PROJECT DOCUMENTATION

Implementation of compiler IFJ21

Team 132, variant II.

Ondřej Keprt, xkeprt03: 40%
Oleg Trofimov, xtrofi00: 30%
Maxim Gerasimov, xgeras00: 30%
Sláma Ondřej, xslama32: 0%

December 9, 2021

Contents

Introduction

This project should test students teoretical knowledge about finite automats, languages and compilers and their application to practice. Project goal was implementation of compiler for language IFJ21, which is subset of language Teal/Lua, to three adress code IFJcode21.

1 Lexical anylisis

When creating a compiler, we started with the implementation of lexical analysis. Lexical analysis is implemented, including additional support functions and structures, in the source file `scanner.c`.

The main function of this analysis is `read_token`, which reads character by character from the source file and passes this to the `token` structure. For the correct operation of this function, which will be an automaton, the following functions will be used:

- struct `String` for Representation of dynamic string
- funkce `str_init` for initialization of memory
- funkce `str_free` for freeing up memory
- funkce `extend_buffer` for memory expansion
- funkce `str_add_char` for saving read characters
- Token_data `union` data, which token contains
- struct `Token` contains data and type of `token`
- funkce `create_token` alocate memory for token
- funkce `add_str_to_token` add identif to data.str
- funkce `delete_token` freeing up memory data.str if was a mistake

The `token` consists of types and data. Token types can be *keywords*, *identifiers*, *integers*, *floating point numbers*, *strings*, *operators*, *EOF*, and other characters that can be used in the language IFJ21. The `union` type is used to write `data`. If the token type is a string or an identifier, then the data will be stored as a string. If the token type is a number (integer or floating point) then the data will be stored as a number (integer or floating point).

The entire lexical analyzer is implemented in the form of a deterministic finite automaton. The state machine sequentially reads the symbol and, depending on the symbol, enters the corresponding state as indicated in the diagram 1. If an incorrect character is read in this case, the program will terminate and give an error. Otherwise, we read the characters until we get a `token` that we will return and terminate the program.

To process escape sequences in a string, an `array` of size 3 was created, which is sequentially filled with read numbers of type $'\backslash ddd'$ in the range 000-255, which we will later convert to a number corresponding to a character in an ASCII table.

2 Parser

Parser functions were implemented in three files:
`parser.c, parser_recursive_descent.c, parser_precedence.c`
For expressions were assigned precedence analysis. All other constructions were processed by recursive descent. In `parser.*` are defined functions and data structures, which are used in both approaches.

Comunication between parser parts, was implemented by passing pointer on data structure `parser_data_t`, where were stored all information about analysis state and loaded data. Precedence analysis processed tokens dermined for expression, generated instructions for calculations and with function `push_precedence_token` added precedence token at the end of `data->expression_list`, and returned control to recursive descent. After that, recursive descent continued in program processing.

Lexical analyzator was called with function `read_token`, which returned pointer on structure of token, which was on standart input.

2.1 Recursive descent

Recursive descent is implemented based on LL(1) grammar 8 and LL – table 9. For each non – terminal was created function, which simulate that non – terminal. In this function is sequence of comparing lexical tokens from lexical analysis and deciding, which rule should be used for generating new non – terminals or terminals. Unfortunatelly, there are two cases, which we can not decide only by LL(1) grammar. It is when we got token representing identifier, when we are in terminal `<assignment>` or `<init2>`. Identifier can be start of function call or identifier of local variable. In this case we used help of semantic analysis, we searched in global table of symbols and if we find function, we generate function call. If we did not find, we generate non – terminal for expression.

Alongside with syntax analysis, there are semantics controls and semantics actions, such as generating instructions or type controls. All function return bool value, true means all is ok, false error occured. As parameter they all have pointer on structure `parset_data_t`, which contains all information needed for analysis.

2.2 Precedence

Main body of precedence evaluating is in `parser_precedence.c`, mainly function precedence.

After initialized buffer, precedence table and rule table. Three main functions are `check_rule`, `precedence_compare` and `reduce_fnc`.

