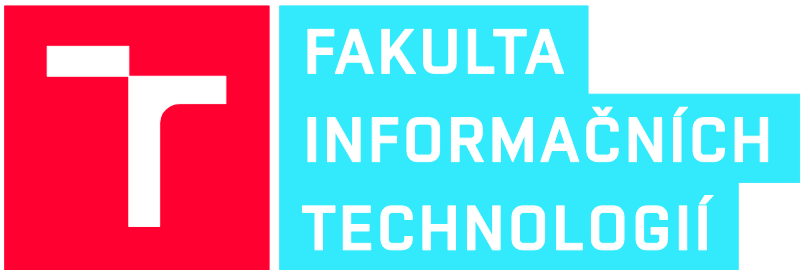


Dokumentácia projektu pre predmety IFJ a IAL

Interpret jazyka IFJ15

Tím 030, varianta b/3/II

13. Decembra 2015

Vedúci tímu:	**Ján Mochňak**	**(xmochn00)**	**20%**
Členovia:	Tibor Dudlák	(xdudla00)	20%
	Martin Krajňák	(xkrajn00)	20%
	Dávid Prexta	(xprext00)	20%
	Patrik Segedy	(xseged00)	20%

Obsah

1 Úvod

Táto dokumentácia popisuje implementáciu interpretu imperatívneho programovacieho jazyka IFJ15. Jazyk IFJ15 je podmnožinou jazyka C++ 11. V dokumentácii sa nachádza spôsob riešenia lexikálnej analýzy, syntaktickej analýzy a interpretácie. Kapitola 3 sa zaoberá riešením algoritmov do predmetu IAL a kapitola 4 sa venuje práci v tíme.

2 Riešenie interpretu

V tejto kapitole je popísané vlastné riešenie lexikálnej analýzy, syntaktickej analýzy, generovania 3 adresného kódu a interpretácie.

2.1 Lexikálna analýza

Úlohou lexikálnej analýzy je:

- Rozpoznanie tokenov - lexém zo vstupného súboru a odstránenie bielych znakov
- Zistenie lexikálnych chýb vo vstupnom súbore

Lexikálny analyzátor číta vstupný súbor a rozoznáva v ňom lexémy, ktoré ďalej predáva syntaktickej analýze. O ďalší token musí syntaktický analyzátor požiadať lexikálny analyzátor prostredníctvom funkcie `scanner_get_next_token()`.

Činnosť lexikálnej analýzy je riadená konečným automatom, ktorého diagram je zobrazený v prílohe A.

2.2 Syntaktická a sémantická analýza

Syntaktická analýza sa skladá z dvoch častí:

- Syntaktická analýza pre kontext jazyka
- Syntaktická analýza pre spracovanie výrazov

Vstupom syntaktickej analýzy je token vrátený syntaktickým analyzátorom a výstupom syntaktickej analýzy je trojadresný kód.

Pri konštrukcii syntaktickej analýzy pre kontext jazyka založeného na LL-gramatike sme využili metódu *rekurzívneho zostupu.* Tokeny vrátené lexikálnou analýzou sú spracovávané pomocou pravidiel LL gramatiky. Ak je potrebné spracovať výrazy, je zavolaná precedenčná analýza. LL gramatika sa nachádza v prílohe B.

Spracovanie výrazov syntaktickej analýzy je implementované *precedenčnou syntaktickou analýzou.* Pri spracovávaní výrazov využívame prevod do reverznej poľskej notácie (postfixovej notácie), kvôli zjednodušeniu vyhodnocovania výrazu. Vďaka postfixovej notácii vieme vygenerovať inštrukcie z výrazu. Precedenčná tabuľka je priložená v prílohe C.

Počas syntaktickej analýzy taktiež vyhodnocujeme sémantické chyby, kontrolujeme definície a deklarácie premenných a funkcií.

2.3 Generovanie 3 adresného kódu a inštrukcií

Trojadresný kód, z ktorého sme vytvorili inštrukcie, sme generovali počas syntaktickej analýzy. Inštrukcie pre riadiace štruktúry sme generovali počas rekurzívneho zostupu. Počas precedenčnej syntaktickej analýzy sa vygenerujú inštrukcie pre porovnanie výrazov a aritmetické operácie.

2.4 Interpretácia

Interpret dostane zoznam obsahujúci všetky inštrukcie, ktoré následne interpretuje. Interpret rozozná typ a parametre inštrukcie a na základe týchto údajov sa spustí konkrétny proces, ktorý túto inštrukciu vykoná.

