VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Project Documentation

Implementation of a compiler for the imperative programming language IFJ20

9. 12. 2020

Team 128. variant II

BOOLTHEN, BASE, FUNEXP, MULTIVAL, UNARY

Mihola David	**xmihol00**	38 %
Foltyn Lukáš	xfolty17	30 %
Sokolovskii Vladislav	xsokol15	32 %

Contents

1 Introduction 3
1.1 Project description . . . 3
1.2 Project structure . . . 3

2 Architecture 4
2.1 Lexical Analysis . . . 4
2.2 Syntax Analysis . . . 6
2.3 Semantic Analysis . . . 6
2.4 Optimization . . . 7
2.4.1 Full and Partial Constant Expression Evaluation . . . 7
2.4.2 Constant Propagation . . . 8
2.4.3 Conditioned Function Call Optimization . . . 8
2.4.4 Conversion to Three Address Code . . . 9
2.4.5 Built-in Functions Inlining . . . 9
2.5 Code Generation . . . 9

3 Data Structures 10
3.1 Table of Symbols . . . 10
3.2 Memory Management Unit . . . 10
3.3 Vector . . . 11
3.4 Dynamic String . . . 11
3.5 Token Linked List . . . 11

4 Testing 11

5 Teamwork 12
5.1 Communication . . . 12
5.2 Workload Division . . . 12

6 Conclusion 12

A LL-Grammar 13
A.1 LL-Table . . . 17

B Precedence Table 18

C Finite State Automata 19

D Examples **22**

D.1 Hash Tables Design 22

D.2 Hash Tables Usage 23

D.2.1 IFJ20 Source Code Example 23

D.2.2 IFJ20 Source Code Explanation 23

D.2.3 Graphical Example 24

D.3 Optimization 25

D.3.1 IFJ20 Source Code Example 25

D.3.2 Generated IFJcode20 program 25

1 Introduction

1.1 Project description

This is a documentation of a team project that was implemented within "Formal Languages and Compilers" and "Algorithms" courses at VUT FIT. We were assigned with a task to create a program, formally known as compiler, that reads source code written in IFJ20 language and compiles it to the target IFJcode20 language. The IFJ20 language is a subset of Go language.

1.2 Project structure

The project consists of multiple separate modules. Each module implements specific functionality and provides other modules with a communication interface. The common pattern for each module is to being divided into three files: `{modul_name}.c`, `{modul_name}.h` and `{modul_name}_private.h` and having one global data structure, which starts with a `G_` prefix and continues with the module name.

Files with the `_private.h` suffix emulate encapsulation of the given module's private data structures and functions.

- `enums.h` File consisting of various enumeration definitions (f.e. the enumeration of token and error types), which are used by numerous modules throughout the whole program.
- `functions{.c,.h}` Files implementing generation of the IFJcode20 code of functions that are built-in to the IFJ20 language. For further information visit section Optimization.
- `generator{.c,.h,_private.h}` Files implementing the IFJcode20 code generation. For further information visit section Code Generation.
- `main{.c,.h}` Files containing the implementation of a main function of the program.
- `memory{.c,.h,_private.h}` Files implementing a memory management unit. For further information visit the section Data Structures.
- `optimizer{.c,.h,_private.h}` Files implementing the compiler intermediate code optimization. For further information visit section Optimization.
- `oRPN{.c,.h,_private.h}` Files containing the implementation of expression analysis and expression optimization. For further information visit sections Syntax Analysis and Optimization.
- `parser{.c,.h,_private.h}` Files containing the implementation of a syntax analyser based on the LL-grammar. For further information visit section Syntax Analysis.

- `register{.c,.h}` Files containing the implementation of auxiliary IFJcode20 variables used during expression calculation. For further information visit section Optimization.
- `scanner{.c,.h,_private.h}` Files consisting of the lexical analyser implementation. For further information visit section Lexical Analysis.
- `sematic{.c,.h,_private.h}` Files consisting of the semantic analyser implementation. For further information visit section Semantic Analysis.
- `symtable{.c,.h}` Files implementing the table of symbols. For further information visit section Data Structures.
- `token{.c,.h,_private.h}` Files containing implementation of the auxiliary functions used as an interface when working with the token stream represented as a single linked list. For further information visit section Data Structures.
- `vector{.c,.h,_private.h}` Files implementing the auxiliary functions used as an interface for working with abstract data types such as dynamic strings and various vectors. For further information visit section Data Structures.

2 Architecture

2.1 Lexical Analysis

Lexical analysis is the first phase of the compilation. The lexical analyser, also known as a scanner, was implemented in the `scanner` module. The core of the scanner is a deterministic finite state automata (FSM). The automata is implemented according to the automata state diagram. Based on the first read symbol at the start of each lexeme, the scanner decides whether the lexeme is going to be parsed directly or by an auxiliary functions representing given part of the FSM.:

- `number()` processes floating point numbers and integers in hexadecimal, octal or binary notation
- `convert_string()` processes any string literals which are bounded with double quotes
- `keyword_idf()` processes identifiers and determines whether the given identifier is a keyword by searching a table of keywords.
- `skip_comment()` ignores one-line and block comments.

We decided to divide the scanner in such a way due to the clarity of the code and simplification of debugging.

`number()` auxiliary function is extended by the implementation of BASE extension. Our implementation supports two notations of an octal number f.e. `0123` and `0o123`, therefore a number in octal base can start ether with `0` or `0o` sequence of characters.

Usage of `underscore` character is based on the compatibility with Go language. To ensure that underscore separates successive digit we use flag variable that is set to true after reading an underscore, then we check if in this particular situation underscore is a legal character, if not lexical error (1) occurs. During the lexical analysis all underscores are removed and number is passed to the parser where it is converted to the decimal base.

The parsing of one lexeme can either end up with a lexical error (1), or a valid token type is returned. If the given token has any data they will be stored in an auxiliary data structure. Both the token type and the token data are then passed to the syntax analyser where they are appropriately evaluated and finally the token is created. Scanner is called by the syntax analyser, the token types are integer values, which corresponds to the token types defined in the `enums.h` header file. Token types returned by the scanner:

- `keyword` (else, if, func etc.)
- `variable identifier`
- `function identifier`
- `operator` (+, /, == etc.)
- `string literal`
- `equal operator` (=, +=, -=, *=, /=, :=)
- `type` (string, float64, int, bool)
- `opening curly bracket` {
- `closing curly bracket` }
- `opening bracket` (
- `closing bracket`)
- `comma` ,
- `new line` '\n'
- `semicolon` ;
- `constant`

2.2 Syntax Analysis

Syntax analysis is the main phase of the compilation, also known as parsing. The parser is implemented by the method of non recursive predictive analysis guided by the LL-grammar table in the `parser` module. We chose this method by intuition in the beginning of the semester based on a little practical knowledge of automata from previous courses. At that time we were not familiar with any theory regarding the topic of formal languages and automata.

