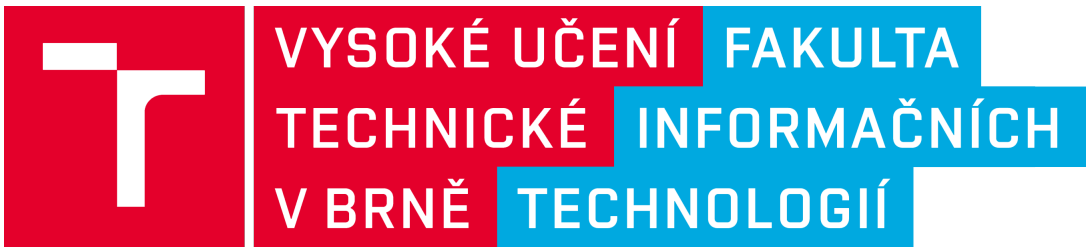


Dokumentácia k projektu z predmetov IFJ a IAL

Implementace překladače imperativního jazyka IFJ21

Tým 102, varianta II

Členovia tímu:

Novák David	*xnovak2x*	...	25%
Kozub Tadeáš	*xkozub06*	...	25%
Klimecká Alena	*xklime47*	...	25%
Bubáková Natália	*xbubak01*	...	25%

December 2021

Obsah

1 Úvod

V tejto dokumentácii je popísaný postup implementácie prekladača imperatívneho jazyka IFJ21 založenom na jazyku Teal/Lua do cieľového jazyka IFJcode21. Náš tím si zvolil variantu II, a tak je tabuľka symbolov implementovaná pomocou tabuľky s rozptýlenými položkami („hash table“).

Celý proces pozostáva zo štyroch hlavných častí, a to **lexikálnej analýzy** pre čítanie vstupu a **syntaktickej analýzy** rozdelenej do 2 častí. S ňou je úzko previazaná **sémantická analýza** a napokon je program zakončený samotným **generovaním kódu** v cieľovom jazyku. Naše vypracovanie neobsahuje žiadne zo zaregistrovaných rozšírení, no využíva niekoľko abstraktných dátových štruktúr. Tie sú spomenuté v závere, hneď za popisom práce v našom tíme či celkového prístupu k projektu.

2 Lexikálna analýza

Lexikálna analýza spočíva v scanneri, ktorý v priebehu programu na `get_token()` číta vstupný zdrojový kód a parseru ďalej posúva dôležité informácie v podobe štruktúry `token_t`.

Scanner

Scanner má za úlohu skenovať vstup, načítať a riadne roztriediť jednotlivé tokeny. Vstupný text je tak spracovaný komplexnou implementáciou konečného automatu bez epsilonových prechodov, viď príloha 8.1. Ten spracúva jednotlivé znaky na základe funkčnosti zdrojového jazyka. Pracuje na báze dynamicky alokovaného reťazca ukončeného po prečítaní bieleho znaku. Konečný automat takto vytesňuje komentáre a rozlišuje funkčné tokeny, teda konkrétne *kľúčové slová*, *dátové typy*, *identifikátory*, *celé* a *desatinné čísla*, *reťazce* zapísané v úvodzovkách, a jednotlivé funkčné *symboly* vrátane všetkých podporovaných *operátorov*, *zátvoriek*, *znaku priradenia*, *znaku „:“* využívaného v deklarácii funkcií, a napokon znaku *EOF* pre koniec vstupu.

Štruktúra tokenu

Token je základná jednotka zostavená scannerom a následne je využívaná naprieč celým programom. Pozostáva z premyslenej štruktúry `token_t` s prvkami „type“, „spec“, „attribute“ a „line“.
Prvé dva sú napĺňané vopred definovanými číselnými hodnotami pre rozlíšenie funkčnosti jednotlivých znakov (napr. token môže byť typu SYMBOL a špecifikácie PLUS). Tu je *typ* dôležitý pre hlavný parser a *špecifikácia* je obzvlášť využívaná v precedenčnom parseri. Prvok „attribute“ nesie hlavnú informáciu, a to obsah samotného tokenu. Pre presnejši výpis chyby a jednoduchšie debugovanie je zahrnutý prvok „line“ s informáciou pozície tokenu vrámci zdrojového kódu.

