Transpiler (source-to-source compiler)

License GPLv3 UPorto FEUP

`A source-to-source compiler and low-level assembler written in Haskell.`

Index

Identification of the topic and group

- **Short description:** A source-to-source compiler and low-level assembler written in Haskell.
- **Group:** T08_G01
- **Group members:**
  - José Miguel Moreira Isidro (up202006485@fe.up.pt)
  - José António Santos Costa (up202004823@fe.up.pt)
- **Contribuition:**

- José Miguel Moreira Isidro: 50%
- José António Santos Costa: 50%

Installation and Execution

Installation

Requisites

To run this game you need a running Haskell environment, preferably GHCi.

Execution

1. Open `ghci`, GHC's interactive environment;
2. Load `main.hs`, located in the `src` folder;
3. Call the `testAssembler` with a list of code instructions to run them. There are a few examples in `tests.hs`.
4. Call the `testParser` with a the code string to parse and run. There are a few examples in `tests.hs`.
5. To run all the predefined tests, call the `main` function with no arguments.

Assembler (First Part)

The first part of the project was a sucess, with the `run` function succeeding in all given tests.

Data Types

Variable and Value

- The Variable type represents the variable names to be stored in State (storage);
- The Value data can be either an Integer value or a Boolean. This way, we can have this data represent all values for both the Stack and States. Moreover, we opted for the representation so we can easily operate and compare values without needing to convert them to other types.

```haskell
-- Data type for variables
type Variable = String
-- Data type for values
data Value
  = MyInt Integer
  | MyBool Bool
  deriving (Show)
```

Stack

- The Stack is a list of Values.

```haskell
-- Data type for the machine stack
type Stack = [Value]
```

State

- The State is a list of Variable-Value pairs.

```haskell
-- Data type for the machine state (storage)
type State = [(Variable, Value)]
```

Functions

Stack Functions

These **Stack** functions are very straightforward. After convertin the stack to a string, it represents the stack as an ordered list of values, separated by commas and without spaces, with the leftmost value representing the top of the stack.

```haskell
createEmptyStack :: Stack
createEmptyStack = []

stack2Str :: Stack -> String
stack2Str []     = ""
stack2Str [x]    = showVal x
stack2Str (x:xs) = showVal x ++ "," ++ stack2Str xs
```

Below is the definition of the `showVal` function, used to convert a **Value** data to string. This function is also used the `state2Str` function.

```haskell
showVal :: Value -> String
showVal (MyInt intVal) = show intVal
showVal (MyBool True)  = "True"
showVal (MyBool False) = "False"
```

In addition to the above, we have also implemented some auxiliary functions to check and update the stack, such as `push`, `pop`, `top` and `isEmpty`.

State Functions

The **State** functions also straightforward, however a little more complex. After converting the state to a string, it represents the state as an list of pairs variable-value, separated by commas and without spaces, with the pairs ordered in alphabetical order of the variable name. Each variable-value pair is represented without spaces and using an ”=”.

```
createEmptyState :: State
createEmptyState = []

state2Str :: State -> String
state2Str state = intercalate "," [var ++ "=" ++ showVal val | (var, val)
<- sortedState]
  where
    sortedState = sortBy (\(var, _) (val, _) -> compare var val) state
```

Code Interpreter

The `run` function is the interpreter for a simple stack-based virtual machine, which executes machine instructions represented by the `Inst` data type. The machine operates on a stack (`Stack`) and maintains a state (`State`) to store variable-value pairs. The function takes a triple `(Code, Stack, State)` as its argument, where `Code` is a list of machine instructions, `Stack` is the current state of the stack, and `State` is the storage for variables.

The `run` function iteratively processes machine instructions until the code is empty. Using `case (...) of`, it pattern matches on each instruction and performs the corresponding operation. It supports operations like arithmetic operations (`Push`, `Add`, `Mult`, `Sub`), boolean operations (`Tru`, `Fals`, `Equ`, `Le`, `And`, `Neg`), variable manipulation (`Fetch`, `Store`), control flow (`Branch`, `Loop`), and a no-operation (`Noop`). The result of each operation updates the stack and state accordingly.

The function handles errors such as attempts to perform operations on an empty stack or unexpected operand types, raising run-time errors when necessary. The `run` function returns the final state of the stack, allowing users to inspect the resulting values after executing a sequence of machine instructions.

Furthermore, we improved on the previous implementation of the `calc` (from TP classes), so it handles most of the operations performed by the machine. The code for these functions is available in `assembler.hs`.

