Extending the OOPS Compiler

Martin Ring

Studiengang Informatik
Universität Bremen
February 15, 2012

1 TRUE and FALSE

Including the `TRUE` and `FALSE` literals is easy. First we include it in the list of keywords in `de.martinring.oopsc.lexical.To`

```
...
val keywords = SortedSet(
  AND,BASE,BEGIN,CLASS,DO,ELSE,ELSEIF,END,EXTENDS,FALSE,IF,IS,ISA,METHOD,MOD,
  NEW,NOT,NULL,OR,PRIVATE,PROTECTED,PUBLIC,READ,RETURN,SELF,THEN,TRUE,WHILE,
  WRITE,ACCESS,ASSIGN,CLOSING_PARENTHESES,COMMA,DIVIDE,EQUAL,GREATER,
  GREATER_OR_EQUAL,LESS,LESS_OR_EQUAL,MINUS,NOT_EQUAL,OPENING_PARENTHESES,
  PLUS,SEMICOLON,TIMES,TYPE_OF)
...
case object FALSE     extends Keyword("FALSE")
...
case object TRUE      extends Keyword("TRUE")
...
```

And now it has to be included in the parser

```
def literal: Parser[Expression] = positioned (
    number                        { Literal.Int( ) }
  | FALSE                         { Literal.False }
  | TRUE                          { Literal.True }
  | NULL                          { Literal.Null }
  | SELF                          { VarOrCall(new RelativeName("SELF")) }
  | BASE                          { VarOrCall(new RelativeName("BASE")) }
  | NEW ~> name                   { x => New(x) at x }
  | "(" ~> disjunction <~ ")"
  | varorcall
  | failure("expression expected"))
```

2 ELSE and ELSE IF

We extend our `If` type by a new parameter `elseBody`

```
case class If(condition: Expression, body: List[Statement],
              elseBody: List[Statement]) extends Statement
```

Again, we introduce two new keywords in the `Lexical`

```
val reserved = Set(
  "CLASS", "IS", "END", "METHOD", "BEGIN", "READ", "NEW",
  "WRITE", "IF", "THEN", "WHILE", "DO", "MOD", "SELF",
  "TRUE", "FALSE", "ELSE", "ELSEIF")
```

Now we need new production rules in the syntax:

```
def statement: Parser[Statement] =
    ...
  | "IF" ~> relation ~
    "THEN" ~ (statement*) ~
     opt(elseIf)  <~
    "END" <~ "IF" { case cond~ ~body~elseBody => If(cond, body, elseBody getOrElse Nil) }
    ...

def elseIf: Parser[List[Statement]] =
    "ELSE" ~> (statement*)
  | "ELSEIF" ~> relation ~ "THEN" ~
    (statement*) ~
    opt(elseIf)   { case cond~ ~body~elseBody => List(If(cond,body,elseBody getOrElse Nil)) }
```

We also need to adjust the output in `Output.scala` and the code generation in `Code.scala`:

```
  case If(condition, body, elseBody) => for {
    condition <- generate(condition)
    body <- sequence(body map generate)
    elseBody <- sequence(elseBody map generate)
    elseLabel <- nextLabel
    endLabel <- nextLabel
  } yield Instructions("IF")(
    Instructions("CONDITION")(
      condition,
      R5 << ~R2 || "Get condition from stack",
      R2 <<- R1,
      R5 << (R5 === 0) || "if 0 then",
      Instruction("JPC", R5, elseLabel) || "jump to END IF"),
    Instructions("THEN")(
      Instructions("")(body :*),
      R0 << endLabel),
    Instructions("ELSE")(
      Label(elseLabel),
      Instructions("ELSE BODY")(elseBody :*)),
    Label(endLabel))
```

We introduce a new label `elseLabel` to which we jump if the condition is false. After the body of `THEN` we jump to `endLabel`