Universidad Nacional Autónoma de México

Ingeniería en Computación

Compiladores

Parser & SDT

ALUMNO:
Hernández Diaz Sebastian

Grupo:
#5
Semestre:
2025-I

México,CDMX. Noviembre 2024

Index.

1. Introduction

1.1 Problem statement

A compiler is a program that converts source code, usually written at a high level, from a programming language into machine code. Lexical analysis, syntax analysis, semantic analysis, code optimization, and output code generation are the usual tasks they do. [1] Each of these steps must be finished for the compiler's design to be constructed correctly. However, the construction of a syntax and semantic analyzer that corresponds to the subsequent compilation phases will be the unique focus of this project.
The source code was converted into a series of tokens of distinct types, including keywords, identifiers, operators, and literals, by the compiler's lexical analyzer in the previous step. [2]

In order to confirm that the source code's structure complies with the grammar rules of the computer language being used, the syntax analyzer examines this stream of tokens during this step. The code's hierarchical structure and the connections between its elements are represented by the syntax tree structure that the parser creates. [3][4] Semantic analysis, which follows type compatibility and variable initialization, makes sure the code makes sense within the language's rules once the syntax of the code has been confirmed. The process entails building a symbol table, a data structure that records the kinds, scope, and other characteristics of identifiers, including variables, functions, and classes. [5][6]

1.2 Motivation

To make sure that code is logically and appropriately organized in accordance with the language's principles, syntax and semantic analyzers must be implemented. Building a system that can precisely analyze and interpret source code and identify mistakes early in the compilation process is the driving force behind the development of these technologies. While a powerful semantic analyzer makes sure that the code complies with logical standards, such as type compatibility, scope limitations, and the usage of function arguments, a decent syntax analyzer aids in preventing basic structure problems. When combined, these analyzers increase code reliability, lower runtime errors, and facilitate and expedite the software development process.

1.3 Objectives

- Understand the role of syntax and semantic analysis in the compilation process.
- Develop a syntax analyzer that can accurately parse and detect structural errors.
- Implement a semantic analyzer that verifies logical rules within the code.
- Demonstrate the importance of early error detection through syntax and semantic analysis.

2. Framework

2.1 Parsing: Bottom-Up

The process o organizing a list of tokens (derived from lexical analysis) into a tree structure that depicts the syntax of the source code is known as parsing.

- Bottom-up Beginning with the leaves (tokens), parsing progresses to the parse tree's root.
- Tokens and smaller constructs are reduced into larger constructs using this method until we reach the grammar's initial symbol.

- Shift-reduce parsing is a method used in bottom-up parsing. It functions using two primary actions:
  The next input token is moved onto a stack using the shift function.
- Reduce: Uses a production rule to combine items on the stack into a higher-level construct.

Bottom-Up Parsing: Shift-Reduce Mechanism

The shift-reduce method is crucial to bottom-up parsing. This procedure entails:

- **Shifting** tokens one-by-one onto a stack until a rule can be applied.
- **Reducing** means recognizing when the tokens on the stack match a grammar rule’s

right side, and then replacing these tokens with the rule's left side.

Because it reduces backtracking, the shift-reduce approach is quite effective. Bottom-up parsing is appropriate for complicated languages since it occurs in real-time and reads tokens only once. Furthermore, ambiguities can be resolved by using conflict resolution strategies like "shift/reduce" and "reduce/reduce."

2.2 LALR(1) Parsing.

Characteristics:

1. Look-ahead: The (1) in LALR(1) means the parser uses one token of look-ahead to decide the next action.
2. Efficient Parsing Tables: LALR(1) reduces the size of the parsing tables by combining similar states, making it more memory-efficient.
3. Suitable for Real-world Languages: Many modern programming languages can be parsed using LALR(1) parsers, which makes it a popular choice.

Parsing: Combining States for Efficiency

The LALR(1) parsing method is a specific refinement of LR parsing that merges similar states, reducing memory usage without losing accuracy. In LALR parsing:

- **Lookahead (1)**: Each parsing decision is based on the next immediate token. This "1-token lookahead" optimizes performance and keeps parsing tables concise.
- **Merging states**: Unlike SLR (Simple LR), LALR(1) groups states with similar transitions, allowing large grammars to be parsed in a smaller memory footprint.

