
# Project IFJ/IAL
# Compiler for IFJ2019 language

Team 020, variant 2

Yadlouski Pavel (`Team Leader`): 35%
Korniienko Oleksii : 35%
Aghayev Raul : 30%
Bělohlávek Jan : 0%

Brno, 2019

# Contents

**1 Introduction** **2**

**2 Implementation description** **2**
2.1 Lexical analyze . . . 2
2.2 Syntax and semantic analyze . . . 2
2.2.1 Precedence analyzes . . . 2
2.3 Target code generation . . . 3
2.3.1 Generation start . . . 3
2.3.2 Asserting functions . . . 3
2.3.3 IF condition . . . 4
2.3.4 While loop . . . 4

**3 Inner structures and auxiliary modules** **4**
3.1 Hash table . . . 4
3.2 Stack . . . 4
3.3 Dynamic string . . . 4
3.4 Double-way linked list . . . 5

**4 Ideas for optimization** **5**

**5 Team work** **5**

# 1 Introduction

Our task was to implement compiler for IFJ2019 language. That language is subset of Python3 language. Target language is IFJcode19.

# 2 Implementation description

## 2.1 Lexical analyze

We started implementation from lexical analyzes (scanner) in case that without this module whole compiler would work at all. Scanner is implemented as finite state machine(2). It read character by character from *STDIN* and this character controls the state of machine. *Token* is implemented as structure that holds type of token and attribute of token if it exist. *Attribute* is union that depend on token type contain integer/float value, string (also name of IDs and functions) or type of key word (*_WHILE_*, *_DEF_*, etc.).

Our automate can accept all mentioned tokens from project specification and some additional for more comfortable operations. For example we add *TOKEN_INDEND* and *TOKEN_DEDEND* for LL-grammar needs. Also for text block and one-line sting we use the same token *TOKEN_STRING*. Both of them support hexadecimal codes of characters, that is converted to actual ASCII character. For more convenience with processing numbers, strings and other use cases we implement *dynamic string*(3.3), where we write down characters and then process this string. When name of function or key word is readied, then it written to dynamic string. After that this string compered with reserved words in IFJ2019 language and decided if it is one of key words or identifier.

## 2.2 Syntax and semantic analyze

The syntactic analyzer (*parser*) is implemented using the *Recursive descent* method based on *LL-grammar* (5) in the *LL-table* (1). For each rule described in the *LL-grammar*, there is a case in switch (type of token) that decides which of the rules will be analyzed later. Using the *get_token* function, the syntactic analyzer starts the lexical analyze, which returns the token by checking it for lexical errors beforehand. In order to return the correct code in case of an error, we use the *ret_code* variable, into which the return codes of all called functions inside the program are written. Thus, if after calling some function *ret_code* turned out to be not equal to 0, then we return this *ret_code* to the called function.

In order to avoid errors with the redefinition of variables in the loop, a *dynamic string* (3.3) and the *flag_while* variable were created, which increased by 1 when we entered the cycle and decreased by 1 when the cycle ended. If *flag_while* was not equal to 0 during code generation, then the entire target code, except for the definition of variables, was written to the dynamic string, and after the loop ended it was displayed.

### 2.2.1 Precedence analyzes

Fro processing expressions *precedence analyzes* is used. It implemented in separate files *preced_analyze.c/.h* and parser call this analyzes by default after token *TOKEN_ASSIGN*, after key_word *if* and *while*. Then this analyze get next tokens until token *TOKEN_EOL*.

For getting accurate cell in table, we convert actual token to column and upper terminal from stack to row number. Then based on symbol in cell symbol is aggregated: push on stack or reduced by rule (using function *reduce_rule*).

Token *None* in our compilation we processed in a special way. In precedence analyses it work same as integer value, but when turn came to code generation, we take top element from stack. After that, type of element is taken. Then based on this type equivalent element with *None* of appropriate type is written to the top of stack. After that in expression like $a! = None$ in target code *None* will be correctly compared with any type of ID.

## 2.3 Target code generation

Files *codgen.c* and *codgen.h* contains functions for generating *IFJcode19* code. All of the functions are called from parser or precedence analysis and from main function during the execution of the program. Program generates all type of functions, that means for all frames, stack conditions, and any kinds of modifications and calls functions out of other functions.For code generating functions program requires:

1. Input tokens for analyzing and performing appropriate actions(for example when we are printing some value, we need to know what type of value (in *token->type*) should we print

2. Frames for working with appropriate frame

3. Counters for number of the function variables(due to the fact that our program does not create new variables every time, we use the same variable, the only change is the last digit, that increases its amount every time we recall the function by one) or number of if's conditions and while's cycles(in case if there is while cycle in while cycle we need to know where should we return).

