Haskell Interpreter and Compiler

Group: T12_G04

Luís Filipe da Silva Jesus - up202108683 - 50%
Miguel Diogo Andrade Rocha - up202108720 - 50%

Project Description

•This Haskell project is a comprehensive simulation of a programming language processor. It encompasses an `assembler` for executing low-level instructions, and a `compiler` system, complete with a `lexer` and `parser`, to translate and interpret high-level program constructs. The project effectively demonstrates the complete cycle of code processing, from high-level syntax to executable machine instructions.

Part 1: Interpreter

•The interpreter is built on a `stack-based` architecture, a prevalent model known for its simplicity and efficiency in executing instructions.
•This architecture is characterized by the following key components:

Instruction Set - (Code):

•This defines the set of operations that the interpreter can perform. The instruction set includes arithmetic operations (like addition, subtraction and multiplication), boolean logic (like AND operations), and control flow commands (such as loops and conditional branches). Each instruction is designed to manipulate the stack and state in a way that reflects the high-level intention of the code.

Stack:

•The central data structure in this architecture. It's used dynamically during execution to store operands and intermediate results. This approach simplifies the evaluation of expressions, especially nested or complex ones.

•We define a custom data type to represent the stack, which is a list of `StackDataType` values. This data type can hold either an `IntValue` or a BoolValue. This allows us to store both integer and boolean values in the stack.

```
data StackDataType = IntValue Integer | BoolValue Bool deriving (Show, Eq)

type Stack = [StackDataType]
```

State:

- It represents the current context of the program being interpreted. This includes a mapping from variable names to their corresponding values. The state is crucial for maintaining and updating the values of variables as the program executes.
- We define a custom data type to represent the state, which is a list of (String, StackDataType) tuples. This data type can hold a variable name and its corresponding value.

```
type State = [(String, StackDataType)]
```

Executing Instructions:

- The core of the interpreter's execution is encapsulated within the run function. This essential component is responsible for interpreting and executing the instructions, represented by the Code type.
- The run function operates by sequentially processing the list of instructions contained within Code, one instruction at a time.

Stack and State Manipulation:

- For each instruction, we created auxiliary functions which manipulate the Stack and State as per the instruction's requirements.
- For example, for the add instruction, we created:

```
add :: Stack -> Stack
add ((IntValue elem1):(IntValue elem2):stack) = (IntValue  (elem1 + elem2)):stack
add _ = error "Run-time error"
```

  - This function adds the top two IntValues on the stack, replacing them with their sum.
- For the store instruction, we created:

```
storeElem :: String -> Stack -> State -> State
storeElem varName ((IntValue elem):stack) state = (varName, (IntValue elem)) : filter ((/=
varName) . fst) state
storeElem varName ((BoolValue elem):stack) state = (varName, (BoolValue elem)) : filter
((/= varName) . fst) state
storeElem _ [] state = error "Run-time error"
```

- This function stores the top element of the stack in the state, with the given variable name, updating the state accordingly.

Control Flow Instructions:

- The instructions `branch` and `loop` are used to control the flow of execution. These instructions are used to implement conditional statements and loops, respectively.
- For example, for the `branch` instruction, we created:

```
branch :: Code -> Code -> Stack -> Code
branch code1 code2 ((BoolValue elem):stack)
| elem = code1
| otherwise = code2
branch _ _ _ = error "Run-time error"
```

  - Combined with the run function, this function evaluates the top element of the stack, and executes the first code block if it's `True`, or the second code block if it's `False`.

Error Handling:

- The interpreter is designed to handle errors that may occur during execution, such as `wrong arguments` for instructions.

- These errors are detected by the interpreter, and are reported to the user.

- In these cases, an exception with the string `"Run-time error"` is thrown, and the program is terminated.

For example, for the `mult` instruction, we created:

```
mult :: Stack -> Stack
mult ((IntValue elem1):(IntValue elem2):stack) = (IntValue (elem1 * elem2)):stack
mult _ = error "Run-time error"
```

We throw a Run-time error if the top two elements of the stack are not `IntValues`, since the `mult` instruction is only defined for integers. This is both checking the type of the elements and the number of elements in the stack.

Part 2: Compiler

- In this part of the project, the objective was to execute lines of high-level code. To achieve this, we needed to convert the given string into `instructions` (code), which the `run` function, could understand and process.
- In order to that, we divided this part into three main components: `lexer`, `parser` and `compiler`.

Lexer:

•The lexer is responsible for tokenizing the input string, and converting it into a list of tokens. Each token represents a single element of the input string, such as a keyword, an operator, a variable name or even integers.

•We defined a custom data type to represent the tokens:

```
data Token
= NumToken Integer
| VarToken String
| AddToken
| SubToken
| MultToken
| TrueToken
| FalseToken
| AndToken
| NegToken
| WhileToken
| DoToken
| IfToken
| ThenToken
| ElseToken
| AssignToken
| EquNumToken
| EquBoolToken
| LeEquToken
| SemicolonToken
| LeftParToken
| RightParToken
deriving (Show)
```

•The lexer is implemented as a recursive function, which processes the input string character by character, and converts it into a list of tokens:

```
myLexer :: String -> [Token]
myLexer [] = []
myLexer (char:rest)
| isDigit char = numLexer (char:rest)
| isAlpha char = stringLexer (char:rest)
| isSpace char = myLexer rest
| otherwise = symbolLexer (char:rest)
```

•In order to identify integers, keywords, variable names, and symbols, we created helper functions like numLexer, stringLexer and symbolLexer.

