Compilers Construction:

Team Members:

Karam Khaddour, Louay Farah, Saleem Asekrea, Suleiman Karim Eddin

Team Name:

KSS

Technological Stack and Language Properties

- **Source language**: Typed and Imperative language (Project I)
- **Implementation language**: C++
- **Tool for creating the parser**: Bison-based parser
- **Target platform**: LLVM bit-code

Lexer

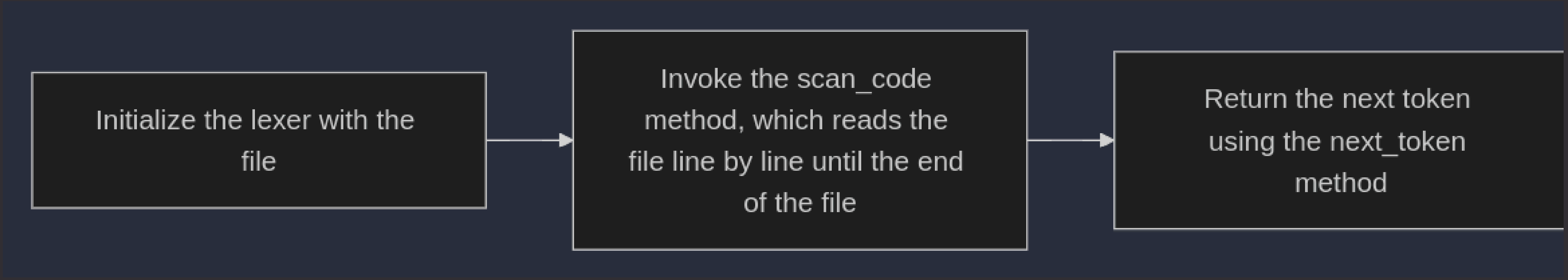
Lexer Overview

- The Lexer class is responsible for scanning the source code and breaking it into tokens, which are then passed on to the parser for syntax analysis.
- **Member Variables**:
- **tokenized_code**: A vector that stores the tokens generated from the input code.
- **error_messages**: A vector to store any errors encountered during tokenization.
- **keywords**: A map of predefined keywords and their associated token types (e.g., KEYWORD).
- **fin**: An input file stream used to read the source code.
- **code**: Holds the raw source code being tokenized.
- **ind**: A counter to keep track of the position within the code.

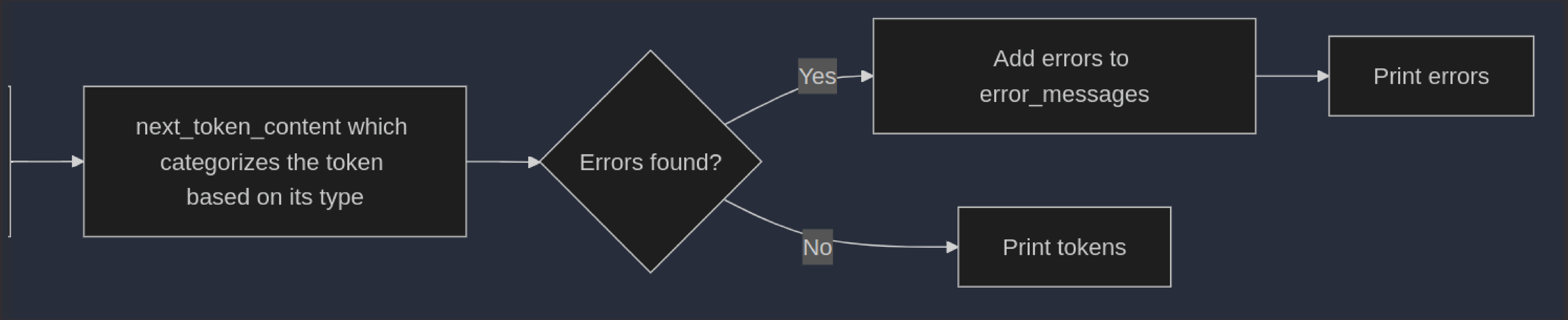
token.hpp file

Element	Description
TokenType Enum	Defines various token types such as KEYWORD, WHILE, INTEGER_LITERAL, etc.
Class Token	Represents a token with attributes:
	- TokenType type: The type of token
	- string content: The value of the token
Member Functions	- typeToString(): Converts TokenType to a string representation.
	- operator string(): Converts the token to a string for easy display.
	- operator<<: Overloads ostream for token output.
Usage	Provides structure for our lexical analysis

