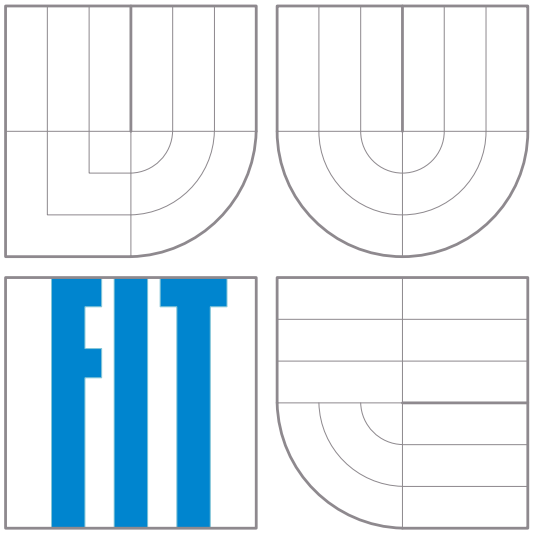


Dokumentácia k projektu pre predmety IFJ a IAL

Interpret jazyka IFJ13

Tým 1, varianta a/1/I

15. decembra 2013

Autori: Vladimír Čillo (xcillo00) 25 % - Vedúci
Oliver Nemček (xnemce03) 25%
Štefan Martiček (xmarti62) 25%
Filip Ilavský (xilavs01) 25%

Rozšírenia: FUNEXP
ELSEIF
MINUS
LOGOP

Obsah

1 Úvod

Tento dokument popisuje vývoj interpretu jazyka IFJ13 vo variante a/1/I - vstavaná funkcia *find_string* využíva Knuth-Moris-Prattov algoritmus, tabuľka symbolov je implementovaná pomocou binárneho stromu a vstavaná funkcia *sort_string* využíva pre radenie algoritmus Quick sort.

2 Analýza

Etapa analýzy a návrhu tvorila prevažnú časť vývojového cyklu programu (cca 30 dní). Našim cieľom bolo vytvoriť jednoduchú a efektívnu implementáciu. Pre úplnosť uvádzame, že implementované riešenie pozostáva z nasledujúcich častí:
Lexikálny analýzátor, syntaktický analyzátor, sémantický analyzátor spojený s generátorom vnútorného kódu (inštrukčnej pásky) a samotný interpret - t.j. funkčný celok ktorý vykonáva vygenerovaný kód.

2.1 Rozdelenie práce v rámci týmu

Lexikálna analýza, Dokumentácia - Vladimír Čillo
Syntaktická analýza - Štefan Martiček
Sémantická analýza, Interpret - Oliver Nemček
Vstavané funkcie - Filip Ilavský

2.2 Priebeh vývoja

Prvým krokom vývoja bolo vytvorenie systému správy zdrojových kódov. Na tento účel nám poslúžil privátny repozitár na stránkach www.github.com.

Vvysoké riziko častých zmien v návrhu viedlo k snahe o inkrementálny prístup k vývoju riešenia, korý spočíval najmä v častých schôdzach týmu a spoločnej diskusii o jednotlivých fázach implementácie, čo nám umožnilo odhaliť veľke množstvo problémov skôr, ako by spôsobili väčšiu časovú stratu.

3 Implementácia

Nasledujúce riadky obsahujú popis modulov, z ktorých sa prográm skladá a zároveň dokumentujú vybrané vlastnosti našej implementácie.

3.1 Lexikálny analyzátor - Scanner

Lexikálny analyzátor tvorí vstupný bod interpretu a zabezpečuje prvotné spracovanie zdrojového kódu.

3.1.1 Modul file_io.c

Vzhľadom na snahu o efektivitu sme sa rozhodli súbor so zdrojovým kódom prečítať (nakopírovať do pamäti) naraz a následne ho spracovať v jednom priechode.Vstupom lexikálneho analyzátoru je teda ukazovateľ na reťazec. Funkcie getc() a ungetc() zo štandardnej knižnice sme nahradili vlastnými, upravenými tak, aby dokázali pracovať s našou pamäťovou reprezentáciou zdrojového kódu. Aktuálnu pozíciu v "súbore"si scanner uchováva v globálnej premennej.

3.1.2 Modul scanner.c

Modul scanner.c obsahuje implementáciu konečného automatu, ktorý vykonáva lexikálnu analýzu a niekoľkých ďalších pomocných funkcí z našej dielne nahradzujúcich funkčne ekvivalentné štandardné funkcie, ktoré boli pre naše potreby príliš neefektívne.

