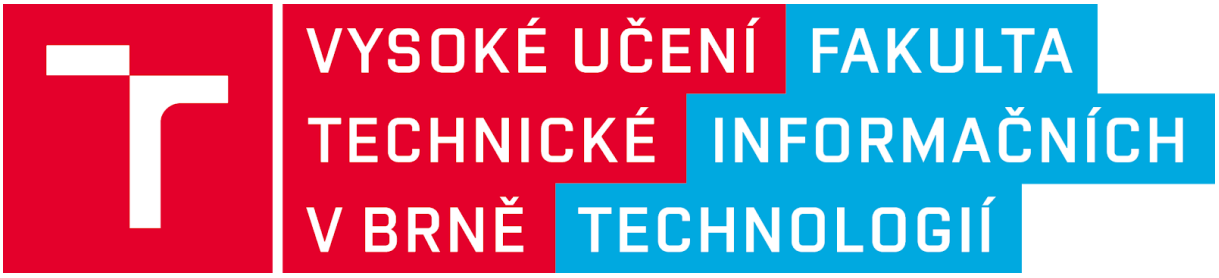


Dokumentace k projektu

Implementace překladače imperativního jazyka IFJ20

Tým 126, varianta II

Řešitelé:

Julie Gyselová (xgysel00)	25 %
Adrián Kálazi (xkalaz00)	25 %
Kevin Lackó (xlacko08)	25 %
Juraj Hrdlica (xhrdli14)	25 %

Rozšíření:

BASE

UNARY

BOOLTHEN

Úvod

Tato dokumentace popisuje vývoj překladače pro jazyk IFJ20, který je podmnožinou jazyka Go. Vybrali jsme si variantu zadání II, podle které jsme tabulku symbolů implementovali pomocí tabulky s rozptýlenými položkami.

Návrh

Lexikální analýza

První částí projektu, která byla implementována, je scanner, který provádí lexikální analýzu. Podle konečného automatu, který jsme na začátku navrhli, jsme v modulech *scanner* a *scanner_states* implementovali scanner, který načítá znaky ze standardního vstupu a převádí je na tokeny (struktura), které ukládá do jednosměrně vázaného seznamu.

Zvlášť byl řešen případ identifikátorů a klíčových slov. Protože konečný automat neumožňuje rozlišení těchto dvou případů, vždy nejprve ověřujeme, že načtená sekvence znaků neodpovídá některému z klíčových slov, než vracíme token označující identifikátor.

Pokud scanner narazí na posloupnost znaků, která neodpovídá žádnému tokenu, celý program končí s příznakem lexikální chyby.

Syntaktická analýza

Syntaktická analýza

Syntaktická analýza je prováděna v modulech *parser* a *rules* metodou rekurzivního sestupu ve dvou průchodech. V prvním průchodu se ukládají a kontrolují pouze hlavičky funkcí a ve druhém průchodu se kontroluje celý program.

Gramatika sestavená podle specifikace jazyka IFJ20, podle které jsme náš parser implementovali, není typu LL(1), který by byl ideální. Ve dvou případech je potřeba znát dva tokeny, aby bylo možné rozhodnout, které pravidlo bude použito. Ačkoliv by pravděpodobně bylo možné sestavit gramatiku tak, aby byla typu LL(1), v této dokumentaci přikládáme gramatiku a LL-tabulku, které odpovídají naší implementaci. Pravidla (31-33, 49-50), u kterých je potřeba znát dva tokeny místo jednoho, jsou v gramatice i v LL-tabulce zvýrazněná.

Precedenční syntaktická analýza

Pro zpracování výrazů je použitá metoda syntaktické analýzy zdola nahoru pomocí precedenční tabulky.

Funkce *parse_expr* postupně načítá tokeny do zásobníku, určuje jejich typ a pomocí tabulky určuje vztah mezi prvním nezpracovaným tokenem na vrcholu zásobníku a tokenem na vstupu.

Pokud tabulka určí, že může být použito některé z pravidel, volá se funkce *reduce*, která zjišťuje které pravidlo se použije. Pokud žádné pravidlo neodpovídá tokenům na vrcholu zásobníku, syntaktická analýza končí s příznakem syntaktické chyby.