3 Syntaktická analýza

Významná časť programu sa venuje analýze syntaxe vstupného programu. Tá je založená na parseri, resp. dvoch; hlavný parser je pre spracovanie vonkajšej štruktúry programu metódou zhora-dole („top-down“) a precedenčný parser je určený pre podrobné spracovanie jednotlivých výrazov metódou zdola-hore („bottom-up“). Tieto metódy rozparsujú vstup do formy abstraktne popisujúcej derivačný strom a jeho podstromy. Tento postup tak umožňuje priamy prístup pre kontrolu sémantiky a ďalej tvorí vhodný materiál pre generátor.

Hlavný parser - „top-down“

Pre základné parsovanie sme zvolili metódu rekurzívneho zostupu. Parser tak prechádza kód zhora nadol, a teda z vonkajšej štruktúry sa rekurzívne zahlbuje na báze pravidiel LL1 gramatiky s epsilon pravidlami, popísanej v prílohe 8.3. Parser tak prechádza z prostredia *global* do jednotlivých funkcií cez hlavičku a telo funkcie (prostredie *local*). Tie ďalej pozostávajú z niekoľkých zreťazených parametrov, prvkov (akými sú cyklus, vetvenie, priradenie) a jednotlivých identifikátorov či výrazov. Po prečítaní tokenu, ktorý značí častý výskyt výrazu, je zavolaný precedenčný parser, ktorého výsledok je skontrolovaný a spracovaný. Výsledok parsovania je ukladaný do stromovej abstraktnej dátovej štruktúry AST, ktorej ukazateľ je následne odovzdaný generátoru výsledného kódu.

Precedenčný parser - „bottom-up“

Precedenčná analýza sa venuje výrazom z podmienok, z priradenia alebo tým navrátených z funkcie kľúčovým slovom „return“. Obsah výrazu je ukladaný do abstraktnej dátovej štruktúry DLL. V prípade, že výskyt výrazu bol potvrdený, zoznam je odovzdaný precedenčnému parseru ako páska s tokenmi ukončená znakom „$“. Táto časť pásku precedenčne spracúva na základe precedenčnej tabuľky určujúcej priority (viď v prílohe 8.2). A výrazy zostavené z relačných, aritmetických či iných operátorov sú postupne redukované podľa pravidiel precedenčnej gramatiky využívajúc špeciálne zostavený zásobník. Výsledkom tohto parseru je rozparsovaný postfixový výraz, uložený v DLL, predaný do AST a následne spracovaný až v časti generátora.

4 Sémantická analýza

Táto časť úzko súvisí s tou predošlou, nakoľko už takto rozpracovaný zdrojový kód sa dá jednoducho kontrolovať i po sémantickej stránke.

V precedenčnej časti overuje kompatibilitu typov a delenie celočíselnou konštantou „0“ (a to aj v prípade, že táto konštanta vyplýva z aritmetických pravidiel, *napr x // (0+0), x / (y*0)*). V druhej časti, hlavný parser úzko pracuje s tabuľkou symbolov, do ktorej ukladá jednotlivé deklarácie funkcií i premenných, a ďalej tak vhodne rieši konflikty. V prípade chyby ukončí program odpovedajúcou návratovú hodnotu a spolu s riadkom výskytu vypíše chybovú hlášku na štandardný chybový výstup.

5 Generovanie výsledného kódu

Poslednú a najzásadnejšiu fázu programu tvorí generátor. Jeho vstup pozostáva z odkazu na dátovú štruktúru syntaktického stromu AST, ktorého obsah je už po syntaktickej a sémantickej analýze rozparsovaný a skontrolovaný na prítomnosť chýb, ktoré bolo doposiaľ možné zistiť. Generátor tak prechádza AST a na základe dostupných inštrukcií cieľového jazyka vytvára kód v IFJcode21.

6 Práca v tíme

Pre správnu činnosť programu bolo prvoradé si uvedomiť jeho komplexnosť a primerane ho rozdeliť na jednotlivé časti a úlohy pre členov tímu. Práca v tíme bola zpočiatku rozdelená len orientačne a až v priebehu boli úlohy podrobnejšie zadelené na základe ich náročnosti a času jednotlivých členov.

