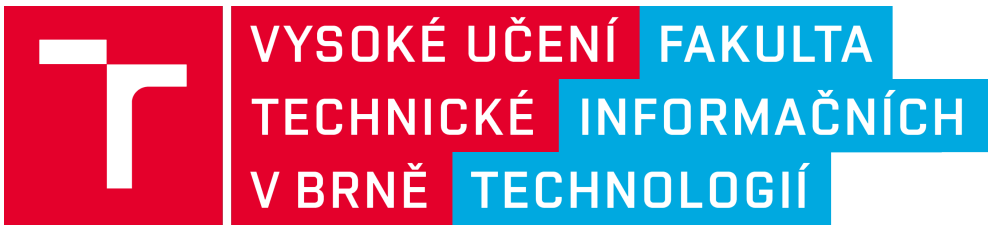


Dokumentace k projektu do předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ20

Tým 86, varianta II

Samuel Valaštín	`xvalas10`	25%
Jonáš Tichý	`xtichy29`	25%
Daniel Gavenda	`xgaven08`	25%
Miroslav Štěpánek	`xstepa68`	25%

9. prosince 2020

Obsah

1 Úvod

Cílem projektu je vytvoření překladače, který ze standardního vstupu načítá zdrojový program jazyka `IFJ20` (podmnožina jazyka GO) a na standartní výstup generuje mezikód v jazyku `IFJcode20`.

2 Použité datové struktury

2.1 Tabulka s rozptýlenými položkami

Základem tabulky s rozptýlenými položkami je efektivní hashovací funkce, u které jsme se inspirovali `sdbm` algoritmem[1]. Pro implementaci jsme zvolili metodu explicitně zřetězených prvků. V programu tato struktura slouží k ukládání údajů o deklaracích proměnných a funkcí. Pro umožnění práce s rámci a zastiňování proměnných jsme se rozhodli vytvořit pro každý rámec unikátní tabulku symbolů o velikosti 509 prvků. Toto číslo bylo zvoleno z důvodu adekvátní velikosti a faktu, že jde o prvočíslo.

2.2 Lineárně vázaný seznam

Tato struktura je využívaná v syntaktické analýze pro ukládání jednotlivých tokenů. Při sémantické analýze a generování kódu slouží k zachování hierarchie vnořených rámců, reprezentovaných tabulkami symbolů.

2.3 Parse tree

Tento strom tvořený uzly `PTNode` explicitně reprezentuje zdrojový kód a je později použit pro tvorbu `AST`.

2.4 AST

`AST` slouží k co nejjednodušší reprezentaci zdrojového kódu, přičemž stále zachovává jeho význam. Tohoto zjednoduššní je dosaženo vynecháním nepotřebných tokenů, jako jsou například odřádkování, závorky nebo klíčová slova. Uzly, které vytvářejí vlastní rámec, obsahují ve své struktuře také ukazatel na příslušnou tabulku symbolů. Jednotlivé uzly jsou reprezentovány strukturou `ASTNode`.

3 Lexikální analýza

První částí při tvorbě překladače byla implementace lexikální analýzy. Hlavní funkcí tohoto modulu je `getToken`. Prostřednictvím této funkce načítáme znaky ze standartního vstupu a transformujeme je na tokeny. Struktura tokenu se skládá z typu a atributu. Atribut je typu `union` a může nabývat hodnoty podle typu tokenu. Pokud je token typu `string` nebo identifikátor, tak nabývá hodnoty `char`$*$. Pro typ `integer` nabývá hodnoty `int64_t`, pro typ `float64` hodnoty `double` a v ostatních případech je atribut prázdný. Lexikální analyzátor je implementovaný nekonečným cyklem, ve kterém pomocí `switch` vybíráme stavy automatu na základě načteného znaku. Pokud načtený znak není v daném stavu povolený, pak funkce vrací příslušnou návratovou hodnotu volající funkci, která zajistí bezpečné ukončení programu. V opačném případě se načítají další znaky a přechází se mezi stavy dokud neskončí v koncovém stavu automatu. Pro zpracování escape sekvencí využíváme buffer, který načítá slovo v hexadecimální soustavě a následně ho převede do `ASCII`.

4 Syntaktická analýza

V této části projektu používáme pro analýzu výrazů precedenční syntaktickou analýzu a pro vše ostatní metodu rekurzivního sestupu za použití námi vytvořené LL – gramatiky. Komunikace s lexikálním analyzátorem probíhá pomocí již dříve zmiňované funkce `getToken`. Jednotlivé tokeny se ukládají do jednostranně vázaného seznamu, pro možnost rychlého a efektivního přístupu k dříve načteným tokenům.

