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

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

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
```

Syntax Analysis

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

```
statements      ::= { statement }

statement       ::= READ memberaccess ';
                |   WRITE expression ';'
                |   IF relation
                    THEN statements
                    END IF
                |   WHILE relation
                    DO statements
                    END WHILE
                |   memberaccess [ ':=' expression ] ';'
```

Stack Machine

- Reverse Polish Notation (RPN) *Example:*

  a:=2

  1 + (7*a) * 3

Operation	Stack
Push 2	2
a:= pop	
Push 1	1
Push 7	1, 7
Push a	1, 7, 2
Mult	1, 14
Push 3	1, 14, 3
Mult	1, 42
Addi	43

Operations of the Stack Machine

- Literals, variables, *NEW*, *SELF*
  - Push a value onto the stack
- Unary operators
  - Replace the top of the stack
  - '-', *DEREF*, *BOX*, *UNBOX*, '.'-attribute

Operations of the Stack Machine

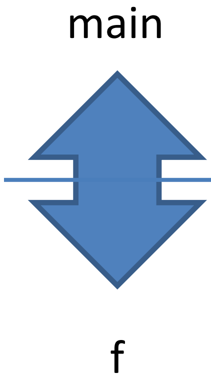
- Binary operators
  - Pop two entries from the stack and push the result onto the stack
  - Left operand lies below right operand
  - +, -, *, /, *MOD*, =, #, <, <=, >, >=

- 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

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



Address	Method frame, for call of f
R3-2	SELF
R3-1	Return address
R3	Predecessor frame (main)
R3+1	a
R3+2	b
…	…
R2	Last intermediate value

Code Conventions

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

Address	Method frame, for call of f
R3-2	SELF
R3-1	Return address
R3	Predecessor frame (main)
R3+1	a
R3+2	b
…	…
R2	Late intermediate value

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%

```
literal   ::= number
            | NULL
            | TRUE
            | FALSE
            | SELF
            | NEW identifier
            | '(' expression ')'
            | varorcall
```

Task (b): ELSEIF ELSE Syntax

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()*

5%