3.1.3 Popis činnosti

Scanner transforumuje vstupný súbor na postupnosť tokenov, ktoré volaním funkcie scanner_get_token() postupne predáva syntaktickému analyzátoru. Keďže celý súbor bol v rámci inicializácie scanneru nakopírovaný do pamäte, lexiálny analyzátor nevyužíva dynamickú alokáciu pamäte.

Štruktúra token obsahuje typ tokenu, číslo riadku, na ktorom sa token nachádza, počet znakov (v prípade identifikátorov a reťazcov) a union na uchovanie atribútu.

3.2 Syntaktická analýza

Syntaktický analyzátor - parser - kontroluje syntaktickú správnosť programu a zároveň riadi celú fázu prekladu. Postupnosť tokenov, ktorú získava od lexikálneho analyzátoru transformuje na pseudoinštrukcie, ktoré predáva na kontrolu sémantickému analyzátoru, ktorý generuje inštrukčné pásky (tento mechanizmus je podrobnejšie popísaný v časti 3.3).
Syntaktická analýza je implementovaná metódou rekurzívneho zostupu a vychádza z nasledujúcej LL-gramatiky:

3.2.1 LL-gramatika

1:	<begin>	$\Longrightarrow$	<?php + biely znak <st_list>
2:	<st_list>	$\Longrightarrow$	while (expr1 { <st_list2> <st_list>
3:	<st_list>	$\Longrightarrow$	if (expr1 { <st_list2> <st_list3> <st_list>
4:	<st_list>	$\Longrightarrow$	function id (<par> <par_list> { <st_list2> <st_list>
5:	<st_list>	$\Longrightarrow$	return expr2 <st_list>
6:	<st_list>	$\Longrightarrow$	premenna = expr2 <st_list>
7:	<st_list>	$\Longrightarrow$	EOF
8:	<st_list2>	$\Longrightarrow$	while (expr1 { <st_list2> <st_list2>
9:	<st_list2>	$\Longrightarrow$	if (expr1 { <st_list2> <st_list3> <st_list2>
10:	<st_list2>	$\Longrightarrow$	return expr2 <st_list2>
11:	<st_list2>	$\Longrightarrow$	premenná = expr2 <st_list2>
12:	<st_list2>	$\Longrightarrow$	}
13:	<st_list3>	$\Longrightarrow$	else { <st_list2>
14:	<st_list3>	$\Longrightarrow$	elseif (expr1 { <st_list2> <st_list3>
15:	<st_list3>	$\Longrightarrow$	*epsilon*
16:	<par>	$\Longrightarrow$	, id parametra [1]
17:	<par>	$\Longrightarrow$	*epsilon*
18:	<par_list>	$\Longrightarrow$	id parametra <par_list>
19:	<par_list>	$\Longrightarrow$	)

Počiatočný neterminál: <st_list>
Neterminály: <st_list>, <st_list2>, <st_list3>, <par>, <par_list>
Terminály: všetky tokeny
expr1 - výraz ukončený znakom ')'
expr2 - výraz ukončený znakom ';'
EOF - koniec vstupu

[1] Jedná sa v podstate o premennú, označenie id parametra je použité kôli kompatibilite so zadaním.

3.2.2 Modul expressions.c

Modul expressions.c zapúzdruje precedenčnú syntaktickú analýzu, ktorá na základe precedenčnej tabuľky kontroluje správnosť aritmetických a logických (rozšírenie LOGOP) výrazov. Precedenčná analýza sa riadi nasledujúcimi pravidlami :

3.2.3 Pravidlá pre spracovanie výrazov

1:	E $\Longrightarrow$	E op E
2:	E $\Longrightarrow$	(E)
3:	E $\Longrightarrow$	var
4:	E $\Longrightarrow$	f(E)
6:	E $\Longrightarrow$	f(E, x)
7:	E $\Longrightarrow$	E and E
8:	E $\Longrightarrow$	E or E
9:	E $\Longrightarrow$	E || E
10:	E $\Longrightarrow$	E && E
11:	E $\Longrightarrow$	! E
12:	E $\Longrightarrow$	- E

op = operátor $\in$ {., !==, ===, +, -, *, /, <=, >=, <, >}
var = premenná
f = identifikátor funkcie
x = ľubovolný počet neterminálov E oddelených čiarkami

3.2.4 Rozšírenie ELSEIF

Implementácia tohto rozšírenia si vyžiadala úpravu gramatiky tak, aby príkaz if po ktorom nenasleduje $else$ nebol hodnotený ako synatktická chyba. Ďalej bolo potrebné pridať $elseif$ medzi kľúčové slová a upraviť príslušnú časť syntaktickej analýzy tak, aby toto kľúčové slovo mohlo byť interpretované ako $else$ a if. V rámci sémantickej analýzy a generátoru kódu tým pádom neboli potrebné žiadne zmeny, pričom platí konvencia párovania $else$ s najbližším if.