Gramatická pravidla pro analýzu výrazů

1. E -> (E)
2. E -> +E, -E, !E
3. E -> E * E, E / E
4. E -> E + E, E - E
5. E -> E < E, E <= E, E > E, E >= E
6. E -> E == E, E != E
7. E -> E && E
8. E -> E || E
9. E -> i

Sémantická analýza

Sémantická analýza je prováděna zároveň se syntaktickou analýzou podle sémantických pravidel jazyka IFJ20. Na příslušných místech v parseru jsou prováděny kontroly - zda použité funkce a proměnné jsou definované, zda souhlasí datové typy a počty proměnných a výrazů, které do nich jsou přiřazovány, zda nedochází k dělení nulou a zda návratové hodnoty funkce odpovídají její hlavičce.

Při zjištění sémantické chyby končí program s odpovídající návratovou hodnotou, a navíc na standardní chybový výstup vypisuje chybovou zprávu s podrobnostmi o chybě. Chybové hlášky byly inspirovány Go překladačem[3].

Generování IFJcode20

Generování výstupního kódu IFJcode20 je poslední fází našeho překladače. Generátor se skládá z předpřipravených funkcí v modulu *generator*, které se volají z odpovídajících míst v parseru.

Na začátku generátor vždy vypíše definice vestavěných funkcí jazyka IFJ20 a zavolá funkci main. Za zmínku také stojí to, že příkaz CREATEFRAME je použit vždy pouze na

začátku funkce a rozlišování proměnných napříč různými rámci je řešeno pomocí unikátních názvů.

Použité datové struktury

Jednosměrně vázaný seznam

Jednosměrně vázaný seznam je struktura, kterou jsme v projektu hojně využívali. Implementovali jsme jej proto velmi obecným způsobem, abychom jej mohli využít na více místech (např. v sémantické analýze při hlídání návratových typů funkcí).

Tabulka symbolů

Abychom správně mohli řešit rámce, je naše tabulka symbolů ve skutečnosti implementovaná jako zásobník tabulek s rozptýlenými prvky. V těchto tabulkách ukládáme různé informace o proměnných (názvy, datové typy a hodnoty) a o funkcích (názvy, seznam vstupních argumentů a seznam návratových datových typů).

Tabulka s rozptýlenými položkami

Implementace tabulky s rozptýlenými položkami byla inspirována druhým projektem předmětu IJC[1]. Jako hashovací funkci jsme použili algoritmus FNV-1a[2], který se nám zdál nejoptimálnější.

Implementace

Způsob práce v týmu

Překladač jsme postupně vyvíjeli v průběhu října, listopadu a prosince. Při práci jsme k verzování používali systém Git a k vytváření testů, které jsme všichni psali k různým částem projektu, jsme využili framework Googletest. Komunikovali jsme výhradně přes platformu Discord a vzhledem k aktuální situaci, všechny naše schůzky probíhaly online.

Rozdělení práce v týmu

- Julie Gyselová (25 %)
  - Návrh gramatiky
  - Syntaktická analýza
  - Analýza výrazů
  - Dokumentace
- Adrián Kálazi (25 %)
  - Build setup (Make, CMake, CI)
  - Lexikální analýza (implementace přechodu mezi stavy)
  - Dynamické pole znaků
  - Generování výstupního kódu
- Kevin Lackó (25 %)
  - Návrh gramatiky
  - Syntaktická analýza
  - Sémantická analýza
  - Tabulka symbolů
- Juraj Hrdlica (25 %)
  - Návrh konečného automatu
  - Lexikální analýza

