Projektová dokumentace
Implementace překladače imperativního jazyka IFJ24

Tým **xrepcim00**, TRP-izp variant
Extensions: **FUNEXP**

December 4, 2024

Michal Repčík	**(xrepcim00)**	28 %
Alex Marinica	(xmarina00)	22 %
Šimon Bobko	(xbobkos00)	28 %
Martin Kandera	(xkande00)	22 %

Contents

1 Workflow and Team Contributions

1.1 Team Member Responsibilities

- **Michal Repčík (xrepcim00)**:
  - Designed and implemented the FSM for lexical analysis (Lexer)
  - Designed and implemented the Abstract Syntax Tree (AST)
  - Implemented the parser (excluding expression parsing)
- **Alex Marinica (xmarina00)**:
  - Designed and implemented symbol table
  - Designed and implemented semantic analysis
- **Šimon Bobko (xbobkos00)**:
  - Defined the language grammar
  - Implemented the LL parsing table
  - Developed the expression parsing module
  - Written test cases for semantic analysis
- **Martin Kandera (xkande00)**:
  - Designed and implemented code generation
  - Written test cases for code generation

The points are allocated based on each individual's participation and performance (the results from the first two test submissions). This is why members xrepcim00 and xbobkos00 received more points.

1.2 Workflow

- **Week 1 - 2**:
  - Task organization, project scheduling, and studying task requirements
  - Token, FSM and lexer design and implementation
- **Week 3 - 4**:
  - Lexer optimization and debugging
  - AST design and node creation
- **Week 5 - 6**:
  - AST and parser implementation (including expression parser)
  - Starting on semantic analysis and code generation
- **Week 7 - 9**:
  - Finishing and merging all modules
  - Testing and debugging completed compiler

2 Error Handling

To manage errors, we added an external variable, `error_tracker`, to track the current error state, and a `set_error` function to update the tracker. This function only updates if the current state is `NO_ERROR`, preventing overwriting previous errors. Error states are predefined in the `ErrorType` enumeration.

Details on error tracking can be found in `error.c` and `error.h`.

