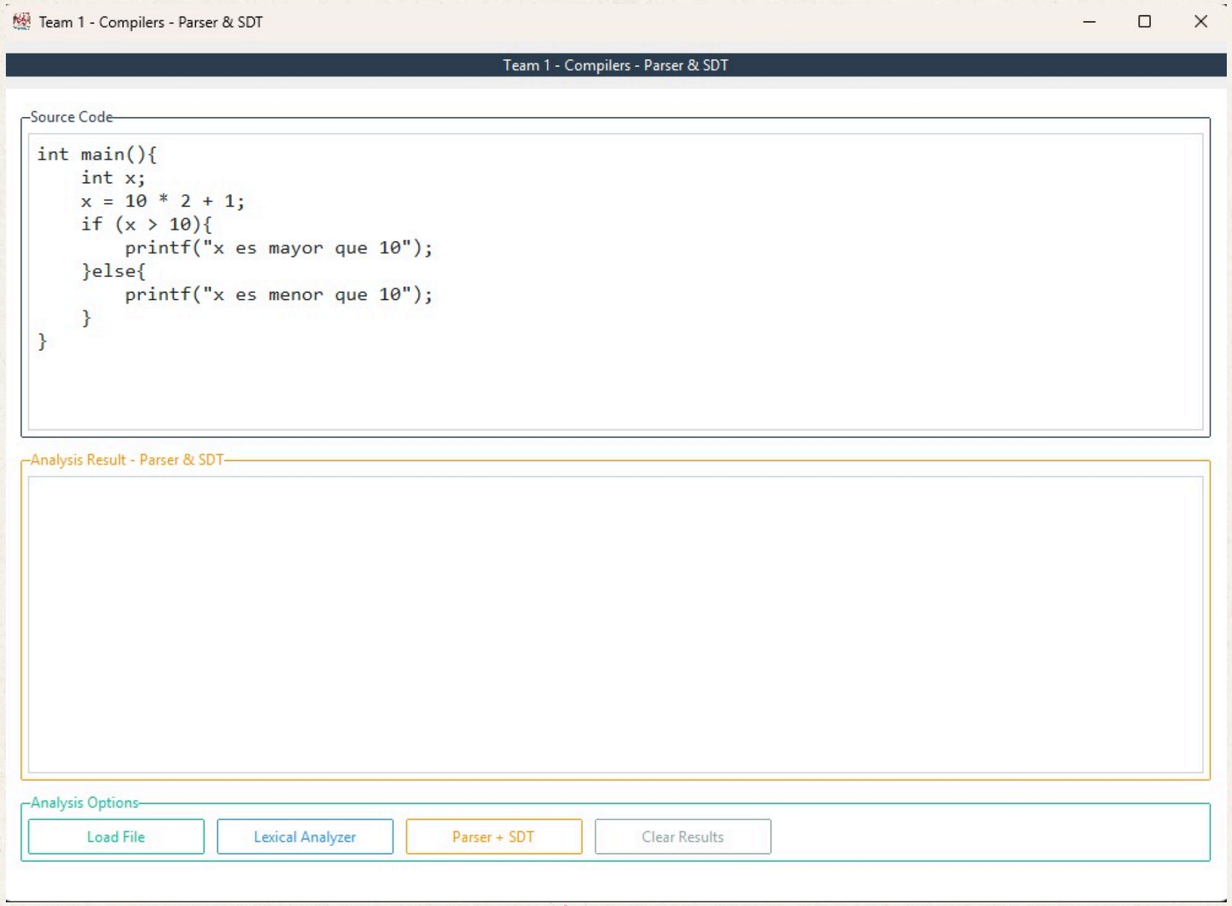
Universidad Nacional Autónoma de México
Facultad de Ingeniería



PARSER & SDT

COMPILERS (0434)
Group: 5
Professor: RENE ADRIAN DAVILA PEREZ

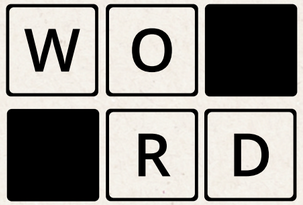
MEMBERS:

