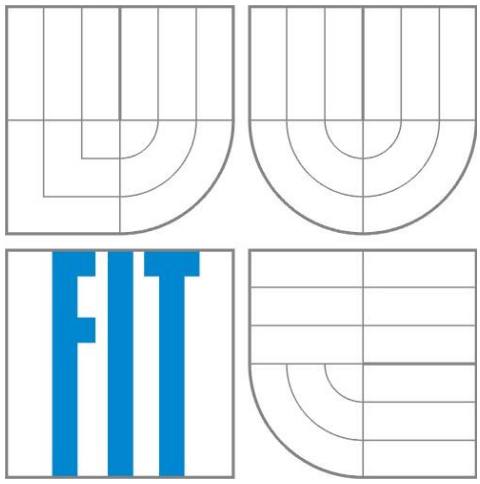
Vysoké učení technické v Brně
Fakulta Informačních technologií



Dokumentace k projektu pro předměty IFJ a IAL

Implementace interpretu imperativního jazyka IFJ11

Tým 48, varianta b/4/II

11. prosince 2011

Rozšíření:
1. Operátor modulo

Autoři:	Jakub Kvita (vedoucí),	xkvita01@stud.fit.vutbr.cz,	20%
	Ondřej Cienciala,	xcienc02@stud.fit.vutbr.cz,	20%
	Robert Pecsérke,	xpecse00@stud.fit.vutbr.cz,	20%
	Martin Krippel,	xkripp00@stud.fit.vutbr.cz,	20%
	Petr Pyszko,	xpyszk02@stud.fit.vutbr.cz,	20%

OBSAH

Úvod

Tento dokument popisuje implementaci interpretu imperativního jazyka IFJ11, který je podmnožinou jazyka LUA. Projekt jsme řešili v týmu pěti lidí. Při přihlašování jsme si vybrali specifickou variantu b/4/II, tzn. tabulka symbolů je implementována pomocí **hashovací tabulky**, vyhledávání podřetězce **Boyer-Mooreovým** algoritmem a řazení **Merge-sort** algoritmem.

1 Práce v týmu

1.1 Vytvoření týmu

Tento projekt je poměrně rozsáhlý, proto bylo důležité zvolit správnou taktiku při týmové práci. První schůzka se uskutečnila již na začátku semestru a zvolili jsme si na ní vedoucího týmu, který bude zajišťovat rozdělování práce a celkový dohled nad stavem projektu. Dále jsme se také dohodli na komunikačních kanálech.

Pro správu zdrojových souborů jsme si zvolili systém **SVN** a pro sdílení dokumentů online aplikaci **Google documents**. Vzdálenou komunikaci v rámci celého týmu jsme realizovali pomocí soukromé skupiny na sociální síti **Facebook**. Vzájemná, specifičtější komunikace pak probíhala pomocí klientů **ICQ** a podobně. Osobní komunikace probíhala pravidelnými schůzkami celého týmu, na nichž jsme řešili specifické problémy a navrhovali rozhraní i řešení celého interpretu.

1.2 Rozdělení práce

Postupem času jsme díky přednáškám získali představu, co se od nás očekává, a začali jsme si rozdělovat práci. Nakonec jsme se dohodli na tomto rozdělení práce. Každou část zpracoval jeden člen týmu.

1. parser pro výrazy, správa SVN a komunikačních kanálů, dohled na vývoj (vedoucí);
2. lexikální analyzátor, vytvoření kostry projektu (moduly, hlavičkové soubory, makefile);
3. syntaktická a sémantická analýza;
4. vestavěné funkce, tabulka symbolů, vytvoření kostry dokumentace;
5. interpret.

