1 Part II: Java Explorer (Jex)

Example program to compute and print the factorial of 3:

```
import java.lang.*;
int fac (int x) {                 /* defines a function */
        int res;                  /* defines a local variable */
        res = 1;                  /* sets a local variable */
        while (x > 1) {
                res = res * x;    /* uses local variable and argument */
                x = x - 1;
        }
        return res;
}
int a; a = fac(3);                /* defines global variable, calls f */
System.out.println(a);            /* java class System, static attribute out,
                                           dynamic method call */
```

2 Unqualified Names

An unqualified name (it does not appear right of a ".") is defined,

- if the definition is in the same or an enclosing block.
- if the name is defined before accessing it.

If there is more than one definition, then the innermost definition is used.

```
int i;
int foo(int n) {
        int i;
        if (n == 0) {
                return i;
        }
        return foo(n-1);
}
```

If an unqualified name is not defined it is assumed to be a Java class.

- We look for it on the $CLASSPATH using imports as in Java.

3 Qualified Names

If a qualified name is not followed by parenthesis:

- Classname.field refers to a static field of the class.
- object.field refers to a field of the object.

If they are followed by parenthesis:

- Classname.method(...) calls the static method of the class.
- object.method(...) call the method of the object.
- We choose dynamically (at call time) the best fitting method for the argument types, using reflection.
- If we cannot find an appropriate method an error occurs.

4 Types

- Primitive types
  - **int** (1,2,7)
  - **boolean** (true, false)
- Java classes
  - java.lang.String ("hello")
  - java.io.PrintStream (System.out)

In a type position, a name is always referring to a class.

5 The new expression

We can generate new objects using new

- **new** Classname (...)
- We choose dynamically the best constructor, depending on the arguments.

6 Operator expressions

We have binary operators with the following precedence:

- $* \ /$
- $+ \ -$
- $== \ != \ < \ > \ \leq \ \geq$
- &&
- $||$

$==$ and $!=$ also work on objects, the others only on primitive types.

7 Concrete Syntax of Jex

```
Program      =   { Statement | Definition }.
Definition   =   Formal ";"
             |   FunDef
             |   "import" { IDENT "." } ( "*" | IDENT ) ";"
             .
FunDef       =   Type IDENT "(" [ Formals ] ")" Statement.
Type         =   "int" | "boolean" | IDENT.
Formals      =   Formal { "," Formal }.
Formal       =   Type IDENT.
```

```
Statement  =  "if" "(" Expr ")" Statement [ "else" Statement ]
           |  "while" "(" Expr ")" Statement
           |  "{" { Statement | Formal ";" } "}"
           |  Expr ";"
           |  [ Expr "." ] IDENT "=" Expr ";"
           |  "return" Expr ";"
           .
Expr       =  NUMLIT | STRINGLIT | "True" | "False"
           |  [ Expr "." ] IDENT [ "(" [ Exprs ] ")" ]
           |  Expr Operator Expr
           |  "new" Type "(" Exprs ")"
           .
Operator   =  "+" | "−" | "∗" | "/"
           |  "==" | "!=" | "<" | ">" | "≤" | "≥"
           |  "&&" | "||"
           .
Exprs      =  Expr { "," Expr }.
```

8 Lexical Syntax of Jex

- Tokens

```
ident     = letter { letter | digit }.
numlit    = digit { digit }.
stringlit = "\"" { "\" noNewline | noEscapeOrQuoteOrNewlines } "\"".
```

All quoted terminals in the concrete syntax are valid tokens.

- Auxiliary

```
letter       = "a" | ... | "z" | "A" | ... | "Z".
digit        = "0" | ... | "9".
noNewline    = ...
```

- White space and comments

```
whitespace = " " | "\n" | "\t" | "\r".
comment    = "/" "*" { noStar | "*" { "*" } noStarOrSlash } "*" "/".
```