- 320075669
- 423066940
- 423035975
- 320199866
- 320061682

OBJECTIVES

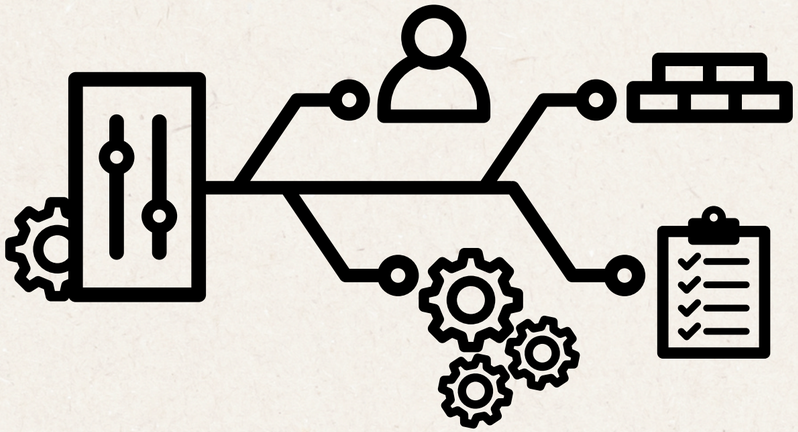
THE GRAMMAR

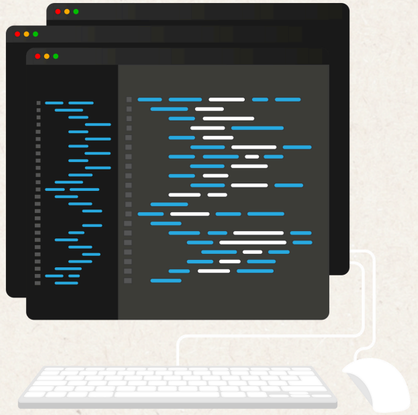
Define a formal grammar whose productions match the input tokens



PROCEDURE

Implement a method and procedure to read the input tokens, construct the parse tree and verify all the SDT rules.





RESULTS

Show a clear and distinguishable output for all possible scenarios.

GRAMMAR

CONSTRUCTION OF OUR GRAMMAR:

Our grammar is based in a basic C structure where you can write a simple C program

FORM OF THE GRAMMAR:

- `program`: The start symbol. It must be a `main` function, defined as:

  ```
  'program : INT ID LPAREN RPAREN LBRACE statements RBRACE'
  ```

  The action for this rule semantically checks if the `ID` (p[2]) is "main".

- `statements`: A list of one or more `statement` non-terminals.
- `statement`: A `statement` can be one of the following: `declaration`, `assignment`, `if_statement`, `printf_statement`, or `return_statement`.
- `declaration`: A type followed by an ID and a semicolon:

  ```
  'declaration : tipo ID SEMICOLON'
  ```

- `assignment`: An ID, an equals sign, an expression, and a semicolon:

  ```
  'assignment : ID IGUALS expression SEMICOLON'
  ```

- `if_statement`: The grammar supports both `if` and `if-else` structures:

  ```
  'if_statement : IF LPAREN expression RPAREN LBRACE statements RBRACE
                | IF LPAREN expression RPAREN LBRACE statements RBRACE
                ELSE LBRACE statements RBRACE'
  ```

- `expression`: Defined recursively to handle binary operations respecting precedence, parentheses, numbers, and variable IDs.

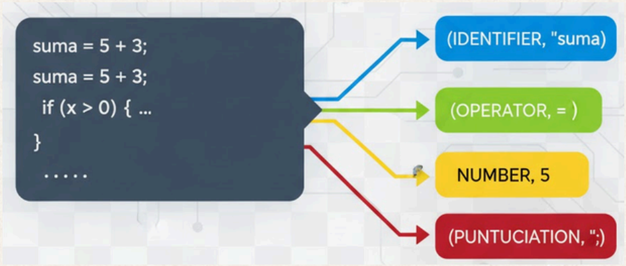
CODE VERIFY

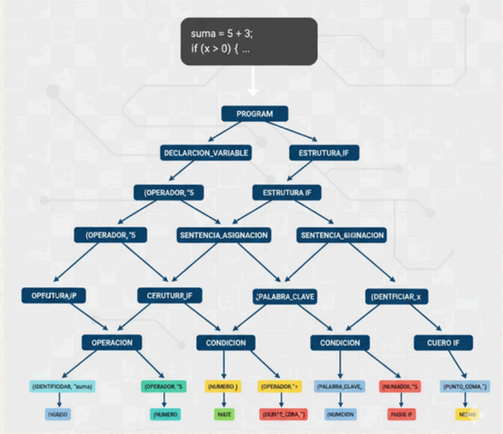
PROYECT PURPOSE:
We developed a code analyzer in python language that allows us detect lexic, semantic and syntactic erros before compilation.