Firstly `check_rule` controles if such combination of tokens exists. Secondly, depending on what precedence is, `precedence_compare` returns commands: push, reduce or no precedence. And last `reduce_fnc` creates instruction for code generator and popping out reduced tokens.

My actions are writen in table 2

3 Table of symbols

Our task was implement table of symbols as hash table. Our hash table was implemented as structure, which contains item counter (`size`), array of pointers on lists of synnonyms (`array`), array size (`arr_size`) and pointer on next hash table. For deciding index, where item will be stored is used `sdbm` hash function, which is used in berkley database [**?**].

There are two hash tables pointers in `parser_data_t`. One for global frame, where are stored data only about functions. Second, contains data about local variables. Table for local variables is implemented by stack hierarchy. For each scope is created new hash table and pushed on the top of the stack. When we are trying to add variable, we search only on top of stack, if there is variable with same name. When we are looking for variable, we search item in top level, when we can not find it, we continue in lower level, until we are out of levels. When we are leaving from scope, we pops top level hash table.

Hash table stores one type of structure in lists, which is same for functions and local variables, but data, which contains are different.

Item which represents function, contains key, pointer on list of parameters, pointer on list of return types. This lists use datastructure `data_token_t`. When function is declared, parameter list contains only data types as return list. Defined function have parameter names stored in parameter list. Type of item is set to `function_declared or function_defined`. Item have pointer on next item or `NULL`, if there is none and `frame_ID` is undefined value.

Item which represents local variable have set key, data type (string, number, integer) and `frame_ID`, where was defined. Pointers on lists are set on `NULL`.

4 Code generating

At the beginnig of project we wanted implement only one – pass compiler, but there was problem with definitions of variables inside while loop. So we decided, that we create list of instructions, which is stored inside `parset_data_t`. So we added instruction at the end of list, except variable definitions. If parser recognize, that on the input is loop construction, he made special pointer to the list, where was inserted all instructions for defining variable. In `parset_data_t` is another special list, which contains

At the end of analysis, parser called function `generate_code`, which printed instructions and freed memory.

5 Work in team

5.1 Versioning system

As version system we were using Git, with hosting on Github. Every team member worked in own branch. When feature was created, branch was pulled to main branch and used by others.

5.2 Communication

For communication we used own Discord server with specialized chat thread for IFJ project. In case of some big problem, we used voice rooms for call and and solved problem with screen sharing.

All team members had access to this server, including Ondřej Sláma, but he did not respond. Team leader and other members send him some messages, with work to do, but he did not show any effort or any work. In the end, 3.12.2021 he wrote message to team leader and offered help, but by this time, all work was almost done or devided to other members.

5.3 Distribution of points

We divided the work on the project taking into account its complexity and labor intensity. Thus, everyone received a percentage rating in the amount indicated in the table 1.

Team member	Accomplished work
Ondřej Keprt, xkeprt03: 40%	team management, organization of work, control over the execution of work, consultation, project structure, testing, documentation, syntax and semantic analysis ⇒ recursive descent, hash table, code generating ⇒ prepared functions, design, generating program flow, function calls...
Oleg Trofimov, xtrofi00: 30%	lexical analysis, testing, documentation
Maxim Gerasimov, xgeras00: 30%	syntax and semantic analysis ⇒ precedence, generating expressions, testing, documentation
Sláma Ondřej, xslama32: 0%	nothing is done

