VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentácia k projektu:

Implementácia prekladača imperatívneho jazyka IFJ21

Tím 117, varianta I

Jiřina Frýbortová (xfrybo01) = ∞ %
Jana Kováčiková (xkovac59) = ∞ %
Alexander Rastislav Okrucký (xokruc00) = ∞ %
Patrik Skaloš (xskalo01) = ∞ % - vedúci tímu

1 Rozdelenie práce

Už od začiatku sme sa snažili na všetkom pracovať spolu. Naša práca začala pri spoločnom návrhu konečného automatu a celý skener sme tiež naprogramovali spoločne. Veľmi nám to pomohlo presne pochopiť, čo a ako je potrebujeme urobiť. Sme tiež presvedčení, že nám to pomohlo napísať menej chybový kód.

Ďalším krokom bola implementácia binárneho stromu - tu sa naša práca rozdelila a nepracovali sme na ňom všetci. Z binárneho stromu sme ďalej naprogramovali zásobník stromov - tabuľku symbolov. Parser bola jedna z najzložitejších častí a tak sme znovu zvolili spoločnú prácu - od návrhu gramatiky až po samotnú implementáciu a testovanie. Precedenčnú analýzu sme začali najprv robiť spolu, no z časových dôvodov ju mal nakoniec na starosti vedúci nášho tímu - Patrik Skaloš. Generátor mal na starosti Alexander Okrucký, no každý sme prispeli svojou časťou.

Rozdelenie práce na ktorej sme nerobili spoločne:

Jiřina Frýbortová - binárny strom, dokumentácia
Jana Kováčiková - binárny strom, dokumentácia, prekreslenie konečného automatu do finálnej podoby
Alexander Rastislav Okrucký - generátor cieľového kódu, testy
Patrik Skaloš - tabulka symbolov, precedenčná analýza, makefile, testy

Vyššie spomenuté časti projektu sme síce mali rozdelené, no navzájom sme si aj napriek tomu pomáhali a intenzívne o problémoch aj riešeniach komunikovali.

2 Organizácia

Dá sa povedať, že najdôležitejší bol verzovací nástroj Git. Súbory sme v ňom mali rozdelené do vetí podľa celkov, tak aby sa na nich dalo pracovať paralelne a nevznikali konflikty. Zároveň sme v ňom mali histórі všetkých súborov a tak sme sa kedykoľvek vedeli vrátiť k pôvodnej verzii.

Na komunikáciu sme využívali aplikáciu Discord v ktorej sme si vytvorili kanály na bežné písanie, termíny ale aj hlasový kanál na naše meetingy. Výhodou bola určite možnosť pripnutia správ aby sme mali dôležité veci vždy po ruke a nemuseli ich pracne hľadať v histórii konverzácie.

Ďalšou veľmi dôležitou aplikáciou bola Trello (to-do list). Vytvorili sme si stĺpce, ktoré značili či už je daná úloha hotová, pracuje sa na nej alebo je potrebné ju otestovať. Do stĺpcov sme následne pridávali úlohy a priebežne sme tento zoznam aktualizovali. K jednotlivým úlohám sa dali taktiež písať podúlohy, čo určite pomohlo v prehľadnosti a efektivite práce.