Rozdelenie práce

Novák David	líder tímu, tabuľka symbolov, AST štruktúra, generátor výstupneho kódu
Kozub Tadeáš	syntaktická a sémantická analýza (hlavný parser)
Klimecká Alena	konečný automat, lexikálna analýza, testovanie, kontrola sémantiky, Makefile
Bubáková Natália	návrh LL gramatiky, syntaktická a sémantická analýza (precedenčný parser), dokumentácia

Komunikácia

Správne fungovanie tímu zabezpečovala častá komunikácia a stretnutia na pravidelnej báze.
Ako komunikačný kanál sme si zvolili platformu Discord, kde sme vytvorili server dedikovaný tomuto projektu a niekoľko rôzne zameraných kanálov pre zefektívnenie komunikácie.
Stretnutia z počiatku prebiehali v týždňových intervaloch, a to online formou. No, v neskoršiom štádiu ich nahradili častejšie osobné stretnutia, a to najmä medzi jednotlivými členmi tímu, ktorí spolu na niečom pracovali.

7 Použité materiály a abstraktné dátové štruktúry

Na záver je vhodné spomenúť, že všetky informácie potrebné pre pochopenie a samotné implementovanie projektu sme čerpali z výuky a odporúčanej literatúry predmetu IFJ a jemu príbuzným predmetom. Veľkú časť algoritmov dátových štruktúr pokryla výuka a domáce úlohy z predmetu IAL, zatiaľčo tabuľku symbolov sme zostavili najmä vďaka predmetu IJC.

Zásobník

Zásobník precedenčnej analýzy je inšpirovaný predmetom IAL. Avšak základnu dátovú jednotku nahradil token (štruktúra `token_t`), a tiež bol obohatený o funkcie vkladania značenia za prvý terminál alebo funkcie pre získanie prvého terminálu pre účely precedenčného parsovania.

DLL

Pri štruktúre obojsmerne viazaného zoznamu sme čerpali z domácej úlohy IAL. Tá si informácie podáva tiež cez jednotku token, využíva sa pre komunikáciu medzi precedenčným parserom a hlavným parserom, prípadne generátorom.

