Brno University of Technology
Faculty of Information Technology

Formal Languages and Compilers

IFJ2020

December 8, 2020

Dominik Horky xhorky32 25%
Lukas Hais xhaisl00 25%
Roman Janiczek xjanic25 25%
Jan Pospisil xpospi94 25%

Contents

1 Lexical analysis (scanner)

1.1 About

The first step of creating compiler for IFJ20 was implementing the scanner. The scanner is implemented as a *ravenous finite state automaton* (see Figure 4), which performs tokenization of input file. In `scanner.c` this is implemented as one big repetitive `while` loop which reads 1 character after another from `stdin` and jumps between predefined *states*, in switch case, in `scanner.h` by rules specified in automaton. If scanner reads an invalid character it stops it's action and returns `1`, which indicates error in lexical analysis, otherwise returns valid token, which is then processed in parser, and waits for another function call from parser to get another token from input.

1.2 Token

Token is predefined structure in `scanner.h` containing 2 core parts: *TokenType* & *TokenAttribute*

1.2.1 Token types

- EOF
- EOL
- Empty - when scanner encounters comment
- Identifier
- Keyword
- Integer
- String
- Float
- Boolean - with BOOLEAN extension
- Arithmetic operators (+, -, *, /)
- Relational operators (==, !=, >, <, >=, <=)
- Logical operators (&&, ||, !)
- Assignment operators (=, :=, +=, . . .) - some of which are supported only thanks to UNARY extension
- Brackets ((,), {, })
- Delimiters (comma, semicolon)

1.2.2 Token attribute

- String
- Integer
- Float
- Boolean - BOOLEAN extension
- Keyword - there are many predefined keywords in `scanner.h`, most of them are reserved keywords defined by IFJ20 specifications and `EMPTY` keyword when none of the above attributes should be used.

1.3 Usage

The main function of scanner will be `getToken()`, which behavior is described in 1.1. The parser calls `getToken()` function which takes token as it's parameter. If the reading was successful, token is filled with valid information about what it represents and then returns from function. Function `getToken()` distinguishes between 4 return codes it can return:

- `0` - if reading was successful and token is returned
- `1` - if lexical error (determined by automaton)
- `2` - if given token equals to NULL

- `99` - if internal problem (ex. malloc failure)

If returned code does not equals to `0`, program should stop it's action, clean after itself and inform user about what happened.

2 Syntax analysis (parser)

2.1 About

Second most important part of our compiler is parser. Parser takes token stream from scanner and, depending on rules specified in LL table (see Figure 2), transforms them into Abstract Syntax Tree (AST), which in our case is Binary Search Tree (BST). In `parser.c` this is implemented as various functions which acts as LL rules (see. Figure 1). If token stream taken from parser has invalid syntax, parser stops it's action and returns `2`, which indicates error in syntax analysis, also it can perform some necessities from semantic analysis and call `semantic_analysis()` from `semantic.c` file and `precedent_analyse()` function from `precedent.h`.

2.1.1 Syntax analysis - Top-Down parsing

For a syntax analysis we decided to use Top-Down parsing, which seemed optimal for our case, with recursive descent parsing. The top-down parsing performs construction of BST from root and then proceeds towards it's leaves.

2.1.2 Recursive Descent Parsing

This parsing technique consists of few function, one for each non-terminal in the grammar, and recursively parses the input to make a BST tree.

2.1.3 Parsing using precedent syntax analysis

Because some expressions contain arithmetic and logical operators, we have to check priority for those operators and evaluate correct order in which we perform those operations. Priority for those operators are specified by precedent table (see Figure 3). Precedent syntax analysis is not part of `parser` files but is implemented in standalone `precedent` files. At start it takes token from parser and then, one by one, gets next tokens and determines how to correctly evaluate them. After that they are generated to `stdin`.