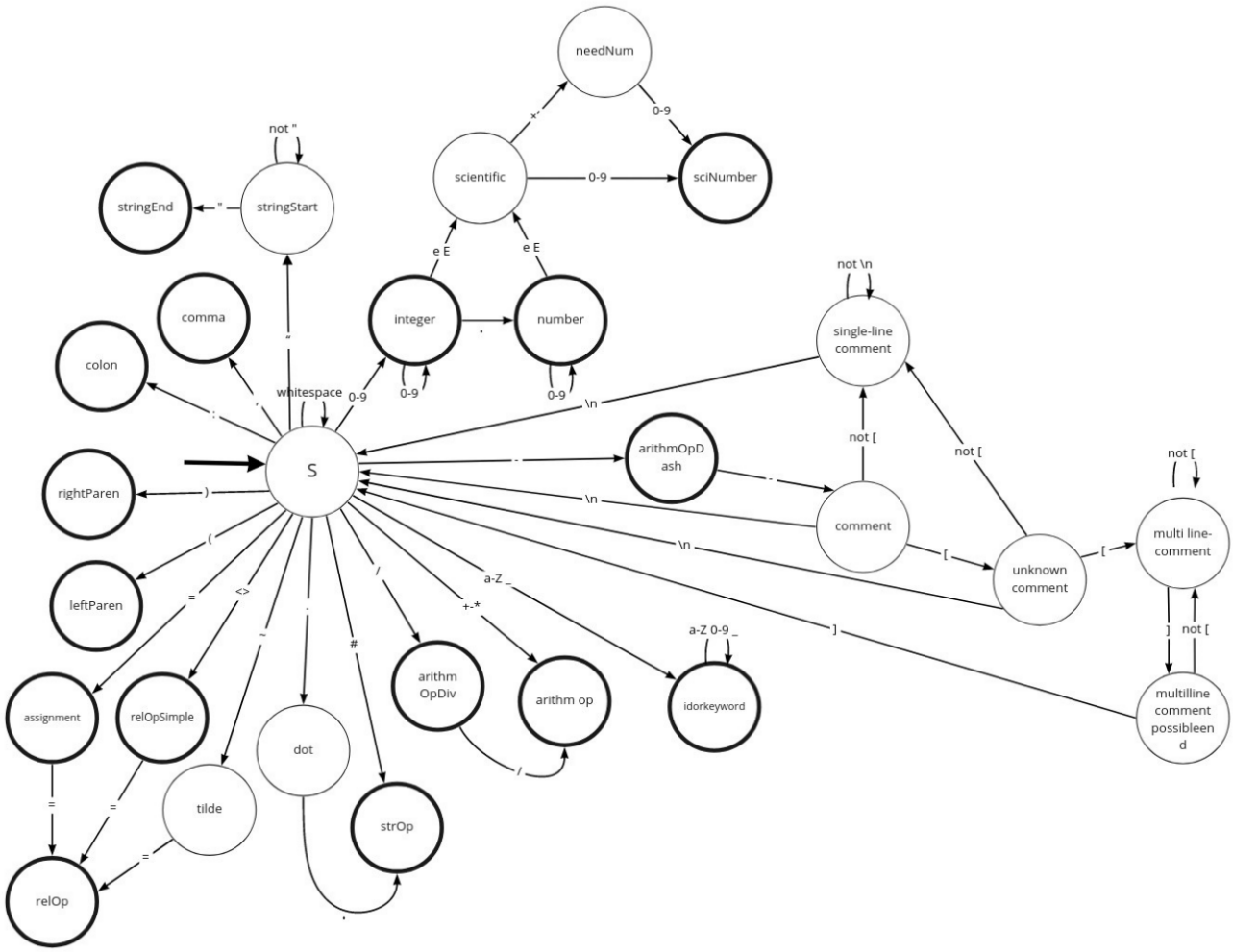
3 Realizácia

Syntaxou riadený preklad sme realizovali pomocou jednoprechodovej analýzy zdrojového kódu. Náš program nevytvára medzikód ale priamo generuje cieľový kód v jazyku IFJcode21. V prípade nájdenia chyby sa preklad kódu ihneď a bez zotavenia ukončí, na štandardný chybový výstup vypíšeme chybovú hlášku a približné číslo riadku, na ktorom k chybe došlo.

3.1 Lexikální analýza

3.1.1 Návrh konečného automatu

Počiatočný stav automatu, je kruh S. Šípky v konečnom automate značia, aký znak (množina znakov) je potrebný na prechod z predchádzajúceho do nového stavu. Končený stav v automate značíme hrubším zvýraznením namiesto dvoch kruhov.



Graf konečného automatu nájdete taktiež v prílohách, kde sa náchadza vo väčšom rozlíšení.

3.1.2 Skener

Podľa návrhu konečného automatu sme realizovali implementáciu skeneru. Skener náčítava postupne znaky až kým nenačíta celý token ktorý predá parseru. Ak pri načítaní skener načíta o znak viac, vrátime ho na stdin. Scanner umožnuje funkciu stash, čo znamená že token vieme vrátiť.

Niektoré stavy, ktoré sú konečné a nedá sa z nich prejsť do iného stavu v skaneri neimplementujeme a na miesto prechodu do týchto stavov vrátime aktuálny token.

