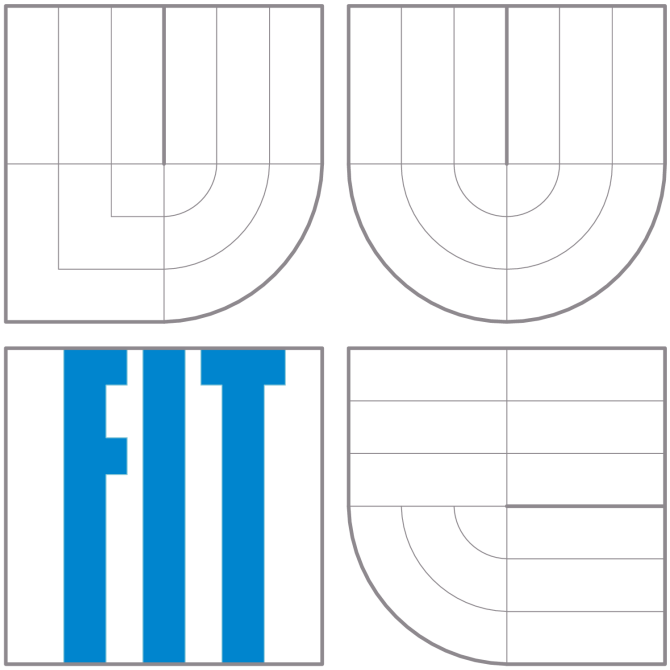


Dokumentace k projektu do předmětů IFJ a IAL

Implementace interpretu imperativního jazyka IFJ14

Tým 40, varianta a/4/II

2. prosince 2014

Autoři:
Kulda Jiří, xkulda00, vedoucí týmu, 25%
Nechutný Stanislav, xnechu01, 25%
Lukeš Petr, xlukes06, 25%
Jásenský Michal, xjasen00, 25%

Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

1 Úvod

Tato dokumentace popisuje návrh a implementaci interpretu imperativního jazyka IFJ14. Seznámí Vás s našimi postupy implementace interpretu i s problémy, které nás v průběhu práce provázely. Dozvíte se také informace o naší týmové spolupráci a dělbě práce. Následující dokument je pro přehlednost rozdělen na několik kapitol a podkapitol.

2 Práce v týmu

Projekt je založen na práci v týmu a dělbě práce. Na začátku jsme si rozdělili práci na jednotlivých modulech a stanovili si komunikační rozhraní. Díky intenzivní práci členů týmu vznikla první verze interpretu poměrně rychle. Od této chvíle bylo upuštěno od předem rozdělené práce na modulech a nastala oprava chyb a případně úprava komunikačního rozhraní, kdy členové týmu modifikovali i kódy jim nepříslušící. Práci v týmu a sdílení kódů nám velmi usnadňoval verzovací systém git.

3 Lexikální analyzátor

Lexikální analyzátor je implementován konečným automatem (Obr 1, Obr 2), který načítá zdrojový kód ze souboru a čte z něj jednotlivé tokeny.

Vstupní soubor získá lexikální analyzátor z globální proměnné *global* popsané v *garbage.h*. Je volán syntaktickým analyzátorem, kterému vrací strukturu *TToken* obsahující typ tokenu a případně dodatečná data.

Syntaktický analyzátor volá funkci *token_get()*. Ta pomocí funkce *token_init()* naalokuje token a popřípadě mu přiřadí odpovídající hodnotu podle načteného lexému ze zdrojového souboru a vrátí jej. Syntaktický analyzátor dále používá další dvě funkce. *token_return_token()* slouží k navrácení již načteného tokenu k opětovnému načtení při dalším volání funkce a *token_get()*. *token_free()* uvolní strukturu tokenu z paměti.

4 Syntaktický analyzátor

Syntaktický analyzátor tvoří jádro celého překladače. Volá lexikální analyzátor a zpracovává navrácené tokeny. Z těchto tokenů podle gramatických pravidel vytváří vnitřní kód.

