



Universidad Nacional Autónoma de México

Computer Engineering

Compilers

Students

Names	Account Numbers
Cuevas Quintana Amir	320166866
Guerrero Lopez Uriel Ivan	320046045
Perez Rojo Abraham	320149517
Sandoval Hernandez Darinka	320137156
Sanchez Mayen Tristan Qesen	320187072

Group:

5

Semester:

2025 - 2

Mexico, CDMX. June 6th 2025

Contents

1	Introduction	3
2	Theoretical Framework	3
2.1	How YACC Works	3
2.1.1	Grammar Rules	3
2.1.2	AST	3
2.2	Tracking Line Numbers and Positions	4
2.3	Embedded Actions	4
2.4	Parsers	4
2.5	Debugging and Logging	4
2.6	Python -OO Mode	5
2.7	Type of parsing	5
3	Body	5
3.1	Grammar	5
4	Results	8
5	Conclusions	13

1 Introduction

- **Problem Statement:** Develop a parser that constructs abstract syntax trees for semantic analysis using a programming language and an appropriate parsing library.
- **Motivation:** The parser is an important component for semantic analysis, which is an essential phase in the creation of a compiler which is the main objective of this course.
- **Objectives:** Design and implement a program that is theoretically capable of generating abstract syntax trees suitable for semantic processing.

2 Theoretical Framework

2.1 How YACC Works

YACC stands for Yet Another Compiler Compiler, its function is the part that takes a grammar and a list of tokens and figures out how they fit together. In PLY, we define grammar rules using functions and docstrings.

2.1.1 Grammar Rules

Each grammar rule is a Python function where the rule itself goes in the docstring, e.g:

```
def p_expression_plus(p):  
    'expression : expression PLUS expression'  
    p[0] = p[1] + p[3]
```

`p[i]` is how you access the tokens and parts of the rule. `p[0]` is the result of the rule.

2.1.2 AST

We can build an AST (Abstract Syntax Tree) while parsing by either: Returning nested tuples/lists or defining your own classes to represent nodes in the tree.

For example, if we use classes like we do in Java:

```
class BinOp:  
    def __init__(self, left, op, right):  
        self.left = left  
        self.op = op  
        self.right = right
```

2.2 Tracking Line Numbers and Positions

PLY (YACC module) provides a feature that allows to identify where the errors happen, these are divided by types:

- `p.lineno(i)`: line number of symbol `i`.
- `p.lexpos(i)`: character position of symbol `i`.
- `p.linespan(i)`: tuple with starting and ending line numbers.
- `p.lexspan(i)`: tuple with starting and ending positions.

This is only available if we parse with tracking enabled, we do this with:

```
parser.parse(data, tracking=True)
```

2.3 Embedded Actions

If you want to execute some code when a specific token is matched, even if it's not part of the final result, here we can use the empty rules function that YACC provides.

```
def p_statement_block(p):
    'statement : LBRACE new_scope statements RBRACE'
    pop_scope()

def p_new_scope(p):
    'new_scope :'
    push_scope()
```

We can use this for creating or destroying scopes on languages like C or Java.

2.4 Parsers

We can build and use multiple parsers in one program if need to. We just need store them as objects and pass them explicitly:

```
parser = yacc.yacc()

parser.parse(text, lexer=lexer) /In case we also want to use multiple lexers
```

2.5 Debugging and Logging

PLY library use the debugging tool that Python has by default:

```
import logging
logging.basicConfig(filename="parselog.txt", level=logging.DEBUG)

parser.parse(text, debug=True)
```

2.6 Python -OO Mode

PLY use docstrings to define grammar rules. If we run Python with -OO (optimize and remove docstrings), the grammar won't work. One workaround is setting the docstring manually using decorators:

```
def _(doc):
    def decorate(func):
        func.__doc__ = doc
        return func
    return decorate

@_("expr : expr PLUS expr")
def p_expr(p):
    ...
```

2.7 Type of parsing

YACC uses a LALR(1) parser, which stands for Look-Ahead LR with 1 lookahead token and is efficient and does not slow down as the grammar grows. The biggest performance is because of the lexical analysis, heavy semantic actions or big AST construction and line/position tracking (when tracking=True).

