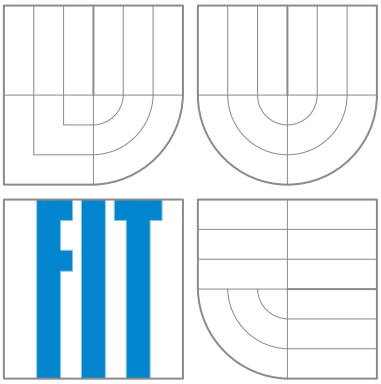
Vysoké učení technické v Brně
Fakulta informačních technologií



Dokumentace k projektu IFJ/IAL
INTERPRET JAZYKA IFJ11
Tým 089, varianta b/1/I

11. prosince 2011

Rozšíření: **MUTREC** : Podpora vzájemné rekurze funkcí
IFTHEN : Podpora příkazu if-then-elseif-else-end dle manuálu
LOCALEXPR : Inicializace proměnné přiřazením výrazu
FORDO : Podpora cyklu for...do...end
MODULO : Podpora binárního operátoru "%"

Branislav Blaškovič (xblask00) - vedoucí: 25%
Pavlína Bártíková (xbarti00): 25%
Tomáš Goldmann (xgoldm03): 25%
Karel Hala (xhalak00): 25%

Obsah

1 Úvod

Tento dokument je dokumentace k projektu **Interpret jazyka IFJ11** do předmětu IFJ - Formální jazyky a překladače a IAL - Algoritmy. Cílem projektu bylo vytvořit program, který dokáže interpretovat jazyk IFJ11, jenž je podmnožinou jazyka **Lua**.

Na výběr bylo několik rozšíření, z kterých jsme si vybrali:

