Shrimp

an interpreted imperative programming language that wants to believe

Lorenzo Loconte

Introduction

Shrimp is a simple imperative programming language designed during the course of *Formal Methods for Computer Science* at Universita' degli Studi di Bari Aldo Moro. **Shrimp** uses an *eager evaluation strategy*. In order to ensure that, the interpreter executes the code using the *call by value* method.

Software Modules

The program is composed by three main components:

- The **parser**
- The **optimizer**
- The **interpreter**

The **parser** takes in input the source code and convert it to an intermediate representation. The intermediate representation (IR) have the structure of a n-ary tree having the non-terminals of the grammar as internal nodes and commands, identifiers and constants on the leaves.

The **optimizer** takes in input the intermediate representation given by the parser. The result of the optimizer is an *optimized* intermediate representation. It evaluates the constant expressions (both arithmetic and boolean) that might be present in the source code and replace them with the resulting constants. The optimizer also checks for empty commands block and useless branch statements.

The **interpreter** execute the semantics present in an intermediate representation. The basic idea is to use a **state** (or environment) that collects the values of the variables during the execution of the program. The result of the interpretation is the resulting state, that is a set of assignments to the variables.

The Parser

The parser can be viewed as a function from a string to a list of pairs of values and strings.

```
newtype Parser a = Parser {unwrap :: String -> [(a, String)]}
```

Note that the parser have a special function called `unwrap` that takes the function out from the parser. The parser is implemented in a "monadic" way. That is, I implemented the following interfaces: *monad plus* and *monad alternative* as suggested in *Monadic Parsing in Haskell* (Hutton & Meijer). In order to implement the monad interface I also implemented the *functor* and *applicative* interfaces as in the following code snippet.

```
instance Functor Parser where
fmap f p = Parser (\cs ->
  [(f a, cs') | (a, cs') <- unwrap p cs])

instance Applicative Parser where
pure a = Parser (\cs -> [(a, cs)])
p <*> q = Parser (\cs -> concat
  [unwrap (fmap a q) cs' | (a, cs') <- unwrap p cs])

instance Monad Parser where
return a = pure a
p >>= f = Parser (\cs -> concat
  [unwrap (f a) cs' | (a, cs') <- unwrap p cs])

instance MonadPlus Parser where
zero = Parser (const [])
p `plus` q = Parser (\cs -> unwrap p cs ++ unwrap q cs)

instance MonadAlternative Parser where
(<|>) p q = Parser (\cs ->
  case unwrap (p `plus` q) cs of
    [] -> []
    (x : _) -> [x])
```

The `plus` operator defined on parsers concatenates the result of each one. Moreover, the alternative monad give us an operator `<|>` useful for combining parsers in a mutually exclusive way. Plus monad and alternative monad classes are defined as following.

```
class (Monad m) => MonadPlus m where
zero :: m a
plus :: m a -> m a -> m a

class (MonadPlus m) => MonadAlternative m where
(<|>) :: m a -> m a -> m a
many :: m a -> m [a]
many m = some m <|> return []
some :: m a -> m [a]
some m = liftA2 (:) m (many m)
```

where `liftA2` is the composition of `fmap` and the applicative operator `<*>`. The functions `many` and `some` are called *combinators* and are used to define the concept of repeated parsing. Using a monadic definition of parser, it permits us to easily build a parser for each structure of the programming language. For example, it's possible to define a parser for symbols, identifiers and integers as in the following code snippet.

```
item :: Parser Char
item = Parser (\case "" -> []; (c : cs) -> [(c, cs)])

satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do c <- item; if p c then return c else zero

token :: Parser a -> Parser a
token p = do space; v <- p; space; return v

space :: Parser String
space = many $ satisfy isSpace

identifier :: Parser String
identifier = token $ some $ satisfy isLetter

constant :: Parser Int
constant = read <$> token (some $ satisfy isDigit)

char :: Char -> Parser Char
char c = satisfy (c ==)
```

That is, I firstly defined an `item` function that reads a character from the string. Then I defined a function called `satisfy` that apply a constraint to the character read by the parser. In the end, using both the combinators `many` and `some` I built parsers for identifiers, constants and also spaces that are the main blocks of the context-free grammar. In the same way, I also defined parsers for keywords and symbols.

```
keyword :: String -> Parser String
keyword cs = token $ word cs

word :: String -> Parser String
word [c] = do char c; return [c]
word (c : cs) = do char c; word cs; return (c : cs)

symbol :: Char -> Parser Char
symbol c = token $ char c
```

In order to build parsers for both arithmetic and boolean expression the `<|>` operator between parsers is used. So, I define the parser for arithmetic expressions using other sub-parsers that are used in order to maintain the operators precedence. The parser for boolean expressions is similar to the parser for arithmetic expressions and so it's omitted.

```
arithmeticExpr :: Parser ArithmeticExpr
arithmeticExpr = do
  a <- arithmeticTerm
    do symbol '+'; Add a <$> arithmeticExpr
    <|> do symbol '-'; Sub a <$> arithmeticExpr
    <|> do return a

arithmeticTerm :: Parser ArithmeticExpr
arithmeticTerm = do
  a <- arithmeticFactor
    do symbol '*'; Mul a <$> arithmeticTerm
    <|> do symbol '/'; Div a <$> arithmeticTerm
    <|> do symbol '%'; Mod a <$> arithmeticTerm
    <|> do return a