Because it strikes a balance between speed and versatility, LALR(1) parsing is often used for parsing programming languages with complex but deterministic grammars, such as Java and C.

The LALR(1) parsing feature in Python is supported by PLY (Python Lex-Yacc), which will facilitate the effective implementation of this parser.

2.3 Syntax-Directed Translation (SDT)

Using grammatical rules to produce intermediate code or carry out other parsing operations is known as syntax-directed translation, or SDT.We use Embedded SDT:

In an embedded SDT, semantic actions are embedded within the grammar rules of the parser. These actions can:

- Check types.
- Store and look up identifiers in a symbol table.

Every parser rule can have a corresponding action to carry out operations like checking for type compatibility or adding symbols to a table.

Steps for SDT with Type Validation and Symbol Table

1. **Define a Symbol Table**: This will store information like variable names, types, scopes, and other attributes. The symbol table can be a dictionary in Python, with keys as variable names and values as attributes like type and scope.
2. **Type Checking**: Within each rule, use embedded actions to verify types. For example, ensure both sides of an assignment are compatible (e.g., cannot assign a string to an integer variable).
3. **Semantic Actions**: Define functions within your parser rules that:
    - Add symbols to the table (like variable declarations).
    - Check if symbols are already defined (to prevent duplicate definitions).
    - Verify types during expressions (like ensuring `int` is not added to `string`).
4. **Error Handling**: Embedded SDT also allows for **error reporting** if something unexpected is found. For instance, if a variable is used before it's declared, the SDT action can print a clear error.

The parser is in charge of both syntax and semantic correctness in embedded SDT since semantic operations like code generation stages, error checks, or type validations are encapsulated within grammar rules.Why we use Embedded SDT?

- **Single Pass**: SDT embedded in parsing allows the compiler to conduct syntax and semantic checks in a single traversal of the source code.
- **Immediate Feedback**: Errors are detected at the moment the relevant rule is applied,

simplifying debugging.

- **Intermediate Code Generation**: SDT can also be used for producing intermediate representations or machine code directly during parsing.

SDT and parsing work closely together to offer flexibility and power, particularly in compilers where syntax and semantics frequently overlap.

In SDT, type checking is essential since it confirms that variables and expressions are used with data types that are compatible. The following are important type-checking factors:

- **Assignment Type Consistency**: Ensuring that a variable only stores data of its declared type.
- **Operation Validity**: Operators (like + or *) must work with compatible types.
- **Function Parameters and Returns**: Parameters passed to functions and their return values should match declared types.

The parser's structured data repository for program identification details is called a symbol table. It helps with type validation and guarantees that variables are declared before being used.

Attributes Stored:

    - **Name**: The identifier’s name.
    - **Data Type**: Type information, such as int, float, or string.
    - **Scope**: Defines whether the identifier is in global or local scope, helping manage variable visibility.
    - **Location/Memory Address**: If targeting machine code, the address can be included for code generation.
    - **Other Attributes**: Like dimensions of an array or access permissions.
- **Use in Semantic Analysis**:
    - When a variable or function is used, the symbol table is consulted to confirm its declaration and check its type.
    - During assignments and operations, it allows type consistency checks by comparing the types of involved variables.

In order to make parsing reliable and easy to use, error management is essential. It offers helpful feedback and keeps the parser from terminating suddenly on the first error.

Common Error Types:

- **Syntax Errors**: Incorrect structures that don't match any grammar rule (like missing parentheses).
- **Type Errors**: Operations or assignments that use incompatible data types.
- **Scope Errors**: Accessing variables outside their visibility, such as using a local variable outside its function.
- **Undeclared Variable Errors**: Using variables that haven't been declared or initialized.

- **Recovery Strategies**:
  - **Panic Mode**: The parser skips tokens until it finds a suitable point to resume parsing, preventing one error from causing cascading issues.
  - **Error Productions**: Special grammar rules are included to handle specific common errors.
  - **Error Messages**: Providing detailed messages that help users understand and locate the problem in their code.