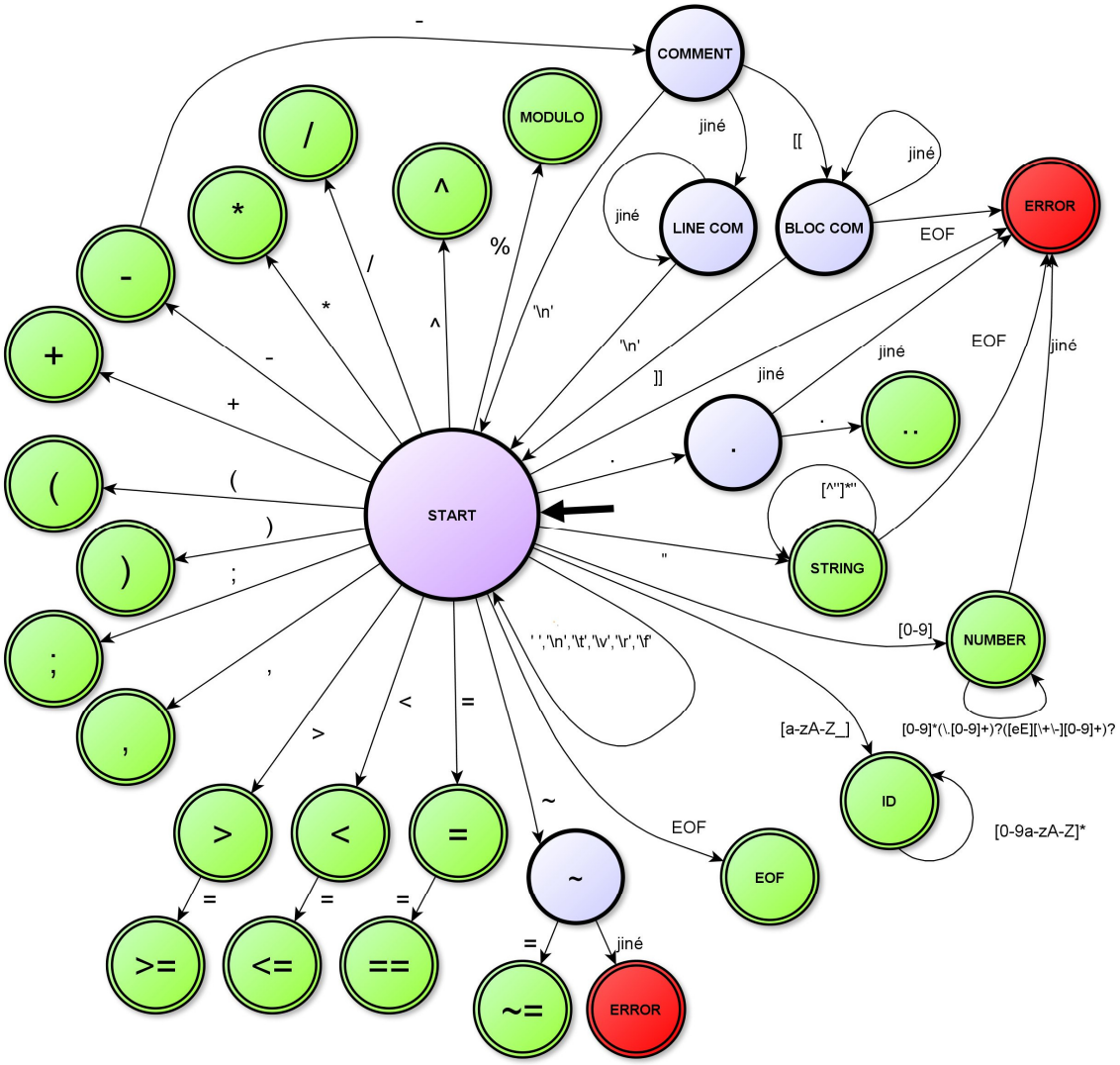
2 Návrh a implementace

2.1 Lexikální analýza

Lexikální analýza je reprezentována funkcí **get_token(),** která vrací **token**. **Token** obsahuje kromě číselné identifikace typu lexému také atribut, což je v paměti dynamicky alokovaný řetězec. V něm jsou uložena data, které je nutné uchovat pro pozdější zpracování jako např. jméno identifikátoru, obsah řetězcového literálu apod. Paměť atributu je alokována postupně podle potřeby, což umožňuje pracovat s nekonečně dlouhými řetězci (omezeno pouze velikostí operační paměti). Poslední složkou tokenu je číslo řádku, na kterém se lexém objevil.