Table 1: Distribution of work in the team

6 Deterministic finite automaton

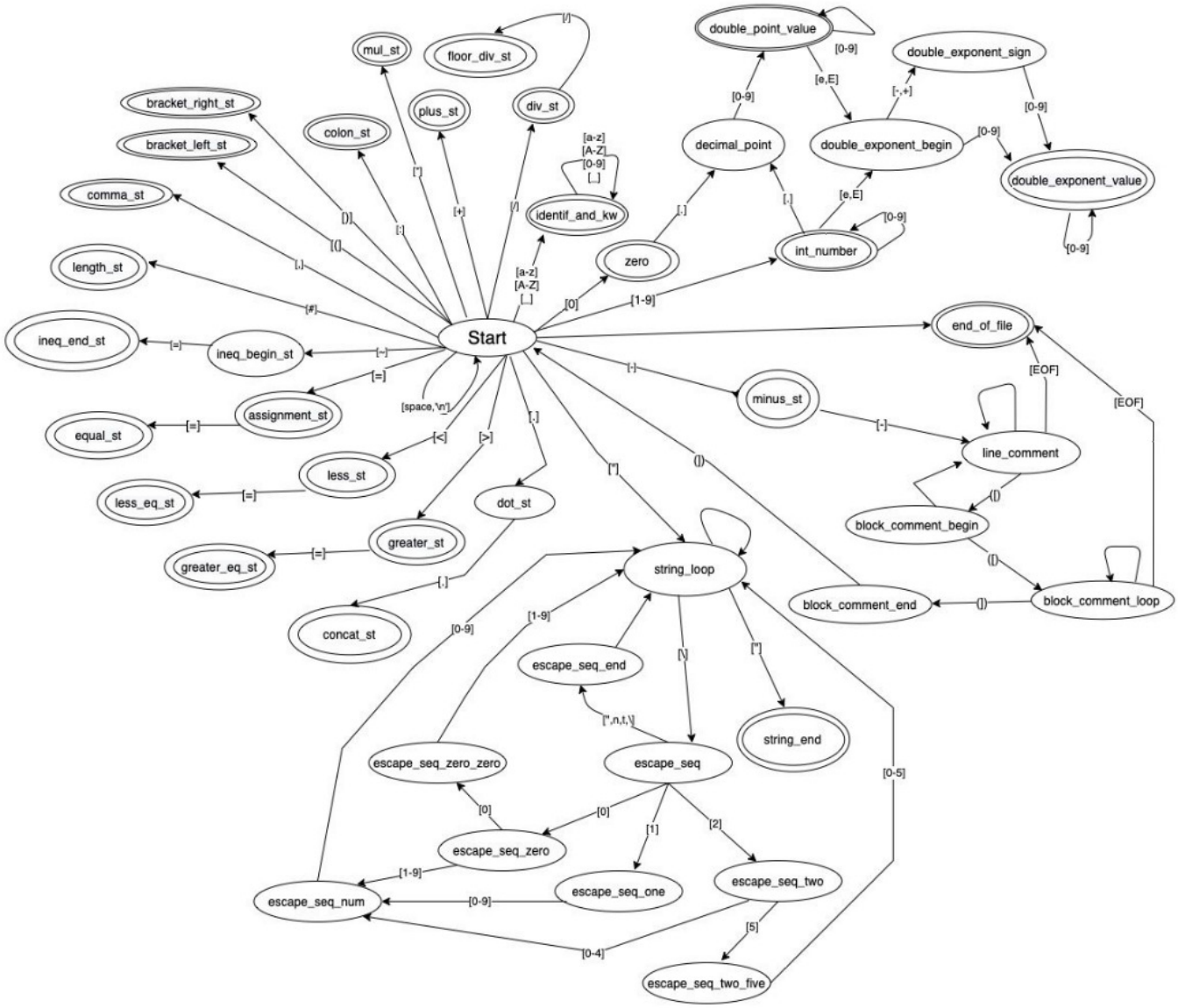

Figure 1: Deterministic finite automaton

7 Precedence table

	stdin																				
buffer	#	*	/	//	+	-	..	<	<=	>	>=	==	~=	(	)	id	int	number	str	nil	$
#	=	<	<	<	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	<	>
*	R	>	>	>	>	>	>	<	<	<	<	<	<	<	>	<	<	<	<	<	>
/	>	>	>	>	>	>	>	<	<	<	<	<	<	<	>	<	<	<	<	<	>
//	>	>	>	>	>	>	>	<	<	<	<	<	<	<	>	<	<	<	<	<	>
+	>	<	<	<	>	>	>	<	<	<	<	<	<	<	>	<	<	<	<	<	>
-	>	<	<	<	>	>	>	<	<	<	<	<	<	<	>	<	<	<	<	<	>
..	>	<	<	<	<	<	>	<	<	<	<	<	<	<	>	<	<	<	<	<	>
<	>	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	<	>
<=	>	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	<	>
>	>	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	<	>
>=	>	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	<	>
==	>	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	<	>
~=	>	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<		<	<	<	<	<	>
)	>	>	>	>	>	>	<	>	>	>	>	>	>		<	<	<	<	<	<	>
id	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>						>
int	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>						>
num	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>						>
str	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>						>
nil	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>						>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<						>

