Universidad Nacional Autónoma de México

Computer Engineering

Compilers

Parser documentation

TEAM 7:
320030772
320323719
320181520
423007262
424147295

Group:
5
Semester:
2026-I

Mexico City, Mexico. November 2025

Contents

1 Introduction

In software development, we write code using certain programming languages that are translated into machine language. This process is performed by a compiler that processes the source code in 2 phases: The analysis phase and the synthesis phase.[1] In this project we will focus on the analysis phase, which is divided in 3 other phases, the lexical, syntax and semantic phases.[1] So far, we obtained tokens as an output of the lexical phase, so to use them, we will pay special attention to the subsequent phases.

The motivation for this project is to process the tokens provided as output for the lexical analyzer, thus, we need to comprehend the process behind the next phases and implement it into software.

To process the tokens, we need to develop a parser, which is the program used to perform the syntax analysis, where we need to check wether the provided code is properly written by following a specified grammar. After that the parser will organize the tokens using semantic rules into ASTs, which is the output needed for the semantic analyzer.

With all that said, our general objective is to design a parser; however, to be more precise, we need to establish some specific objectives, which are as follows.

- To analyze and comprehend the processes behind the syntax and semantic phases.
- To design and implement a parser that validates code according to the grammar of the C language.
- To integrate SDT rules during parsing for semantic actions.
- To generate the output required as input for the semantic analyzer.

2 Theoretical Framework

2.1 Compiler structure

As said before, if we were to examine the compilation process in more detail, we could see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. [1] A typical decomposition of a compiler into phases is: Lexical analysis, Syntax analysis, Semantic analysis, Intermediate Code Generator, Code optimization and Target Code generation. The first three phases are classified as the analysis phase and the last three are the synthesis phase.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part. [1]

The lexical analyzer classifies the source code into tokens and gives a stream of tokens to the syntax analyzer to validate the structure through a context free grammar.

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a syntax tree, this tree is a representation that illustrates the structure of the grammar used in the compiler.

2.2 Context free grammar

The CFGs (Context free grammars) have a relevant role in the syntax phase. The grammar is the structure that the language follows, in the context of compilers, it is important to understand structured grammars, specifically the Chomsky hierarchy. The Chomsky hierarchy is one of the classifications used in computing theory. Briefly it has four classifications: type 0 is when there are no restrictions in the production rules, type 1 is when the production rules depend of the context of the symbols, type 2 is when the production rules are independent of context and type 3 is when the language uses regular expressions, which is a form to reduce the language into an expression that follows certain patterns.

The syntax analysis requires the use of type 2 grammars, also known as Context Free Grammars. The basic structure of this grammar is:

$$A \to \alpha$$

Where:
$A \in N$
$\alpha \in (N \cup T)^*$
$T - Terminal\ symbols$
$N - Non\ Terminal\ Symbols$
$* - Star\ closure$

Basically, this is the way we represent production rules that can produce any desired grammar. The left side consists of nonterminal symbols that can be replaced by the right side, which is composed of nonterminal symbols, terminal symbols, or a combination of both. [2]

Grammars offer significant benefits for language designers and compiler writers.

- A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language. [1]

- From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program. [1]
- As a side benefit, the parser-construction process can reveal syntactic ambiguities and trouble spots that might have slipped through the initial design phase of a language. [1]
- The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors. [1]
- A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language. [1]

2.3 Context-Free Grammars in Compilers

To implement a Context-Free Grammar (CFG) in a compiler, the produced grammar must be unambiguous. Ambiguity in a CFG occurs when, for a given input string, the grammar can produce more than one possible parse tree. To obtain an unambiguous CFG, each input string must correspond to exactly one parse tree.

Some important elements in the implementation of a CFG within a compiler are the associativity and precedence of operators.