1. MUTREC
2. IFTHEN
3. LOCALEXPR
4. FORDO
5. MODULO

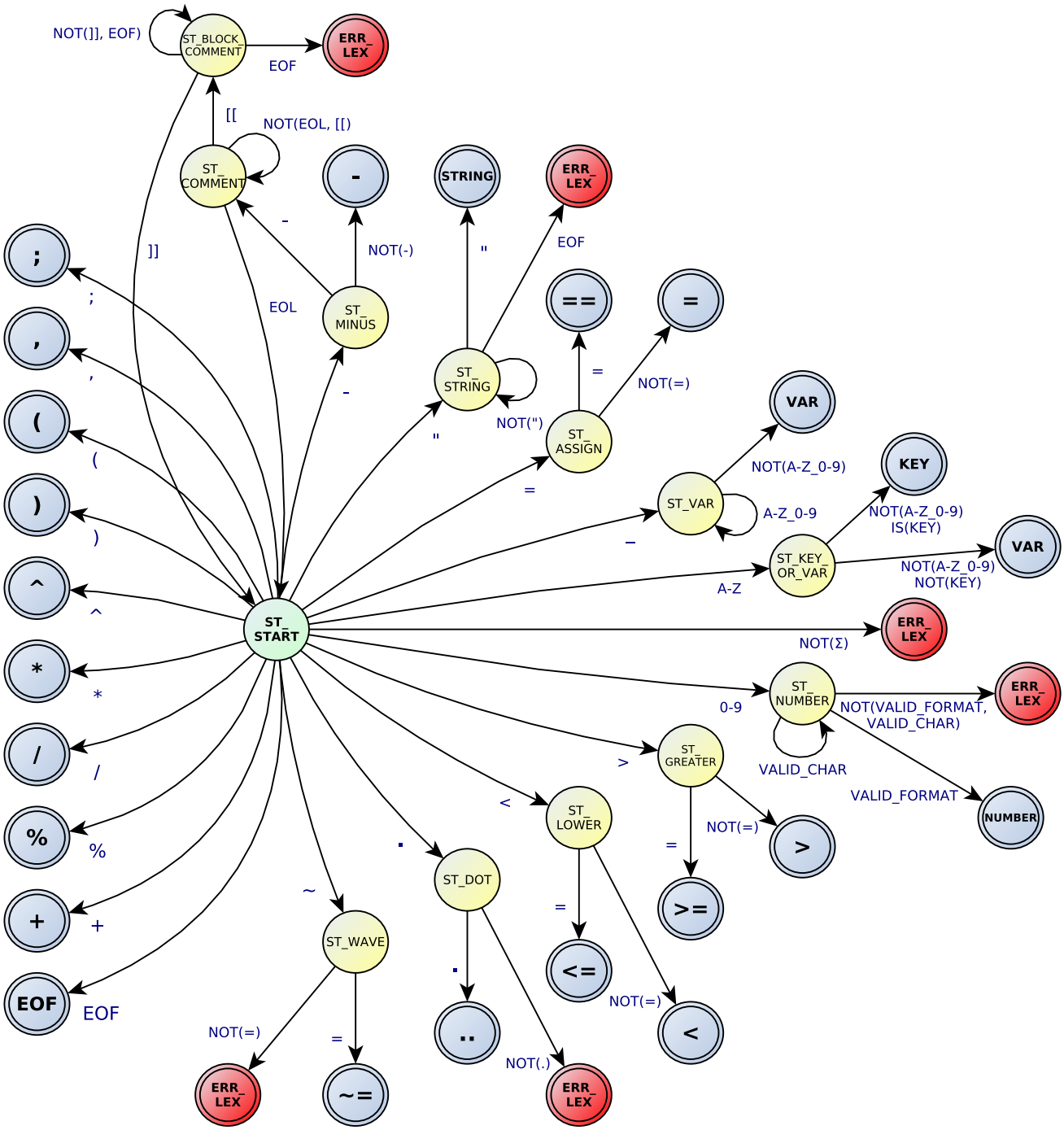
2 Struktura projektu

Program je rozdělený do čtyř logických celků - **lexikální analyzátor, syntaktický analyzátor, sémantický analyzátor a interpret.** Ve skutečnosti syntaktická a sémantická analýza pracuje společně. Syntaktický analyzátor si zažádá token od lexikálního analyzátoru a ověří jeho sémantiku, když je to potřebné. Pokud se program zpracuje až do konce bez chyby, spustí se interpret, který na základě instrukční pásky provede potřebné úkony.

3 Lexikální analyzátor

Lexikální analyzátor načítá po znacích vstupní program a rozděluje ho na tokeny. Je naimplementovaný pomocí konečného automatu (Obrázek 1). Lexikální analyzátor se spouští pouze v případě, když si to vyžádá syntaktická analýza a vrací vždy pouze jeden následující token.

3.1 Konečný automat



Obrázek 1: Konečný automat

4 Syntaktický analyzátor

Syntaktický analyzátor postupně přijímá tokeny z lexikálního analyzátoru. Tokeny může zpracovávat dvěma způsoby. Prvním způsobem je rekurzivní sestup založený na LL gramatice (Tabulka 1). Druhou možností je precedenční syntaktická analýza, která se zde využívá pro výrazy. Při úspěšně splněných podmínkách generuje instrukční pásku pro interpret.

4.1 LL gramatika

Lexikální analyzátor dokáže vrátit vždy právě jeden token. Proto je LL gramatika založená na postupném ověřování tokenů. Funkce jsou navrhnuty tak, že ověřují aktuální token nebo požádají o následující. Token se nedá vrátit zpět. Pokud se nedá rozhodnout na základě jednoho tokenu, funkce požádá o další token, který pak dokáže přenechat následující funkci, když ho nepotřebuje.

<body>	→	<while>
<body>	→	<for>
<body>	→	<assign>
<body>	→	<write>
<body>	→	<if>
<body>	→	<return>
<body>	→	<call_function>
<body>	→	<read>
<body>	→	epsilon
<while>	→	while <expr> do <body> end ;
<if>	→	if <expr> then <if_body> end ;
<if_body>	→	<body>
<if_body>	→	elseif <expr> then <if_body>
<if_body>	→	else <body>
<for>	→	for id = <expr>, <expr> <for_step> do <body> end ;
<for_step>	→	, <expr>
<for_step>	→	epsilon
<return>	→	<expr> ;
<write>	→	write (<expr> <write_param>) ;
<write_param>	→	, <expr> <write_param>
<write_param>	→	epsilon
<assign>	→	id = <expr> ;
<call_function>	→	id = id (<cfunc_param>) ;
<cfunc_param>	→	id <cfunc_param_n>
<cfunc_param>	→	literal <cfunc_param_n>
<cfunc_param>	→	nil <cfunc_param_n>
<cfunc_param>	→	epsilon
<cfunc_param_n>	→	, <cfunc_param> <cfunc_param_n>
<cfunc_param_n>	→	epsilon
<read>	→	id = read(<read_param>) ;
<read_param>	→	positive_number
<read_param>	→	"*n"
<read_param>	→	"*l"
<read_param>	→	"*a"
<function>	→	function id (<func_param>) <body> end ;
<func_param>	→	id <func_param_n>
<func_param>	→	epsilon
<func_param_n>	→	, id <func_param_n>
<func_param_n>	→	epsilon