3. Body

The PLY library, created and utilized for the development of Lex and Yacc (lexical analyzer and syntax analyzer), was used to implement the syntax and semantic analyzer in Python. It permits the development of both elements while following their individual regulations, but with several exceptions that facilitate their correct application.

The lexical analyzer that was previously shown was modified. Among these is the inclusion of reserved words, which the syntax analyzer will employ. Additionally, token categories were introduced, each with a unique operator. Furthermore, a function for identifying decimal points was developed, and a unique function was included to read a text file that contained the code that needed to be examined.

The YACC library and the tokens specified in the lexical analyzer are imported for the syntax analyzer. After declaring these components, we generate a symbol table called "symbol_table" in which we store any input data that corresponds to the specified grammar. The data is not recorded in the table if it does not follow the grammatical rules. Additionally defined is the

parser's initial rule.

These are defined as functions in the production rules declaration, performing necessary checks during syntax analysis. Verifying type consistency in assignments and expressions for instance, making sure strings aren't allocated to integer variables is made possible via these actions. Functions were also included to help manage and track the use of each variable or function by storing identifiers in the symbol table upon declaration.

If the type matches, the variable is added to the symbol table, and a success message is displayed: “Parsing Success! SDT verified”. If there is a semantic error, a message shows parsing success but an SDT error: “Parsing Success! SDT error!”.

In order to minimize cascading mistakes, error handling mechanisms were introduced for both syntax and semantics. When syntax faults are detected, the parser employs a recovery strategy known as "panic mode," which permits it to ignore specific tokens until it reaches a restart point. Furthermore, an auxiliary function was written to verify the type of each variable and its value. It verifies that the value is either true or false if the type is "bool," a floating-point number if it is "float," or an integer if it is "int."

The Symbol Table implementation uses a Python dictionary structure, with each key denoting an identifier (like the name of a variable or function) and its value being a collection of related information. These characteristics include scope (global or local), data type (such as int, float, or string), and additional context-specific information like array sizes or memory addresses if code generation is taken into account. The symbol table helps with type validations in complex expressions in addition to enabling confirmation that variables are declared before use.

An effective parsing technique called LALR(1), which combines comparable states to minimize memory usage without compromising accuracy, is used by the syntax analyzer built with PLY. Using the shift-reduce technique, this bottom-up method constructs the parse tree from the tokens all the way up to the root, where:

- ➔ Shift: The current input token is pushed onto the stack.
- ➔ Reduce: A set of tokens on the stack is replaced by a higher-level symbol when it matches the right side of a grammar rule.

The parser can make accurate decisions by using LALR(1) with a one-token lookahead, which maximizes the creation of parsing tables without needlessly expanding their size. This is

particularly helpful in languages with intricate grammars.

The following is the LALR(1) parser automaton that is produced in a.out file following the execution of PLY.YACC:

LALR(1)

state 0

(0) S' -> . statement

(1) statement -> . type ID EQUALS value

(2) type -> . INT

(3) type -> . FLOAT

(4) type -> . BOOL

INT shift and go to state 3

FLOAT shift and go to state 4

BOOL shift and go to state 5

statement shift and go to state 1

type shift and go to state 2

state 1

(0) S' -> statement .

state 2

(1) statement -> type . ID EQUALS value

ID shift and go to state 6

state 3

(2) type -> INT .

ID reduce using rule 2 (type -> INT .)

state 4

(3) type -> FLOAT .

ID reduce using rule 3 (type -> FLOAT .)

state 5

(4) type -> BOOL .

ID reduce using rule 4 (type -> BOOL .)

state 6

(1) statement -> type ID . EQUALS value

EQUALS shift and go to state 7

state 7

(1) statement -> type ID EQUALS . value

(5) value -> . NUMBER

(6) value -> . SPOT

(7) value -> . BOOL_TRUE

(8) value -> . BOOL_FALSE

NUMBER shift and go to state 9

SPOT shift and go to state 10

BOOL_TRUE shift and go to state 11

BOOL_FALSE shift and go to state 12

value shift and go to state 8

state 8

(1) statement -> type ID EQUALS value .

$end reduce using rule 1 (statement -> type ID EQUALS value .)

state 9

(5) value -> NUMBER .