3 Lexer[4]

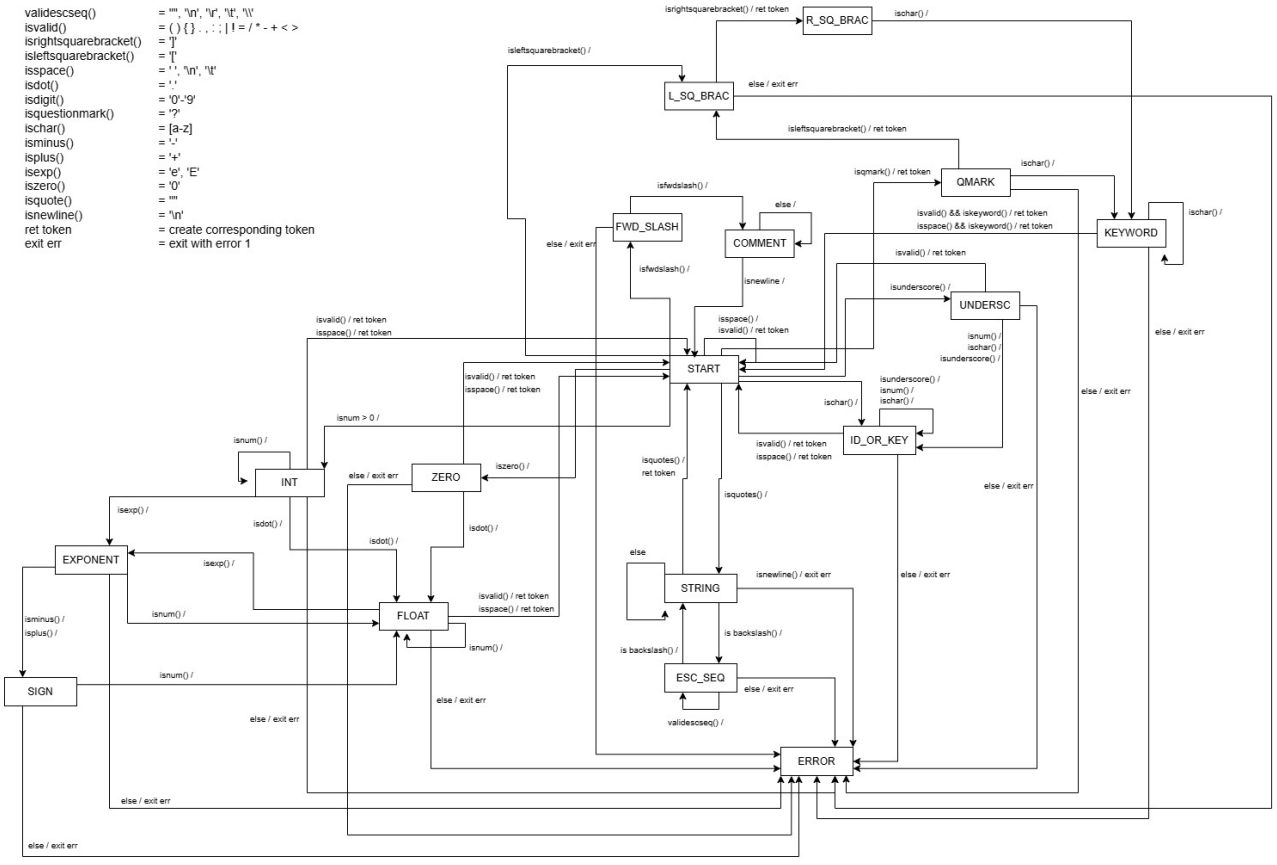
3.1 Overview

The lexer (or scanner) is the first compiler stage, converting raw source code into tokens for the parser. It operates in sync with the parser, invoked on-demand to optimize memory use and detect syntax errors faster.

3.2 Finite State Machine Representation

The lexer uses a Finite-State Machine (FSM) to identify patterns in the source code. Each state represents a step in token recognition, with transitions driven by character inputs.

The FSM diagram in higher resolution can be viewed at the following link:
https://shorturl.at/rr1NE.



3.3 Tokens

The lexer converts lexemes into tokens, each comprising a type and optionally a value. Most tokens avoid allocating memory for values unless necessary, improving efficiency.

For example, given the source code:

```
var x : ?i32 = 10;
```

The lexer generates:

Lexeme	Token Type	Token Value
var	TOKEN_VAR	NULL
x	TOKEN_IDENTIFIER	"x"
:	TOKEN_COLON	NULL
?	TOKEN_Q_MARK	NULL
i32	TOKEN_I32	NULL
=	TOKEN_ASSIGN	NULL
10	TOKEN_INT	"10"
;	TOKEN_SEMICOLON	NULL

Table 1: Example of Tokens Generated by the Lexer

Detailed token implementation can be found in `token.c` and `token.h` files.

3.4 Optimization

The lexer uses a keyword hash table and an ASCII lookup table for speed and efficiency.

3.4.1 Keyword Hash Table

The hash table distinguishes keywords from identifiers. It uses the djb2 hash function for quick lookups with perfect hashing. This design is scalable and outperformed alternatives in speed tests.

Implementation details are in `keyword_htab.c` and `keyword_htab.h`.

3.4.2 ASCII Lookup Table

A stack-based ASCII lookup table validates characters after keywords, identifiers, or terms. With only 17 elements, lookups have $O(1)$ complexity, reducing overhead.

Implementation details are in `ascii_lookup.c` and `ascii_lookup.h`.