Funkce **get_token()** je vnitřně implementovaná jako stavový automat (viz diagram níže). Pokud v koncovém stavu přijde nepovolený znak, funkce daný znak vrátí zpět jako nepřečtený a vrátí patřičný **token**. Pokud však s nepovoleným znakem není v koncovém stavu, je nastavena globální chybová proměnná **err_flg**, kterou musí testovat funkce volající **get_token()**. Tato proměnná je taktéž nastavena v případě různých vnitřních chyb (nedostatek operační paměti atd.).

Pro čtení znaků ze zdrojového souboru a pro jejich případné navrácení se používají funkce **next_char()** a **back_char()**, které počítají aktuální řádek ve zdrojovém souboru a zaobalují funkce **getc()** a **ungetc()**.



Obrázek č. 1 Diagram stavového automatu scanneru

2.2 Syntaktická analýza

Funkčním jádrem našeho interpretu je syntaktický analyzátor. Jeho činnost je spojená s činností sémantického analyzátoru. Pokud zdrojový program splňuje pravidla jazyka IFJ11, pak vygeneruje příslušné instrukce pro interpretační část projektu, jinak ukončí svoji činnost a vrátí příslušný chybový kód.

Syntaktická analýza je rozdělena do dvou částí – výrazy jsou zpracovávány precedenční analýzou a zbytek programu rekurzivním sestupem.

2.2.1 Syntaktická analýza rekurzivním sestupem

Je založena na LL gramatice sestavené z popisu jazyka IFJ11.

Pravidla LL gramatiky:

Legenda:
<neterminál>
terminál

1: <literal> → **number**
2: <literal> → **string**
3: <literal> → **bool**
4: <literal> → **nil**

5: <prog> → <func-def> <func-def-seq> **eof**

6: <func-def-seq> → <func-def> <func-def-seq>
7: <func-def-seq> → **;**

8: <func-def> → **function id (** <params> <var-dec-seq> <stat-seq>

9: <var-dec-seq> → <variable-declaration> <var-dec-seq>
10: <var-dec-seq> → epsilon

11: <variable-declaration> → **local id** <ending>

12: <ending> → **;**
13: <ending> → **= <**literal> **;**

14: <stat-seq> → <statement> <stat-seq>
15: <stat-seq> → **end**

16: <alt-stat-seq> → <statement> <alt-stat-seq>
17: <alt-stat-seq> → **else**

18: <statement> → **id =** <right-value> **;**
19: <statement> → **write (** <expresion-seq> **;**
20: <statement> → **if** <expresion> **then** <alt-stat-seq stat-seq> **;**
21: <statement> → **while** <expresion> **do** <stat-seq> **;**
22: <statement> → **return** <expresion> **;**

23: <right-value> → **read (** <literal> **)**
24: <right-value> → <expresion>
25: <right-value> → **id (** <input-param-seq>

26: <expresion-seq> → <expresion> <next-expresion>
27: <expresion-seq> → **)**

28: <next-expresion → **,** <expresion next-expresion>
39: <next-expresion> → **)**

30: <params> → **id** <next-param>
31: <params> → **)**

32: <next-param> → **, id** <next-param>
33: <next-param> → **)**

34: <input-param-seq> → <input-param> <next-input-param>
35: <input-param-seq> → **)**

36: <input-param> → <literal>
37: <input-param> → **id**

38: <next-input-param> → **,** <input-param> <next-input-param>
39: <next-input-param> → **)**

2.2.2 Analýza výrazů

Založena na precedenční SA.

U konstrukce Precedenční tabulky jsme se odchýlili od návrhu, který byl prezentován na přednáškách v implementaci pátého dodatečného symbolu, označujícího konec výrazu. Panem Medunou označovaný jako „happy face“ jsme umístili na klasické místo – průsečík „**$**“ / „**$**“ a následně také na průsečík „**$**“ / „**)**“ , protože v určitých případech se stalo, že celý výraz byl umístěn v závorkách, které ale nepatřily do výrazu. Rekurzivní SA vzala „(“ jako součást pravidla a očekávala výraz, za kterým je „)“. To způsobovalo konflikty mezi jednotlivými částmi SA a tímto jsme to vyřešili.

Další implementace analýzy, používá zásobník s velice specifickými operacemi. Ten jsme také implementovali, včetně těchto operací, které později ušetřily mnoho času a práce.