3.2.5 Rozšírenie FUNEXP

V tomto prípade bola opäť potrebná úprava gramatiky a precedenčná tabuľka bola doplnená o identifikátor funkcie a čiarku. Ďalšie zmeny neboli potrebné, nakoľko sémantický analyzátor disponuje funkciou $evalf()$, ktorá vyhodnocuje funkcie bez ohľadu nato, kde sa volanie funkcie nachádza.

3.2.6 Rozšírenie MINUS

Implementácia rozšírenia MINUS vyžadovala zmeny v precedenčnej analýze, konkrétne bolo potrebné rozlíšiť binárne a unárne mínus, čo v praxi znamená, že $5--2$ nie je považované za syntaktickú chybu, ale táto posupnosť je interpretovaná ako dva tokeny unárneho mínus, takže výsledkom tejto operácie bude $5-(-2)=7$.

3.2.7 Rozšírenie LOGOP

Toto rozšírenie vnieslo do lexikálneho analýzátoru 4 nové typy tokenov (lexémy and, or, &&, ||), pričom *and* a *or* sú považované za kľúčové slová. Precedenčná analýza bola upravená doplnenením príslušných logických operátorov do precedenčnej tabuľky a pribli 2 nové inštrukcie - AND a OR.

3.3 Sémantická analýza

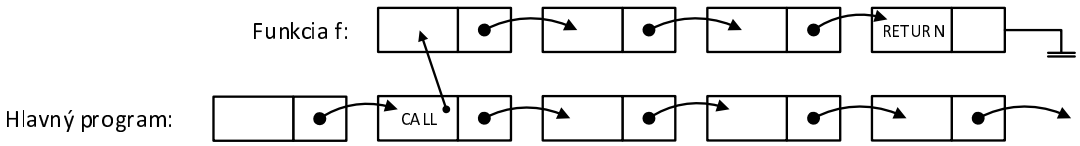
Fáza sémantickej analýzy prebieha paralelne so syntaktickou analýzou, pričom je riadená syntaktickým analyzátorom. V rámci sémantickej analýzy prebieha kontrola deklarácií premenných (premenná má priradenú hodnotu => považuje sa za deklarovanú) a definícií funkcií, pričom kontrola definícií funkcií sa spustí až v okamihu, ked syntaktický analyzátor spracuje celý vstup. Keďže jazyk IFJ13 je dynamicky typovaný, typové kontroly prebiehajú až vo fáze interpretácie.

3.3.1 Inštrukčné pásky

V rámci našej implementácie odpovedá každej funkcii jedna inštrukčná páska (lineárny zoznam inštrukcií). Naša inštrukčná sada pozostáva z inštrukcií START, CREATE, MOV, RET, PUSH, COND, JMP, CALL, CALL_BUILTIN, INC, DEC, PLUS, MINUS, CONCAT, AND, OR, EQUAL, NONEQUAL, DIV, MUL, LESS, GREATER, LESSEQ, GREATEREQ.

Každý program je tvorený minimálne jednou - hlavnou inštrukčnou páskou. V prípade volania funkcií je na inštrukčnú pásku pridaná inštrukcia CALL, ktorej operandom je ukazovateľ na pásku danej funkcie. Pred zavolaním funkcie sa na zásobník uloží adresa nasledujúcej inštrukcie za inštrukciou CALL. Túto adresu zo zásobnika vyberie inštrukcia RETURN na konci volanej funkcie a program pokračuje inštrukciou na tejto adrese.

Tu nastáva problém, že v čase syntaxou riadeného prekladu ešte nemusí byť známy skutočný počet parametrov funkcie. Nami implementované riešenie tomuto faktu nevenuje pozornosť, nakoľko v prípade funkcie volanej s väčším počtom parametrov ako je počet jej formálnych parametrov sa tento problém prejaví iba prebytočnými inštrukciami PUSH na inštrukčnej páske. Tie sú odstránené hneď ako sa nájde definícia funkcie. (Pozn.: Predávanie parametrov funkciám je podorobnejšie popísané v časti 3.4.1.) Pri návrhu tohto riešenia sme využili poznatky z kurzu Asemblery, kde sme sa dôkladne oboznámili s problematikou nelinearity kódu, najmä pokiaľ ide o predávanie riadenia a parametrov procedúram, funkciám atď.