3.5 Lexer Architecture

The lexer consists of the `Lexer` structure and `get_token` function. The `Lexer` structure includes:

- `src` – Pointer to the input source (file or `stdin`).
- `ascii_l_table` – The ASCII lookup table.
- `keyword_htab` – The keyword hash table.
- `buff` – A dynamic string storing token values.
- `buff_len` – Maximum buffer length for memory management.

`get_token` processes the source code using the FSM, creating tokens or setting an error state if tokenization fails.

Details are in `lexer.c` and `lexer.h`.

4 Parser [3]

The parser (syntax analyzer) ensures token sequences match the language grammar.

Our parser uses recursive descent, retrieving tokens with `advance_token` and validating them with `check_token`. Since the grammar is relatively simple, the parser tracks only a single instance of the current token, occasionally storing token values when necessary for Abstract Syntax Tree (AST) node creation.

Details are in `parser.c` and `parser.h`.

4.1 Grammar [5]

The grammar for the language was initially defined in Extended Backus-Naur Form (EBNF), which contains more expressive constructs and is easier to read. The implementation can be found on the next page or here.

This form of grammar was used to implement the parser. Later, it was rewritten into standard Backus-Naur Form (BNF) for the purpose of constructing the LL table.

```
<program>           ::=  <prolog> | <top_level_decl>*
<top_level_decl>    ::=  <fn_decl>
<identifier>        ::=  [a-zA-Z] [a − zA − Z0 − 9]*
<integer>           ::=  [0-9]+
<float>             ::=  <integer> "." [("e" | "E") ("+" | "-")] <integer>
<string_literal>    ::=  []*
<data_type>         ::=  ["?"] ("void" | "u8" | "i32" | "f64" | "[]u8")
<fn_decl>           ::=  "pub" "fn" <identifier> "(" [<param_list>]) ")"
                    <data_type> <block>
<param_list>        ::=  <param> ("," <param>)*
<param>             ::=  <identifier> ":" <data_type>
<const_decl>        ::=  "const" <identifier> [":" <data_type>] "=" <expression> ";"
<var_decl>          ::=  "var" <identifier> [":" <data_type>] "=" <expression> ";"
<expression>        ::=  <term> (("+" | "-") <term>)*
<term>              ::=  <factor> (("*" | "/") <factor>)*
<factor>            ::=  <number> — <identifier> — <string_literal> — "(" <expression> ")"
<number>            ::=  <integer> — <float>
<block>             ::=  "" <statement>* ""
<statement>         ::=  <identifier> <statement_suffix> — <const_decl> — <var_decl>
                    | <if_statement> — <while_statement> — <return_statement>
<statement_suffix>  ::=  "=" <expression> ";" | "(" [<arg_list>] ")";
<assignment>        ::=  <identifier> "=" <expression> ";"
<fn_call>           ::=  <fn_identifier> "(" [<arg_list>]) ")"
<fn_call_statement> ::=  <fn_call> ";"
<fn_identifier>     ::=  <built_in_fn> — <identifier>
<arg_list>          ::=  <expression> ("," <expression>)*
<built_in_fn>       ::=  "ifj" "."
                    "write" | "readstr" | "readi32" | "readf64" |
                    "i2f" | "f2i" | "string" | "length" |
                    "concat" | "substring" | "strcmp" | "ord" | "chr"
<if_statement>      ::=  "if" "(" <expression> [<bin_expression>]) ")"
                    [<element_bind>] <block> "else" <block>
<while_statement>   ::=  "while" "(" <expression> [<bin_expression>]) ")"
                    [<element_bind>] <block>
<bin_expression>    ::=  <binary_operator> <expression>
<element_bind>      ::=  "|" <identifier> "—"
<return_statement>  ::=  "return" [<expression>] ";"
<binary_operator>   ::=  "==" | "!=" | "<=" | ">=" | ">" | "<"
```

Grammar in Extended Backus-Naur From (EBNF)

