Vysoké učení technické v Brně
Fakulta informačních technologií

Formální jazyky a překladače

Projekt IFJ
IFJcode21
Tým 51, varianta I

Králik Dalibor (xkrali20) (vedoucí) - 25%
Sehnoutek Patrik (xsehno01) - 25%
Dovhalenko Dmytro (xdovha00) - 25%
Procházka Ivo (xproch0h) - 25%

8. prosince 2021

Obsah

1 Úvod **2**

2 Návrh a implementace **2**

3 Lexikální analýza **2**

4 Syntaktická analýza **2**

5 Precedenční syntaktická analýza pro zpracování výrazů **2**

6 Zásobník pro precedenční syntaktickou analýzu **3**

7 Generování mezikódu **3**
7.1 Výrazy . . . 3
7.1.1 Instrukce CONCAT a STRLEN . . . 3
7.2 Proměnné . . . 3
7.2.1 Deklarace a inicializace . . . 3
7.2.2 Přiřazení hodnot proměnným . . . 3
7.3 Podmínky . . . 3
7.4 Návěští konstrukcí if a while . . . 3
7.4.1 While . . . 3
7.4.2 If . . . 4
7.5 Funkce . . . 4
7.5.1 Návěští . . . 4
7.5.2 Volání . . . 4
7.5.3 Definice . . . 4

8 Tabulka symbolů **4**

9 Rozdelení práce a práce v týmu **4**
9.1 Komunikace . . . 4
9.2 Verzování projektu . . . 4
9.3 Rozdělení práce . . . 4

1 Úvod

Cílem projektu bylo vytvořit překladač, který načítá zdrojový kód jazyka IFJ21, který je podmnožinou jazyka Teal. Překladač má tento zdrojový kód načíst, spracovat a vygenerovat mezikód IFJcode21 pro dopředu připravený interpret, který má tento mezikód vykonat.

2 Návrh a implementace

Překladač se skládá z více částí - lexikální analýza, syntaktická analýza, precedenční analýza, tabulka symbolů a generování kódu. Toto rozložení bylo potřebné pro správné vypracování projektu a hlavně pro správnou práci a komunikaci v týmu. Umožnilo nám to možnost paralelního programování a dokázali jsme tak naplno využít každého člena týmu. Nakonec se tyto části propojili a vytvořili už hotový překladač.

3 Lexikální analýza

První části tvorby kompilátoru byla implementace lexikální analýzy. Základní částí lexikální analýzy je funkce read_token, která ze standardního vstupu čte znak po znaku a následně převádí na strukturu Token. Struktura Token se skládá ze dvou částí, name – typ tokenu a value – atributy tokenu. Typy jsou identifikátor, klíčové slovo, celočíselný a desetinný literál a všechny aritmetické, relační a řetězcové operátory a speciální znak konce řádku EOF. Funkce read_token implementuje deterministický konečný automat (DKA). DKA jsme si předem navrhli prostřednictvím diagramu číslo 1. Následně jsme tenhle diagram převedli do kódu v jazyce C. V kódu je DKA implementován pomocí while cyklu a if-else větvení, přičemž každé větvení označuje právě jeden stav DKA.Pokud načítaný znak nevyhovuje žádné z podmínek v daném stavu a stav není ukončující, je program ukončený chybovým hlášením 1. Pokud však načítaný znak nevyhovuje žádné z podmínek v daném stavu, ale stav je ukončující, tak funkce read_token využije funkci ungetc na vrácení znaku na standartní vstup a token pomocí návratové hodnoty pošle do syntaktické analýzy.Identifikátory, klíčová slova, řetězcové, celočíselné i desetinné literály jsou ukládané znak po znaku do proměnné name v struktuře Token, která je implementovaná prostřednictvím dynamického pole znaků. U identifikátoru a klíčových slov se před posláním tokenu syntaktické analýze nejdřív zkontroluje, pomocí seznamu klíčových slov, či jde o klíčové slovo nebo ne.

4 Syntaktická analýza

Syntaktická analýza je založená na LL-gramatice, kterou můžete vidět na obrázku č. 2 a v metodě rekurzivního sestupu. Syntaktická analýza zpracovavala na základě LL-gramatiky tokeny, které si sama vyžádala pomocí funkce read_token(Token *token). Následně tento token zpracovala a řídila sa pravidli gramatiky. Pro každý neterminál v LL-gramatice je vytvořená samostatná funkce, v které sa řeší gramatická pravidlá spojená s tímto neterminálem. Při implementaci jsme používali datovou strukturu Data_t, do které jsme ukládali všechny potřebná data na komunikaci s různými částmi programu jako je například precedenční analýza nebo generování kódu IFJcode21. V této datové struktuře jsme předávali například ukazatel na token, ukazatel na vrchol framu tabulky symbolů, datové typy pro typové kontroly a podobně. Celá práce syntaktického analyzátoru je založená na metodě syntaxí řízeného překladu. Syntaktická analýza nevytváří abstraktní syntaktický strom, ale přímo využíva zásobník rekurzívního sestupu na sématické kontroly – typové kontroly, správny počet parametrů funkce při volání, správny počet priřazovaných hodnot při vícenásobném přiřazení a pod., a na generovaní výsledného kódu.

