Dokumentácia k projektu

Interpret jazyka IFJ15

tím 067, varianta a/2/I

Rozšírenia

- FUNEXP
- SIMPLE
- WHILE

Autori

- Michal Cyprian, xcypri01, 2BIA, 22%
- Michal Ďurista, xduris04, 2BIA, 19%
- Radovan Sroka, xsroka00, 2BIB, 21%
- Tomáš Sýkora, xsykor25, 2BIB, 19%
- Tomáš Ščavnický, xscavn00, 2BIB, vedúci tímu, 19%

1. Úvod

V tejto dokumentácii je vysvetlený princíp fungovania interpretu jazyka IFJ15 spolu s popisom vývoja tohto interpretu. IFJ15 je podmnožina jazyka C++. Celý interpret sme rozdelili do 4 hlavných častí:

- lexikálna analýza
- syntaktická analýza
- precedenčná analýza
- sémantická analýza
- generovanie a vykonávanie kódu

Tieto podmnožiny interpretu budú popísané v nasledujúcej kapitole.

2. Štruktúra interpretu

Interpret jazyka IFJ15 vypracovaný naším tímom sa skladá zo štyroch hlavných častí: lexikálny analyzátor, syntaktický analyzátor, sémantický analyzátor a interpreter. Okrem toho v projekte využívame nami zostrojené buffre na jednotné ukladanie informácií, čo nám zaručuje jednoduché a bezpečné narábanie s pamäťou. Interpret vykonáva syntaxou riadený preklad, z čoho vyplýva, že syntaktický analyzátor je jeho ústrednou časťou.

1. Lexikálny analyzátor

Zdrojový kód jazyka IFJ15 prijímaný interpretom je tokenizovaný pomocou lexikálneho analyzátora (ďalej lexer), ktorý je navrhnutý na princípe konečného automatu (str. 6).

Lexer na pokyn syntaktického analyzátora prečíta zo zdrojového súboru jeden lexém, identifikuje jeho typ a vytvorí štruktúru TToken. Štruktúra TToken pozostáva zo samotného lexému a typu lexému. Táto štruktúra je následne lexerom uložená do dynamického bufferu. Po uložení lexer vráti syntaktickému analyzátoru index na danú štruktúru TToken v buffry, aby s ňou mohol ďalej pracovať.

Na nasledujúcej strane je zobrazený graf konečného automatu, použitého v našom lexeri.

2. Syntaktický analyzátor

Syntaktický analyzátor (ďalej SA) tvorí ústrednú časť celého interpretu. Kontroluje syntaktickú správnosť zdrojového kódu zo vstupu, definovanú podľa LL gramatiky (str. 7).

SA má na vstupe lexikálny analyzátor, ktorý mu na požiadanie vráti jeden lexém zo zdrojového kódu. Následne môže byť výraz vyhodnotený v precedenčnej analýze a následne vráte späť SA.

Na základe tokenov z lexikálneho analyzátora sa SA snaží symulovať tvorbu derivačného stromu. Postupuje v tom na princípe rekurzívneho zostupu. Pokial sa derivačný strom nepodarí nasimulovať, jedná sa o syntaktickú chybu a analýza sa ukončí.

SA využíva tabulku symbolov ako primárnu dátovu štruktúru.
Komunikácia medzi SA a tabulkov symbolov je vykonávaná nepriamo. Sprostredkuje ju sémantický analyzátor, s ktorým SA úzko komunikuje.
Výstupom SA sú inštrukcie na inštrukčnom buffery.

3. Precedenčný analyzátor

Kedykoľvek syntaktická analýza narazí v zdrojovom kóde na výraz, predá všetky zdroje precedenčnej analýze (ďalej PA).
PA pracuje so zásobníkom, na ktorý ukladá tokeny získané zo scannera a postupne ich redukuje na neterminály na základe krátkych a dlhých pravidiel.
Jadrom PA je precedenčná tabuľka (str. 8). Podľa obsahu bunky na aktuálnych indexoch PA vyberie operácou, ktorú má vykonať v nasledujúcom kroku. Ako indexy do tabuľky sa používajú typy tokenov na vrchole zásobníka a na vstupe. PA postupne vyhodnocuje priority operátorov a vykonáva redukcie, paralelne volá funkcie sémantickej kontroly a následne generuje inštrukcie na výpočet hodnoty výrazu.
Ak sa vo výraze nachádza volanie funkcie PA ho identifikuje, zariadi jej vykonanie a na vrchol zásobnika vloží jej navratovú hodnotu.
V prípade že je výraz syntakticky korektný PA zabezpečí aby sa počas interpretácie dostala na akutálny vrchol behového zásobníka výsledná hodnota výrazu a predá syntaktickej analýze jeho typ.