[1] `http://www.cse.yorku.ca/~oz/hash.html`

V rámci rekurzivního sestupu se kromě syntaktické analýzy provádí také tvorba parse tree, který slouží k přesné reprezentaci zdrojového kódu. Tento strom je později použit k tvorbě `AST`.

4.1 Analýza za využití rekurzivního sestupu

Metoda rekurzivního sestupu za použití LL – gramatiky je v naší implementaci řešena souborem konstantních globálních proměnných v souboru `grammar.h`. Pomocí ukazatelů můžeme takto efektivně reprezentovat naši LL – gramatiku bez nutnosti implementace nadbytečného počtu funkcí. Celá syntaktická analýza je tedy reprezentována konstantami v `grammar.h` a jedinou funkcí `recursive` v souboru `parser.c`, která provádí rekurzivní sestup nad gramatikou a kontroluje validitu načtených tokenů. Zbytek funkcí v tomto souboru slouží buď k precedenční syntaktické analýze, tvorbě `AST` nebo uvolňování alokované paměti.

4.2 Precedenční syntaktická analýza

Zpracování výrazů probíhá způsobem bottom-up ve funkci `precExp`, která je volána pokaždé, když rekurzivní funkce najde možné místo výstkytu výrazu. Zde je kontrolována syntaktická správnost výrazu a dále je zde volána funkce `precTable`, která na základě precedenční tabulky rozhoduje, zda bude proveden `shift`, `reduce` nebo se jedná o neplatný stav. Funkce `precShift` do seznamu vloží zarážku, jejíž pozice se nachází před posledním prvkem, který není uzel. Pokud žádný prvek není nalezen, vloží se zarážka na začátek celého seznamu. Následně je vložen i samotný token. Funkce `precReduce` pak redukuje tokeny do struktury `PTNode`, dokud nenarazí na již zmíněnou zarážku. Vše probíhá až do chvíle, kdy nastane některý z chybových stavů nebo se podaří výraz zredukovat do jediné `PTNode`, ve které jsou pak dílčí operace seřazeny podle priority.

4.3 Tvorba AST

Proběhne-li kontrola syntaxe bez nalezení chyby, pak přichází na řadu tvorba `AST`. Tento strom slouží k co nestručnější reprezentaci zdrojového kódu, která je významově identická. K tvorbě využíváme rekurzivního sestupu nad dříve vytvořeným `parse tree`, který zjednodušujeme vynecháním nepotřebných tokenů. Díky tomuto zjednodušení dosáhneme vyšší rychlosti při následných průchodech stromu při sémantické analýze a generování kódu.

4.4 Konverze výrazu do AST

Výrazy jsou za pomocí reklurzivní funkce `PTtoAST` překonvertovány ze struktury `PTNode` do struktury `ASTNode`. Dále jsou odstraněny všechny závorky, které byly u precedence nutné, ale dále již nejsou žádoucí.

5 Sémantická analýza

Tato část je tvořena dvojím průchodem `AST`. V prvním průchodu dochází k naplnění tabulky symbolů globálního rámce definicemi uživatelských funkcí. Tento krok je důležitý pro zajištění možnosti volání funkcí před jejich samotnou definicí ve zdrojovém kódu. Následující průchod provádí kompletní sémantickou analýzu a naplňování příslušných tabulek symbolů. Dochází zde ke kontrole správného volání funkcí, definic, přiřazení, podmínek, cyklů a příkazů `return` pro zajištění bezchybnosti kódu před předáním stromu generátoru mezikódu. Zdrojové kódy sémantické analýzy jsou umístěny v samostatném souboru `semantics.c`, pro větší přehlednost ve struktuře programu.

5.1 Analýza výrazů

Sémantika výrazu je kontrolována funkcí `checkExpression`, která postupuje rekurzivně po uzlech `AST`, dokud neurčí datové typy všech operandů výrazu. Tyto typy mezi sebou porovná a zkontroluje, zda jsou

identické. Dále je také kontrolováno, zda jsou operátory přítomné ve výrazu kompatibilní s datovým typem `string`, zda není jmenovatel při dělení nulová konstanta nebo zda se ve výrazu nachází pouze jeden relační operátor, který musí být kořenem celého `AST stromu`.