2.4 Associativity

Associativity is related to the recursion in a CFG. When a CFG contains left recursion, that is, when a production rule generates the same non-terminal on the left side of the expression, the grammar is said to have left associativity. Similarly, when a CFG has right recursion, meaning that a production rule generates the same non-terminal on the right side of the expression, it is considered right associative. We usually tried to eliminate the left recursion as the botoom up parsers cannot handle it.[1]

$$E \rightarrow E + T \mid T \qquad \text{(Left recursion)}$$
$$E \rightarrow T + E \mid T \qquad \text{(Right recursion)}$$

2.5 Precedence

Precedence defines the order in which operators are evaluated. In a CFG, precedence can be represented by the depth of the parse tree. An operator that appears deeper in the tree has higher precedence, while an operator closer to the root has lower precedence. [1]

For instance, in arithmetic expressions, multiplication typically has higher precedence than addition. This can be represented in the CFG as follows:

$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (E) \mid id
\end{aligned}
$$

2.5.1 BNF

The BNF (Backus–Naur Form) is a formal notation used to describe the syntax of programming languages and, in general, of Context-Free Grammars (CFG).[3]
Its objective is to represent, in a structured and readable way, the production rules that define how the valid expressions of a language are constructed. The basic notation of a rule in BNF is:

```
<non-terminal> ::= <expression>
```

Where:

- **Non-terminals:** they are written between angle brackets (<>) and represent abstract syntactic categories, such as `<expression>`, `<statement>`, or `<identifier>`.
- **Terminals:** they are the actual symbols of the language, such as keywords, operators, or punctuation marks (e.g., `if`, `+`, `;`).
- **Operator `::=`:** it is read as "is defined as" or "can be replaced by."
- **Alternatives (`|`):** they allow expressing options within the same production rule.

For example:

```
<expression> ::= <expression> "+" <term> | <term>
<term> ::= <number> | <identifier>
```

These rules indicate that an expression (`<expression>`) can be a sum of an expression and a term, or just a single term. They also indicate that a term (`<term>`) can be a number (`<number>`) or an identifier (`<identifier>`).

It serves to:

- Define the grammar of the source language.
- Guide the design of the syntactic analyzer (parser), as it needs to know which combinations of tokens are valid.
- Facilitate communication between those who design the language and those who implement its analysis tools.

2.5.2 Types of parsers

There are some types of parsers but the classification can be ilustrated as:

- Top down parsers: Can be with backtracking (Brute Force) or whitout backtracking (Recursive Descent Parsers, Predictive parsers, LL(1), LL(k))
- Bottom up parsers: Can be operator precedence parsers or LR parsers(LR(0), SLR(0), LALR(1), CLR(1))

The parsers usually have a stack, input buffer and the parsing table.

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder. Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string. [1]

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production. The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds. [1]//

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol. Define FIRST(α),where a is any string of grammar symbols, to be the set of terminals that begin strings derived. Define FOLLOW(A),for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form. [1]

2.6 Semantic Analysis

We associate information with a language construct by attaching attributes to the grammar symbol(s) representing the construct. A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions.

A syntax-directed definition (SDD) is a context-free grammar together with, attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X.

Attributes may be of any kind: numbers, types, table references, or strings, for instance. The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

3 Development

For the construction of the syntactic analyzer (parser), its development was managed through several functional modules, which will collectively work from a sequence of tokens; as seen previously in the lexical analyzer, the plan is to generate a structure of the language's grammar, which will be the basis for this phase of the project.

First, we established the CFG that we will use; in this case, we chose a reduced version of the C language grammar. It is based on our previous lexer, which returns tokens belonging to the C language. Below, we have included the CFG used, written in BNF notation.

3.1 Grammar Definition and Refinement

3.1.1 Initial Grammar Specification

The development of the parser began with a high-level, abstract grammar intended to model a subset of the C language. This initial grammar, provided below, served as the conceptual starting point.