V prípade EOF vraciame prázdny token a komentáre aj biele znaky sú ignorované (sú samozrejme zaznamenané ako oddelenie tokenov).

Keď skener narazí na chybu, vráti parseru adekvátny chybový kód (podľa nájdenéj chyby lexikálny alebo interný) a prázdny token.

V skeneri neriešime, či sa jedná o identifikátor alebo kľúčové slovo – považujeme ich za rovnaký typ tokenu. Dá sa povedať, že so skeneru vraciame len tieto typy tokenov: meno (identifikátor, kľúčové slovo), literál, operátor a špeciálne znaky (čiarka, dvojbodka).

Samotná implementácia skeneru je nasledovná: Po zavolaní funkcia scanner() číta znaky zo štandardného vstupu a podľa pravidiel konečného automatu a aktuálneho znaku mení stavy. Pri načítaní zakázaného znaku vráti chybu, pri neočakávanom znaku (ak neexistuje pravidlo na prechod) tento znak vráti na štandardný vstup a vráti token volajúcej funkcii. Štruktúra tokenu obsahuje informáciu o type tokenu a jeho obsahu (meno identifikátoru, literál, ...).

3.1.3 Tabuľka symbolov

Tabuľku symbolov sme implementovali pomocou binárneho vyhľadávacieho stromu, čo však nestačilo. Vo výsledku ju tvorí zásobník binárnych vyhľadávacích stromov (ďalej BST) – každý prvok zásobníku obsahuje ukazateľ na jeden BST (jeho koreň), hĺbku zanorenia BST a ukazateľ na nasledujúci prvok. Každý prvok BST klasicky obsahuje kľúč, ukazatele na ľavý a pravý prvok a dáta. Štruktúra pre tieto dáta obsahuje:

- **string**: meno identifikátoru vo vygenerovanom kóde
- **bool**: značí či je identifikátor premennou alebo funkciou
- Dáta dôležité, ak je identifikátor **premennou**:
  - **integer**: enumerácia dátového typu
- Dáta dôležité, ak je identifikátor **funkciou**:
  - **bool**: značí, či bola funkcia deklarovaná
  - **bool**: značí, či bola funkcia definovaná
  - **pole typu integer**: enumerácie dátových typov parametrov
  - **pole typu string**: mená parametrov
  - **pole typu integer**: enumerácie dátových typov návratových hodnôt

Nad tabuľkou symbolov máme implementované tieto operácie:

- **STInit**: inicializácia
- **STPush**: pridanie nového (prázdneho) BST na vrchol zásobníku
- **STPop**: odstránenie BST z vrcholu zásobníku
- **STInsert**: vloženie nového prvku do BST na vrchole zásobníku – dáta sú inicializované na nepovolené hodnoty
- **STFind**: vráti prvok BST so zhodným vyhľadávacím kľúčom
- Pre nasledujúce operácie sú implementované dve funkcie – na nastavenie (**STSet**) a čítanie (**STGet**):

  - **IsVariable**: pravdivostná hodnota, či je identifikátor premennou (ak nie je pravda, je funkciou)
  - **VarDataType**: dátový typ premennej
  - **FnDefined**: pravdivostná hodnota, či bola funkcia definovaná
  - **FnDeclared**: pravdivostná hodnota, či bola funkcia deklarovaná
  - **Name**: meno identifikátora v generovanom kóde
- Pre nasledujúce operácie s nafukovacími poľami sú taktiež dve funkcie – na pridanie na koniec poľa (**STAppend**) a čítanie (**STGet**) z poľa:
  - **ParamType**: dátový typ parametra funkcie
  - **ParamName**: meno parametra funkcie
  - **RetType**: dátový typ návratovej hodnoty funkcie
- **STGetDepth**: vráti číslo reprezentujúce zanorenie BST na vrchole zásobníku
- **STFindUndefinedFunctions**: hľadá, či je v BST na vrchole zásobníku funkcia, ktorá bola deklarovaná ale nebola definovaná