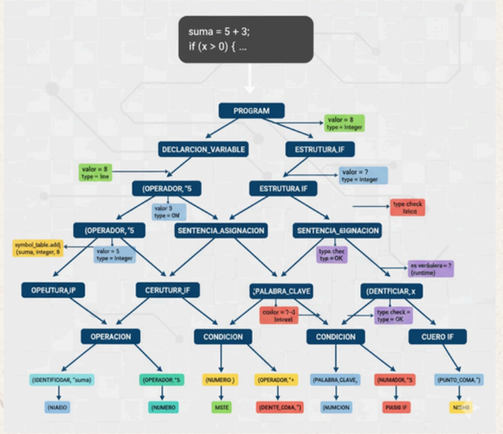
CHARACTERISTICS:

- Key characteristics:
- Analize source code and generate its syntax tree
- Validate in real time the language rules.
- Provide clear and detailed messages of errors.

¿WHAT IS A LEXICAL ANALYZER, PARSER AND SDT?







LEXICAL ANALYZER (LEXER)
Breaks the program text into tokens, like keywords, identifiers, or numbers. It's like splitting a book into words.

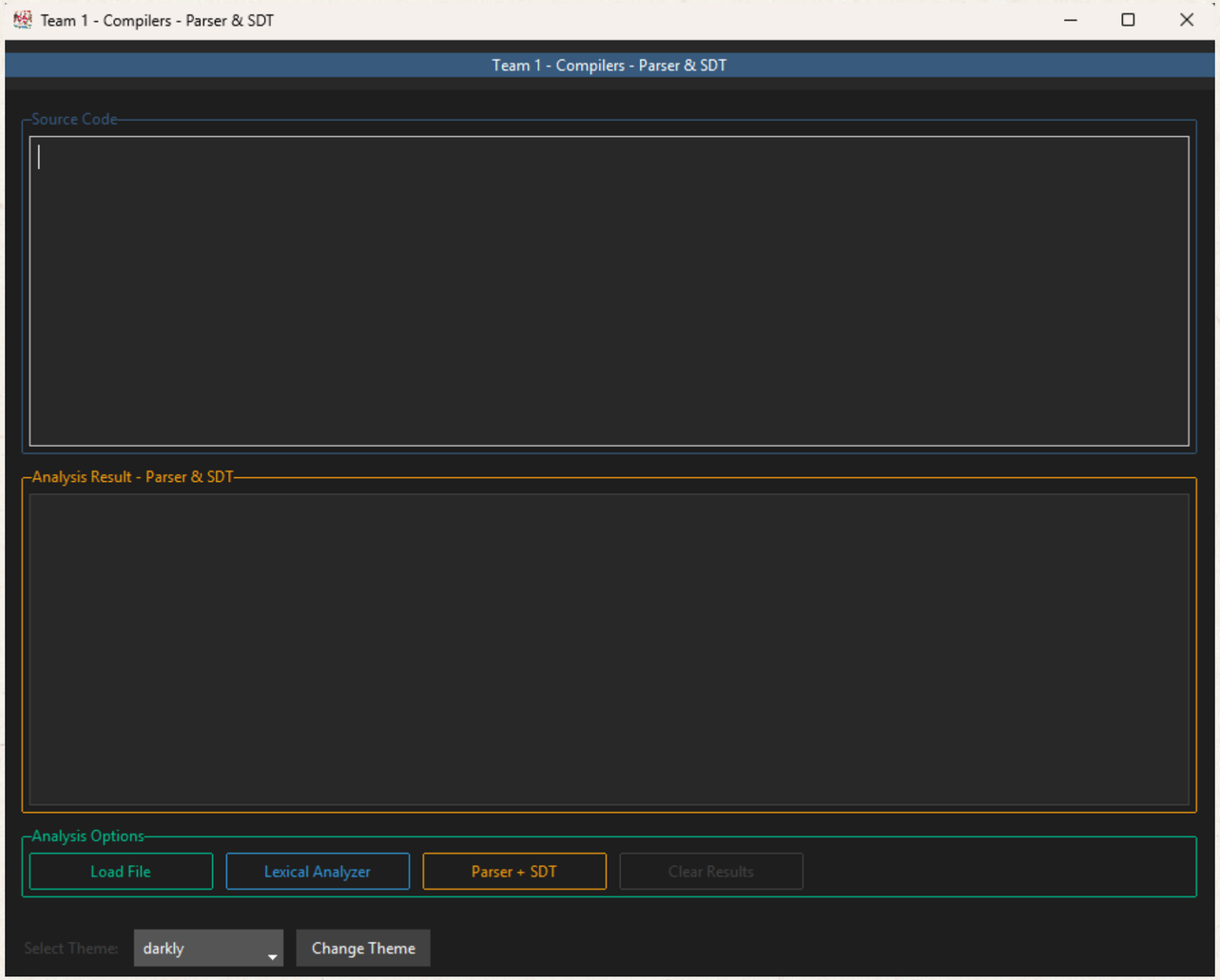
PARSER (SYNTAX ANALYZER):
Takes those tokens and checks if they follow the language rules, creating a tree structure that represents the program.

