Stack-Based Language Interpreter and Compiler

Trabalho Prático 2

Introduction

Summary of Project

This project involves the design and development of a stack-based interpreter and a corresponding compiler for a simple yet powerful domain-specific language (DSL). The primary objective was to create an educational tool that illustrates the fundamental concepts of computational theory, particularly how high-level programming constructs are translated into executable instructions that a machine can understand.

The interpreter, constructed in Part 1 of the project, is capable of executing a series of low-level bytecode instructions. These instructions perform various operations such as arithmetic calculations, boolean evaluations, variable management, and control flow handling. The core of the interpreter is its stack-based execution model, which operates by manipulating a stack to evaluate expressions, store temporary results, and control the program flow.

In Part 2, the compiler front-end translates high-level statements and expressions from our DSL into the bytecode that the interpreter can execute. This process involves tokenizing the high-level language constructs into meaningful symbols (lexing) and then organizing these tokens into a structured representation that reflects the user's intentions (parsing).

To showcase the capabilities of our system, a factorial computation program was developed. This example program takes a high-level factorial algorithm and compiles it into bytecode. When executed by the interpreter, it accurately computes the factorial of a given number, demonstrating the system's ability to handle iterative processes and conditionals.

Part 1: Interpreter

Architectural Design

The interpreter is designed around a stack-based architecture, a common choice for simple and efficient execution models. This design involves the following components:

- Stack: A dynamic data structure used to store operands and intermediate results during execution.
- State: A mapping from variable names to their values (either integers or booleans), representing the current state of the program.
- Instruction Set: A collection of operations that the interpreter can perform, such as arithmetic operations, boolean logic, and control flow commands.

The choice of a stack-based model allows for straightforward implementation of expression evaluation and supports nested and complex expressions without the need for explicit register management. Here are the core components of the interpreter's architecture:

```
-- StackElement data type
data StackElement = IntVal Integer | BoolVal Bool deriving Show

-- Stack type
type Stack = [StackElement]

-- State type
type State = [(String, StackElement)]
```

Instruction Set Overview

The interpreter supports a range of instructions, each corresponding to a fundamental operation in the language:

- Arithmetic Instructions: Push, Add, Mult, Sub for manipulating integer values.
- Boolean Instructions: Tru, Fals, Equ, Le, And, Neg for boolean operations.
- Control Flow Instructions: Branch, Loop for implementing conditional execution and loops.
- Memory Access Instructions: Fetch, Store for variable access and assignment.
- Miscellaneous: Noop as a no-operation instruction.

Each instruction directly correlates to the high-level constructs in the DSL, ensuring a seamless translation from source code to executable bytecode.

```
-- Definition of Inst and Code
Data Inst =
  Push Integer| Add | Mult | Sub | Tru | Fals | Equ | Le | And | Neg | Fetch
String | Store String | Noop |
  Branch Code Code| Loop Code Code
  deriving Show
type Code = [Inst]
```

Here's a brief explanation of what each instruction in Inst data type does:

- Push Integer: Pushes an integer value onto the stack. Used for loading numeric literals.
- Add: Pops the top two integers from the stack, adds them, and pushes the result back onto the stack. Used for arithmetic addition.
- Mult: Similar to Add, but performs multiplication on the top two integers on the stack.
- Sub: Subtracts the top integer from the next top integer on the stack and pushes the result.
- Tru: Pushes a boolean ‘True’ value onto the stack.
- Fals: Pushes a boolean ‘False’ value onto the stack.
- Equ: Checks if the top two values on the stack are equal. Pushes ‘True’ if they are, ‘False’ otherwise.

- Le: Checks if the second value on the stack is less than or equal to the top value. Pushes 'True' if so, 'False' otherwise.
- And: Performs a logical AND on the top two boolean values on the stack.
- Neg: Negates the top boolean value on the stack.
- Fetch String: Fetches the value of the variable (identified by the string) from the state and pushes it onto the stack.
- Store String: Pops the top value from the stack and stores it in the state with the variable name provided by the string.
- Noop: Represents a no-operation instruction. It does nothing and is usually used for syntactical purposes.
- Branch Code Code: Acts as an if-else construct. It pops the top value from the stack. If 'True', it executes the first block of code; if 'False', the second.
- Loop Code Code: Used for loops. The first code block is a condition; if it evaluates to 'True', the second block (the loop body) is executed, and the loop continues.
- Instruction Set Overview

The execution engine of the interpreter is primarily centered around the run function, which is responsible for interpreting and executing the bytecode represented by the Code type. The mechanics of this engine involve managing and manipulating the Stack and State according to the given instructions.

