1 A Recursive Function

This is a recursive function, which computes the factorial:

```
int x;
x = 5;
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        int m;
        m = factorial(n − 1);
        return m + n;
    }
}
System.println(x + factorial(3));
```


2 How do we store variables?

- For each global variable we need one storage place.
- The easiest way is to store it in GlobalSym.
- For each local variable or argument we need one storage place per call.
- In a compiler for a traditional language, such variables are placed on the stack.
- For each function call we need an environment, where we store the values of arguments and functions.

3 Environments

- For each function call, we have an instance of the class Environment.
- We have an array args to store the arguments and an array locals to store the local variables.

```
class Environment {
    JexValue[ ] args;
    JexValue[ ] locals;

    public Environment (JexValue[ ]args, int localCount) { ... }

    public JexValue get(int i) { ... }

    public void set(int i, JexValue val) { ... }
}
```

- Environments are the dynamic equivalent of scopes.

4 Static and Dynamic Things

- Environments are the dynamic version of scopes
- JexValue are the dynamic version of JexSymbol
- For static things the structuring of blocks is the important structuring mechanism.
- For dynamic things, the call-structure is the important structuring mechanism.
- Scopes are opened and closed at the beginning and end of blocks.
- Environments are opened and closed at the beginning and end of a call.

5 How did we interpret Expressions?

```
public class Evaluator implements Tree.Visitor {
    int val;

    public static int eval(Tree tree) {
        Evaluator ev = new Evaluator();
        tree.apply(ev);
        return ev.val;
    }

    public void caseNumLit(Tree.NumLit tree) {
        val = tree.num;
    }
```



```
    public void caseOperation(Tree.Operation tree) {
        switch (tree.op) {
            case Tokens.PLUS:
                val = eval(tree.left) + eval(tree.right);
                break;
            case Tokens.MINUS:
                val = eval(tree.left) − eval(tree.right);
                break;
            case Tokens.TIMES:
                val = eval(tree.left) ∗ eval(tree.right);
                break;
            case Tokens.DIV:
                val = eval(tree.left) / eval(tree.right);
                break;
            default: throw new InternalError();
        }
    }
}
```


6 How do we interpret Jex

- We write a visitor **class** Interpreter
- When we interpret a part of the program, we usually want to compute a value. We give the visitor an attribute val, which is returned by the visitor.
- When we have to interpret the sum E + F of two expressions E we call interpret() on both subexpressions, add the two resulting values and store them in val.
- However, before we do the addition, we have to check, that the results of the two subexpressions are both integers. We can do that by calling isInteger() on the JexValues.

7 How to interpret Jex (2)

- Sometimes, at the left hand side of the assignment, we want to store something at an expression.
- This happens only for a few kinds of expressions.
- We give a field storeVal to the visitor. If this variable is set (!= null), then instead of computing a value, we are storing the value given in storeVal.
- For example for a variable:
    - If (storeVal == null) we return the current value of the variable.
    - In this case, we also have to check, whether the variable was already initialized.
    - If (storeVal != null) we set the variable to storeVal.
    - Here we have to check, whether the value has the correct type to store it in the variable.

8 How to interpret Jex(3)

- The next problem is the **return**.
- If we interpret a block, we need to interpret the statements one by one.
- but if one of the statements is a return (or contains one) then we shouldn't interpret the rest.
- We give a field **boolean** isReturn to the interpretation visitor, which is set in case of a return.
- In a block we now interpret the statements one by one until we are finished or isReturn was set by the last statement.

9 The Interpreter Visitor

```
public class Interpreter implements Tree.Visitor {
    Environment env;
    JexValue storeVal;
    JexValue val;
    boolean isReturn;

    JexValue interpret(Tree tree, Environment env, JexValue storeVal){
        Interpreter ip = new Interpreter();
        ip.env = env;
        ip.storeVal = storeVal;
        ip.isReturn = false;
        ip.val = null;
        tree.apply(ip);
        this.isReturn = ip.isReturn;
        return ip.val;
    }
}
```


10 Example: While

```
public void caseWhile(Tree.While tree) {
    JexValue b;
    while (!isReturn) {
        b = interpret(tree.expr);
        if (!b.isBoolean())
            throw new JexException(tree.pos,
                "condition in while not boolean");
        if (!b.getBoolean())
            break;
        val = interpret(tree.body);
    }
}
```


11 The Interpreter Specification

Here we give the semantics quite informally. Also it is often only given, what you have do in the correct cases.

```
Program     = DEFLIST { Definition | Statement }
                  interpret every subpart
            ;
Definition  = Formal
                  do nothing
            | FUNDEF Type ident { Formal } Statement
                  do nothing
            | IMPORT { ident } boolean
                  do nothing
            ;
Formal      = VARDEF Type ident
                  do nothing
            ;
```

Statement = ASSIGN Expr Expr
interpret the right expression obtaining val
interpret the left expression with storeVal set to val
| IF Expr Statement Statement
interpret the condition obtaining val
if val is true interpret statement 1 **else**
interpret statement 2
| WHILE Expr Statement
interpret the condition, as **long** as it is true
interpret the statement and **if** isReturn is not set
reinterpret the condition

| BLOCK { Statement | Formal }
interpret one subpart after the other until you are finished or isReturn is set
| EVAL Expr
interpret the expression obtaining val
| RETURN Expr
interpret expression obtaining val, set isReturn
;

Expr = NUMLIT **int**
set val to a JexValue **for** the integer
| STRINGLIT String
set val to a JexValue **for** the string
| BOOLEANLIT **boolean**
set val to a JexValue **for** the **boolean**

| IDENT ident
LOAD: load val from the environment (through symbol)
STORE: store storeVal in the environment (through symbol)
| FUNCALL ident { Expr }
interpret all the arguments
build the **new** calling environment
call interpret on the function–statement
with **this new** environment obtaining val
| METHODCALL Expr ident { Expr }
interpret the receiver and all the arguments
call the callMethod method on the receiver
| FIELDACCESS Expr ident
LOAD: interpret the receiver and
call the getField method on the receiver
STORE: interpret the receiver and
call the setField method on the receiver

| OPERATION Expr Expr op
interpret the two subexpressions, then execute the appropriate operation
| NEW Type { Expr }
interpret type and the argument expressions
call the getNew method on the type
;

Type = IDENT ident
same as in Expression
| INTEGERTP
return the **int class** as a JexValue
| BOOLEANTP
return the **boolean class** as a JexValue
;

12 Example: Ident

```
public void caseIdent(Tree.Ident tree) {
    if (storeVal != null) {
        tree.sym.store(env, storeVal);
    } else {
        val = tree.sym.load(env);
    }
}
```

- STORE: Is it really a variable?
- STORE: Does the value fit the type of the variable?
- LOAD: Was the variable initialized?

13 Exceptions

- In an interpreter we can have two kind of failures.
  - Failure of the interpreter. (e.g. We try to cast to a GlobalSym, where we have indeed a LocalSym). This is considered a bug in the interpreter/compiler.
  - Failure of the user program. (e.g the user tries to add a boolean value to an integer).
- Failures of the user program need to be treated specially.
- We introduce a special exception, JexException, which signals a user error and contains a string describing the exception.
- JexException are caught in the main interpreter loop.
- JexValue throws JexException, if something goes wrong in the reflection; otherwise it throws Error.
- So before calling val.getInteger(), you should make sure that this is allowed by calling val.isInteger().