Brno University of Technology

Faculty of Information Technology

Formal Languages and Compilers

IFJ23 Language Compiler Documentation

Team xvinog00 — TRP

Zdeněk Němec	xnemec0d	4%
Nazar Neskoromnyi	xnesko00	0%
Sava Rakić	xrakic00	4%
Alina Vinogradova	**xvinog00**	**92%**

December 6, 2023

Contents

1 Introduction

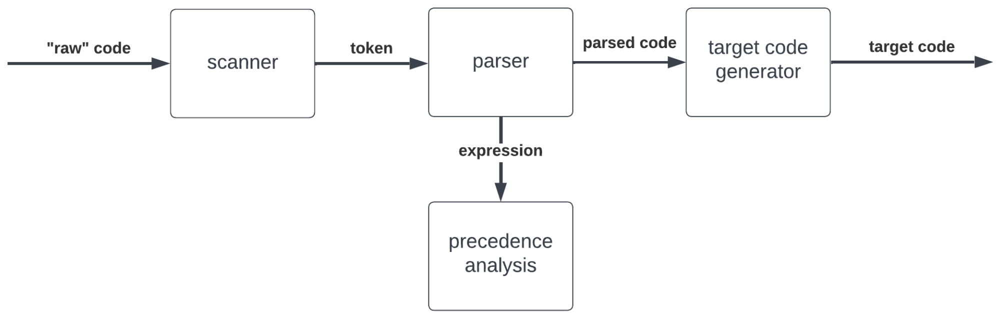
The goal of the project was to design and develop a compiler for the IFJ23 language, which is a subset of the Swift language. Separate modules of the solution are a scanner (tokenization and lexical analysis), a parser (syntactic, semantic, and precedence analysis), and a target code generator.

2 Implementation

This chapter will describe the functionality of the individual modules of our solution, as well as how these modules communicate, i.e. the overall structure of the project.

2.1 General structure

The structure of the project is shown in this diagram:



The main part is the parser, through which all input code passes to check for syntactic and semantic rules, after which the data goes to the generator to generate the target code

2.2 Tokenization and lexical analysis

The implementation of this module is located in scanner files. The main function of the scanner is get_token, which takes as a parameter a pointer to an existing token that is created in the parser before calling this function.

This function works as follows: it reads 1 symbol using `getc` of the built-in C function and analyzes this symbol with the help of a deterministic finite automaton diagram drawn up for it. In a switch statement inside this function, the scanner checks whether a lexical error occurred while reading the code or not, and constructs tokens as well. Further, if an error occurs, this function returns LEXICAL_ERROR. If the token was successfully read, the function fills the token passed as a parameter with the required data and returns NO_ERRORS

2.3 Syntactic and Semantic analysis

For syntactic analysis, an LL(1) grammar was designed, which means it is sufficient to read a single token to select a particular rule. After selecting a rule, we recursively step through the rule, checking the order and presence of the correct terminals. The implementation is located in the parser files.

While analyzing incoming terms, we check the program's semantics. For this purpose, a hash table data structure is implemented, which represents the symbols table of the program. More details about its implementation will be described in the sections below.

2.3.1 Stack of symtables

In addition to the standard symbol table, a table data stack was implemented in symbstack files to represent different scopes in different program parts. The global symbol table is always accessible from any part of the program — it stores global variables and function definitions. A stack is used to distinguish between variable definitions at different, sometimes nested levels of a program. A stack is used to distinguish between variable definitions at different, sometimes nested levels of a program.

2.3.2 Function calls analysing

One of the interesting parts of the solution is the implementation of function call analysis. The specification implies that a function call does not necessarily have to be lexically after its definition. It follows that to properly analyze the semantics of the call, we should save the calls to each function and, later, when we reach the end of the file before we finish the analysis, we have to evaluate these calls and check if the given function has been hyphenated or not and, if so, check the semantics of the parameters.

Our solution is to store function calls in the symbol table, more specifically in the structure of the call (`parser_func_call_t`) in the symbol attributes of the function type symbol and to introduce an additional indicator that indicates if the function has been defined or not (`func->isDefined`). Because of this, we can control the semantics of function calls in all cases.

2.4 Precedence analysis

To solve the problem that expressions are quite difficult to express and process using LL(1) grammars, a separate submodule responsible for analyzing expressions arriving at the input was implemented (expr files). We also implemented a stack data structure (files precstack) with which we analyze incoming tokens.

