



NATIONAL AUTONOMOUS UNIVERSITY OF MEXICO

FACULTY OF ENGINEERING

Computer Engineering

Compilers

Final Project

Students:

319026678

319292709

319157293

116001689

319318261

Group: 5

Semester: 2025-1

México, CDMX. November 25, 2024

Contents

1	Introduction	4
1.1	Motivation	4
1.2	Goals	4
2	Theoretical framework	5
2.1	Compiler	5
2.2	Compiler Architecture	6
2.2.1	Lexical Analysis	6
2.2.2	Syntax Analysis	6
2.2.3	Semantic Analysis	6
2.2.4	Intermediate Code Generation	6
2.2.5	Three Address Code	6
2.2.6	Code Optimization	7
2.2.7	Target Code Generation	7
2.2.8	Backpatching	7
2.3	Context-free grammar	8
2.4	Automaton	8
2.5	Deterministic Finite Automaton	8
2.6	Regular expression	8
2.7	Token	9
3	Development	9
3.1	Grammar	9
3.1.1	Expanded Grammar	9
3.1.2	Reduced Grammar	10
3.1.3	Grammar and Explanation	10
3.2	Lexer	12
3.3	Syntactic Analysis	13
3.4	PLY	14
3.4.1	lex.py	14
3.4.2	yacc.py	15
3.5	Semantic Analysis	15
3.5.1	SDT	15
3.6	Intermediate Code Generation	18
3.7	Code Optimization	18
3.8	Target code generation	20
3.8.1	Three Addres Code	20
3.8.2	addInstruction Method	21
3.8.3	Method: addInstruction _{index}	21
3.8.4	TAC quadruples	22
3.8.5	TAC Triples	22
3.9	Linker	22
3.9.1	How the PVM Works	23

4	Implementation	24
4.1	Build	25
4.2	How to run the compiler?	26
5	Result	26
5.1	Test 1	26
5.2	Test 2	26
5.3	Test 3	27
5.4	Test 4	28
6	Conclusion	30
7	References	31

1 Introduction

For this project, we will develop the compiler in a way that allows us to implement and understand each of its phases, gaining insight into how high-level code is transformed into machine-readable code. Throughout the process, we will explore how Python, along with its libraries, can be utilized for the construction of the compiler. In addition, we will delve into its internal architecture and examine the functionality of each phase in detail.

The compiler is a fundamental tool that reveals the internal processes linking programming languages to hardware. This includes lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and final code generation. The project aims to tackle the challenges of implementing each phase while addressing any issues that arise during the process, with the goal of understanding their importance and functionality.

Furthermore, it highlights the essential role compilers play in modern software development and language engineering, enabling the efficient execution of programs and making programming more accessible through high-level languages. By combining theoretical knowledge with practical implementation, this project not only emphasizes the complexities of building a compiler but also underscores its relevance, as it serves as a bridge between human-readable code and hardware execution.

1.1 Motivation

After studying each phase of the compiler and completing the Lexical, Syntax, and Semantic Analysis stages, our challenge is to continue with the remaining phases to complete the compiler development process. This goal not only represents a challenge but also an opportunity to reinforce the knowledge we have been studying throughout the course or acquiring through programming languages. It provides an opportunity to not just create programs but also to understand how they work internally, how the source code we write is processed by the computer to be executed, and how it connects to the computer's hardware.

We will gain a practical understanding of the fundamental concepts of the remaining phases, such as intermediate code generation, code optimization, and target code generation. This project will also allow us to test our abilities to analyze and resolve difficulties that arise during the compiler development process, collaborate as a team, and ultimately achieve the goal of creating a functional compiler.

1.2 Goals

1. Understanding Compiler Phases. Explore and implement the different phases of a compiler.
2. To use Python and its libraries to build the compiler, and in doing so, understand how it works and its relationship within the project.

3. Implement a semantic analyzer that ensures the source code complies with the semantic rules defined by the language.
4. Implement the intermediate code generation phase using the representation known as Three Address Code (TAC), acting as a bridge between high-level code and machine code, ensuring that transformations are clear and consistent.
5. Incorporate a code optimization phase to improve the performance and efficiency of the intermediate code.
6. Implement object code generation as the final step of the compiler, translating the intermediate code into machine-specific instructions for a target architecture.
7. The compiler must read an input file containing instructions written in a high-level programming language.
8. It must process the source code strings through lexical, syntactic, and semantic analyzers, generating intermediate representations.
9. It must use a linker to construct an object file that can be executed.

2 Theoretical framework

2.1 Compiler

A compiler is a specialized program designed to translate a source program written in one programming language (the source language) into an equivalent program in another language (the target language), such as machine code, bytecode, or even another high-level language. This translation enables the code written by a programmer to be executed either directly by the computer's hardware or within an intermediate runtime environment.

Compilers carry out this process through multiple stages, including lexical analysis, syntax analysis, semantic analysis, optimization, and code generation. Additionally, an essential function of a compiler is to identify and report any errors in the source code that are detected during the translation process, ensuring the correctness and reliability of the generated program.

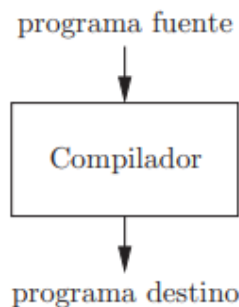


Figure 1: Compiler.

2.2 Compiler Architecture

2.2.1 Lexical Analysis

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters that make up the source program and groups them into meaningful sequences, known as lexemes. For each lexeme, the lexical analyzer produces an output token in the form: (token-name, attribute-value), which is passed to the next phase, syntax analysis.

In the token, the first component, token-name, is an abstract symbol used during syntax analysis, while the second component, attribute-value, points to an entry in the symbol table for that token.

2.2.2 Syntax Analysis

The second phase of the compiler, the parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

2.2.3 Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.

2.2.4 Intermediate Code Generation

During the translation of a source program into the object code for a target machine, a compiler often generates an intermediate representation, also known as intermediate code or intermediate text. This intermediate code is typically a low-level or machine-like representation that can be viewed as a program for an abstract machine.

