Formal Languages and Compilers

2024/2025

Team xvaculm00,
variant vv-BVS
FUNEXP

Daniel Pelánek (xpeland00) : 25%
Jakub Havlík (xhavlij00) : 25%
Martin Vaculík (xvaculm00) : 25%
David Bujzaš (xbujzad00) : 25%

Brno, December 4, 2024

Contents

Implementation

1.1 Lexer

Lexer creates tokens from the user's input. It's implementation is based on finite automata, where checking for errors becomes quite easy, because every sequence of symbol not leading to a final state means an error.

For loading the correct values and length of tokens, we use a "two pointer" approach. The input is first loaded into a dynamic array, which in theory is not ideal considering very big inputs.

1.1.1 Deviation from FA

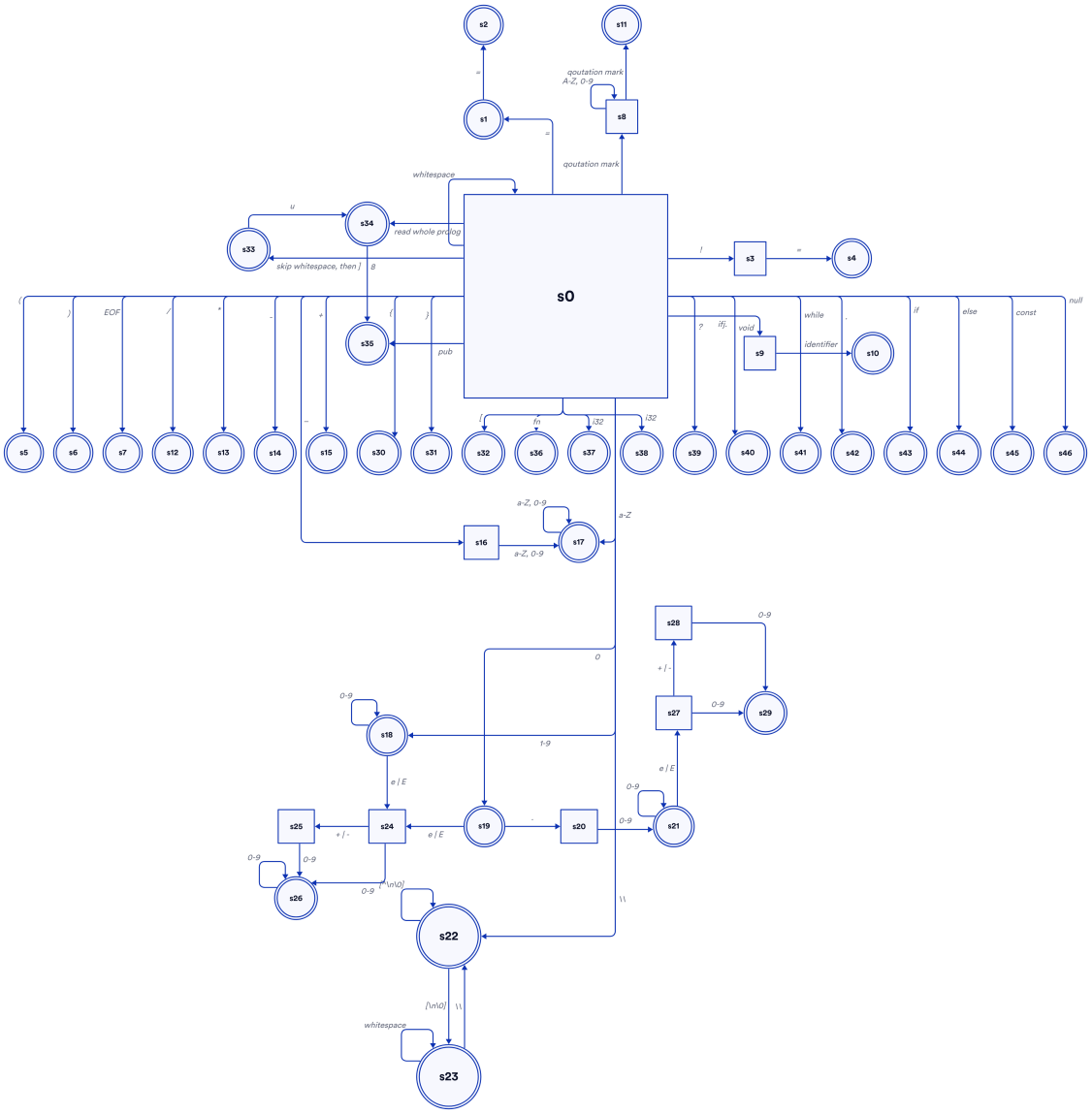
During the loading of phase of a token, we first need to distinguish, which type of token we are loading. For example if the token starts with sequence `"ifj."` we know we are loading a `TBUILDIN` token. This applies to when we load `"[]"`, we know we are loading `[]u8` token. However sometimes when we load the first characters, we cannot say for sure which token we are loading. That is why we try to load the longest token possible while distinguishing between integer, float, ID, and then we call another function that will gradually check which exact type of token it actually is (or error). This is not the exact implementation of finite automaton, because the finite automaton that we would get would be nondeterministic, however it helps us to load more clearly. Lexer is initially designed as an automaton, but this slightly different approach made the implementation more modular, but still remaining an automaton.