The parser is also implemented as finite state automata. It uses a state transition table, which is directly implemented in the source code as a matrix of function pointers. Each function represents one transition of the parsing automata. The called function is selected based on the incoming token type from the scanner and the current state of the automata. The sequence of called functions simulates the top-down parsing method based on the LL-grammar. To enable the parsing automata to correctly parse any given IFJ20 source code, it is supplemented by two stacks, which ensure correct application of some of the parsing rules.

- `"curly bracket" stack` holds an information of nested multi-line statements such as `if - else if - else` conditions and `for` cycles.
- `"expression" stack` holds an information about an ongoing expression type such as function parameter expression, assignment expression etc.

The parsing of an expression is implemented by the precedence syntax analysis according to the operator precedence table in the module `oRPN`. However, the precedence table is not present in the source code. The precedence of operators is encoded to the operators themselves by the scanner. As a result precedence of two operators can be obtained simply by comparing them.

As the parsing is performed, the parser enriches the token stream with auxiliary tokens used during the semantic analysis, optimization and target code generation.

2.3 Semantic Analysis

The semantic analysis of the IFJ20 source code is implemented in the modules `parser`, `oRPN` and `semantic`.

The module `parser` analyses whether variables used in expressions were defined or not. If not, error (3) is generated. Furthermore the module parser contains one additional co-automata, which evaluates, if a function returns correctly. The main parsing automata provides the co-automata with tokens representing start and end of `if`, `else if`, `else` statements, the `return` statement token, as well as the token representing function body end. As the last mentioned token is added, the evaluation ends. Error (6) is generated, if the function does not return correctly. Last but not least all called function are analysed, if they were also defined, at the end of the parsing process. If any function was not defined, error (3) is generated.

The semantic analysis of constant expressions is executed in the module `oRPN` as these expressions are evaluated.

The rest of the semantic analysis is performed in the module `semantic`. The non constant expressions are analysed with a simulation of their evaluation based on their data types. When a data type mismatch occurs, error (5) is generated. Error (5, 7) also occurs when the data type of the evaluated expression does not match the assigned variable or when there is a mismatch of the number of assigned variables. Similar error (6) occurs, when evaluated expression does not match the type of function parameter/return value or the number of assigned parameters/return values do not match.

2.4 Optimization

The optimization consists of several procedures relating to constant expression evaluation, constant propagation, conditioned function call optimization, conversion to three address code and built-in functions inlining.

2.4.1 Full and Partial Constant Expression Evaluation

The optimization of an expression is implemented in the module `oRPN`, which is also responsible for the correct expression parsing.

Each expression is separated into two parts. The separation is performed based on the arithmetic `plus`, `minus` and logic `or` operators. The first part consist only of constant values and can be directly evaluated during the compilation. The second part contains variables, function calls and constants, which cannot be separated. Furthermore the second part can be partially evaluated based on the order of incoming operators and position of constant values in a given subexpression.

For the number of different operator precedence levels (7), that we were assigned with, it is necessary to divide the calculation to three layers with additional constant accumulators. `L1 - L3` operators are evaluated in the first layer. `L4` operators are evaluated on all three layers. The first part of an `L4` subexpression is transferred to the second layer, as the operator is added to the expression. When an `L4` operator is evaluated based on its precedence, it proceeds its left operand from the second layer, where it was previously stored, and its right operand from the first layer, where it is currently evaluated. Then the result of an `L4` operator evaluation is stored in the third layer, which is used for evaluation of `L5 - L7` operators. Direct transition from the first to the third layer also must be arranged, to ensure correct evaluation of boolean subexpression, in which an `L4` operator is not present. Both the first and the third layer have one constant accumulator, where the results of constant subexpressions are accumulated. Additionally there is a stack of a limited size, where constant values can be temporarily stored before their evaluation.

2.4.2 Constant Propagation

The constant propagation optimization is implemented in the module `symtable`, which provides an interface for the modules `oRPN` and `parser`. When an expression evaluation ends with assign and the result is a constant value, the value with the corresponding type can be stored in the table of symbols to the given assigned variable. On the other hand, when a variable to which a constant value has been assigned previously is part of an expression, the value with its corresponding type is used instead.

A variable holds its constant value on a given scope until a new scope is created (i.e. before conditional and iterative statements), its value can be used in all directly succeeding scopes generated by one statement (i.e. `if - else if - else` statement). Variable can retain its constant value after such a statement only when its value has not been changed in any of the statement branches. Therefore all unassigned variables with constant value must be preemptively assigned before a statement of this kind. Furthermore constant values cannot be used inside iterative blocks, as their value can change multiple time based on the number of iterations.

2.4.3 Conditioned Function Call Optimization

The optimization of functional calls is implemented in the module `optimizer`, with the functions possible for optimization being labeled by the module `symtable`. There are two conventions of functional calls implemented.

The first convention is inspired by the x86 pascal calling convention, in which function parameters are pushed to the stack in a left-to-right order. Therefore the called function must assign its parameters in the opposite order. The return values of a function are again pushed on the stack in a left-to-right order and the callee assigns the return values the same way as a function its parameters. This calling convention is used with recursive function, which must create a new frame at each call.

The second calling convention is based on an idea, that creating new frame and defining function parameters at each function call is unnecessary. This convention can be used only for non-recursive functions. The callee is responsible for assigning values to all parameters of a called function. The parameters of non-recursive functions are defined globally. The return value is passed equally as in the first convention.

Whether function is recursive or not, is decided by the module `symtable`, as mentioned above. During the parsing phase of the source code, the currently defined function is assigned with function calls executed in its body (each unique called function is assigned only once). As the parsing ends a graph is created. The nodes of the graph are represented by the functions and the edges represent the function calls. Each node of the graph is then searched with a Breadth-first search method. Function is declared as recursive, when a cycle containing the currently searched node is created. To ensure cycles not being created outside the currently search node, each node can be searched at most once in a given search. The Breadth-first search method was chosen rather than f.e. Depth-first search method, because recursive functions usually call directly themselves.

2.4.4 Conversion to Three Address Code

The conversion of an expression stored in the token stream in a reverse polish notation (RPN) to three address code (TRA) is implemented in the module `optimizer`.

Not all expressions can be converted from the RPN to TRA. The conversion is performed only on expressions that do not contain function calls. The conversion of an expressions with function calls would not be effective, as the return values, as mentioned above, are returned on the stack and the RPN is already appropriate for stack calculation.

Intermediate results can appear during the conversion to TRA. These results must be temporarily stored. Auxiliary IFJcode20 variables for storing such intermediate results are provided to the `module` optimizer by the module `register`.

2.4.5 Built-in Functions Inlining

The built-in functions inlining is implemented in modules `functions` and `optimizer`.

Functions, that can be directly converted to a single IFJcode20 instruction (i.e. `WRITE`, `STRLEN`, `READ`, `INT2FLOAT/S`, `FLOAT2INT/S`), are inlined by the module `optimizer`. The tokens representing such functions in the token stream are replaced by tokens directly representing IFJcode20 instructions. The function call is completely omitted.

Functions requiring more instructions or generation of some more complicated logic are inlined by the module `functions` during the target code generation. IFJcode20 instructions `CALL` and `RETURN` are omitted, as well as the stack frame establishment, and only the necessary IFJcode20 instructions are generated in place of the function call.

