IFJ25 Compiler

Tým xcapkoe00 varianta vv-BVS

Erik Čapkovič	**xcapkoe00,** 25%
Lukáš Denkócy	xdenkol00, 25%
Adam Gajdošík	xgajdoa01, 25%
Jakub Králik	xkralij00, 25%

Implemented bonuses

ONELINEBLOCK
FUNEXP
EXTFUN

2025/2026

1 Planning of the project

For a successful completion of the IFJ 2025 project, we have carefully planned the entire development from the start to finish. Our first step was picking the variant vv-BVS which we found easier to develop, then we started to research about how compilers work via the Internet and lectures. When we had a rough idea of what an compiler is and what parts it consists of, we started to assign each member of our team a part to develop.

1.1 Division of work and responsibilities

Name	Login	Primary Responsibilities
Jakub Králik	xkralij00	Lexical analysis
Lukáš Denkócy	xdenkol00	Syntactic analysis
Adam Gajdošík	xgajdoa01	Semantic analysis
Erik Čapkovič	xcapkoe00	Code generation

We have split these responsibilities so that each of our member of the team will have 25% distribution of work.

1.2 Working in a team

Our team was created on Discord, so we personally did not know each other. To ensure a smooth development we had multiple calls on Discord discussing design decisions and overall project structure. We made a decision that we will use git for storing our code base, and each of our member will have a branch. When someone pushes something new, we will review it and test it. If the tests were correct, we would merge the push onto the main branch.

1.3 Coding rules and standards

- **Consistency:** We made sure that the code written by one team member was indistinguishable from the code written by another.
- **Maintainability:** Each function will have a detailed description of its purpose and parameters.

1.4 Plan for implementation

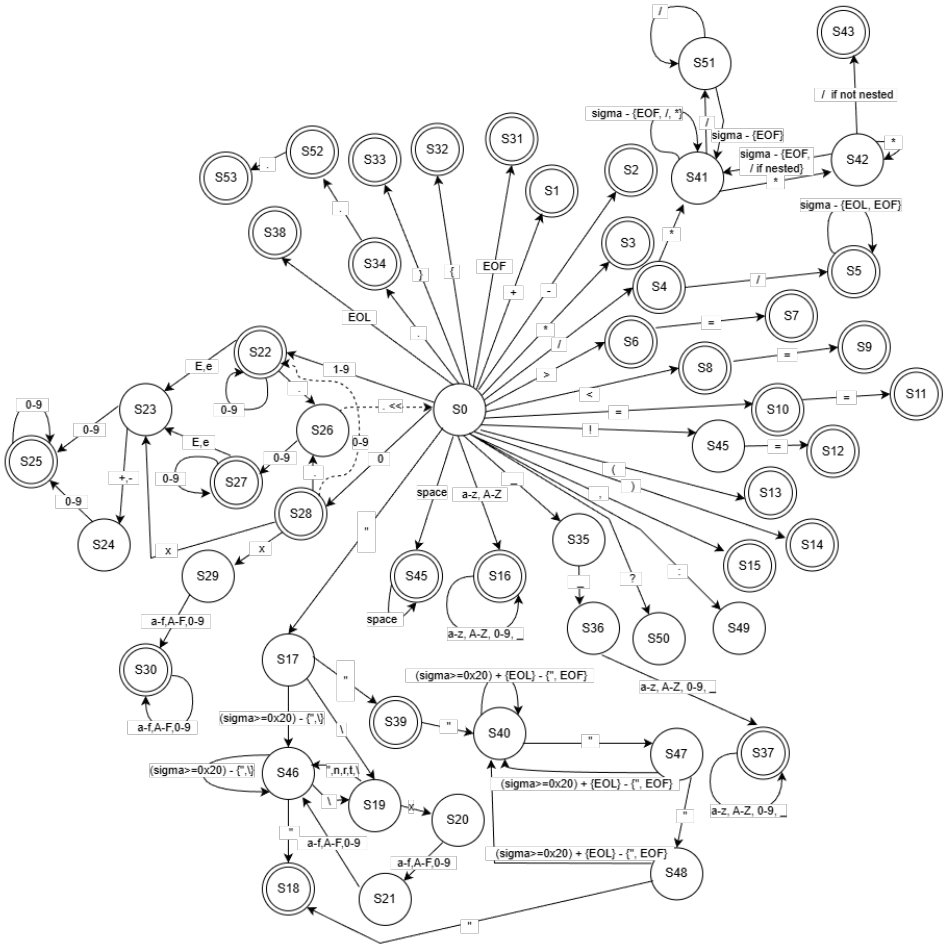
During our calls, we came to the conclusion that we will structure the development sequentially, starting with the lexical analysis implemented as a scanner in the file scanner.c, then the syntactic analysis implemented as a parser in file `parser.c` and `expr_parser.c` that will only handle expressions. Then the semantic analysis and code generation will be implemented directly in the parser. We chose this method because we will not need to create a complex abstract syntax tree, which boosts the effectiveness of the compiler. All files will have a header file, so all the other modules can use each other's functions if needed.