3.2 Syntaktická analýza

3.2.1 LL tabuľka

	eps	id	require	if	global	local	function	return	while	type	assign	colon	comma	(	"ifj21"	expr
<start>			1													
<req>															2	
<codeBody>	4	7			6		5									
<fnCall>														9		
<fnCallArgList>	10	11														
<nextFnCallArg>	12												13			
<fnCallArg>		14								15						
<stat>	17	18		20		19		22	21							
<statWithId>							25				24	23				
<nextAssign>																
<fnDefinitionParam TypeList>	29	30														
<nextFnDefinition ParamType>	31											32				
<retArgList>	34															35
<retNextArg>	36												37			
<fnDeclaration ParamTypeList>	39									40						
<nextFnDeclaration ParamType>	41												42			
<fnRetTypeList>	44											45				
<nextRetType>	46												57			
<newIdAssign>	49										50					

3.2.2 Bezkontextová gramatika

01. <start> → require <req><codeBody>
02. <req> → "ifj21"

04. <codeBody> → eps

05. <codeBody> → function [id] (<fnDefinitionParamTypeList>) <fnRetTypeList><stat>end <codeBody>
06. <codeBody> → global [id] : function (<fnDeclarationParamTypeList>) <fnRetTypeList><codeBody>
07. <codeBody> → [id] <fnCall><codeBody>

09. <fnCall> → (<fnCallArgList>)
10. <fnCallArgList> → eps
11. <fnCallArgList> → <fnCallArg><nextFnCallArg>
12. <nextFnCallArg> → eps
13. <nextFnCallArg> → , <fnCallArg><nextFnCallArg>
14. <fnCallArg> → [id]
15. <fnCallArg> → [literal]

17. <stat> → eps
18. <stat> → [id] <statWithId><stat>
19. <stat> → local [id] : [type] <newIdAssign><stat>
20. <stat> → if <expr>then <stat>else <stat>end <stat>
21. <stat> → while <expr>do <stat>end <stat>
22. <stat> → return <retArgList><stat>
23. <statWithId> → , [id] <nextAssign><expr>
24. <statWithId> → = <expr>
25. <statWithId> → <fnCall>
26. <nextAssign> → , [id] <nextAssign><expr>
27. <nextAssign> → =

29. <fnDefinitionParamTypeList> → eps
30. <fnDefinitionParamTypeList> → [id] : [type] <nextFnDefinitionParamType>
31. <nextFnDefinitionParamType> → eps
32. <nextFnDefinitionParamType> → , [id] : [type] <nextFnDefinitionParamType>

34. <retArgList> → eps
35. <retArgList> → <expr><retNextArg>
36. <retNextArg> → eps
37. <retNextArg> → , <expr><retNextArg>

39. <fnDeclarationParamTypeList> → eps
40. <fnDeclarationParamTypeList> → [type] <nextFnDeclarationParamType>
41. <nextFnDeclarationParamType> → eps
42. <nextFnDeclarationParamType> → , [type] <nextFnDeclarationParamType>

44. <fnRetTypeList> → eps
45. <fnRetTypeList> → : [type] <nextRetType>
46. <nextRetType> → eps
47. <nextRetType> → , [type] <nextRetType>
49. <newIdAssign> → eps
50. <newIdAssign> → = <expr>

3.2.3 Precedenčná tabuľka

	#	*	/	//	+	-	..	<	<=	>	>=	==	~=	(	)	id	$
#		>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
*	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
//	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
-	<	<	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
..	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
==	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
~=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>	>	>	>		>		>
id	>	>	>	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<		<	

3.2.4 Parser

Pri parseri sme zvolili rekurzívny zostup. Pri spracovaní výrazu voláme precedenčnú analýzu.

Redefiníciu premenných riešime tak, že sa všetky definície ukladajú do bufferu a ak sa tam už raz daná funkcia nachádza, neuloží sa znovu. Premenné deklarujeme až na konci funkcie. Pred telom funkcie skáčeme na deklarácie premenných takže sa v skutočnosti vygenerujú ako prvé.

- že token předáváme jako parametr PA

3.2.5 Precedenčná analýza

Precedenčná analýza je implementovaná algoritmom ukázaným na prednáške predmetu IFJ a pomocou precedenčnej tabuľky. Pracuje s pomocnou dátovou štruktúrou – zásobníkom, ktorý obsahuje symboly (symboly sú v tomto kontexte chápané ako identifikátory – operandy, operácie a podobne).