Pro správný chod syntaktické analýzy bylo zapotřebí vytvořit gramatická pravidla.
Stanovili jsme tato gramatická pravidla:

<file>	->	<var> <function> **begin** <main> **end.**
<function>	->	ε
<function>	->	**function** *fID*(<args>) **:** *returnType*; <forward> <var> **begin** <main> **end;** <function>
<args>	->	ε
<args>	->	*ID* **:** *type* <argsx>
<argsx>	->	ε
<argsx>	->	**;** <args>
<var>	->	ε
<var>	->	**var** <vari>
<vari>	->	ε
<vari>	->	*ID* **:** *TYPE*; <vari>
<main>	->	ε
<main>	->	<code><main>
<code>	->	ε
<code>	->	*ID* **:=** <expr>
<code>	->	**if** <expr> **then** <block> <else>
<code>	->	**while** <expr> **do** <block>
<code>	->	**repeat** <code> **until** <expr>;
<code>	->	**for** *ID* **:=** <expr> **to** <expr> **do** <block>
<code>	->	**case** *ID*: <case>
<block>	->	**begin** <main> **end**
<else>	->	ε
<else>	->	**else** <block>
<case>	->	<expr> **:** <block>
<case>	->	**else** <block>
<case>	->	ε
<forward>	->	ε
<forward>	->	**forward** ;

Tab 1: Pravidla jazyka IFJ14

Pro řešení syntaktické analýzy jsme se rozhodli použít rekurzivního sestupu. Pro tyto účely jsme vytvořili tuto LL tabulku:

	ID	;	var	=	return	if	while	repeat	case	ε	for	begin	program	function	else	expr	forward
<function>										2				3			
<args>	5									4							
<argsx>		7								6							
<var>			9							8							
<vari>	11									10							
<main>										12							
<code>	15					16	17	18	20	14	19						
<block>												21					
<else>										22					23		
<case>										26					25	24	
<forward>										27							28

Tab 2: LL tabulka

Precedenční syntaktická analýza provádí kontrolu výrazů pomocí precedenční tabulky (Tab 3). Tato tabulka byla vytvořena pomocí pravidel jazyka IFJ14.

	Unary -	not	*	/	div	mod	and	+	-	or	xor	=	<>	<	<=	>	>=	in	(	)	ID	func	array	,	$
Unary -	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	#	>
not	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>	>
*	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>	>
/	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>	>
div	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>	>
mod	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>	>
and	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>	>
+	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>	>
-	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>	>
or	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>	>
xor	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>	>
=	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	>	>
<>	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	>	>
<	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	>	>
<=	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	>	>
>	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	>	>
>=	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	>	>
in	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	>	<	<	<	>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	=	#
)	#	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	#	>	#	#	#	>	>
ID	#	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	#	>	#	#	#	>	>
func	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	=	#	#	#	#	#	#
array	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	=	#	#	#	#	#	#
,	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	=	#
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	#	<	<	<	#	#

Tab 3: Precedenční tabulka

5 Vestavěné funkce

Vestavěné funkce jsou základní funkce, které jsou implementovány přímo v interpretu a jsou využitelné v jazyce IFJ14. Podle zadání jsme měli využít pro řazení řetězce algoritmus Merge-sort, pro vyhledávání podřetězce v řetězci Knuth-Morris-Prattův algoritmus a pro implementaci tabulky symbolů hashovací tabulku.

5.1 Řazení řetězce:

Tato funkce je realizována algoritmem nazývaným Merge-sort. Jedná se o algoritmus, který využívá rekurzivního dělení pole na poloviny.

V naší implementaci funkce *find* očekává strukturu *symbolVariable*, ze které získá seřazovaný řetězec a jeho délku. Tuto strukturu si zkopíruje a řazení provádí na kopii, kterou nakonec vrací jako výsledek řazení. Algoritmus začíná od dvou prvních znaků levé poloviny a seřadí je , pokračuje dokud nenarazí na rozdělení polovin. Poté provede to samé s pravou polovinou. Při vynořování z rekurzivního volání spojí poloviny do sebe.

Mezi výhody Merge-sort algoritmu patří logaritmická časová složitost. Jeho nevýhodou je nutnost alokace pomocného řetězce o stejné velikosti řazeného řetězce.