4. Sémantický analyzátor

Sémantická analýza predstavuje v našej implementácii interpretu IFJ15 wrapper nad tabulkou symbolov. Tým výrazne odbremeňuje SA a PA od práce s binárnymi stromami. Sémantická analýza sa využíva najm

5. Interpret

Postupne vykonáva inštrukcie trojadresného kódu z inštrukčného buffera, ktorý bol generovaný syntaktickým, precedenčným a sémantickým analyzátorom. Aktuálne hodnoty premenných získa interpret z behového zásobníka.

6. Implementačné špecifiká

Resources, buffer, TToken, debug

3. IAL

1. Heap sort

Heap sort je radiaca metóda využívajúca špeciálnu dátovú štruktúru halda. Halda umožňuje veľmi efektívne vykonať operácie vloženia prvku a výberu najväčšieho prvku. To umožňuje efektívne zoradenie poľa jednoducho pomocou vloženia jeho prvkou do haldy a následovne postupného vyberania najväčšieho prvku.

2. Knuth-Morris-Prattov algoritmus

Algoritmus používaný na vyhľadávanie podreťazca v reťazci. Na vyhľadanie podreťazca využivá konečný automat. Výhodou tohto algoritmu je, že pri výskyte nezhody medzi časťou reťazca a podreťazcom vie algoritmus rozhodnúť, kde môže nastať ďalšia zhoda a preto môže preskočiť už porovnané znaky.

3. Tabuľka symbolov

Tabuľka symbolov je implementovaná pomocou binárneho vyhľadávacieho stromu (ďalej BVS). (…)
Každý BVS v našom interprete reprezentuje jeden blok kódu. Uzly BVS reprezentujú funkcie a premenné deklarované v danom bloku kódu. Fakt, že funkcie nemôžu byť v jazyku IFJ15 definované vo vnúty iných funkcií a neexistuju globálne premenné, nám zjednodušilo prácu tým spôsobom, že sme si vymysleli istý *general scope*, v ktorom sú definované všetky funkcie a vedeli sme, že v ďalších blokoch, do ktorých SA vstúpi budú definované len premenné.
BVS sa ukladajú do dynamického buffera. Pri vstupe do bloku kódu sémantická analýza naalokuje a uloží BVS reprezentujúci daný scope do dynamického buffera a jeho index pushne na zásobník aktuálnych blokov kódu. Pri výstupe z bloku kódu sa z daného zásobnika index popuje. Tento systém zaručí platnosť premenných im odpovedajúcich scope-och.

4. Rozšírenia

1. FUNEXP

Ak sa vo výraze nachádza volanie funkcie PA ho identifikuje, zariadi jej vykonanie a na vrchol zásobnika vloží jej navratovú hodnotu.
V prípade že je výraz syntakticky korektný PA zabezpečí aby sa počas interpretácie dostala na akutálny vrchol behového zásobníka výsledná hodnota výrazu a predá syntaktickej analýze jeho typ.

2. SIMPLE

3. WHILE

5. Vývoj

1. Rozdelenie práce

- Michal Cyprián - precedenčná analýza, inštrukčná sada, string buffer
- Michal Ďurista - lexikálna analýza, vstavané funkcie
- Radovan Sroka - syntaktická analýza, interpret, debug modul
- Tomáš Sýkora - binárný vyhľadávací strom, sémantická analýza
- Tomáš Ščavnický - lexikálna analýza, sémantická analýza

2. Použité prostriedky

Ako komunikačné médium sme v tíme používali aplikáciu Slack. Okrem toho sme najprv raz, a posledný mesiac pred odovzdaním aj viac krát, do týždňa mali osobné stretnutia, kde sme zhodnotili aktuálny stav a nastavili ciele, ktoré je potrebné dosiahnuť.

Na archivovanie programovej časti projektu sme využili Git, pričom ako hosting pre náš repozitár sme využili službu GitHub, kde sme vďaka školskému účtu mali súkromný hosting zadarmo.