2.2.5 Three Address Code

As we already know, there are two main types of intermediate code generation: graph-based representations such as the AST (Abstract Syntax Tree) and DAG (Directed Acyclic Graph), and linear representations such as Three-Address Code, the basic instruction of three-address code is designed to represent the evaluation of arithmetic expressions, although not exclusively, and has the following general form: $x = y \text{ op } z$.

The main types of three-address code statements include binary and unary assignments, such as $a := b \text{ op } c$ (binary) and $a := -b$ (unary). There are also simple copy instructions represented as $a := b$ and indexed copy instructions like $a := T[b]$ and $T[b]$

`:= a`. Unconditional jumps are defined by expressions like `goto B1`, where `B1` is a symbolic label that translates into a memory address. On the other hand, conditional jumps are used in structures like `if (a) goto B1` or `if (a < b or c < d) goto B2`, and they are commonly found in the transformation of `while` and `for` loops. Procedure calls include forms such as `call f(a, b, ...)` and assigned returns like `x := call f(a, b, ...)`. Finally, there are assignments related to memory addresses and pointers.

Three-address code uses temporary variables, which requires the implementation of a function to generate them automatically. In this context, the variables `x`, `y`, and `z` represent two operands and a result, which is where the term "three-address" comes from. These variables can be a name, a constant, or a temporary variable generated by the compiler. At the end of the process, the abstract syntax tree is converted into lines of intermediate code.

2.2.6 Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

2.2.7 Target Code Generation

This phase takes as input a program in an intermediate representation (such as stack code, three-address code, or tree-structured representations) and translates it into machine code for the target machine. The code generator maps the intermediate instructions into sequences of machine instructions that perform the same tasks. During this process, registers or memory locations are selected for the variables used in the program.

The target code generation, along with the corresponding optimizer, forms the back end of the compiler. The back end is responsible for refining the code, removing redundancies, and improving resource usage. For example, it may eliminate unnecessary temporary variables or optimize memory usage to produce efficient machine code.

Typically, the intermediate code has already gone through an initial optimization phase to reduce redundancy and improve space usage. However, additional improvements are often applied during this phase to ensure that the output machine code runs efficiently on the target machine. It's important to note that a specific back end must be written for each target machine to generate the corresponding code.

2.2.8 Backpatching

In machine code generation, two main types of code generators can be distinguished: single-pass and multi-pass generators, which differ in the number of times they need to process the intermediate code to produce the final object code. During this process, one of the most important challenges is handling forward jumps, which occur when a reference to a label appears before the label is defined.

To address this situation, careful management of labels and their corresponding memory addresses is required. When an instruction includes a jump to a label that is not yet defined, it is not possible to immediately substitute the corresponding address. The most commonly used strategies to solve this problem are as follows:

- **Backpatching (single-pass):** This method uses a table to store instructions that reference labels that have not yet been defined. Each time a label is encountered, its address is recorded in a label table. After processing all the intermediate code, the code generator revisits the table of incomplete instructions and completes the pending jumps with the correct memory addresses.
- **Multi-pass generators:** In this case, the process is divided into several stages. In the first pass, the locations where the labels are defined are identified and recorded. In subsequent passes, the references to these labels are reviewed and replaced with their corresponding addresses, thus resolving the forward jumps.

An additional alternative is the generation of assembly code. In this approach, the assembler handles forward jumps using strategies similar to those mentioned above. This method offers the advantage of delegating the resolution of a specific problem to the assembler, simplifying the design of the code generator.

2.3 Context-free grammar

It is an entirely different formalism for defining a class of languages. It is a set of formal rules that describe how to construct valid strings in a language. These grammars are widely used in the creation of programming languages and in the syntactic analysis of compilers.

2.4 Automaton

An automaton is a mathematical model for a finite state machine, in which, given an input of symbols, it transitions through a series of states according to a transition function (which can be expressed as a table). This transition function specifies which state to move to based on the current state and the symbol read.

2.5 Deterministic Finite Automaton

A Deterministic Finite Automaton (DFA) is defined as a mathematical model consisting of a set of states, an initial state, an alphabet, a set of accepting states, and a transition function. It uses the transition function to determine the next state based on the input it receives. A DFA accepts a word if the final state reached by the DFA after reading the word is one of the accepting states.

2.6 Regular expression

A regular expression is a pattern that the regular expression engine attempts to match in input text. A pattern consists of one or more character literals, operators, or constructs.

Regular expressions provide a powerful, flexible, and efficient method for processing text. The extensive pattern-matching notation of regular expressions enables you to quickly parse large amounts of text to:

- Find specific character patterns.
- Validate text to ensure that it matches a predefined pattern (such as an email address).
- Extract, edit, replace, or delete text substrings.
- Add extracted strings to a collection in order to generate a report.

For many applications that deal with strings or that parse large blocks of text, regular expressions are an indispensable tool.

2.7 Token

A token is a predefined sequence of characters that cannot be broken down further. It is like an abstract symbol that represents a unit. A token can have an optional attribute value.

3 Development

3.1 Grammar

To begin the development of our compiler, it was necessary to implement a grammar capable of correctly generating all the required strings, including blocks like the main function of a C program, conditional structures such as if, and the ability to perform arithmetic operations like addition, subtraction, multiplication, and division. Additionally, it was essential to ensure that the grammar could interpret these constructs accurately and precisely.

A significant challenge was minimizing ambiguities in the grammar. Ambiguities can create multiple ways of interpreting the same input, complicating both syntactic analysis and semantic checks. Therefore, after conducting multiple analyses, iterations, and tests, the following grammar was chosen:

3.1.1 Expanded Grammar

```
1 Program → TYPE MAIN LP RP LB StatementList RB
2
3 StatementList → Statement | Statement StatementList
4
5 Statement → VariableDeclaration EQUAL Expression SEMICOLON
6             | VariableDeclaration SEMICOLON
7             | IF LP BooleanExpression RP LB StatementList RB
8
```

```

9 VariableDeclaration → TYPE ID
10
11 Expression → Expression + Factor
12             | Expression - Factor
13             | Factor
14
15 Factor → Factor * Term
16         | Factor / Term
17         | Term
18
19 Term → LP Expression RP
20       | CONSTANT
21       | ID
22
23 BooleanExpression → BooleanTerm OR BooleanTerm
24                   | BooleanTerm AND BooleanTerm
25                   | BooleanTerm
26
27 BooleanTerm → Expression RELATION Expression

```

3.1.2 Reduced Grammar

```

1  A → TYPE MAIN LP RP LB B RB
2  B → C | C B
3  C → D EQUAL E SEMICOLON | D SEMICOLON | K LP H RP LB B RB
4  D → TYPE ID
5  E → E + F | E - F | F
6  F → F * G | F / G | G
7  G → LP E RP | CONSTANT | ID
8  H → I OR I | I AND I | I
9  I → E RELATION E
10 K → IF

```

3.1.3 Grammar and Explanation

$A \rightarrow \text{TYPE MAIN LP RP LB B RB}$

The main block of the program is generated using this production. It ensures that every program has a ‘main’ block with a defined type (e.g., ‘int’) and a body enclosed in curly braces ‘. The body can contain statements or declarations represented by ‘B’.