⟨*program*⟩ ::= ⟨*declarations*⟩ ⟨*functions*⟩ ⟨*main*⟩

⟨*declarations*⟩ ::= ⟨*declaration*⟩

⟨*functions*⟩ ::= ⟨*function*⟩

⟨*main*⟩ ::= `int main ( ) {` ⟨*block*⟩ `}`

⟨*declaration*⟩ ::= ⟨*type*⟩ ⟨*list_var*⟩ `;`

⟨*type*⟩ ::= `int` | `float` | `char`

⟨*list_var*⟩ ::= ⟨*var*⟩ `,` ⟨*var*⟩

⟨*var*⟩ ::= ⟨*id*⟩ | ⟨*id*⟩ `[` ⟨*integer*⟩ `]`

⟨*function*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ `(` ⟨*parameters_opt*⟩ `) {` ⟨*block*⟩ `}`
  | `void` ⟨*id*⟩ `(` ⟨*parameters_opt*⟩ `) {` ⟨*block*⟩ `}`

⟨*parameters_opt*⟩ ::= ⟨*parameters*⟩ | ϵ

⟨*parameters*⟩ ::= ⟨*parameter*⟩ `,` ⟨*parameter*⟩

⟨*parameter*⟩ ::= ⟨*type*⟩ ⟨*id*⟩

⟨*block*⟩ ::= ⟨*sentence*⟩

⟨*sentence*⟩ ::= ⟨*declaration*⟩ | ⟨*assignment*⟩ `;` | ⟨*function_ call*⟩ `;` | ⟨*if*⟩
 | ⟨*while*⟩ | ⟨*for*⟩ | ⟨*return*⟩ `;`

⟨*assignment*⟩ ::= ⟨*id*⟩ `=` ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*expression*⟩ ⟨*op*⟩ ⟨*expression*⟩ | `(` ⟨*expression*⟩ `)` | ⟨*id*⟩
 | ⟨*constant*⟩ | ⟨*literal*⟩

⟨*op*⟩ ::= `+` | `-` | `*` | `/`
 | `%` | `==` | `!=` | `<`
 | `>` | `<=` | `>=` | `&&` | `||`

⟨*if*⟩ ::= `if (` ⟨*expression*⟩ `) {` ⟨*block*⟩ `}` ⟨*else_ if_ opt*⟩ ⟨*else_ opt*⟩

⟨*else_ if_ opt*⟩ ::= `else if (` ⟨*expression* ⟩`) {` ⟨*block*⟩ `}` ⟨*else_ if_ opt*⟩ | ϵ

⟨*else_ opt*⟩ ::= `else {` ⟨*block*⟩ `}` | ϵ

⟨*while*⟩ ::= `while (` ⟨*expression*⟩ `) {` ⟨*block*⟩ `}`

⟨*for*⟩ ::= `for (` ⟨*assignment*⟩ `;` ⟨*expression*⟩ `;` ⟨*assignment*⟩ `) {` ⟨*block*⟩ `}`

⟨*function_ call*⟩ ::= `printf (` ⟨*string*⟩ `)` | `printf (` ⟨*string*⟩ `,` ⟨*list_ args*⟩ `)`
 | `scanf (` ⟨*string*⟩ `)` | `scanf (` ⟨*string*⟩ `,` ⟨*list_ args*⟩ `)`
 | ⟨*id*⟩ `(` ⟨*list_ args_ opt*⟩ `)`

⟨*list_ args_ opt*⟩ ::= ⟨*list_ args*⟩ | ϵ

⟨*list_ args*⟩ ::= ⟨*expression*⟩ `,` ⟨*expression*⟩

⟨*string*⟩ ::= `"` ⟨*literal*⟩ `"`

⟨*literal*⟩ ::= character ⟨*literal*⟩ | digit ⟨*literal*⟩ | symbol ⟨*literal*⟩ | ϵ

3.1.2 Structural and Logical Refactoring

While this initial grammar was a useful model, it contained several fundamental structural and logical flaws that required immediate refactoring for implementation in Bison.

Program Structure
The initial grammar enforced a rigid `<declarations> <functions> <main>` sequence. This is not representative of C, which allows variable and function definitions to be interleaved. Furthermore, `<main>` was treated as a special rule, when syntactically it is just another function.