Compiler and Parser (Second Part)

Data Types

These data structures form the foundation for representing arithmetic expressions, boolean expressions, and statements in a compiler. They are used as translation tools between the tokenized input string and the machine code.

Arithmetic Expressions

The Aexp data type is used to model expressions that can contain variables, numeric constants, and the arithmetic operations of addition, subtraction, and multiplication.

```
-- Data type for arithmetic expressions
data Aexp
  = Var String                    -- Variable
  | Num Integer                   -- Numeric constant
```

```
    | AddA Aexp Aexp                -- Addition
    | SubA Aexp Aexp                -- Subtraction
    | MultA Aexp Aexp               -- Multiplication
    deriving Show
```

Boolean Expressions

The Bexp data type is used to model expressions that can contain boolean values (TrueB and FalseB), boolean operations (like AndB for logical AND and NegB for negation), and comparison operations (like IntEqual for integer equality, BoolEqual for boolean equality, and LessEqual for less than or equal to comparison).

```
-- Data type for boolean expressions
data Bexp
    = TrueB
    | FalseB
    | AndB Bexp Bexp                -- And operation
    | IntEqual Aexp Aexp            -- Integer Equality comparison
    | BoolEqual Bexp Bexp           -- Boolean Equality comparison
    | LessEqual Aexp Aexp           -- Less than or equal comparison
    | NegB Bexp                     -- Negation
    deriving Show
```

Statements

The Stm data type is used to model statements that can be an assignment (where a variable is assigned the result of an arithmetic expression), an if-then-else control flow (where based on a boolean expression, one of two lists of statements is executed), or a while loop (where a list of statements is repeatedly executed as long as a boolean expression evaluates to true).

```
-- Data type for statements
data Stm
    = Assign String Aexp            -- var := Aexp
    | If Bexp [Stm] [Stm]           -- if Bexp then Stm1 else Stm2
    | While Bexp [Stm]              -- while Bexp do Stm
    | NoopStm                       -- No operation
    | Aexp Aexp                     -- Arithmetic expression
    | Bexp Bexp                     -- Boolean expression
    deriving Show
```

Program

The Program type is a list of such statements, representing a sequence of operations or commands in a program.

```
type Program = [Stm]
```

Compiler

The main compiler function `compile :: [Stm] -> Code` compiles a list of statements (`Stm`) into machine code (`Code`). It relies on pattern matching to handle the various types of statements, including assignments, if-then-else statements, while loops, and standalone arithmetic or boolean expressions. This function depends on the auxiliar functions `compA :: Aexp -> Code` and `compB :: Bexp -> Code`, that with similar behaviour, are responsible for arithmetic and boolean expressions respectively.

```
compile :: Program  -> Code
compile [] = []
compile (stm:rest) = case stm of
  Assign var aexp -> compA aexp ++ [Store var] ++ compile rest
  If bexp aexp1 aexp2 -> compB bexp ++ [Branch (compile aexp1) (compile
aexp2)] ++ compile rest
  While bexp aexp -> Loop (compB bexp) (compile aexp) : compile rest
```

In the code above the function deconstructs a matching statement into the appropriate stack ordered machine code. Furthermore, it recursively calls on itself when a statement is comprised of other statements, and calls the boolean and arithmetic compiler when needed.

Parser

The parsing is done through a series of functions that build the abstract syntax tree (AST) of the program.

Lexer

The `lexer :: String -> [String]` function is an auxiliary component for parsing, it processes a given string and breaks it into a list of tokens (strings).

```
lexer :: String -> [String]
lexer [] = []
lexer str
    | "<=" `isPrefixOf` str = "<=" : lexer (drop 2 str)
    | "==" `isPrefixOf` str = "==" : lexer (drop 2 str)
    | ":=" `isPrefixOf` str = ":=" : lexer (drop 2 str)
    | otherwise = case head str of
        ' ' -> lexer (tail str) -- ignore spaces
        '(' -> "(" : lexer (tail str)
        ')' -> ")" : lexer (tail str)
        ';' -> ";" : lexer (tail str)
        '=' -> "=" : lexer (tail str)
        '+' -> "+" : lexer (tail str)
        '-' -> "-" : lexer (tail str)
        '*' -> "*" : lexer (tail str)
        ':' -> ":" : "=" : lexer (tail str)
        _   -> (head str : takeWhile condition (tail str)) : lexer
(dropWhile condition (tail str))
  where
    condition x = x `notElem` " ()=;=+-*<:"
```

Parsing Functions

buildData

The buildData function takes a list of tokens and constructs a list of statements (`[Stm]`). It uses the `findFirstNotNested` function to find the first semicolon (`;`) that is not nested within parentheses.