3 Body

PLY uses LALR(1) parsing with a single token lookahead, processing the token stream from the previously project Lexer.

We built the parser using YACC, and it teams up with our lexical analyzer to break down the input code's structure, getting it ready for semantic analysis. We followed a Bottom-Up parsing approach using an LALR(1) parser generated by PLY, which is conceptually similar to traditional YACC parsers. This method builds the parse tree starting from the tokens and progressively reduces them into higher-level syntactic structures, based on the defined grammar.

The following is the simplified grammar used to implement the parser:

3.1 Grammar

```
programa      → INT ID ( ) { statements }
statements    → statements statement | statement
statement     → declaration | assignment | if_statement | printf | return ;
declaration   → tipo ID ;
tipo          → INT | FLOAT | CHAR | DOUBLE | LONG | SHORT
assignment    → ID = expression ;
expression    → expression + term
              | expression - term
              | expression > term
```

```

| expression < term
| expression == term
| term
term      → term * factor | term / factor | factor
factor    → NUMBER | ID
if_statement → IF ( expression ) { statements } ELSE { statements }
printf     → PRINTF ( STRING ) ;
return     → RETURN expression | RETURN ;

```

We developed the parser directly from a context-free grammar (CFG), which defines the valid rules of the source language. Each of these rules was then implemented as a Python function, enabling the parser to do a few key things:

- Groups tokens from the previous lexer into syntactic structures such as statements, expressions, and control structures.
- Constructs an Abstract Syntax Tree (AST), in which each node represents a language construct. For example, arithmetic expressions are modeled as binary operations, and declarations are modeled as nodes containing the type and identifier. These structures are generated as nested tuples that are then used to generate the visual tree with Graphviz where a PNG image is created showing the hierarchy of the source code.
- Applies correctly operator precedence and associativity, which allows for the resolution of ambiguities in mathematical or logical expressions.

In addition to syntactic analysis, we implement basic semantic validation. It keeps track of declared variables in a symbol table, which is essentially a Python dictionary called `variables`. This makes sure you can't use a variable until it's been declared and given a value. When you assign something to a variable, its value in this dictionary gets updated right away during parsing. The `if` statement which is handled by the `p_if_statement` function, evaluates its condition. Then, it uses an auxiliary function called `execute` to run the correct part of the code, whether it's the `then` or `else` branch. The `return` statement is supported with or without an expression and is processed by the `p_return` function. Similarly, `printf` statements are managed by `p_printf` letting you display text.

This compiler can accept a simplified version of the C language. The supported symbols include variable declarations with basic data types (`int`, `float`, `char`, `double`, `long`, `short`), arithmetic and relational operators, assignment statements, conditional `if-else` statements, and simple function calls such as `printf` and `return`. The valid structure of the source code consists of a single function header of the form `int main()`, followed by a block of statements enclosed in braces. Within this block, variable declarations, assignments, expressions, and conditional structures can be used freely. The use of the `if` structure is central to the parser's design and is fully supported with both `then` and `else` branches.

The parser also includes basic syntax error handling, implemented through the `p_error` function. When a syntax error is detected, the parser generates a descriptive message that indicates the problematic token, its line number, and column position within the source code. These error messages are stored in the `erroresPAR` list for later reporting. If the error occurs at the end of the input, a specific message is produced to suggest possible missing braces or semicolons. This mechanism provides valuable feedback during parsing and helps users correct structural issues in their source code.