```

```
arithmeticFactor :: Parser ArithmeticExpr
arithmeticFactor =
  do Constant <$> constant
  <|> do Identifier <$> identifier
  <|> do symbol '-'; Neg <$> arithmeticExpr
  <|> do symbol '('; a <- arithmeticExpr; symbol ')'; return a
```

In the end we have a parser for every command described in the grammar. So, I defined a parser for the following commands: `skip`, `assignment`, `if-then-else`, and `while-do`.

```
assignment :: Parser Command
assignment = do
  d <- identifier
  symbol '='
  a <- arithmeticExpr
  symbol ';'
  return (Assignment d a)

branch :: Parser Command
branch = do
  keyword "if"
  symbol '('
  b <- booleanExpr
  symbol ')'
  keyword "then"
  c1 <- block
  do
    keyword "else"
    c2 <- block
    keyword "end if"
    symbol ';'
    return (Branch b c1 c2)
    <|> do
      keyword "end if"
      symbol ';'
      return (Branch b c1 [Skip])

loop :: Parser Command
loop = do
  keyword "while"
  symbol '('
  b <- booleanExpr
  symbol ')'
  keyword "do"
  c <- block
  keyword "end while"
  symbol ';'
  return (Loop b c)

skip :: Parser Command
skip = do
  keyword "skip"
  symbol ';'
  return Skip
```

It's important to notice that the grammar also allows for `if-then` statements (i.e. without the `else` command block). A parser that works on both `if-then-else` and `if-then` statements is implemented by combining the two parsers using the `<|>` operator. Moreover, using the `many` combinator and the `<|>` operator, the parsers for both multiple commands and a single command are straightforward.

```
block :: Parser Block
block = many command

command :: Parser Command
command = assignment <|> branch <|> loop <|> skip
```

The parser for the entire program is defined as in the following code snippet.

```
program :: Parser Block
program = do keyword "shrimp"; block

parse :: String -> Result (Block, String)
parse cs = case unwrap program cs of
  [] -> Error EmptyProgram
  [(b, cs)] -> Ok (b, cs)
```

The Optimizer

The *optimization* process is an intermediate step between the parsing and the interpretation of the program itself. Currently, the main optimization step is related to the execution of constant values. That is, if an expression in a loop is defined only on constant values, it's better to optimize the computation of that expression by replacing it with the result. This procedure is done before the interpretation of the program. For example, consider the following arithmetic expression, expressed in intermediate representation, that we wish to optimize.

```
let expr = Div (
    (Mul (Identifier "x") (Sub (Constant 5) (Constant 3)))
    (Add (Constant 9) (Constant 1)))
```

If we apply the optimization step to this arithmetic expression, we obtain the equivalent but more efficient arithmetic expression.

```
let expr' = Div (Mul (Identifier "x") (Constant 2)) (Constant 10)
```

The implementation of the *optimization* process in Haskell is straightforward, due to simple recursion functions. A very similar optimization process is also implemented on boolean expressions. The *optimization* process also includes a basic optimization on commands such as `skip`, `if-then-else` and `while-do`. First of all, all the `skip` commands are removed from the intermediate representation. Moreover, if the condition of a `if-then-else` command is always *true* then the entire statement is replaced with the first block of commands. In a similar way, if the condition is always *false* then the entire statement is replaced with the second block of commands. Furthermore, this approach is also used for `while-do` commands. That is, if the condition of a `while-do` command is always *false* then the entire statement is completely removed. However, if the condition is always *true* then an exception named *Infinite Loop* is raised. In other words, the optimizer is capable of detecting trivial infinite loops and prevents the interpretation of such programs.

The Language Syntax

The syntax for the **Shrimp** programming language can be denoted using EBNF (Extended Backus Naur Form) as following:

```
Type ::= "int"
Integer ::= [0-9]+
Identifier ::= [a-zA-Z_]+
Program ::= "shrimp" Block
Block ::= [Command]*
Command ::= {Assignment | Branch | Loop}
Assignment ::= Identifier "=" ArithmeticExpr ";"
Branch ::= "if" "(" BooleanExpr ")" "then" Block
           ["else" Block] "end if" ";"
Loop ::= "while" "(" BooleanExpr ")" "do"
         Block "end while" ";"

ArithmeticExpr ::=
      ArithmeticTerm "+" ArithmeticExpr
    | ArithmeticTerm "-" ArithmeticExpr
    | ArithmetciTerm
ArithmeticTerm ::=
      ArithmeticFactor "*" ArithmeticTerm
    | ArithmeticFactor "/" ArithmeticTerm
    | ArithmeticFactor "%" ArithmeticTerm
    | ArithmeticFactor
arithmeticFactor ::=
      Integer
    | Identifier
    | "-" ArithmeticExpr
    | "(" ArithmeticExpr ")"

BooleanExpr ::=
      BooleanTerm "or" BooleanExpr
    | BooleanTerm
BooleanTerm ::=
      BooleanFactor "and" BooleanTerm
    | BooleanFactor
BooleanFactor ::=
      "true"
    | "false"
    | "not" BooleanExpr
    | ArithmetciExpr "eq" ArithmeticExpr
    | ArithmetciExpr "neq" ArithmeticExpr
    | ArithmetciExpr "lt" ArithmeticExpr
    | ArithmetciExpr "gt" ArithmeticExpr
    | ArithmetciExpr "leq" ArithmeticExpr
    | ArithmetciExpr "geq" ArithmeticExpr
    | "(" BooleanExpr ")"
```