1.1.2 Structure of lexer, tokens, passing tokens

As previously mentioned, out lexer has two pointers, both starting at 0. Then we have an input length, to know if we would advance out of the input, and the input alone. Token contains type, value, length and two other values which will be explained in more detail in section 1.3.

Our lexer doesn't give the sequence of tokens all together, instead for a new token parser needs to call a function `get_next_token`, which has some advantages when dealing with incorrect inputs. This approach can sometimes end the compilation earlier because of syntactical error, even though there is some invalid input that can't be represented as a lexeme later in the given program.

1.1.3 Lexer FA



1.2 Symbol table

We chose the version of assignment with a height balanced binary tree. The regular way to do this is with an AVL tree but our tree of choice was red black tree, which is also a version of height balanced binary tree. We think it's objectively the slighly better choice. It is also a self balancing tree as AVL but it has faster insertions and deletions with a very slight speed downgrade for searching. Even so the main reason for choosing it was not objective. Some of our team members had previous experience with AVL trees and wanted to learn how red black trees work.

The implementation is based on a series of videos from Michael Sambol. It describes quite well the operations of red black tree and strategies used in them but does not explain them. For that we either used wikipedia or we sat in front of a piece of a paper for a long time.

The parser uses one global s-table which stores all function entries and to aggregate variable entries from nesting levels which are popped from the stack of contexts.

1.2.1 Context stack

Each context is basically just an s-table except it stores only variable entries and checks for redefinition of variables when they are inserted. A new context is pushed on the stack whenever parser goes to a new level of nesting: function declaration, if statement (else is a different context with the same nesting level) and while statement. And is popped when parser leaves the nesting level. Upon popping every variable is given a unique base64 suffix (it also has a ? at the beginning so that no variable can be defined as it) of length 8 and is inserted into the global s-table. This way code generation does not need to worry about variable scope.

1.2.2 Entry

An entry contains information about a variable or function. It contains it's type, for function it contains it's arguments and for var a can mutate flag. It also owns information for semantic analysis. Flags if a variable was used and if it was assigned. We have also implemented a more complex constant expression system as it is in zig. All expressions whose value can be inferred during compile time are evaluated. If this `comp_time` value is assigned into a variable it's value is put into entry and then all references of this var are replaced with a constant value (all peculiar rules of IFJ24, such as type conversion for arithmetic operands, are followed).

1.3 Expression Parser

Expression parser gets a sequence of tokens, that should be an expression, this is passed from parser, which determines where the expression ends. Algorithm we used to parse expressions was explained during IFJ lectures. For example expression a+b is represented by token ID, token PLUS, token ID, where values are stored in the token itself. As mentioned in section 1.1.2, token has also other other values, which are used here. Those values are `bool isProcessed`, and a pointer to AST Node. The reasoning behind those was that we need to distinguish between id that was reduced and processed, which creates a node. When `isProccesed` is set to true, the token contains also a valid pointer to a node. Then when we reduce two processed tokens such as E+E, we are a parent node plus and add nodes of both operands as it's children.

