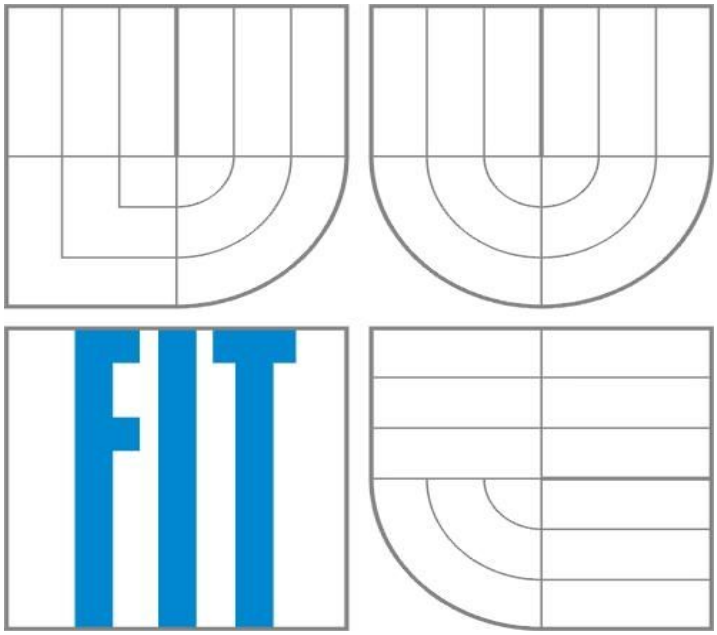
Vysoké učení technické v Brně

Fakulta informačných technologií



Dokumentácia k projektu do predmetu IFJ

Tým 032, varianta II

Nguyen Hoang Duong,	xnguye14,	33.33%	
Martin Nizner,	xnizne00,	33.33%	
Ján Gula,	xgulaj00,	33,33%	(vedúci)

Obsah

1. Úvod 2
2. Popis implementácie 2
2.1. Lexikálna analýza 2
2.2. Syntaktická analýza 2
2.3. Sémantická analýza 3
2.4. Generovanie inštrukcií a cieľového jazyka 3
2.5. Tabuľka symbolov 4
2.6. Vstavané funkcie 4
3. Práca v tímu 4
3.1. Rozdiel práce 4
3.2. Vývoj 5
4. Záver 5
5. Referencie 5
Konečný automat 6
LL- gramatika 7
Precedenčná tabuľka 9

1. Úvod

Hlavným účelom tejto dokumentácie je vysvetliť spôsob implementovania prekladača imperatívneho jazyka IFJ17. Popisuje našeho postupu riešenia projektu a vysvetluje jednotlivé kroky implementovania. Súčastou tejto dokumentácie je LL-gramatika, ktorá je jadrom našej syntaktickej analýzy, konečný automat pre lexikálnu analýzu a precedenčná tabuľka pre spracovanie výrazov.

2. Popis implementácie

Projekt bol implementovaný pomocou jazyka C a skladá sa z viacerých častí, ktoré sú na sebe závislé a ktoré spolu tvoria funkčný program.

2.1 Lexikálna analýza

Lexikálna analýza je vytvorená na základe princípu konečného automatu. Slúži na načítanie znakov zo štandardného vstupu a podľa ich vlastnosti ich transformuje na rôzne tokeny. Každý token reprezentuje 1 konkrétny znak alebo postupnosť znakov. V prípade, že lexikálna analýza nedokáže rozpoznať nejaký znak alebo postupnosť znakov, nahlási lexikálnu chybu a tým pádom sa program končí a vráti návratový kód 1. Ak lexikálna analýza prebiehala bez chýb, zachováva sa hodnota postupnosti znakov, ktorá prešla lexikálnou kontrolou a ako výstup bude názov tokenu, ktorý túto postupnosť reprezentuje.