4. String that will work as a helper for a while cycles. Due to the fact that there should be only one definition of a variable in the cycle, we use *dynamic string* for holding everything that while cycle contains(even in case we have more than 1 while cycle in while cycle),except the definitions of the function, after that code is directly generated on standard output (without the definition). This method helped us to solve one of the most interesting problems we had during the creation of the program.

### 2.3.1 Generation start

At the beginning program generates jump to main and all of inserted functions (*len, inputf, inputs, inputi, substr, ord, chr*).All of this functions will be defined at beginning of output, and could be called on request, straight after this definition we label the main, in order to skip the untimely definition of functions

### 2.3.2 Asserting functions

For asserting functions,(even in if and while) our compiler analyzes everything that should be assigned to a value by precedence analyzes, creates a variable for temporary storage(can be more than one), one by one sends value to a stack, then execute some operations(in case if it is needed) and returns back to stack, and this manipulations are repeated again until the end when this value is sent to a storage, this algorithm helps us to analyze extremely long expressions such as $(a = 10 + (c - d) + d/10 * 12...)$, and after this the output value can be assigned to any ID we need, in time when operations are calculated program also checks for any kind of mistakes, and converts values to float or int in case if it needed.

### 2.3.3 IF condition

*If* conditions work is comparing two of variables from the stack(also we have added checks of types for avoiding incompatible types of operands) and according to output of comparison do execute output (call appropriate function). Program can compare endless amounts of variables in the left and in the right side, all of this variables will be declared and used later.

### 2.3.4 While loop

For generating *while* loop's program uses a unique technology, the reason of that is defining variables in while cycles. Computing of conditions uses the same technology as if or asserting functions. Code generator sends everything that while function contains in a string so that program can send it to output later (but the point is that program does not execute the repetition of defining the variable more than once for while due to the fact that there is not such an implementation for defining).

# 3 Inner structures and auxiliary modules

## 3.1 Hash table

Implementation of *hash table* we took from *IAL homework* and extend it for our needs. First of all item of hash table is extended with following elements:

- *type* – type of token stored in hash table (variable or function)
- *param_ count* – count or function parameters
- *is_ declared* – flag shows if ID/function was declared

Then we decide to link global and local hash tables to linked list, so local table point to "parent" global table. Also we implement function for looking through all linked tables.

## 3.2 Stack

Implementation of *stack* we took from *IAL homework*. It is used for counting INDENDs and DEDENDs

## 3.3 Dynamic string

For more convenient processing characters we implement *dynamic string*. It is a structure with:

- *str* – array of characters
- *len* – count of characters in array
- *size* – size of allocated memory for string

Also it is used for writing down output text in some cases. For example in while loop we write all code to dynamic string and then put it to the right place.

### 3.4 Double-way linked list

This module is used only in precedence analyzes. Initially we used stack, but over time, we began to seem that using *double-way linked list* for purposes of precedence analyzes in our implementation is better then one-way linked list because with this type of stack we can check elements that stack contains. This advantage is used for searching top terminal in stack.

## 4 Ideas for optimization

During developing process we got some ideas for optimization, but in case that we didn't have so much time to implement they are not implemented in out project.

First optimizations is that we might implement analyze of static expressions. It means, that if statement contain only static values (for example $a = 10 + 20.5$), then precedence analyzes could process this expression in it inner algorithms and write down to the target code only needed conversions and similar functions that are needed only in specific situations.

## 5 Team work

Our team began work on project in middle of October. There we set job distribution. For our project we used Git version control with GitHub.

| | |
|---|---|
| Pavel | Lexical analyze, semantic analyze, hash table, dynamic string, stack, double-linked list, FSM, precedence table, team lid, GitHub support, documentation |
| Oleksii | Syntax analyzes, lexical analyzes (string block), code generation, LL-grammar, LL-table, documentation (semantic + syntax analyzes) |
| Raul | Code generation, documentation (code generation) |
| Jan | Nothing |

During development process one member left our team.