2.5 Code Generation

The target code generation is the final phase of the compilation process implemented in the `generator` module.

Input of the generator is the token stream enriched by the module `optimizer`. The process of optimization (i.e. using the module `optimizer`) could be omitted in the previous version of the compiler. Due to time limitations associated with the project deadline, we decided not to include this feature in the final version of the compiler as we were fully focused on the debugging of the optimized version, which took us longer than expected.

The process of generation is quite simple in comparison to the other parts of the project. The generator reads the token stream, which contains control tokens, data tokens and tokens unused by the generator. Each control token determines, how the next sequence of data tokens will be generated. Data tokens are generated based on their token type and the generator flags set by the previous control token. Unused tokens by the generator are skipped and have no effect on the generation phase.

Generation of cycles and conditions is limited by the size of an integer data type on each scope of the source code program. We are confident, that the limit will not be surpassed by a meaningful program, as todays computer main memory size limitations will not allow storing compiler intermediate code of a program with such a magnitude.

3 Data Structures

3.1 Table of Symbols

We chose the assignment with a hash table used as the abstract data type for storing identifiers. This option appeared to us as far more suitable for such a task. In reality, we use multiple hash tables and a vector of hash tables, to ensure appropriate data storage.

The table of function identifiers is represented by a separate link list chaining hash table. (i.e. Hash table synonyms are stored in a linked list.) Furthermore each function is assign with a table of variable identifiers, where all its variables (including parameters and named return values) are stored. The tables of variable identifiers are also implemented as separate chaining hash tables, this time using a circular linked list. The circular linked list ensures faster lookup of variables on a given scope level. For an example visit section Examples.

3.2 Memory Management Unit

The memory management unit (MMU) is implemented as an open addressing hash table. (i.e. Hash table synonyms are stored at a first unused address.) The functionality of the MMU is to ensure correct memory deallocation at exit at any given time of the program execution.

The starting size of the MMU is set to 64 items. The MMU is doubled and its items rehashed, as the item load factor exceeds 0.625. We chose this value, because hash tables with open addressing perform the best, when the load factor is around 0.6 and 0.625 can be represented without truncation error (IEEE 754 standard). The size of the hash table is not equal to a prime number, as the addressing is always performed with step of a size 1.

Every memory allocation of local auxiliary structures, such as vectors, is overwatched by the MMU. However, the MMU does not work as a garbage collector by any means and should not be used in such a way.

3.3 Vector

Vectors (resizable arrays in this case) are implemented as structures with flexible array member. The starting size of a vector is set to 8 items, as the average usage is around 5 items. If the number of stored items in a vector exceeds its size, the vector size is doubled. On the other hand, when items are removed, the size of the vector remains unchanged.

All vectors offer operations common to stack (i.e. push and pop) and usually are used this way. Other operations might be unique to vectors of a given type.

3.4 Dynamic String

Dynamic strings are implemented similarly as vectors, apart from its size, which is set to 64 characters. Dynamic strings also offer more operations like push substring or read.

3.5 Token Linked List

The token linked list (LL), referred to as a token stream, was implemented in `token` module. The module provides an interface for manipulation of the token stream by other modules.
The token stream is used throughout the whole compilation process from the parser to the generator.

The LL data structure is implemented as a regular LL with some minor tweaks. The tail points to the second to last element of the LL, which enabled us to refer to the last two elements of the LL. Furthermore the LL is also responsible for moving the third part of a `for` loop header to the end of the `for` loop body, which makes the token stream correct for the generation phase.

4 Testing

To verify the correctness of our compiler we have written several auxiliary functions and shell scripts. The main shell script we used for testing is called `cmp_with_go_output.sh`, it uses the interpret of the IFJcode20 language that was provided to us within the course. The essence of the script is in its name, it feeds our compiler with programs written in the IFJ20 language, interprets them and compares the output with the same programs compiled by the Go compiler.

The script also checks, if there are any memory leaks or other memory usage problems caused by our compiler. Moreover, floating point numbers of both outputs are truncated before the comparison, as their printing convention slightly differs.

5 Teamwork

Our team consists of three members

- `xfolty17` Foltyn Lukáš xfolty17@stud.fit.vutbr.cz
- `xmihol00` **Mihola David xmihol00@stud.fit.vutbr.cz**
- `xsokol15` Sokolovskii Vladislav xsokol15@stud.fit.vutbr.cz

David Mihola took on a leadership role during the first weeks of development, from the first meeting he had a vision of how the project could be implemented. He was sharing his vision with the other members of the team and was giving them assignments.

5.1 Communication

As our primary communication channel we chose a Discord chat. We used our Discord chat for making regular calls, sharing our ideas and reporting on completed tasks. At the first stages of development we were trying to use GitHub built-in Kanban table for organising our work flow, however in the process of development we referred to the table less and less and moved all our communication to the discord chat. Since it is not our first team project we did not have any problems with communication and distribution of roles.

5.2 Workload Division

Foltyn Lukáš	lexical analysis, table of symbols, built-in functions, documentation
Mihola David	syntax analysis, semantic analysis, optimization, code generation, design and team leading, documentation
Sokolovskii Vladislav	lexical analysis, code generation, token stream, documentation

6 Conclusion

This project helped us to understand the structure of a compiler and deepened our knowledge of data structures and algorithms.

Working in a team taught us, how to clearly express our thoughts and find compromises with other members of the team.

During the implementation we also improved our knowledge of C language and learned the syntax of Go language. Furthermore we improved at using Git/GitHub and planning of our work.

Last but not least we may apologize for any spelling or other language related errors,
as non of us is a native English speaker.

A LL-Grammar

1. `<INITIAL> -> EOL <INITIAL>`
2. `<INITIAL> -> package <MAIN_PCKG_NEXT>`
3. `<MAIN_PCKG_NEXT> -> main EOL <FUNC_NEXT>`
4. `<MAIN_PCKG_NEXT> -> EOL <MAIN_PCKG_NEXT>`
5. `<FUNC_NEXT>  -> EOL <FUNC_NEXT>`
6. `<FUNC_NEXT>  -> EOF`[1]
7. `<FUNC_NEXT>  -> func <FUNC_NAME>`
8. `<FUNC_NAME>  -> func_id`[2] `<FUNC_PARAM_FIRST>`
9. `<FUNC_NAME>  -> EOL <FUNC_NAME>`
10. `<FUNC_PARAM_FIRST> -> EOL <FUNC_PARAM_FIRST>`
11. `<FUNC_PARAM_FIRST> -> id <FUNC_PARAM_TYPE>`
12. `<FUNC_PARAM_FIRST> -> ) <FUNC_RET_TYPE>`
13. `<FUNC_PARAM_TYPE> -> type`[3] `<FUNC_COMMA_END>`
14. `<FUNC_COMMA_END> -> ) <FUNC_RET_TYPE>`
15. `<FUNC_COMMA_END> -> , <FUNC_PARAM_NAME>`
16. `<FUNC_PARAM_NAME> -> EOL <FUNC_PARAM_NAME>`
17. `<FUNC_PARAM_NAME> -> id <FUNC_PARAM_TYPE>`
18. `<FUNC_RET_TYPE> -> type`[3] `<OCB_NEXT>`
19. `<FUNC_RET_TYPE> -> ( <RET_TYPE_OB>`
20. `<FUNC_RET_TYPE> -> { EOL <STATEMENT> } EOL <FUNC_NEXT>`
21. `<RET_TYPE_OB> -> type`[3] `<RET_TYPE_COMMA>`
22. `<RET_TYPE_OB> -> id <NAMED_RET_TYPES>`