Pravidla:

```
1:    <E>  → <E>   ^   <E>
2:    <E>  → <E>   *   <E>
```

```
 3:    <E> → <E>  /   <E>
 4:    <E> → <E>  %   <E>
 5:    <E> → <E>  +   <E>
 6:    <E> → <E>  -   <E>
 7:    <E> → <E>  ..  <E>
 8:    <E> → <E>  <   <E>
 9:    <E> → <E>  <=  <E>
10:    <E> → <E>  >   <E>
11:    <E> → <E>  >=  <E>
12:    <E> → <E>  ~=  <E>
13:    <E> → <E>  ==  <E>
14:    <E> →  (  <E>  )
15:    <E> →    BOOL
16:    <E> →   NUMBER
17:    <E> →   STRING
18:    <E> →     NIL
```

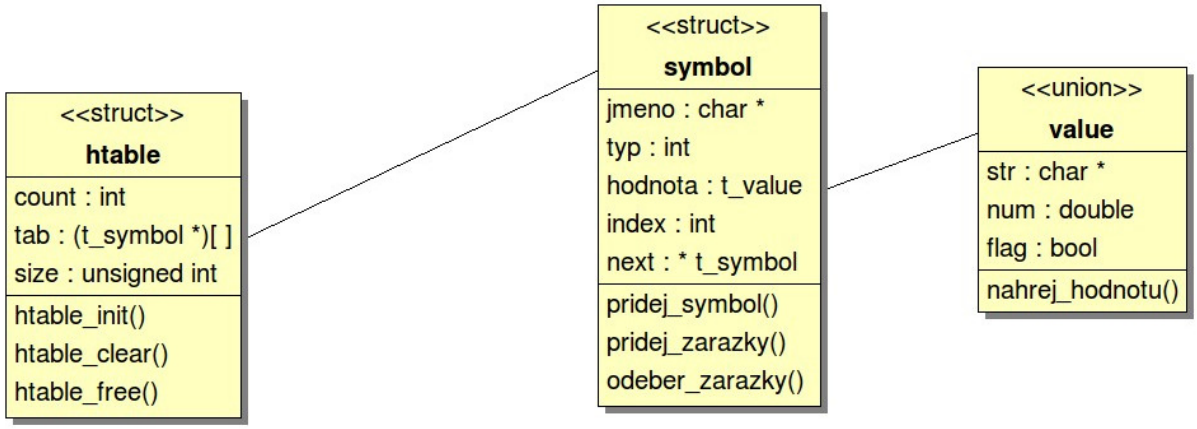
2.3 Sémantická analýza

V jazyce IFJ11 má funkce main význačné postavení, ale nepoužíváme token „main“ , proto všechna její specifika (poslední funkce programu, nemá žádné formální parametry) jsou kontrolovány sémantickou analýzou.

Dále gramatika popisující jazyk IFJ11 není úplně deterministická. Tento problém se vyskytuje u pravidel typu přiřazení návratové hodnoty funkce do proměnné [identifikátor = identifikátor2 ();] a přiřazení hodnoty výrazu do proměnné, přičemž výraz může být i proměnná, tedy [identifikátor = identifikátor2 ;]. Tento problém opět řeší sémantická analýza.

2.4 Implementace vnitřních funkcí a tabulky symbolů

2.4.1 Struktura tabulky symbolů



Obrázek č. 2 Schéma tabulky symbolů

Tabulka symbolů byla předmětem mnoha sporů, ale nakonec jsme se shodli na schématu uvedeném výše. Tabulka **htable** je struktura obsahující pole ukazatelů na jednotlivé symboly, které se indexuje podle jména proměnné (mapovací funkcí). Za zmínku určitě stojí proměnná **count**, která nám určuje pořadové číslo symbolu. Struktura symbolů obsahuje údaje o daném symbolu a v unii **t_value** její hodnotu dle typu. Ve schématu jsou také obsaženy základní funkce, náležící daným strukturám. Pro oddělení jednotlivých úrovní symbolů (zanořené funkce) jsme implementovali funkci **pridej_zarazky(),** která vytvoří novou úroveň. Pracuje se pak jen s lokální úrovní (tj. po zarážky).