$B \rightarrow C \mid C B$

This production defines the list of statements or declarations that can appear inside blocks like the ‘main’ block or within an ‘if’ block. It allows either a single statement (‘C’) or multiple statements (‘C B’).

$C \rightarrow D \text{ EQUAL } E \text{ SEMICOLON} \mid D \text{ SEMICOLON} \mid K \text{ LP } H \text{ RP LB B RB}$

This production describes different types of statements: 1. ‘D EQUAL E SEMICOLON’: A variable declaration (‘D’) with an assignment (‘E’). 2. ‘D SEMICOLON’: A simple variable declaration without initialization. 3. ‘K LP H RP LB B RB’: A conditional ‘if’ statement with a condition (‘H’) and a block of statements (‘B’).

$$D \rightarrow \text{TYPE ID}$$

This production defines variable declarations, where ‘TYPE’ specifies the data type (e.g., ‘int’, ‘float’), and ‘ID’ is the variable’s identifier.

$$E \rightarrow E + F \mid E - F \mid F$$

This production handles arithmetic expressions involving addition (‘+’) and subtraction (‘-’). It also allows single terms (‘F’) to be used as expressions, adhering to left-associative evaluation.

$$F \rightarrow F * G \mid F / G \mid G$$

This production describes higher-precedence arithmetic operations: multiplication (‘*’) and division (‘/’). It ensures that these operations are evaluated before addition or subtraction, following the rules of operator precedence.

$$G \rightarrow \text{LP E RP} \mid \text{CONSTANT} \mid \text{ID}$$

This production defines the possible factors in an expression: 1. ‘(E)’: A parenthesized expression, allowing grouping for precedence. 2. ‘CONSTANT’: A numerical constant. 3. ‘ID’: A variable.

$$H \rightarrow I \text{ OR } I \mid I \text{ AND } I \mid I$$

This production defines boolean expressions, allowing the use of logical ‘OR’ and ‘AND’ operators. The precedence of ‘AND’ is higher than ‘OR’, ensuring correct evaluation.

$$I \rightarrow E \text{ RELATION } E$$

This production defines relational comparisons between arithmetic expressions (‘E’). The ‘RELATION’ operator can be ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, or ‘!=’, enabling constructs like ‘x < 5’ or ‘y == 3’.

$$K \rightarrow \text{IF}$$

This production introduces the keyword ‘IF’, which is used in conditional statements.

3.2 Lexer

Lexical analysis is the first step in the process of compiling or interpreting a program, where the input source code is divided into a sequence of tokens, each representing a fundamental element of the language's syntax. As we already know, within lexical analysis there are different types of tokens, thus, we can find the following:

- **A keyword:** like *if*, *else*, *while*, which have a special meaning in the programming language.
- **An identifier:** the name you give to a variable, function, or class, for example, *name*, *age*, *calculateArea*.
- **A literal:** a constant value, such as a number (3.14), a string ("Hello, world"), or a boolean value (`true` or `false`).
- **An operator:** a symbol that represents an operation, such as `+`, `-`, `*`, `/`, `=`.

Now, tokens serve the purpose, to some extent, of simplifying the analysis of the code. By breaking the code into tokens, the compiler can analyze it more easily and efficiently, additionally, they facilitate the construction of the syntax tree, which is a graphical representation of the program's structure, where the tokens are the nodes of this tree. It also allows error detection, if the compiler find a sequence of tokens that is invalid according to the language's rules, it can generate an error message.

Keeping this in mind, if the input of a lexical analyzer is a sequence of characters to categorize them into tokens, the output should be a precise count and classification of the valid tokens (according to the language's rules) that exist in a string of characters, or in other words, a program written in a high-level language. Below we have an example of input that we could receive for this compiler.

Listing 1: Example of source code input.

```
1 int main() {  
2     int num = 25;  
3     int count = 0;  
4  
5     if (count == 0) {  
6         count = 3;  
7     }  
8 }  
9
```

As we can see, we have a high-level code fragment in which different types of tokens appear. In the previous part we defined that there can be identifiers, keywords, literals, operators, etc. However, the way our lexical analyzer classifies tokens follows the same convention but with another notation, which is shown below:

```
TYPE, ID, MATH1, MATH2, CONSTANT, SEMICOLON,  
EQUAL, LP, RP, LB, RB, IF, MAIN, RELATION,  
AND, OR
```

In the table above, all the types of tokens that the compiler can recognize according to the defined rules are presented, in this sense, the tokens found in the source code fragment shown are classified as follows:

```
{
    int : TYPE,
    main : MAIN,
    ( : LP,
    ) : RP,
    { : LB,
    } : RB,
    num : ID,
    count : ID,
    25 : CONSTANT,
    0 : CONSTANT,
    = : EQUAL,
    if : IF,
    3 : CONSTANT
}
```

The question arises: how does a lexical analyzer identify tokens? This involves key concepts such as formal languages, regular grammars and finite automata, which are essential in its design and operation. Regular grammars, often described by regular expressions, provide rules for constructing these patterns. Thus, a lexical analyzer relies on regular expressions to define and identify tokens, such as integers, identifiers, operators and keywords, etc. In this case our grammar, which is described in the previous point, is the one that gives us the starting point so that we can create regular expressions easier than a grammar and thus begin to classify through a finite automaton if a token is recognized, accepted or simply does not comply with the defined rules.