It then splits the list into two parts before and after the semicolon, after this it calls the `buildStm` function on the initial statement and recursively calls itself on the rest of the tokens.

buildStm

The buildStm function takes a list of tokens representing a statement and constructs the corresponding statement (`Stm`). Depending on the first token, it calls different procedures (`buildIfStm`, `buildWhileStm`, `buildAssignStm`) to construct the specific type of statement.

```haskell
buildStm :: [String] -> Stm
buildStm list
  | head list == "if" = buildIfStm list
  | head list == "while" = buildWhileStm list
  | otherwise = buildAssignStm list
  where
    buildIfStm list
      | head (tail elseStatements) == "(" = If (buildBexp (tail bexp))
(buildData thenStatements) (buildData (tail elseStatements))
      | otherwise = If (buildBexp (tail bexp)) (buildData thenStatements)
[buildStm (tail elseStatements)]
      where
        (bexp, rest) = break (== "then") list
        (thenStatements, elseStatements) = break (== "else") (tail rest)
    buildWhileStm list
      | head (tail doStatements) == "(" = While (buildBexp (tail bexp))
(buildData (tail doStatements))
      | otherwise = While (buildBexp (tail bexp)) [buildStm (tail
doStatements)]
      where
        (bexp, doStatements) = break (== "do") list
    buildAssignStm list = Assign (head var) (buildAexp (tail aexp))
      where
        (var, aexp) = break (== ":=") list
```

Taking advantage of the nested `where` cases, we can differentiate the functionality of the function accordingly to the first character of the token list (`[String]`). Then, using the prelude function `break`, it divides the token list accordingly (in 3 or 2 parts) and further builds the statement (`Stm`) using other build functions.

buildAexp & buildBexp

These functions take a list of tokens and construct arithmetic expressions (`Aexp`) and boolean expressions (`Bexp`), respectively.

They use the `findFirstNotNested` function to find the last operator that is not nested within parentheses. Based on the operator found, they construct the corresponding expression. If no operator is found, it assumes the expression is wrapped in parentheses and calls itself recursively on the expression without the parentheses.

findFirstNotNested

An essential function to the functionality of the above functions is the `findFirstNotNested :: [String] -> [String] -> Maybe Int` which is designed to find the index of the first token in a list of tokens that is not nested within parentheses or other specific constructs.

```haskell
findFirstNotNested :: [String] -> [String] -> Maybe Int
findFirstNotNested targets = find 0 0
  where
    find _ _ [] = Nothing
    find depth index (x:rest)
      | x == "(" || x == "then" = find (depth + 1) (index + 1) rest
      | x == ")" || (x == "else" && depth /= 0) = find (depth - 1) (index
+ 1) rest
      | depth == 0 && x `elem` targets = Just index
      | otherwise = find depth (index + 1) rest
```

The function makes use of a `depth` variable that acts as a counter to keep track of the level of nesting of the current traversal of input token, whose value is increased when an opening parenthesis (`(`) or a `then` statement is encountered and decreased when a closing parentheses (`)`) or a `else` statement.

With this approach, when the depth is equal to 0 and the current token being analyzed is the same as the target one, the index of this token is returned.

parse

The `parse :: String -> [Stm]` function takes a string representing the program code and returns the parsed program, which is a list of statements (`[Stm]`).

```haskell
parse :: String -> Program
parse = buildData . lexer
```

It achieves this by first using the `lexer` function to tokenize the input string and then using the `buildData` function to build the AST of the program.

Conclusions

In conclusion, this project provided a valuable opportunity to delve deeper into Haskell programming and its application in building an assembler, compiler and parser. The process of tokenizing an input string and building an Abstract Syntax Tree (AST) from it was both challenging and rewarding, and it offered practical insights into how programming languages are processed.

The project deadlines were reasonable, allowing for a thorough exploration of the problem space without undue pressure. The project guide initially seemed unclear, however, we successfully navigated through this complexity.

The group worked effectively together, successfully implementing the proposed features and producing well-documented and organized code. This project not only enhanced our Haskell programming skills but also deepened our understanding of the language's unique features and quirks.

Moving forward, we are excited to apply the knowledge and skills gained from this project to future challenges in programming language processing and other areas of software development.

Bibliography

- GitHub Copilot
- https://www.haskell.org