2 Development phase

2.1 Lexical analysis

The development of scanner was Jakub's `(xkralij00)` part. Firstly he started designing a finite state machine (FSM) which models the complete and exact behavior of the scanner by deciding a correct state transition by recognizing every terminal symbol, including complex structures like nested block comments and multi-line literals.

S0	S_START	S13	S_LEFT_PAREN	S26	S_FLOAT_START
S1	S_ADD	S14	S_RIGHT_PAREN	S27	S_FLOAT
S2	S_SUB	S15	S_COMMA	S28	S_INT0
S3	S_MUL	S16	S_ID	S29	S_NUM_HEX_START
S4	S_DIV	S17	S_STRING_START	S30	S_NUM_HEX
S5	S_SINGLE_LINE_COMMENT	S18	S_STRING_READ	S31	S_EOF
S6	S_GREATER	S19	S_STRING_BACKSLASH	S32	S_LEFT_BRACE
S7	S_GREATER_EQ	S20	S_STRING_HEX_START	S33	S_RIGHT_BRACE
S8	S_LESS	S21	S_STRING_HEX_END	S34	S_DOT
S9	S_LESS_EQ	S22	S_INT	S35	S_UNDERLINE
S10	S_ASSIGNMENT	S23	S_EXP_START	S36	S_DOUBLE_UNDERLINE
S11	S_EQ	S24	S_EXP_SIGN	S37	S_GLOBAL_ID
S12	S_NOT_EQ	S25	S_EXP	S38	S_EOL
S39	S_STRING_START2	S40	S_MULTI_LINE_LITERAL_CONTENT	S41	S_BLOCK_COMMENT
S42	S_BLOCK_COMMENT_2	S43	S_BLOCK_COMMENT_3	S44	S_NOT
S45	S_SPACE	S46	S_STRING	S47	S_MULTI_LINE_LITERAL_END1
S48	S_MULTI_LINE_LITERAL_END2	S49	S_COLON	S50	S_QUESTION
S51	S_BLOCK_COMMENT_SLASH	S52	S_DDOT	S53	S_DDDOT



2.1.1 Implementing FSM

The transition logic of the FSM is implemented in file scanner.c as a function `FSM(FILE *file, tToken token)`.

FSM function core logic is driven by a main while loop processing one symbol at a time from the file. In this loop, there is a switch statement that evaluates the symbol and determines the next state. All sequences starting with a-z A-Z are first declared as (`T_ID`). When the identifier is fully scanned, the function `isKeyword` is called. Function `isKeyword` compares token data with exact keywords like `"class", "while", "if"...` using function `strcmp`. If the function returns 0, then we set the token type to the keyword matched.

2.2 Syntactic analysis

The development of the parser was Lukaš's (`xdenkol00`) part. He started implementing the parser right after the scanner was completed. The parser's main responsibility is to validate the grammatical structure of tokens from the scanner. Parser is implemented using recursive descent.

2.2.1 LL Grammar

```
PROGRAM           :: <PROLOG> <CLASS_DEF> <EOF>
PROLOG            :: import <STRING> for <ID> <EOL>
                     IFJ25 specifically: import "ifj25" for Ifj

CLASS_DEF         :: class Program  <EOL> <FUNC_LIST>
FUNC_LIST         :: <FUNC_DEF> <FUNC_LIST_NEXT>
FUNC_LIST_NEXT    :: <FUNC_DEF> <FUNC_LIST_NEXT> | ϵ
FUNC_DEF          :: static <ID> ( <PARAMS> ) <BLOCK> <EOL>      standard function
                   | static <ID> <BLOCK> <EOL>                   getter (0 param)
                   | static <ID> = ( <ID> ) <BLOCK> <EOL>        setter (1 param)

PARAMS            :: <ID> <PARAMS_NEXT> | ϵ
PARAMS_NEXT       :: , <ID> <PARAMS_NEXT> | ϵ
BLOCK             ::  <EOL> <STATEMENT_LIST>  |  <EXPRESSION>

STATEMENT_LIST    :: <STATEMENT> <EOL> <STATEMENT_LIST> | ϵ
STATEMENT         :: <VAR_DECL>
                   | <ASSIGN>
                   | <IF_STAT>
                   | <WHILE_STAT>
                   | <RETURN_STAT>
                   | <BLOCK>
                   | <EXPRESSION>

VAR_DECL          :: var <ID>
                   | var <ID> = <EXPRESSION>
ASSIGN            :: <ID> = <EXPRESSION>

IF_STAT           :: if ( <EXPRESSION> ) <BLOCK> else <BLOCK>
WHILE_STAT        :: while ( <EXPRESSION> ) <BLOCK>
RETURN_STAT       :: return <EXPRESSION>
FUNC_CALL         :: <ID> ( <ARG_LIST> )

ARG_LIST          :: <EXPRESSION> <ARG_NEXT> | ϵ
ARG_NEXT          :: , <EXPRESSION> <ARG_NEXT> | ϵ

EXPRESSION :: parsed by precedence analysis
```