LL Parsing Table

	$	pub	fn	void	u8	i32	f64	[]u8	?	(	)	,	:	;	const	var	=	+	-	*	/	{	}	ifj	if	else	while	|	return	<binary_operator>	identifier	integer	float	string_literal
<program>	1	1													1	1																		
<top_level_decl_list>	3	2													2	2																		
<top_level_decl>		4													5	6																		
<data-type>				7	7	7	7	7	7																									
<type_prefix>				9	9	9	9	9	8																									
<fn_decl>		10																																
<param_list>											12																			11				
<param_list1>											14	13																						
<param>																														15				
<const_decl>															16																			
<var_decl>																17																		
<type_assign>													18				19																	
<expression>										20																				20	20	20	20
<expression1>											23	23		23				21	22										23					
<term>										24																				24	24	24	24
<term1>											27	27		27				27	27	25	26								27					
<factor>										31																				29	28	28	30
<factor_suffix>										32	33	33		33				33	33	33	33								33					
<number>																																34	35	
<block>																						36												
<statement_list>															37	37									37		37		37		37			
<statement_list1>															38	38							39		38		38		38		38			
<statement>															40	41									43		44		45		42			
<assignment>																															46			
<assignment1>										47														48						47	47	47	47
<arg_list>										49	50																			49	49	49	49
<argument>										51																				51	51	51	51
<argument1>											53	52																						
<build_in_fn>																								54										
<if_statement>																									55									
<else_statement>															57	57							57		57	56	57		57		57			
<while_statement>																											58							
<bin_expression>											60																			59				
<element_bind>																						62						61						
<return_statement>																													63					
<return_statement1>										64				65																64	64	64	64

Figure 1: LL Parsing Table

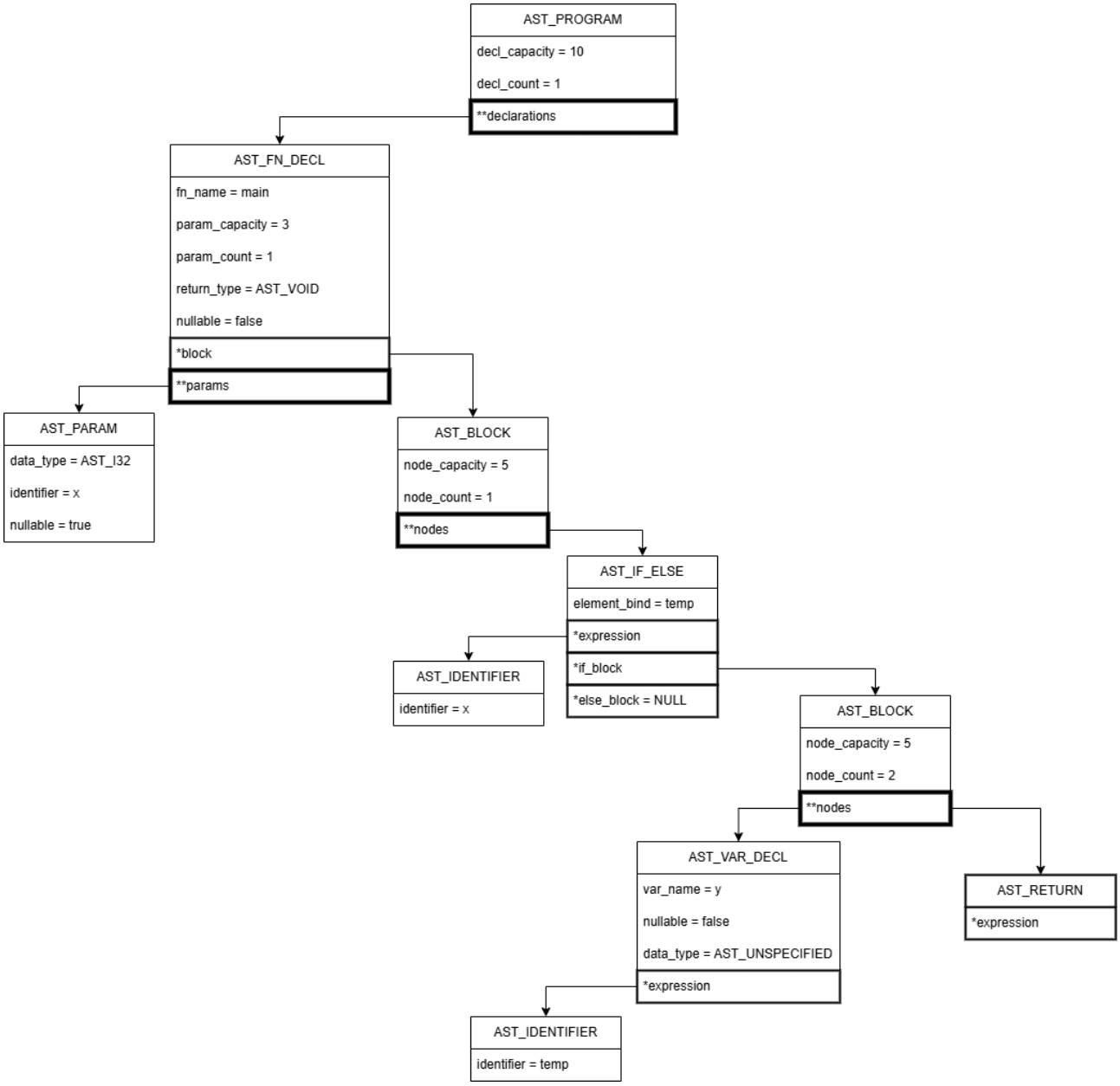
4.2 Abstract Syntax Tree [2]

