Project documentation

Implementation of the Compiler for the Imperative Language IFJ23

Team xpomsa00

Variant Hash Table Implementation Using Open Addressing

December 6, 2023

Jakub Pomsár	xpomsa00	25%
Šimon Cagala	xcagal00	25%
Marie Koláríková	xkolar77	25%
Adrián Ponechal	xponec01	25%

Contents

1 Teamwork

1.1 Work distribution

xpomsa00:	Team leader, symbol table, Top-Down parser, testing, documentation
xcagal00:	Top-Down parser, LL grammar, documentation
xkolar77:	Lexical analyzer, code generator, CI/CD setup, documentation
xponec01:	Precedence analysis, documentation

The workload was evenly distributed, ensuring that each team member contributed equally, and each will receive the same amount of points (25 %).

1.2 Communication & Git

Communication among team members primarily occurred through *Discord* or personal interactions. During the course of the project, we employed the *Git* version control system to manage the evolution of our codebase. GitLab served as our remote repository, providing a centralized platform for collaboration and version tracking. The integration of *CI/CD* testing on *GitLab* ensured that changes introduced to the codebase were systematically tested.

1.3 Testing

Initially, unit tests were crafted to guarantee the correct behavior of individual modules. As the project progressed, integration tests and end-to-end tests were developed to ensure seamless integration and communication among these modules.

Additionally, trial submissions were employed, with achieving correctness rates of **40%** and **65%**, respectively. Towards the completion of the project, student's tests were incorporated for further validation. As mentioned earlier in 1.2, *CI/CD* testing played a crucial role in verifying the correctness of individual pushed branches.

2 Lexical analyzer

The lexical analyzer[1], implemented as a finite-state machine (2.2), decomposes source code from standard input into a sequence of tokens. Individual tokens are represented by the final states.

2.1 Implementation

When the automaton is in the final state and is not able to process the input char, it returns a token to the parser. When the automaton is not in the final state and is not able to process the input char, it returns a lexical error. Each state is represented by a function. The current state is stored in a global variable as a pointer to the corresponding function. Each states sets the function pointer to the next state using the `NEXT_STATE` macro, depending on the loaded char. The token is represented by a structure containing information about the token type, value and flag that indicates the presence of `EOL` before the processed token. Token value is implemented as a `union` and can contain the name of an identifier, an int, string or double literal value or a boolean `is_nilable`. The token type is implemented using the enum type.

The main loop performs a passage through the automaton by repeated invocation of the function referenced by the variable. Additional global variables, that help remember information that have a lifetime longer than one state, are added in the lexical analyzer. The communication interface with the lexical analyzer is implemented using the function `get_token()`, which has one parameter that represents a pointer to the variable to load the token into. The Lexical analyzer includes counters (`actual_line` and `actual_column`) to tell where the error occurred.

[1] Implemented in module `lexical_analyzer.c` with the corresponding header file `lexical_analyzer.h`