6 Generování kódu

Vstupem při generovaní kódu je ukazatel na kořen abstraktního syntaktického stromu. Výstupem je mezikód `IFJcode20`, který je generovaný na standartní výstup.

6.1 Struktura výsledného programu

Mezikód začíná prologem a skokem na hlavní funkci `main`. Následně se generují jednotlivé funkce. Vzhledem k zadání pracujeme s běžným lokálním rámcem. V případě volání funkce vytváříme dočasný rámec, do kterého ukládáme parametry funkce. Vestavěné funkce generujeme jen v případě, že jsou volané ve zdrojovém kódu vstupního programu. Jejich samotné generování se vykoná až po vygenerování vstupního kódu. Speciálním případem vestavěné funkce je `print`, který generujeme, vzhledem k proměnlivému počtu parametrů a snížení ceny interpretace kódu, pro každé volaní zvlášť.

6.2 Řešení různého zanoření proměnných

Vzhledem k možnému problému interpretace, při výskytu proměnných se stejnými identifikátory, v rozdílných zanořeních, na začátku každé funkce předem definujeme všechny proměnné. Název proměnné se skladá ze jména proměnné, za kterým následuje „$*$“ a hloubka zanoření. Výjimku tvoří definice pomocných proměnných při vícenásobném přiřazení, které ukládáme jako `tmpVar` a číslo proměnné.

6.3 Generování výrazů

Vyhodnocování výrazů probíhá rekurzivně, přičemž používáme instrukce, které pracují se zásobníkem. V případě výskytu operace „$+$“ a „$/$“ se před vyhodnocením zanoříme a určíme typ literálu nebo identifikátoru pomocí tabulky symbolů.

7 Práce v týmu

7.1 Způsob práce

Jako vývojové prostředí jsme si určili `Visual Studio Code`, které nabízí jednoduchou implementaci verzovacího systému `Git`. Jako společný vzdálený repozitář jsme používali webovou službu `GitHub`. Vzhledem k náročnosti projektu jsme v průběhu vývoje využívali párové programování, přičemž jsme se rozdělili na slovenskou a českou část.

7.2 Rozdělení práce

- Daniel Gavenda – lexikální analýza, generátor kódu
- Samuel Valaštín – lexikální analýza, generátor kódu
- Jonáš Tichý – syntaktická analýza, sémantická analýza, AST, tabulka symbolů, dokumentace
- Miroslav Štěpánek – syntaktická analýza, syntaktická a sémantická analýza výrazů, dokumentace

7.3 Komunikace

Na projektu jsme začali pracovat týden po zveřejnění zadání. Stihli jsme uskutečnit jednu schůzi, na které jsme si přiblížili detaily zadání a rozdělili jsme si práci na projektu. Vzhledem k situaci jsme přesunuli komunikaci na vlastní server Discord. Každý týden jsme uspořádali konferenční hovor, při kterém jsme si sdělili nový postup ve vývoji projektu.

8 Závěr

Projekt se nám podařilo dokončit s dostatečnou časovou rezervou. Díky dobré týmové komunikaci jsme dokázali vyřešit mnoho problémů, které při vývoji nastaly a nasbírat cenné zkušenosti z oblasti vývoje a týmové spolupráce.

Odkazy

[1] Jack Crenshaw. *Let's Build a Compiler*. [online]. [cit. 9.12.2020]. 1997-2008. URL: `https://compilers.iecc.com/crenshaw/`.

[2] Ruslan Spivak. *Let's Build A Simple Interpreter*. [online]. [cit. 9.12.2020]. 2015. URL: `https://ruslanspivak.com/lsbasi-part1/`.

[3] Hayo Thielecke. *Typed trees and tree walking in C with struct, union, enum, and switch*. [online]. [cit. 9.12.2020]. URL: `https://www.cs.bham.ac.uk/~hxt/2015/c-plus-plus/trees-in-c-slides.pdf`.

