PFL

Our group is T09_G11

- Diogo Alexandre Soares Martins - 202108883 (50 %)
- Pedro Filipe Pinho Oliveira - 202108669 (50 %)

Installation and Usage

To use this library you should use GHCI to load the file Main.hs in the src folder. It contains 2 functions to test the assembler, `testAssembler` and the parser, `testParser`.

Description

This project consists of a parser for a small imperative programming language and an assembler to a set of low-level machine instructions, written in Haskell. The possible statements in the language are:

- An assignment statement (e.g. `x := 1;`)
- An if-then-else statement (e.g. `if x > 0 then x := 1; else x := 0;`)
- A while statement (e.g. `while x > 0 do x := x - 1;`)
- A sequence of statements (e.g. `x := 1; y := 2;`)

The arithmetic expressions can be composed of:

- Integer constants (e.g. `1`)
- Variables (e.g. `x`)
- Addition (e.g. `x + 1`)
- Subtraction (e.g. `x - 1`)
- Multiplication (e.g. `x * 2`)

The boolean expressions can be composed of:

- Boolean constants (e.g. `True`)
- Boolean comparisons (e.g. `True = False`)
- Boolean negation (e.g. `not True`)
- Boolean conjunction (e.g. `True and False`)
- Relational operators (e.g. `x <= 0`, `x == 0`)

Implementation

The project is divided into 2 parts, the parser and the interpreter. Two files are involved in the parser, `Parser.hs` and `Compiler.hs`, and one file is involved in the interpreter, `Runner.hs`.

Parser

Parser.hs The parser was implemented using the Parsec library. In this file we define the data types for the language, a lexer and a parser for each type of statement and expression, and a top-level parser for the program.

The data types created are the following:

- Aexp: defines arithmetic expressions
- ABinOp: dfines binary arithmetic operators
- Bexp: defines boolean expressions
- BBinOp: defines binary boolean operators

- RBinOp: defines binary relational operators
- Stmt: defines the types of statements
- Program: defines the type of a program

We also have a language definition for the lexer, which identifes variables name syntax, reserved words and operators:

```
language =
    emptyDef {
        identStart = lower,
        identLetter = letter,
        reservedNames = ["if", "then", "else", "while", "do",
                         "not", "and", "True", "False"],
        reservedOpNames = ["+", "-", "*", ":=", "<=", "=", "and", "not"]
    }
```

Then we have a per-type parser, which parses the input string into the corresponding data type. For example, the parser for arithmetic expressions is the following:

```
aExprOperators = [ [Infix (ABinary Mul <$ reservedOp "*") AssocLeft],
                   [Infix (ABinary AddOp <$ reservedOp "+") AssocLeft,
                    Infix (ABinary SubOp <$ reservedOp "-") AssocLeft]]

aExprTerm = parens aExpr <|> varParser <|> intParser

aExpr :: Parser Aexp
aExpr = buildExpressionParser aExprOperators aExprTerm
```

It uses the `buildExpressionParser` from the Parsec.Expr module to parse expressions with operators and precedence, defined in `aExprOperators`. The `aExprTerm` parser defines the possible terms in an arithmetic expression, which can be a parenthesized expression, a variable or an integer constant.

A more simple parser is the assign parser, which parses an assignment statement:

```
assignParser :: Parser Stm
assignParser = Assign <$> identifier <*> (reservedOp ":=" *> aExpr)
```

It uses the `<$>` operator to apply the `Assign` constructor to the result of the `identifier` parser, which parses a variable name, and the result of the `aExpr` parser, which parses an arithmetic expression. The `reservedOp` parser is used to parse the assignment operator.

The top-lever parser takes a string and parses it into a `Program` data type using the `parse` function from the Parser module, which takes a parser and a string to parse. We use the `initParser` parser to parse whitespaces and then start parsing the program with the `statement` parser:

```
parse :: String -> Program
parse input =
    case P.parse initParser "" input of
        Left err -> error $ show err
        Right x -> x

initParser :: Parser Stm
initParser = whiteSpace >> statement
```

The parser will generate an abstract syntax tree for the program, which the compiler will use to generate the machine code.

Compiler.hs The data types defined for the machine code are the ones defined in the project proposal:

- Inst: the instructions available
- Code: is a list of instructions

The compiler is composed of 3 functions:

- `compA`: compiles an arithmetic expression into a list of instructions
- `compB`: compiles a boolean expression into a list of instructions
- `compile`: compiles a statement into a list of instructions

The arithmetic expression compiler takes arithmetic expressions, which can be a integer constant, a variable or and arithmetic operation. When an arithmetic operation is found, each side of the operation is evaluated and then the instruction is added to the list. For example, the expression `x + 1` is compiled into the following instructions:

```
compA (ABinary AddOp (Var "x") (IntVal 1)) == [Fecth "x", Push 1, Add]
```

The boolean expression compiler takes boolean expressions, which can be a boolean constant, a boolean comparison, a boolean negation or a boolean conjunction. Similar to the arithmetic expression compiler, when a boolean comparison is found, each side of the comparison is evaluated and then the instruction is added to the list.

Is important to note that when a arithmetic subtraction or an relational less or equal is found, the right side of the operation is evaluated first. This is because this instructions compare the top of the stack with the second element of the stack. For example, in `x - 1`, the result of `1` is pushed to the stack first, and then the value of `x` is fetched and the subtraction is performed.

The statement compiler takes statements, which can be an assignment, an if-then-else statement, a while statement or a sequence of statements.

- Assignment: an arithmetic expression is evaluated and then the `Store` instruction is added to the list
- If-then-else: a boolean expression is evaluated and then the `Branch` instruction is added to the list. The `Branch` instruction takes two lists of instructions. The result of the boolean expression determines the executed code block.
- While: a boolean expression is evaluated and then the `Loop` instruction is added to the list. The `Loop` instruction takes two lists of instructions. The result of the boolean expression determines the executed code block.

Interpreter

The interpreter for the low-level machine instructions is implemented in a single file, `Runner.hs`.

This file contains definitions of two data structures, `Stack` and `State`, and functions to manipulate them and print them.

`Stack` is used to store (with the push `push` function) local values that will be extracted (with the `top` and `pop` functions) soon after to compute the value of an expression.

`State` is used to store (with the `store` function) variables and their values, so they can be used in future expressions or statements (with the `fetch` function).

These data structures are used during run-time by the `run` function, which interprets `Code` (a list of instructions), and alters the provided `Stack` and `State` and returns the final version of all three, after all code has been executed.

This function is implemented by matching the first element of `Code`, that is, the first `Inst` to one of the possible types of instruction and makes the appropriate changes to the `Stack` and the `State` as per the specification of the instruction set. Instructions that get a value from the `Stack` and require that value to be of a certain type, print (using the `error` function) a run-time error, stating what type that instruction expected to be at the top of the stack, and stop execution.

For instance, when the instruction is an addition, it gets matched in `run (Add:code, stack, state)`, which pops two values from the `Stack`, adds them, pushes the result to the `Stack` and then calls `run` on the rest of the `code` with the current `stack` and `state`. In the case of the addition, the two values must be integers, if they're not, an run-time error message appears and execution is stopped.

The `loop` instruction is essentially a sequence of branches. It recursively calls `Branch`, and if the condition is true executes the body of the loop and the `Branch` instruction again, otherwise it executes a `Noop` and proceeds to calls `run` on the rest of the `code` with the current `stack` and `state`.

Bibliography

- Parsec
- Parsec Module
- Parsec.Token Module
- Parsec.Expr Module
- Parsec Tutorial