6. Záver

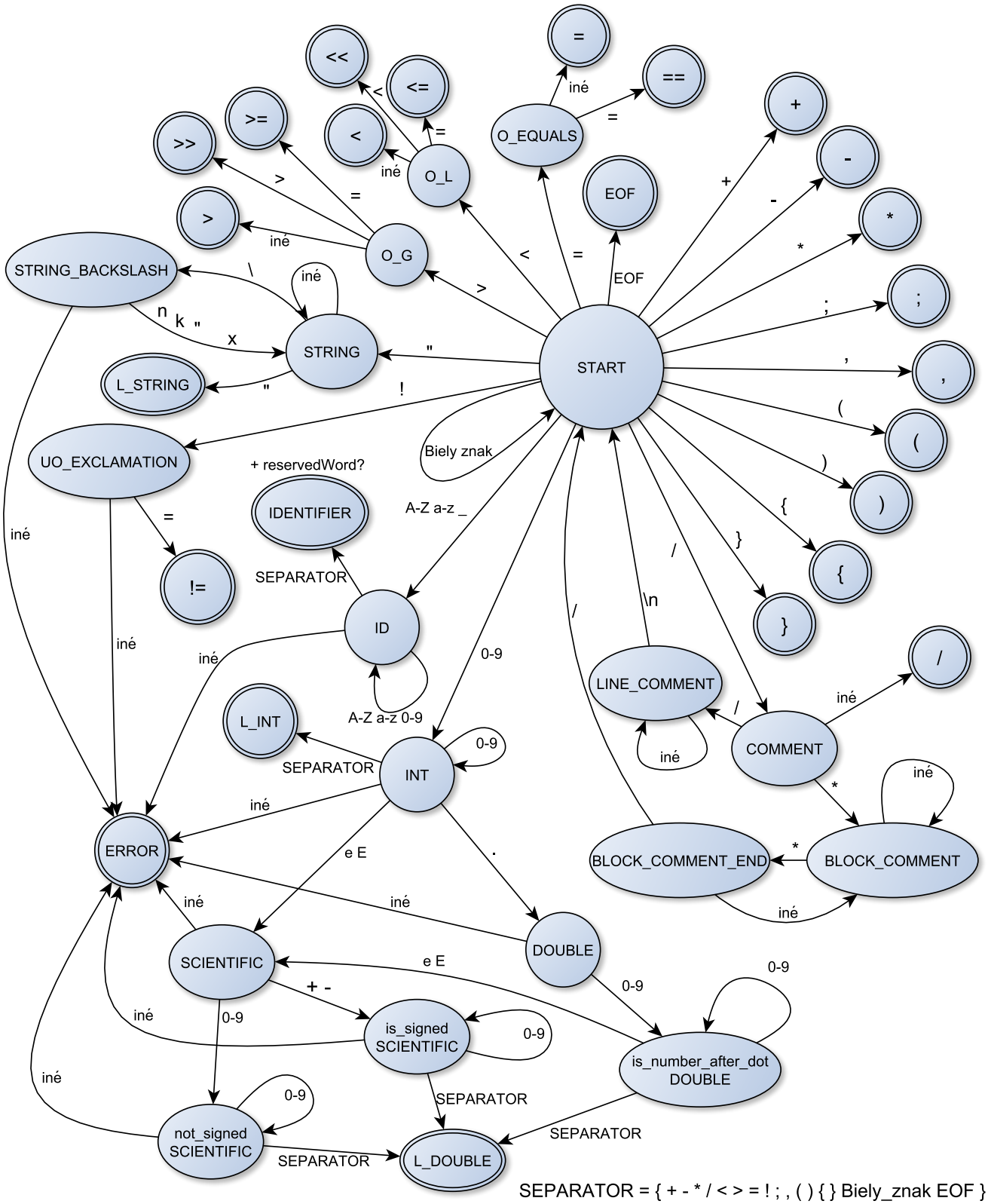
Tento projekt bol veľmi plodný z hľadiska toho, čo sme sa pri ňom naučili. Počnúc navrhovaním architektúry mierne väčšieho programu, cez využivanie služieb ako git a GitHub a Slack, až po chápanie a smerovanie dynamiky v týme ľudí so všetkým, čo k tomu patrí.

Tempo prednášok pre náš tím nebol problém. Ku koncu projektu sme čerpali informácie najmä z minuloročných prednášok a tiež sme sa často snažili riešiť problémi vlastným spôsobom.

Body som sa, ako vedúci, rozhodol rozdeliť nerovnomerne, pretože som chcel aspoň symbolicky (a do mieri, ktorú mi bodovanie projektu umožňuje) ohodnotiť členov tímu, ktorí prejavili nadpriemernú aktivitu.