Once we have this classification ready, and we know how many tokens there are and that they are all well accepted, we can move on to the next stage which consists of performing a syntactic and semantic analysis respectively.

3.3 Syntactic Analysis

Parsing is the phase in which the sequence of tokens produced during the lexical analysis is examined and organized according to the formal grammar language. This step constructs a syntactic structure, often represented as a parse tree, which verifies the code's syntax's correctness and prepares it for further analysis and eventual execution of compilation. By ensuring that the program follows the syntactical rules, parsing lays the foundation for understanding the program's logic and behavior.

We defined a grammar that would allow us to accept control structures, which are fundamental in a programming language. To complement this, the grammar incorporates essential constructs for variable declarations, expressions, and flow control, ensuring a comprehensive foundation for parsing and semantic analysis.

To verify the correct structure of the program as we say, we use that PLY library, specifically the yacc.py. This parser includes support for variable declarations, arithmetic

expressions, assignments, control structures, and semantic error handling. The following outlines how this analysis works and its relationship with the language syntax.

3.4 PLY

For the construction of lexer and syntactic Analysis, we use the library called "PLY", as it implements functionalities used for lexical and syntactic analysis in compilers.

PLY, also known as Python Lex Yacc, is a library that in its last version (PLY-4.0) requires Python 3.6 or newer. It is an implementation of the lex and yacc tools used to construct various phases of a compiler, such as lexical, syntactic, and semantic analysis, to replicate the functions and behavior of these tools.

As mentioned above, PLY comprises two modules, lex.py and yacc.py. The module lex.py breaks down an input text into tokens based on regular expression rules. On the other hand, yacc.py is used to accept the structure of sentences according to context-free grammar using the top-down LALR (1) strategy.

3.4.1 lex.py

The lexical analyzer turns a sequence of characters (a string) into a sequence of tokens. In the PLY tool, the types of tokens must be defined by initializing the tokens variable. For example, in our case we defined:

```
int c = 3 * b + a;
```

A tokenizer splits the string into individual tokens:

```
'int','c','=','3','*','b','+','a'
```

Tokens are usually given names to indicate what they are.

```
'KEYWORD','ID','EQUALS','NUMBER','TIMES','ID','PLUS','ID'
```

More specifically, the input is broken into pairs of token types and values. For example:

```
('KEYWORD','int'),('ID','c'), ('EQUALS','='), ('NUMBER','3'),  
('TIMES','*'), ('ID','b'), ('PLUS','+'), ('ID','a')
```

The identification of tokens is typically done by writing a series of regular expression rules.

3.4.2 yacc.py

Yacc.py is used to recognize the syntax of the language specified in the form of a context-free grammar. yacc.py uses LR parsing and generates its parsing tables using the LALR(1) algorithm (by default) or the SLR table generation algorithm.

The two tools are designed to work together. Specifically, `**lex.py**` provides an external interface in the form of a `'token()'` function that returns the next valid token from the input stream. yacc.py calls this function multiple times to retrieve tokens and invoke grammatical rules. The output of yacc.py is often an Abstract Syntax Tree (AST). However, this is entirely up to the user. If desired, yacc.py can also be used to implement simple single-pass compilers.

Like its Unix counterpart, yacc.py provides most of the features you would expect, including extensive error checking, grammar validation, support for empty productions, error tokens, and ambiguity resolution using precedence rules. In fact, everything possible in traditional Yacc should also be supported in PLY.

Since generating parsing tables is relatively expensive, PLY caches the results and saves them to a file. If no changes are detected in the input source, the tables are read from the cache. Otherwise, they are regenerated.

3.5 Semantic Analysis

Semantic analysis is a crucial phase in the compilation process, where the primary focus is to ensure that the source code adheres to the semantic rules defined by the grammar.

3.5.1 SDT

To achieve this, Syntax-Directed Translations (SDTs) are utilized, enhancing the grammar with semantic rules and associated actions. These rules specify how the meaning of a construct is interpreted based on its syntactic form. By integrating semantic checks and facilitating the generation of intermediate code during parsing, SDTs provide a structured approach for transforming high-level constructs into representations that are compatible with subsequent stages of the compilation process.

3.5.1.1 Implemented Semantic Rules for SDT

For this stage, we considered the following semantic rules:

1. **Variable Declarations:**

- A variable must be declared before it is used.
- Variables with the same identifier cannot be declared within the same scope.

2. **Assignments:**

- The type of the assigned value must match the declared type of the variable.

3. Arithmetic Operations:

- Operands in arithmetic operations must be of the correct type (e.g., do not mix `char` with `int`).

4. Conditional Statements (**if**):

- The condition must be a valid boolean expression.
- The internal block must adhere to scope and context rules.

5. Boolean Expressions:

- Comparisons must be made between operands of the same type.
- The result must always be of boolean type (`bool`).

6. Parentheses and Priorities:

- Parentheses must respect the priorities of operations.

7. Type Checking:

- Compatibility of types in expressions and declarations must be ensured.

8. Scope Management:

- Each block (`if`, functions, etc.) must respect the scope rules for variables.

PLY uses a particular approach to express these semantic rules by integrating semantic actions into the grammar productions. These actions are fundamental for the implementation of Syntax-Directed Translations (SDTs), as they allow for validations and the generation of intermediate representations during syntactic analysis.

Semantic actions are defined within the functions associated with the grammar rules, using the `p` object. This object contains the elements on the right-hand side of the production, making it easier to access the values needed for validations and transformations. Additionally, the `p[0]` attribute is used to store the synthesized results that are propagated to the parent node.

For example, validations such as data types, prior declarations of variables, and compatibility in assignments are performed directly within the semantic actions:

Listing 2: Semantic Action for Assignment

```
1 def p_assignment(p):
2     '''Statement : ID EQUAL Expression SEMICOLON'''
3     if p[1] not in symbol_table:
4         print(f"Semantic error: Variable '{p[1]}' not declared.")
5     elif symbol_table[p[1]]['type'] != p[3]['type']:
6         print(f"Semantic error: Type mismatch in \
7         assignment to '{p[1]}'.")
```


In this case:

The first validation ensures that the variable being assigned ($p[1]$) has been previously declared in the symbol table.