Flow



Initialize the lexer with the file
Invoke the scan_code method, which reads the file line by line until the end of the file
Return the next token using the next_token method



next_token_content which categorizes the token based on its type
Errors found?
Yes
Add errors to error_messages
Print errors
No
Print tokens

Token Classification Functions

- is_bracket
- is_boolean
- is_integer
- is_real
- is_literal
- is_keyword
- is_identifier : here we are using regex .
- is_pancutator
- is_operator

Token Parsing Functions

- next_token_content
- next_token
- token_type

Example Input Code

```
// 1. Add Two Numbers
routine add_numbers(a: integer, b: integer) : integer is
    var result: integer;
    result := a + b;
    return result;
end

var a : integer is 1;
var b : integer is 2;
add_numbers(a,b);
```

Example Output Tokens

```
KEYWORD routine
IDENTIFIER add_numbers
LPAR (
IDENTIFIER a
PUNCTUATOR :
KEYWORD integer
PUNCTUATOR ,
IDENTIFIER b
PUNCTUATOR :
KEYWORD integer
RPAR )
PUNCTUATOR :
KEYWORD integer
KEYWORD is
KEYWORD var
IDENTIFIER result
PUNCTUATOR :
KEYWORD integer
PUNCTUATOR ;
IDENTIFIER result
PUNCTUATOR :
OPERATOR =
IDENTIFIER a
OPERATOR +
IDENTIFIER b
PUNCTUATOR ;
KEYWORD return
IDENTIFIER result
PUNCTUATOR ;
KEYWORD end
KEYWORD var
IDENTIFIER a
PUNCTUATOR :
KEYWORD integer
KEYWORD is
INT 1
PUNCTUATOR ;
```

```
KEYWORD var
IDENTIFIER b
PUNCTUATOR :
KEYWORD integer
KEYWORD is
INT 2
PUNCTUATOR ;
IDENTIFIER add_numbers
LPAR (
IDENTIFIER a
PUNCTUATOR ,
IDENTIFIER b
RPAR )
PUNCTUATOR ;
```

Syntax Analysis

ast.hpp

Base Class

- AST_Node: Represents a generic AST node.

Attributes:

- Node_Type type: Specifies the node type.
- std::vector<AST_Node *> children: Stores child nodes.

Derived Classes

- Non termainl Node
- termainl Node

Identifier_Node, Type_Node, Boolean_Node, Integer_Node, Real_Node, Operator

ast.cpp

print_ast_helper

- Purpose: Recursively traverses the Abstract Syntax Tree (AST) and prints node information.

Parameters:

- node: The current AST node to process.
- indent: The level of indentation for nested nodes.
- output_file: The file pointer where the AST information is written.

Functionality:

- Checks for the node type and prints its corresponding information (e.g., identifier name, type, value).
- Recursively calls itself for each child node to traverse the tree and print all nodes.

print_ast

- Purpose: Initiates the printing process and writes the AST to a file.

Parameters:

- node: The root AST node to start the traversal.
- indent: The initial indentation for the root node.
- file_name: The output file name where the AST will be saved.

Functionality:

- Opens the specified file, calls print_ast_helper to process the AST, and then closes the file.

grammer.y

Union Definition (%union)

- Purpose: Defines the data types for different non-terminal symbols used in the grammar. This section links the grammar rules to the types of data they hold.
- what each grammar symbol can hold
- int int_val;

Type Definitions (%type)

- Purpose: Specifies which data types are associated with non-terminal symbols in the grammar.

Token Definitions (%token)

- Purpose: Defines the terminal symbols (tokens) for the grammar.
- %token <int_val> INTEGER_LITERAL

We followed the strucutre as in the I project

Grammar Rules Section (The Main Rules)

- Purpose: Defines the actual grammar rules for parsing the input and generating the AST.
- Each rule describes how the language's constructs (such as a program, variable declaration, or expression) are parsed.

For example:

- program rule starts the parsing process by expecting declarations.
- variableDeclaration rule defines how variables are declared in the language.
- Some rules include semantic actions (code inside curly braces {...}) that create AST nodes. For example, the rule for variableDeclaration creates a node for variable declaration and links it to the child nodes (identifier, type, and expression).