This structure was corrected by defining a flexible, recursive `program` rule that consumes a stream of external declarations.

```
/* Refactored Grammar Structure */
program:
    /* empty */
  | program declaracion_externa
  ;

declaracion_externa:
    declaracion
  | funcion
  ;
```

The specialized `<main>` rule was removed and is now parsed by the general `funcion` rule.

The Expression-Statement Model

The most critical logical flaw was the classification of assignments and function calls as distinct *statement* types. In C, these are *expressions* that become statements when followed by a semicolon. The initial grammar made it impossible to parse common idioms like `a = b + my_func();` or `x = y = 5;`.

The grammar was refactored to adopt C's expression-oriented model:

- `<assignment>` and `<function_call>` were removed from the `<sentence>` rule.
- These constructs were added as valid rules within the `<expression>` (now `expr`) non-terminal.
- A new statement type, `expr_opcional T_SEMICOLON`, was added to `sentencia`.

```
/* Corrected Grammar Logic */
sentencia:
    /* ... other rules (if, while, etc.) ... */
  | expr_opcional T_SEMICOLON
  ;

expr:
    /* ... other rules (operators, constants) ... */
  | T_ID T_ASSIGN expr
  | expr T_LPAREN lista_args_opt T_RPAREN
  ;
```

This change also corrected the `<for>` rule. The initial grammar improperly restricted the `for` loop's initialization and increment clauses to only accept `<assignment>` rules. The final grammar correctly uses `expr_opcional`, allowing any valid expression (or none at all).

```
for_sent:
    T_FOR T_LPAREN expr_opcional T_SEMICOLON
                   expr_opcional T_SEMICOLON
                   expr_opcional T_RPAREN
           sentencia
  ;
```

Removal of Redundant Rules
The initial grammar's `<else_if_opt>` rule was removed. An "else if" is not a unique syntactic construct in C; it is simply an `else` block that contains another `if` statement. This is handled naturally by the nesting of the standard `if_sent` rule.

3.1.3 Resolution of Grammatical Ambiguities

After refactoring, the grammar was implemented in Bison, which revealed several `shift/reduce` conflicts, indicating grammatical ambiguities.

Operator Precedence and Associativity
The `expr` rule, by defining all binary operations at a single level, was massively ambiguous. Rather than rewriting the grammar with numerous non-terminals (e.g., `term`, `factor`), Bison's precedence directives (`%left`, `%right`, `%nonassoc`) were used to establish the complete C operator precedence table.

Unary vs. Binary Operator Ambiguity
A key conflict arose with tokens shared by unary and binary operations (e.g., `*`, `&`, `-`). The parser could not distinguish binary multiplication (`a * b`) from unary dereference (`*b`).

This was resolved by assigning a high precedence level to a "pseudo-token" (`T_UMINUS`) and using the `%prec` directive to "tag" the unary rules with this higher precedence, resolving the ambiguity.

```
/* Precedence table (high precedence) */
%right T_INC T_DEC T_NOT T_TILDE T_UMINUS T_SIZEOF

/* Grammar rules for 'expr' */
expr:
    /* ... */
  | T_MINUS expr %prec T_UMINUS
  | T_AMPERSAND expr %prec T_UMINUS  /* Address-of */
  | T_STAR expr %prec T_UMINUS      /* Dereference */
```

The "Dangling Else" Conflict
A classic `shift/reduce` conflict was generated by the `if-else` grammar. The parser could not determine whether an `else` should associate with the nearest or most distant `if`. This was resolved using a standard Bison idiom, giving `T_ELSE` higher precedence than a "fictional" token (`T_IFX`) representing an `if` without an `else`.

```
/* In precedence table */
%nonassoc T_IFX
%nonassoc T_ELSE

/* In 'if_sent' grammar rule */
if_sent:
    T_IF T_LPAREN expr T_RPAREN sentencia %prec T_IFX
  | T_IF T_LPAREN expr T_RPAREN sentencia T_ELSE sentencia
```

Function Parameter Ambiguity
The grammar failed to parse `int main(void)`. An attempt to add `parametros_opt ::= T_VOID` created a new `shift/reduce` conflict, as the parser could not determine if `T_VOID` was the "no parameters" marker or the *type* of a parameter (expecting a `T_ID` to follow).