5 Precedenční syntaktická analýza pro zpracování výrazů

Precedenční analýza je volána syntaktickou analýzou shora dolů při výskytu odpovídajícího pravidla v LL gramatice. Je implementována v souboru expressions.c, a její rozhraní je v souboru expressions.h. Zpracovávání výrazů se provádí na základě precedenční tabulky, obrázek č. 4. Vzhledem k tomu, že některé operátory, jako + -, * / // a relační operátory mají stejnou asociativitu a prioritu, je možné je zpracovávat stejným způsobem a zjednodušit precedenční tabulku. Sloupec a řádek **i** symbolizuje identifikátor, číslo nebo řetězec. Výraz může obsahovat všechny symboly z precedenční tabulky, mezi které patří operátory, literály (v tabulce označeny za **i**) a závorky. Tyto symboly jsou terminály. Symbol $ reprezentuje symboly, které výraz obsahovat nemůže. Sloupce

tabulky označuji symbol ve vstupním tokenu a řádek symbol na vrcholu pomocného zásobníku. Každá buňka precedenční tabulky reprezentuje kombinace symbolů na vrcholu zásobníku a symbolu ve vstupním tokenu na základě které provádějí odpovídající operace. Pro znak z tabulky < provádí funkce do_shift, která vloží na vrchol zásobníku terminál ze vstupního tokenu a zavolá funkce read_token, která načte sledující token. Pro znak > podle existujících pravidel provede redukce odpovídajícího pravidlo počtu položek ze zásobníku pomocí funkce do_reduce. Během provádění redukce se otestuje sémantika operandů, zavolá se generování kodu a provede přetypování, jestli je to potřebné. Pro znak = zavolá se funkce do_equal, která provede redukci podle pravidla (E)->E a zavolá načtení následujícího tokenu. To opakujeme dokud vstupní symboly můžou být součásti výrazů a podle precedenční tabulky můžeme provést jednu z operací. Precedenční analýza je úspěšná pokud na vrcholu zásobníku zůstal jeden výsledný neterminal a ve vstupním tokenu je symbol, který nemůže být součásti výrazu, s výjímkou unikátních případů, jinak končí neúspěšně.

6 Zásobník pro precedenční syntaktickou analýzu

Při provádění precedenční analýzy se používá zásobník implementovány v souboru expressions_stack.c a jeho rozhrání se nachází v souboru expressions_stack.h. Struktura položky zásobníku obsahuje symbol, který se vkládá na zásobník z tokenu, datový typ tohoto symbolu a ukazatel na další položku. Zásobník má implementované základní operace inicializace zásobníku init_stack, vložení položky push, odstranění položky pop, načtení vrchní položky top, kontrola na prázdný zásobník is_empty a zrušení zásobníku destroy. Pro účely precedenční analýzy byly implementovaný další funkce jako top_type pro zjištění typu položky na vrcholu zásobníku, změnu tohoto typu change_top_type, načtení položky za vrcholem top1 a zjištění její typu top1_type.

7 Generování mezikódu

7.1 Výrazy

Pro generování aritmetických, relačních a konverzních instrukcí využíváme výhradně jejich zásobníkové varianty. Dále v generování výrazů využíváme instrukce PUSHS, pro ukládání konstant a hodnot proměnných na zásobník.

7.1.1 Instrukce CONCAT a STRLEN

Instrukce, které nemají zásobníkovou variantu, jmenovitě CONCAT a STRLEN, jsou de facto převedeny na zásobníkové varianty. Instrukce vyžadují push dat na zásobník ve správném pořadí. Při generování jsou vygenerovány také instrukce POPS, které uloží data do pomocných proměnných GF@T-Nsymb, se kterými daná instrukce dále pracuje. Výsledek akce se uloží do pomocné proměnné GF@T-Nvar a pushne se na zásobník.

7.2 Proměnné

7.2.1 Deklarace a inicializace

Při definování proměnných je proměnným automaticky přiřazena hodnota nil@nil.

7.2.2 Přiřazení hodnot proměnným

Přiřazení hodnot proměnným probíhá vždy prostřednictvím instrukce POPS. Část kódu zodpovědná za generování výrazů ukládá hodnoty na zásobník a POPS je přiřazuje proměnným.

7.3 Podmínky

Pro uložení hodnot na levé a pravé straně podmínky jsou využívány pomocné proměnné LF@T-Nl a LF@T-Nr, kde N značí číslo unikatní k dané podmínce.