The parser operates based on a structured grammar, implemented through parsing functions such as `p_program`, `p_statements`, `p_statement`, and `p_expression_binop`. This grammar supports sequences of statements, arithmetic and relational expressions, variable declarations and assignments, and basic I/O operations through `printf`. The `if` structure, with mandatory `else` handling, demonstrates the parser's capability to process branching logic and perform dynamic evaluation during parsing.

The compiler was tested with multiple code examples covering variable declarations, assignments, arithmetic and relational expressions, `if-else` structures, and combinations of `printf` and `return` statements. The parser successfully generated the corresponding abstract syntax trees for all test cases, with graphical representations produced using the `build_graph` function to validate the correctness of the parsing process. Additionally, object code was generated for valid input programs through integration with the external module `Asm_Obj`, demonstrating the complete pipeline from lexical analysis to parsing, semantic validation, and code generation.

4 Results

The compilation process begins with the user providing a source code written in C. This code serves as the input that will go through the different phases of the compiler.

```
int main(){
    int x;
    x = 10 * 2 + 1;
    if (x>10){
        printf("x es mayor a 10");
    }else{
        printf("x es menor a 10");
    }
}
```

Figure 1: Source Code Input

The following image shows the grammar generated by Yacc after parsing the source code written in C. This grammar represents the syntax rules used by the parser to validate and structure the input program. Each rule is labeled and follows a bottom-up structure that breaks the program into statements, declarations, expressions, etc.

```
3 Grammar
4
5 Rule 0 S' -> program
6 Rule 1 program -> INT ID LPAREN RPAREN LBRACE statements RBRACE
7 Rule 2 statements -> statements statement
8 Rule 3 statements -> statement
9 Rule 4 statement -> declaration
10 Rule 5 statement -> assignment
11 Rule 6 statement -> if_statement
12 Rule 7 statement -> printf
13 Rule 8 statement -> return SEMICOLON
14 Rule 9 statement -> return
15 Rule 10 declaration -> tipo ID SEMICOLON
16 Rule 11 tipo -> INT
17 Rule 12 tipo -> FLOAT
18 Rule 13 tipo -> CHAR
19 Rule 14 tipo -> DOUBLE
20 Rule 15 tipo -> LONG
21 Rule 16 tipo -> SHORT
22 Rule 17 assignment -> ID EQUALS expression SEMICOLON
23 Rule 18 expression -> expression PLUS term
24 Rule 19 expression -> expression MINUS term
25 Rule 20 expression -> expression GT term
26 Rule 21 expression -> expression LT term
27 Rule 22 expression -> expression EQUALS term
28 Rule 23 expression -> term
29 Rule 24 term -> term TIMES factor
30 Rule 25 term -> term DIVIDE factor
31 Rule 26 term -> factor
32 Rule 27 factor -> NUMBER
33 Rule 28 factor -> ID
34 Rule 29 if_statement -> IF LPAREN expression RPAREN LBRACE statements RBRACE ELSE LBRACE statements RBRACE
35 Rule 30 printf -> PRINTF LPAREN STRING RPAREN SEMICOLON
36 Rule 31 return -> RETURN expression SEMICOLON
37 Rule 32 return -> RETURN SEMICOLON
```

Figure 2: Grammar generated by YACC

As seen in Figure 2, the grammar includes rules for variable declarations, expressions, assignments, and conditional statements. This structure provides

the foundation for both syntax checking and abstract syntax tree generation.

After parsing the input code using the grammar defined in the previous section, Yacc generates an Abstract Syntax Tree (AST). This tree represents the hierarchical syntactic structure of the source program.