Tabulka 1: LL gramatika

4.2 Precedenční syntaktická analýza

Precedenční syntaktickou analýzou (analýza zdola nahoru) jsou řešeny *výrazy*. Je zde využita tabulka priorit (Obrázek 2). Funkce pro zpracování výrazů si ukládá jednotlivé tokeny na zásobník a poté je v závislosti na prioritách jednotlivých prvků ukládá do trojadresného (upraveného binárního) stromu (strom, který má větve: operand, operátor, operand), jenž je pak použit interpretem. Na rozdíl od klasické precedenční analýzy zde není brána do úvahy priorita identifikátorů. Sémantika je ověřována přímo při zpracovávání výrazů. Porovnává se zde kompatibilita mezi jedním operandem a jedním operátorem (datové typy proměnných nelze v rámci syntaktické analýzy určit, tudíž jejich kompatibilitu kontroluje až interpret).

	+	-	*	/	%	^	..	<	>	<=	>=	~=	==	(	)	i	$
+	>	>	<	<	<	<	>	>	>	>	>	>	>	<	>	x	>
-	>	>	<	<	<	<	>	>	>	>	>	>	>	<	>	x	>
*	>	>	>	>	>	<	>	>	>	>	>	>	>	<	>	x	>
/	>	>	>	>	>	<	>	>	>	>	>	>	>	<	>	x	>
%	>	>	>	>	>	<	>	>	>	>	>	>	>	<	>	x	>
^	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	x	>
..	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	x	>
<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	x	>
>	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	x	>
<=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	x	>
>=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	x	>
~=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	x	>
==	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	x	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	x	x
)	>	>	>	>	>	>	>	<	<	<	<	<	<	x	>	x	>
i	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	x	x	x

Obrázek 2: Tabulka priorit

5 Sémantický analyzátor

Hlavní úlohou sémantického analyzátoru je kontrolovat logiku napsaného programu v úzké spolupráci se syntaktickým analyzátorem. Generuje globální a lokální tabulku symbolů. Dále kontroluje také správnost výrazů.

5.1 Globální tabulka symbolů

Globální tabulka symbolů zaznamenává funkce. Jeden program má vždy pouze jednu globální tabulku symbolů.

5.2 Lokální tabulka symbolů

Lokální tabulka symbolů se vytváří při začátku deklarace funkce. Uchovává lokální proměnné v rámci jedné funkce. S ukončením syntaktické analýzy dané funkce se ruší i její lokální tabulka symbolů.

6 Interpret

Interpretační část v tomto projektu má za úkol zpracovávat instrukční pásku a vyhodnocovat výrazové stromy. Použité řešení pracuje přímo s výstupem ze syntakticko-sémantické analýzy. Jelikož syntakticko-sémantická analýza přímo generuje instrukční pásku, a zároveň do ní vkládá ukazatel na výrazové stromy, mohli jsme vynechat generaci vnitřního kódu.

6.1 Instrukce

Interpret postupně prochází instrukční pásku a vyhodnocuje jednotlivé instrukce. Tyto instrukce jsou poněkud na vyšší funkční úrovni než instrukce "assemblerovského"typu. Například funkce, která se spouští po načtení instrukce alternativy - INST_CALL_IF. Struktura této instrukce obsahuje odkaz na výrazový strom, ten se vyhodnotí pomocí funkce pro vyhodnocování výrazu a zjistí se, zda je výsledek nastaven tak, aby mohlo dojít k provedení *if*. Pokud není, nastaví se na instrukční pásce jako aktuální instrukce ta, kterou začíná blok dat *else*.

