Assignment 2: Syntax Analysis with Bison

CS 464/5607: Compiler Design

Spring 2026

1 Introduction

In this assignment, you will implement the second phase of a compiler: the **Syntax Analyzer** (or Parser).

Your task is to use **Bison** to generate a parser that consumes the stream of tokens produced by the lexer and verifies if they follow the grammatical rules of our language. Furthermore, you will construct an **Abstract Syntax Tree (AST)** that represents the hierarchical structure of the source code.

2 Understanding Bison (The Parser Generator)

2.1 What is Bison?

Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employed in C programs. While Flex handles the *words* (lexical analysis), Bison handles the *grammar* (syntax analysis) by defining how those words fit together to form meaningful statements.

2.2 Why are we using it?

Writing a parser by hand (e.g., a Recursive Descent parser) for a complex language can be tedious and error-prone. Bison automates the difficult task of constructing the parsing tables and state machines required to validate code structure. It allows you to focus on defining the high-level grammar rules rather than worrying about the low-level details of lookahead tokens and state transitions.

2.3 How Bison Works

When you run the command `bison -d parser.y`, the following happens in the background:

1. **Grammar Processing:** Bison reads your `.y` file to understand the tokens, non-terminals, and grammar rules.
2. **State Machine Construction:** It constructs an LALR(1) parsing table (Look-Ahead Left-to-Right) which represents the valid states and transitions of your language.
3. **C Code Generation:** It generates a C source file (typically `parser.tab.c`) containing the `yyparse()` function. This function uses the parsing table to process tokens one by one.
4. **Header Generation:** The `-d` flag instructs Bison to generate a header file (`parser.tab.h`). This file exports the Token IDs (enums) so that your Lexer knows exactly what integer values to return for each token type.

Note: Just like Flex, you must have Bison installed on your development environment to generate the parser code.

3 Provided Files

The following files are provided in the skeleton code.

3.1 Lexer Source

- `lib/lexer.l`: This file contains the Lexical Analyzer definitions using Flex. During the build process, this file is compiled to generate the C code that handles token generation.

3.2 Source Files (Your Task)

- `src/parser.y`: The Bison grammar file. This is where you will define the grammar rules, precedence, and AST construction logic.
- `src/ast.cpp`: The implementation file for the Abstract Syntax Tree. You must implement the node creation and management functions here.
- `src/ast.h`: The header file defining the `ASTNode` structure and NodeType enums.

3.3 Driver Code (Do Not Modify)

- `src/driver.cpp`: The entry point that calls `yyparse()` and prints the resulting AST or error messages.

4 The Abstract Syntax Tree (AST)

An **Abstract Syntax Tree (AST)** is a tree representation of the abstract syntactic structure of source code. Unlike a Parse Tree, which contains every detail of the grammar (including parentheses and semicolons), an AST contains only the structural details essential for analysis and code generation.

4.1 Why do we need it?

The parser validates the code, but simply returning "Success" or "Failure" is insufficient for a compiler. We need a data structure that captures the meaning of the program so that subsequent phases (Semantic Analysis, Code Generation) can traverse it.

4.2 API Functions

You must implement the following functions in `src/ast.cpp`:

4.2.1 `create_node`

```
ASTNode* create_node(NodeType type, char* data, int line, ASTNode* left,
    ASTNode* right);
```

Allocates memory for a new `ASTNode`, initializes its fields, and returns a pointer to it.

- `type`: The kind of node (e.g., `NODE_VAR_DECL`, `NODE_IF`).
- `data`: A string for storing values like identifiers ("x"), operators ("+"), or literals ("10").

4.2.2 `append_node`

```
ASTNode* append_node(ASTNode* head, ASTNode* new_node);
```

Used for handling lists of statements or declarations. This function should attach `new_node` to the end of the `next` chain starting at `head`.

4.2.3 `free_ast`

```
void free_ast(ASTNode* node);
```

Frees all memory associated with the AST nodes to prevent memory leaks.