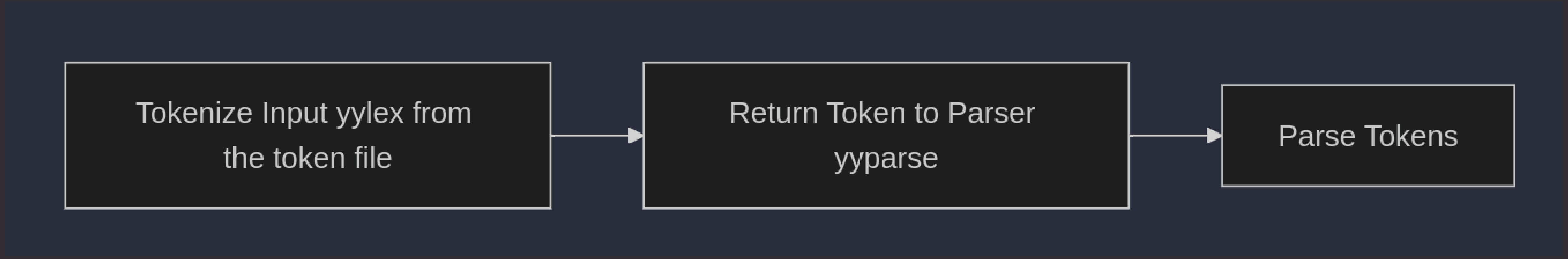
```
typeDecleration
    : TYPE identifier IS type ';' {
     $$ = new None_Terminal_Node("TYPE_DECLARATION");
     $$->children.push_back($2);
     $$->children.push_back($4);
    }
;

recordType
    : RECORD '{' variableDeclerations '}' END {
        $$ = new None_Terminal_Node("RECORD_TYPE");
        $$->children.push_back($3);
    }
;

variableDeclerations
   : variableDeclerations variableDeclaration {
        $$ = $1;
        $$->children.push_back($2);
     }
     | /* empty */ {
        $$ = new None_Terminal_Node("VARIABLE_DECLARATIONS");
     }
;
```

```
for_expression :
    FOR identifier range LOOP body END{
        $$ = new None_Terminal_Node("FOR_STATEMENT");
        $$->children.push_back($2);
        $$->children.push_back($3);
        $$->children.push_back($5);
    }
;
range :
    IN expression RANGE expression {
      $$ = new None_Terminal_Node("RANGE_EX");
      $$->children.push_back($2);
      $$->children.push_back($4);
    }
    | IN REVERSE expression RANGE expression{
      $$ = new None_Terminal_Node("RANGE_REVERSE");
      $$->children.push_back($3);
      $$->children.push_back($5);
    }
;
```

```
;
jumpStatement
  : return_exp {
    $$ = new None_Terminal_Node("JUMP_STATEMENT");
    $$->children.push_back($1);
  }
  | continue_exp {
    $$ = new None_Terminal_Node("JUMP_STATEMENT");
    $$->children.push_back($1);
  }
  | break_exp {
    $$ = new None_Terminal_Node("JUMP_STATEMENT");
    $$->children.push_back($1);
  }
;
return_exp
 : RETURN ';'{
   $$ = new None_Terminal_Node("RETURN_EX");
 }
 | RETURN expression ';' {
     $$ = new None_Terminal_Node("RETURN_EX");
     $$->children.push_back($2);
 }
;
continue_exp
 : CONTINUE ';'{
     $$ = new None_Terminal_Node("CONTINUE_EX");
 }
```



- The **yylex** function is responsible for tokenizing the input. It reads each token after using the static set_tokens function, parses it, and returns an integer corresponding to a specific token type.
- For each line, it reads the tokenType (such as IDENTIFIER, INTEGER_LITERAL) and tokenValue (the actual value).
- Based on the token value and type, it returns an appropriate token to the parser.

- **yyparse()** is called to begin parsing the tokens.
- The parser uses the tokens generated by yylex to understand and process the structure of the code.

Semantic Analysis

semantic.hpp

- Defines the Semantic Analysis class used for checking semantic correctness and optimizing the Abstract Syntax Tree (AST).