1.3.1 Structures

For our implementation we have decided to use double linked list (inspired by our project in course IAL) to implement stack. It was not mandatory, but we thought it could improve speed of some operations (inserting etc.) and use it to access more than one item on top of the stack. We use two double linked lists – one for input, the second one represents our processed part, which will be the output after the algorithm terminates. The algorithm ends when input contains only `$`, signaling end of input, and left side is reduced to a single token represented by AST node, which is also a root of the created AST sub-tree.

1.3.2 Precedence table

This section shows the precedence table implemented. Based on the precedence table expression parser chooses one of four actions, as explained on lectures. Empty places represent an error state.

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

1.4 Abstract syntax tree

Our AST is designed to have only the needed information in it. Nodes are created only for bigger parts of code. If we don't count expression nodes, which have it's own type for every operator, we use only ten node types (`func_call`, `func_decl`, `var_decl`, `if`, `while`, `else`, `nnul_var_decl`, `return`). The other information needed is stored inside of the ASTs union.

Our nodes each have two children and adhere to a logical structure. For example if a node has to contain an expression (`if`, `var_decl`, `assignment`, `while`...) the left one is always the expression. More detailed information about the nodes is documented in `ast.h`, where we show what data each AST node type can have.

1.5 Parser

The method used to parse a token stream from lexer is recursive descent which is based on our LL grammar. The code follows our LL grammar quite firmly. Most non-terminals have a corresponding function in code which decides how to apply the rule from LL grammar. These functions are also responsible for the creation of AST nodes and for chaining them together correctly.

It uses the **Parser** data structure which sees previous and next token obtained from lexer. Three functions are used to interface with this data structure. The first one is **advance** which just moves ahead one token. The second one is **check** which compares next token with given token and returns a boolean value based on the result. The last one is **consume**; it works similar to check but instead of returning a boolean it errors (with 2) if the given token does not match next.

Parser and Lexer data structures are both global variables defined in parser.c. We think it is the correct decision and believe the risk and possible slowdown, connected to having global variables, is greatly overshadowed by the improved readability, easier to work with function calls and declarations, ultimately leading to fewer errors.

The whole parser structure is inspired by Crafting Interpreters which Daniel Pelánek read through a few years back but purposefully have not looked at before the project. So, the inspiration is purely from memory.

1.5.1 LL-grammar

Every nonterminal or terminal, which has ' around it, is a single token.

The only disparity between our code and this grammar is that we don't use the stmt-id nonterminal. We can see two tokens at a time so we check it immediately and go to the corresponding nonterminal. If this were an LL(2)-grammar we could make it equivalent.

1. start → prolog (decl)*
2. decl → func-decl
3. block → (stmt)*
4. stmt → if-stmt
5. stmt → var-decl
6. stmt → assignment
7. stmt → return
8. stmt → stmt-id
9. stmt → while-stmt
10. stmt → ε
11. var-decl → 'var' id (ε | ':' type) '=' expr ';'
12. var-decl → 'const' id (ε | ':' type) '=' expr ';'
13. func-decl → 'pub' 'fn' id '(' (id ':' type ',')* (ε | (id ':' type)) ')' type '{' block '}'
14. assignment → '=' expr ';'
15. assignment → '_' '=' expr ';'
16. return → 'return' expr ';'
17. if-stmt → 'if' '(' expr ')' (ε | '|' id '|') '{' block '}' 'else' '{' block '}'
18. while-stmt → 'while' '(' expr ')' '{' block '}'
19. type → '?' type_spec
20. type → type_spec
21. type_spec → f32
22. type_spec → i32
23. type_spec → []u8
24. func-call → '(' (expr ',')*(ε | expr) ')'
25. prolog → 'const' 'ifj' '=' '@import' '(' '"ifj24.zig"' ')' ';'
26. stmt-id → id assignment
27. stmt-id → id func-call

