CSE341 HW2 Report

Gpp Lexer/Interpreter

R.Hilal SAYGIN

Implementation with LISP

Lexer

Overview

I designed a lexer to tokenize input text for a simple programming language.
It identifies keywords, operators, identifiers, integers, fractional numbers, and comments.
The program processes the input string, splits it into tokens, and determines the type of each token based on predefined rules. Therefore handling lexical analysis, which is the first stage of a compiler or interpreter pipeline is accomplished.

Execution Flow

1. **Token Initialization**:
    - Initializes lists of predefined **keywords** and **operators**.
    - State variables are declared for tracking the current parsing context (e.g., `*state*`, `*error-state*`, `*numpre-state*`).
2. **Main Function (`gpp-lexer`)**:
    - It reads the input either from a file (using `read-input-file`) or directly from the user.
    - The input is then passed to `split-words` for token extraction.
    - If any lexical error is detected, it prints an error message and terminates the program.

Functions

1. **`read-input-file`**:
    - Reads the content of a specified file and returns it as a string.
    - Uses `with-open-file` to handle file I/O.
2. **`split-words`**:
    - Recursively processes the input string to extract individual tokens.
    - It checks for comments, keywords, operators, whitespace, and identifiers.
    - When a token is identified, it calls `set-final-state` to determine the token type.
    - If a lexical error occurs, it updates `*error-state*` and stops further processing.
3. **`extract-token`**:
    - Extracts a single token from the input string based on the current character.
    - Handles special cases such as comments (`;;`), operators (`+`, `-`, `*`, `/`, `(`, `)`), and whitespace.
    - Returns the extracted token and the updated cursor position.

4. **`set-final-state`**:
    - Determines the type of a given token.
    - Sets the appropriate state based on whether the token is a keyword, operator, integer, fractional number, or identifier.
    - Updates `*error-state*` if the token is invalid.
5. **`set-operator-state`**:
    - Sets the lexer state based on the operator token.
    - Maps specific operators (`+`, `-`, `*`, `/`, `(`, `)`) to corresponding state values (`OP_PLUS`, `OP_MINUS`, `OP_MULT`, etc.).
6. **`reset-states`**:
    - Resets all state variables to their initial values.
    - Called after processing each token to ensure that states do not persist across tokens.

Validation Functions

1. **`is-valuei`**:
    - Checks if a given token is a valid integer.
    - Recursively verifies that all characters in the token are digits.
2. **`is-valuef`**:
    - Checks if a given token is a valid fractional number (e.g., `3f2` representing `3/2`).
    - Ensures that the token has a valid numerator and denominator separated by `f`.
    - Handles edge cases such as leading zeros.
3. **`is-leading-zero`**:
    - Detects if a token has a leading zero (e.g., `012`), which is typically invalid for numeric literals.
4. **`is-identifier`**:
    - Validates if a token is a valid identifier.
    - An identifier can start with an alphabetic character or an underscore (`_`) and may contain digits.
    - If the token does not conform to these rules, a lexical error is raised.

Data Structures

1. **Keywords List (`*keywords*`)**:
    - A predefined list of keywords used in the target language (e.g., `and`, `or`, `if`, `deffun`, `exit`).
    - Tokens matching these strings are classified as keywords.
2. **Operators List (`*operators*`)**:
    - A predefined list of operators (`+`, `-`, `*`, `/`, `(`, `)`).
    - Tokens matching these characters are classified as operators.
3. **State Variables**:
    - `*state*`: Tracks the current state of the lexer (e.g., `start`, `VALUEI`, `VALUEF`, `IDENTIFIER`).
    - `*error-state*`: Stores the current error state, if any.
    - `*comment-state*`: Indicates whether the lexer is currently processing a comment.
    - `*numpre-state*`, `*numpos-state*`, `*f-state*`: Used for tracking the parsing of numeric values.