V řešení našeho interpretu používáme tři tabulky symbolů. Při syntaktické analýze se vytvoří TS pro lokální proměnné a TS pro literály, kde místo jména používáme řetězec s hodnotou indexu (od TS pro proměnné ji odlišujeme zápornými indexy). TS pro proměnné se po syntaktické a sémantické analýze smaže a interpret si vytvoří novou, kde již nepracuje se jmény, ale s kladnými čísly převedenými na řetězec.

2.4.2 Funkce sort()

Tuto funkci jsme implementovali pomocí **Merge_sortu**. Zvolili jsme její rekurzivní variantu. Funkce nejdříve vytvoří pomocné pole o stejné velikosti jako pole, které chceme seřadit. Funkce rekurzivně dělí pole vždy na dvě poloviny, dokud je co dělit, a poté si nakopíruje do pomocného pole dvě seřazené posloupnosti, které následovně slévá dohromady, a vytvoří tak seřazenou posloupnost.

2.4.3 Funkce find()

Boyer-Moorův algoritmus je založen na zrcadlovém přístupu k vyhledávání (hledáme od konce a postupujeme k začátku). Pro každý vyhledávaný podřetězec je ještě před začátkem vyhledávání vytvořena tabulka **Last**, která každému znaku v ASCII přiřadí jeho nejpravější index v podřetězci. V případě neshody porovnávaných znaků se pak skočí o Last[neshodny_znak], nebo o délku podřetězce (vybírá se minimální hodnota).

2.5 Interpret

Je to poslední fáze zpracování zdrojového programu. Interpret zpracovává posloupnost tříadresného kódu vygenerovaného syntakticko-sémantickým analyzátorem. Data tříadresného kódu jsou implementována jako tabulka (pole) lineárních seznamů instrukcí pro každou funkci.

Pro možnost skoků v instrukcích používáme pomocný zásobník návratových hodnot (ukazatelů na instrukce, které se mají provádět u funkcí) a zásobník návěští pro tvorbu cyklů a podmínek. Pro předávání parametrů do funkce je implementována fronta parametrů, do které se musí před každou funkcí nakopírovat všechny parametry a ze které se bere návratová hodnota po ukončení funkce.

Interpret se ukončí, když se dostane na instrukci RETURN a zásobník návratových hodnot je prázdný.

Instrukce:

LOCAL	PLUS	EQ
READ_N	MINUS	SUBSTR_FIND
READ_L	POW	SUBSTR_CUT
READ_A	MUL	SORT
READ_NUMB	DIV	TYPE
WRITE	MOD	CALL_FUN
LABEL	CONC	RETURN
IFNOTGOTO	LT	QUE_UP
GOTO	GT	QUE_DOWN
NOT	LE	QUE_CLEAN
COPY	GE	

3 Závěr

Tento projekt nás provázel celým semestrem a zabral nám všem mnoho času. Během vývoje jsme řešili mnoho problémů, a proto jsme využili i možnosti konzultace, kde nám konzultující potvrdil naše návrhy. Projekt jsme rozšířili o operátor **MODULO**.

Značným problémem bylo množství dynamicky alokované paměti a její korektní uvolňování. Tento problém jsme nakonec vyřešili systémem **UDMA** (user dynamic memory allocation), který tvořil obal pro funkce malloc(), realloc() a free(), čímž jsme získali záznam o naší práci s pamětí. Před koncem interpretace tento záznam projdeme a uvolníme zbylou paměť.

Pro projekt jsme vytvořili sadu testů, které jsme dělili do tematických celků. Každý celek byl zaměřen na různé chyby, nebo testoval určitou vlastnost jazyka. Pro účely testování jsme taktéž vytvořili různé funkce pro průběžné výpisy hodnot (výpis tabulky symbolů, seznamu instrukcí atd.). Díky tomuto testování se nám nakonec podařilo vytvořit plně funkční interpret.

Metriky

Počet souborů:	17 souborů
Počet řádků zdrojového textu:	8051 řádků
Velikost spustitelného souboru:	52,7 KB (systém Linux, 32 bitová architektura, při překladu bez ladicích informací)
Počet commitů na SVN:	167