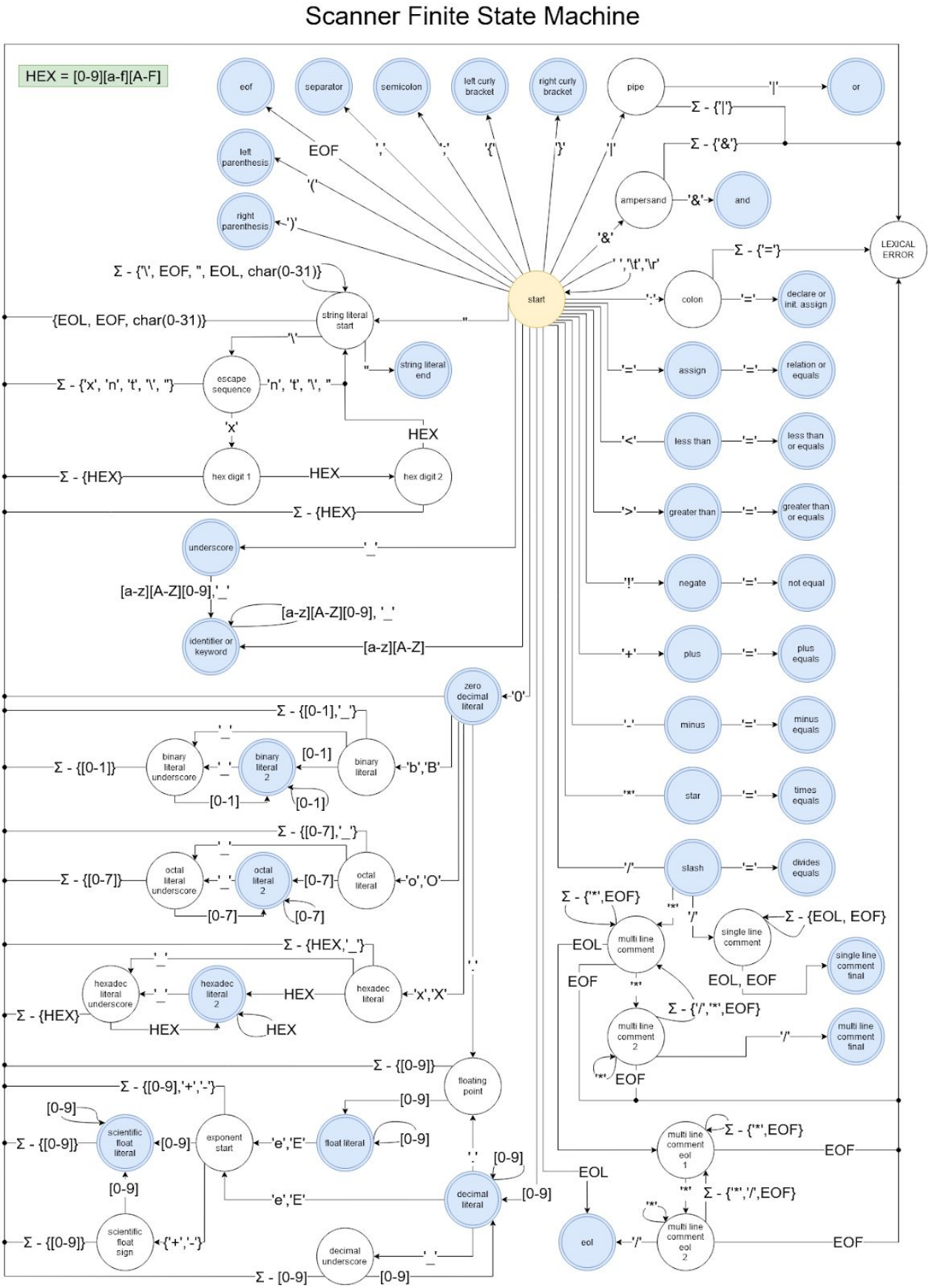
Použité zdroje

[1] https://www.fit.vutbr.cz/study/courses/IJC/public/DU2.html

[2] http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-1a

[3] https://play.golang.org

Konečný automat



Precedenční tabulka

	(	)	un	* /	+ -	< <= > >=	== !=	&&	||	i	$
(	<	=	<	<	<	<	?	<	<	<	
)		>	>	>	>	>	>	>	>		>
un	<	<	<	>	>	>	>	>	>	<	>
*/	<	>	<	>	>	>	>	>	>	<	>
+-	<	>	<	<	>	>	>	>	>	<	>
< <= > >=	<	>	<	<	<	>	>	>	>	<	>
== !=	<	?	<	<	<	<	>	>	>	<	>
&&	<	>	<	<	<	<	<	>	>	<	>
||	<	>	<	<	<	<	<	<	>	<	>
i		>		>	>	>	>	>	>		>
$	<		<	<	<	<	<	<	<	<	