The process is as follows: we step by step convert incoming tokens into symbols, which we analyze using the precedence table. Based on this analysis, we further reduce the incoming expression, with data type control, and with the final data type determined.

2.4.1 Different types of expressions

The grammar of the IFJ23 language does not imply any clear separation between individual statements (as, for example, `;` in C), which has become a difficulty in the development of precedence analysis. The solution is "recursive" logic - **the end of each statement is always the beginning of the next one**. For this purpose, an additional variable `origin` was created, which is specified by the parser when switching to precedence analysis. In total, three cases were detected - an expression from the return, an expression from if and while conditions, and an ordinary, e.g. mathematical expression. The variable `origin` allows us to define one of the three cases listed above, which allows us to decide definitively what will be the end of the expression.

2.5 Target code generator

Due to various time and team communication difficulties, this part was not implemented in our solution.

3 Data structures

This section will describe several different additional data structures that are implemented and used in our solution.

3.1 Symbols table

This data structure is used to help us save the data during the analysis. The basis of our table was taken from last year's IFJ22 project.[1]

3.1.1 Implementation

The table is implemented using a hash table data structure with `djb2` hash algorithm in files symtable.[2] In addition to last year's implementation, the following changes have been made:

- Functions `symt_add_func` and `symt_add_var` are implemented and used instead of a single `symt_add_symb` for better usage and control over every symbol we are trying to add into the table.
- Implemented functions for saving function calls, as well as for adding parameters to these calls.
- The general structure of symbol attributes (variables and functions) has been changed for the subsequent saving of data required for the IFJ23 language.

The method for resolving collisions in the hash table has also changed. If such a situation occurs, open-addressing with linear probing with an interval set to 1 is used to find the nearest free position for the same hash code.

3.1.2 Usage

In our solution, we're saving all the necessary data into different tables, depending on the current scope (global one, local, nested in another local, etc.) Using this method we can easily do the semantics control for different types of redefinitions, data type matches, etc.

3.2 Stacks

There are multiple stack implementations used in our solution for different parts of the project.

3.2.1 Stack of symbols tables

To solve the problem with different scopes during program analysis, a stack structure was implemented, each element of which is a separate local table. The stack consists of four basic functions (`symbstack_init`, `symbstack_free`, `symbstack_push`, `symbstack_pop`) and one additional one related to searching for items in existing local tables, starting with the deepest scope, going up to the highest local one. Implementation of this data structure is located in symbstack files.

3.2.2 Stack of precedence symbols

Implementation of this stack was taken from the IFJ22 project [3]. Similar to the stack of tables logic, it also has five basic functions (`prec_stack_init`, `prec_stack_free`, `prec_stack_is_empty`, `prec_stack_push`, `prec_stack_pop`, `prec_stack_head`) and two specifics for our solution functions: `prec_stack_is_empty` checking if there's `NONTERM` at the head of the stack and at the same time first terminal in stack is `EMPTY` and `prec_stack_first_terminal`, that returns first terminal met in given stack. In the source code, this submodule is located in precstack files.

4 Team performance

At the beginning of the semester, the team chose a course of work that we tried to stick to throughout the semester. Unfortunately, we were unable to achieve clear communication and adherence to established task terms.

Team Member	Assigned Work
Alina Vinogradova	Lexical, Syntactic, Semantic, and Precedence analysis; Data structures; Testing; General Project Structure; Team Management; Tasks distribution DFA diagram; LL-grammar; LL-table; Precedence Table Documentation
Zdeněk Němec	Debug helper header file
Nazar Neskoromnyi	—
Sava Rakić	Generator design