SDT (SYNTAX-DIRECTED TRANSLATION):
SDT converts the syntax into meaning while the parser goes through the program.

WORKING PRINCIPLE

Source code ➔ Lexical Analyzer ➔ Parser ➔ SDT ➔ Clear results

- **Load file:** Allows to write in the interface, although “.txt” files can be uploaded.
- **Lexical analyzer:** identifies the code componentes.
- **Parser:** constructs the synctactic structure and generates the synctatctic tree.
- **SDT:** verifies the code meaning.
- **Clear results:** Allows to clean the results to test some others.



¿HOW DOES THE PARSER WORKS?

STEP 1: LEXICAL ANALYZER

Our program reads the source code and turns it into little pieces called **“tokens”** (key words, numbers, identifiers, punctuators).

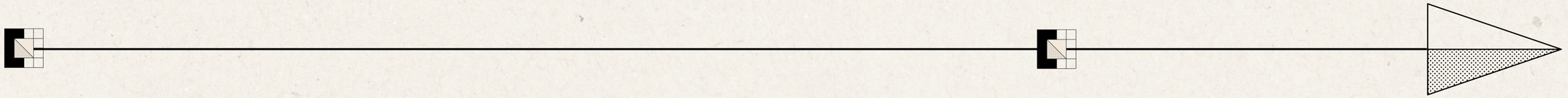
STEP 2: PARSER

- Then, tokens are processed by the parser, that verifies if they follow the language rules and then create a tree which represents the program **(parse tree)**.
- In our program, we used **PLY** (Python Lex-Yacc) for consctucting this parser.

STEP 3: SDT

- While the parser analyzes, the sistem also applies semantic rules: verifies the variables are declared before using them, the types are compatibles, and operations make sense.
- In others words, **it converts the syntasis in meaning and real results**.

¿HOW DOES THE PARSER WORKS?



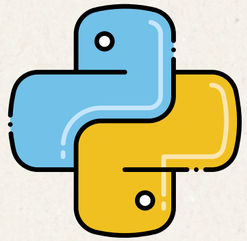
STEP 4: ERROR DETECTION

- If any lexic, syntactic or semantic error presents, the program reports it clearly.
- This makes the code more reliable and safer before executing or generating results.

STEP 5: RESULTS

- Program let us see the results in the graphic interface, even if there could be an error.

PLY (PYTHON LEX-YACC) LIBRARY



WE USED PLY (PYTHON LEX-YACC), WHICH REPRODUCES THE COMPILERS LEX/YACC TOOL IN PYTHON.
BESIDES WE USED GRAPVIZ TO GENERATE THE SYNCTAIC TREES

- LALR(1) PARSING:
- PLY UTILIZES THE LALR(1) PARSING TECHNIQUE, KNOWN FOR ITS EFFICIENCY AND SUITABILITY FOR A WIDE RANGE OF GRAMMARS.

- LEX.PY (LEXICAL ANALYZER):
- THIS MODULE HANDLES THE PROCESS OF LEXICAL ANALYSIS, WHICH INVOLVES BREAKING DOWN AN INPUT STRING OF CHARACTERS INTO A SEQUENCE OF MEANINGFUL UNITS CALLED TOKENS.

```python
import ply.yacc as yacc
from LEX_C import tokens, analyze_code, get_lexical_errors
import os
try:
    from graphviz import Digraph
    GRAPHVIZ_AVAILABLE = True
except ImportError:
    GRAPHVIZ_AVAILABLE = False
```

- YACC.PY (PARSER GENERATOR):
- THIS MODULE FACILITATES PARSING, WHICH INVOLVES TAKING THE STREAM OF TOKENS GENERATED BY THE LEXER AND ORGANIZING THEM INTO A HIERARCHICAL STRUCTURE (OFTEN AN ABSTRACT SYNTAX TREE OR AST) BASED ON A DEFINED GRAMMAR.

