Compiler Practical Summer 2013
Syntax Analysis of Classes, Methods, and Statements

Berthold Hoffmann (B. Gersdorf, T. Röfer)
hof@informatik.uni-bremen.de
Cartesium 2.48
Structure

1. Lexical Analysis
2. Syntax Analysis
3. Stack Machine
4. Conventions for Code Generation
5. Task (a): Adding TRUE and FALSE
6. Task (b): Adding ELSE and ELSEIF
Lexical Analysis

Character sequence:

{ This program just writes its input. }  
CLASS Main IS
   METHOD main IS
      c : Integer;
   BEGIN
      READ c;
      WHILE c ≠ -1 DO
         WRITE c; | write character
         READ c; | and read next one
      END WHILE
   END METHOD
END CLASS

Lexeme sequence:

CLASS IDENT: Main
   IS
   METHOD IDENT: main
   IS
   IDENT: c
   COLON
   IDENT: Integer
   SEMICOLON
   BEGIN
   READ
   IDENT: c
WHILE
   IDENT: c
   NEQ
   MINUS
   NUMBER: 1
   DO
   WRITE
   IDENT: c
   SEMICOLON
   READ
   IDENT: c
   SEMICOLON
   END
WHILE
   END
   METHOD
   END
   CLASS
Lexeme sequence:

```
CLASS
IDENT: Main
IS
METHOD
IDENT: main
IS
IDENT: c
COLON
IDENT: Integer
SEMICOLON
BEGIN
READ...
```

Syntax tree structure:

```
CLASS Main
METHODS
METHOD main
VARIABLES
c: Integer
BEGIN
READ c
WHILE
NEQ c
MINUS
1: _Integer
DO
WRITE c
READ c
```
Syntax Tree: JAVA Classes

CLASS Main
  METHODS
    METHOD main
  VARIABLES
    c : Integer
  BEGIN
    READ c
    WHILE
      NEQ c
      MINUS
      1 : _Integer
      DO
      WRITE c
    READ c

Program, ClassDeclaration
  LinkedList<MethodDeclaration>
    MethodDeclaration
      VarDeclaration
      LinkedList<Statement>
        ReadStatement
        VarOrCall
        WhileStatement
          BinaryExpression, VarOrCall,
          UnaryExpression, LiteralExpression
        LinkedList<Statement>
        WriteStatement, VarOrCall
        ReadStatement, VarOrCall
Lookahead During Analysis

• Lexical analysis
  – One character ahead: `LexicalAnalysis.c`
  – Read: `LexicalAnalysis.nextChar()`

• Syntax analysis
  – One symbol ahead: `LexicalAnalysis.symbol`
  – Read: `LexicalAnalysis.nextSymbol()`
  – Convenience methods
    – `SyntaxAnalysis.expectSymbol(...)`
    – `SyntaxAnalysis.expect[Resolvable]Ident(...)`
Syntax Analysis: Classes, Methods

```
program ::= classdecl
classdecl ::= CLASS identifier IS
            { memberdecl }
            END CLASS
memberdecl ::= vardecl ';
             | METHOD identifier IS methodbody
vardecl ::= identifier { ',' identifier } ':' identifier
methodbody ::= { vardecl ';
                BEGIN statements
                END METHOD
```
Syntax Analysis of Statements

\[
\begin{align*}
\text{statements} & ::= \{ \text{statement} \} \\
\text{statement} & ::= \text{READ memberaccess } ';' \\
& \mid \text{WRITE expression } ';' \\
& \mid \text{IF relation} \\
& \quad \text{THEN statements} \\
& \quad \text{END IF} \\
& \mid \text{WHILE relation} \\
& \quad \text{DO statements} \\
& \quad \text{END WHILE} \\
& \mid \text{memberaccess [ '}\text{:='} \text{expression} ] ';$
\end{align*}
\]
Reverse Polish Notation (RPN)