4.2.4 Note on print_ast

The function `void print_ast(ASTNode* node, int level)` has already been implemented for you. **DO NOT MODIFY THIS FUNCTION.** It formats the output exactly as required by the test cases.

5 The Parser (parser.y)

5.1 The %union

Bison parsers use a union to hold different types of semantic values. In your skeleton code, it is defined as:

```
%union {
    char* sval;           // For lexemes (IDs, Literals)
    struct ASTNode* node; // For AST Nodes
}
```

- `sval`: Used for tokens that carry raw text, such as `T_ID` (identifiers) or `T_INT_LIT` (numbers).
- `node`: Used for non-terminals (like `program`) that produce an AST node as a result.

Example Usage:

```
%token <sval> T_ID
%type <node> program
```

5.2 Operator Precedence and Associativity

Ambiguities in expressions (like `1 + 2 * 3`) must be resolved using precedence rules. In Bison, you define these using `%left`, `%right`, or `%nonassoc`.

- Operators declared **later** have **higher** precedence.
- `%left` indicates left-associativity (e.g., `a - b - c` becomes `(a - b) - c`).

Example:

```
/* Low Precedence */
%left T_PLUS T_MINUS
%left T_MULT T_DIV
/* High Precedence */
```

5.3 Grammar Rules

You must define the grammar rules for the language constructs. A Bison rule defines a non-terminal (LHS) in terms of terminals and other non-terminals (RHS), followed by a block of C code that executes when the rule is matched.

```
LHS : RHS_1 RHS_2 ... RHS_N { Action Code }
```

5.3.1 Example: Assignment Statement

To help you understand how to construct the AST, let's look at how you might implement an **assignment** rule (e.g., `x = 5;`).

```
assignment
    : T_ID T_ASSIGN expression T_SEMI {
        /* $1 corresponds to T_ID (the variable name, e.g., "x")
           $3 corresponds to expression (the value, e.g., "5")
        */
```

```

        // 1. Create a node for the variable on the Left Hand Side
        ASTNode* varNode = create_node(NODE_VAR_USE, $1, yylineno, NULL, NULL);

        // 2. Create the main Assignment node
        //    Left Child = variable, Right Child = expression
        $$ = create_node(NODE_ASSIGN, "=", yylineno, varNode, $3);
    }
    ;
```

Explanation of Symbols:

- `$$` (Dollar-Dollar): This represents the **result** of the current rule. In the example above, we assign the new `NODE_ASSIGN` pointer to `$$`, which passes it up to the parent rule (e.g., `statement`).
- `$1`: This refers to the value of the **first** component on the RHS. Since `T_ID` is defined as `<sval>` in the %union, `$1` holds the string name of the identifier (e.g., "count").
- `$3`: This refers to the **third** component, which is `expression`.

5.3.2 Implementation Hints

- **Program Structure:** A program is a list of declarations (variables or functions).
- **Expressions:** Use the precedence rules defined above to handle math logic without writing complex recursive grammar rules manually.
- **Actions:** Inside the curly braces {}, use the `create_node` function to build the tree.
  - `$$` refers to the result of the LHS (the parent node).
  - `$1`, `$2`, etc., refer to the values of the symbols on the RHS.

5.4 Node Configuration Reference

When calling `create_node` inside your grammar actions, you must map the children pointers (`left` and `right`) correctly. Since the AST is strictly binary, nodes that logically require three components (like `if-else` or `for`) must use nested child nodes.

- **Variable Declaration:**
  - `left`: The Type node (e.g., "int", "float").
  - `right`: The Initialization expression (or `NULL` if not initialized).
  - *Note:* For arrays, the size is embedded directly into the variable name string (e.g., "arr[10]") or handled via the variable identifier logic.
- **If Statement:**
  - `left`: The Condition expression.
  - `right`: The Body.
    - If an `else` exists, `right` points to a special `NODE_BLOCK` (named "IfElseBranches").
    - Inside this block:
      - `left`: The "Then" Block.
      - `right`: The "Else" Block.
- **While Loop:**
  - `left`: The Condition expression.
  - `right`: The Body Block.
- **For Loop:**
  - **Syntax:** `for (init; condition; update;) { body }`