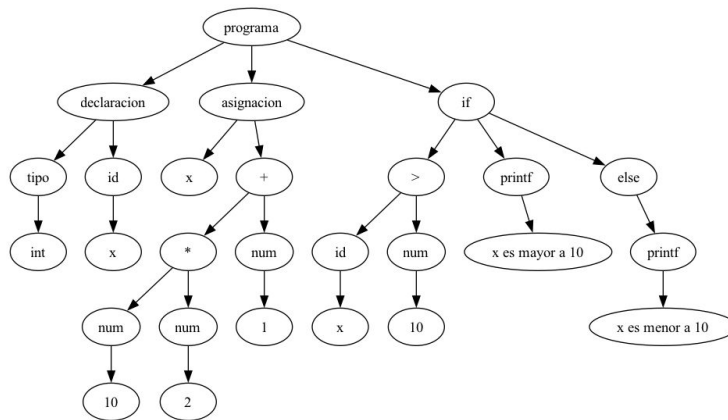


Figure 3: Abstract Syntax Tree (AST) generated from the input code

During the parsing process, several common syntax errors can be detected. The following examples illustrate typical mistakes and their explanations:

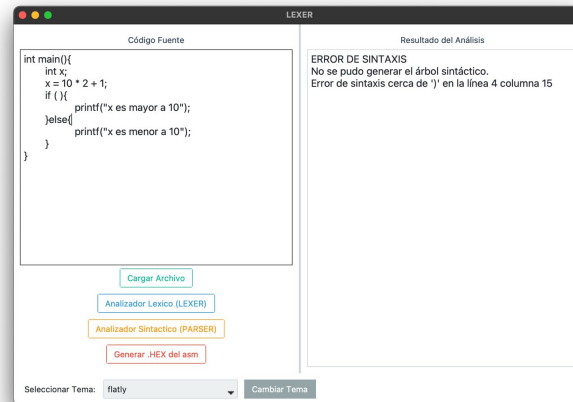


Figure 4: Syntax error caused by an empty if statement: the condition exists, but the body of the if is missing, resulting in incomplete syntax.

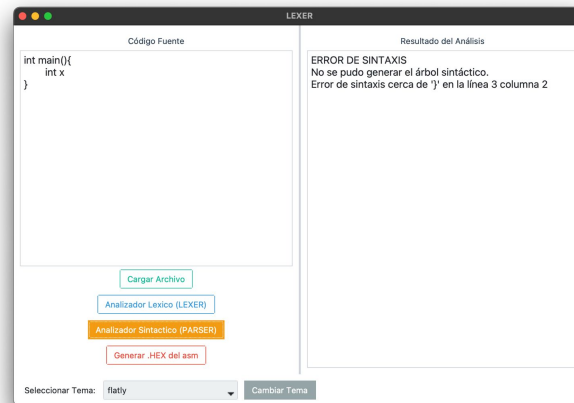


Figure 5: Syntax error due to a missing semicolon (;) at the end of a statement, which is required to properly terminate instructions.

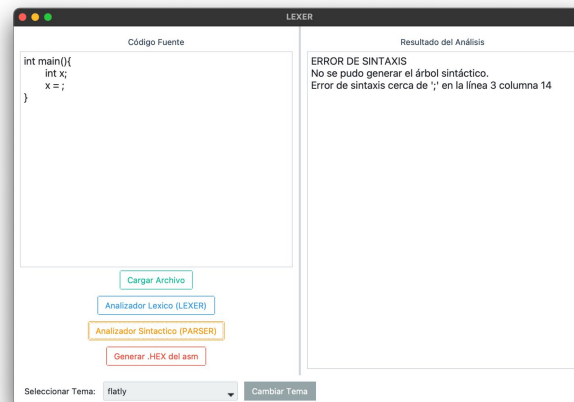


Figure 6: Semantic error where variable x is declared but not assigned a value, leading to undefined behavior or errors during execution.

After constructing the AST, the compiler generates the assembly code by traversing the tree. Each node is translated into one or more assembly instructions corresponding to the operation it represents. This process involves evaluating expressions, managing registers, and handling control flow to produce an executable sequence of instructions.