Figure 2: Precedence table

8 LL grammar

1. <intro>→<prolog> <prog>
2. <prog>→ <fce_decl> <prog>
3. <prog>→ <fce_def> <prog>
4. <prog>→ identif <call_fce> <prog>
5. <prog>→ ϵ
6. <prolog>→require "ifj21"
7. <fce_decl>→global identif : function (<type_list>) <ret_list>
8. <type_list>→ϵ
9. <type_list>→<type><type_list2>
10. <type_list2>→,<type><type_list2>
11. <type_list2>→ϵ
12. <ret_list>→: <type><type_list2>
13. <ret_list>→ϵ
14. <fce_def>→function identif (<param_def_list>) <ret_list> <st_list> end
15. <param_def_list>→ϵ
16. <param_def_list>→<param> <param_def_list2>
17. <param_def_list2>→,<param> <param_def_list2>

18. <param_def_list2>→ϵ
19. <param>→identif : <type>
20. <st_list>→<statement> <st_list>
21. <st_list>→ϵ
22. <statement>→local identif:<type><init>
23. <init>→ϵ
24. <init>→=<init2>
25. <init2>→identif<call_fce>
26. <init2>→<expression>
27. <statement>→identif <after_id>
28. <after_id>→<call_fce>
29. <after_id>→<identif_list> = <assignment>
30. <identif_list>→, identif<identif_list>
31. <identif_list>→ϵ
32. <assignment>→<expression><expression_list2>
33. <assignment>→identif<call_fce>
34. <expression_list>→<expression><expression_list2>
35. <expression_list>→ϵ
36. <expression_list2>→,<expression><expression_list2>
37. <expression_list2>→ϵ
38. <statement>→if<expression>then<st_list>else<st_list>end
39. <statement>→while<expression>do<st_list>end
40. <st_list>→return<expression_list>
41. <call_fce>→(<value_list>)
42. <value_list>→ϵ
43. <value_list>→<value><value_list2>
44. <value_list2>→,<value><value_list2>
45. <value_list2>→ϵ
46. <value>→integer_value
47. <value>→number_value

48. <value>→string_value
49. <value>→identif
50. <value>→nil
51. <type>→integer
52. <type>→number
53. <type>→string

9 LL – table

	do	else	end	function	global	if	integer	local	nil	number	require	return	string	then	while	(	)	:	,	=	integer_value	number_value	string_value	EOF	identif	#
<intro>											1															
<prog>				3	2																			5	4	
<prolog>											6															
<fce_decl>					7																					
<type_list>							9			9			9				8									
<type_list2>			11	11	11	11		11				11			11		11		10					11	11	
<ret_list>			13	13	13	13		13				13			13			12						13	13	
<fce_def>				14																						
<param_def_list>																	15								16	
<param_def_list2>																	18		17							
<param>																									19	
<st_list>		21	21			20		20				40			20										20	
<statement>						38		22							39										27	
<init>		23	23			23		23				23			23					24					23	
<init2>									26							26					26	26	26		25/26	26
<identif_list>																			30	31						
<expression_list>		35	35						34							34					34	34	34		34	34
<expression_list2>		37	37			37		37				37			37				36						37	
<call_fce>																41										
<value_list>									43								42				43	43	43		43	
<value_list2>																	45		44							
<value>									50												46	47	48		49	
<type>							51			52			53													
<assigment>									32							32					32	32	32		32/33	32
<expression>																										
<after_id>																28			29	29						

∞

Figure 3: LL – table created from LL(1) grammar
There are 2 places, where we can not decide only on LL(1) grammar, here we use semantics check for help