Na prácu s premennými pri vykonávaní inštrukcií využívame *call stack*. Pre každú volanú funkciu sa vytvorí nový *call stack*, obsahujúci parametre funkcie. K parametrom sa pristupuje na základe *offsetu*.

Výsledok každej operácie sa vloží na vrchol zásobníku.

3 Využité algoritmy

Táto kapitola popisuje spôsob implementácie algoritmov potrebných do predmetu IAL.

3.1 Boyer-Moorov algoritmus

Boyer-Moorov algoritmus (BMA) vychádza z úvahy, že pri porovnávaní vzorky s reťazcom je možné preskočiť také vzorky, o ktorých sa dá prehlásiť, že sa nemôžu rovnať. Čím dlhšia je vyhľadávaná vzorka, tým väčší počet znakov v reťazci môže algoritmus BMA preskočiť. Vo všeobecnosti je algoritmus rýchlejší ak je dĺžka vyhľadávanej vzorky väčšia. [1]

```
Vzorka: example

Text: here is a simple example
```

```
5.                                                 e x a m p l e
4.                                       e x a m p l e
3.                             e x a m p l e
2.                       e x a m p l e
1. e x a m p l e
   h e r e   i s   a   s i m p l e   e x a m p l e
```

Tabuľka 1 Príklad vyhľadávania BMA

1. Porovnávanie prebieha od posledného znaku vzorky k prvému. Porovnáva sa posledný znak vzorky, znak „s“ sa nerovná znaku „e“ a nenachádza sa v hľadanej vzorke, preto sa vyhľadávanie posunie o dĺžku vzorky.
2. Znak „p“ sa nerovná znaku „e“, ale nachádza sa vo vzorke. Preto sa vzorka posunie a zarovná sa podľa pozície znaku „p“.
3. Znaky „e“ až „m“ vzorky sa zhodujú s textom. Znak „i“ sa nezhoduje so znakom „a“ a „i“ sa ani nenachádza v hľadanej vzorke. Začiatok vzorky sa posunie za pozíciu znaku „i“.
4. „x“ sa nerovná hľadanému „e“, ale nachádza sa vo vzorke. Vzorku zarovnáme, aby znak „x“ vo vzorke bol na rovnakej pozícii ako v texte.
5. Postupne kontrolujeme znaky sprava a zistíme, že sme našli hľadaný reťazec.

3.2 Shell sort

Metóda radenia so znižujúcim sa prírastkom. Pracuje na princípe bublinového vkladania, a teda pracuje na princípe vkladania. Na rozdiel od typických metód s kvadratickou zložitosťou, ako je Bubble-insert, sa prvky presúvajú k miestu, na ktoré patria s väčším krokom. Shell sort je nestabilná metóda radenia, ktorá pracuje „in situ“.

Pracuje na opakovaných priechodoch podobných bublinovému vkladaniu, keď vymieňa prvky vzdialene o rovnaký krok.

Napríklad v prvej etape je s použitím kroku 4 každá zo 4 sekvencií spracovaná jedným bublinovým priechodom. V druhej etape sa krok zníži na 2, vytvoria sa 2 podsekvencie a každá sa spracuje jedným bublinovým priechodom. V poslednej etape sa na celú sekvenciu aplikuje bublinový priechod s krokom jedna. Tím je radenie ukončené. [1]

V našej implementácii zvolíme prvú medzeru ako polovicu z dĺžky reťazca. Každú nasledujúcu medzeru vypočítame ako *medzera/2,2*. Autorom tejto sekvencie medzier pre Shell sort je Marcin Ciura.

3.3 Tabuľka symbolov pomocou tabuľky s rozptýlenými položkami

Na tabuľku pre uchovávanie symbolov sme podľa zadania využili tabuľku s rozptýlenými položkami. Výhodou tabuľky s rozptýlenými položkami je rýchlosť hľadania symbolov v nej uložených. Tabuľka pozostáva z poľa ukazovateľov na jednosmerne viazaný zoznam synoným, každá položka zoznamu obsahuje symbol a ukazovateľ na nasledujúcu položku zoznamu.

V tabuľke sa symbol vyhľadá na základe mapovacej (*hashovacej)* funkcie, ktorá na základe hľadaného reťazca vypočíta index do poľa ukazovateľov na zoznam, v ktorom by sa hľadaný prvok mal nachádzať. Následne sa prechádza zoznamom, pokým prvok nie je nájdený alebo kým sa nedostane na koniec zoznamu.