Obr. 1: Schéma volania funkcie f na inštrukčnej páske

3.4 Interpret

V prípade že celková analýza zdrojového kódu nenájde chybu, inštrukčná páska vygenerovaná v časti sémantickej analýzy je predaná interpretu, ktorý postupne vykonáva jednotlivé inštrukcie.

3.4.1 Predávanie parametrov

Predávanie parametrov volaným funkciam zabezpečujú inštrukcie CREATE a PUSH. Inštrukcia CREATE skontroluje, či je na zásobníku dostatočne veľké spojité miesto pre uloženie potrebného počtu parametrov (premenných). Ak nie, potrebné miesto sa prialokuje. Následne nato inštrukcia PUSH na toto pamäťové miesto uloží potrebný parameter (Jednému parametru odpovedá 1 vykonanie inštrukcie PUSH).

3.5 Vstavané funkcie

Implementácie vstavaných funkcií sa (až na výnimky uvedené nižšie) nachádzajú v module built-in.c.

3.5.1 Funkcia sort_string - Quicksort

Funkcia sort_string, podľa špecifikácie v zadaní, pre radenie používa algoritmus quicksort a jej implementácia sa nachádza v module ial.c.

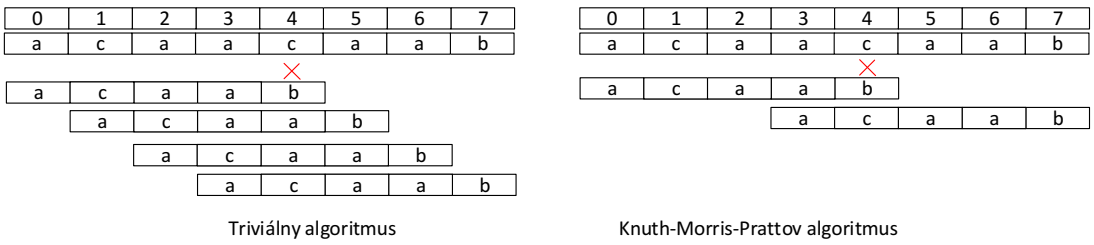
Quicksort, je metóda radenia je založená na rozdelení množiny vstupných dát na dve časti, na základe pseudomediánu (hodnota ktorá je v strede, niekedy označovaná ako *pivot*) a následnom preusporiadaní prvkov tak, že v jednej časti ostanú hodnoty menšie alebo rovné a v druhej väčšie ako pivot. Aplikáciou tohto mechanizmu na všetky časti vznikne zoradené pole.

Z uvedeného popisu vyplýva, že tento algoritmus je možné zápísať iteratívne alebo rekurzívne. Zvolili sme rekurzívnu variantu, pretože sa nám zdá prehľadnejšia.

3.5.2 Funkcia find_string - Knuth-Morris-Prattov algoritmus

Knuth-Morris-Prattov algoritmus (ďalej len KMP) je vyhľadávací algoritmus, ktorý funguje na báze pomocného vektoru fail vytvoreného na základe hľadaného reťazca. Vektor fail slúži na určenie najbližšieho možného miesta (indexu) k aktuálnej pozícii porovnávania, od ktorého môže pri zlyhaní algorimus pokračovať. Ako vidíme na obrázku 2, pri nezhode KMP nezačína, na rozdiel od triviálneho algoritmu, porovnávať na indexe 1, ale na indexe 3.

KMP je zložený z dvoch častí, prvá časť je vytváranie fail vektoru z hľadaného reťazca, druhá samotné vyhľadávanie podreťazca. Implementácia tejto funkcie sa nachádza v module ial.c.



Obr. 2: Porovnanie KMP s triviálnym algoritmom pri vyhľadávaní reťazca “acaab” v texte “acaacaab”

3.6 Tabuľka symbolov

Podľa špecifikácie v zadaní, je tabuľka symbolov implementovaná pomocou binárneho stromu a nachádza sa v module ial.c. Užitočné data uzlu binárneho stromu tvorí štruktúra *metadata_var*, ktorá obsahuje ukazovateľ na meno premennej, dĺžku tohto mena a poradie danej premennej v zásobníku parametrov

a lokálnych premenných. Tabuľka symbolov obsahuje len informácie o premenných, informácie o funkciách sú uložené v tabuľke funkcií, ktorá je takisto implementovaná pomocou binárneho stromu.

Tabuľka symbolov zaniká po sémantickej analýze a je nahradená zásobníkom parametrov a lokálnych premenných.

