Second PFL Project - Haskell

T10_G04

- Catarina Isabel Moreira Canelas up202103628 (50%)
- Inês Martin Soares up202108852 (50%)

Project Description

This project involves creating a virtual machine and a compiler for a simple imperative programming language. The machine operates with a set of instructions and has a stack-based evaluation system. The imperative language includes arithmetic and boolean expressions, assignments, sequences of statements, if-then-else statements, and while loops.

In summary, the program we developed receives an expression, converts it into a list of tokens with the lexer function, this list is then converted into a list of sublists where the list of tokens is divided with every token ";". The idea behind the list of lists was to help when parsing the list. After parsing, there would be a list of statements ready to be compiled. After compiled into a list of instructions, it can be run with the run function and get the final result.

Part 1: Virtual Machine

(a) Define Types

- Stack: Create a type to represent the machine's stack.
- State: Define a type to represent the machine's state.

(b) Implement Functions

- createEmptyStack: Create an empty machine's stack.
- createEmptyState: Create an empty machine's state.
- stack2Str: Convert a stack to a string.
- state2Str: Convert a machine state to a string.
- run: Implement an interpreter for programs, given a list of instructions, stack, and initial storage.

Part 2: Compiler

(a) Define Types

- Aexp: Represent arithmetic expressions. (Casobase is for integer values and Casobase2 is for variables, Multiplication, Sum and Difference operations)
- Bexp: Represent boolean expressions. (True/False values, Equal between boolean values and another Equal for Integer values, And operation, Not operation and Less or Equal operation between Integer values)
- Stm: Represent statements. (Assignments, conditional If with a boolean condition and "then" and "else" statements, Loop and sequences of statements)

(b) Implement Compiler

- compile: Compiler function that translates a program in the imperative language into a list of machine instructions.
- compA: Compile arithmetic expressions.
- compB: Compile boolean expressions.

(c) Implement Parser

- parse: Create a parser that transforms an imperative program (given as a string) into its representation in the Stm data structure.

Code Description

Data Types

- **Inst**: Represents instructions for the virtual machine.
- **Code**: A list of instructions.
- **Stack**: A list of integers representing the evaluation stack.
- **State**: A list of pairs representing the storage state where variables are associated with integer values.
- **Aexp**: Represents arithmetic expressions.
- **Bexp**: Represents boolean expressions.
- **Stm**: Represents statements.

Helper Functions

- `createEmptyStack`: Returns an empty stack.
- `stack2Str`: Converts a stack to a string for display purposes.
- `createEmptyState`: Returns an empty storage state.
- `state2Str`: Converts a storage state to a string for display purposes.
- `divide`: Converts a list of tokens (ex: output of lexer function) into sublists divided by the ";" token. It was implemented to help with the parse function.
- `compA`: Compiles arithmetic expressions into its respective instructions.
- `compB`: Compiles boolean expressions into its respective instructions.

Interpreter Function: run

- `run`: Takes a tuple `(Code, Stack, State)` and executes the virtual machine instructions until the code is empty. It returns a tuple containing the remaining code, the resulting stack, and the updated state.

Interpreter Execution

- The `run` function pattern matches on different instructions and executes them accordingly.
- Arithmetic and boolean operations manipulate the stack.
- `Fetch` retrieves the value of a variable from the state and pushes it onto the stack.
- `Store` updates the value of a variable in the state.
- `Noop` is a dummy instruction that does nothing.
- `Branch` and `Loop` control flow instructions determine the next set of instructions based on the top of the stack.

Error Handling

- The interpreter includes basic error handling. For example, attempting to fetch a variable not in the state results in a run-time error.

Example of instruction (part 1)

```
testAssembler [Push 3, Push 4, Add, Store "result", Noop]
```

- Pushes 3 and 4 onto the stack, adds them, stores the result in the state.

Lexer (part 2)

The lexer function is a lexical analyzer that converts a string into a list of tokens. Tokens include arithmetic operators, parentheses, semicolons, assignment operators, comparison operators, keywords (if, then, else, not, and, while, do), boolean values (true, false), integers, and variables.

Examples

Ex.1:

```
y := 1; while not (x = 1) do (y := y * x; x := x - 1)
```

- This is lexed into a list of tokens l1.