Here's a breakdown of the execution engine's mechanics:

- Sequential Execution:
- The 'run' function processes the list of instructions ('Code') sequentially, one instruction at a time.
- Each instruction leads to some operation on the 'Stack', the 'State', or both.

```
run :: (Code, Stack, State) -> (Code, Stack, State)
```

Stack Manipulation:

- The stack is a central feature in this execution model. It's where values are stored temporarily for computation.
- Arithmetic instructions ('Add', 'Sub', 'Mult') pop operands from the stack, perform the operation, and push the result back.

```
run (Add:code, IntVal x:IntVal y:stack, state) = run (code, IntVal (x + y):stack,
state)
```

State Management:

- The ‘State’ maintains variable bindings (name-value pairs).
- Instructions like ‘Fetch’ and ‘Store’ are used for reading from and writing to the ‘State’.

```
run (Store var:code, val:stack, state) = run (code, stack, store var val state)
```

Control Flow:

- Instructions ‘Branch’ and ‘Loop’ handle the control flow based on conditions.
- They alter the normal sequential flow of execution by deciding which set of instructions to execute next.

```
run ((Branch trueBranch falseBranch):code, BoolVal x:stack, state) =
  if x
  then run (trueBranch ++ code, stack, state)
  else run (falseBranch ++ code, stack, state)
```

Error Handling:

- The engine includes mechanisms to handle errors, such as type mismatches or invalid operations.
- When an error is encountered, the ‘run’ function halts execution and typically signals an error state.

```
run (inst:_, _, _) = error ("Unrecognized instruction or invalid types: " ++ show
inst)
```

Execution Termination:

- The process continues until there are no more instructions to execute, at which point the ‘run’ function returns the final state of the stack and the program state.

Error Handling and Runtime Exceptions

Error handling and management of runtime exceptions are critical components of the interpreter's robustness and reliability. In the context of the stack-based interpreter, several mechanisms are employed to ensure that errors are detected, reported, and handled appropriately.

Here are the main errors to be known:

Handling Undefined Variables:

- When the ‘Fetch’ instruction is used, the interpreter checks if the requested variable exists in the state. If the variable is not found, it raises an error.

```
fetch var state = case lookup var state of
  Just val -> val
  Nothing  -> error "Run-time error"
```

- Type Safety Checks for Logical Operations:
- The 'And' operation requires that both operands on the stack are boolean values ('BoolVal'). If one or both operands are not boolean, the interpreter throws a runtime error.

```
run (And:code, x:y:stack, state) = case (x, y) of
    (BoolVal xb, BoolVal yb) -> run (code, BoolVal (xb && yb):stack, state)
    _ -> error "Run-time error"
```

Runtime Exceptions for Invalid Instructions:

- If the interpreter encounters an unrecognized instruction, it terminates execution and raises an error. This error handling is crucial for catching any malformed or corrupt instructions.

```
run (inst:_, _, _) = error ("Unrecognized instruction or invalid types: " ++ show
inst)
```

Part 2: Compiler and Parser

For this part of the project, we were tasked with running lines of imperative coding, but first we needed to transform them into assembly code so that our `run` function could understand it. We decided to do the following strategy: transform the given string into tokens, then, `parse` the tokens to organize them into statements and then `compile` the statements into the assembly code that our `run` function can understand and process.

Language Definitions

- **Aexp**: Defines arithmetic expressions in the language.
- **Bexp**: Defines boolean expressions in the language.
- **Stm**: Defines statements in the language. It includes arithmetic expressions, boolean expressions, assignment statements, skip statements, if statements, and while statements.
- **Program**: Defines a program as a list of statements.

Compilation Functions

- **compA**: Compiles arithmetic expressions into stack-based intermediate code (`Code`).
- **compB**: Compiles boolean expressions into stack-based intermediate code (`Code`).
- **compile**: Compiles a program into stack-based intermediate code (`Code`).

Parser Functions

- **parse**: Parses a string representation of a program into the abstract syntax.
- **parseStms**: Parses a list of tokens into a program and the remaining tokens.
- **parseStm**: Parses a list of tokens into a single statement and the remaining tokens.
- **parseInt**: Parses a list of tokens into an integer expression and the remaining tokens.