2.2 Finite-state machine diagram

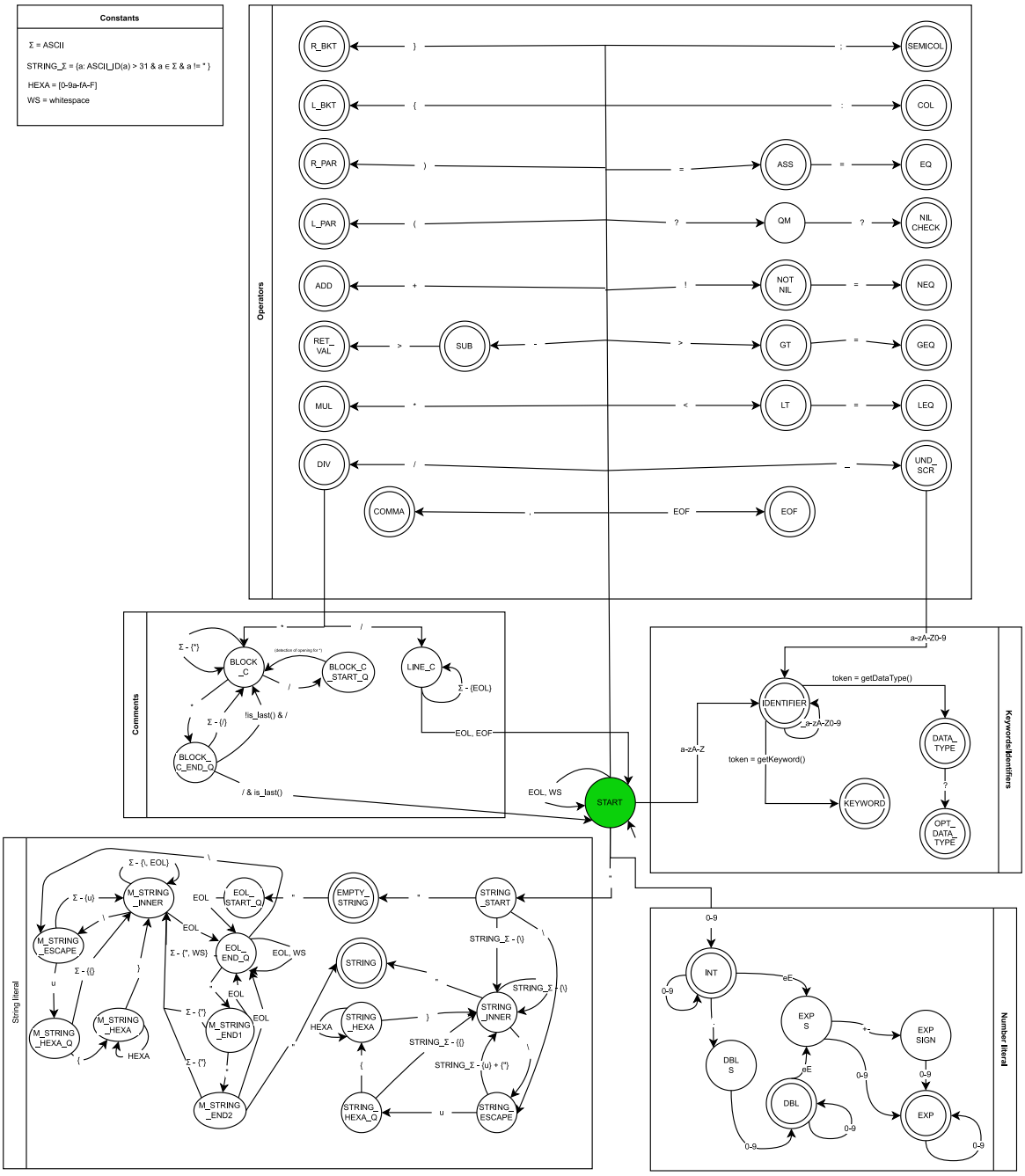

Figure 1: Finite-state machine diagram

3 Symbol Table

3.1 Implementation Overview

The symbol table[2], implemented as a hash table using open addressing, dynamically resizes when the load surpasses 65%, handling conflicts through double hashing.

3.1.1 Record Structure

Each record holds essential details: variable/function name, unique identifier, data type, variable mutability status, definition, declaration, nilability, variadic parameters and the return type of functions. The record also points to a linked list preserving parameters.

3.2 Hash Functions

Two hash functions are employed: the first (utilized consistently) is *sdbm* variant used in the IJC class [2], and the second, *djb2*, is used for double hashing as introduced in the IAL class [1].

3.3 Key Implementation Aspects

3.3.1 Item Deletion

Instead of traditional deletion, the symbol table marks an item's 'activity' attribute as false. Although technically present, the item is considered inactive, creating the appearance of deletion.

3.3.2 Resizing

When the load surpasses **65%**, the symbol table undergoes resizing, creating a new symbol table of size that is two times larger and is a prime number. During this process, every item (excluding inactive items) from the old symbol table is rehashed and transferred to the new one.

3.3.3 Global and Local Instances

A **global symbol table** persists throughout execution, as well as a **stack of local symbol tables**[3] (representing individual scopes), which are *created, stacked, and popped* as needed.

4 Top-Down Parser

4.1 Implementation

The parser[4] is controlled by the main `parse()` function. It starts off by initializing the `Parser` data structure (4.1.1) and code generator (6) as well as generating the specified built-in functions. Next it utilizes the lexical analyzer to load tokens from `stdin`, which are then stored in the `Parser.buffer`(4.1.2) structure. After all necessary data is initialized, the parser begins two-stage syntax and semantic analysis (4.2), the only exception being expressions, which are parsed using precedence analysis in a separate module[5] (5).