2.2 Syntaktická analýza

Syntaktická analýza prijíma postupnosť tokenov od lexikálnej analýzy a zisťuje, či je daná syntaktická konštrukcia správna na základe vyhľadávania abstraktného syntaktického stromu. Ak sa žiaden AST(Abstract Syntactic Tree) nenašiel, syntaktická analýza ukonči prácu programu s návratovým kódom 2. Naša implementácia syntaktickej analýzy používa prístup zhora dolu (Top-Down approach) s využitím precedenčného syntaktického analyzátoru, ktorý slúži na vyhodnocovanie výrazov. Pokiaľ sa v zadanej postupnosti tokenov nenachádza žiaden výraz, syntaktická analýza prechádza siete príkazov CASE v snahe o nájdenie vetvy, ktorá by zodpovedala postupnosti tokenov. Pri objavení takejto vetvy AST vieme, že je daná konštrukcia jedným z platných pravidiel nášho jazyka a nenastala žiadna chyba. Každá väčšia vetva

syntaktického stromu má svoju funkciu. V prípade, že je postupnosťou tokenov rekurzívne volanie napríklad príkazu IF, nastane rekurzia aj v syntaktickej analýze kde sa vynáranie z jednotlivých funkcii spustí pri natrafení na END IF. Syntaktická analýza taktiež generuje príkazy pre interpreter a kontroluje návratové kódy lexikálnej analýzy pri pýtaní tokenov alebo sémantickej analýzy pri výrazoch.

2.3 Sémantická analýza

Sémantická analýta je volaná syntaktickou analýzou práve v týchto situáciach:

- pri analýze výrazov - kontroluje typovú kompatibilitu operandov pomocou pomocného zásobníku, ktorý obsahuje všetky operandy a operátory daného výrazu (vrátané zátvoriek) a má pre každý operand uložený jeho typ. Ak je operand premenná, kontroluje sa aj jej inicializácia pomocou tabuľky symbolov. Pri spracovaní výrazov sa najprv pomocou precedenčnej tabuľky porovná hodnota na vrchole zásobníku s hodnotou načítaného tokenu získaného pomocou lexikálnej analýzy. Následne sa vykoná dané pravidlo v precedenčnej tabuľke (uloženie tokenu na vrchol zásobníku spolu so znakom '<', uloženie tokenu na vrchol zásobníku bez žiadneho znaku - ide o pravidlo =, redukcia obsahu zásobníku - pravidlo '>'). Pri redukcií sa redukovaný obsah zásobníku odstráni a nahradí sa 1 nonterminálom na vrchole zásobníku. Spracovanie výrazu končí, keď sa v celom zásobníku nachádza práve 1 nonterminál a zároveň žiadne terminály.
- pri deklarácií premenných - kontroluje pomocou tabuľky symbolov, čí sa nejedná o redefiníciu premennej v danom rámci.
- pri deklarácií a definícii funkcie - kontroluje pomocou globálnej tabuľky symbolov, či sa nejedná o redeklaráciu, prípadne redefiníciu funkcie. Takisto kontroluje čí deklarácia funkcie nebola uvedená až za jej definíciou.

2.4 Generovanie inštrukcií a cieľového jazyka

Inštrukcie sú vygenerované pokiaľ nezlyhá kontrola syntaktickej a sémantickej analýzy. Každá inštrukcia je vygenerovaná ako trojadresný kód a je ukladaná do tzv. inštrukčného listu. Štruktúra inštrukčného kódu obsahuje následujúce zložky: typ inštrukcie, 3 adresy reprezentujúce operandy potrebné pre danú inštrukciu, typ každého operandu, či sa jedná o premennej alebo konštante, druh framu pre jednotlivého operandu, kde sa daný operand nachádza. Na konci syntaktickej a sémantickej analýzy prebieha proces generovania všetkých inštrukcií uložených v listu do cieľového jazyka IFJcode17

2.5 Tabuľka symbolov

Tabuľka symbolov je implementovaná pomocou hašovacej funkcie, tento spôsob nám umožní ukladať veľké množstvo položiek a pre vyhľadávanie položky v tabuľke je tento spôsob veľmi rýchly a efektívny. Behom práce na projektu sme postupne pridali do tabuľky rôzne informácie o každej položke, ktoré sú podľa nás dôležité a nezbytné pre proces implementácie. Každá položka v tabuľke symbolov obsahuje užívateľom dátové štruktúry, ktoré uchovávajú 2 typy informácií: 1 typ je pre premenných - názov premennej, typ premennej, jej deklarácia a inicializácia, druh rámca, kde sa táto premenná nachádza (GF, LF, TF), druhý je pre funkcie - rovnako obsahuje všetky informácie ako má premenná, avšak má navyše zoznam parametrov a argumentov a ich typy.