6.2 Pole hodnot

Každá funkce má konečný počet proměnných, které jsou definované při deklaraci a jsou uloženy do tabulky symbolů. Tímto se dá zjistit kolik proměnných funkce obsahuje a místo názvu proměnných se dají ukládat jejich číselné kódy, ve své podstatě indexy. Na straně interpretu se potom může vytvořit tzv. pole hodnot, které obsahuje indexy (bývalé názvy proměnných) a na které se dá odvolávat při vyhodnocování výrazů a přiřazování hodnot.

6.3 Zásobník

Zásobník slouží k přecházení mezi jednotlivými funkcemi, ukládají se na něj jak datové struktury (argumenty), tak i adresy instrukcí (odkud byla funkce volána a návratová adresa). V implementaci ještě ukládáme na zásobník celkový počet argumentů, které je možno vyzvednout ze zásobníku. Maximální velikost zásobníku se nastavuje podle potřeby a po daných blocích se provádí realokace. K tomuto dochází při používání rekurzivního volání funkcí.

6.4 Volání funkce

Pokud je volaná funkce, dojde k nastavení volání funkce. Za voláním funkce se nacházejí jednotlivé argumenty, které se budou postupně přidávat na zásobník. Jakmile se zjistí, že už nenásleduje v instrukční pásce další argument, nastaví se instrukce deklarace funkce jako aktuální a vytvoří se tabulka hodnot. Za deklarační instrukcí můžou následovat parametry. Do jednotlivých parametrů se ze zásobníku přidávají jejich hodnoty argumentů. Pokud by došlo k tomu, že na zásobníku již žádné argumenty nejsou a stále přicházejí instrukce, které říkají, že funkce má další parametry, nastaví se tyto "přebytečné parametry"na datový typ *nil*. Po této nezbytné proceduře dojde k vykonávání těla funkce. Funkce může být opuštěna jak s *return* tak i jako procedura bez *return*. To znamená, že bylo potřeba vytvořit dvojí přístup. První z nich ukončuje funkci bez *return*. Pouze vyčistí zásobník a nastaví návratovou adresu do předešlé funkce. Jestliže se jedná o funkci ukončenou *return*, je zapotřebí vyhodnotit výraz v *return* a následně ho přidat do proměnné v předešlé funkci, která se vztahuje k volání této funkce.

6.5 Vnitřní funkce

Vnitřní funkce jsou identifikovány pomocí typu tokenu, který je z lexikální analýzy. Hlavní rozdíl mezi voláním vnitřní funkce a funkce deklarované skriptem tvoří absence deklarace pro vnitřní funkci. Tato záležitost je řešena identifikací při volání funkce. Pokud dojde k volání vnitřní funkce, zavolá se speciální funkce, která tuto záležitost obsluhuje. Dle typu se vyzvedne počet argumentů ze zásobníku a přiřadí se do parametrů. Pak se jednoduchým způsobem funkce zavolá a výsledek se uloží na místo, odkud byla volána funkce (viz. Volání funkce).

7 Řešení vybraných algoritmů z pohledu předmětu IAL

7.1 Boyer-Mooreův algoritmus

Boyer-Mooreův algoritmus je jeden z nejrychlejších algoritmů pro hledání. Funkce porovnává na základě tabulky skoků řetězec, který dostala, a postupuje podle váhy v tabulce skoků. Funkce je implementována podle script z předmětu IAL první heuristikou ve třech funkcích:

t_char_jump()
Funkce pro upravení jednotlivých skoků podle hledaného slova. Ke každému písmenu abecedy přiřadí určitý počet skoků, o kolik se má posunout doprava při hledání.

find_it()
Samotná funkce pro hledání podřetězce v řetězci. Pokud je potřeba najít pouze jedno písmeno, projde postupně celý řetězec. Pro vše ostatní prochází řetězec za pomoci tabulky skoků.

t_boyer_moor()
Funkce pro snazší volání. Dále se tato funkce stará o návratovou hodnotu, typovou kontrolu a volá funkci na vytvoření tabulky skoků.