4 Použité knižnice

Pri riešení projektu sme využili 2 hlavičkové súbory z knižnice *klib*, ktorá je šírená pod licenciou MIT/X11 license a je dostupná z http://attractivechaos.github.io/klib/ . Konkrétne sme využili súbory:

- kvec.h – obsahuje generické dynamické pole
- klist.h – obsahuje generický jednosmerne viazaný zoznam a *memory pool* pre alokáciu tohto zoznamu

Túto knižnicu sme využili z dôvodu jednoduchého vytvárania a alokovania zoznamov a dynamických polí.

5 Práca v tíme

Na prvom stretnutí sme si navrhli sadu nástrojov, pomocou ktorých budeme projekt vypracovávať. Rozhodli sme sa pre verzovací systém git a dohodli sme sa na jednotnom štýle písania kódu. Zvolili sme si spôsob testovania pomocou nástroja CircleCI. Spôsob vývoja sme sa rozhodli realizovať na platforme Linux pre lepšiu kompatibilitu na seba naväzujúcich modulov a taktiež kvôli finálnemu testovaniu na školskom serveri merlin.

5.1 Rozdelenie práce v tíme

Tibor Dudlák vstavané funkcie, testovanie, konečný automat lexikálnej analýzy, návrh spôsobu interpretácie, návrh a implementácia inštrukcií

Martin Krajňák precedenčná analýza, tabuľka symbolov, testovanie, návrh spôsobu interpretácie

Ján Mochňak precedenčná analýza, interpetácia, zabezpečenie spolupráce jednotlivých blokov, testovanie, vedenie tímu, návrh dátových typov, návrh spôsobu interpretácie, návrh a implementácia inštrukcií

Dávid Prexta implementácia syntaktickej, sémantickej a lexikálnej analýzy, testovanie

Patrik Segedy dokumentácia, testovanie, návrh spôsobu interpretácie, návrh a implementácia inštrukcií