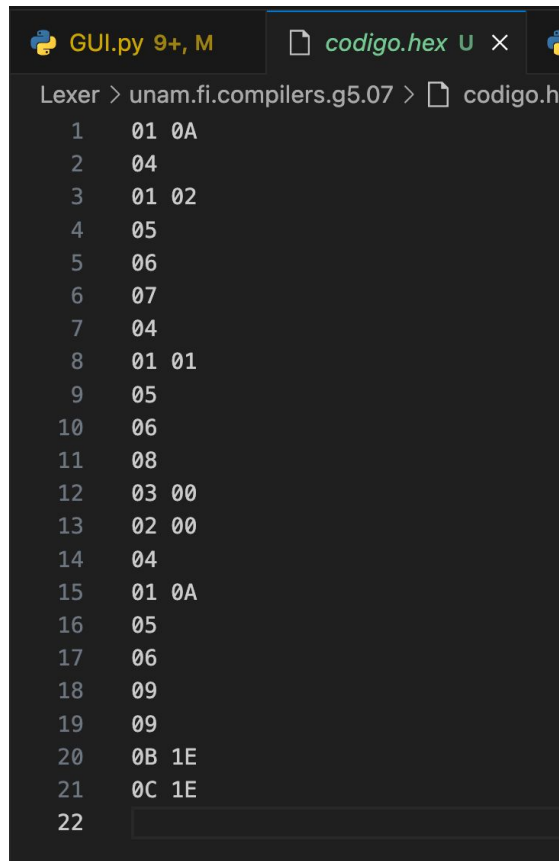
```

Lexer > unam.fi.compilers.g5.07 > codigo.asm
1  x: DEFB 0
2  LD A, 10
3  PUSH AF
4  LD A, 2
5  POP BC
6  LD A, B
7  CALL MUL
8  PUSH AF
9  LD A, 1
10 POP BC
11 LD A, B
12 ADD
13 LD (x), A
14 LD A, (x)
15 PUSH AF
16 LD A, 10
17 POP BC
18 LD A, B
19 CP
20 CP 10
21 JP LE, ELSE_0
22 ; PRINT "x es mayor a 10"
23 JP ENDIF_0
24 ELSE_0:
25 ; PRINT "x es menor a 10"
26 ENDIF_0:

```

Figure 7: Assembly code generation.

Finally, the generated assembly code is translated into machine code represented in hexadecimal format. This hexadecimal code is what the processor directly executes. Each assembly instruction corresponds to one or more machine code bytes, which are commonly expressed as hexadecimal values for readability and compactness.



The image shows a code editor window with two tabs: 'GUI.py 9+, M' and 'codigo.hex U'. The active tab is 'codigo.hex'. The editor content shows a list of 22 lines of machine code in hexadecimal, with line numbers on the left. The code is as follows:

Line	Machine Code
1	01 0A
2	04
3	01 02
4	05
5	06
6	07
7	04
8	01 01
9	05
10	06
11	08
12	03 00
13	02 00
14	04
15	01 0A
16	05
17	06
18	09
19	09
20	0B 1E
21	0C 1E
22	

Figure 8: Representation of the machine code generated from the assembly instructions.

5 Conclusions

In this project, we successfully built a parser that creates abstract syntax trees, which are a key part of understanding and analyzing the meaning of a program's code. By using a suitable programming language and parsing tools, the parser breaks down the source code into a clear structure that makes it easier to check for errors and prepare the code for later steps in the compiler.

This project meets its goal of designing and implementing a system that can generate these trees in a way that supports further processing. The work shows how important the parsing stage is for turning written code into something the computer can actually execute, laying the foundation for building a complete and functioning compiler.

References

- [1] “Lexical Analysis With Flex, for Flex 2.6.4”. flex. Accedido el 28 de febrero de 2025. [En línea]. Disponible: <https://docs.jade.fyi/gnu/flex.html>.
- [2] Nystrom, R. (2021). Crafting interpreters.
- [3] Siek, J. G. (2023). Essentials of Compilation: An Incremental Approach in Racket. MIT Press.