$end reduce using rule 5 (value -> NUMBER .)

state 10

(6) value -> SPOT .

$end reduce using rule 6 (value -> SPOT .)

state 11

(7) value -> BOOL_TRUE .

$end reduce using rule 7 (value -> BOOL_TRUE .)

state 12

(8) value -> BOOL_FALSE .

$end reduce using rule 8 (value -> BOOL_FALSE .)

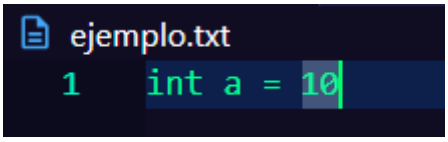
4. Results

Only the data types int, float, and bool are supported by the analyzer that was developed. Therefore, in order to confirm that the syntax and semantic analyzer function as intended, we shall illustrate every scenario that could arise. Likewise, a.txt file is used to read all inputs.

- TEST 1.

```
# Load input file
with open("ejemplo.txt", "r") as file:
    input_string = file.read()  # Read the entire contents of the file
```

Screenshot Test 1 - Loading and reading the contents of the .txt.



Screenshot Test 1.1 - Input “int a = 10 ”.

```
# We pass the example to the parser
result = parser.parse(input_string)
```

Screenshot Test 1.2 - Input to the parser.

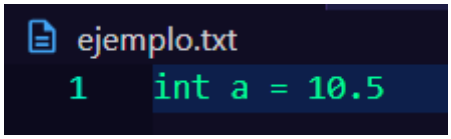
```
LexToken(INT,'int',1,0)
LexToken(ID,'a',1,4)
LexToken(EQUALS,'=',1,6)
LexToken(NUMBER,10,1,8)
Parsing Success! SDT Verified!
```

Screenshot Test 1.3 - Output.

- TEST 2.

```
# Load input file
with open("ejemplo.txt", "r") as file:
    input_string = file.read()  # Read the entire contents of the file
```

Screenshot Test 2 - Loading and reading the contents of the .txt.



Screenshot Test 2.1 - Input “int a = 10.5 ”.

```
# We pass the example to the parser
result = parser.parse(input_string)
```

Screenshot Test 2.2 - Input to the parser.

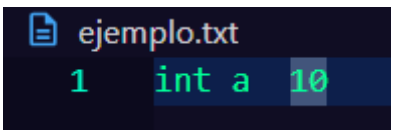
```
LexToken(INT,'int',1,0)
LexToken(ID,'a',1,4)
LexToken(EQUALS,'=',1,6)
LexToken(SPOT,10.5,1,8)
Parsing Success! SDT error!
```

Screenshot Test 2.3 - Output.

- TEST 3.

```
# Load input file
with open("ejemplo.txt", "r") as file:
    input_string = file.read()  # Read the entire contents of the file
```

Screenshot Test 3 - Loading and reading the contents of the .txt.



Screenshot Test 3.1 - Input “int a 10 ”.

```
# We pass the example to the parser
result = parser.parse(input_string)
```

Screenshot Test 3.2 - Input to the parser.

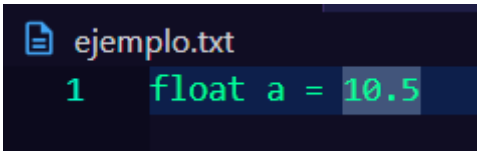
```
LexToken(INT,'int',1,0)
LexToken(ID,'a',1,4)
LexToken(DOUBLE_LITERAL,10.5,1,7)
Parsing error! SDT error!
```

Screenshot Test 3.3 - Output.

- TEST 4.

```
# Load input file
with open("ejemplo.txt", "r") as file:
    input_string = file.read()  # Read the entire contents of the file
```

Screenshot Test 4 - Loading and reading the contents of the .txt.



Screenshot Test 4.1 - Input “float a = 10.5 ”.

```
# We pass the example to the parser
result = parser.parse(input_string)
```

Screenshot Test 4.2 - Input to the parser.