# Abstract

## LL-grammar

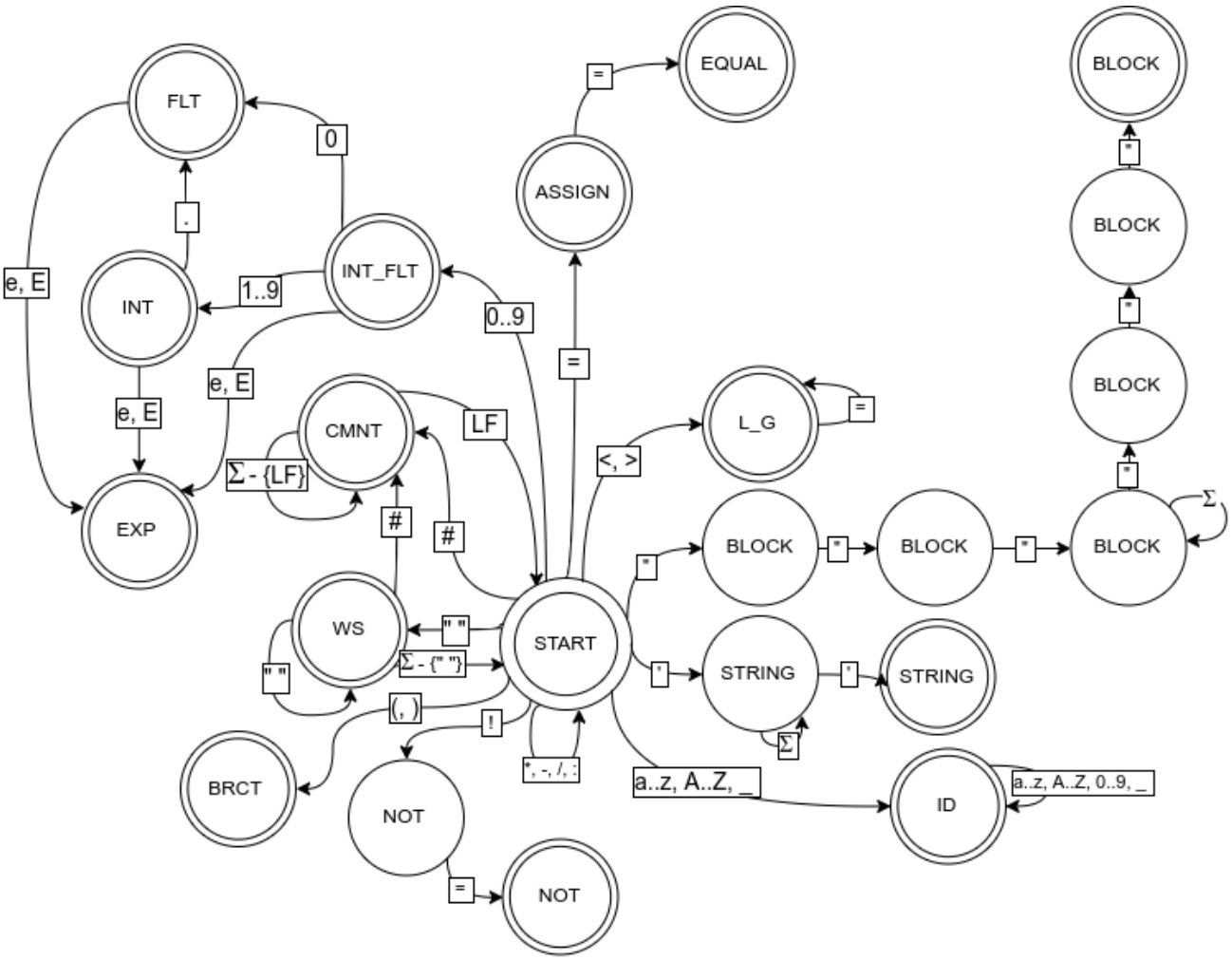
1. <prog> -> EOF
2. <prog> -> ID = <value/func/expr> <prog>
3. <prog> -> FNC ( <arg> <arg_mb> ) <prog>
4. <prog> -> if <cond_mb> : EOL INDENT <prog> DEDENT <prog>
5. <prog> -> return <mb_ret> EOL
6. <prog> -> while <cond_mb> : EOL INDENT <prog> DEDENT <prog>
7. <prog> -> else : EOL INDENT <prog> DEDENT <prog>
8. <prog> -> def FNC ( <arg> <arg_mb> ) : EOL INDENT <prog> DEDENT < prog>
9. <prog> -> EOL <prog>
10. <prog> -> e
11. <prog> -> e <prog>
12. <cond_mb> -> ( <cond_mb> )
13. <cond_mb> -> <func/expr>
14. <mb_ret> -> None
15. <mb_ret> -> <func/expr>
16. <func/expr> -> FNC ( <arg> <arg_mb> )
17. <func/expr> -> <expr>
18. <value/func/expr> -> <expr>
19. <value/func/expr> -> FNC ( <arg> <arg_mb> )
20. <value/func/expr> -> None
21. <i/f/s> -> int
22. <i/f/s> -> float
23. <i/f/s> -> string
24. <arg> -> e
25. <arg> -> ID
26. <arg> -> <i/f/s>
27. <arg_mb> -> ,<arg> <arg_mb>
28. <arg_mb> -> e