1.5.2 LL-table

	'const'	'var'	'pub'	'_'	'?'	i32	f64	[]u8	'}'	'while'	'if'	'return'	id	'='	'('
start	1														
decl			2												
stmt	5	5		6					10	9	4	7	8		
var-decl	12	11													
func-decl			13												
assignment				15										14	
return												16			
if-stmt											17				
while-stmt										18					
type					19	20	20	20							
type_spec						21	22	23							
func-call															24
prolog	25														
block	3	3		3					3	3	3	3	3		
stmt-id														26	27

1.6 Semantic analysis

Semantic analysis is a crucial phase in the compilation process, primarily separate from the parser. The only exception to this separation is the handling of variable and function redefinitions, which are managed during parsing. The semantic analysis consists of two distinct passes:

- First Pass: In this phase, the majority of semantic checks are performed. This includes the analysis of variable declarations, function declarations, assignments, and other constructs. However, data type conversions are not handled in this pass and are deferred to the second pass.
- Second Pass: This phase focuses on data type conversions and ensures that all expressions are evaluated correctly according to the language's type rules.

In our implementation, the AST is traversed recursively after each function is parsed, and the semantic analysis is conducted according to the type of the current node.

Most of the information required for semantic analysis is derived from our Abstract Syntax Tree (AST) structure and the symbol table (symtable). The AST provides a hierarchical representation of the program's structure, while the symtable maintains information about declared variables, functions, and their types.

1.6.1 Variables

For variable declarations, the compiler verifies that the declared type matches the type of the expression assigned to it (e.g., numbers, function calls). If the types do not match, the compiler raises an error.

1.6.2 Function Declarations and Calls

Function type checking is similar to variables but focuses on the return type. The compiler ensures that the declared return type matches the type of the expression provided in the `return` statement.

- **Declarations:** The compiler examines the entire function body, validating each statement to ensure they adhere to type rules.
- **Calls:** When a function is called, the compiler checks:
  - The number of arguments matches the number of parameters in the function signature.
  - The types of the arguments correspond to the types expected by the function parameters.

1.6.3 Conditional Statements (`if` and `while`)

The type checking process for `if` and `while` statements involves the following steps:

1. Verify that the condition is either a boolean or a non-nullable variable.
2. Recursively check each statement within the associated code block to ensure consistency and correctness.

1.6.4 Constant expression evaluation

Our way of evaluating constant expressions at compile time is more complex than IFJ24 requires and is equivalent to Zig (Only when it's allowed by IFJ24). All values that can be evaluated at compile time are evaluated. When a constant value is assigned to a constant variable it's uses are then replaced by it's constant value and it can be a part of another compile time value.

1.7 Code generation

Code generator generates final code in `IFJcode24` from our AST structure. Input of code generator is an array of root nodes, each representing a function declaration. The whole tree is recursively traversed and each type of statement is accordingly generated. At the very start of the output file, header is generated and all global variables that are used for evaluating expressions, conditions and return values. After that, all built-in functions are generated. The generator expects completely valid input code, which is extensively checked by semantic analysis.

1.7.1 Function declaration

Each function declaration creates a new frame and pushes it, this ensures all defined variables can be accessed only in this frame. Then all arguments are defined as variables and correct values from stack are popped into them. Our generator uses pascal-like convention of passing parameters of functions (variant of x86 assembly conventions), so the arguments are defined from right to left. After generating all statements, the function cleans-up and pops the frame. Return value from function in our case isn't passed through frames, but rather by using a global variable for return value.

1.7.2 If-else statements and while loops

In this type of statement, we need to ensure uniqueness of generated labels and prevent redefinition of variables. For labels, the name consists of identifiers, indexes and a global label counter for each function, meaning that situation of two labels with the same name can't ever happen. The problem of redefinition is solved by predefining all variables in all nest levels outside of the loop, implemented in function `predefine_vars`. This means all variable declarations inside if-else statement or a while loop are substituted for a simple assignment. At first we wanted to use frames fix this problem, but this approach was way more complex than the current one.

1.7.3 Expressions

To generate variable declaration, assignments, return statements, function calls[1] or even conditions, we defined a generic way to evaluate an expression. The AST subtree representing the expression is recursively evaluated using the post-order traversal, which essentially means that the expressions are generated in postfix notation. All operations are being processed on data stack one by one (each in a separate call of `eval_exp`).