4.1.1 Parser Data

The `Parser` data structure houses all data needed by the parser during analysis, such as: pointers to global symbol table and local symbol table stack (3), the current function, variable or parameter identifiers being processed, current token loaded from the buffer and various counters and flags denoting the state

[2]The symbol table is implemented in the files `symtable.c` and `symtable.h`
[3]The stack for local symbol tables is implemented in the files `scope.c` and `scope.h`.
[4]`parser.c` and `parser.h` files
[5]`expression.c` and `expression.h`

the parser is currently in. The data structure is initialized at the start of the `parse()` function and a pointer to it is passed as an argument[6] during recursive descend.

4.1.2 Token Buffer

[7] A doubly linked list of tokens, which are stored for the purpose of parsing the input in two stages (4.2). This structure also allows the parser to backtrack in certain situations, where a simple LL(1) analysis is not sufficient, such as when deciding whether an identifier on the right-hand side of assignment is a variable or called function.

4.2 Analysis

For the most part, syntax and semantic analysis happen simultaneously, by recursively calling `Rule` functions which represent the grammar's non-terminals. However, due to the fact that user-defined function calls can be found earlier than their definitions, the input needs to be read twice.

First Stage analyzes only function declarations, which entails adding their identifiers, parameters and return type to the global symbol table. Any tokens that are not a part of a function declaration are ignored by the parser using one of the several `skip()` 'Rules'. After `EOF` is loaded, the `buffer.runner` pointer is reset to the beginning of the list.

Second Stage analyzes syntax and semantics of the rest of the input program. At this point the generation of the output code takes place by utilizing the `code_generator` module (6). At this point, when the parser loads tokens related to function declarations, it skips over them to avoid attempts to redefine items already stored in the symbol table.

4.2.1 Syntax

At the centre of syntax analysis lies the `GET_TOKEN` macro, which moves the `buffer.runner` pointer and returns the a token from the list. The types of tokens are then asserted based on rules defined in LL-grammar (7.1). The current token is also used to derive the next `Rule` function to be called. If at any point during the recursive descend the current token does not correspond to the terminals in rule, or the next rule cannot be derived a relevant error code is returned and recursion ends.

4.2.2 Semantics

Semantic analysis is done utilizing mainly symbol table and variables inside `struct Parser` to ensure the input program corresponds with the specifications of IFJ23 language.

Identifiers are checked for attempts of redefinition or usage of undefined ones (which leads to error 3 and 5, depending on the type of identifier).

Function arguments are asserted to be of the same type as the defined parameters and that the correct parameter label is used (or that none is used in the case of a label utilizing the underscore notation). The `Parser.in_func_body` boolean keeps track of whether the parser is currently inside a function, which together with `Parser.return_found` is used for checking the validity of a potential `return` statement. An extra `return` outside of the function body, or a missing one inside leads to error 6.

5 Expression processing

5.1 Syntax analysis

Whole expression processing is handled by the `expression.c` module with the corresponding header file `expression.h`. The `expr()` function is called whenever an expression is expected by the parser (4). This function accepts a pointer to the `Parser` structure and returns the error code of first error found.

[6] e.g. `Rule prog(Parser* p)`

[7] `token_buffer.c` and `token_buffer.h` files

Expressions are processed in 4 states: shift (<), equal shift(=), reduction(>) and error reduction(empty space). The correct state is chosen by using precedence table 5.1.2, where the rows represent the last terminal on stack and the columns represent an incoming terminal in the current expression.

During shift and equal shift operations a symbol is pushed on stack. In the shift state, the handle is set to the closest terminal from the top of the stack. In contrast with the shift state, there is no need to set the handle in equal shift state. The handle is used also in the reduction state.

Expressions are reduced by a suitable rule in reduction state. During this state all symbols are popped from the stack (from top to handle) to an array of symbols. At most 3 symbols can be pushed to this array. These symbols are used to decide what reduction rule should be used for the operation. If there is no rule for these symbols, an error occurs.

5.1.1 End of expression detection

End of the expression is detected by the precedence table in error state or in the error handling function in reduction state. If the end is found in error handling functions, recovery is performed (the last pointer is set to the first symbol after the expression and the symbol is removed from from stack if necessary).