This ambiguity was fully resolved by eliminating the `parametros_opt` rule and expanding the `funcion` rule to explicitly handle all three valid parameter cases:

```
funcion:
    tipo_specifier T_ID T_LPAREN T_RPAREN T_LBRACE ...
  | tipo_specifier T_ID T_LPAREN T_VOID T_RPAREN T_LBRACE ...
  | tipo_specifier T_ID T_LPAREN parametros T_RPAREN T_LBRACE ...
```

This change made the grammar non-ambiguous for all supported function definition styles.

3.1.4 Final Grammar Characteristics and Limitations

The resulting grammar is a **LALR(1)** (Look-Ahead, Left-to-Right, Rightmost derivation with 1 token of lookahead) grammar. It is parsed bottom-up by Bison and is certified conflict-free, relying on precedence directives to resolve inherent ambiguities in the C language. The grammar's sole output is an Abstract Syntax Tree (AST); all context-sensitive logic (such as type-checking) is deferred to a subsequent Semantic Validation (SDT) phase.

This grammar has several known limitations, enforced by design to manage complexity:

- **No Declarations in `for` Loops:** The grammar does not support `for (int i = 0; ...)`. The `for_sent` rule expects an `expr_opcional`, not a `declaracion`. Supporting this introduces the "typedef-name vs. variable-name" conflict (e.g., `A * b;`), which cannot be resolved by a pure LALR(1) parser without feedback from a symbol table during the parse.

- **Simplified Declarators:** The type declaration system is minimal. The grammar does not parse pointer declarations (e.g., `int *p;`) or mixed declarations (e.g., `int *a, b;`). The `var` rule only accepts simple identifiers and basic array declarations.

- **No Compound Types:** The grammar does not support compound type specifiers like `unsigned long int` or `const char`. Only a single type token (e.g., `T_INT`, `T_CONST`) is permitted per declaration.

- **No Aggregate Type Definitions:** The grammar recognizes the *use* of `struct`, `union`, and `enum` types (e.g., `struct my_struct s;`), but it does not include rules to parse the *definition* of their bodies (e.g., `struct Point { int x; int y; };`).

3.2 Declaration of Parser Type

The parser was built using auxiliary libraries focused on the generation of syntactic analyzers, based on a proposed grammar. One of the libraries chosen was Bison, which generates a bottom-up, LALR(1) type parser. Additionally, since it requires the tokens generated from the previous phase as input, the lexer's .c file is kept. Finally, an essential file is the header file to manage the necessary structures in the compiler.

With this approach, a Makefile was defined for its execution; this will take the source files and establish the build sequence, generate the executables, and finally link them to produce the expected output in the terminal.

Therefore, we will have the desired flow in place to generate each component with its specific functions, leaving the other phases of the compiler for the future.