[1] all parameters of function calls are expressions and they are passed in pascal-like x86 convention, which was explained in ISU course

Testing

We experimented a lot with testing as we explored which paths to take. Each of us did their own simple tests with just sending needed data into units they were working on. But we also used more rigorous methods such as unit testing and created module which generates a graph for our AST.

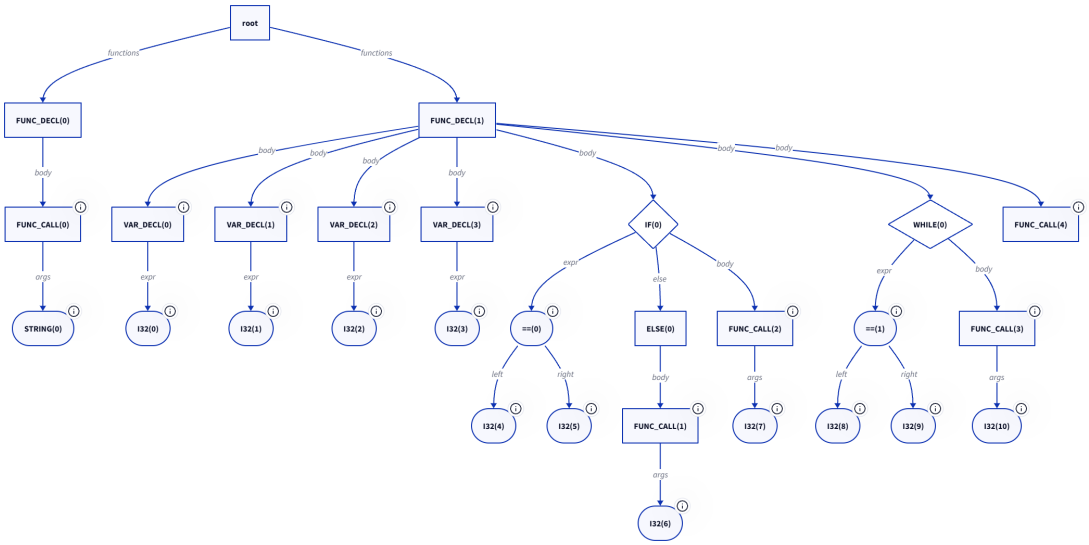
2.1 Unit testing

Our first attempts at unit testing were in google test since some of our team members had experience with it. But we weren't too happy with it because it had some rough edges. Google Test is primarily designed for C++, so integrating C code was initially fidgety until we understood it better. Additionally, the CMake pipeline was too complex for such a simple program. Later we used a testing system from some of our colleagues IFJ24-tests to which we added a lot of our own tests.

2.2 Graph generation

This module traverses the AST and during it generates graph edges and some useful information about the nodes when hovered (This function does not work in pdf even with svg). It adds type to expression nodes, variable names for identifiers, literal values etc. The node shapes were inspired by memories of flowchart programming in high school. Control statement is diamond, Pill shape is expression and rectangle is statement or anything else.

The graph is rendered with D2.



Teamwork and team organization

Immediately after receiving an assignment, we have decided who will do which part of the compiler. Even though on paper it sounds nice, this approach has some disadvantages and final product wouldn't be high quality, if we didn't work on the single parts together. We didn't use any software development methodologies, because there was no reason in doing so. We are close friends and know each other, so when problem occurs we just tell each other and fix it as soon as possible. For our communication we used discord, where we created a server for this purpose. The work was divided based on our experiences with writing different parts of compiler. This helped us to develop the program quicker, without unnecessary mistakes from not having enough experience writing compiler.

3.1 Work distribution

The initial plan was to define all structures together and implement each part of the compiler separately. We started off according to the plan, each of us worked on their part but we started hitting a few roadblocks and eventually decided it would be more productive for us if we tackled the issues at hand together. Each person still wrote their own code (assigned part of the compiler from the initial plan), but we increased the amount of consultation within the team, in order to make sure each part of our compiler worked well together. Even though our approach wasn't the most effective, it proved very useful.

Conclusion

In conclusion the takeaway from this project was greatly improved teamwork and communication, better understanding of how compilers work and in some cases each of us grew as a programmer while tackling the issues we faced.