2.2.2 Precedence table

Stack \\ Input	* /	+ -	< > <= >=	== !=	is	TYPE	(	)	ID	LITERAL	FUNC	$
* /	>	>	>	>	>	E	<	>	<	<	<	>
+ -	<	>	>	>	>	E	<	>	<	<	<	>
< > <= >=	<	<	>	>	>	E	<	>	<	<	<	>
== !=	<	<	<	>	>	E	<	>	<	<	<	>
is	<	<	<	<	E	<	<	>	<	<	<	>
TYPE	>	>	>	>	>	>	E	>	E	E	E	>
(	<	<	<	<	<	<	<	=	<	<	<	E
)	>	>	>	>	>	>	E	>	E	E	E	>
ID	>	>	>	>	>	E	E	>	E	E	E	>
LITERAL	>	>	>	>	>	E	E	>	E	E	E	>
FUNC	>	>	>	>	>	E	E	>	E	E	E	>
$	<	<	<	<	<	<	<	E	<	<	<	E

This table describes the precedence relation between the terminal on the top of the stack(row) and and lookahead terminal(column)

- <: **Shift**
- >: **Reduce**
- =: **Match/Pair**
- **E**: **Syntax Error**

2.2.3 Implementation

In the implementation, we use `symtables` that store information about identifiers and `symstack`, which is a stack of `symtables` used for scope nesting and management. `Symtable` is a self-balancing binary search tree the one on the top of the stack is the current scope, and every table below is an enclosing scope. The lowest table in the stack is the global symtable.

The main function of the parser is `parse_program`. This function initializes the `symstack` and creates a global symtable and pushes it onto the stack, then starts to parse the program using the recursive descent strategy based on the LL grammar.

If the parser encounters an expression such as a variable assignment or a condition check in an if statement. The function `parse_expression` from `expr_parser.c` gets called. Using the precedence table, it will determine the order of the evaluation of the expression.

3 Semantic analysis

All static semantic checks were Adam's (`xgajdoa01`) part. Semantic analysis can be divided into two parts:

- **Static** These are the checks made during compile time (variable compatibility, redefinition of variables ...)
- **Runtime** Due to the dynamically typed nature of the language, it is often impossible to determine at compile time if operands in an operation are compatible. This can be caused by the built-in read function that takes input from the user. Implementation of this is explained in `Code generation`.

3.1 Static semantic analysis

All main semantic check functions are located in file `semantic.c`, and all the smaller ones are directly made in the `parser.c`. Symtables are the main source of information about identifiers like variables and functions. Every time a new variable is defined program runs a quick check if the variable name wasn't `defined` earlier in the current scope. Similarly, if a variable is used in an assignment or expression, the program checks if the variable was `defined` earlier in the current scope or enclosing ones. These checks prevent illegal variable definition and redefinition.

3.1.1 Function checks

Another critical part of static semantic analysis is checking all functions.

- **Definition, Redefinition** This procces is almost identical to variable however, functions with the same name can exist. It can be a normal function a getter or a setter. Because of this, we must also look up in the global scope for function type and argument count.
- **Argument count and expected argument type**

Since these checks weren't as time-consuming as we had previously planned. Adam created a doubly linked list structure to store the 3AC code and basic functions like `emit` located in `3AC.c`. Which Erik (`xcapkoe00`) can use during the development of generator.

4 Code generation

The last component of the compiler was Erik (`xcapkoe00`) part. Generator is directly implemented in `parser.c` and `expr_parser.c`. This means that the `Ifjcode25` is directly emitted as the parser proceeds, without building and using an abstract syntax tree. For the most efficient temporary variable management and creation, the code generator uses mostly `stack instructions` for arithmetic and relational calculations.

4.1 Runtime semantic checks

In some cases, we don't know variable types of operands during compile time, the generator emits instructions to validate data types later dynamically during program execution. These checks will ensure safety for all operations involving variables and verify argument types for all built-in function of `Ifj25`.

4.2 Generation syntax

Program entry point:

```
# ######################
# Program entry point
# ######################
LABEL %start
CREATEFRAME
PUSHFRAME
CALL main$0%func
POPFRAME
EXIT int@0
```

Function declaration:

```
# ####################
# Function declaration: main
# ####################
LABEL main$0%func
DEFVAR LF@%retval
MOVE LF@%retval nil@nil
# If function has parameters a, b
DEFVAR LF@%a
DEFVAR LF@%b
MOVE LF@%a LF@%param0
MOVE LF@%b LF@%param1
...
...
POPS LF@%retval
RETURN
```