7.4 Návěští konstrukcí if a while

7.4.1 While

Pro každé while jsou vygenerována 2 návěští, while_N a end_while_N, kde N je unikatní číslo k danému while. Návěští While_N slouží pro opakování ve smyčce a návěští end_while_N pro vyskočení ze smyčky.

7.4.2 If

Pro každé if jsou vygenerována 2 návěští, else_N a end_if_N, kde N je unikatní číslo k danému if. Návěští else_N slouží pro vstup do else větve a návěští end_if_N slouží pro skok na konec if.

7.5 Funkce

7.5.1 Návěští

Návěští funkce je totožné s jménem funkce.

7.5.2 Volání

Před voláním funkce nejdříve vytvoříme nový dočasný rámec. Vytvoříme si proměnné arg, do kterých vložíme hodnoty, které předáváme dané funkci. Až poté voláme funkci instrukcí CALL.

7.5.3 Definice

U definic funkcí nejprve dochází k pushnutí dočasného rámce na zásobník. Následně přiřadíme parametrům jejich předané hodnoty. Definici ukončujeme generováním instrukcí POPFRAME a RETURN.

8 Tabulka symbolů

Tabulku symbolů jsme navrhli jako lehce modifikovaný jednosměrně provázaný seznam, obsahující jak ukazatel na první prvek seznamu, tak i ukazatel na poslední prvek seznamu. Prvky tohoto seznamu dále nesou, kromě dalších potřebných informací, binární strom, ve kterém jsou uloženy informace o proměnných a funkcích kódu IFJ21. Nad touto strukturou jsou dále vytvořeny různé funkce pro přístup, modifikaci a odstraňování dat.

9 Rozdelení práce a práce v týmu

9.1 Komunikace

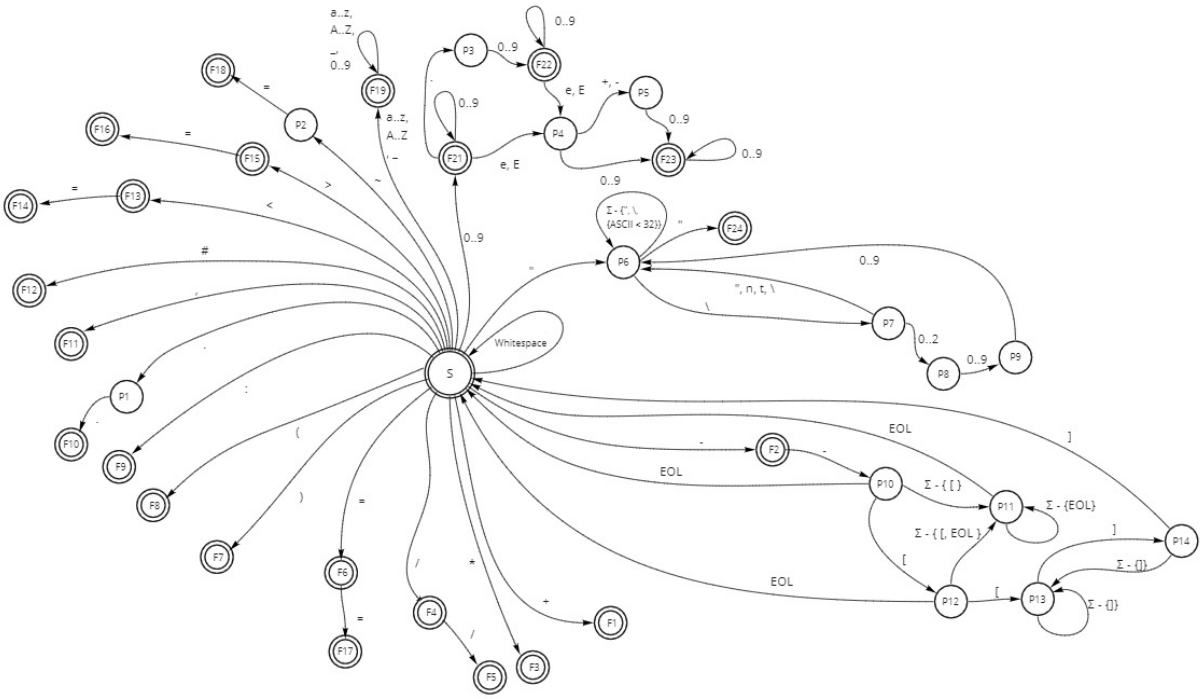
Jako komunikační kanál jsme používali Discord. Založili jsme si vlastní server, kde jsme se pravidelně setkávali, komunikovali a sdíleli potřebné informace.

9.2 Verzování projektu

K verzování zdrojových kódů jsme používali GitHub, na ktorém jsme si vytvořili repozitář a v něm jsme pracovali.