Error Handling

- The lexer uses `*error-state*` to track lexical errors.
- If an invalid token is detected (e.g., an unrecognized character or an improperly formatted number), the lexer sets `*error-state*` and prints an error message.
- The program terminates upon detecting a lexical error, ensuring that only valid tokens are processed.

Special Cases

1. **Handling Comments**:
    - The lexer recognizes comments that start with `;;` and continue until the end of the line.
    - The entire comment is added as a token of type `COMMENT`.
2. **Whitespace and Operators**:
    - Whitespace characters (`Space`, `Tab`, `Return`, `Newline`) are used to separate tokens but are not included as tokens themselves.
    - Operators are handled separately and immediately returned as tokens when encountered.

Interpreter

Execution Flow

1. **Loading Lexer**:
    - using `(load "gpp_lexer.lisp")`.
    - The lexer is implemented for tokenization of the input code represented as pairs (`token`, `value`).
2. **Grammar Rules Definition**:
    - Grammar rules (e.g., `*EXP1*`, `*EXP2*`, `*EXP4*`, etc.) define the syntax of expressions, function calls, boolean operations, and control structures.
    - These rules serve as templates for parsing and evaluating the input code.
3. **Main Driver Function (`gpp-driver`)**:
    - The entry point for the interpreter.
    - It calls the main interpreter function `gppinterpreter` with optional command-line arguments.

Functions

1. **`gppinterpreter`**:
    - This is the core function of the interpreter, responsible for tokenization, parsing, syntax checking, and evaluation.
    - It uses the `gpp-lexer` function to obtain the list of tokens from the input file.
    - The parsing process starts by calling `gpp-parse`, which recursively processes tokens and builds a parse stack.
    - After parsing, it evaluates the expressions using `evaluate-parse-stack`.
2. **`gpp-parse`**:
    - Handles the parsing logic using a recursive approach.
    - It processes tokens, manages the function definition state (`*func_def_state*`), and calls `reduce-stack` to simplify the parse stack based on grammar rules.

    - When a nested expression or function is detected, it calls itself recursively to handle sub-expressions.
3. **`reduce-stack`**:
    - Application of grammar rules to simplify the parse stack.
    - It checks for matching grammar rules using `matching-rule` and reduces the stack based on these rules.
    - It handles specific cases such as function definitions (`KW_DEFFUN`) and various expressions (`EXP1`, `EXP2`, etc.).
4. **`evaluate-and-reduce`**:
    - This function evaluates and reduces expressions based on the matched grammar rule.
    - It handles arithmetic operations (`EXP1`, `EXP2`, `EXP3`), variable assignments (`EXP5`, `EXP6`), conditional expressions (`EXP4`), and boolean operations (`EXPB1`, `EXPB2`, etc.).
    - For function calls, it retrieves function properties from the function table and evaluates user-defined functions.

Data Structures

1. **Symbol Table (`*symbol_table*`)**:
    - A list of `(id, value)` pairs storing variables and their values.
2. **Function Table (`*function_table*`)**:
    - A list of function properties: `(function_name, argument_list, expression_list)`.
3. **Parse Stack**:
    - A list used to store tokens and their associated values during parsing.
    - Simplified and reduced during the parsing process based on grammar rules.

Results

```
>> (defvar b 3f4)

Syntax matched successfully. Tokens: (EXP)

3f4
```

Correct tokenization and interpretation of var assignment.

```
>> defvar b 3f4

*** - CHAR: index 12 should be less than the length of the string
```

Wrong var assignment example in gpp language

```
>>  (list 1 2 123f12)

Syntax matched successfully.n Tokens: (OP_OP KW_LIST EXP EXP EXP OP_CP)
```

Correct list definition.

Varying length of list definitions handled.

```
>> (list 1 2 34 5 123f12)

Syntax matched successfully.n Tokens: (OP_OP KW_LIST EXP EXP EXP EXP EXP OP_CP)
```