[1] The end of a syntax analysis
[2] func_id = `id(`
[3] type = {`int`, `float`, `string`, `bool`}

23. `<RET_TYPE_OB> -> ) <OCB_NEXT>`

24. `<RET_TYPE_OB> -> EOL <RET_TYPE_OB>`

25. `<RET_TYPE_COMMA> -> , <MUL_RET_TYPES>`

26. `<RET_TYPE_COMMA> -> ) <OCB_NEXT>`

27. `<MUL_RET_TYPES> -> type`[3] `<RET_TYPE_COMMA>`

28. `<MUL_RET_TYPES> -> EOL <MUL_RET_TYPES>`

29. `<NAMED_RET_TYPES> -> type`[3] `<NRET_TYPE_COMMA>`

30. `<NRET_TYPE_COMMA> -> ) <OCB_NEXT>`

31. `<NRET_TYPE_COMMA> -> , <RET_NAME>`

32. `<RET_NAME> -> id <NAMED_RET_TYPES>`

33. `<RET_NAME> -> EOL <RET_NAME>`

34. `<OCB_NEXT> -> { EOL <STATEMENT> } EOL <FUNC_NEXT>`

35. `<STATEMENT> ->` ε

36. `<STATEMENT> -> EOL <STATEMENT>`

37. `<STATEMENT> -> for <FOR1>; <EXPRESSION>; <FOR3> { EOL <STATEMENT> } EOL <STATEMENT>`

38. `<STATEMENT> -> if <EXPRESSION> { EOL <STATEMENT> } <ELSE_COND>`

39. `<STATEMENT> -> id <ASSIGN>`

40. `<STATEMENT> -> return <RETURN>`

41. `<STATEMENT> -> func_id`[2] `<EXPRESSION> ) EOL <STATEMENT>`

42. `<ASSIGN> -> EQ`[4] `<EXPRESSION> EOL <STATEMENT>`

43. `<ASSIGN> -> , <MULTI_ASSIGN>`

44. `<MULTI_ASSIGN> -> id <MULTI_ASSIGN_COMMA>`

45. `<MULTI_ASSIGN> -> EOL <MULTI_ASSIGN>`

[3]type = {`int, float, string, bool`}
[2]func_id = `id(`
[4]EQ = {`+=, -=, *=, /=, =, :=`}

46. <MULTI_ASSIGN_COMMA> -> , <MULTI_ASSGN>

47. <MULTI_ASSIGN_COMMA> -> {:=, =} <EXPRESSION> EOL <STATEMENT>

48. <ELSE_COND> -> EOL <STATEMENT>

49. <ELSE_COND> -> else <IF_COND>

50. <IF_COND> -> { EOL <STATEMENT> } EOL <STATEMENT>

51. <IF_COND> -> if <EXPRESSION> { EOL <STATEMENT> } <ELSE_COND>

52. <FOR1> -> id <FOR1_ASSIGN>

53. <FOR1> -> ε

54. <FOR1> -> EOL <FOR1>

55. <FOR1_ASSIGN> -> := <EXPRESSION>

56. <FOR1_ASSIGN> , <FOR1_MULTI_ASSGN>

57. <FOR1_MULTI_ASSGN> -> id <FOR1_MA_COMMA>

58. <FOR1_MULTI_ASSGN> -> EOL <FOR1_MULTI_ASSGN>

59. <FOR1_MA_COMMA> -> , <FOR1_MULTI_ASSGN>

60. <FOR1_MA_COMMA> -> := <EXPRESSION>

61. <FOR3> -> id <FOR3_ASSIGN>

62. <FOR3> -> ε

63. <FOR3> -> EOL <FOR3>

64. <FOR3_ASSIGN> -> EQ[4]\{:=} <EXPRESSION>

65. <FOR3_ASSIGN> -> , <FOR3_MULTI_ASSGN>

66. <FOR3_MULTI_ASSGN> -> id <FOR3_MA_COMMA>

67. <FOR3_MULTI_ASSGN> -> EOL <FOR3_MULTI_ASSGN>

68. <FOR3_MA_COMMA> -> , <FOR3_MULTI_ASSGN>

69. <FOR3_MA_COMMA> -> = <EXPRESSION>

[4]EQ = {+=, -=, *=, /=, =, :=}

70. `<RETURN> -> EOL <STATEMENT>`

71. `<RETURN> ->` ε `<EXPRESSION> EOL <STATEMENT>`

We also alowed the IFJ20 source code to end with `EOF` directly behind a function body in the final implementation, which is not possible by the LL-grammar specified above.

New lines in expressions are not parsed in the operator precedence analysis. New line in an expression is a valid character, if the previously returned token type by the scanner is one of the following: `operator, comma, if keyword, semicolon, any equal type, opening bracket, function identifier, new line`.