Table 1: Work distribution in the team

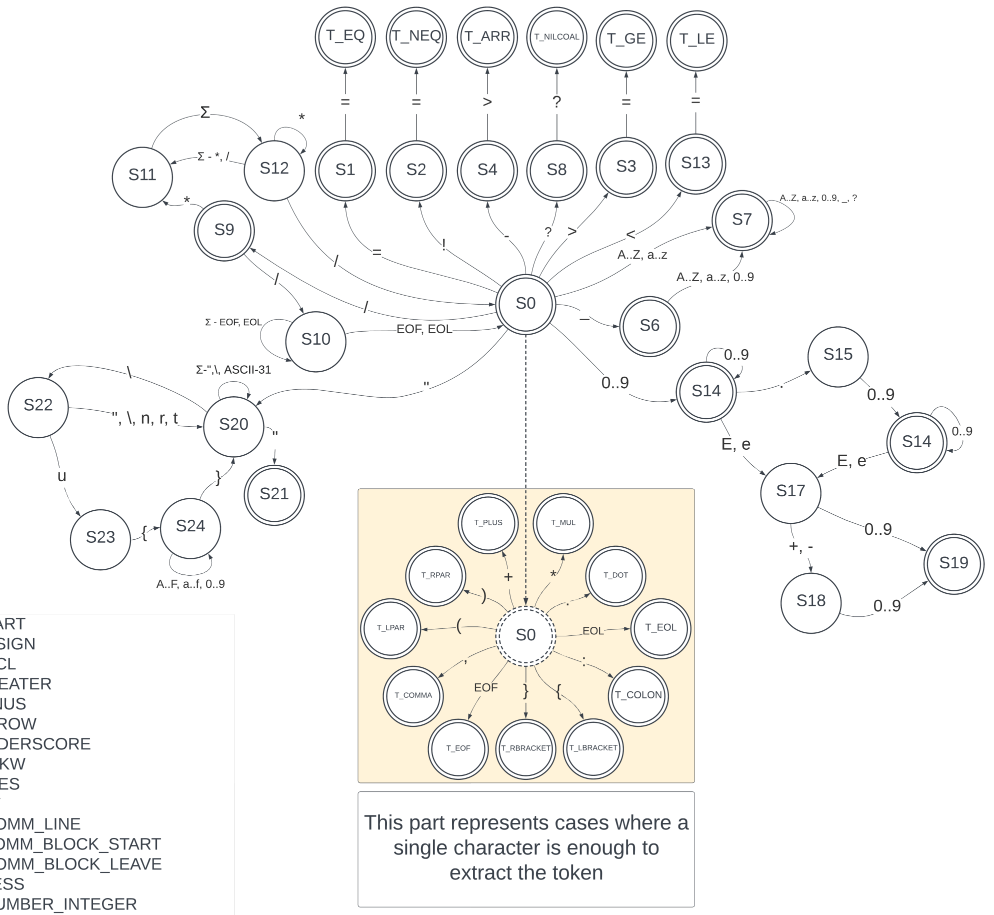
References

[1] Alina Vinogradova. *IFJ22 symbols table implementation.* URL: `https://github.com/jsemaljaa/ifj22-compiler/blob/main/src/symtable.c`.

[2] *Hash algorithms.* URL: `http://www.cse.yorku.ca/~oz/hash.html`.

[3] Alina Vinogradova. *IFJ22 precedence symbols stack implementation.* URL: `https://github.com/jsemaljaa/ifj22-compiler/blob/main/src/symstack.c`.

A DFA Diagram



S0	STATE_START
S1	STATE_ASSIGN
S2	STATE_EXCL
S3	STATE_GREATER
S4	STATE_MINUS
S5	STATE_ARROW
S6	STATE_UNDERSCORE
S7	STATE_ID_KW
S8	STATE_QUES
S9	STATE_DIV
S10	STATE_COMM_LINE
S11	STATE_COMM_BLOCK_START
S12	STATE_COMM_BLOCK_LEAVE
S13	STATE_LESS
S14	STATE_NUMBER_INTEGER
S15	STATE_NUMBER_DOUBLE_START
S16	STATE_NUMBER_DOUBLE_END
S17	STATE_NUMBER_EXP_START
S18	STATE_NUMBER_EXP_SIGN
S19	STATE_NUMBER_EXP_END
S20	STATE_STRING_START
S21	STATE_STRING_END
S22	STATE_STRING_ESCAPE
S23	STATE_STRING_SEQ
S24	STATE_STRING_SEQ_HEX