5.1.2 Symstack data structure

A stack implemented in the `symstack.c` module is used for analyzing syntax. It contains inserted terminals and non-terminals of the current expression. This structure is a list of nodes linked from top to bottom. Every node contain data and a pointer to the previous node. The data stores information about a specific expression symbol, like a token, and basic flags that contain information about the corresponding expression symbol. The expression symbol can be either an operand, an operator or a reduced expression.

	*/	+-	??	i	RO	(	)	!	$
*/	>	>	>	<	>	<	>	<	>
+-	<	>	>	<	>	<	>	<	>
??	<	<	<	<	<	<	>	<	>
i	>	>	>		>		>	>	>
RO	<	<	>	<	=	<	>	<	>
(	<	<	<	<	<	<	=	<	
)	>	>	>		>		>		>
!	>	>	>	>	>	>	>	>	>
$	<	<	<	<	<	<	<		>

Table 1: Precedence table

Note: *i - identifier*, *RO- relational operator*

5.2 Semantic analysis

Expression semantic analysis takes place in the reduction phase of processing the current operation. This is done across several functions with the`process_` prefix. If possible, these functions will trigger implicit conversion of the operands. Implicit conversion is handled by `convert_if_retypeable` function.

The validity of identifiers used in an expression is handled by the `process_operand` function. These functions utilize the `symtable` module to check whether the identifiers were previously defined.

6 Code generator

Code generator is implemented in the `code_generator.c` module with the corresponding header file `code_generator.h`. There is no optimizer implemented in the project. Supportive functions are called from the parser or expression modules. The generator utilizes the stack which greatly reduces the number of generated compiler variables. Most constructions are generated directly, the exception being declarations

inside a cycle. Declaration of variables inside loops is overridden on assignments. Inside the cycle, everything except for variable declarations is stored in a buffer, while declarations are written directly to `stdout`, after the end of the cycle, the entire content of the buffer is written to `stdout`, this is done to prevent variable redeclarations.

7 Grammar

7.1 LL-Grammar

1. $\langle program \rangle \Rightarrow \langle statement \rangle\ \langle program \rangle$
2. $\langle program \rangle \Rightarrow$ `func ID(` $\langle param - list \rangle\ \langle func - return - type \rangle$ `{` $\langle func - body \rangle\ \langle program \rangle$
3. $\langle program \rangle \Rightarrow$ `EOF`
4. $\langle statement \rangle \Rightarrow \langle mutb \rangle\ \langle define \rangle$
5. $\langle statement \rangle \Rightarrow$ `ID` $\langle id - type \rangle$
6. $\langle statement \rangle \Rightarrow$ `while EXPRESSION` `{` $\langle block - body \rangle$
7. $\langle statement \rangle \Rightarrow$ `if` $\langle cond - clause \rangle$ `{` $\langle block - body \rangle$ `else` `{` $\langle block - body \rangle$
8. $\langle define \rangle \Rightarrow$ `ID` $\langle define - cont \rangle$
9. $\langle define - cont \rangle \Rightarrow$ `:` $\langle type \rangle\ \langle opt - assign \rangle$
10. $\langle define - cont \rangle \Rightarrow$ `=` $\langle assign - exp \rangle$
11. $\langle opt - assign \rangle \Rightarrow$ `=` $\langle assign - exp \rangle$
12. $\langle opt - assign \rangle \Rightarrow \varepsilon$
13. $\langle id - type \rangle \Rightarrow$ `=` $\langle assign - exp \rangle$
14. $\langle id - type \rangle \Rightarrow$ `(` $\langle arg - list \rangle$
15. $\langle cond - clause \rangle \Rightarrow$ `let ID`
16. $\langle cond - clause \rangle \Rightarrow$ `EXPRESSION`
17. $\langle arg - list \rangle \Rightarrow \langle arg \rangle\ \langle arg - next \rangle$
18. $\langle arg - list \rangle \Rightarrow$ `)`
19. $\langle arg - next \rangle \Rightarrow$ `,` $\langle arg \rangle\ \langle arg - next \rangle$
20. $\langle arg - next \rangle \Rightarrow$ `)`
21. $\langle arg \rangle \Rightarrow$ `ID` $\langle opt - arg \rangle$
22. $\langle arg \rangle \Rightarrow \langle literal \rangle$
23. $\langle param - list \rangle \Rightarrow \langle param \rangle\ \langle param - next \rangle$
24. $\langle param - list \rangle \Rightarrow$ `)`
25. $\langle param - next \rangle \Rightarrow$ `,` $\langle param \rangle\ \langle param - next \rangle$
26. $\langle param - next \rangle \Rightarrow$ `)`
27. $\langle param \rangle \Rightarrow$ `LabelID ParamID :` $\langle type \rangle$
28. $\langle param \rangle \Rightarrow$ `_ ParamID :` $\langle type \rangle$