9.3 Rozdělení práce

Každý člen týmu splnil svoje povinnosti a pracoval přesne na tom, na čem jsme se dohodli. Dalibor Králik pracoval na rekurzívním sestupu a syntaxí řízeném překladě (sémantické kontroly a navázání funkcí generovaní kódu), Patrik Sehnoutek pracoval na lexikální analýze, makefile, testech a spolupodílel se na tvorbě generovaní kódu. Dmytro Dovhalenko pracoval na precedenční analýze a na syntaxí řízeném překladě v precedenční analýze (sémantické kontroly a správné dosazení funkcí generovaní kódu) a Ivo Procházka pracoval na tabulce symbolů (symtable) a spolupodílel sa na tvorbě generovaní kódu. Spoločně jsme se střetávali online i offline, spoločně jsme navrhovali stavový automat lexikální analýzy, precedenční tabulku, LL-gramatiku, datové struktury potřebné pro tabulku symbolů, precedenční a syntaktickou analýzu. Jelikož jsme každý na projektu pracovali se stejným úsilím a časovou dotáciou, dohodli jsme se, že každý z nás si zaslouží stejný % podíl na projektu a to 25% každý.



- F1 Plus
- F2 Minus
- F3 Multiplicate
- F4 Divide
- F5 Intiger divide
- F6 Equall
- F7 Left bracket
- F8 Right bracket
- F9 Colon
- F10 Concatenation
- F11 Comma
- F12 Length of string
- F13 Less
- F14 Less or equall
- F15 More
- F16 More or equall
- F17 Is equal to
- F18 Does not equall
- F19 Identifier or keyword
- F20 Single line comment
- F21 Intiger literal
- F22 Decimal literal
- F23 Intiger or decimal literal with exponent
- F24 String literal

- P1 Single dot
- P2 Tilda
- P3 Decimal dot
- P4 Exponent
- P5 Optional plus or minus
- P6 String start
- P7 Escape character
- P8 First ASCII number
- P9 Second ASCII number
- P10 Initialisation of comment
- P11 Single-line comment
- P12 First bracket of multi-line comment
- P13 Multi-line comment
- P14 End of multi-line comment

Obrázek 1: Stavový automat pre lexikálnu analýzu

1. <prog>-> require <exp><prog-con>
2. <prog-con>->ε
3. <prog-con>-> global ID : function(<par-type>) <ret-type><prog-con>
4. <prog-con>-> function ID (<params>)<ret-type><st-list>end<prog-con>
5. <prog-con>-> ID(<arg>)<prog-con>
6. <st-list>-> ε
7. <st-list>->ID<item><st-list>
8. <st-list>-> if <exp> then <st-list> else <st-list> end <st-list>
9. <st-list>-> while <exp> do <st-list>end<st-list>
10. <st-list>->local ID : <type> <init> <st-list>
11. <st-list>-> return <ret-val>
12. <item>->(<arg>)
13. <item>->, ID <item-n>
14. <item>->=<assign>
15. <item-n>->, ID <item-n>
16. <item-n>->=<assign>
17. <assign>->ID(<arg>)
18. <assign>-><exp><assigns>
19. <assigns>->,<exp> <assigns>
20. <assigns> ->ε
21. <init>-> ε
22. <init>->=<init-value>
23. <init-value>-><exp>
24. <init-value>->ID(<arg>)
25. <ret-val>->ε
26. <ret-val>-><value>
27. <value>-><exp> <values>
28. <value>->ID (<arg>)
29. <values>->ε
30. <values>->, <exp><values>
31. <arg> ->ε
32. <arg>-><exp> <args>
33. <args>->ε
34. <args>->,<exp><args>
35. <ret-type>-> ε
36. <ret-type>-> : <type><types>
37. <par-type>-> ε
38. <par-type>-> <type><types>
39. <types>-> ε
40. <types>-> ,<type><types>
41. <params>-> ε
42. <params>-> ID : <type><params_n>
43. <params_n>->, ID :<type><params_n>
44. <params_n>-> ε
45. <type>-> integer
46. <type>-> number
47. <type>-> string

Obrázek 2: LL-gramatika

	require	global	function	ID	if	while	local	return	(	,	=	:	integer	number	string	$
<prog>	1															
<prog-con>		3	4	5												2
<st-list>				7	8	9	10	11								6
<item>									12	13	14					
<item-n>										15	16					
<assign>				17												18
<assigns>										19						20
<init>											22					21
<init-value>				24												23
<ret-val>				26												25, 26
<value>				28												27
<values>										30						29
<arg>																31, 32
<args>										34						33
<ret-type>												36				35
<par-type>													38	38	38	37
<types>										40						39
<params>				42												41
<params_n>										43						44
<type>													45	46	47	

Obrázek 3: LL-tabulka LL-gramatiky

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

Obrázek 4: Tabulka precedenční analýzy