7. Tabulky a grafy

1. Graf konečného automatu implementovaného v lexikálnej analýze



											PREDICT
1	<program> →	<func>	<program_n>								int, double, string
2	<program_n> →	epsilon									$
3	<program_n> →	<program>									int double string
4	<zlozeny_prikaz> →	<deklaracia_var	;	<zlozeny_prikaz>							int double string auto
5	<zlozeny_prikaz> →	<prikaz>	;	<zlozeny_prikaz>							id
6	<zlozeny_prikaz> →	<construction>	<zlozeny_prikaz>								if for while do cout cin return
7	<func> →	<static_part>	<tail_func>								int, double, string
8	<static_part> →	<type>	id	(	<params>	)					int double string
9	<tail_func> →	{	<zlozeny_prikaz>	}							{
10	<tail_func> →	;									;
11	<deklaracia_var> →	<type>	id	<tail_var>		ifi					int double string
12	<deklaracia_var> →	auto	id	=	<rvalue>						auto
13	<tail_var> →	epsilon									;
14	<tail_var> →	=	<rvalue>								='
15	<priradenie> →	id	=	<rvalue>							id
16	<func_call> →	(	<args>	)							(
17	<args> →	epsilon									)
18	<args> →	<rvalue>	<args_n>								rvalue
19	<args_n> →	epsilon									)
20	<args_n> →	,	<rvalue>	<args_n>							,
21	<params> →	<type>	id	<params_n>							int double string
22	<params> →	epsilon									)
23	<params_n> →	,	<type>	id	<params_n>						,
24	<params_n> →	epsilon									)
25	<type> →	double									double
26	<type> →	int									int
27	<type> →	string									string
28	<prikaz> →	id	<tail_prikaz>								id
29	<tail_prikaz> →	<func_call>									(
30	<tail_prikaz> →	=	<rvalue>								='
31	<construction> →	<if_else>									if
32	<construction> →	<for_cycle>									for
33	<construction> →	<cin>									cin
34	<construction> →	<cout>									cout
35	<construction> →	<return>									return
36	<construction> →	<while>									while
37	<construction> →	<do>									do
38	<if_else> →	if	(	<rvalue>	)	<tail_if>	<else>				if
39	<tail_if> →	<deklaracia_var	;								int double string auto
40	<tail_if> →	<construction>									if for while do cout cin return
41	<tail_if> →	<prikaz>	;								id
42	<tail_if> →	{	<zlozeny_prikaz>	}							{
43	<else> →	epsilon									int double string auto id if for while do cout cin return } !else
44	<else> →	else	<tail_constr>								else
45	<for_cycle> →	for	(	<for_first>	;	<for_second>	;	<for_third>	)	<tail_constr>	for
46	<for_first> →	<deklaracia_var>									int double string auto
47	<for_second> →	<rvalue>									rvalue
48	<for_third> →	id	=	<rvalue>							id
49	<while> →	while	(	<rvalue>	)	<tail_constr>					while
50	<do> →	do	{	<zlozeny_prikaz>	}	while	(	<rvalue>	)	;	do
51	<return>	return	<rvalue>	;							return
52	<tail_constr> →	<deklaracia_var	;								int double string auto
53	<tail_constr> →	<construction>									if for while do cout cin return
54	<tail_constr> →	{	<zlozeny_prikaz>	}							{
55	<tail_constr> →	<prikaz>	;								id
56	<cout> →	cout	<cout_params>	;							cout
57	<cout_params> →	<<	<rvalue>	<cout_params_n>							<<
58	<cout_params_n> →	epsilon									;
59	<cout_params_n> →	<<	<rvalue>	<cout_params_n>							<<
60	<cin> →	cin	<cin_params>	;							cin
61	<cin_params> →	>>	id	<cin_params_n>							>>
62	<cin_params_n> →	epsilon									;
63	<cin_params_n> →	>>	id	<cin_params_n>							>>
	<rvalue> →	send to PA	{term (Literal (3, “hello”, …)), vyraz, id}								rvalue
	<id>										

3. Tabulka precedenčnej analýzi

	=='	**>**	**<**	**>=**	**<=**	**!=**	**+'**	**-'**	*****	**/**	**)**	**(**	**$**	**id**	**li**
=='	R	R	R	R	R	R	S	S	S	S	R	S	R	S	S
>	R	R	R	R	R	R	S	S	S	S	R	S	R	S	S
<	R	R	R	R	R	R	S	S	S	S	R	S	R	S	S
>=	R	R	R	R	R	R	S	S	S	S	R	S	R	S	S
<=	R	R	R	R	R	R	S	S	S	S	R	S	R	S	S
!=	R	R	R	R	R	R	S	S	S	S	R	S	R	S	S
+'	R	R	R	R	R	R	R	R	S	S	R	S	R	S	S
-'	R	R	R	R	R	R	R	R	S	S	R	S	R	S	S
*****	R	R	R	R	R	R	R	R	R	R	R	S	R	S	S
/	R	R	R	R	R	R	R	R	R	R	R	S	R	S	S
)	R	R	R	R	R	R	R	R	R	R	R	E	R	E	E
(	S	S	S	S	S	S	S	S	S	S	H	S	E	S	S
$	S	S	S	S	S	S	S	S	S	S	E	S	E	S	S
id	R	R	R	R	R	R	R	R	R	R	E	R	E	E	E
li	R	R	R	R	R	R	R	R	R	R	E	R	E	E	E