Figure 1: Deterministic Finite Automata

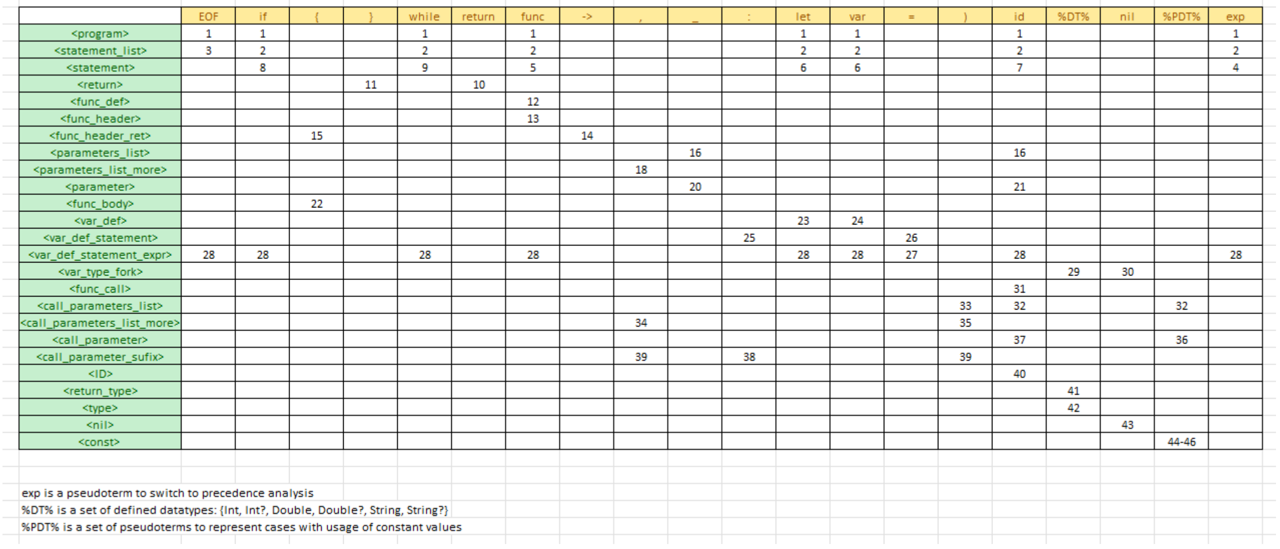
B LL grammar

1. `<program> ::= <statement_list>`
2. `<statement_list> ::= <statement> <statement_list>`
3. `<statement_list> ::= EOF`
4. `<statement> ::= <expression>`
5. `<statement> ::= <func_def>`
6. `<statement> ::= <var_def>`
7. `<statement> ::= <func_call>`
8. `<statement> ::= if <expression> { <statement_list> } else { <statement_list> }`
9. `<statement> ::= while <expression> { <statement_list> }`
10. `<return> ::= return <expression>`
11. `<return> ::=` ε
12. `<func_def> ::= <func_header> <func_body>`
13. `<func_header> ::= func <ID> ( <parameters_list> ) <func_header_ret>`
14. `<func_header_ret> ::= -> <return_type>`
15. `<func_header_ret> ::=` ε
16. `<parameters_list> ::= <parameter> <parameters_list_more>`
17. `<parameters_list> ::=` ε
18. `<parameters_list_more> ::= , <parameter> <parameters_list_more>`
19. `<parameters_list_more> ::=` ε
20. `<parameter> ::= _ <ID> : <type>`
21. `<parameter> ::= <ID> _ : <type>`
22. `<func_body> ::= { <statement_list> <return> }`
23. `<var_def> ::= let <ID> <var_def_statement>`
24. `<var_def> ::= var <ID> <var_def_statement>`
25. `<var_def_statement> ::= : <var_type_fork> <var_def_statement_expr>`
26. `<var_def_statement> ::= = <expression>`
27. `<var_def_statement_expr> ::= = <expression>`
28. `<var_def_statement_expr> ::=` ε
29. `<var_type_fork> ::= <type>`
30. `<var_type_fork> ::= <nil>`

31. `<func_call> ::= <ID> ( <call_parameters_list> )`
32. `<call_parameters_list> ::= <call_parameter> <call_parameters_list_more>`
33. `<call_parameters_list> ::=` ε
34. `<call_parameters_list_more> ::= , <call_parameter> <call_parameters_list_more>`
35. `<call_parameters_list_more> ::=` ε
36. `<call_parameter> ::= <const>`
37. `<call_parameter> ::= <ID> <call_parameter_suffix>`
38. `<call_parameter_sufix> ::= : <ID>`
39. `<call_parameter_sufix> ::=` ε
40. `<ID> = id`
41. `<return_type> ::= <type>`
42. `<type> ::= Int | Int? | Double | Double? | String | String?`
43. `<nil> ::= nil`
44. `<const> ::= type_int`
45. `<const> ::= type_double`
46. `<const> ::= type_string`
47. `<expression> ::= prec_expr`
48. `<expression> ::= <func_call>`

Using `<const>` as a pseudo term for const values
In `<expression>` switching to the precedence analysis