4 Záver

Implementované riešenie, pokiaľ nám je známe, spĺňa všetky formálne požiadavky definované zadaním, navyše je interpret rozšírený o logické výrazy, podporu volania funkcií ako súčasť výrazu, podporu konštrukcie if-elseif-else a unárne mínus. Projekt predstavoval pre všetkých členov prínos praktických skúseností a celkovo ho z pohľadu riešiteľov hodnotíme ako zaujímavý.

5 Metriky kódu

Počet súborov:	23
Riadkov kódu celkovo:	8365
Z toho komentáre:	14%
Počet príkazov:	4440
Počet funkcí:	103
Priemerne príkazov na funkciu:	38.3

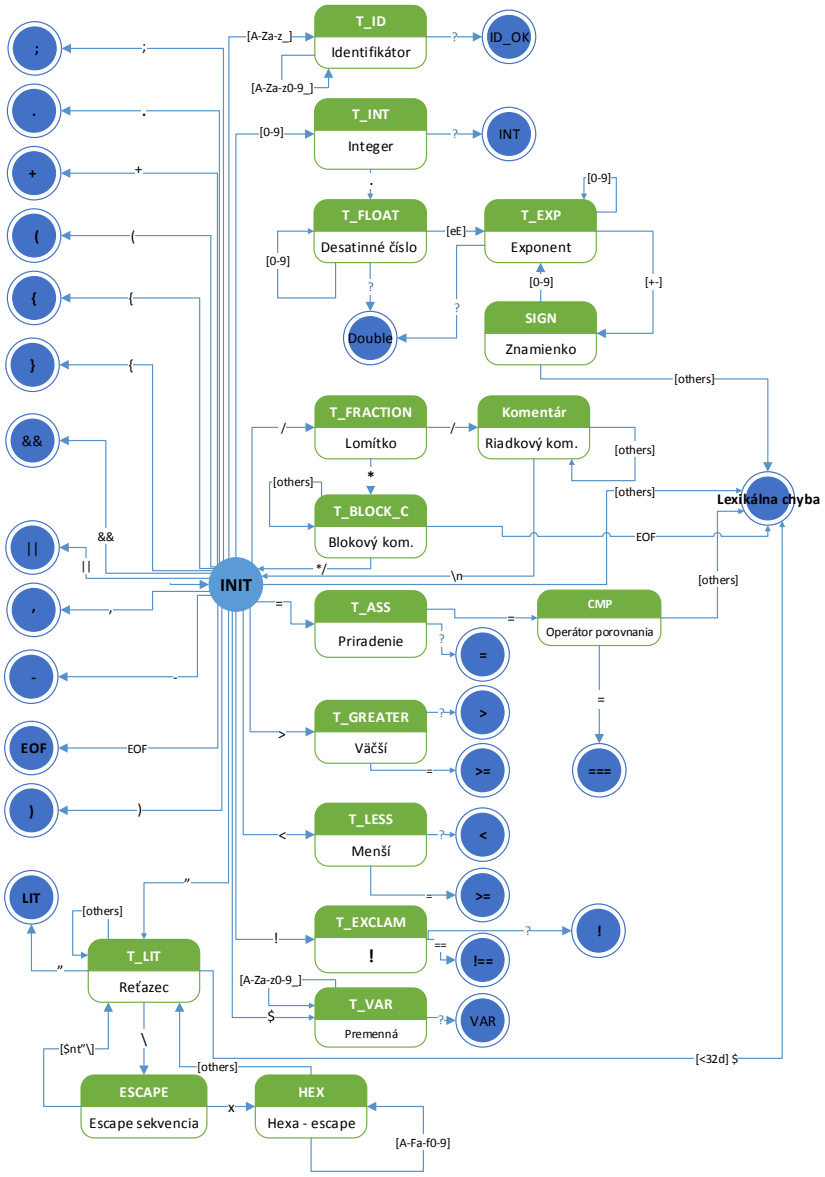
Literatúra

[1] Prof. Ing. Jan M. Honzík, CSc., *ALGORITMY - Studijní opora*. Verzia: 12-K, 2012.

[2] Prof. RNDr. Alexander Meduna, CSc., Ing. Roman Lukáš, Ph.D. *Formální jazyky a překladače - Studijní opora* Verze: 1.2006+revize 2009-2012

6 Prílohy

6.1 Model lexikálneho analyzátoru



Pozn.: Znaky v hranatých zátvorkách definujú množinu znakov ktoré spôsobia daný prechod; symbol ? označuje oddeľovač, t.j. znak, ktorý nie je súčasťou tokenu a po jeho prijatí je volaná operácia ungetc(), ktorá zabezpečí znovunačítanie daného znaku pri ďalšom volaní scanneru; others označuje všetky ostatné znaky