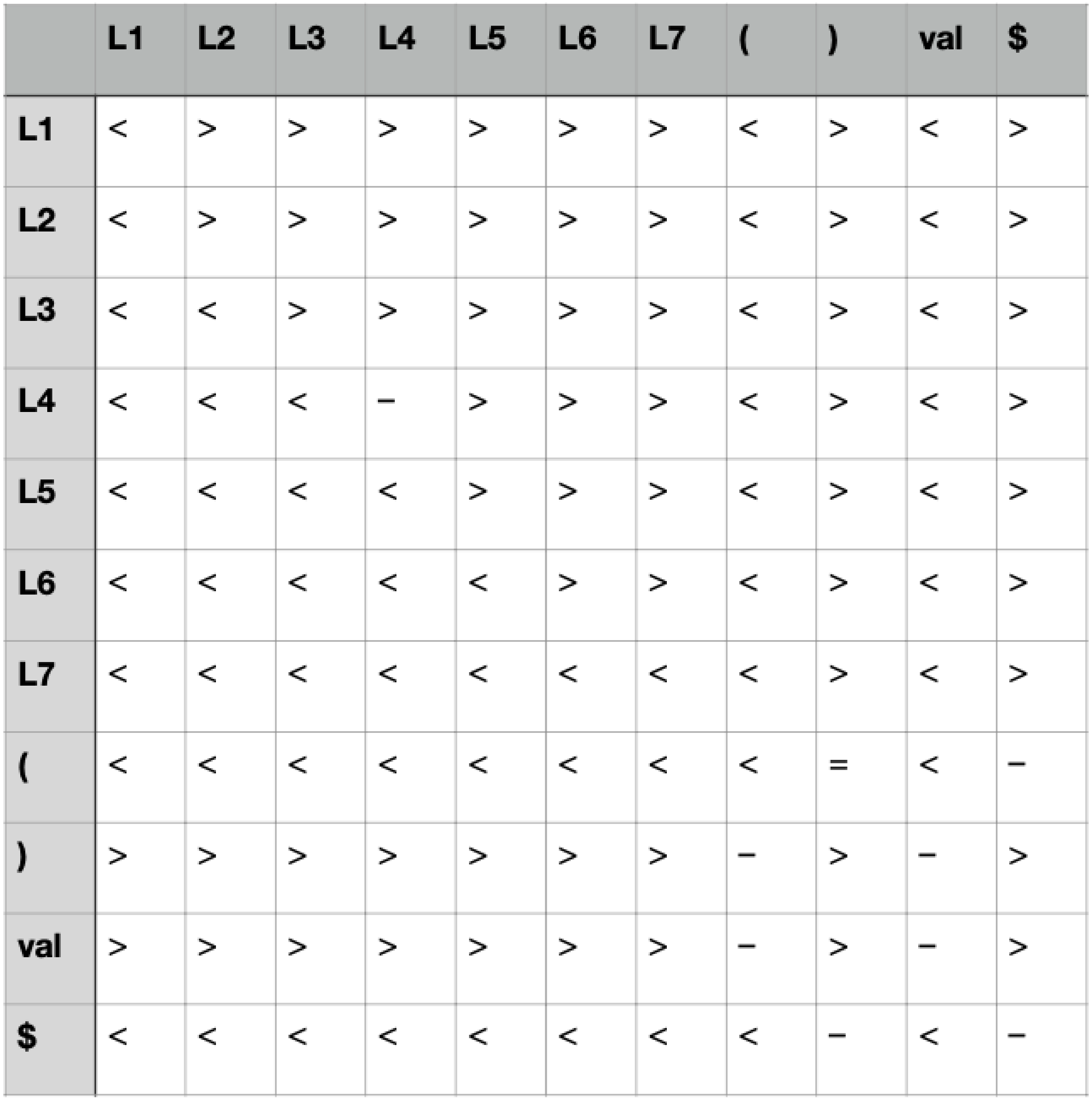
A.1 LL-Table

	package	main	EOL	func	func_id	id	)	type	,	(	{	If	else	for	return	EQ	{:=, =}	=	EQ / {:=}	:=	EOF	$
<INITIAL>	2		1																			
<MAIN_PCKG_NEXT>		3	4																			
<FUNC_NEXT>			5	7																	6	
<FUNC_NAME>			9		8																	
<FUNC_PARAM_FIRST>			10			11	12															
<FUNC_PARAM_TYPE>								13														
<FUNC_COMMA_END>							14		15													
<FUNC_PARAM_NAME>			16			17																
<FUNC_RET_TYPE>								18		19	20											
<RET_TYPE_OB>			24			22	23	21														
<RET_TYPE_COMMA>							26		25													
<MUL_RET_TYPES>			28					27														
<NAMED_RET_TYPES>								29														
<NRET_TYPE_COMMA>							30		31													
<RET_NAME>			33			32																
<OCB_NEXT>											34											
<STATEMENT>			36		41	39						38		37	40							35
<ASSIGN>									43							42						
<MULTI_ASSIGN>			45			44																
<MULTI_ASSIGN_COMMA>									46								47					
<ELSE_COND>			48										49									
<IF_COND>											50	51										
<FOR1>			54			52																53
<FOR1_ASSIGN>									56											55		
<FOR1_MULTI_ASSIGN>			58			57																
<FOR1_MA_COMMA>									59											60		
<FOR3>			63			61																62
<FOR3_ASSIGN>									65										64			
<FOR3_MULTI_ASSIGN>			67			66																
<FOR3_MA_COMMA>									68									69				
<RETURN>			70																			71

Figure 1: Parsing table based on the given LL-grammar.

B Precedence Table

L1 = {`+, -, !`} – unary operators
L2 = {`*, /`}
L3 = {`+, -`} – binary operators
L4 = {`==, !=, >=, <=, >, <`}
L5 = {`&&`}
L6 = {`||`}
L7 = {`comma, +=, -=, *=, /=, =, :=` }



	L1	L2	L3	L4	L5	L6	L7	(	)	val	$
L1	<	>	>	>	>	>	>	<	>	<	>
L2	<	>	>	>	>	>	>	<	>	<	>
L3	<	<	>	>	>	>	>	<	>	<	>
L4	<	<	<	–	>	>	>	<	>	<	>
L5	<	<	<	<	>	>	>	<	>	<	>
L6	<	<	<	<	<	>	>	<	>	<	>
L7	<	<	<	<	<	<	<	<	>	<	>
(	<	<	<	<	<	<	<	<	=	<	–
)	>	>	>	>	>	>	>	–	>	–	>
val	>	>	>	>	>	>	>	–	>	–	>
$	<	<	<	<	<	<	<	<	–	<	–

Figure 2: Table respresenting the precedence of operators.