5.2 Vyhledávání řetězce v podřetězci:

Tuto funkci jsme měli realizovat pomocí Knuth-Morris-Prattova algoritmu se složitostí O(m+n). Tento algoritmus vyhledá v řetězci první výskyt podřetězce. Při úspěšném nalezení podřetězce v řetězci vrátí index do řetězce, na němž začíná hledaný podřetězec.

Algoritmus je založen na konečném automatu. Vytvoří tzv. Vektor *fail*, který udává o kolik znaků se máme posunout při neúspěšném porovnání a opět zkusit porovnat. Toto posouvání o více pozic urychluje vyhledávání, protože nedochází k opětovnému porovnání již porovnaných částí.

V naší implementaci algoritmus očekává dva parametry typu *symbolVariable*, ze kterých vyčte řetězce. Při neúspěšném vyhledávání vrací hodnotu 0, jinak navrací pozici prvního znaku podřetězce v řetězci. Navracení hodnoty probíhá vytvořením a naplněním nové struktury *symbolVariable*, která je pak funkcí navracena.

5.3 Tabulka symbolů:

Tabulka symbolů je tvořena pomocí hashovací tabulky, která má v nejlepším případě složitost O(1) a v nejhorším případě lineární složitost. Hashovací tabulka je struktura obsahující pole ukazatelů o N prvcích a integer s číslem N, které se používá v hashovací funkci.

Tyto ukazatelé ukazují na prvky, které tvoří jednosměrně vázaný seznam struktur obsahujících informaci o názvu a typu funkce nebo proměnné, ukazatel na strukturu reprezentující proměnnou nebo funkci a ukazatel na další položku v seznamu. Vyhledáváni se provádí tak, že hashovací funkcí získáme index na místo v poli kde je ukazatel na jednosměrně vázaný seznam ve kterém se poté vyhledává podle názvu proměnné nebo funkce sekvenčně. Díky hashovací funkci je vyhledávání a vkládání prvku do hashovací tabulky velmi rychlé.

6 Interpret

Jeho úkolem je vykonávání kódu. V našem případě jsme si zvolili interpret pracující s tříadresným kódem, který je vygenerován generátorem.

Generátor je volán syntaktickým analyzátorem a generované instrukce ukládá do seznamu instrukcí. Instrukce generuje podle pravidel. Generátor se skládá ze dvou částí, kde jedna část se stará o generování výrazů a druhá o ostatní kód.

Interpret na základě instrukce ze seznamu instrukcí rozhoduje jaké akce mají být vykonány. Máme celkem 23 instrukcí, které interpret vyhodnocuje. Matematické funkce jsou řešeny pomocí *do_math()*. Tato funkce detekuje sémantické chyby jako je například dělení nulou. Dále také detekuje chybu typové kompatibility ve výrazu.

7 Rozšíření

7.1 MINUS

Do pravidel jazyka IFJ14 bylo přidáno pravidlo pro unární mínus. V interpretu je řešeno jako otočení znaménka a v případě proměnné typu řetězec je vrácena chyba.

7.2 REPEAT

Řešen podobně jako cyklus *while*, ale podmínka je testována na konci bloku kódu.

7.3 ELSEIF

Při ukončení bloku části *if* je přidána skoková instrukce pro přeskočení bloku části *else* a opačně.

7.4 BOOLOP

Do precedenční tabulky byly přidány pravidla pro logické operace. V interpretu řešeny pomocí dvou na sobě nezávislých proměnných, kde první z nich ukládá částečný výsledek a druhá aktuálně zpracovávaný výraz.

7.5 FOR

Řešen obdobně jako ostatní cykly, ale s pevným počtem průchodů.