Gramatika

LL-gramatika

1. Program -> Eol_opt_n Prolog Eol_opt_n Functions Eol_opt_n eof
2. Prolog -> *package main eol*
3. Functions -> Function Eol_opt_n Functions
4. Functions -> *eps*
5. Function -> *func id* Param_list Returntype_list *l_curly eol*
   Eol_opt_n Statements *r_curly*
6. Param_list -> *l_parenthesis* Params *r_parenthesis*
7. Params -> Param Param_n
8. Params -> *eps* .
9. Param_n -> *comma* Eol_opt_n Param Param_n
10. Param_n -> *eps*
11. Param -> *id* Paramtype
12. Paramtype -> *int*
13. Paramtype -> *float64*
14. Paramtype -> *string*
15. Paramtype -> *bool*
16. Returntype_list ->*l_parenthesis* Returntypes *r_parenthesis*
17. Returntype_list -> *eps*
18. Returntypes -> Returntype Returntype_n
19. Returntypes -> *eps*
20. Returntype_n -> comma Eol_opt_n Returntype Returntype_n
21. Returntype_n -> eps
22. Returntype -> *int*
23. Returntype -> *float64*
24. Returntype -> *string*
25. Returntype -> *bool*
26. Statements ->Def_ass_call Eol_opt_n Statements
27. Statements -> Conditionals Eol_opt_n Statements
28. Statements -> Iterative Eol_opt_n Statements
29. Statements -> Return Eol_opt_n Statements
30. Statements -> *eps*
31. Def_ass_call -> Var_define *eol*
32. Def_ass_call -> Function_call *eol*
33. Def_ass_call -> Assignment *eol*
34. Var_define_opt -> Var_define
35. Var_define_opt -> *eps*
36. Var_define -> *id define_op* Expression
37. Conditionals -> Conditional Conditional_n *eol*
38. Conditional_n -> *else* Else
39. Conditional_n -> *eps* .
40. Else -> Conditional Conditional_n
41. Else -> *l_curly eol* Eol_opt_n Statements *r_curly* .
42. Conditional -> *if* Expression *l_curly eol*
43. Conditional -> Eol_opt_n Statements *r_curly*
44. Iterative -> *for* Var_define_opt *semicolon* Expression *semicolon*
45. Iterative -> Assignment_opt *l_curly eol*
46. Iterative ->Eol_opt_n Statements *r_curly eol*
47. Assignment_opt -> Assignment
48. Assignment_opt -> *eps*
49. Assignment -> Assignment_list
50. Assignment ->Id Assign_op Exprs_funcall
51. Assignment_list -> Ids assign Eol_opt_n Exprs_funcall

52. Ids -> Id Id_n
53. Id_n -> *comma* Id Id_n
54. Id_n -> *eps*
55. Id -> *id*
56. Id -> *underscore*
57. Assign_op -> *plus_assign* Eol_opt_n
58. Assign_op -> *minus_assign* Eol_opt_n
59. Assign_op -> *multiply_assign* Eol_opt_n
60. Assign_op -> *divide_assign* Eol_opt_n
61. Assign_op ->*assign* Eol_opt_n
62. Exprs_funcall -> Expression
63. Exprs_funcall -> Function_call
64. Function_call -> *id l_parenthesis* Eol_opt_n Arguments *r_parenthesis*
65. Arguments -> Argument Argument_n
66. Arguments -> *eps*
67. Argument_n -> *comma* Eol_opt_n Argument Argument_n
68. Argument_n -> *eps*
69. Argument -> *id*
70. Argument -> Literal
71. Return -> *return* Expressions_opt *eol*
72. Expressions_opt -> Expressions
73. Expressions_opt -> *eps*
74. Expressions -> *expression* Expression_n
75. Expression_n -> *comma expression* Expression_n
76. Expression_n -> *eps*
77. Literal -> *int_lit*
78. Literal -> *float_lit*
79. Literal -> *string_lit*
80. Literal -> *true*
81. Literal -> *false*
82. Eol_opt_n -> *eol* Eol_opt_n
83. Eol_opt_n -> *eps*

LL tabulka

	$	eps	eol	false	true	string	float	int	,	return	id	)	(	=	/=	*=	-=	+=	_	}	{	;	for	if	else	:=	bool	string	float64	int	func	main	package	eof
Program		1	1																														
Prolog																																2	
Functions		4																												3			
Function																														5			
Param_list													6																				
Params		8									7																						
Param_n		10							9																								
Param											11																						
Paramtype																										15	14	13	12				
Returntype_list		17											16																				
Returntypes		19																								18	18	18	19				
Returntype_n		21							20																								
Returntype																										25	24	23	22				
Statements		30								29	26								26				28	27									
Def_ass_call											31, 32, 33								33														
Var_define_opt		35									34																						
Var_define											36																						
Conditionals																								37									
Conditional_n		39																							38								
Else																					41			41									
Conditional																								21									
Iterative																							44										
Assignment_opt		48									47								47														
Assignment											49, 50								49, 50														
Assignment_list											51								51														
Ids											52								52														
Id_n		54							53																								
Id											55								56														
Assign_op														61	60	59	58	57															
Exprs_funcall											63																						
Function_call											64																						
Arguments		66		65	65	65	65	65			65																						
Argument_n		68							67																								
Argument				70	70	70	70	70			69																						
Return										71																							
Expressions_opt		73																															
Expressions																																	
Expression_n		76							75																								
Literal				81	81	79	78	77																									
Eol_opt_n		83	82																														