2.2 Functions

- Parser - Main function of compiler, prepares symtable and stack for further usage, then calls `program`
- Program - Strips comments and EOLs and then calls `prolog`, `eolM` and `functionsBlock`
- Prolog - Checks whether file starts with `package main`
- EolM - Checks whether prologue is ended with EOL and then calls `eolR`
- EolR - Strips EOLs
- FunctionsBlock - Checks that token is KEYWORD `func` and calls `function` and `functionNext`, after that checks that `main` function was found only once
- Function - Checks that next token is IDENTIFIER, if so then it saves it's name to functions symtable. Then it looks for `(` and calls `arguments` func. If `arguments` was successful and current token is `)` it calls `functionReturn` and `commandBlock`
- FunctionNext - Checks if there are any other functions, if so, calls `function` and `functionNext`
- Arguments - Checks whether token is IDENTIFIER, if so calls `type` and `argumentNext`
- Type - Checks whether token equals to `int`, `float64`, `string` or `bool`
- ArgumentNext - If token is comma, then perform action as `arguments` function

- FunctionReturn - If token is `(` calls `functionReturnType`, otherwise return `0`, because return `()` could be omitted.
- FunctionReturnType - If token equals to KEYWORD, calls `type` and `functionReturnTypeNext` and then looks for `)`, if it equals to `)`, continue with program
- FunctionReturnTypeNext - Checks that token is COMMA, another token should be KEYWORD and calls `type` and `functionReturnTypeNext`
- CommandBlock - Checks that token is {, then gets another which should be EOL, if so strips remaining EOLs. If next token does not equal to }, calls `commands` and then checks that token equals to }, then look for one required EOL and strip remaining EOLs
- Commands - Calls `command`, checks that command is ended with EOL, strips remaining EOLs and then determines whether token equals to }, if so returns result, otherwise calls `commands`
- Command - Contains switch statement which determines next action. If token equals to IDENTIFIER calls `statement`. If token equals to KEYWORD go to another switch statement determining what kind of action we should perform based on passed KEYWORD:
  - IF - call `commandBlock` and `ifElse` functions.
  - FOR - call `forDefine` and then checks that token equals to `;`
  - RETURN - call `returnCommand`
  - default - invalid KEYWORD passed
- Statement - Contains another switch statement which distinguishes between those token types:
  - `(` - calls `arguments` function and checks that token equals to `)`
  - `=` - calls `assignment` function
  - `:=` - calls `assignment` function
  - `+=`, `-=`, `*=`, `/=` - calls `unary` function
  - default - calls `multipleID`, then checks that token equals to = and calls `assignment`
- MultipleID - Checks that token equals to COMMA, increases number of IDs, then checks that next token is IDENTIFIER and calls `multipleID`
- Assignment - Contains switch that distinguishes between those token types:
  - IDENTIFIER - checks that next token is `(`, calls `arguments` and checks that next token is `)`
  - default - calls `expressionNext` function
- Unary - Only checks that token is unary type
- ExpressionNext - Checks whether token equals to COMMA, then checks that number of IDs - 1 is greater than 0, then calls `expresssionNext` function
- IfElse - If token is KEYWORD and equals to ELSE then it calls `ifElseExpanded` function
- IfElseExpanded - If token equals to IF KEYWORD then calls `commandBlock` and `ifElse` functions, otherwise calls `commandBlock` only
- ForDefine - Checks that token's type equals to IDENTIFIER, if so checks whether next token equals to `:=`
- ForAssign - Determines whether token is IDENTIFIER and next token is =
- ReturnCommand - Checks that token is KEYWORD RETURN and calls `returnStatement`
- ReturnStatement - todo

2.3 Usage

The only function which program should call is `program()`, which calls other functions as it progresses through token stream (see 4.2). If no syntax error is found while parsing tokens, go through stack of symtables and calls semantic analyzer's function `semantic_analysis` (see 3). Parser is able to return various codes:

- `0` - if everything was successful
- `1` - if lexical error

- `2` - if syntax error
- `3` - if semantic error in program (undefined function, variable, ...)
- `4` - if semantic error in type assignment to new variable
- `5` - if semantic error in type compatibility (arithmetics, ...)
- `6` - if semantic error in program (invalid number of params or return values)
- `7` - if other semantic error
- `9` - if zero division
- `99` - if internal error
- `-1` - if internal warning

3 Semantic analysis

3.1 About

Semantic analysis is run after parser checks correct syntax of sequence of tokens and checks the semantic of it. For example valid data types, valid return types and existence of passed identifiers. Semantic analysis is performed right away in parser, but when we evaluate function call we use `semantic_analysis` function defined in `semantic.c` file.

3.2 Function

- Semantic_analysis - Check if is function is present in Global Stack, if so check if number of return values equals number of variables and then cycle through them and check their types, otherwise **TODO**.

