PROJECT 2

UC - Programação Funcional e em Lógica
Turma 2

Luis Vieira Relvas (up202108661) *(Contribuição: 70%)*
Válter Ochôa de Spínola Catanho Castro (up201706545) *(Contribuição 30%)*

Contents

PART 1

Description

This code defines a simple stack-based virtual machine with an instruction set represented by the Inst data type. The virtual machine operates on a stack (Stack) and a state (State). The goal of the code is to execute a sequence of instructions (Code) and produce the final state of the stack and memory.

Instruction Set

The instruction set includes operations such as pushing values onto the stack (Push), arithmetic operations (Add, Mult, Sub), logical operations (True, Fals, Equ, Le, And, Neg), variable manipulation (Fetch, Store), control flow (Branch, Loop), and a no-operation instruction (Noop).

Stack and State

The Stack is a data structure that holds either integers or booleans. The State is a mapping between variable names and their corresponding values on the stack.

Execution

The run function takes a tuple consisting of a code to execute, an initial stack, and an initial state. It iteratively processes the instructions until the code is empty, resulting in a final stack and state.

Example:

In this example we will use this test from the provided ones:

```
testAssembler [Push 5,Store "x",Push 1,Fetch "x",Sub,Store "x"] == ("","x=4")
```

Step-by-Step:

1) Push 5 onto the stack.
2) Store x on the top of the stack into the variable x in the state.
3) Push 1 onto the stack.
4) Fetch the value of the variable “x” from the state and push it onto the stack.
5) Subtract the top value (1) from the second-to-top value (5) on the stack.
6) Store the result of the subtraction (4) into the variable “x” in the state.

Expected Result:

After executing the code, the expected result is `(“”, “x=4”)`, indicating that the output string is empty (“”) and the state variable “x” is assigned the value 4 (“x=4”).

Error Handling

The run function includes error handling for various scenarios, such as attempting arithmetic or logical operations with an insufficient number or incompatible types of operands, accessing an undefined variable, or encountering a run-time error during control flow.

Functions

createEmptyStack
Creates an empty stack.

stack2Str
Converts a stack to a string representation.

createEmptyState
Creates an empty state.

state2Str
Converts a state to a string representation.

run
Executes a sequence of instructions on a given stack and state, producing the final stack and state. Includes error handling for various scenarios.

Testing

The testAssembler function is provided to test the assembler. It takes a compiled code as input and returns the final stack and state after running the code. This function is useful for verifying the correctness of the compiler and ensuring that the generated code produces the expected results.

PART 2

Description

This code defines a simple imperative programming language with arithmetic and boolean expressions. The language includes integer variables, arithmetic expressions (Aexp), boolean expressions (Bexp), and statements (Stm). The goal is to compile a program written in this language into a stack-based virtual machine code.

Syntax

Arithmetic Expressions (Aexp)

- **IntLit Integer**
- **Var String**
- **AddA Aexp Aexp**
- **SubA Aexp Aexp**
- **MultA Aexp Aexp**

Boolean Expressions (Bexp)

- **BoolLit Bool**
- **Aexp Aexp**
- **Equ2 Bexp Bexp**
- **EquB Bexp Bexp**.
- **LeB Bexp Bexp**
- **NegB Bexp**.
- **AndB Bexp Bexp**

Statements (Stm)

- **Assign String Aexp:** Represents variable assignment.
- **LoopW Bexp [Stm]:** Represents a while loop.
- **BranchIf Bexp [Stm] [Stm]:** Represents a conditional branching.

Program

A program is a sequence of statements (Program).

Compiler (compile)

- **compile:: [Stm] -> Code:** Takes a program and generates a stack-based code. It uses **compileStm** to compile individual statements.

Compilation of Statements (compileStm)

- **compileStm :: Stm -> Code:** Compiles individual statements to stack-based code.

Compilation of Arithmetic Expressions (compA)

- **compA :: Aexp -> Code:** Compiles arithmetic expressions to stack-based code.

Compilation of Boolean Expressions (compB)

- **compB :: Bexp -> Code:** Compiles boolean expressions to stack-based code.

Parser (parser)

The parser converts a string representation of a program into the internal representation.

- **parse :: String -> Program:** Takes a string and produces a parsed program.

