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

Lexical analyzer is implemented in module `lexical_analyzer.c` with the corresponding header file `lexical_analyzer.h`. The lexical analyzer decomposes source code from standard input to a sequence of tokens. A lexical analyzer has been implemented using a finite-state machine (2.2). 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, returns a token to the parser. When the automaton is not in the final state and is not able to process the input char, returns lexical error. Each state is represented by a function. The current state is stored in global variable as pointer to the corresponding function. Next state is set in function of current state by modifying variable with pointer to current state. The token is represented by a structure containing information about the token type, value and flag that signs preceding eol before processed token. The value is value of a literal or the name of an identifier or flag of nilable data type and is implemented using the union data type. The token type is implemented using the enum type.

The main loop performs 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 has an counters (`actual\_line` and `actual\_column`) to tell where the error occurred.

2.2 Finite-state machine diagram

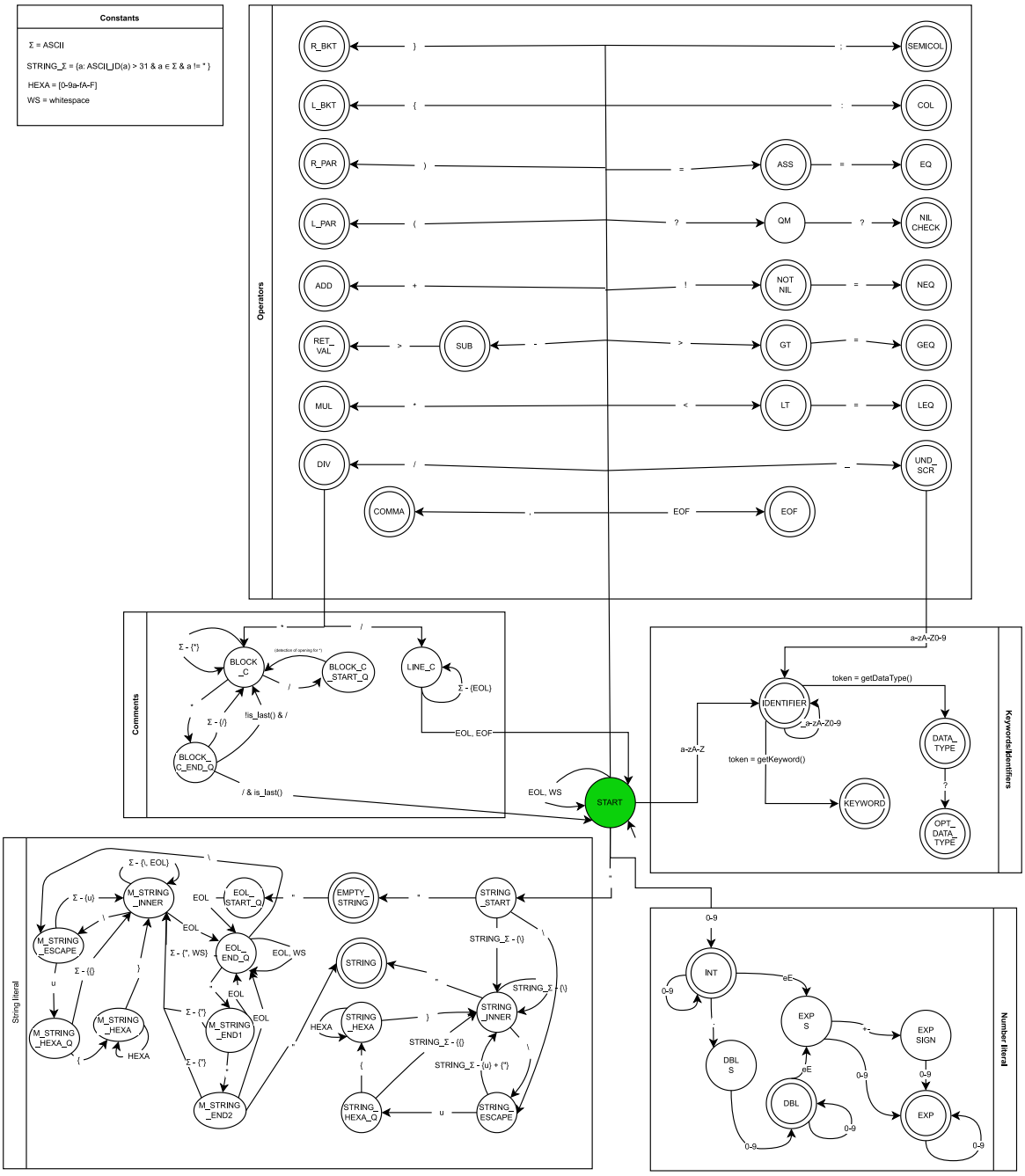

Figure 1: Finite-state machine diagram

3 Symbol Table

3.1 Implementation Overview

The symbol table[1], 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, mutability status, definition, declaration, nilability, variadic parameters for functions, and a linked list preserving parameters. The record also stores the return type for functions.

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 new symbol table of a 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**[2] (representing individual scopes), which are *created, stacked, and popped* as needed.

4 Top-Down Parser

4.1 Implementation

The parser[3] 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 ready, the parser begins two-stage syntax and semantic analysis (4.2), with the exception of expressions, which are parsed using precedence analysis in a separate module[4] (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 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[5] during recursive descend.

[1] The symbol table is implemented in the files `symtable.c` and `symtable.h`

[2] The stack for local symbol tables is implemented in the files `scope.c` and `scope.h`.

[3] `parser.c` and `parser.h` files

[4] `expression.c` and `expression.h`

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

4.1.2 Token Buffer

[6] 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 the declarations of functions, which includes adding their identifiers, parameters and type to the global symbol table. Any tokens that 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 trying 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 module `expression.c` with the corresponding header file `expression.h`. The `expr()` function is called whenever expression is expected by `parser.c` module. This function accepts pointer to parser data in `Parser` structure and returns the error code of first error found.