- **parseSubOrSumOrProdOrIntOrPar**: Parses a list of tokens into an arithmetic expression involving subtraction, addition, multiplication, integers, or parentheses.
- **parseSumOrProdOrIntOrPar**: Parses a list of tokens into an arithmetic expression involving addition, multiplication, integers, or parentheses.
- **parseProdOrIntOrPar**: Parses a list of tokens into an arithmetic expression involving multiplication, integers, or parentheses.
- **parseVar**: Parses a list of tokens into a variable expression.
- **parseIntOrPar**: Parses a list of tokens into an integer expression or a parenthesized expression.
- **parseLeqOrEqOrTrueOrFalseOrParOrAexp**: Parses a list of tokens into a boolean expression involving less than or equal, equal, true, false, or parentheses.
- **parseEqBoolOrAnd**: Parses a list of tokens into a boolean expression involving equal, boolean and, or parentheses.
- **parseEqBoolOrNot**: Parses a list of tokens into a boolean expression involving equal, boolean not, or parentheses.
- **parseNotOrLeqOrEq**: Parses a list of tokens into a boolean expression involving boolean not, less than or equal, equal, true, false, or parentheses.

Lexer

- **myLexer**: Tokenizes a string into a list of tokens for the parser.

Token Definition

- **Token**: Defines the different types of tokens in the language.

Tests

To verify the correctness of our code, unit tests have been done, testing simple operations (while, if-then-else, etc), as well as some possible errors due to parenthesis.

Here are some unit tests added:

```
-- testParser "x := 10; while (not(x == 0)) do (x := x - 1; y := x * 2;);" ==
("","x=0,y=0")

-- testParser "if (not(not True)) then x := 1; else y := 2;" == ("","x=1")

-- testParser "x := 10; while (not(x == 0)) do (x := x - 1; y := x * 2; if (y ==
0) then z := 1; else z := 0;); a := b;"

-- testParser "x := 10; while (not(x == 0)) do (x := x - 1; y := x * 2; if (y ==
0) then z := 1; else z := 0;); a := b / 2;"

-- testParser "x := 10; while (not(x == 0)) do (x := x - 1; y := x * 2; if (y ==
0) then z := 1; else z := 0;); a := b * 2;"
```

Conclusion

Achievements of the Project

- **Functional Interpreter and Compiler**: Successfully developed a stack-based interpreter and a compiler for a simple, yet expressive, programming language. This language supports basic arithmetic, boolean logic, and control structures like loops and conditionals.
- **Robust Error Handling**: Implemented comprehensive error handling mechanisms, ensuring the interpreter gracefully manages and reports common errors like type mismatches, undefined variables, and stack underflows.
- **Educational Tool**: Created a tool that can be used for educational purposes, offering insights into the basics of programming, language interpretation, and compiler design.
- **Simple State Management**:Effectively managed program state, demonstrating the fundamental concepts of variable storage and retrieval.
- **Clear Implementation**: Maintained a clear and understandable code structure, making the project accessible for educational use and further development.

Educational Impact

- **Understanding Core Programming Concepts**: The project serves as an excellent resource for understanding fundamental programming concepts such as variables, control flow, and basic data types.
- **Introduction to Compiler and Interpreter Design**: Provides a practical introduction to the basics of compiler and interpreter design, demystifying how high-level code is translated into executable instructions.
- **Debugging and Error Handling**: The project's focus on error handling equips learners with the skills to identify and rectify common programming errors, an essential skill in software development.
- **Inspiration for Further Learning**: By exposing learners to the inner workings of an interpreter, the project sparks curiosity and provides a foundation for more advanced studies in computer science and programming language theory.
- **Hands-On Experience**:Offers hands-on experience in developing a fundamental piece of software, encouraging practical learning over theoretical study.

In summary, the project successfully meets its objectives by providing a functional and educational tool that lays the groundwork for deeper exploration into programming and software development. It stands as a testament to the practical application of theoretical concepts in computer science, offering a valuable learning experience.

Programação Funcional e em Lógica

Turma 11 - Grupo 11

- Rodrigo dos Santos Arteiro Rodrigues (up202108749@fe.up.pt) -> 50%
- Pedro Guilherme Pinto Magalhães (up202108756@fe.up.pt) -> 50%