- These functions are responsible for identifying the tokens, and recursively calling the myLexer function to process the rest of the input string.
- We used the functions isDigit, isAlpha and isSpace from the Data.Char module to identify the type of each character.

Consider this helper function for the lexer, responsible for identifying integers:

```
numLexer :: String -> [Token]
numLexer str = NumToken (read num) : myLexer rest
    where (num, rest) = span isDigit str
```

Using the span function, we can identify the longest prefix of the input string that satisfies the given predicate. In this case, we use the isDigit function to identify the longest prefix of digits, which corresponds to the integer value. The rest of the string is then processed recursively by the myLexer function.

Data Types Definition:

- We defined three datas in Haskell to represent expressions and statements of this imperative language:
    - Aexp for arithmetic expressions:
      `data Aexp = Num Integer | Var String| AddAexp Aexp Aexp | SubAexp Aexp Aexp | MultAexp Aexp Aexp deriving (Show, Eq)`
    - Bexp for boolean expressions:
      `data Bexp = BoolBexp Bool | NegBexp Bexp | EquNumBexp Bexp Bexp | EquBoolBexp Bexp Bexp | LeNumBexp Bexp Bexp | AndBexp Bexp Bexp | AexpBexp Aexp deriving (Show,Eq)`
    - Stm for statements:
      `data Stm = AssignStm String Aexp | IfStm Bexp [Stm] [Stm] | WhileStm Bexp [Stm] deriving Show`

Compiler:

- The compiler is responsible for converting the list of statements into a list of instructions (code), which can then be executed by the run function.
- The compiler is implemented as a recursive function, which processes the list of statements one by one, and converts them into a list of instructions (Code).
- To do so, it calls the compA and compB functions, which are responsible for converting arithmetic and boolean expressions into instructions, respectively.

Parser:

•The parser is responsible for converting the program string into a list of statements, which can then be compiled and executed. It does so by using the tokens generated by the lexer. It is implemented as a recursive function, which processes the list of tokens one by one, and converts them into a list of statements.

•The language includes arithmetic expressions, boolean expressions, and statements such as assignments, if statements, and while loops. The parser is designed to take a list of tokens generated by a lexer and produce a corresponding abstract syntax tree (AST) that represents the program's structure. The code follows a top-down parsing approach, breaking down the parsing process into functions that handle specific aspects of the language's syntax.

Consider the functions of the parser:

•parseAexpOrParen: Parses arithmetic expressions or expressions within parentheses, handling numeric literals, variable names, and nested expressions.
•parseMultOrAexpOrParen Parses multiplication operations in arithmetic expressions, recursively handling nested expressions.
•parseOperationsOrParen: Top-level function for parsing arithmetic expressions, including addition and subtraction operations, by calling the previous function and itself for nested expressions.
•parseBexpOrParen: Parses boolean expressions or expressions within parentheses, handling boolean literals and nested boolean expressions.
•parseAexpOrBexpOrParen Parses both boolean and arithmetic expressions, recursively calling the relevant functions for nested expressions.
•parseLeOrBexp: Parses less than or equal operations in boolean expressions, recursively handling nested expressions.
•parseLeOrEquOrBexp: Parses integer equal operations in boolean expressions, recursively handling nested expressions.
•parseLeOrEquOrNegOrBexp: Parses negation operations in boolean expressions, recursively handling nested expressions.
•parseLeOrEquOrNegOrBoolEquOrBexp: Parses boolean equal operations in boolean expressions, recursively handling nested expressions.
•parseBoolOperations: Top-level function for parsing boolean expressions, including AND operations, by calling the previous function and itself for nested expressions.
•parseSingleStm: Parses a single statement, handling variable names, assignment operations, and expressions within parentheses.
•parseStatement: Parses a sequence of statements, including assignments, if statements, and while loops. Handles the different types of statements by calling the relevant parsing functions.

If and While Statement Parsing Sections:

- parseStatement (IfToken : rest): Parses if statements, including both 'then' and 'else' blocks, handling nested statements.
- parseStatement (WhileToken : rest): Parses while statements, handling nested statements.

It is important to note that we call parser in the following way:

```
parse = buildData . myLexer
```

BuildData is a function that builds the AST from the list of tokens generated by the lexer. it calls the parseStatement function, which is the top-level function for parsing statements, and returns the AST. That means - as can be deduced by the functions explained above - that the parser is able to parse arithmetic expressions, boolean expressions, and statements such as assignments, if statements, and while loops since the parseStatement function calls the relevant parsing functions for each of these cases and this process is recursive.

```
buildData :: [Token] -> Program
buildData tokens =
   case parseStatement tokens of
      Just (stms, []) -> stms
      _ -> error "Error parsing program"
```

Tests

- We tested our code using the provided test cases and it passed all of them.

- We also created some random test cases.

Conclusion

- This project was a great opportunity to learn about the Haskell programming language, and to apply the concepts learned in class to a real-world project.

- We were able to implement a functional interpreter, compiler, lexer and parser, which can process and execute high-level code. This allowed us to gain a deeper understanding of the concepts involved in the compilation process, and the challenges associated with it.

•We also learned about the importance of code organization and modularity, and how it can be used to improve the readability and maintainability of the code.

•Overall, this project was a great learning experience even though we considered it very challenging and time-consuming.