4 Code Generator

4.1 About

Code generator's functions are called from parser when specific criterion is met and it generates *IFJcode20* code depending on what we are currently generating and it sends it to `stdout`. If everything were generated correctly we are able to send it to interpreter (which was written by our dear professors) and run it.

4.2 Functions

- GenerateHeader - generates header required by ic20int and jump to main function
- GenerateFunction - generates beginning of function or entire function code if it's internal function (if internal, label starts with _, otherwise `$`)
- GenerateFuncArguments - generates function's arguments and pops them from stack
- GenerateFuncCall - generates call for specified function
- GenerateFuncReturn - generates pushes of return values to stack
- GenerateFuncEnd - generates end of function (return to last instruction and frame pop)
- GenerateDefinitions - generates definition of variable
- GenerateAssignments - generates variable's assignment
- GeneratorPrint - generates internal function for printing, outputs 1 argument to `stdout`
- GenerateIfScope - generates `if`, `else if`, `else` scope
- GenerateForBeginning - generates counter variable definition
- GenerateForExpression - generates assignment + expression of `for` loop
- GenerateForCondition - generates condition for ending `for` loop
- GenerateForAssignment - generates label of `for` assignment
- GenerateForAssignmentEnd - generates jump to `for` expression (after assignment is done)
- GenerateForScope - generates beginning of `for` loop scope
- GenerateForScopeEnd - generates end of `for` loop scope

- IgnoreIfScope - determines whether `if`, `else if` or `else` scope will be ignored and not generated
- GeneratorSaveID - adds identifier to end of list for definition or assignment
- GeneratorGetID - gets last added identifier
- GenerateUsedInternalFunctions - generates used internal functions (if none, don't generate)
- GeneratorPrintCheck - check that printArguments is true, if so pop + print the value to stdout, otherwise do nothing
- GenerateCodeInternal - generates internal code (internal function code, header)
- SetUpCodeInternal - sets up code variable for generating internal function code
- IsFuncInternal - checks that given name is really internal function

4.3 Usage

In contrast with scanner or parser we don't call just one function here, which would then proceed to call other functions, but we have to call corresponding function to where we are currently in parser and what we need to generate. For that function names should be great indicator to know what to use.

5 Algorithms and data structures

Due to specifications in assignment we had to create some special data structures.

5.1 Binary Search Tree (BST)

We have implemented an Abstract Syntax Tree, which type is BST. Thanks to second homework from IAL, all of us had to implement this, so we scraped everything from those homeworks and created new file which suits our needs in this case. More about this could be found in `symtable` header and c file.

5.2 Symbol stack

We also need a stack for symbols to use in our compiler. It has few functions that performs *initialization, pushing, poping and freeing* the stack. For more detailed info see `symstack` files

5.3 Dynamic string

Because scanner needs to save characters to string, we are unable to use just plain C `char` data type, so we downloaded `str` header and c file from `jednoduchy_intrpreter` archive which was passed down upon us from our dear professors. This allowed us using it as dynamic string. We also tweaked this file a bit, so it suits our exact needs.

6 Teamwork

6.1 Versioning system

We decided that the best option for hosting our code and so will be using github repository, so we created one, set rules about merging to master and started developing in our own branches on parts of compiler that were assigned to us. Every time someone wanted to push some changes to master, he created *pull request* and other teammates had to do review on this code before merging.

6.2 Communication

For communication between teammates and few meetings, we decided for using Discord, where we created our own server. The decision was pretty simple, because all of us already have been using discord and were comfortable with it.

6.3 Who did what

- *Dominik Horky* - Team Leader, parser, generator, symtable, LL grammar, makefile
- *Roman Janiczek* - Parser, symtable, code review, github guru
- *Lukas Hais* - Scanner, scanner's automaton, code review, documentation, presentation
- *Jan Pospisil* - Precedent, expression, generator

7 Overview

7.1 Compiling

We have decided to write our own makefile, which we used for compiling our source code, because none of us had experience with using *CMake* and also it was specified in assignment to be able to compiler our program with `make` command in terminal. We also added some other parts as `make debug` used when debugging code, `make tests` which make tests for *symtable* and `make clean` that deletes everything that other `make *` commands created.