- YACC.PY (PARSER GENERATOR):
- THIS MODULE FACILITATES PARSING, WHICH INVOLVES TAKING THE STREAM OF TOKENS GENERATED BY THE LEXER AND ORGANIZING THEM INTO A HIERARCHICAL STRUCTURE (OFTEN AN ABSTRACT SYNTAX TREE OR AST) BASED ON A DEFINED GRAMMAR.

RESULTS

1

Source Code

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

2

Analysis Result - Parser & SDT

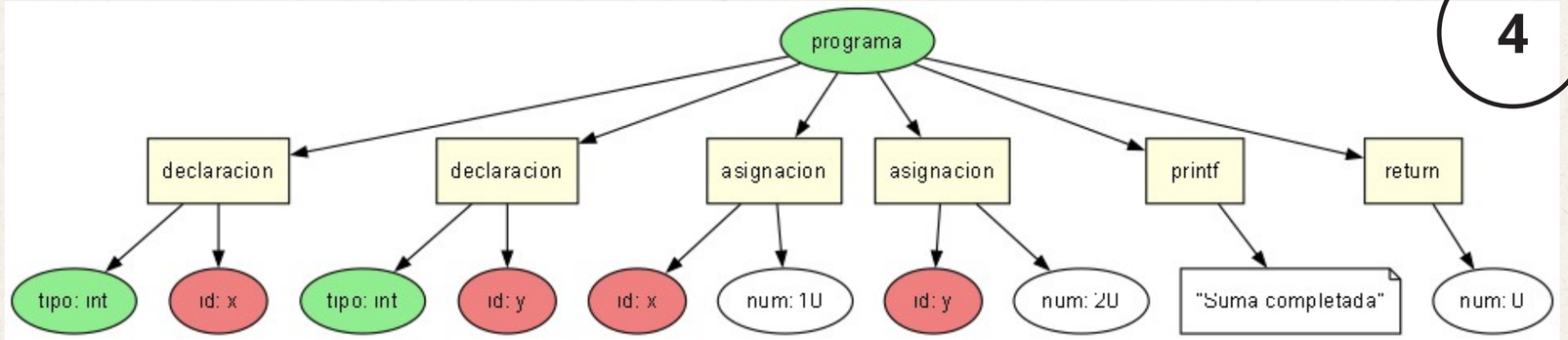
```
=== ANALYSIS PROCEDURE ===

1.  LEXICAL ANALYSIS COMPLETED
   - Tokens recognized: 28
   - Lexical errors: 0

2.  PARSE TREE CONSTRUCTION COMPLETED
=== PARSE TREE ===
program
  declaration
    tipo
      int
    x
  declaration
    tipo
      int
```

3

Analysis Result - Parser & SDT

```
3.  SDT RULES VERIFICATION
✓ Variable declaration verification
✓ Type verification in expressions
✓ Arithmetic operations verification

4.  TREE VISUALIZATION
   Imagen del árbol sintáctico generada: arbol_sintactico.jpg

=== FINAL RESULT ===
 Parsing Success!
 SDT Verified!

The input ended in a syntactically valid state and SDTs were satisfied.

=== SDT INFORMATION ===
Variables in symbol table: 3
  - x: ('tipo', 'int') = 10
  - y: ('tipo', 'int') = 20
```

4



PARSING SUCCESS ✓
SDT VERIFIED ✓

1

Source Code

```
int main() {
    int x;
    int y;
    x = 10;
    y = 20;
    printf("Suma completada");
    return 0;
}
```