9 Přílohy

1. $< prog > \rightarrow < multiEol >$ **package** $< multiEol >$ **main** **EOL** $< progBody >$
2. $< progBody > \rightarrow < multiEol > < func > < progBody >$
3. $< progBody > \rightarrow < multiEol >$
4. $< func > \rightarrow$ **func** **ID** $< funcParams > < funcRetTypes >$ **{** **EOL** $< body >$ **}**
5. $< funcParams > \rightarrow$ **(** $< params >$ **)**
6. $< funcParams > \rightarrow$ **(** **)**
7. $< params > \rightarrow$ **ID** $< type >$ **,** $< params >$
8. $< params > \rightarrow$ **ID** $< type >$
9. $< funcRetTypes > \rightarrow$ **(** $< retTypes >$ **)**
10. $< funcRetTypes > \rightarrow$ **(** **)**
11. $< funcRetTypes > \rightarrow \epsilon$
12. $< retTypes > \rightarrow < type >$ **,** $< retTypes >$
13. $< retTypes > \rightarrow < type >$
14. $< for > \rightarrow$ **for** $< forDec >$ **;** $< expr >$ **;** $< forInc >$ **{** **EOL** $< body >$ **}**
15. $< forDef > \rightarrow < define >$
16. $< forDef > \rightarrow \epsilon$
17. $< forInc > \rightarrow < assign >$
18. $< forInc > \rightarrow \epsilon$
19. $< if > \rightarrow$ **if** $< expr >$ **{** **EOL** $< body >$ **}** **else** **{** **EOL** $< body >$ **}**
20. $< body > \rightarrow < statement >$ **EOL** $< body >$
21. $< body > \rightarrow \epsilon$
22. $< statement > \rightarrow < If >$
23. $< statement > \rightarrow < For >$
24. $< statement > \rightarrow < define >$
25. $< statement > \rightarrow < assign >$
26. $< statement > \rightarrow < funcCall >$
27. $< statement > \rightarrow < ret >$
28. $< statement > \rightarrow \epsilon$
29. $< funcCall > \rightarrow$ **ID** **(** $< funcCallArgs >$ **)**
30. $< funcCall > \rightarrow$ **ID** **(** **)**
31. $< funcCallArgs > \rightarrow < operand >$ **,** $< funcCallArgs >$

32. $< funcCallArgs > \rightarrow < operand >$
33. $< multiExpr > \rightarrow < expr >$ **,** $< multiExpr >$
34. $< multiExpr > \rightarrow < expr >$
35. $< define > \rightarrow$ **ID** := $< eolOpt > < expr >$
36. $< ret > \rightarrow$ **return** $< multiExpr >$
37. $< ret > \rightarrow$ **return**
38. $< assign > \rightarrow < multId >$ = $< eolOpt > < funcCall >$
39. $< assign > \rightarrow < multId >$ = $< eolOpt > < multiExpr >$
40. $< lParenthsOpt > \rightarrow$ **(** $< multiEol > < lParenthsOpt >$
41. $< lParenthsOpt > \rightarrow \epsilon$
42. $< rParenthsOpt > \rightarrow$ **)** $< rParenthsOpt >$
43. $< rParenthsOpt > \rightarrow \epsilon$
44. $< operand > \rightarrow$ **ID**
45. $< operand > \rightarrow$ **intLit**
46. $< operand > \rightarrow$ **fltLit**
47. $< operand > \rightarrow$ **strLit**
48. $< type > \rightarrow$ **int**
49. $< type > \rightarrow$ **string**
50. $< type > \rightarrow$ **float64**
51. $< multId > \rightarrow$ **ID** **,** $< multId >$
52. $< multId > \rightarrow$ **ID**
53. $< multiEol > \rightarrow$ **EOL** $< multiEol >$
54. $< multiEol > \rightarrow \epsilon$
55. $< eolOpt > \rightarrow$ **EOL**
56. $< eolOpt > \rightarrow \epsilon$