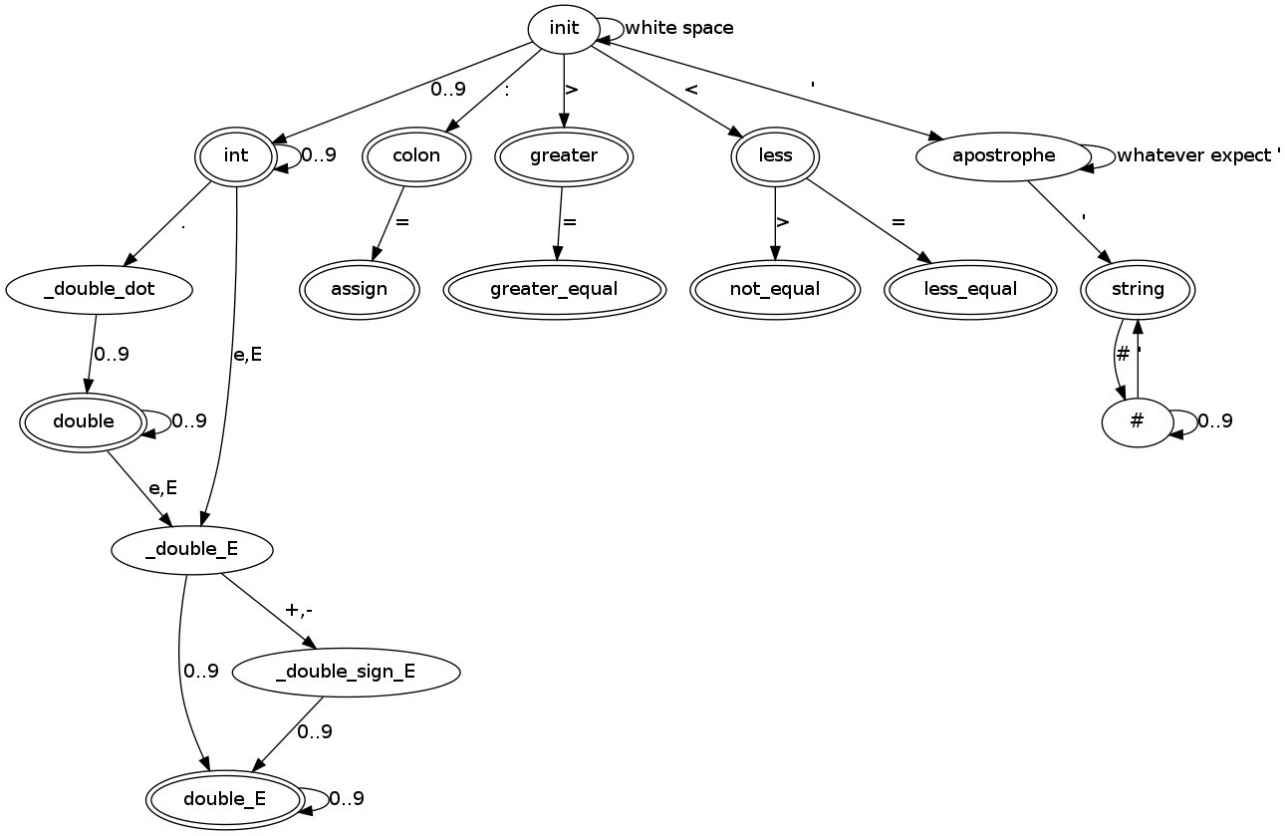
7.6 MSG

Chybová hlášení jsou vypisována ve formátu uvedeném v zadání jazyka IFJ14.

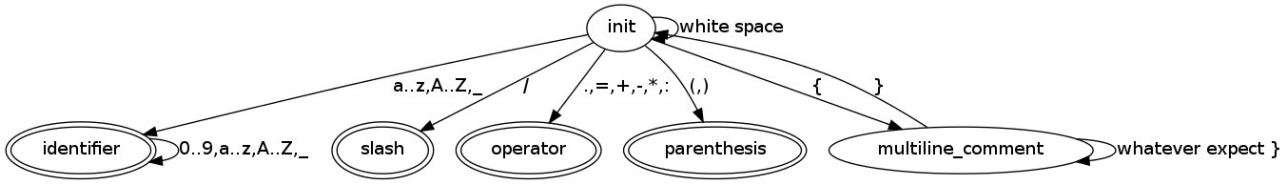
7.7 FUNEXP

Řešen pomocí rekurzivního volání precedenční analýzy.

Reference



Obr 1: Konečný automat popisující lexikální analyzátor část 1



Obr 2: Konečný automat popisující lexikální analyzátor část 2