C Finite State Automata

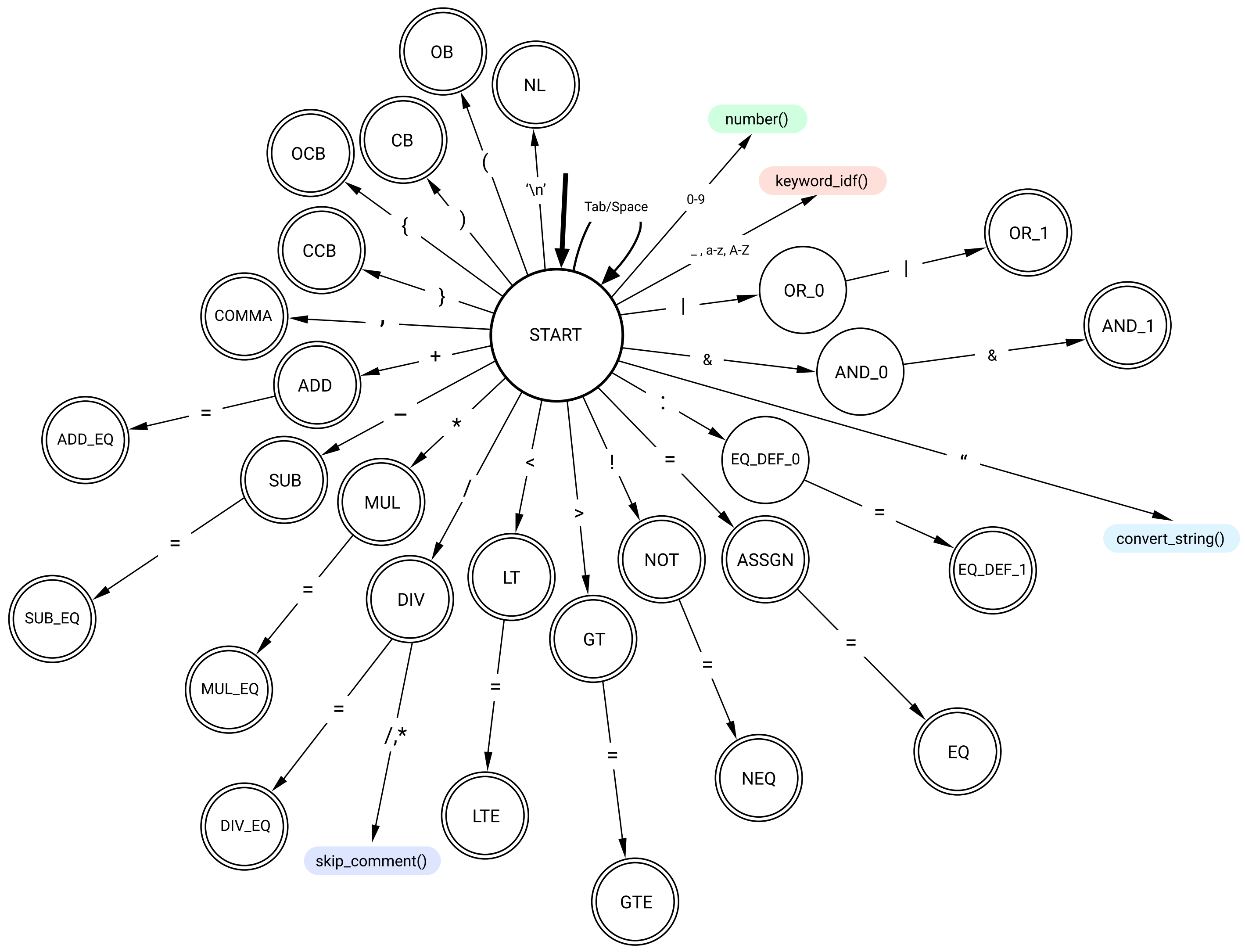


Figure 3: Finite state automata representing the scanner

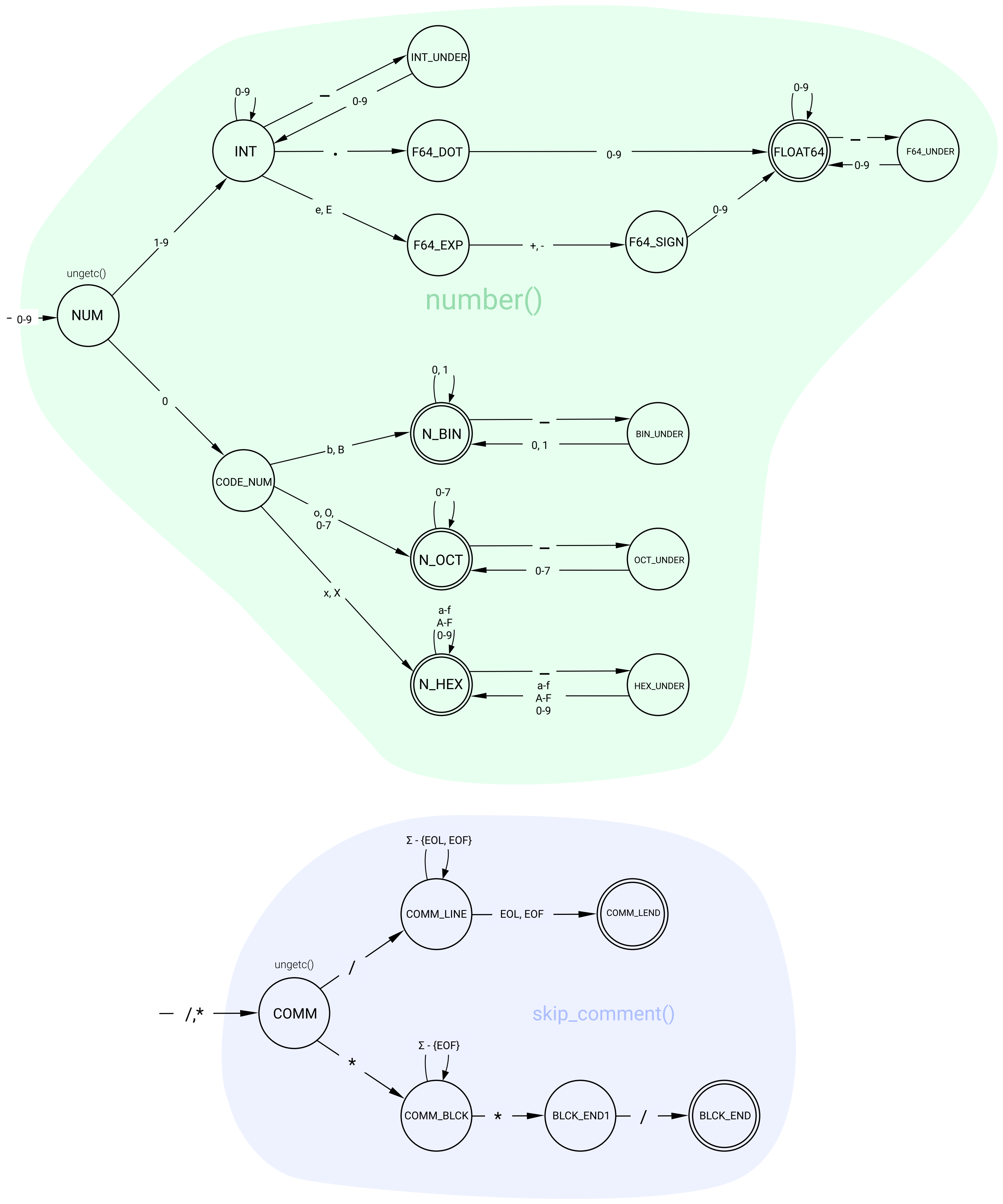


Figure 4: Auxiliary functions of the scanner

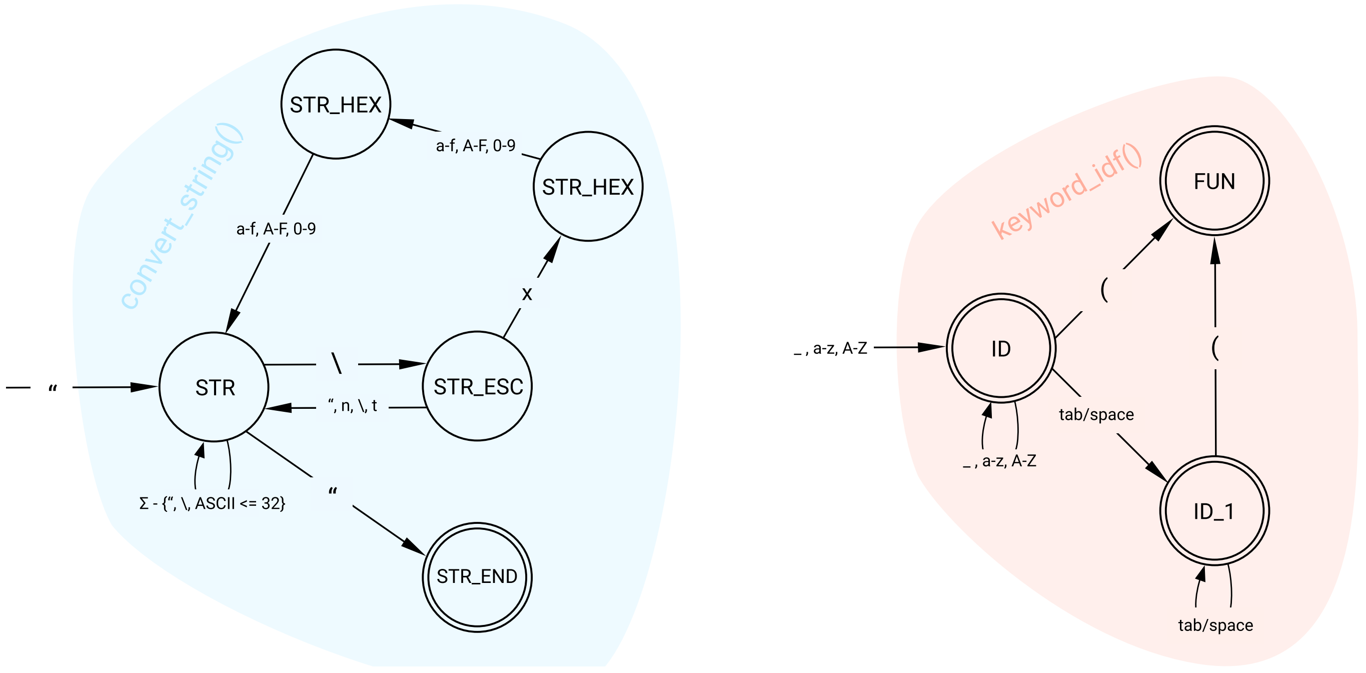


Figure 5: Auxiliary functions of the scanner

EOL - end of line

EOF - end of file

Σ - alphabet except of EOF

States `ID` and `ID_1` can also return a **`keyword`** token type, if the read identifier matches with a table of keywords or a `constant` if the read identifier is `true` or `false`.

D Examples

D.1 Hash Tables Design

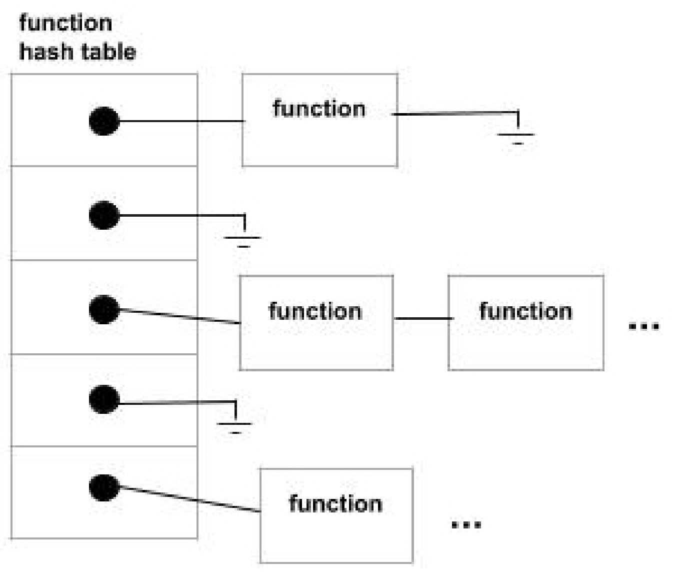


Figure 6: Hash table of functions

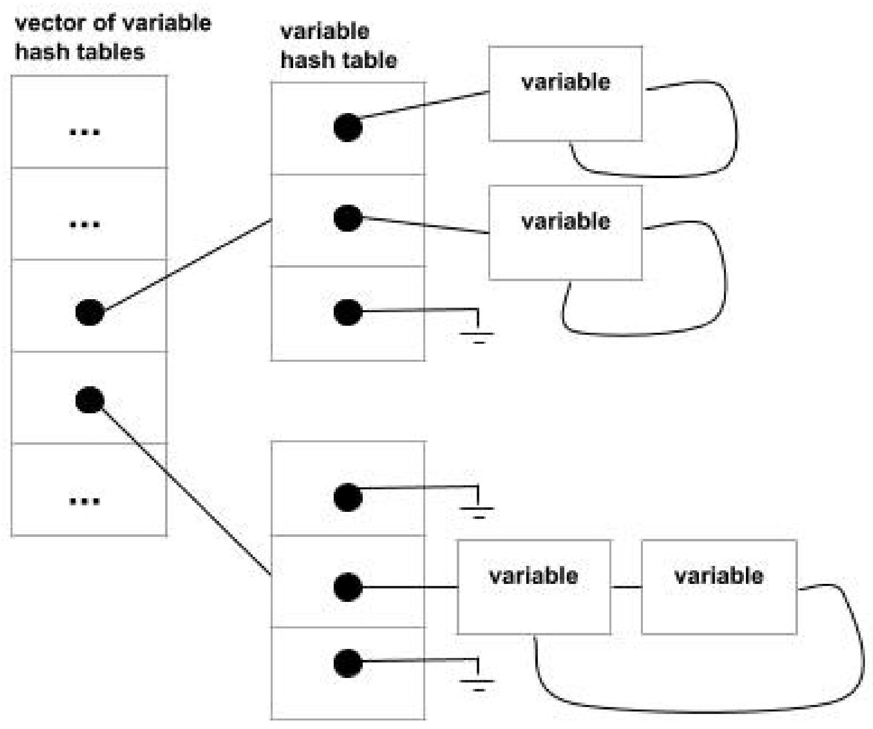


Figure 7: Hash tables of variables

D.2 Hash Tables Usage

D.2.1 IFJ20 Source Code Example

```
package main

func foo() (int) { // 1.  function definition
     x := 4 // 2.  variable definition
     if x > 3 { // 3.  start of a new scope
         x := 6 // 4.  variable definition
     } // 5.  end of a scope
    return x
}

func main() { // 6.  another function definition
}
```

D.2.2 IFJ20 Source Code Explanation

1. Newly added function to the function hash table is assigned with a new variable hash table. The assigned slot in the vector of variable hash tables becomes active.

2. Newly defined variable `x` is stored into the currently active variable hash table on scope level 0.

3. Parser scope counter is increased.

4. Newly define variable `x` with the same name, therefore the same hash function, but in higher scope (1). The variable is stored to the variable hash table in front of the variable from lower scope, so it can be found first.