Example:

\[ a := 2 \]

\[ 1 + (7 \times a) \times 3 \]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push 2</td>
<td>2</td>
</tr>
<tr>
<td>a := pop</td>
<td></td>
</tr>
<tr>
<td>Push 1</td>
<td>1</td>
</tr>
<tr>
<td>Push 7</td>
<td>1, 7</td>
</tr>
<tr>
<td>Push a</td>
<td>1, 7, 2</td>
</tr>
<tr>
<td>Mult</td>
<td>1, 14</td>
</tr>
<tr>
<td>Push 3</td>
<td>1, 14, 3</td>
</tr>
<tr>
<td>Mult</td>
<td>1, 42</td>
</tr>
<tr>
<td>Addi</td>
<td>43</td>
</tr>
</tbody>
</table>
Operations of the Stack Machine

- Literals, variables, \textit{NEW, SELF}
  - Push a value onto the stack

- Unary operators
  - Replace the top of the stack
  - '-', \textit{DEREF, BOX, UNBOX, '}-attribute
Operations of the Stack Machine

• Binary operators
  – Pop two entries from the stack and push the result onto the stack
  – Left operand lies below right operand
  – +, -, *, /, MOD, =, #, <, <=, >, >=

• Statements
  – Pop values from the stack, but do not push onto it (any exceptions?)
  – READ, WRITE, :=, .-method ...
  – Stack is empty after a statement has been executed
Code Conventions

- **R0**: instruction counter
- **R1**: The value 1
- **R2**: Stack pointer
- **R3**: Frame pointer
- **R4**: Heap pointer

<table>
<thead>
<tr>
<th>Address</th>
<th>Method frame, for call of f</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3-2</td>
<td>SELF</td>
</tr>
<tr>
<td>R3-1</td>
<td>Return address</td>
</tr>
<tr>
<td>R3</td>
<td>Predecessor frame (main)</td>
</tr>
<tr>
<td>R3+1</td>
<td>a</td>
</tr>
<tr>
<td>R3+2</td>
<td>b</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>R2</td>
<td>Last intermediate value</td>
</tr>
</tbody>
</table>

CLASS Main

METHOD f IS a, b: Integer;
BEGIN END
METHOD main IS BEGIN
  f; | Aufruf von Methode ‘f’
END METHOD
END CLASS
Code Conventions

- **push Rx**
  - ADD R2, R1
  - MMR (R2), Rx
- **pop Rx**
  - MRM Rx, (R2)
  - SUB R2, R1

<table>
<thead>
<tr>
<th>Address</th>
<th>Method frame, for call of f</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3-2</td>
<td>SELF</td>
</tr>
<tr>
<td>R3-1</td>
<td>Return address</td>
</tr>
<tr>
<td>R3</td>
<td>Predecessor frame (main)</td>
</tr>
<tr>
<td>R3+1</td>
<td>a</td>
</tr>
<tr>
<td>R3+2</td>
<td>b</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>R2</td>
<td>Late intermediate value</td>
</tr>
</tbody>
</table>
Task (a): TRUE and FALSE

• **TRUE** and **FALSE** are keywords (i.e., symbols)
  – `enum Symbol.Id`
  – `LexicalAnalysis.LexicalAnalysis(...)`

• **TRUE** and **FALSE** are literals
  – `SyntaxAnalysis.literal()`
  – … Typ `ClassDeclaration.boolType`

• **TRUE** and **FALSE** are values
  – **FALSE**: 0
  – **TRUE**: 1

5%
Task (b): ELSEIF ELSE Syntax

\[
\text{statement ::= READ memberaccess ';'
                 | WRITE expression ';
                 | IF relation
                 THEN statements
                 { ELSEIF relation THEN statements }
                 [ ELSE statements ]
                 END IF
                 | WHILE relation
                 DO statements
                 END WHILE
                 | memberaccess [ ':=' expression ] ';
\]
Task (b): ELSEIF ELSE

- **ELSE** and **ELSEIF** are keywords (i.e., symbols)
  - `enum Symbol.Id`
  - `LexicalAnalysis.LexicalAnalysis(...)`

- **ELSE** and **ELSEIF** extend a statement
  - `SyntaxAnalysis.statement(...) only there?`
  - `class IfStatement`
Task(b): Syntactic Sugar

• **ELSEIF** is „syntactic sugar“: it can be reduced to **ELSE IF**

• **ELSE** branch needs to be supported in the syntax tree
  – `IfStatement.contextAnalysis(...)`
  – `IfStatement.print(...)`

• **ELSE** needs additional code
  – `IfStatement.generateCode()`