AST

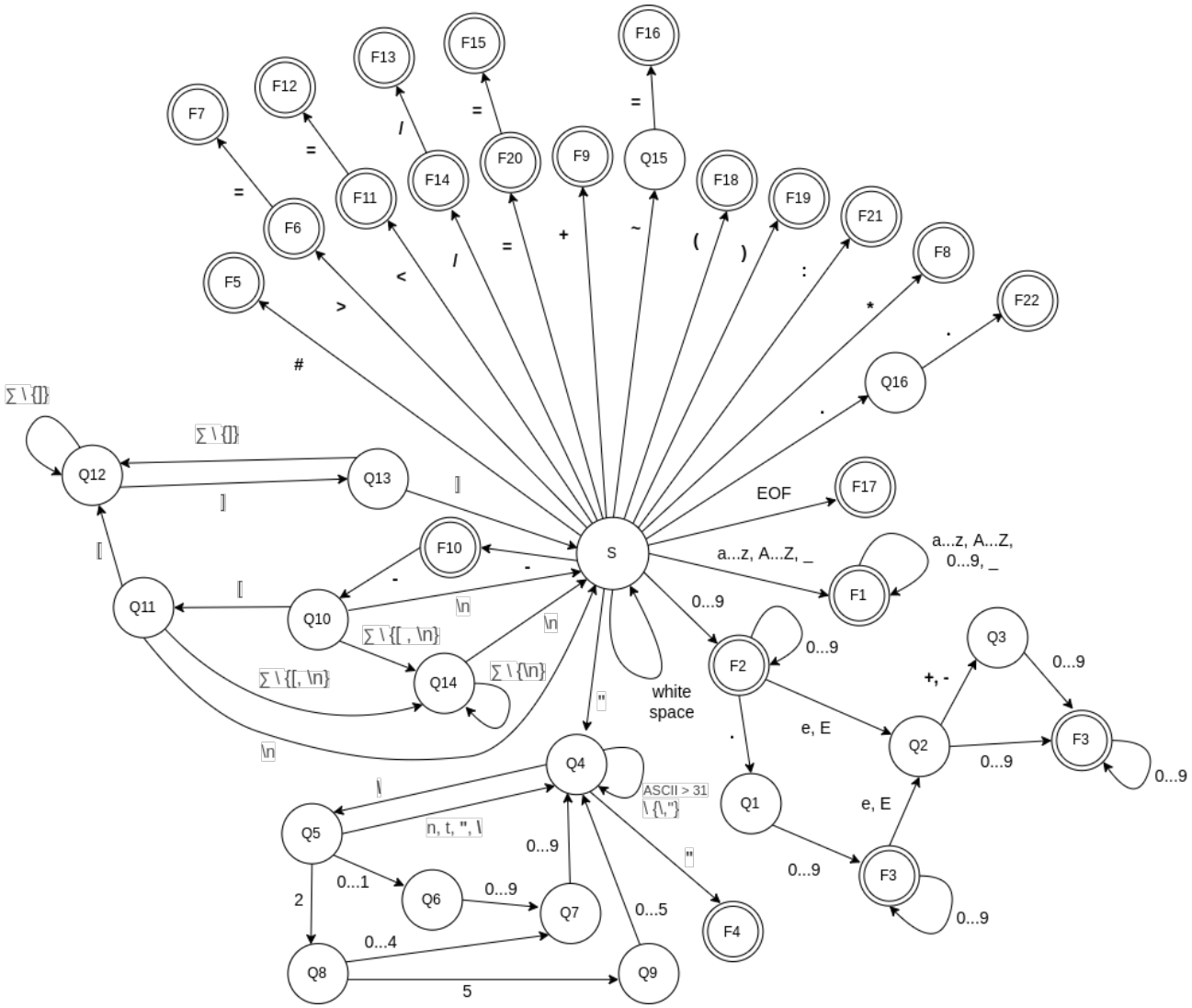
Abstraktný syntaktický strom zohráva kľúčovú úlohu pri zostavovaní abstrakcie derivačného stromu programu. Obsahuje viacero dátových jednotiek, tak aby obsiahli všetky potrebné typy vstupu predávaného generátoru.

Tabuľka s rozptýlenými položkami

Tabuľka symbolov je implementovaná ako štruktúra, ktorá má dva základné prvky. Prvý prvok je hashovacia tabuľka pre všetky funkcie, ktoré môžu byť iba globálne. Druhý prvok je zásobník hashovacích tabuliek, ktorý se využíva na premenné. Každý prvok zásobníku je jedna hashovacia tabuľka, ktorá simuluje vnorenie. Na jednotlivých indexoch tabulky sú lineárne zoznamy, v ktorých sa nachádzajú synonymá. Lineárne zoznamy sú tiež použité pre ukladanie typov parametrov a návratových hodnôt funkcií v hashovacej tabulke pre funkcie.

8 Prílohy

Konečný automat



Legenda:

F1	Keyword, Identifier, Datatype	F5	#	F14	/
F2	Integer	F6	>	F15	==
F3	Decimal	F7	>=	F16	~=
F4	String	F8	*	F17	EOF
		F9	+	F18	(
		F10	-	F19	)
		F11	<	F20	=
		F12	<=	F21	:
		F13	//	F22	..