The second validation checks whether the type of the value being assigned ($p[3]['type']$) matches the declared type of the variable.

If either condition fails, a semantic error is reported, making this an essential part of the semantic analysis phase.

3.5.1.2 Complete SDT for the Example

```
int main() {
    int num = 25;
    int count = 0;

    if (count == 0) {
        count = 3;
    }
}
```

Fragment	Tokens	Rule	Result
int main() {	TYPE, MAIN, LB	Rule 4	Main block opens.
int num = 25;	TYPE, ID, EQUAL, CONSTANT, SEMICOLON	Rules 1, 2	num: {int, 25}.
int count = 0;	TYPE, ID, EQUAL, CONSTANT, SEMICOLON	Rules 1, 2	count: {int, 0}.
if (count == 0) {	IF, LP, ID, RELATION, CONSTANT, RP, LB	Rule 3	count == 0 valid.
count = 3;	ID, EQUAL, CONSTANT, SEMICOLON	Rules 1, 2	count: {int, 3}.
} }	RB, RB	Rule 4	No errors.

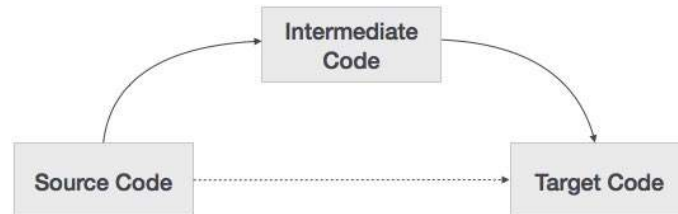
Table 1: Semantic Analysis Table for the Example

The applied semantic rules are:

1. Declare variables before use.
2. Types of assigned values must match declared types..
3. Conditions must be boolean. Operands in comparisons must have the same type.
4. Blocks must open and close properly.

3.6 Intermediate Code Generation

For this phase of our compiler, Intermediate Code Generation is the part that converts intermediate representation of source code into a form that can be readily executed by the target system.



Its main purpose is to act as a bridge that facilitates code optimization and enables the efficient generation of object code. In our case, the intermediate code is generated in a linear form using the model known as Three Address Code (TAC). This format is characterized by breaking down complex instructions into simple operations that usually involve at most three operands: two source operands and one destination operand. This approach allows operations to be expressed clearly and directly, making it suitable for transformations and optimizations.

In the next section, we will delve deeper into how the Three Address Code is structured and managed, its implementation, and the process of translating complex expressions into this intermediate representation, which is one of the final steps toward generating the final code.

3.7 Code Optimization

Code optimization is a very important stage in the compilation process, aimed at improving the quality of the generated code in terms of both efficiency and performance. It transforms the intermediate code generated by the compiler to reduce execution time, resource usage such as memory or CPU, and maximize execution speed on the target hardware, without altering the functional behavior of the program, while providing a significant performance advantage over other compilers.

There are two main types of optimization techniques, depending on whether they are related to the hardware architecture or not. First, we will discuss Machine-Independent Optimization; these techniques are applied to the intermediate code and do not depend on the hardware. Their goals are to reduce redundancy and improve the overall efficiency of the code, ensuring it is easier to translate into the target code. Some examples of this type of optimization include Constant Folding, Constant Propagation, Common Subexpression Elimination, Algebraic Simplification, and Dead Code Elimination.

As the second form of optimization, we have Machine-Dependent Optimization. These techniques take advantage of specific hardware features to maximize the performance of the program on a particular system. As before, we also have some examples, such as Register Allocation, Instruction Scheduling, and Peephole Optimization.

In the developed compiler, the following optimizations have been implemented based on theoretical foundations:

1. Constant Folding:

Evaluate constant operations at compile time and replace them with their result.

Example:

```
int a = 5 + 3;
```

Optimized as:

```
int a = 8;
```

2. Constant Propagation:

Replace known constant variables directly in expressions.

Example:

```
int a = 10;  
int b = a + 5;
```

Optimized as:

```
int b = 15;
```

3. Algebraic Simplification:

Apply algebraic identities to reduce expression complexity.

Example:

```
int x = a * 1; // Simplified  
int y = a + 0; // Eliminated
```

4. Short-Circuit Evaluation in Logical Operators:

Avoid unnecessary operand evaluation in AND and OR expressions.

Example:

```
if (false && x > 5) { ... }
```

Optimized as:

```
if (false) { ... }
```

Following the example handled so far, we would have:

```
int main() {  
    int num = 25;  
    int count = 0;  
  
    if (count == 0) {  
        count = 3;  
    }  
}
```

After optimization within the if conditionals using constant propagation and arithmetic operator reduction, we would have:

```
if (True):  
    count = 3
```

3.8 Target code generation

Target code generation is the final phase in the compilation process, where the intermediate representations produced during semantic analysis are translated into executable code for a specific architecture or platform. This phase converts the high-level language into a format that can be directly executed by hardware or interpreted by a virtual machine.

In our case, we have chosen to translate the intermediate code into a Python program. To achieve this, we utilize the three Address Code. The three-address code, TAC, is a representation that is closer to machine language or assembly. This process is carried out based on the intermediate instructions stored in the Program structure, which uses a linked list to organize the TAC. Each instruction in TAC typically consists of three operands: a destination, a source, and an operator. This format is more abstract than machine code but closer to it than high-level language constructs.

3.8.1 Three Address Code

In our implementation, the TAC instructions are contained by the program structure. These instructions are stored in a linked list, which allows for easy management and manipulation of the generated code. Each node in the list represents a single TAC instruction and the linked list provides efficient access and modification during the compilation process. This method ensures that the intermediate code can be efficiently translated into target machine code or assembly language later in the compilation process.

With the three address code we use the grammar to construct the target code. The above because each production rule generate TAC instructions during the syntactic and semantic analysis.

During the target code generation phase, each instruction in the TAC is translated into one or more instructions. The method write instructions in the Program class handles this translation. It traverses the linked list containing the TAC and generates target-specific instructions.