Class Declaration

- AST_Node *root - The root node of the AST.
- Semantic_Analysis(AST_Node *rootNode)
    - Initializes the semantic analysis with the AST root.
- void Semantic_Analysis_Checks(AST_Node *node)
    - Performs semantic checks on the AST.
- void optimize(AST_Node *node)
    - Optimizes the AST by removing unreachable and unused code.

semantic checks

- **check_return**: Ensures return statements are within functions.
- **check_continue**: Ensures continue statements are within loops.
- **check_break**: Ensures break statements are within loops.
- **checkVariableDeclarations**: Verifies variables are declared before use.
- **checkRoutineDeclarations**: Verifies functions are declared before use.
- **checkTypeDeclarations**: Ensures types are declared before use.

optimize

- **remove_unreachable_code**: Eliminates code after a jump statement (like return or break) that can't be executed.
- **remove_unused_routines**: Removes unused routines from the AST.
- **remove_unused_varible**: Removes variables that are declared but not used.
- **remove_unused_types**: Removes type declarations that are not referenced.

codegen

codegen.hpp

Purpose:

- The codegen.hpp file declares the Codegen class, which is responsible for converting an Abstract Syntax Tree (AST) into LLVM Intermediate Representation (IR).

- Constructor initializes the root node of the AST.
- start_llvm function triggers the LLVM code generation process.

codegen.cpp

Purpose:

- The codegen.cpp file defines the methods and functions for generating LLVM IR from the AST nodes.

code_generation:

- Implements a switch-case structure to handle different AST node types (e.g., variable declaration, loops, function calls).
- Calls specific code generation functions (Varible_Decleration_code_Gen, If_statement_code_gen, etc.).

InitializeModule:

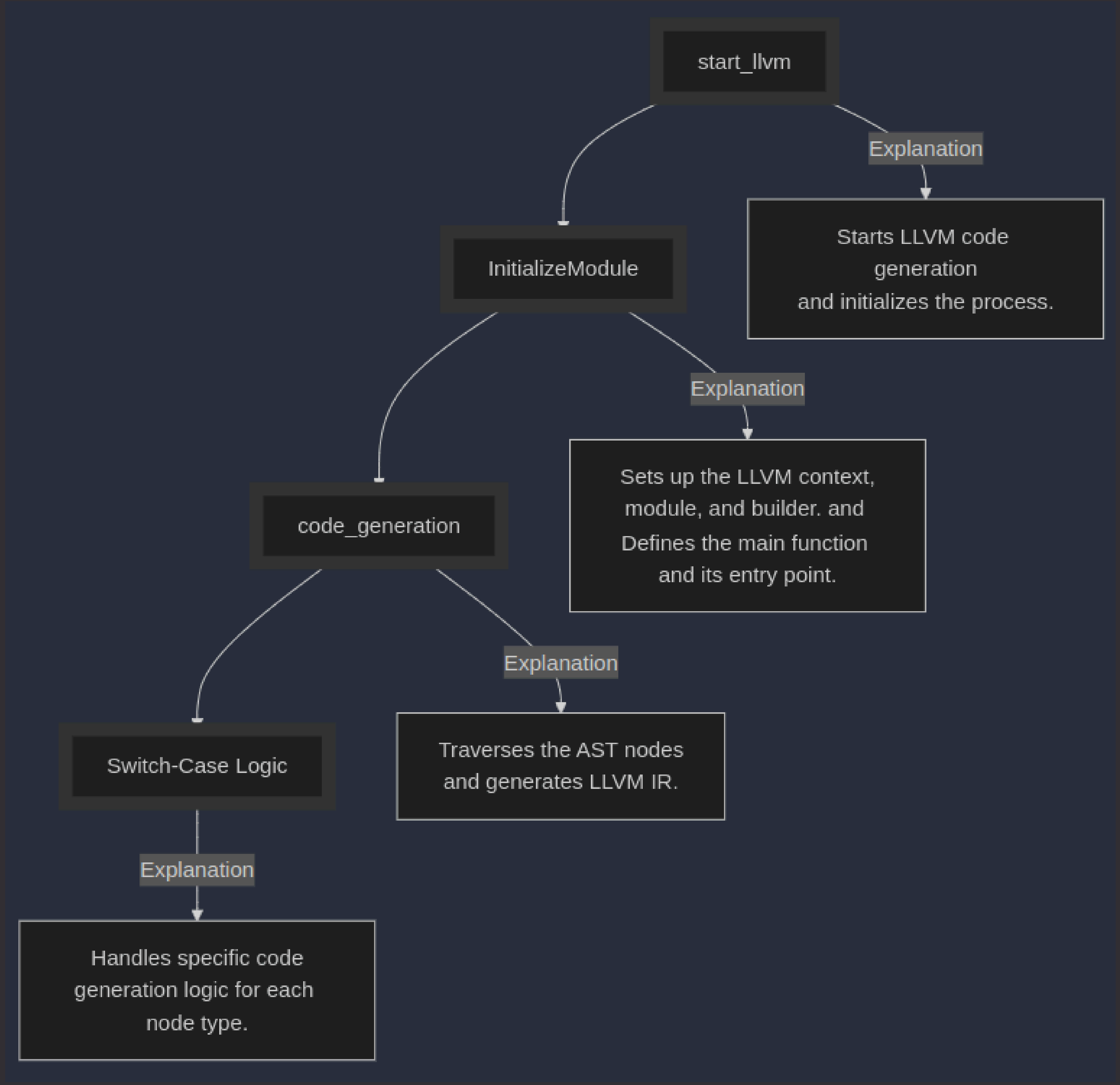
- Sets up LLVM components (LLVMContext, Module, IRBuilder).
- Defines the main function and its entry point.