5. Rotating with the circular linked list until all variables of currently closed scope are marked as inactive.

6. Currently used variable hash table is deactivated, the process continues with action 1.

D.2.3 Graphical Example

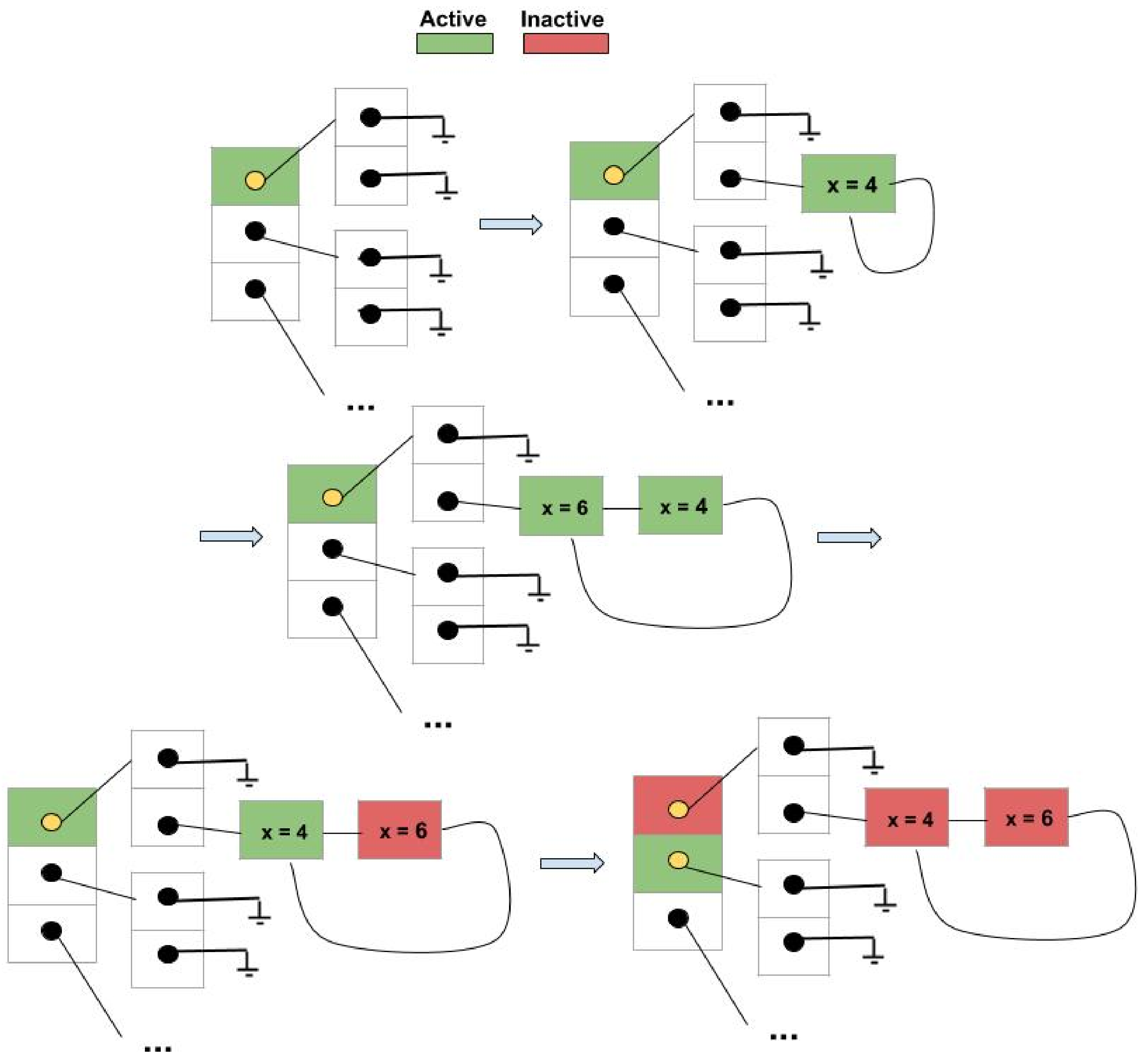


Figure 8: Example of development of the hash table during execution time

D.3 Optimization

D.3.1 IFJ20 Source Code Example

```
package main

func foo (x int, y float64) int {
    return x+float2int(y)
}

func main() {
    a := 1+2*6/(5+2-1)*2
    b := 3*a-5
    c, _ := inputf()
    d := int2float(a+b)

    if a > b {
        b = a
    } else if c >= d {
        print(foo(a*b, c+d), " ", c, "\n")
    }

    b = 5+4*a*b/2/a+a
    print(a, " ", b, "\n")
}
```

D.3.2 Generated IFJcode20 program

```
.IFJcode20
# definition of compiler variables
DEFVAR GF@?temp&1
DEFVAR GF@?temp&2
DEFVAR GF@?temp&3
DEFVAR GF@?temp&4
DEFVAR GF@?temp&5
DEFVAR GF@?temp&6
DEFVAR GF@?_&

# definition of variables of non recursive functions
DEFVAR GF@x&13
DEFVAR GF@y&14
DEFVAR GF@a&15
DEFVAR GF@b&16
DEFVAR GF@c&17
DEFVAR GF@d&18
```

```
# definition of registers
DEFVAR GF@r%1&
DEFVAR GF@r%2&
JUMP $main

# $foo function definition
LABEL $foo
# memory frame creation omitted
# return x + float2int(y)
PUSHS GF@x&13
# float2int() function inlined, with the use of temporarily variable
FLOAT2INT GF@?temp&1 GF@y&14
PUSHS GF@?temp&1
ADDS
# value returned on the stack
RETURN
# definition end

# main function definition
LABEL $main
# memory frame creation omitted

# c, _ := inputf()
# inputf() function inlined
# read value is not verified, as the error signifying variable is _
READ GF@c&17 float

# d := int2float(a+b), a == 5, b == 10
# direct conversion, with the use of constant propagation
INT2FLOAT GF@d&18 int@15

# a := 1+2*6/(5+2-1)*2, previously evaluated to a := 5
# b := 3*a-5, previously evaluated to b := 10
# preemptive assign of variables with constant value before if statement
MOVE GF@a&15 int@5
MOVE GF@b&16 int@10

# if a > b {
# unconditioned jump based on a compile time calculated values
JUMP $main_if1_1

# b = a
# a == 5
MOVE GF@b&16 int@5
# }
JUMP $main_ifend1
LABEL $main_if1_1

# else if c >= d {
# calculation of GTE by using LT and NOT
# use of a register for storing intermediate result
LT GF@r%1& GF@c&17 GF@d&18
JUMPIFNEQ $main_if1_2 GF@r%1& bool@false

# foo(a*b, c+d), a == 5, b == 10
# assigning/calculating the parameters of called function
MOVE GF@x&13 int@50
ADD GF@y&14 GF@c&17 GF@d&18
CALL $foo
# assigning the return value from the stack after a function call
POPS GF@?temp&1

# print(foo(a*b, c+d), " ", c, "\n")
# print() function inlined
WRITE GF@?temp&1
WRITE string@\032
WRITE GF@c&17
WRITE string@\010

# }
LABEL $main_if1_2
LABEL $main_ifend1

# b = 5+4*a*b/2/a+a
# the value of variable b is unknown after the if statement, a == 5
# 4*a is reduced to 20
MUL GF@r%2& int@20 GF@b&16
# 2/a is reduced to 10 (operator inversion)
IDIV GF@r%2& GF@r%2& int@10
# 5+...+a - separated constants reduced to 10
ADD GF@b&16 int@10 GF@r%2&

# print(a, " ", b, "\n"), a == 5
# print() function inlined
WRITE int@5
WRITE string@\032
WRITE GF@b&16
WRITE string@\010

EXIT int@0
# definition end
```