This process involves mapping abstract operations like addition or assignment into the corresponding target machine instructions, ensuring that operations are executed correctly on the target architecture

The Three-Address Code (TAC) instructions are dynamically generated during syntax and semantic analysis. These instructions represent an intermediate representation (IR) of the program, enabling easier translation to machine code or optimization. The TAC instructions are managed through specific methods that handle their addition or insertion in a dynamically maintained list of instructions. Here's a detailed explanation of the methods involved:

3.8.2 addInstruction Method

This method adds a new instruction to the end of the instruction list. It is commonly used for operations like:

- Arithmetic Operations: $+$, $-$, \times , \div
- Assignments: $=$
- Relational expressions: $<$, $>$, $=$

Example of usage:

```
program.instructions.addInstruction(opcode, operand1, operand2, result)
```

3.8.3 Method: addInstruction_{Index}

This method inserts an instruction at a specific position in the list of instructions, its useful for managing control flow structures, such as if, loops or function calls.

Example of usage:

```
program.instructions.addInstructionIndex(index, opcode, operand1, operand2, result)
```

Both methods allow instructions to be dynamically adjusted, ensuring flexibility in generating code for various constructs:

- Sequential Statements: Directly added using addInstruction.
- Control Flow Constructs: Inserted or modified using addInstruction_{Index}.
This structure offers a standardized way to represent each TAC instruction and ensures flexibility in handling operations with varying numbers of operands.

3.8.4 TAC quadruples

As we explained, in this implementation, the quadruple representation of TAC is utilized. A quadruple consists of four fields:

- Operator: Specifies the operation to be performed (e.g., +, -, *, if, etc.).
- Operand 1: The first operand for the operation.
- Operand 2: The second operand (if applicable) for the operation.
- Result: The location (temporary variable or final variable) where the result is stored.

For example:

```
int c = 3 * b + a;
```

We generate the Three Address Code:

```
t1 = 3 * b
t2 = t1 + a
c = t2
```

And the TAC table is represented:

Operator	Operand 1	Operand 2	Result
*	3	b	t1
+	t1	a	t2
=	t2		c

Table 2: TAC Quadruples

3.8.5 TAC Triples

A triples is a record structure with three fields, which we call OP, OP1 and OP2, where OP1 and OP2 are the arguments of OP, and they are either pointers to the symbol table or pointers to the structure itself. Since three fields are used, this intermediate code format is known as triples. This will avoid entering temporary names (such as t1, t2 etc), generated during three address code generation, into the symbol table.

3.9 Linker

One of the reasons Python is known for is the way it executes its source code (.py files). While most of us know that Python is a high-level programming language which is interpreted and not compiled like other languages, the reality is that this is not quite correct, although it is not incorrect either. When we download Python, what we are really installing in our computer is a C based interpreter named CPython, this interpreter has the purpose of performing a compilation process that although it does not translate the source code to executable machine code, it does translate it to intermediate code of lower

level than the source code, this code is known as bytecode which is stored in .pyc files, and it is the type of code that the interpreter, more specifically, the Python Virtual Machine (PVM) understands. This is how we get to the interpretation stage that most of us are familiar with. That is why the second time we run a Python program it turns out to be faster since this process of compilation or translation to bytecode is no longer performed, but is directly interpreted and executed.

CPython is then, an interpreter that allows the execution of .py files which is written in the C programming language. If we try to detail this process we would have the following.

Once we have ready the code to execute (.py file) what we normally do is: in the terminal run the command “python file.py”, and what starts to happen behind is that the operating system loads the CPython executable in memory and starts running. The input to this program is, in this case, the .py file that it is going to parse.

In this execution, CPython attempts to convert the source code into a Bytecode code that is stored in a .pyc file and in a cache folder for future executions. Once the Bytecode is generated, the Python Virtual Machine (PVM), which is nothing more than a module integrated in the CPython program, is in charge of interpreting and processing the Bytecode instructions. Should the .py code require processes involving screen printing or file manipulation, for example, CPython will call C functions to communicate with the operating system and interact with the standard output to resolve a “print”.

3.9.1 How the PVM Works

The PVM interprets the bytecode, manages its own execution stack and directly performs the necessary operations.

Suppose you have the following code:

Listing 3: Python Code

```
1 def suma(a, b):  
2     return a + b  
3  
4 x = suma(5, 7)
```

The bytecode corresponding to the above code is as follows:

```
LOAD_NAME suma  
LOAD_CONST 5  
LOAD_CONST 7  
CALL_FUNCTION 2  
STORE_NAME x
```

The Python Virtual Machine (PVM) executes the instructions in the bytecode as follows:

1. **Loads `suma` into its internal stack.**
2. **Loads the values `5` and `7` onto the stack.**
3. **Executes `CALL_FUNCTION`**, invoking the function `suma` and passing the values `5` and `7` as arguments.

4. Inside `suma`, it performs `a + b`, using the stack to temporarily store the values.
5. **Returns the result (12) and stores it in `x`.**

4 Implementation

This project, developed as the culmination of a Compiler Design course, builds upon prior work involving the creation of both a lexical analyzer and a syntax-semantic analyzer. The previously constructed lexical analyzer employed regular expressions to identify key components of the target language, such as reserved words (keywords), identifiers, punctuation, operators, constants, literals, and special characters. A deterministic finite automaton (DFA), implemented using an ASCII-based transition matrix, was utilized to detect these elements efficiently.

Regarding this project, the PLY library, specifically its ‘lex’ module, was used to construct the lexical analyzer. In ‘lex’, each token type is defined using a function prefixed with ‘t_’, followed by the token name. These functions contain regular expressions that match various elements of the language, such as keywords, operators, identifiers, and literals. Similar to the manual implementation, PLY generates a finite automaton internally to ensure a precise and efficient tokenization process.

In the second partial project, a syntax analyzer was developed based on a grammar that represented the accepted structures or statements. The top-down parsing technique was selected, and LR(0) parsing tables were created due to their compatibility with the existing environment. The approach later transitioned to SLR parsing, incorporating action, goto, and reduction rules. This phase also introduced a semantic analyzer to evaluate the logical correctness of statements.