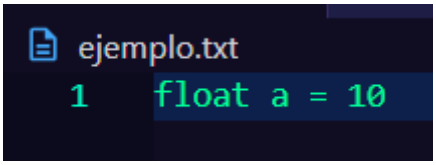
```
LexToken(FLOAT,'float',1,0)
LexToken(ID,'a',1,6)
LexToken(EQUALS,'=',1,8)
LexToken(SPOT,10.5,1,10)
Parsing Success! SDT Verified!
```

Screenshot Test 4.3 - Output.

- TEST 5.

```
# Load input file
with open("ejemplo.txt", "r") as file:
    input_string = file.read()  # Read the entire contents of the file
```

Screenshot Test 5 - Loading and reading the contents of the .txt.



Screenshot Test 5.1 - Input “float a = 10 ”.

```
# We pass the example to the parser
result = parser.parse(input_string)
```

Screenshot Test 5.2 - Input to the parser.

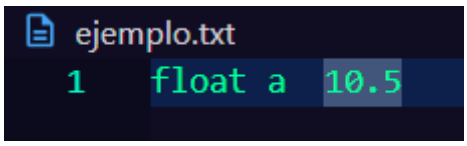
```
LexToken(FLOAT,'float',1,0)
LexToken(ID,'a',1,6)
LexToken(EQUALS,'=',1,8)
LexToken(NUMBER,10,1,10)
Parsing Success! SDT error!
```

Screenshot Test 5.3 - Output.

- TEST 6.

```
# Load input file
with open("ejemplo.txt", "r") as file:
    input_string = file.read()  # Read the entire contents of the file
```

Screenshot Test 6 - Loading and reading the contents of the .txt.



Screenshot Test 6.1 - Input “float a 10.5 ”.

```
# We pass the example to the parser
result = parser.parse(input_string)
```

Screenshot Test 6.2 - Input to the parser.

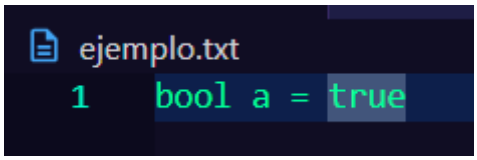
```
LexToken(FLOAT,'float',1,0)
LexToken(ID,'a',1,6)
LexToken(SPOT,10.5,1,9)
Parsing error! SDT error!
```

Screenshot Test 6.3 - Output.

- TEST 7

```
# Load input file
with open("ejemplo.txt", "r") as file:
    input_string = file.read()  # Read the entire contents of the file
```

Screenshot Test 7 - Loading and reading the contents of the .txt.



Screenshot Test 7.1 - Input “bool a = true ”.

In this case, whether true or false is provided, the result indicates that it executed correctly.

```
# We pass the example to the parser
result = parser.parse(input_string)
```

Screenshot Test 7.2 - Input to the parser.

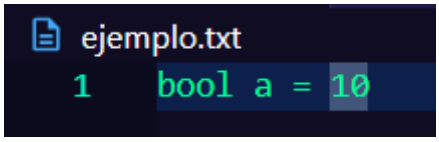
```
LexToken(BOOL,'bool',1,0)
LexToken(ID,'a',1,5)
LexToken(EQUALS,'=',1,7)
LexToken(BOOL_TRUE,'true',1,9)
Parsing Success! SDT Verified!
```

Screenshot Test 7.3 - Output.

- TEST 8

```
# Load input file
with open("ejemplo.txt", "r") as file:
    input_string = file.read()  # Read the entire contents of the file
```

Screenshot Test 8 - Loading and reading the contents of the .txt.



Screenshot Test 8.1 - Input “bool a = 10 ”.

```
# We pass the example to the parser
result = parser.parse(input_string)
```

Screenshot Test 8.2 - Input to the parser.

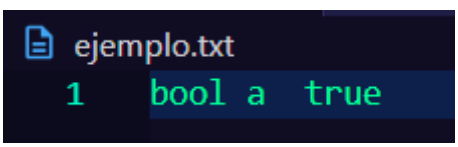
```
LexToken(BOOL,'bool',1,0)
LexToken(ID,'a',1,5)
LexToken(EQUALS,'=',1,7)
LexToken(NUMBER,10,1,9)
Parsing Success! SDT error!
```

Screenshot Test 8.3 - Output.