C LL Table



	EOF	if	{	}	while	return	func	->	,	_	:	let	var	=	)	id	%DT%	nil	%PDT%	exp
<program>	1	1			1		1					1	1			1				1
<statement_list>	3	2			2		2					2	2			2				2
<statement>		8			9		5					6	6			7				4
<return>				11		10														
<func_def>							12													
<func_header>							13													
<func_header_ret>			15					14												
<parameters_list>										16						16				
<parameters_list_more>									18											
<parameter>										20						21				
<func_body>			22																	
<var_def>												23	24							
<var_def_statement>											25			26						
<var_def_statement_expr>	28	28			28		28					28	28	27		28				28
<var_type_fork>																	29	30		
<func_call>																31				
<call_parameters_list>															33	32			32	
<call_parameters_list_more>									34						35					
<call_parameter>																37			36	
<call_parameter_sufix>									39		38				39					
<ID>																40				
<return_type>																	41			
<type>																	42			
<nil>																		43		
<const>																			44-46	

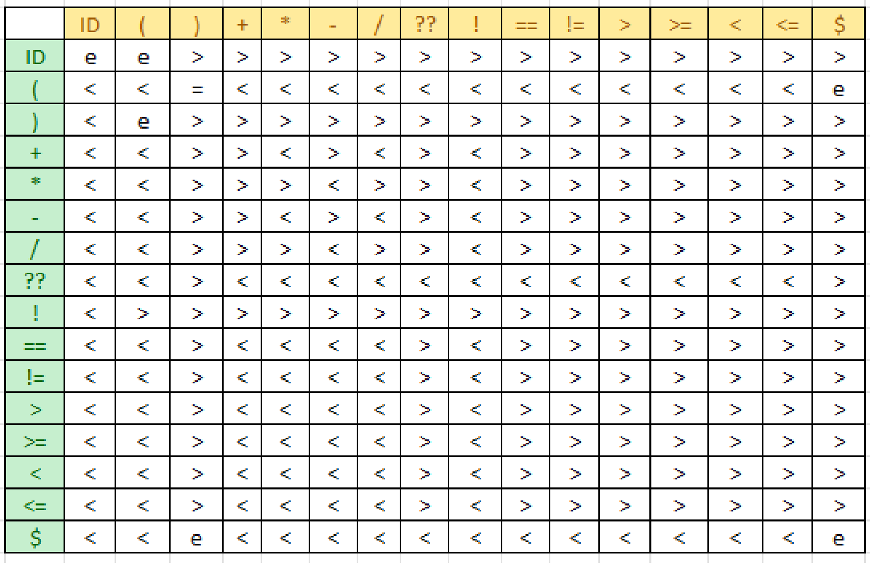
exp is a pseudoterm to switch to precedence analysis

%DT% is a set of defined datatypes: {Int, Int?, Double, Double?, String, String?}

%PDT% is a set of pseudoterms to represent cases with usage of constant values

Figure 2: LL Table

D Precedence Table



	ID	(	)	+	*	-	/	??	!	==	!=	>	>=	<	<=	$
ID	e	e	>	>	>	>	>	>	>	>	>	>	>	>	>	>
(	<	<	=	<	<	<	<	<	<	<	<	<	<	<	<	e
)	<	e	>	>	>	>	>	>	>	>	>	>	>	>	>	>
+	<	<	>	>	<	>	<	>	<	>	>	>	>	>	>	>
*	<	<	>	>	>	<	>	>	<	>	>	>	>	>	>	>
-	<	<	>	>	<	>	<	>	<	>	>	>	>	>	>	>
/	<	<	>	>	>	<	>	>	<	>	>	>	>	>	>	>
??	<	<	>	<	<	<	<	<	<	<	<	<	<	<	<	>
!	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>
==	<	<	>	<	<	<	<	>	<	>	>	>	>	>	>	>
!=	<	<	>	<	<	<	<	>	<	>	>	>	>	>	>	>
>	<	<	>	<	<	<	<	>	<	>	>	>	>	>	>	>
>=	<	<	>	<	<	<	<	>	<	>	>	>	>	>	>	>
<	<	<	>	<	<	<	<	>	<	>	>	>	>	>	>	>
<=	<	<	>	<	<	<	<	>	<	>	>	>	>	>	>	>
$	<	<	e	<	<	<	<	<	<	<	<	<	<	<	<	e

Figure 3: Precedence table

Precedence rules

1. `E -> id`
2. `E -> (E)`
3. `E -> E+E`
4. `E -> E*E`
5. `E -> E-E`
6. `E -> E/E`
7. `E -> E??E`
8. `E -> E!`
9. `E -> E==E`
10. `E -> E!=E`
11. `E -> E>E`
12. `E -> E>=E`
13. `E -> E<E`
14. `E -> E<=E`