Expressions are processed in 4 states: shift ($<$), equal shift($=$), reduction($>$) and error reduction(empty space). Concrete state is chosen by using precedence table 5.1.2, where the row means the last terminal on stack and the column means incoming terminal in the current expression.

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

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

Expressions are reduced by appropriate rule in reduction state. During this state are moved all symbols from stack (from top to handle) to array of symbol. Maximum pushed symbols to this array are 3. Symbols from this array are used to choose the reduction rule for an operation. If there is no rule for these symbols, no rule is selected and error is processed.

5.1.1 End of expression detection

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

5.1.2 Symstack data structure

For syntax analysis is used stack implemented in `symstack.c` module. It contains inserted terminals and non-terminals of the current expression. This structure is also linked list of nodes from top to bottom. Every node contain data and pointer to previous node. In data there is stored information about concrete expression symbol, like token and basic flags that contains information about corresponding expression symbol. Expression symbol might be operand, operator or reduced expression.

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

Table 1: Precedence table

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

5.2 Semantic analysis

Semantic is controlled in reduction phase during processing current operation. For this purpose are implemented functions which names start with prefix process. If it is allowed, those functions will trigger implicit conversion of the operands. Implicit conversion is handled by `convert_if_retypeable` function.

Whether used identifiers are defined are handled by `process_operand` function. This functions uses `symtable` module in order to find out wheter the variable is defined.

6 Code generator

Lexical analyzer is implemented in module `code_generator.c` with the corresponding header file `code_generator.h`. There is no optimizer implemented in the project. Supportive functions are called from semantic analyzer and precedential analyzer. The code generator works with the stack and thanks to that there is not generated large number of compiler variables. We are able to generate most constructions directly, the exception is the declaration inside a cycle. Declaration of variables inside loops is overridden on assignments.Inside the cycle, everything that is not a variable declaration is stored in the buffer, variable declarations are written directly to stdout, after the end of the cycle, the entire content of the buffer is written to stdout, this will prevent re-declaration of variables.

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.