CS3423

Assignment - 1 | Compilers - 2

CS22BTECH11051@IITH.AC.IN - Armaan

CS22BTECH11038@IITH.AC.IN - Sriman

Instructions

The following is the folder structure of our submission.

```
.
├── build
│   ├── makefile
│   └── texput.log
├── ding-dong.o
├── src
│   ├── lexer.l
│   └── parser.y
└── test
    ├── input
    ├── output
    ├── parsed
    ├── printed
    └── tokens
```

1. `cd` into the build directory
2. run `make clean` to clean all binaries and files generated midway
3. run `make` to compile the `lexer` and `parser` files in the appropriate directories
4. run `make run1` or `make run2` or `make run3` or `make run4` or `make run5` to transpile the respective input<x> file to the appropriate directories.
5. Please find the inputs at `./test/input` and the outputs at `./test/outputs`

Assumptions

Input Validity

It is assumed that the input source code adheres to the defined syntax. The lexer and parser do not include extensive error recovery mechanisms, implying:

- **User Responsibility**: Users are expected to provide syntactically correct code, and any deviations will result in parsing errors.

Language Constructs

The lexer and parser are designed around a predefined set of language constructs and reserved words. This assumption means:

- **Limited Language Features**: New features or constructs must be explicitly added to the lexer and parser definitions, which may require modifications to existing code.

Error Handling

Error handling is minimal, primarily focusing on critical errors. This assumption indicates:

- **Basic Robustness**: The implementation is robust enough to identify and report syntax errors but may not recover from them gracefully.
- **Enhancement Opportunities**: There is room for enhancing error handling to improve user feedback during compilation.

Environmental Setup

The successful compilation and execution of the lexer and parser assume a specific environmental setup, including:

- **Compiler Compatibility**: The code is intended to be compiled with a compatible C compiler, which may not support all extensions available in newer compilers.
- **File Handling**: The implementation relies on the presence of specific output files for logging tokens and parser outputs, necessitating proper file permissions and availability.

Justifications

Lexical Analysis

The lexer is responsible for tokenising the input source code. The design choice to use regular expressions for defining token patterns is justified due to the following reasons:

- **Readability**: The use of regular expressions enhances the readability of the token definitions, making it easier for future maintainers to understand the language grammar.

Token Information

The `token_info` function logs token details including line number, token type, and the actual token string. This decision serves several purposes:

- **Debugging**: The detailed logging of token information aids in identifying issues during lexical analysis and assists in debugging by providing insights into token generation.
- **Error Reporting**: Capturing line numbers and token types helps in generating meaningful error messages for the user, thus improving the overall user experience.

BNF Grammar Implementation

The parser utilises BNF (Backus-Naur Form) for defining the language grammar, which is justified by:

- **Clarity**: BNF provides a clear structure for defining the syntax of the programming language, enhancing the understandability of language constructs.
- **Flexibility**: This approach allows easy modifications and extensions to the grammar as the language evolves.

Memory Management

Dynamic memory allocation is employed for creating token strings and expressions. The following justifications support this choice:

- **Scalability**: Dynamic allocation allows the program to handle varying sizes of tokens and expressions, accommodating a range of user input.
- **Controlled Lifespan**: Memory can be allocated and freed appropriately, preventing memory leaks and ensuring efficient use of resources.

Operator Precedence and Associativity

The implementation of operator precedence and associativity in the parser ensures that expressions are evaluated correctly. This decision is critical for:

- **Correctness**: It guarantees that the parser respects the standard rules of arithmetic and logical operations, leading to expected and correct results during evaluation.
- **User Expectations**: Following conventional operator precedence aligns the language behaviour with user expectations derived from other programming languages.