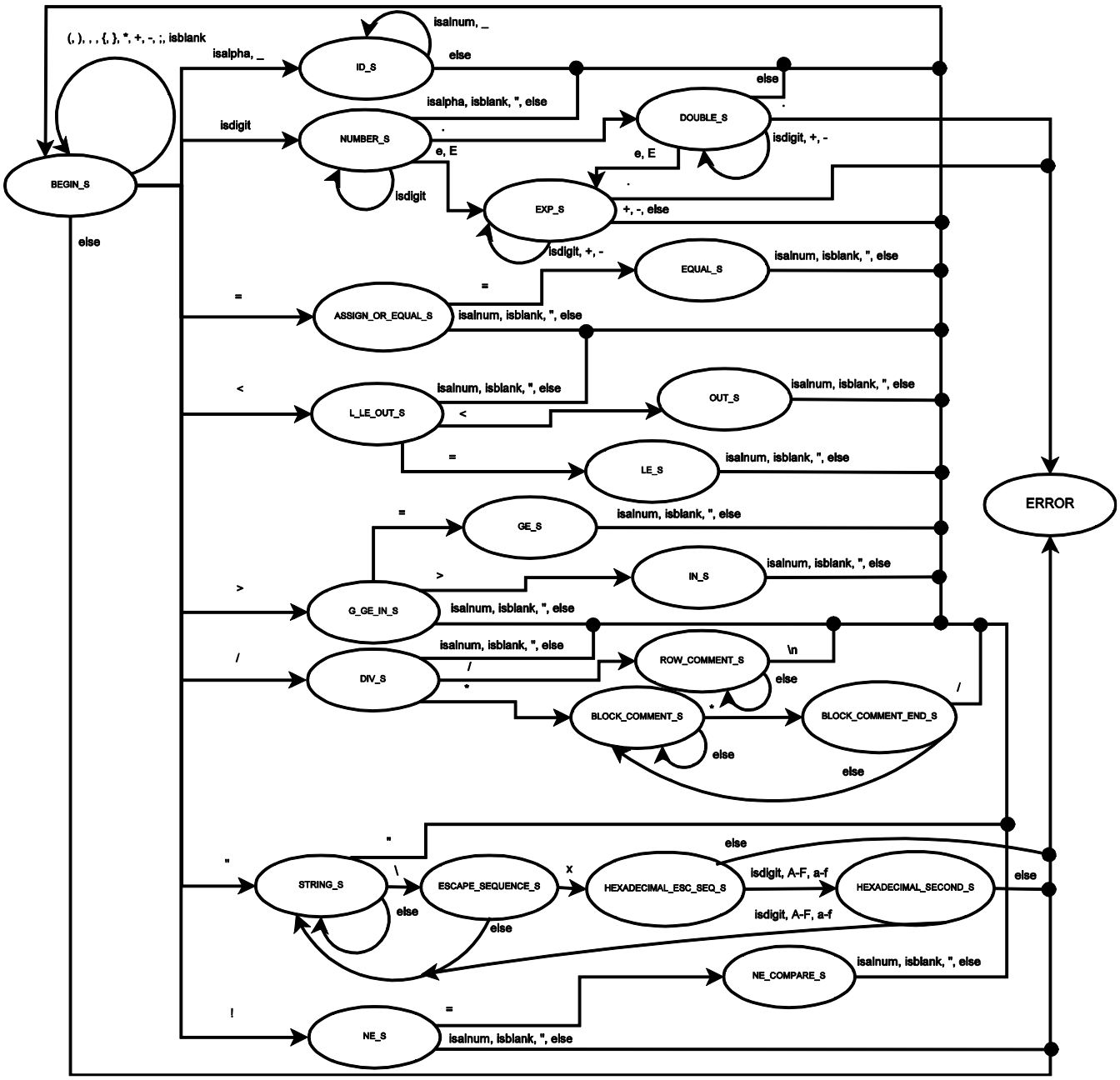
7. Závěr

Implementace interpretu imperativního jazyka IFJ15 nám všem přišel zprvu jako nepředstavitelný projekt a jako značná výzva. Několik týdnů jsme byli docela zmatení a nevěděli jsme kde začít a jak postupovat dále. Avšak samostudium, poslouchání přednášek a studium opory i slidů, převážně z předmětu IFJ, nám poskytlo dostatečnou představu o tomto problému a pustili jsme se do jeho realizace. Během vývoje interpretu jsme narazili na několik úskalí, které se ale časem vyřešily buď formou dotazů na fóru, nebo osobními emaily s cvičícími.
Projekt nám celkově poskytl spoustu nových technických vědomostí a aspektů řešení komplexních problémů při práci v týmu. Za celý semestr jsme nenarazili na žádné problémy s týmovou prací či komunikací jednotlivců v týmu.
Projekt hodnotíme jako přínos pro budoucí práci ve větších týmech i při řešení větších problémů. Celkově nám to pomohlo při představě o komplexnosti překladačů obecně a o jejich činnosti.

8. Literatura a reference na čerpané zdroje

Slidy a studijní opora předmětu IFJ
Studijní opora předmětu IAL

I. Diagram konečného automatu



II. LL-gramatika

```
PROGRAM -> FUNCTION_LIST
FUNCTION_LIST -> FUNCTION NEXT_FUNCTION_LIST
NEXT_FUNCTION_LIST -> ε
NEXT_FUNCTION_LIST -> FUNCTION NEXT_FUNCTION_LIST
FUNCTION -> type fid ( PARAMETER_LIST ) FBLOCK
PARAMETER_LIST -> ε
PARAMETER_LIST -> PARAMETER NEXT_PARAMETER_LIST
PARAMETER -> type id
NEXT_PARAMETER_LIST -> ε
NEXT_PARAMETER_LIST -> , PARAMETER NEXT_PARAMETER
FBLOCK -> ;
FBLOCK -> BLOCK
BLOCK -> { STAT_LIST }
STAT_LIST -> ε
STAT_LIST -> STAT STAT_LIST
STAT -> BLOCK
STAT -> id = RVAL ;