For the current project, PLY’s ‘yacc’ module, which employs LALR(1) parsing, was utilized. This approach offered significant advantages, such as grammar expansion, reduced code complexity, and seamless integration with intermediate code generation. PLY allows for modification of production rules and inclusion of additional instructions to enhance modularity. In ‘yacc’, grammar rules are defined as functions prefixed with ‘p_’, constructing parse trees that represent the source code’s syntactic structure. Additionally, PLY resolves grammar conflicts (e.g., ambiguities) through precedence and associativity rules while analyzing input from left to right, using rightmost derivation in reverse with a lookahead symbol, and also the stack as we use in our implementation.

In summary, PLY was chosen for this project because it integrates the internal implementations developed in previous stages, offering similar strategies, methods, and functionality while streamlining the process. For this phase of the project, the focus shifted to intermediate code generation using Syntax-Directed Translation (SDT), optimization techniques, and the implementation of a linker.

Syntax-Directed Translation (SDT) involves embedding semantic actions within grammar productions to link syntactic rules with corresponding semantic operations. These actions are executed during reductions in the parsing process, allowing tasks like Abstract Syntax Tree (AST) construction, arithmetic expression evaluation, or semantic error checking to

be performed as the program's structure is built according to the grammar.

In intermediate code generation, Three-Address Code (TAC) often uses quadruples as a representation. Each quadruple consists of four components: an operator, two operands, and a result. This structure is well-suited for representing arithmetic, logical, and assignment operations, providing a clear and modular format that simplifies optimization and final code generation processes.

The quadruple table serves as a data structure to store intermediate instructions during the compilation process. Each row corresponds to an instruction, with columns for the operator, operands, and result. The compiler systematically decomposes expressions into smaller operations, generating corresponding quadruples that are added to this table, ensuring efficient tracking and management of intermediate code.


4.1 Build


For the implementation of our code, an important requirement is that the user have installed Python and the PLY library.

```
python -m pip install ply
```

The compiler development and execution process generates different essential files that play specific roles in different stages of compilation. Each of them is described below:

1. Files Generated by Using the PLY Library. This is used to perform lexical and syntactic analysis. During execution, PLY automatically generates temporary files containing the parsing tables, which are necessary for the correct functioning of the compiler. These files are created in the same directory where the main program is located and contain the precompiled information of the rules defined in the `lex.py` and `yacc.py` modules.

 `parser.out`




 `parsetab`

2. Output File. Once the compiler completes all the compiler phases, it produces an output file. `"a.py"`

 `a`

Archivo de origen Python

3. Necessary Codes for Compiler Execution. The compiler requires several additional files that implement the fundamental phases of the compilation process, such as:

 compiler linkedList tokrules

These components work together to take an input file, process it through the compiler phases, and produce an object file as the final result.

4.2 How to run the compiler?

With the above in mind, to run the compiler it is necessary to open a terminal in the location of the folder where the compiler.py file and its dependencies are stored. Next step you just execute the command:

```
python compiler.py test.txt
```

Where the python command executes the Python interpreter and we send as arguments the source file “compiler.py” which contains most of the programming logic of the compiler, followed by the name of the .txt file containing the code to compile, in this case test.txt, it is important that test.txt is also in the same folder where compiler.py is located.

5 Result

Throughout the development process, we tested our code using a variety of inputs, covering a wide range of cases that span the different phases of the compilation process. These tests allowed us to evaluate the robustness and accuracy of our lexical, syntactic, and semantic analysis, as well as the optimization and intermediate code generation processes, ensuring that the compiler’s workflow is consistent with expectations.

5.1 Test 1

In the tests conducted, we obtained consistent results that would not have been possible without the correct execution of each compiler phase, from token identification to the generation of intermediate code using quadruples. In this section, we will address some tests related to the compiler’s functionality.

5.2 Test 2

When running the compiler with the provided program, the process unfolds through several phases. In the lexical analysis phase, the code is broken down into tokens such as ‘int’, ‘main’, ‘num’, ‘=’, ‘25’, ‘count’, ‘0’, ‘if’, ‘==’, and others, corresponding to keywords, identifiers, and operators. In the syntax analysis phase, the program’s structure

```

≡ test.txt
1  int main(){
2      int num=25;
3      int count=0;
4
5      if(count==0){
6          count=3;
7      }
8  }

```

Figure 2: Input 1

is validated according to the rules of the C language, and the code passes without errors. Next, in the semantic analysis phase, the variables ‘num’ and ‘count’ are checked for correctness, and the condition of the ‘if’ statement (‘count == 0’) is verified as valid, leading to the execution of the block where ‘count’ is assigned the value 3. No semantic errors are detected. The code compiles and runs correctly, and the results are consistent with

```

a.py > ...
1  num=25
2  print(f"num = {num}")
3  count=0
4  print(f"count = {count}")
5  t1=count==0
6  if(t1):
7      count=3
8      print(f"count = {count}")
9

```

Figure 3: Output 1

expectations. The value of the *count* variable is modified within the *if* block, while the value of *num* remains unchanged. The compiler would carry out the mentioned phases without generating errors.

5.3 Test 3

When the compiler runs the provided program, it performs lexical analysis, breaking the code into tokens, identifying keywords, operators, identifiers, and values. In this case, keywords like ‘int’, ‘float’, and ‘char’ are recognized, as well as arithmetic and comparison operators. The variables ‘a’, ‘b’, ‘c’, ‘d’, and ‘h’ are identified, along with the values assigned to them. In the syntax analysis phase, the compiler checks if the structure of the code is valid according to C language rules. No syntax errors are detected, and the code is considered well-formed. During semantic analysis, the compiler ensures that the operations are valid: the variable assignments are correct, and the arithmetic operations and the ‘if’ condition are semantically consistent. Notably, the operation ‘c = 3 * b + a’ involves an implicit conversion of ‘b’ to ‘int’ before the operation, which is allowed in C.

```

≡ test2.txt
1
2  int main ()
3  {
4      int a = 45;
5      float b = 32;
6
7      int c = 3 * b + a;
8      char d = 'b';
9
10     if (a == 34)
11     {
12         d = 'r';
13     }
14
15     int h = 32 / a;
16 }

```

Figure 4: Input 2

During execution, the value of 'a' is 45, 'b' is 32, and 'c' is correctly calculated as '3 * b + a', i.e., 141. The 'if' condition '(a == 34)' is false because 'a' is 45, so the value of 'd' remains unchanged as 'b'. Finally, the division 'h = 32 / a' results in 0, since both variables are integers, and integer division discards the decimal part. Finally, the program executes correctly, with the variables initialized and modified as expected, yielding results consistent with the arithmetic and comparison operations defined in the code.