2.6 Vstavané funkcie

Všetky vstavané funkcie sú implementované pomocou inštrukcie jazyka IFJcode17 a ich názvy a parametre sú uložené v globálnom rámci (frame). Každá z týchto funkcií je označená pomocou návestia (label), ktoré má tvar %x kde x je názov danej funkcie (napr. %length). Tento label sa používa v prípade volania tejto funkcie pomocou inštrukcie *call <label>*.

3. Práca v týmu

3.1 Rozdiel práce

Projekt sme začali robiť hneď po zverejnení zadania. Rozdelili sme práce tak, aby každý mohol robiť to, čo mu najviac vyhovuje.
Nguyen Hoang Duong - lexikálna analýza, generovanie inštrukcií, vstavané funkcie.
Martin Nizner - lexikálna analýza, syntaktická analýza, sémantická analýza, vstavané funkcie, testovanie.
Ján Gula - lexikálna analýza, syntaktická analýza, tabuľka symbolov, testovanie.

3.2 Vývoj

Pre spracovania projektu sme sa rozhodli použiť webovú službu Github, ktorá nám umožní mať presný prehľad o aktuálnom stave projektu. Každý z nás mohol aktuálny zdrojový kód hocikedy pridávať a upravovať. Komunikácia v tímu prebieha u nás bez najmenšieho problému, kedže sa stretávame skoro každý deň. Pri stretnutí debatujeme o problémoch, ktoré nastali počas riešenia projektu a spolu navrhneme spôsob riešenia daných problémov.