The AST was designed as a collection of nodes, each representing a specific part of the code. Our AST nodes are less modular than those in typical compilers, but given the simple and immutable grammar, this design choice was practical and effective.

Below is a simple example of an AST for this code:

```
pub fn main (x : ?i32) void {
    if (x) |temp| {
        var y = temp;
        return;
    }
}
```

Note: This code is intentionally not semantically correct and serves only as a simplified example.



Complete AST implementation can be found in `ast.c` and `ast.h` files.

4.3 Expression Parser

The expression parser implementation is inspired by the precedence syntactical analysis. Expressions are parsed based on operator precedence, which is defined in the `get_precedence()` function. It uses two stacks: an operator stack, which holds AST `OperatorType`-s, and an operand stack, which holds `ASTNode`-s. Additionally, the parser supports both user-defined and built-in function calls as part of expressions.

Implementation of expression parser can be found within `parser.c` and `parser.h` files. Stack implementation can be found in `stack_exp.h`, `stack_exp.c`, `ast_node_stack.h` and `ast_node_stack.c` files.

Operator	Precedence
TOKEN_EQU, TOKEN_NOT_EQU	1
TOKEN_LESS, TOKEN_LESS_EQU, TOKEN_GREATER, TOKEN_GREATER_EQU	2
TOKEN_PLUS, TOKEN_MINUS	3
TOKEN_MULT, TOKEN_DIV	4

Table 2: Operator Precedence Table

5 Semantic Analysis[1]

The implementation of semantic analysis in this project employs semi-dynamic allocation of resources, where the symbol table uses predefined values for initial capacities and resizes dynamically as needed. The semantic analyzer relies on a recursive descent approach to traverse the Abstract Syntax Tree (AST) and perform semantic checks on various language constructs.

The **symbol table (symtable.c)** is central to the semantic analysis. It uses a hash table with the djb2 algorithm for efficient symbol storage and retrieval. Each symbol is stored as either a `VarSymbol` or a `FuncSymbol`, depending on whether it represents a variable or a function. Variables are characterized by attributes such as type, nullability, value, and usage flags. Functions include details about parameters, return type, and scope stack. The hash table expands when the load factor reaches a predefined threshold (0.75), ensuring that lookups remain efficient even as the number of entries grows.