start_llvm:

- Initializes the LLVM environment.
- Recursively generates code for the entire AST.

Finalizes the LLVM IR with a ret void statement and writes the generated IR to output.ll.

how it works



- Varible_Decleration_code_Gen
- Type_Declaration_codegen
- Assign_code_gen
- If_statement_code_gen
- If_else_statement_code_gen
- While_code_gen
- For_code_gen
- Routine_decleration_code_gen
- Routine_call_code_gen
- PrintNodeCodeGen

Initialize Lexer with Input File
Explanation
Lexer reads the input file and tokenizes the source code.
Parser Uses yyparse
Explanation
Parser starts using yyparse to analyze the token stream.
yyparse Invokes yylex for Tokens
Explanation
yyparse calls yylex to return tokens for parsing.
Semantic Analysis: Initialize and Perform Checks
Explanation
Semantic analysis is performed to check for errors.
Semantic Optimization: Invoke optimize
Explanation
Optimization is applied to improve the AST.
Codegen: Initialize Codegen Instance
Explanation
Codegen instance is initialized, and LLVM IR is generated.
Perform LLVM IR Generation on AST
Explanation
LLVM IR code generation is applied to the AST.
RESET

List of fully implemented language features

Code gen

- Varible_Decleration_code_Gen
- Type_Declaration_codegen
- Assign_code_gen
- If_statement_code_gen
- If_else_statement_code_gen
- While_code_gen
- For_code_gen
- Routine_decleration_code_gen
- Routine_call_code_gen
- PrintNodeCodeGen

- check_return
- check_continue
- check_break
- checkVariableDeclarations
- checkRoutineDeclarations
- checkTypeDeclarations

- remove_unreachable_code
- remove_unused_routines
- remove_unused_varible
- remove_unused_types

- nested recoreds
- nested loops
- 2D array
- print
- function that take record and return record

List of unimplemented language features

- array of recordes
- function can’t accept arrays only recoreds
- break, continue only work insde one block (loop), if there is an if inside loop it will not work we didn’t have time time to fix it .
- Variable scopes are not handeled correct 100%
- need to work more about error handling .
- semantic checks has problem with records .
- we didn’t coverd /= , *= only :=

Distribution of work in the team during the project

Team Member	Stage	Work Done
Saleem	Lexer	Built the lexer (classes and basic functionality)
Karam	Lexer	Updated the lexer to support more features and operations
Saleem	Parser	Wrote the grammar file and built the AST class, primarily focusing on variable declarations and statements.
Karam	Parser	Wrote the grammar file and built the AST class, primarily focusing on Expressions and Type declarations .
Louay	Parser	built the AST class, and connect parser with lexer
Karam	Semantic Checks	Wrote the semantic checks
Saleem	Optimization	Wrote the optimization features
Saleem	Code Generation	Worked on code generation, build code gen for for loop, while loop , continue, break,
Karam	Code Generation	Worked on code generation, build code gen for If_statement, If_else_statement, recoreds , array .
Sulieman	Code Generation	• Worked on code generation, build Assign_code_gen, Routine_decleration_code_gen Routine_call_code_gen also apply vistor pattern for the code gen (not merged wirth main branch)

Demo

Github Repository

https://github.com/saleemasekrea000/Compiler

Q&A

Thank You