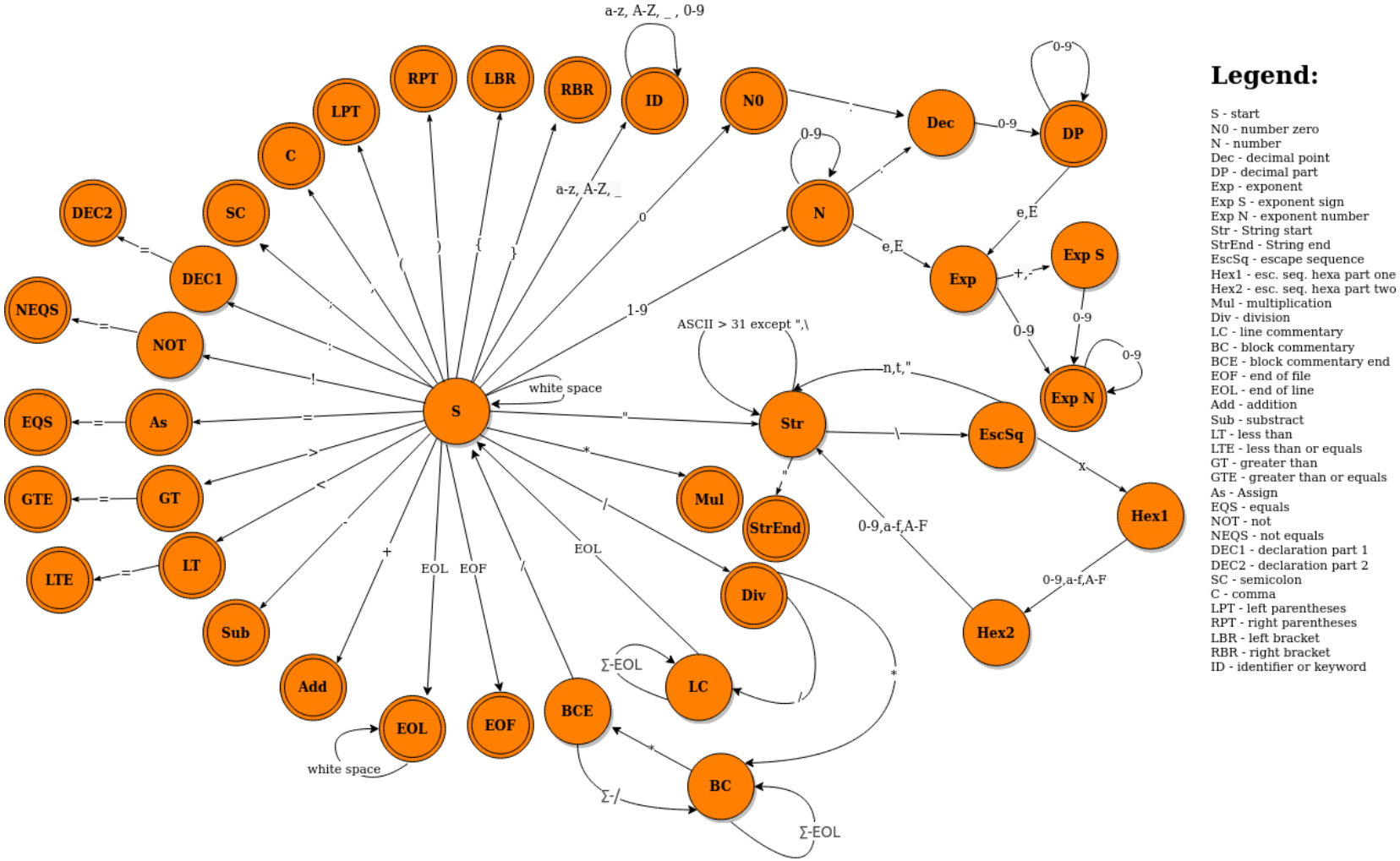
Tabulka 1: LL – gramatika

	+, -	*, /	**Rel**	(	)	**Lit, ID**	**$**
+, -	>	<	>	<	>	<	>
*, /	>	>	>	<	>	<	>
Rel	<	<		<	>	<	>
(	<	<	<	<	=	<	
)	>	>	>		>		>
Lit, ID	>	>	>		>		>
$	<	<	<	<		<	

Tabulka 2: Precedenční tabulka

	EOL	ID	eps	func	for	if	return	int	string	float64	(	)	intLit	strLit	fltLit
$< prog >$	1		1												
$< progBody >$	2,3		2,3												
$< func >$				4											
$< funcParams >$											5,6				
$< params >$		7,8													
$< funcRetTypes >$			11								9,10				
$< retTypes >$								12,13	12,13	12,13					
$< for >$					14										
$< forDef >$	16	15													
$< forInc >$	18	17													
$< if >$						19									
$< body >$		20	21		20	20	20								
$< statement >$		24,25,26	28		23	22	27								
$< funcCall >$		29,30													
$< funcCallArgs >$		31,32											31,32	31,32	31,32
$< multiExpr >$			33,34												
$< define >$		35													
$< ret >$							36,37								
$< assign >$		38,39													
$< lParenthsOpt >$			41								40				
$< rParenthsOpt >$			43									42			
$< operand >$		44											45	47	46
$< type >$								48	49	50					
$< multId >$		51,52													
$< multiEol >$	53		54												
$< eolOpt >$	55		56												

Tabulka 3: LL – tabulka



Obrázek 1: Diagram konečného automatu