  - Because a `for` loop has 4 components, it is split across multiple binary nodes:
  - **Top Node (NODE_FOR):**
    * `left`: Initialization Statement.
    * `right`: A placeholder `NODE_BLOCK` (named "LoopRest").
  - **LoopRest Node:**
    * `left`: Condition Expression.
    * `right`: A placeholder `NODE_BLOCK` (named "LoopScope").
  - **LoopScope Node:**
    * `left`: Update Statement.
    * `right`: Loop Body.

  Note: Notice the semicolon at the end of the update statement. This simplifies the implementation by allowing the grammar to reuse the standard Assignment Statement rule directly, rather than requiring a separate rule for assignments without semicolons.

- **I/O Statements:**
  - `data`: Set to `"read"` for `cin` or `"print"` for `cout`.
  - `left`: The variable (for read) or expression (for print).
  - `right`: Always `NULL`.

5.5 Panic Mode Error Recovery

To make your parser robust, you must implement **Panic Mode Recovery**. When a syntax error occurs, the parser should not crash; instead, it should discard tokens until it finds a "synchronizing token" (like a semicolon or a closing brace) and then resume parsing.

How to implement: Use the special `error` token provided by Bison.

```
statement
    : ... (valid rules) ...
    | error T_SEMI { yyerrok; }
    ;
```

This tells Bison: "If you find an error, skip everything until you see a semicolon, then clear the error flag (`yyerrok`) and continue."

5.6 Error Reporting

The function `void yyerror(const char *s)` is already provided in the skeleton code. It is automatically called by Bison when a parse error occurs. It prints the line number and the current token to `stderr`. This has already been implemented in parser.y but you may modify it accordingly.

6 Building and Testing

6.1 Compilation

To build the project, open your terminal in the project root directory and run:

```
make
```

This will compile your parser source code and link it with the provided `lexer.o` object file. The executable will be placed in the `build/` directory.

6.2 Running the Test Suite

We have provided a Python script to automate testing. To run it, execute:

```
python run_tests.py
```

The script compares your parser's output against "Golden Output" files.

- **60%** of the grade is based on the visible tests provided to you.
- **40%** of the grade is based on **Hidden Test Cases** that check for edge cases and robustness.

6.3 Testing with Custom Inputs

You are encouraged to create your own test cases to debug specific issues.

1. Create a text file (e.g., `my_input.txt`) with some code.
2. Run your built parser and redirect the input:

On Windows:

```
build\parser.exe my_input.txt
```

On macOS/Linux:

```
./build/parser my_input.txt
```

This will print the AST to the terminal, allowing you to manually verify if your rules are working as expected.

6.4 Platform Specifics (Windows)

If you are working on Windows natively (Command Prompt or PowerShell, not WSL), you must modify the build scripts to handle file extensions correctly.

1. Modify the Makefile

Windows executables require the `.exe` extension. Find the `TARGET` variable and update it:

```
# Change:
TARGET = $(BUILD_DIR)/syntax

# To:
TARGET = $(BUILD_DIR)/syntax.exe
```

2. Modify run_tests.py

The Python script needs to know it should look for an executable with an extension.

```
# Change:
PARSER_EXEC = os.path.join("build", "syntax")

# To:
PARSER_EXEC = os.path.join("build", "syntax.exe")
```

7 Submission Requirements

Please follow the steps below carefully to ensure your assignment is graded correctly.

7.1 Preparing Your Submission

Before zipping your project, you must clean the directory to remove all generated build files and executables.

1. Run the following command:

   ```
   make clean
   ```

2. Verify that the `build/` directory has been removed.

7.2 Naming Convention

You must compress your project folder into a single **.zip** file. The filename must strictly follow this format:

`<rollnumber>_PA2.zip`

Replace `<rollnumber>` with your actual roll number.
Example:

- If your roll number is **27100289**, your submission file must be named:

  `27100289_PA2.zip`