7.2 Quick sort

Rekurzivní Quick-sort je jeden z nejrychlejších řadících algoritmů. Postupně řadí část pole rekurzivně. Rozdělení pole je založeno na tzv. *pivotu*. Pro rychlost je kritické vybrat nejlepší *pivot*, my jsme zvolili střed pole. Algoritmus je implementován 2 funkcemi:

sort_me()
Samostatná funkce pro seřazení řetězce. Je volána rekurzivně, čímž je docílena požadovaná rychlost nad částmi pole.

quick_sort()
Funkce pro usnadnění volání řazení. Stará se o typovou kontrolu a návrat seřazeného řetězce.

7.3 Tabulka symbolů (BVS)

Tabulka symbolů byla zadána pomocí binárního vyhledávacího stromu. Binární strom je v programu využit na několika místech k různým úkolům. Tabulka symbolů je používána jak pro jména funkcí, tak pro jména jednotlivých proměnných ve funkci. Tento algoritmus je naimplementován rekurzivně, jak je popsáno v opoře pro předmět IAL. Je to nejefektivnější a nejrychlejší řešení.

8 Rozšíření

8.1 MUTREC

Vzájemná rekurze funkcí je naimplementovaná tak, že interpret si při každém volání funkce vytvoří její novou instanci a pole proměnných, které bude ve funkci potřebovat. Díky tomu se při vzájemné rekurzi proměnné navzájem nepřepisují.

8.2 IFTHEN

Syntaktický analyzátor očekává token "IF"následovaný výrazem a tokenem "THEN". Poté následuje sekvence příkazů. Po jejím ukončení může následovat token "ELSE", "ELSEIF"nebo "END". Posloupnost tokenů "ELSEIF"je řešena cyklem. Na konci každé sekvence příkazů je instrukce pro skok na konec celé podmínky.

8.3 LOCALEXPR

Při načítání tokenu "LOCAL"syntaktický analyzátor ověřuje, zda za názvem proměnné následuje token "=". Pokud ano, zavolá funkci na zpracování výrazů. Následně se do instrukční pásky přidají dva nové záznamy. Jeden pro inicializaci proměnné a druhý, který řídí přiřazení výrazu do proměnné.

8.4 FORDO

Rozšíření FORDO bylo naimplementováno pouze s malou úpravou syntaktického analyzátoru. Cyklus je v instrukční pásce uložen jako cyklus while s tím rozdílem, že má příkaz přiřazení před cyklem a příkaz inkrementace v těle cyklu. Díky tomu nebylo potřebné upravovat interpret.

8.5 MODULO

Operátor modulo má stejnou prioritu jako operátor dělení, proto jeho implementace byla jednoduchá. Na straně interpretu bylo potřebné vytvořit si vlastní algoritmus, aby dokázal pracovat i s desetinnými čísly a nebylo potřebné použít další knihovnu jazyka C. Dále stačilo jen vytvořit novou podmínku v lexikálním analyzátoru a přidat modulo do tabulky priorit.

9 Práce v týmu

9.1 Rozdělení práce

Branislav Blaškovič: syntaktický analyzátor, instrukční páska, dokumentace
Pavlína Bártíková: lexikální analyzátor, precedenční analýza pro výrazy, dokumentace
Tomáš Goldmann: interpret
Karel Hala: funkce do předmětu IAL

9.2 Použité prostředky pro komunikaci a týmovou práci

- Verzovací systém Mercurial
- Instant messenger Skype pro skupinovou písemnou komunikaci
- TeamSpeak pro skupinovou hlasovou komunikaci
- Pravidelná osobní setkání

10 Závěr

Na projektu jsme pracovali přes dva měsíce a využili jsme možnost předběžného odevzdání, což nás donutilo k dokončení projektu o několik týdnů dřív. Zbývající týdny do finálního termínu odevzdání jsme využili k dolaďování, hledání chyb a psaní dokumentace.

Projekt byl časově náročný, ale naučili jsme se vytvořit komplexnější program, používat verzovací systém a vyzkoušeli jsme si práci v týmu. Při řešení projektu jsme často využívali školní fórum i osobní konzultace.

Pro tvorbu programu jsme využívali hlavně editor Geany a dokumentaci jsme psali v jazyce LaTeX.

10.1 Literatura

- Přednášky a scripta k předmětům IFJ a IAL
- Müller K,: Programovací jazyky. Vydavatelství ČVUT, Praha 2001

10.2 Metriky kódu

- Počet řádků zdrojového kódu: **6400**
- Počet zdrojových souborů: **15**
- Počet funkcí: **110**