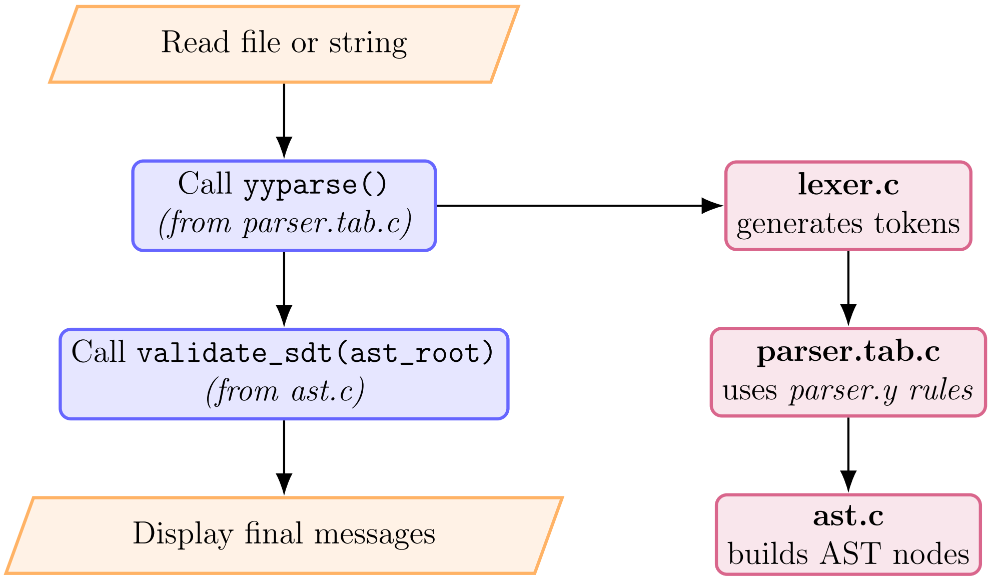

Read file or string
Call yyparse() (from parser.tab.c)
Call validate_sdt(ast_root) (from ast.c)
Display final messages
lexer.c generates tokens
parser.tab.c uses parser.y rules
ast.c builds AST nodes

Figure 1: General workflow of the syntactic analyzer implementation.

For a simpler understanding, the previous diagram is observed, which generally and clearly manages and explains the step-by-step process of this phase and what has been accomplished so far.

3.3 Description of modules

3.3.1 main

It is the start of the program, controls the analysis flow. First, it validates if the given input to check if the user provided a source code or a string to be evaluated, if a source code is provided, the program will use the function ReadFile() to read its content. If a string was provided using -s, the program will reserve memory space using malloc() and copies the string to a variable named HLL_code with strcpy(). Later, the function initScanner will start the scanner that will be used to store the given input. Finally, the main function will call the yyparse() function, which is the code generated by Bison analysis, initiating the parser variables and the stack needed for it to work properly. This function also uses the lexer to get all the tokens from the source code. This function works as the core process, since the value returned by it is the flag that will tell us if the syntax analysis worked properly or not; if it returns 0, it means that the syntax analysis was successful, otherwise, the given input was not written following grammatic rules. If it was successful, the program will validate if the SDT was done properly as well using the function validate_sdt(ast_root).

3.3.2 parser

This is the file that creates the function yyparse() that contains the grammar of the language with the semantic actions defined in C. During the execution the function yyparse() ask for the tokens that generates the lexical analyzer through a function named yylex() and every time that a grammatical rule is reduced the code associates it with that production rule is execute. In this actions are generated nodes of the abstract syntax tree using the functions defined in the file ast.c , mainly the following functions:

- *make_node*()
- *make_op_node*()
- *make_unary_op_node*()
- *make_leaf_int*()
- *make_leaf_float*
- *make_leaf_str*
- *ast_append_sibling*()

This functions allow us to construct the tree in a hierarchy form, uniting the child nodes and the brothers nodes that represent the syntax structure of the program.

The root node of the tree is stored in the global variable *ast_root*.

3.3.3 lexer

This is where the input is converted into tokens, which are used by the parser to check if the grammar rules are followed properly. As stated in a previous report, the tokens are classified in different tokens. This function also controls if the lexical analysis is done properly; if there is an unrecognized token, the lexer will tell in which line the error can be found. This function was reused and modified to work with the parser of bison.

