Gebze Technical University Department of Computer Engineering CSE 331 Fall 2023 Homework 02 Due date: Dec 31st, 2023

Ziya Kadir TOKLUOĞLU 210104004228

1. Abstract Syntax Tree (AST):

Components:

- **AST_Node Class:**
- Represents a node in the AST.
- Attributes:
- **`token`**: The type of the token.
- **`children`**: Child nodes of the current node.
- **`name`**: Identifier or function name.
- **`numerator`**: Numerator value for numeric nodes.
- **`denominator`**: Denominator value for numeric nodes.
- **MyFunction Class:**
- Extends **`AST_Node`** for representing functions.
- Additional Attributes:
- **`parameter`**: Parameter node for the function.
- **`parameter_value`**: Value of the parameter.

Functions:

- **interpret-shell:**
- Initiates the REPL (Read-Eval-Print Loop) for interactive interpretation.
- **interpret-file(filename):**
- Reads and interprets a Lisp file specified by the filename.
- **interpreter(seq):**
- Core interpreter function. Tokenizes the input sequence and builds the AST.
- **parse(tokens):**
- Implements shift-reduce parsing to build the AST from tokens.
- **separate-children(node):**
- Recursive function to separate children of a node for clarity and organization in the AST.

2. Shift-Reduce Parsing:

Overview:

Shift-reduce parsing is a technique used to build the AST by shifting tokens onto a stack and reducing them based on production rules.

Functions:

- **my_shift(token, stack):**
- Shifts a token onto the stack.
- **print-stack(stack):**
- Prints the current state of the stack for debugging.
- **check-reduction(stack):**
- Checks if a reduction can be applied based on the current stack.

- **match-production-rule(tokens):**
- Matches tokens with production rules.
- **perform-reduction(reduction, stack):**
- Performs the reduction based on the identified reduction type.

3. Production Rules:

Rules:

- **plus-exp, minus-exp, mult-exp, div-exp:**
- Expressions involving arithmetic operations.
- **id-exp:**
- Identifier (variable or function name).
- **func-call-void, func-call-one, func-call-two:**
- Function call with different numbers of parameters.
- **call-func-void, call-func-one, call-func-two:**
- Calling a function with different numbers of parameters.
- **if-stat:**
- If statement with conditions and branches.

```
(defun match-production-rule (tokens)

(cond

((equal tokens '("OP_OP" "OP_PLUS" "EXP" "EXP" "OP_CP")) 'plus-exp)

((equal tokens '("OP_OP" "OP_MINUS" "EXP" "EXP" "OP_CP")) 'minus-exp)

((equal tokens '("OP_OP" "OP_MULT" "EXP" "EXP" "OP_CP")) 'mult-exp)

((equal tokens '("OP_OP" "OP_DIV" "EXP" "EXP" "OP_CP")) 'div-exp)

((equal tokens '("OP_OP" "KW_DEF" "EXP" "EXP" "OP_CP")) 'func-call-void)

((equal tokens '("OP_OP" "KW_DEF" "EXP" "EXP" "EXP" "OP_CP")) 'func-call-one)

((equal tokens '("OP_OP" "KW_DEF" "EXP" "EXP" "EXP" "EXP" "OP_CP")) 'func-call-two)

((equal tokens '("OP_OP" "EXP" "OP_OP" "OP_CP" "OP_CP")) 'call-func-void)

((equal tokens '("OP_OP" "EXP" "EXP" "OP_CP")) 'call-func-one)

((equal tokens '("OP_OP" "EXP" "EXP" "EXP" "OP_CP")) 'call-func-two)

((equal tokens '("OP_OP" "KW_IF" "OP_OP" "EXP" "OP_CP" "OP_OP" "EXP" "OP_CP" "OP_OP" "EXP"
"OP_CP" "OP_CP")) 'if-stat)

((equal tokens '("IDENTIFIER")) 'id-exp)

((equal tokens '("VALUEF")) 'valuef-exp)

; Add cases for other production rules
```

(t nil)))

4. Reducing:

Overview:

The reducing process applies production rules to simplify and build higher-level constructs in the AST.

Functions:

- **check-reduction(stack):**
- Checks for applicable reduction types based on the current stack.
- **match-production-rule(tokens):**
- Matches tokens with production rules.
- **perform-reduction(reduction, stack):**
- Performs the reduction based on the identified reduction type.

5. Separating Children:

Overview:

The **`separate-children`** function is a utility function to organize child nodes within an AST node for clarity and ease of interpretation.

Function:

- **separate-children(node):**
- Recursive function to separate children of a node, enhancing the structure of the AST.

These documentation sections provide an overview of each component's purpose, key functions, and their roles in the Lisp interpreter.

You can run first part with command
make if
make void
make one
make two
make basic
make complicated

You can run second part with command
make compile
make run