7.2 About project

This project seemed at first as too big bite to swallow, but as we progressed through it with blindly writing code, finding research materials online and watching IFJ/IAL lectures, we were able to slowly make some parts work. The main problem in our group was probably that we missed a lot of deadlines that we prepared for ourselves throughout time we had until final deadline, so in the end we found ourselves with little to no time and a few broken things either not working correctly or not working at all, but probably performed well in the end.

```
1.    <program> → <prolog> <eol_m> <functions>
2.    <prolog> → package main
3.    <eol_m> → EOL <eol_r>
4.    <eol_r> → EOL <eol_r>
5.    <eol_r> → eps
6.    <functions> → <func> <function_n>
7.    <function_n> → <func> <function_n>
8.    <function_n> → eps
9.    <func> → func ID ( <arguments> ) <func_return> <cmd_block>
10.   <arguments> → ID <type> <arguments_n>
11.   <arguments_n> → , ID <type> <arguments_n>
12.   <arguments_n> → eps
13.   <type> → int
14.   <type> → float64
15.   <type> → string
16.   <type> → bool
17.   <arguments> → eps
18.   <func_return> →  ( <f_type> )
19.   <f_type> → <type> <r_type_n>
20.   <f_type> → eps
21.   <r_type_n> → , <type> <r_type_n>
22.   <r_type_n> → eps
23.   <func_return> → eps
24.   <cmd_block> → { EOL <commands> }  EOL
25.   <commands> → <cmd> EOL <commands>
26.   <commands> → eps
27.   <cmd> → ID <statement>
28.   <statement> → <id_mul> = <assignment>
29.   <id_mul> → , ID <id_mul>
30.   <id_mul> → eps
31.   <assignment> → <expression> <expr_n>
32.   <expr_n> → , <expression> <expr_n>
33.   <expr_n> → eps
34.   <assignment> → ID ( <arguments_fc> )
35.   <statement> → ( <arguments_fc> )
36.   <statement> → := <expression>
37.   <statement> → <unary> <expression>
38.   <unary> → +=
39.   <unary> → -=
40.   <unary> → *=
41.   <unary> → /=
42.   <arguments_fc> → <expression> <expr_n>
43.   <arguments_fc> → eps
44.   <cmd> → if <expression> <cmd_block> <if_else>
45.   <if_else> → else <if_else_st>
46.   <if_else> → eps
47.   <if_else_st> → <cmd_block>
48.   <if_else_st> → if <expression> <cmd_block> <if_else>
49.   <cmd> → for <for_definition> ; <expression> ; <for_assignment> <cmd_block>
50.   <for_definition> → ID := <expression>
51.   <for_definition> → eps
52.   <for_assignment> → ID <id_mul> = <assignment>
53.   <for_assignment> → eps
54.   <cmd> → <return_cmd>
55.   <return_cmd> → return <return_stat>
56.   <return_stat> → <expression>
57.   <return_stat> → eps
58.   <return_cmd> → eps
```

Figure 1: LL grammar

LL table:

	package	main	eol	func	id	(	)	,	int	float64	string	bool	{	}	=	:=	+=	-=	*=	/=	if	else	for	;	return	$
PROGRAM	1																									
PROLOG	2																									
EOL_M			3																							
FUNCTIONS				6																						
EOL_R			4	5																						
FUNC				9																						
FUNCTION_N				7																						8
ARGUMENTS					10		17																			
FUNC_RETURN						18							23													
CMD_BLOCK													24													
TYPE									13	14	15	16														
ARGUMENTS_N							12	11																		
F_TYPE							20		19	19	19	19														
R_TYPE_N							22	21																		
COMMANDS			25		25									26							25		25		25	
CMD			53		27																42		48		53	
STATEMENT						35		28							28	36	37	37	37	37						
ID_MUL								29							30											
ASSIGNMENT					34																					
EXPRESSION																										
EXPR_N			33					32					33													
UNARY																	38	39	40	41						
IF_ELSE			44																			43				
IF_ELSE_ST													45								46					
IF_ELSE_ST_N			47																			47				
FOR_DEFINITION					49																			50		
FOR_ASSIGNMENT					51								52													
RETURN_CMD			57																						54	
RETURN_STAT			56																							

Figure 2: LL grammar table

	+ -	* /	(	)	id	== !=	> >= < <=	$	|| &&	!
+ -	>	<	<	>	<	>	>	>	>	!
* /	>	>	<	>	<	>	>	>	>	!
(	<	<	<	=	<	<	<	!	<	<
)	>	>	!	>	!	>	>	>	>	!
id	>	>	!	>	!	>	>	>	>	!
== !=	<	<	<	>	<	!	!	>	>	<
> >= < <=	<	<	<	>	<	!	!	>	>	!
$	<	<	<	!	<	<	<	=	<	<
|| &&	<	<	<	>	<	<	<	>	>	<
!	!	!	<	>	<	>	!	>	>	<