Algoritmus pracuje v cykle, v ktorom si najprv získa nový token, analyzuje ho (získa o ňom všetky potrebné dáta) a prevedie ho na symbol. Počas chodu prevádza statické kontroly (dátových typov, delenia nulou, ...), sleduje celkovú správnosť vstupného výrazu a generuje kód v jazyku IFJcode21, ktorého behom sa má vyhodnotiť výraz na vstupe precedenčnej analýzy. Volajúcej funkcii nakoniec vráti meno premennej, v ktorej sa bude nachádzať výsledok výrazu (v jazyku IFJcode21) a jeho dátový typ.

Symbol, podľa jeho typu, môže nadobúdať nejaké (alebo všetky) z týchto dát:

- **type**: typ symbolu (výraz, identifikátor, operátor, ...)

- **op**: operand precedenčnej tabuľky (identifikátor, delenie, násobenie, ...)
- **isId**: na rozlíšenie, či je symbol identifikátorom alebo literálom
- **dataType**: dátový typ
- **data**: meno identifikátoru alebo obsah literálu
- **isZero**: na ošetrenie chyby delenia nulou (len staticky)

Podľa aktuálneho symbolu na vstupe a aktuálne najvyššieho terminálneho symbolu na zásobníku sa rozhodne o nasledujúcom kroku. Ak je nasledujúcim krokom redukovanie, volajú sa *funkcie pravidiel*, ktoré skúšajú aplikovať pravidlá na aktuálne symboly na zásobníku. Ak funkcia pravidlo na zredukovanie symbolov nemá, vráti hodnotu -1. Ak ktorákoľvek z týchto funkcii vráti hodnotu 0, pravidlo redukovania bolo nájdené a výraz úspešne zredukovaný. Hodnota vyššia ako 0 značí, že nastala chyba a precedenčná analýza tento chybový kód propaguje syntaktickej analýze. Funkcie pravidiel (redukujú na výraz):

- **iRul**e: zredukovanie identifikátoru
- **strLenRule**: zredukovanie unárneho operátora **#** a identifikátoru pred ním
- **bracketsRule**: zredukovanie výrazu v zátvorkách
- **arithmeticOperatorsRule**: zredukovanie binárneho aritmetického operátora s výrazom na oboch stranách
- **relationalOperatorsRule**: zredukovanie binárneho relačného operátora s výrazom na oboch stranách

Na úspešné a bezchybné riadenie algoritmu sú využité tiež nasledujúce premenné typu bool, ktoré značia:

- **getNewToken**: určuje, či sa má pred ďalším krokom analýzy prijať nový token
- **exprCanEnd**: určuje, či môže analýza bez chyby skončiť počas nasledujúceho kroku
- **exprEnd**: určuje, či už boli prijaté všetky tokeny potrebné na úspešné dokončenie analýzy

3.3 Generovanie kódu

Pri generácii kódu nevyužívame medzikód. Priamo z parseru voláme funkcie na generovanie cieľového kódu IFJcode21 na štandartný výstup.

4 Súbory

- assignment.c - implementácia viacnásobného priradenia
- built_in_functions.c - makrá vstavaných funkcií na generovanie kódu
- char_buffer.c - nafukovacie pole znakov

- generator.c - generovanie kódu
- int_buffer.c - nafukovacie pole integerov
- main.c - inicializácia tabulky symbolov, vkladanie vstavaných funkcií do tabuľky symbolov, volanie parseru
- misc.c - pomocné funkcie, ktoré sa nedali špecificky zaradiť do iných súborov
- parser.c - implementácia parseru
- precedence_analysis.c - spracovávanie výrazov
- scanner.c - načítanie znakov zo vstupu
- string_buffer.c - nafukovacie pole stringov
- symbol_stack.c - zásobníková štruktúra implementovaná ako lineárny zoznam
- symtable.c - operácie s tabuľkou symbolov
- symtable_stack.c - zásobník pre prácu s tabuľkou symbolov
- symtable_tree.c - implementácia binárneho vyhľadávacieho stromu
- token.c - operácie s tokenmi

V hlavičkových súboroch sú obecne len zadeklarované funkcie a voláme z nich iné súbory, súbor misc.h by sme však chceli šepciálne spomenúť, nakoľko v ňom máme definované makrá, ktoré využívame v celom projekte a veľmi nám uľahčili našu prácu.

5 Prílohy

5.1 Návrh konečného automatu