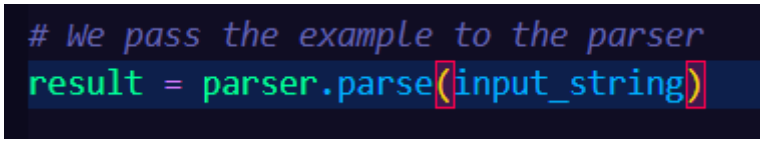
- TEST 9

```
# Load input file
with open("ejemplo.txt", "r") as file:
    input_string = file.read()  # Read the entire contents of the file
```

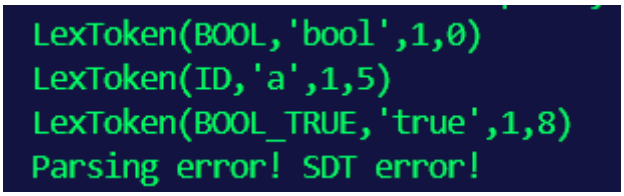
Screenshot Test 9 - Loading and reading the contents of the .txt.



Screenshot Test 9.1 - Input “bool a true; ”.



Screenshot Test 9.2 - Input to the parser.



Screenshot Test 9.3 - Output.

5. Conclusion

The report describes the development of a syntax and semantic analyzer as essential components of a compiler. By building a parse tree from tokens produced by an earlier lexical analysis stage, the syntax analyzer uses a bottom-up LALR(1) parsing method via Python's PLY module. The syntax analyzer verifies that the code complies with the necessary grammar by comparing each token to the programming language's rules and pointing out structural mistakes like omitted operators or improper expressions.

The project integrates syntactic-directed translation (SDT), which embeds semantic checks directly into grammar rules, in addition to syntax analysis. As the code is parsed, this embedded technique does the verification of variable initialization, scope management, and type consistency. For example, SDT allows the analyzer to verify that variables are declared before usage and that assigned values correspond to variable types. In order to assist these tests and guarantee that scope and declaration restrictions are followed, the analyzer also creates a symbol table as a central structure for holding information about identifiers.

In conclusion, this project does a good job of demonstrating how syntax and semantic analyzers function during the compilation process. By putting both analyzers into practice, the project creates a system that guarantees logical correctness and structural accuracy, thereby avoiding typical programming errors early on. By lowering the possibility of runtime errors and promoting a more efficient software development process. Combining syntax with semantic analysis is crucial for developing effective, error resistant compilers, as demonstrated by the combination of syntax directed translation and error management.

6. Bibliography

[1] R. Sheldon, “compiler,” WhatIs, 2022. https://www.techtarget.com/whatis/definition/compiler (accessed Nov. 08, 2024).

[2] R. Dávila , “Compilers. 1st Midterm”, class notes for 0434, División de Ingeniería Eléctrica, Facultad de Ingeniería, Universidad Nacional Autónoma de México, Autumn 2024.

[3] GeeksforGeeks, “Introduction to Syntax Analysis in Compiler Design,” GeeksforGeeks, Sep. 22, 2015. https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/ (accessed Nov. 08, 2024).

[4] R. Dávila , “Parsing Compilers”, class notes for 0434, División de Ingeniería Eléctrica, Facultad de Ingeniería, Universidad Nacional Autónoma de México, Autumn 2024.

[5] Cratecode, “Semantic Analysis in Compiler Design,” Cratecode, Apr. 26, 2023. https://cratecode.com/info/semantic-analysis (accessed Nov. 08, 2024).

[6] R. Dávila , “Syntax Directed Translation Schemes (SDTS). Compilers”, class notes for 0434, División de Ingeniería Eléctrica, Facultad de Ingeniería, Universidad Nacional Autónoma de México, Autumn 2024.

[7] TutorialsPoint. (s.f.). *Compiler Design*. TutorialsPoint. Recuperado el 10 de noviembre de 2024, de https://www.tutorialspoint.com/compiler_design/index.htm

[8] Beazley, D. (s.f.). *PLY (Python Lex-Yacc)*. PLY Documentation. Recuperado el 10 de noviembre de 2024, de http://www.dabeaz.com/ply/