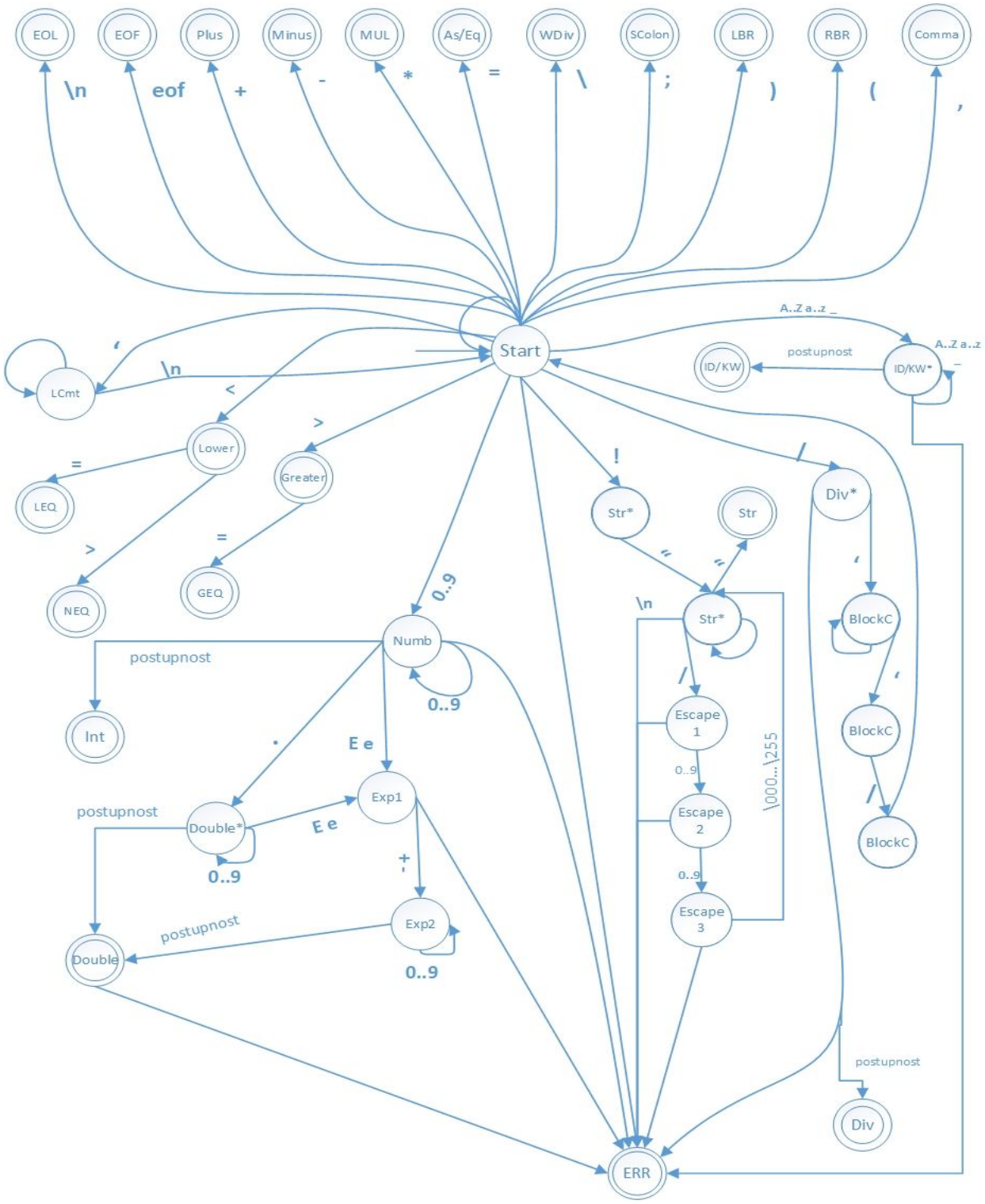
4. Záver

Tento projekt je z nášho pohľadu veľmi náročný avšak aj veľmi zaujímavý. Pomocou tohoto projektu sme sa naučili ako vlastne funguje prekladač a aké procesy prebiehajú v pozadí.

5. Referencie

[1] Prednášky predmetu IFJ

Diagram konečného automatu



Postupnost -> postupnost znakov + - * > < = EOF ; \ /) ‘ isspace

LL gramatika

Terminály: As, Declare, Dim, Do, Double, Else, End, Function, If, Input, Integer, Loop, Print, Return, Scope, String, Then, While ; + - * / ! = < > [ ] { } ( ) EOF e E ' /' '/ !“ **>= <= <>**

Non-terminály: <topLevelTrigger> <commandsCycle> <varDefinition> <ListOfParams> <printAnalysis> <expressionAnalysis> <endAnalysis> <idAnalysis>

<topLevelTrigger>

Declare function name(<ListOfParams>) as data_type

Function name(<ListOfParams> as data_type

Scope

<commandsCycle>

Dim name <varDefinition>

Do while(<expressionAnalysis>) <commandsCycle> loop

If(<expressionAnalysis>) Then EOL <commandsCycle>

Else <commandsCycle>

End <endAnalysis>

Print <printAnalysis>

Return <expressionAnalysis>

<varDefinition>

As data_type

= <expressionAnalysis> As data_type

<ListOfParams>

)

name as data_type<ListOfParams>

, name as data_type <ListOfParams>

<printAnalysis>

!“string“

var_name

func_name

<expressionAnalysis>

integer

string

double

id <idAnalysis>
<endAnalysis>
scope
function
if
<idAnalysis>
< id or constant
> id or constant
= id or constant
* id or constant
/ id or constant
+ id or constant
- id or constant
>= id or constant
<= id or constant
<> id or constant
!= id or constant
(<expressionAnalysis>

Precedenčná tabuľka

	/ *	\	+	-	log	)	(	str	i	$
/ *	>	>	>	>	>	>	<	X	<	>
\	<	>	>	>	>	>	<	X	<	>
+	<	<	>	>	>	>	<	<	<	>
-	<	<	>	>	>	>	<	X	<	>
log	<	<	<	<	>	>	<	<	<	>
)	>	>	>	>	>	>	X	X	X	>
(	<	<	<	<	<	=	<	<	<	X
str	>	X	X	X	<	>	X	X	X	>
i	>	>	>	>	>	>	X	X	X	>
$	<	<	<	<	<	X	<	<	<	X

log - <, >, =, >=, <=, <>
X - chybový stav