Lexer (lexer)

- **lexer :: String -> [Token]:** Breaks the input string into a list of tokens.

Tokenization (Token)

- Token types include arithmetic operators, parentheses, integers, variables, and reserved words.

Parsing Strategy

The parsing strategy involves recursive descent parsing, breaking down the parsing process into functions that handle specific constructs.

Arithmetic Expression Parsing

- **parseIntOrParenExpr**: Parses integer literals, variables, or parenthesized expressions.
- **parseProdOrIntOrPar**: Parses products, integers, or parenthesized expressions.
- **parseSumOrProdOrSubOrIntOrPar**: Parses sums, differences, products, integers, or parenthesized expressions.
- **parseArithmeticExp**: Parses arithmetic expressions.

Boolean Expression Parsing

- **parseBoolOrParenExpr**: Parses boolean literals or parenthesized boolean expressions.
- **parseArithorBoolorParExpr**: Parses less than or equal to expressions, boolean literals, or parenthesized expressions.
- **parseIneorBoolorParExpr**: Parses equality expressions, less than or equal to expressions, boolean literals, or parenthesized expressions.
- **parseArithBoolExpr**: Parses equality expressions, less than or equal to expressions, boolean literals, or parenthesized expressions.
- **parseBexpByHalf**: Parses negated expressions, equality expressions, less than or equal to expressions, boolean literals, or parenthesized expressions.
- **parseBoolExpr**: Parses boolean equality expressions, negated expressions, equality expressions, less than or equal to expressions, boolean literals, or parenthesized expressions.
- **parseBexp**: Parses boolean AND expressions, boolean equality expressions, negated expressions, equality expressions, less than or equal to expressions, boolean literals, or parenthesized expressions.
- **parseBinaryExpr**: Parses binary expressions.

Statement Parsing

- **parseIntOrParenExpr**: Parses integer literals, variables, or parenthesized expressions into arithmetic expressions.
- **parseProdOrIntOrPar**: Parses products into arithmetic expressions.

- **parseSumOrProdOrSubOrIntOrPar**: Parses sums, differences into arithmetic expressions.
- **parseArithmeticExp**: Parses arithmetic expressions in Boolean expressions.
- **parseBoolOrParenExpr**: Parses bools or parenthesized boolean expressions into a Boolean expression.
- **parseArithorBoolorParExpr**: Parses Boolean expressions together with arithmetic expressions.
- **parseIneorBoolorParExpr**: Parses inequality expressions into a Boolean expression.
- **parseArithBoolExpr**: Parses equality expressions, less than or equal to expressions, boolean literals, or parenthesized expressions.
- **parseBexpByHalf**: Parses negated expressions, equality expressions, less than or equal to expressions, boolean literals, or parenthesized expressions.
- **parseBoolExpr**: Parses boolean equality expressions, negated expressions, equality expressions, less than or equal to expressions, boolean literals, or parenthesized expressions.
- **parseBexp**: Parses boolean AND expressions, boolean equality expressions, negated expressions, equality expressions, less than or equal to expressions, boolean literals, or parenthesized expressions.
- **parseBinaryExpr**: Parses binary expressions.
- **parseStms**: Parses a sequence of statements.
- **parseOpenPCase**: Parses a sequence of statements enclosed in parentheses.
- **parseAssign**: Parses assignment statements.
- **parseIf**: Parses if statements.
- **parseWhile**: Parses while statements.
- **parseStmsThen**: Parses statements in the then branch of an if statement.
- **parseStmsElse**: Parses statements in the else branch of an if statement.
- **parseOpenPCaseThen**: Parses a sequence of statements enclosed in parentheses in the then branch.
- **parseDefaultCaseThen**: Parses a single statement in the then branch.
- **parseOpenPCaseElse**: Parses a sequence of statements enclosed in parentheses in the else branch.
- **parseDefaultCaseElse**: Parses a single statement in the else branch.
- **parseStm**: Parses a statement based on the first token.

Main Parser

- **parser**: Main function that orchestrates the parsing process.
- **parseOrThrowError**: Parses a sequence of statements or throws an error.

Testing

The testParser function is provided to test the parser. It takes a program code as input and returns the final stack and state after running the compiled code.