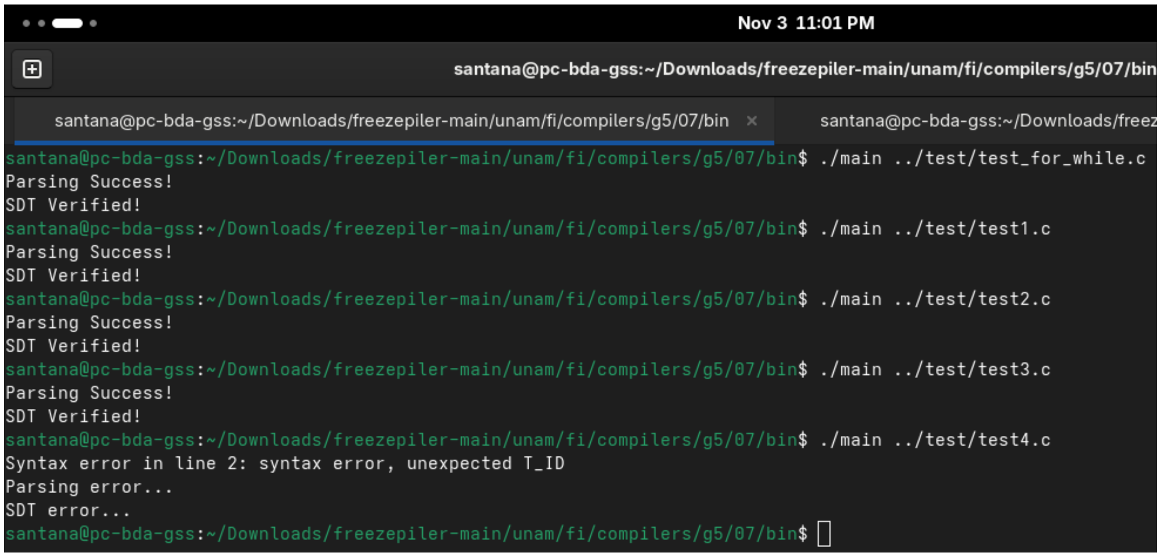
3.3.4 parser_tab

For the parser to communicate with the lexer and viceversa, Bison creates a parsing table that maps all of the token names as well as its numeric ID so it's easier to know which type of token the parser is working with. This parsing table contains all of the grammar's transitions and reductions, as well as the SDT actions that were defined.

3.3.5 ast

In this code we have the functions to construct the AST (Abstract syntax tree). It contains the validation of the semantical rules and the functions needed to print the AST.

4 Results


Nov 3 11:01 PM
santana@pc-bda-gss:~/Downloads/freezepiler-main/unam/fi/compilers/g5/07/bin
santana@pc-bda-gss:~/Downloads/freezepiler-main/unam/fi/compilers/g5/07/bin
santana@pc-bda-gss:~/Downloads/freez

```
santana@pc-bda-gss:~/Downloads/freezepiler-main/unam/fi/compilers/g5/07/bin$ ./main ../test/test_for_while.c
Parsing Success!
SDT Verified!
santana@pc-bda-gss:~/Downloads/freezepiler-main/unam/fi/compilers/g5/07/bin$ ./main ../test/test1.c
Parsing Success!
SDT Verified!
santana@pc-bda-gss:~/Downloads/freezepiler-main/unam/fi/compilers/g5/07/bin$ ./main ../test/test2.c
Parsing Success!
SDT Verified!
santana@pc-bda-gss:~/Downloads/freezepiler-main/unam/fi/compilers/g5/07/bin$ ./main ../test/test3.c
Parsing Success!
SDT Verified!
santana@pc-bda-gss:~/Downloads/freezepiler-main/unam/fi/compilers/g5/07/bin$ ./main ../test/test4.c
Syntax error in line 2: syntax error, unexpected T_ID
Parsing error...
SDT error...
santana@pc-bda-gss:~/Downloads/freezepiler-main/unam/fi/compilers/g5/07/bin$
```

Figure 2: Execution results demonstrating successful parsing and verification of multiple source files.

The provided screenshot illustrates the successful execution of the compiler's parsing phase for several test files. Specifically, four separate compilation attempts were made, each targeting a different source file.

4.1 Successful Parsing Analysis

- **Execution Command**: The command used for each successful test was *./main ../test/test_for_while.c*, *./main ../test1.c*, *./main ../test2.c*, and *./main ../test3.c*, indicating that the executable named main (the parser) was run against four different C source files located in a parent directory's test folder.

- **Parsing Outcome**: For all first four test cases, the output in the terminal was consistent: *Parsing Success! SDT Verified!*

- **Interpretation**: *The Parsing Success!* message confirms that the syntactic analysis of the input source code was completed without detecting any syntax errors. This implies that the code in all four test files adheres to the grammar defined for the language the parser is designed to handle. *The SDT Verified!* message likely refers to the successful application of a Syntax-Directed Translation (SDT) scheme.