LexerL1

Ex.2:

```
x := 2; y := (x - 3)*(4 + 2*3); z := x + x*(2);
```

- This is lexed into a list of tokens l2.

LexerL2

Examples of Divide Function (Helper function)

Ex.1:

```
list = ["y", ":=", "1", ";", "x", ":=", "1", ";", "if", "True", "then", "x", ":=",
"2", "else", "y", ":=", "3"]
```

- This is converted into a list of sublists divided by the ";" token.

DivideList

Ex.2:

```
l1 = lexer "y := 1; while not (x = 1) do (y := y * x; x := x - 1)"
l2 = lexer "x := 2; y := (x - 3)*(4 + 2*3); z := x +x*(2);"
```

- After being run by the lexer function, with the divide function these two lists are converted into lists of sublists.

DivideExampleL1L2

Parser (Not Implemented)

The provided lexer can be used as a foundation for implementing a parser. A parser would be responsible for converting the list of tokens into a structured representation of the program. This representation could be an abstract syntax tree (AST) that corresponds to the syntax and structure of the imperative programming language.

This function was supposed to received an already lexed and divided expression, so it could ouput into a list of statements, like the next example:

```
-- example: y := 1; while ¬(x = 1) do (y := y ∗ x; x := x − 1)

lLex = lexer "y := 1; while not(x = 1) do (y := y * x; x := x - 1)"

lDivided = divide lLex
```

part1Parser

And then, after parsing the expression, it would output the next list of statements:

```
lParsed = [Assignment "y" (CasoBase 1),Loop2 (Not2 (Igual2 (CasoBase2 "x")
(CasoBase 1))) [Assignment "y" (Multiplicacao (CasoBase2 "y") (CasoBase2 "x")),
Assignment "x" (Diferenca (CasoBase2 "x") (CasoBase 1))]]
```

Compiler

The compile function takes a program (represented as a list of statements) and translates it into machine instructions. It uses helper functions compA and compB for compiling arithmetic and boolean expressions, respectively.

Example Compilation

Example of compA

```
```

```
-- 2 + 3 * 4
tree = Soma (CasoBase 2) (Multiplicacao (CasoBase 3) (CasoBase 4))

compiledA = compA tree
```

- When receiving an already parsed expression, in this case an arithmetic expression, compA compiles it into a list of instructions

CompA

```
testA = testAssembler compiled
```

- Then, when running the output received from compA, we can see that it returns a correct result

CompAResult

Example of compB

```
-- True = not True
expressaoB = Igual1 (Folha True) (Not2 (Folha True))

compiledB = compB expressaoB
```

- When receiving an already parsed expression, in this case a boolean expression, compA compiles it into a list of instructions

CompB

```
testB = testAssembler compiledB
```

- Then, when running the output received from compB, we can see that it returns a correct result

CompBResult

Example of Compiler

```
-- y := 1; while ¬(x = 1) do (y := y ∗ x; x := x − 1)

l = [Assignment "y" (CasoBase 1),Loop2 (Not2 (Igual2 (CasoBase2 "x") (CasoBase
1))) [Assignment "y" (Multiplicacao (CasoBase2 "y") (CasoBase2 "x")), Assignment
"x" (Diferenca (CasoBase2 "x") (CasoBase 1))]]

exemploCompile = compile l
```

- After receiving the already parsed expression l, the Compiler compiles the example program (exemploCompile) into a list of machine instructions.

Compiler

- Now, this list of intructions could be run and then it would be obtained the final result

Conclusion

- The project demonstrates the implementation of a simple programming language interpreter and compiler, providing insights into lexing, parsing, and execution of imperative programs.
- While the current implementation is limited in its scope, it serves as a foundational structure that can be extended to support additional language features, optimizations, and error handling.
- Further development could involve enhancing the parser to construct an abstract syntax tree (AST), enabling more sophisticated analyses and optimizations.
- In summary, this project serves as a practical exploration of language processing concepts and provides a starting point for more advanced language development and compiler construction. The codebase can be expanded and refined to accommodate richer language constructs and optimizations, offering a valuable learning experience in the realm of language design and implementation.