To manage scopes, the scope stack (`stack.c`) is implemented as a dynamic array of frames. Each frame contains a local symbol table that represents a distinct block. The stack provides operations to push and pop frames, facilitating nested scopes, and supports efficient lookups by traversing from the top frame downward until a matching symbol is found.

The semantic analysis process begins by populating the global symbol table with function declarations extracted from `AST_PROGRAM`. This step ensures that functions can be used before their declarations are encountered in the AST, aligning with the assignment requirements. The global table allows deferred semantic checks, where actual validation of function bodies occurs only when the function is invoked. This optimization avoids unnecessary evaluation of unused functions, focusing semantic checks on parts of the program that affect its execution.

The analyzer ensures type compatibility, allowing null assignments only for explicitly nullable variables and permitting limited implicit conversions, like integers to floats.

The analysis of expressions involves evaluating operator types, operand compatibility, and nullability. Relational and arithmetic operators require specific type combinations, and implicit conversions are allowed only for literals. The analyzer enforces semantic correctness by raising errors for unsupported operations or mismatched types.

Function calls are validated by checking argument counts, types, and return statements. Built-in functions follow predefined rules, while user-defined functions are analyzed only when invoked, ensuring correctness and flexibility.

6 Code Generator

6.1 Overview

The code generator converts the Abstract Syntax Tree (AST) into intermediate IFJcode24, executable by a virtual machine. It supports constructs like variable declarations, expressions, loops, conditionals, and function calls, ensuring all frames remain local.

Implementation of code generator can be found in `generator.c` and `generator.h` files. Helper function for generator can be found in `generator_instructions.c` and `generator_instructions.h` files.

6.2 Structure and Components

Implemented in `generator.c` and `generator_instructions.c`, the generator relies on:

- **Local Frame Array**: Tracks variables in a dynamically managed local frame.
- **While Stack**: Handles nested `while` loops with unique labels and counters.
- **Temporary Counter**: Generates unique names for variables and labels.

6.3 Workflow

1. **Initialization**: Prepare data structures (local frame, `while` stack).
2. **AST Traversal**: Process nodes recursively, invoking routines for specific constructs.
3. **Instruction Generation**: Emit instructions for variables, control flow, expressions, and function calls.
4. **Resource Cleanup**: Clear local frames and loop stack after processing blocks.

6.4 Features

- **Variable Management**: Variables are declared with `DEFVAR` in dynamically scoped local frames.
- **Control Flow**: Constructs like `if-else` and `while` are implemented using labels and stack-based nesting.
- **Expressions and Operations**: Arithmetic and logical expressions use stack-based evaluation with instructions like `PUSHS`, `ADDS`, and `POPS`.
- **Function Calls**: Functions create a new local frame, and their execution context is restored on return.

6.5 Error Handling

Errors are limited to:

- Memory allocation failures for local frame or loop stack.
- A `NULL` AST root passed to `generate_code`.
- Encountering unsupported AST nodes during traversal.

References

[1] Dan Bernstein. *Hash Functions*. Accessed: 2024-11-26. 1991. URL: https://www.cse.yorku.ca/~oz/hash.html.

[2] Unknown. *AST representation in GCC*. Accessed: 2024-11-04. URL: https://icps.u-strasbg.fr/~pop/gcc-ast.html.

[3] Unknown. *Parsers*. Accessed: 2024-11-01. URL: https://gcc.gnu.org/wiki/HelpOnParsers.

[4] Unknown. *The Lexer*. Accessed: 2024-10-14. URL: https://gcc.gnu.org/onlinedocs/cppinternals/Lexer.html.

[5] Vexu. *zig-spec*. Accessed: 2024-10-24. 2018. URL: https://github.com/ziglang/zig-spec/blob/master/grammar/grammar.peg.