4.2 Failed Test Case

- **Execution Command**: The command used for the failed test was *./main ../test/test4.c*, indicating that the parser executable (*main*) was run against a fifth source file.

- **Parsing Outcome**: The output in the terminal clearly indicates a failure in the analysis process: *Syntax error in line 2: syntax error, unexpected T_ID Parsing error... SDT error...*

- **Interpretation**: *The **Syntax error...** message confirms that the parser successfully detected a structural violation of the language's grammar on line 2 of the source file.* The 'unexpected T_ID' (Identifier Token) specifies the exact nature of the error, where an identifier was found in a context where it was not permitted by the grammar rules. *The **Parsing error...** and **SDT error...** messages confirm that the compilation process was correctly halted.* Since the input code was structurally invalid, the parser could not complete its analysis, thus preventing the subsequent Syntax-Directed Translation (SDT) phase from executing, ensuring the system does not proceed with erroneous code.

Based on these results, the conclusions are presented in the next section.

5 Conclusions

The results show that a syntax analyzer is correctly created to validate the source code based on a defined grammar. This validates that compiler theory, specifically context-free grammars, parsing algorithms, and syntax-directed translation, has been implemented effectively.

The project highlights how the need to build grammars, operator precedence, associativity, and address ambiguity all contribute to the development of a deterministic parsing process. The application of these principles ensured the development of an LALR(1) parser without conflicts to validate the correctness of the constructed grammar.

The creation of the Abstract Syntax Tree (AST) and interaction with a semantic validation also reinforce the importance of coherent structuring of the phases of a compiler. This reflects an understanding of how syntax and semantics analysis work together to represent the logical structure of a program.

In conclusion, the project shows that compiler theory applies in practice and that a description of a formal model is sufficiently detailed so it can be implemented with some support from tools such as Bison into a working model that correctly distinguishes and validates the syntax of programming languages. In particular, the parser is a straightforward demonstration of a successful and rigorous earning of the theoretical aspects.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006.

[2] N. ACADEMY, *Compiler design*, `https://www.youtube.com/playlist?list=PLBlnK6fEyqRjT3oJxFXRgjPNzeS-LFY-q`, Accessed: 2025-11-03, Nov. 2023.

[3] Universidad Nacional Autónoma de México, *Introducción a la Programación*, Agosto 2017. Ciudad Universitaria, Coyoacán, Ciudad de México: Facultad de Contaduría y Administración, UNAM, 2017, Apunte electrónico. Plan de estudios 2012, actualizado 2016. ISBN en trámite. [Online]. Available: `https://cedigec.fca.unam.mx/materiales/informatica/2/LI_1167_140120_A_Introduccion_Programacion_Plan2016.pdf`.

[4] GeeksforGeeks, *Working of lexical analyzer in compiler*, `https://www.geeksforgeeks.org/compiler-design/working-of-lexical-analyzer-in-compiler/`, Accessed: 2025-09-25, 2025.

[5] A. desconocido, *Análisis léxico*, `https://compiladores1.webnode.mx/ice-dea/`, Accedido: 2025-09-24, 2024. [Online]. Available: `https://compiladores1.webnode.mx/ice-dea/`.

[6] A. desconocido, *Función del analizador léxico*, `http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/21_funcin_del_analizador_lxico.html`, Accedido: 2025-09-24, 2022. [Online]. Available: `http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/21_funcin_del_analizador_lxico.html`.

[7] A. W. Appel, *Modern compiler implementation in C*, 1st ed. THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE, 2004.

[8] cppreference.com. "C keywords." Accedido: 2025-09-25. [Online]. Available: `https://en.cppreference.com/w/c/keyword.html`.

[9] cppreference.com. "Punctuation." Accedido: 2025-09-25. [Online]. Available: `https://en.cppreference.com/w/c/language/punctuators.html`.

[10] GeeksforGeeks. "Keywords in c." Accedido: 2025-09-25. [Online]. Available: `https://www.geeksforgeeks.org/c/keywords-in-c/`.