| | if | while | else | def | FNC | ID | NONE | Return | pass | EOL | String | Int | Float | EOF | INDEND | DEDEND | ( | ) | , |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| <prog> | 4 | 6 | 7 | 8 | 3 | 2 | | 5 | 11 | 9 | | | | 1 | | 10 | | | |
| <cond_mb> | | | | | 13 | 13 | | | | | 13 | 13 | 13 | | | | 12 | | |
| <mb_ret> | | | | | 15 | 15 | 14 | | | 14 | 15 | 15 | 15 | | | | 15 | | |
| <func/expr> | | | | | 16 | 17 | | | | | 17 | 17 | 17 | | | | 17 | | |
| <value/fnc/expr> | | | | | 19 | 18 | 20 | | | | 18 | 18 | 18 | | | | 18 | | |
| <i/f/s> | | | | | | | | | | | 23 | 21 | 22 | | | | | | |
| <arg> | | | | | | 25 | | | | | 26 | 26 | 26 | | | | | 24 | |
| <arg_mb> | | | | | | | | | | | | | | | | | | 28 | 27 |

Table 1: LL-table

```
  +    -    *    /   //   <=  >=   <   >   =   !=  (    )    i    f    s   id   $    */
{'>', '>', '<', '<', '<', '>','>','>','>','>','>', '<', '>', '<', '<', '<', '<', '>'}, // +
{'>', '>', '<', '<', '<', '>','>','>','>','>','>', '<', '>', '<', '<', '<', '<', '>'}, // -
{'>', '>', '>', '>', '>', '>','>','>','>','>','>', '<', '>', '<', '<', '<', '<', '>'}, // *
{'>', '>', '>', '>', '>', '>','>','>','>','>','>', '<', '>', '<', '<', '<', '<', '>'}, // /
{'>', '>', '>', '>', '>', '>','>','>','>','>','>', '<', '>', '<', '<', '<', '<', '>'}, // //
{'<', '<', '<', '<', '<', ' ',' ',' ',' ',' ',' ', '<', '>', '<', '<', '<', '<', '>'}, // <
{'<', '<', '<', '<', '<', ' ',' ',' ',' ',' ',' ', '<', '>', '<', '<', '<', '<', '>'}, // >
{'<', '<', '<', '<', '<', ' ',' ',' ',' ',' ',' ', '<', '>', '<', '<', '<', '<', '>'}, // <=
{'<', '<', '<', '<', '<', ' ',' ',' ',' ',' ',' ', '<', '>', '<', '<', '<', '<', '>'}, // >=
{'<', '<', '<', '<', '<', ' ',' ',' ',' ',' ',' ', '<', '>', '<', '<', '<', '<', '>'}, // !=
{'<', '<', '<', '<', '<', ' ',' ',' ',' ',' ',' ', '<', '>', '<', '<', '<', '<', '>'}, // ==
{'<', '<', '<', '<', '<', '<','<','<','<','<','<', '<', '=', '<', '<', '<', '<', ' '}, // (
{'>', '>', '>', '>', '>', '>','>','>','>','>','>', ' ', '>', ' ', ' ', ' ', ' ', '>'}, // )
{'>', '>', '>', '>', '>', '>','>','>','>','>','>', ' ', '>', ' ', ' ', ' ', ' ', '>'}, // i
{'>', '>', '>', '>', '>', '>','>','>','>','>','>', ' ', '>', ' ', ' ', ' ', ' ', '>'}, // f
{'>', '>', '>', '>', '>', '>','>','>','>','>','>', ' ', '>', ' ', ' ', ' ', ' ', '>'}, // s
{'>', '>', '>', '>', '>', '>','>','>','>','>','>', ' ', '>', ' ', ' ', ' ', ' ', '>'}, // id
{'<', '<', '<', '<', '<', '<','<','<','<','<','<', '<', ' ', '<', '<', '<', '<', ' '}  // $
```

Figure 1: Precedence table




START - SCANNER_START

WS - SCANNER_WHITE_SPACE

CMNT - SCANNER_COMMENT

BRCT - SCANENR_BRACET

INT_FLT - SCANNER_INT_OR_FLOAT

INT - SCANNER_INT

FLT - SCANNER_FLOAT

EXP - SCANNER_EXP

STRING - SCANNER_STRING

BLOCK- SCANNER_BLOCK_STRING

L_G - SCANNER_LESS_OR_GRATER

ID - SCANNER_ID

EQUAL - SCANNER_EQUAL

Figure 2: Finite-State Machine