SWE3006 Programming Languages
Assignment 2 Report:
C-AST Generation with Python

Younwoo Park (박연우)
Professor Tamer
System Management Engineering & Computer Science and Engineering
April 11th, 2025

1. Overview

The two goals of this assignment were to 1. Generate abstract syntax tree (AST) from C source code and 2. Traverse the generated AST to compute the output using Python. The project specifically excludes the use of external libraries (e.g., `pycparser`), requiring all functionality—from tokenization to evaluation—to be built from scratch. The implementation adheres to C's operational semantics, particularly for operations like division and modulus, but does not consider conditionals, loops, global variables, and many more. The project was implemented in reference to the `pycparser`[1] from the Python libraries.

2. Node Definitions

The AST nodes are structured using custom classes with attributes and child iterators for traversal. Having to match the output exactly as the `pycparser`, the original code[2] was heavily referenced. The base class `Node` serves as an abstract base class, defining the `show()` function for the AST printing. Major C-like constructs are supported such as `ExprList`, `Compound`, `FuncCall`, `Constant`, `ID`, `BinaryOp`, `UnaryOp`, `Assignment`, `Return`, `FuncDef`, and `FuncDecl`. For more detailed information, please refer to the code.

3. Lexer

The lexical analysis of the C-code is done through the class called **Lexer,** and specifically through the `tokenize()` function. The **Lexer** class takes the C source code and tokenizes it into a list of `(TYPE, VALUE)`. Supported token types include:

- **`KEYWORD`**: C keywords such as `int, return, float, double, void`
- **`ID`**: Identifiers, including variable names and function names
- **`NUM, FLOAT`**: Numeric constants (`NUM` refers to int)
- **`STRING`**: String literals (used in printf)
- **`SYMBOL`**: Operators and punctuation such as =, +, ;, *, etc.

Whitespace is inserted before and after the symbols (in order to parse it into a list appropriate with the `split()` function). The `%` symbol is handled separately in order to differentiate its usage as a modulus operand and a string formatting operand (e.g. `"%d"`). Lastly, the raw tokens are then classified into their respective types.

4. Parser

The **Parser** class takes as input a list of tokens—returned by the Lexer—and converts them into an AST that accurately represents the structure of the source code. It supports function definitions, prototypes, statements, expressions, and type declarations. The parser is implemented using a recursive descent approach and handles operator precedence properly during expression parsing.

---

[1] https://github.com/eliben/pycparser/
[2] https://github.com/eliben/pycparser/blob/main/pycparser/c_ast.py

4-1. Core Functions

`parse()`

This is the Parser's entry point: it loops through the token stream and decides whether each top-level declaration is a function prototype or a function definition—only need to consider these two thanks to the assignment's constraints. There is a helper function called `_is_prototype()` to check if the function declaration has a body.

`_parse_funcdef()`

This function builds a `FuncDef` node for a function definition. It handles:

1. Parsing the return type and name
2. Parsing function parameters using `_parse_type()` and `_parse_identifier()`
3. Parsing the function body as a Compound node containing declarations and statements

`_parse_prototype()`

Builds a `Decl` node with a `FuncDecl` child to represent a function prototype (e.g., `int foo(int x);`).

`_parse_expression(min_prec=0)`

This function handles all binary expressions, respecting operator precedence:

1. Parses the left-hand side (LHS) first
2. Checks if the current operator's precedence is high enough using `min_prec`
3. Recursively parses the right-hand side (RHS) with updated precedence

`_parse_statement()`

Handles parsing of high-level statements like:

- return statements →`_parse_return()`
- Assignment→`_parse_assignment()`
- Function call→`_parse_func_call()`
- Expression statements

`_parse_assignment()` and **`_parse_func_call()`**

Constructas AST nodes for assignment and function call statements. The parser checks the token pattern to determine whether an identifier is being assigned, called as a function, or used as an expression.

`_parse_declaration()`

Parses a variable declaration, creating a `Decl` Node with optional initialization (i.e. `int a;` and. `int a = 3;` are not supported).

4-2. Operation Precedence Handling

The parser uses a `_precedence` dictionary—referenced from the `pycparser`—to store C-like precedence levels for all supported operators. The `_parse_expression` function ensures correct grouping of operations based on these precedence levels, simulating how C would evaluate expressions.

5. Evaluator

The **Evaluator** class is responsible for executing the Abstract Syntax Tree (AST) generated by the parser. It simulates the behavior of a simple C-like interpreter, evaluating expressions, function calls, and statements while maintaining environments for variables and functions.

5-1. Internal Attributes

The evaluator maintains three main components:

```
_variables: dict[str, any]     # Stores local vars for the current function
_functions: dict[str, FuncDef] # Stores function definitions
_return_val: any               # Temporarily stores ret values of functions
```

Each function call resets `_variables` and manages scope isolation by saving/restoring the variables.

5-2. Core Functions

`_eval_FileAST(node)`

Entry point of the evaluator; traverses the AST and adds all function definitions to `_functions`.

`_eval_FuncDef(node)`

Handles execution of a function. It resets `_variables`, ensures any previous return value is cleared, and then evaluates the function body

`_eval_ID(node)`

Returns the value associated with a variable name, which could be a Constant or another AST node. If it's a node, it is recursively evaluated.

`_eval_FuncCall(node)`

This function supports:

- Special handling of the printf function with support for `%d, %f, %lf`
- Execution of user-defined functions, including argument evaluation and scoped variable mapping

When calling user-defined functions:

1. Arguments are evaluated
2. Current variable state is saved
3. A new `_variables` dict is created for the function
4. The function body is evaluated
5. Old state is restored

5-3. C Behaviour Matching

To ensure accurate simulation of C-like behavior, **`_eval_BinaryOp(node`** considers the following:

- Integer division uses truncation toward zero (via `int(left / right)`)
- Modulo follows `left - int(left / right) * right`
- Bitwise operations simulate 32-bit signed overflow using & `0xFFFFFFFF`

6. Conclusion

By integrating a lexer, parser, AST representation, and evaluator, the system is capable of reading C source code, generating an abstract syntax tree, and accurately executing function definitions and expressions with behavior that closely mirrors that of a C compiler. Special attention was given to simulating C-specific semantics, such as integer division rounding toward zero. Overall, the project demonstrates a deep understanding of language processing fundamentals and the intricacies of C-style execution semantics, and it provides a functional and extensible tool for evaluating simple C programs.