STAT -> if ( EXPRESSION ) BLOCK else BLOCK
STAT -> for ( DECLARATION ; EXPRESSION ; id = EXPRESSION ) BLOCK
STAT -> return EXPRESSION ;
STAT -> cin >> IN_LIST ;
STAT -> cout << OUT_LIST ;
STAT -> DECLARATION ;
DECLARATION -> type id DEFINITION
DEFINITION -> ε
DEFINITION -> = EXPRESSION
OUT_LIST -> CALL_PARAMETER NEXT_OUT_LIST
NEXT_OUT_LIST -> ε
NEXT_OUT_LIST -> << CALL_PARAMETER NEXT_OUT_LIST
IN_LIST -> id NEXT_IN_LIST
NEXT_IN_LIST -> ε
NEXT_IN_LIST -> >> id NEXT_IN_LIST
RVAL -> EXPRESSION
RVAL -> fid ( CALL_PARAMETER_LIST )
CALL_PARAMETER_LIST -> ε
CALL_PARAMETER_LIST -> CALL_PARAMETER NEXT_CALL_PARAMETER_LIST
NEXT_CALL_PARAMETER_LIST -> ε
NEXT_CALL_PARAMETER_LIST -> , CALL_PARAMETER NEXT_CALL_PARAMETER_LIST
CALL_PARAMETER -> id
CALL_PARAMETER -> int
CALL_PARAMETER -> double
CALL_PARAMETER -> string
```

III. Precedenční tabulka

	id	int	double	string	+	-	*	/	>	<	>=	<=	==	!=	(	)	;
id	E	E	E	E	>	>	>	>	>	>	>	>	>	>	E	>	>
int	E	E	E	E	>	>	>	>	>	>	>	>	>	>	E	>	>
double	E	E	E	E	>	>	>	>	>	>	>	>	>	>	E	>	>
string	E	E	E	E	>	>	>	>	>	>	>	>	>	>	E	>	>
+	<	<	<	<	>	>	<	<	>	>	>	>	>	>	<	>	>
-	<	<	<	<	>	>	<	<	>	>	>	>	>	>	<	>	>
*****	<	<	<	<	>	>	>	>	>	>	>	>	>	>	<	>	>
/	<	<	<	<	>	>	>	>	>	>	>	>	>	>	<	>	>
>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	>
<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	>
>=	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	>
<=	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	>
==	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	>
!=	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	E
)	E	E	E	E	>	>	>	>	>	>	>	>	>	>	E	>	>
END	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	E/S	S