29. $\langle block - body \rangle \Rightarrow \langle statement \rangle \ \langle block - body \rangle$
30. $\langle block - body \rangle \Rightarrow \}$
31. $\langle func - body \rangle \Rightarrow \langle func - statement \rangle \ \langle func - body \rangle$
32. $\langle func - body \rangle \Rightarrow \}$
33. $\langle func - statement \rangle \Rightarrow \langle mutb \rangle \ \langle define \rangle$
34. $\langle func - statement \rangle \Rightarrow$ `ID` $\langle id - type \rangle$
35. $\langle func - statement \rangle \Rightarrow$ `while EXPRESSION {` $\langle func - body \rangle$
36. $\langle func - statement \rangle \Rightarrow$ `if` $\langle cond - clause \rangle$ `{` $\langle func - body \rangle$
37. $\langle func - statement \rangle \Rightarrow$ `return` $\langle opt - ret \rangle$
38. $\langle func - return - type \rangle \Rightarrow$ `->` $\langle type \rangle$
39. $\langle func - return - type \rangle \Rightarrow \varepsilon$
40. $\langle opt - ret \rangle \Rightarrow$ `EXPRESSION`
41. $\langle opt - ret \rangle \Rightarrow \varepsilon$
42. $\langle opt - arg \rangle \Rightarrow$ `:` $\langle term \rangle$
43. $\langle opt - arg \rangle \Rightarrow \varepsilon$
44. $\langle mutb \rangle \Rightarrow$ `var`
45. $\langle mutb \rangle \Rightarrow$ `let`
46. $\langle assign - exp \rangle \Rightarrow$ `EXPRESSION`
47. $\langle assign - exp \rangle \Rightarrow \langle func - call \rangle$
48. $\langle func - call \rangle \Rightarrow$ `ID(` $\langle arg - list \rangle$
49. $\langle type \rangle \Rightarrow$ `Int`
50. $\langle type \rangle \Rightarrow$ `Double`
51. $\langle type \rangle \Rightarrow$ `String`
52. $\langle term \rangle \Rightarrow$ `ID`
53. $\langle term \rangle \Rightarrow \langle literal \rangle$
54. $\langle literal \rangle \Rightarrow$ `INT_LITERAL`
55. $\langle literal \rangle \Rightarrow$ `DOUBLE_LITERAL`
56. $\langle literal \rangle \Rightarrow$ `STRING_LITERAL`
57. $\langle literal \rangle \Rightarrow$ `nil`

7.2 LL-Table

	func	ID	let	var	if	else	while	return	(	)	{	}	EXP	_	:	=	,	->	Int	Double	String	INT_LIT	DBL_LIT	STR_LIT	nil	EOF	ε
<program>	2	1	1	1	1		1																			3	
<statement>		5	4	4	7		6																				
<define>		8																									
<define-cont>															9	10											
<opt-assign>																11											12
<id-type>																13											
<cond-clause>			15						14				16														
<arg-list>		17								18												17	17	17	17		
<arg-next>										20							19										
<arg>		21																				22	22	22	22		
<param-list>		23								24																	
<param-next>										26							25										
<param>		27												28													
<block-body>		29	29	29	29		29					30															
<func-body>		31	31	31	31		31	31				32															
<func-statement>		34	33	33	36		35	37																			
<func-return-type>																		38									39
<opt-ret>													40														41
<opt-arg>																42											43
<mutb>			45	44																							
<assign-exp>		47																									
<funccall>		48																									
<type>																			49	50	51						
<term>		52																				53	53	53	53		
<literal>																						54	55	56	57		

References

[1] Filip Stanis. 008 - djb2 hash. online, September 2019. https://theartincode.stanis.me/008-djb2.

[2] Ozan Yigit. Hash functions. online, September 2003. http://www.cse.yorku.ca/~oz/hash.html.