Figure 3: Precedent table

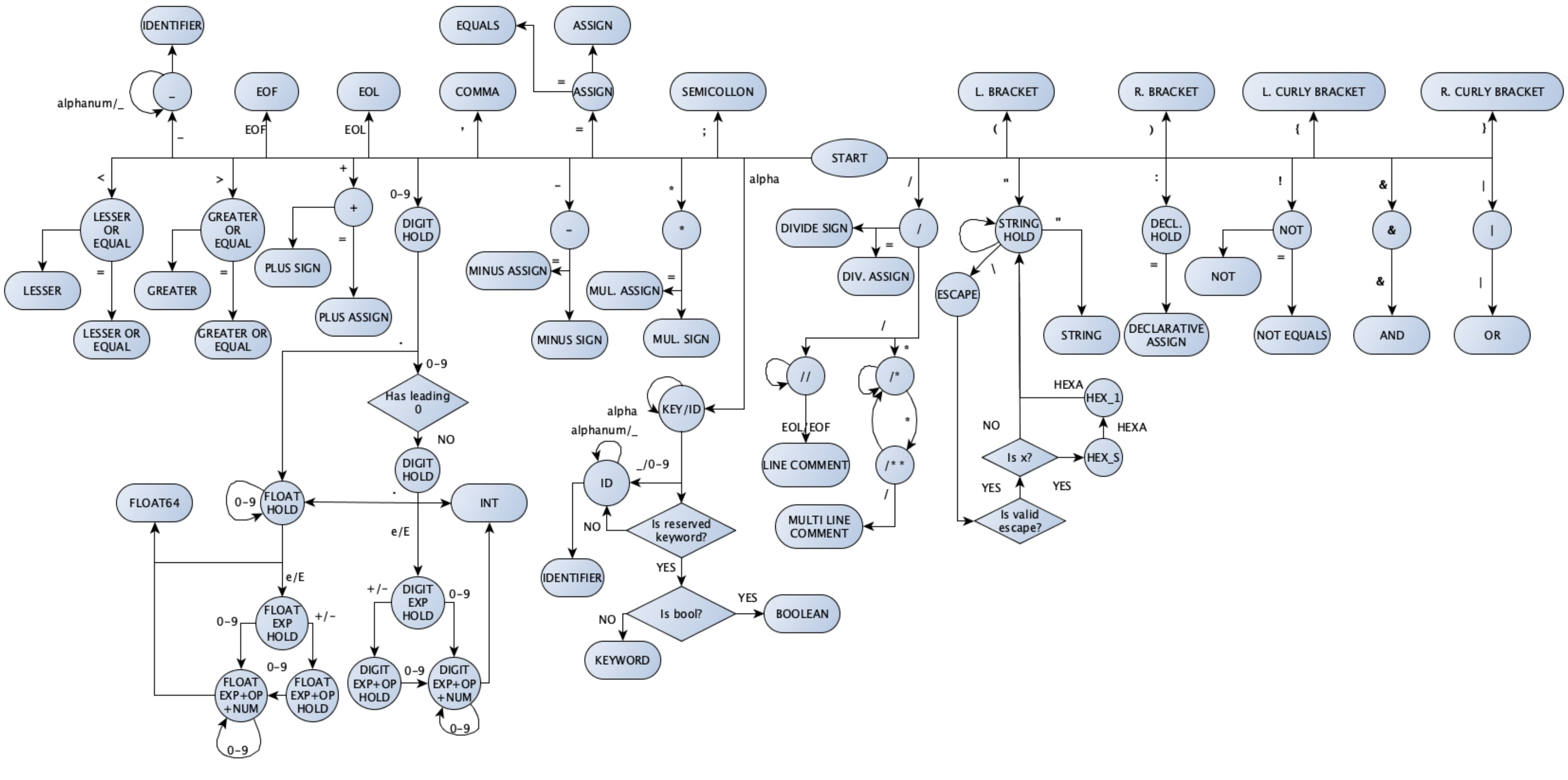

Figure 4: Scannerś automaton