2

Analysis Result - Parser & SDT

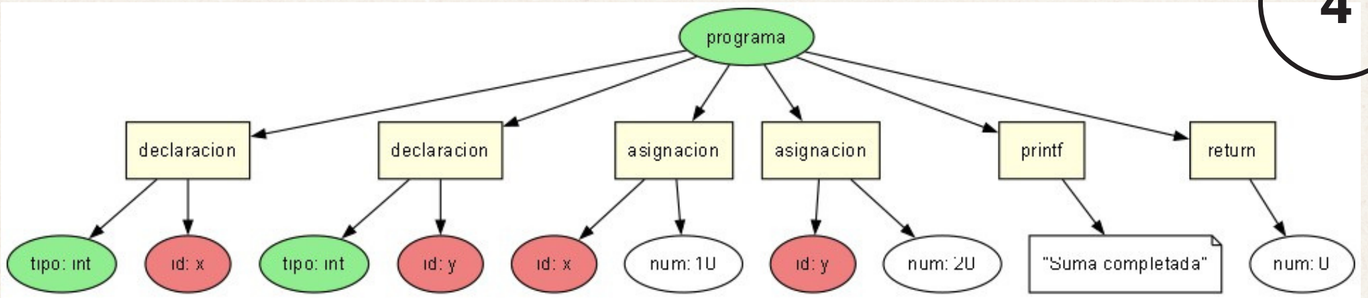
```
=== ANALYSIS PROCEDURE ===

1.  LEXICAL ANALYSIS COMPLETED
   - Tokens recognized: 28
   - Lexical errors: 0

2.  PARSE TREE CONSTRUCTION COMPLETED
=== PARSE TREE ===
program
  declaration
    tipo
      int
```

3

Analysis Result - Parser & SDT

```
4.  TREE VISUALIZATION
   Imagen del árbol sintáctico generada: arbol_sintactico.jpg

=== FINAL RESULT ===
 Parsing Success!
 SDT Verified!

The input ended in a syntactically valid state and SDTs were satisfied.
```

4



PARSING SUCCESS ✓
SDT ERROR ✗

1

Source Code

```
int main() {
    int x;
    int y;
    x = 10;
    z = x + y;
    printf("Resultado");
    return 0;
}
```

2

Analysis Result - Parser & SDT

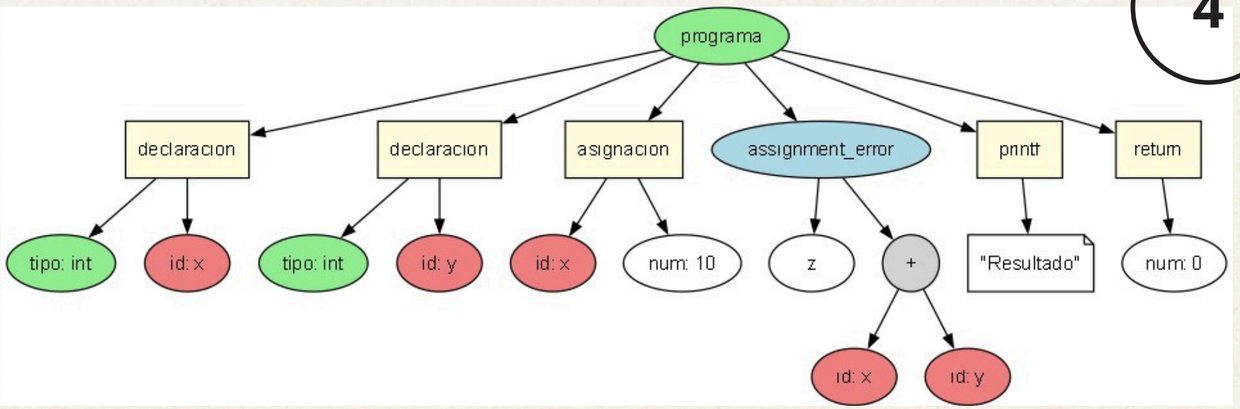
```
=== ANALYSIS PROCEDURE ===

1.  LEXICAL ANALYSIS COMPLETED
   - Tokens recognized: 30
   - Lexical errors: 0

2.  PARSE TREE CONSTRUCTION COMPLETED
=== PARSE TREE ===
```

3

Analysis Result - Parser & SDT

```
✓ Arithmetic operations verification

4.  TREE VISUALIZATION
   Imagen del árbol sintáctico generada: arbol_sintactico.jpg

=== FINAL RESULT ===
 Parsing Success!
 SDT error...
```

4



PARSING ERROR
SDT ERROR

1

Source Code

```
int main( {
    int x;
    x = 5
    printf("Error");
    return 0;
}
```

2

Analysis Result - Parser & SDT

```
=== ANALYSIS PROCEDURE ===

1.  LEXICAL ANALYSIS COMPLETED
   - Tokens recognized: 19
   - Lexical errors: 0

2.  PARSE TREE CONSTRUCTION FAILED

=== FINAL RESULT ===
 Parsing error...
 SDT error...

The input did NOT end in a syntactically valid state.

=== SDT INFORMATION ===
Variables in symbol table: 0
```

THANKS FOR YOUR ATTENTION