Precedenčná tabuľka

	+	−	*	/	//	<	<=	>	>=	==	~=	..	#	*i*	(	)	$
+	>	>	<	<	<	>	>	>	>	>	>	_	<	<	<	>	>
−	>	>	<	<	<	>	>	>	>	>	>	_	<	<	<	>	>
*	>	>	>	>	>	>	>	>	>	>	>	_	<	<	<	>	>
/	>	>	>	>	>	>	>	>	>	>	>	_	<	<	<	>	>
//	>	>	>	>	>	>	>	>	>	>	>	_	<	<	<	>	>
<	<	<	<	<	<	>	>	>	>	>	>	_	<	<	<	>	>
<=	<	<	<	<	<	>	>	>	>	>	>	_	<	<	<	>	>
>	<	<	<	<	<	>	>	>	>	>	>	_	<	<	<	>	>
>=	<	<	<	<	<	>	>	>	>	>	>	_	<	<	<	>	>
==	<	<	<	<	<	>	>	>	>	>	>	_	<	<	<	>	>
~=	<	<	<	<	<	>	>	>	>	>	>	_	<	<	<	>	>
..	_	_	_	_	_	_	_	_	_	_	_	<	_	<	<	>	>
#	>	>	>	>	>	>	>	>	>	>	>	<	_	<	<	>	>
i	>	>	>	>	>	>	>	>	>	>	>	>	_	_	_	>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	_
)	>	>	>	>	>	>	>	>	>	>	>	>	_	_	_	>	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	_	#

Legenda:

- **=** aktuálny token z pásky sa pushne na vrchol zásobniku
- **<** za prvý terminál v zásobniku sa vloží MARK („<“) pre označenie začiatku pravej strany pravidla a pushne sa aktuálny token z pásky
- **>** na vrchu zásobníku sa nachádza pravá strana pravidla, tá sa zredukuje nahradením nonterminálom NONT („E“)
- **#** výraz bol úspešne spracovaný a ukončený
- **_** táto kombinácia je neočakávaná, nastáva syntaktická chyba

Precedenčná gramatika:

1: E → E + E
2: E → E - E
3: E → E * E
4: E → E / E
5: E → E // E
6: E → E < E
7: E → E <= E
8: E → E > E
9: E → E >= E
10: E → E == E
11: E → E ~= E
12: E → E .. E
13: E → # E
14: E → i
15: E → (E)

LL gramatika

(1) <program> → <prologue> <program_body>
(2) <prologue> → require "ifj21"
(3) <program_body> → <func_decl> <program_body>
(4) <program_body> → <func_def> <program_body>
(5) <program_body> → <func_call> <program_body>
(6) <program_body> → ε
(7) <func_decl> → global FUNC_ID : function (<type_list>) : <type_list>
(8) <type_list> → <type> <types>
(9) <type_list> → ε
(10) <types> → , <type> <types>
(11) <types> → ε
(12) <type> → number
(13) <type> → string
(14) <type> → integer
(15) <type> → nil
(16) <func_def> → function FUNC_ID (<param_list>) : <type_list> <func_body> end
(17) <param_list> → ID : <type> <params>
(18) <params> → , ID : <type> <params>
(19) <param_list> → ε
(20) <params> → ε
(21) <func_call> → FUNC_ID (<constant_list)
(22) <constant_list> → CONSTANT <constants>
(23) <constants> → , CONSTANT <constants>
(24) <constant_list> → ε
(25) <constants> → ε
(26) <func_body> → <element> <func_body>
(27) <func_body> → ε
(28) <element> → <if_element>
(29) <element> → <while_element>
(30) <element> → <return_element>
(31) <element> → <func_element>
(32) <element> → <decl_element>
(33) <element> → <assignment>
(34) <if_element → if EXPRESSION then <func_body> else <func_body> end
(35) <while_element> → while EXPRESSION do <func_body> end
(36) <return_element> → return <return_list>
(37) <return_list> → ε
(38) <return_list> → <item> <items>
(39) <item_list> → <item> <items>
(40) <items> → , <item> <items>
(41) <items> → ε
(42) <item> → <func_element>
(43) <item> → ID
(44) <item> → EXPRESSION
(45) <func_element> → FUNC_ID (<arg_list>)
(46) <arg_list> → <arg> <args>
(47) <arg_list> → ε
(48) <args> → , <arg> <args>
(49) <args> → ε
(50) <arg> → CONSTANT
(51) <arg> → ID
(52) <decl_element> → local ID : <type> <decl_assign>
(53) <decl_assign> → ε
(54) <decl_assign> → = <item>
(55) <assignment> → <L_assignment> = <R_assignment>
(56) <L_assignment> → ID <ids>
(57) <ids> → , ID <ids>
(58) <ids> → ε
(59) <R_assignment> → <item_list>