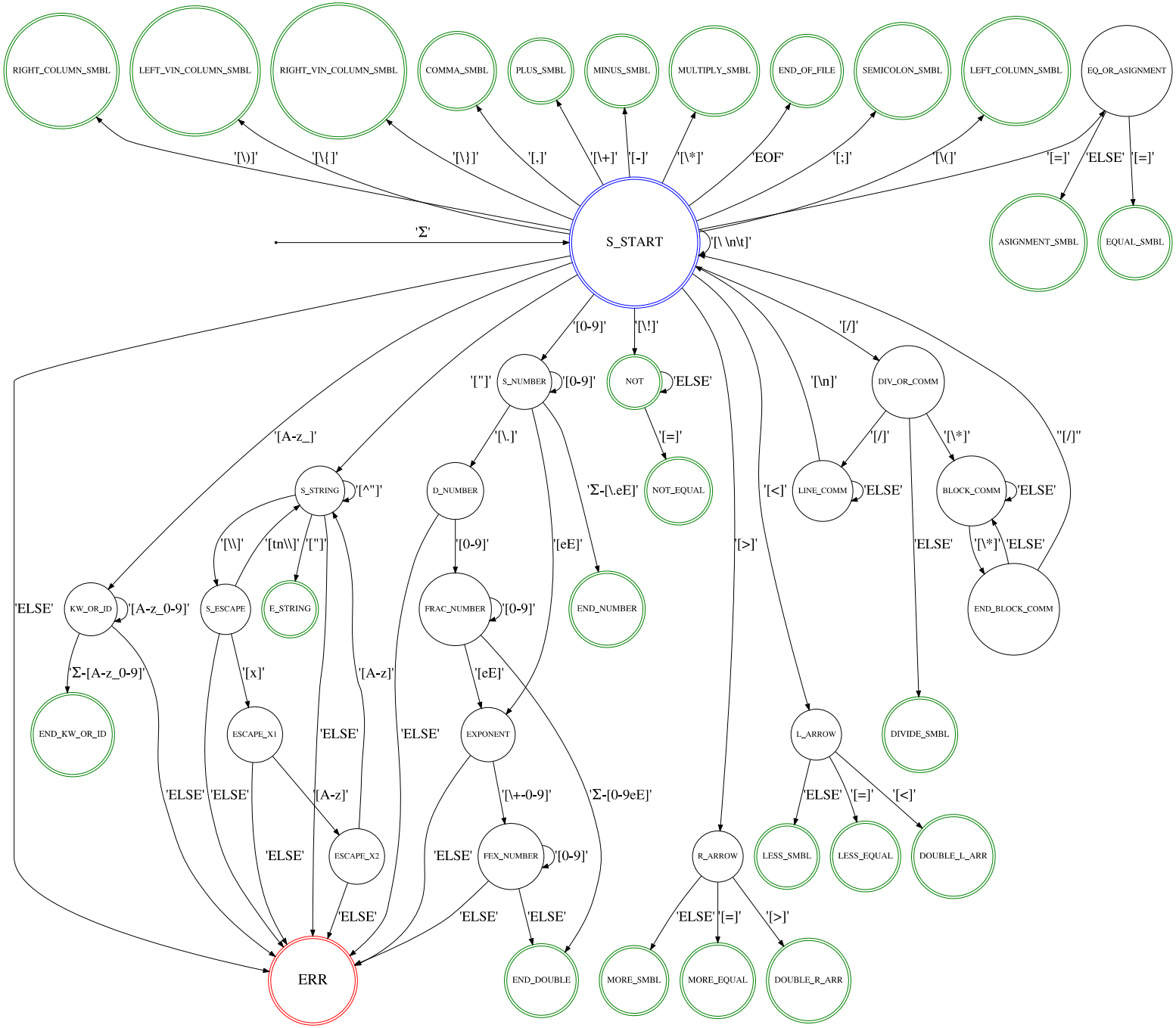
6 Záver

Na tomto projekte sme si odskúšali prácu v tíme na zatiaľ najväčšom projekte počas nášho doterajšieho štúdia. Pri tomto projekte sme mali možnosť naučiť sa pracovať s verzovacím systémom git. Projekt hodnotíme ako veľkú skúsenosť, hlavne čo sa týka práce v tíme, ale taktiež aj ako dobrý prostriedok k naučeniu sa princípu funkcie interpretu, ako aj jeho implementácie.

Literatúra

[1] HONZÍK, Jan. *Algoritmy - Studijní opora*. Verze:14 N. Brno: Vysoké učení technické, 2014.

A Konečný automat lexikálnej analýzy



B LL Gramatika

1. <parse_fn> → int main ( ) { <fn_body> }
2. <parse_fn> → type id (<fn_params>) <fn_end>
3. <fn_end> → ; <parse_fn>
4. <fn_end> → { <fn_body>
5. <parse_fn> → ε

6. <fn_params> → type id <fn_params_n>
7. <fn_params> → ε
8. <fn_params_n> → , type id <fn_params_n>
9. <fn_params_n> → ε

10. <fn_body> → id = <assign> ; <fn_body>
11. <fn_body> → { <fn_body> }
12. <fn_body> → ε
13. <fn_body> → if (<exp>) { <fn_body> } else { <fn_body> } <fn_body>
14. <fn_body> → for (type id <for_dec> ; <exp> ; id = <assign>) <fn_body>
15. <for_dec> → = <assign>
16. <for_dec> → ε

17. <fn_body> → cin >> id <read> <fn_body>
18. <read> → >> id <read>
19. <read> → ;

20. <fn_body> → cout << <print> ; <fn_body>
21. <print> → id <print_n>
22. <print> → literal <print_n>
23. <print_n>→ << <print>
24. <print_n>→ ε

25. <fn_body> → return <exp> ; <fn_body>
26. <fn_body> →<declaration> <fn_body>
27. <declaration> → auto id = <assign> ;
28. <declaration> → type id <init>
29. <init>→ ;
30. <init>→ = <assign> ;

31. <assign> → func (<param_list>)
32. <assign> → id <exp>
33. <assign> → literal <exp>

34. <param_list> → ε
35. <param_list> → id <param>
36. <param_list> → literal <param>
37. <param> → , <param_list>
38. <param> → ε

C Gramatika pre výrazy

1. <exp> → (<exp>)
2. <exp> → <exp> + <exp>
3. <exp> → <exp> - <exp>
4. <exp> → <exp> * <exp>
5. <exp> → <exp> / <exp>
6. <exp> → <exp> < <exp>
7. <exp> → <exp> > <exp>
8. <exp> → <exp> == <exp>
9. <exp> → <exp> >= <exp>
10. <exp> → <exp> <= <exp>

D Precedenčná tabuľka

vstup

zásobník

	+	-	*	/	<	>	<=	>=	'=='	!=	(	)	a	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
'=='	<	<	<	<	>	>	>	>	>	>	<	>	<	>
!=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	'='	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
a	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	