5.4 Test 4

The presented code contains several errors that would prevent it from compiling. The variable 'a' is not defined, causing errors in operations that depend on it. The variable 'b' is also used without being defined, and the assignment 'd = 18' as a 'char' could lead to interpretation issues. The conditional block 'if (a == 34)' will never execute due to the absence of 'a', making the assignment to 'd' unnecessary. Additionally, the 'printf(d)' function should use the appropriate format specifier to print a 'char'. If the *Ghost Variable* optimization were applied, the variable 'd' could be removed, as it has no impact on the result. However, the program would still fail to compile due to the undefined variables 'a' and 'b'. In conclusion, the code presents several issues that need to be addressed for successful compilation. The variables 'a' and 'b' are not defined, leading to errors in subsequent operations. Additionally, the assignment to the variable 'd' will never execute due to the condition of the 'if' statement, making it a *ghost variable* with no impact on the program. The optimization of *ghost variables* could eliminate this variable without affecting the code's functionality. However, syntax and definition errors prevent the code from running correctly.

```

a.py > ...
1  a=45
2  print(f"a = {a}")
3  b=32
4  print(f"b = {b}")
5  t1=3*b
6  t2=t1+a
7  c=int(t2)
8  print(f"c = {c}")
9  d='b'
10 print(f"d = {d}")
11 t1=a==34
12 if(t1):
13     d='r'
14     print(f"d = {d}")
15 t2=32/a
16 h=t2
17 print(f"h = {h}")
18

```

Figure 5: Output 2

```

test3.txt
1
2  int main ()
3  {
4      int c = 3 * b + a;
5      char t= 'a';
6      char d = 18;
7
8      if (a == 34)
9      {
10         d = 'r';
11     }
12     printf("%d")
13     int h = 32 / a
14 }

```

Figure 6: Input 3

```

PS C:\Users\cris\Documents\Proyecto\FinalCompiladores\FinalCompiladores> python compiler.py test3.txt
GhostVariable at line 4. No previously declared variable "b".
PS C:\Users\cris\Documents\Proyecto\FinalCompiladores\FinalCompiladores>

```

Figure 7: Output 3

6 Conclusion

In conclusion, throughout this compiler project, we implemented, studied, and delved into the various phases of a compiler, from lexical analysis to object code generation. We began with lexical analysis, Using regular expressions to identify and classify tokens, which allowed us to efficiently recognize the basic structure of the source code. Subsequently, we implemented syntax analysis through a formal grammar and production rules, utilizing a Top-down reduction strategy, specifically LALR(1), which allowed us to efficiently handle the complex structures of the language. The use of LALR(1) as a parsing strategy allowed us to efficiently handle the syntactic structures of the language, especially in the presence of ambiguities or conflicts, minimizing the number of required states and optimizing the analysis process. This method is capable of incrementally constructing the abstract syntax tree (AST), using a single lookahead symbol to make decisions about which productions to apply at each step.

Semantic analysis was built upon syntax analysis, verifying the program's logic, such as type compatibility and the consistency of variables in operations. For intermediate code generation, we used TAC (Three Address Code), where we were able to compare the differences between quadruples and triples, and also implemented optimization techniques, such as detecting ghost variables and managing memory (OutOfMemory), to improve the efficiency of the generated code. In this project, TAC played a crucial role by providing an abstract way to represent the program, which allowed us to perform several optimizations before generating machine code or the final code. One of the advantages of using TAC is that it facilitates the implementation of optimizations, as it provides an intermediate representation that is easier to manipulate and analyze compared to the source code.

During the optimization phase, we applied several techniques to improve the efficiency of the generated code, with one of the most notable being the elimination of `*ghost variables*`. In this context, if a variable is declared but never used, it is removed from the intermediate code, reducing unnecessary memory and resource usage. This not only enhances the code's efficiency but also improves its readability by removing redundant elements.

Additionally, we conducted research on how Python executes programs and learned to use the symbol table to facilitate both semantic analysis and target code generation. This gave us a deeper understanding of the compilation process, from input to final execution.

When comparing this final project to the previous partial projects, where we manually implemented the various phases of the compiler in C without using any libraries, we noticed a significant difference. The use of the `ply` library for implementing the different components of the compiler streamlined the process, improved efficiency, and allowed the inclusion of more functions and tokens. It also gave us the opportunity to evaluate our implementation, and we observed that it was not far from these professional solutions, but it also highlighted areas for improvement in optimization and resource management.

7 References

References

- [1] A . V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, "*Compilers. Principles, Techniques And Tools*". 2nd ed. Pearson Education. United States of America.
- [2] "Express Learning: Principles of Compiler Design". O'Reilly. [Online]. Available: <https://www.oreilly.com/library/view/express-learning-principles/9788131761267/chap007.xhtml>
- [3] "The Phases of a Compiler Lexical Analysis - javatpoint - www.javatpoint.com." [Online]. Available: <https://www.javatpoint.com/the-phases-of-a-compiler-lexical-analysis>
- [4] OpenAI, "ChatGPT," OpenAI, [Online]. Available: <https://chatgpt.com/>
- [5] Microsoft, ".NET regular expressions" [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions>
- [6] Universidad Europea de Madrid, Intermediate Code Generation: Three-Address Code, Laureate International Universities, [Online]. Available: <https://www.cartagena99.com>
- [7] Universitat Jaume I, "II26 Language Processors: Code Generation," Intermediate Code Generation, [Online]. Available: <https://www3.uji.es/vjimenez/AULASVIRTUALES/PL-0910/T5-GENERACION/codigo.apun.pdf>
- [8] Shaw, A., "Your Guide to the CPython Source Code," Compiling CPython on Windows, (2023, 21 octubre), [Online]. Available: https://realpython.com/cpython-source-code-guide/?utm_source=chatgpt.comcompiling-cpython-windows
- [9] K. D. Cooper, L. Torezon, "Engineering a compiler". 2nd ed. Morgan Kaufmann. London, England.