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 literals 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 literals. 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.

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 ")"
```