Interpreters

Part VII

Runtime

- This last part of the course will be concerned with *runtime* aspects.
- Runtime is when the user program actually executes.
- For a *native* compiler this is when we run the native code.
- For an *interpreter* this is when we interpret it.
- For a *virtual machine* (VM) such as the JVM, this is when the JVM runs the Java bytecode.

Part VII
2
Compilation A 20.06.03
http://lamp.epfl.ch/teaching/compilation/2003/html/

The Runtime Environment

- The *Runtime Environment* (RE) is "where" a program is executed.
- For a native compiler this can be the just the physical machine or the machine plus an operating system.
- For an interpreter the runtime environment is the program that interprets the user program.
- For a VM the runtime environment is the VM.

Design choices for the RE

- How is the program represented?
- How is data represented?
- How are variables represented?

Aspects Influencing Design Decisions.

- The characteristics of the running program depends on the design choices of the RE.
- For example:
  - We can trade speed of execution for memory consumption.
  - We can trade speed of execution for RE simplicity (or speed of implementation).

Example: Modern JVMs

- In a modern JVM there are multiple code representations.
- First all functions are bytecode interpreted. (Compact code, less memory – slower execution.)
- If a function is executed often it is compiled to native code. (Large code, more memory – faster execution.)
- This also implies a more complex implementation with calls between bytecode and native code.

Plan

- We will look at some alternatives for these design choices for REs.
- We will start with a simple interpreter and work our way up to a virtual machine.

Interpreter

- The task of an *Interpreter* is to execute a program.
- As can be guessed from the word, an interpreter usually has the ability to *understand* source code.
- In contrast to a *virtual machine* which usually take it's input in the form of virtual machine code.

Interpreting while Parsing

- For some simple languages the interpretation can be done during parsing.
- This can be done by adding code to the semantic action in the parser description.

Design choices

- How is the program represented?
  - As source code.
- How is data represented?
  - The value of each expression is an integer.
  - We represent a value as a Java Integer.
- How are variables represented?
  - We don't support variables.

A Simple Parser

- Grammar:

```
Expr ::= Expr MINUS Term
       | Expr PLUS Term
       | Term
Term ::= Term TIMES Factor
       | Term DIV Factor
       | Factor
Factor ::= NUMBER | LPAR Expr RPAR
```

Simple Interpreter .cup

```
terminal PLUS, MINUS, TIMES, DIV, LPAR, RPAR;
terminal Integer NUMBER;

non terminal Program;
non terminal Integer Expression, Term, Factor;

precedence left PLUS, MINUS;
precedence left TIMES, DIV;

start with Program;
```

Interpreter .cup

```
Program ::= Expression:e
        {: System.out.println(e.intValue()); :}
        ;
Expression ::= Expression:e PLUS Term:t
        {: RESULT = new Integer(e.intValue() +
                                t.intValue()); :}
        |  Expression:e MINUS Term:t
        {: RESULT = new Integer(e.intValue() -
                                t.intValue()); :}
        |  Term:t
        {: RESULT = t; :}
```


Interpreter .cup

```
Term ::= Term:t TIMES Factor:f
        {: RESULT = new Integer(t.intValue() *
                                f.intValue()); :}
        |  Term:t DIV Factor:f
        {: RESULT = new Integer(t.intValue() /
                                f.intValue()); :}
        |  Factor:f
        {: RESULT = f; :}
Factor ::= NUMLIT:n {: RESULT = n; :}
        |  LPAR Expression:e RPAR
        {: RESULT = e; :}
```


Control Flow

- This simple approach works fine for simple expressions.
- Control flow constructs such as 'if' and 'while' are harder to handle.
- For 'while' we would need to "reparse" the statement that is to be repeated.
- Let us try to construct an interpreter for the "Expressive" language.

Solution

- By representing the code by a data structure we can "reexecute" the same piece of code several times.
- This will lead to a slightly more complicated interpreter, which will require at least two passes over the code.
- The code will first be parsed and stored in the internal representation, then the interpretation will be performed.
- We can use the abstract syntax tree for representing the code.

Variables and Booleans

- We will also handle variables and Booleans in this interpreter.
- This can be done by extending the `Symbol` class with a field for the value of a variable.
- We use an abstract superclass `Value` to represent values.
- We create two subclasses `IntValue` and `BoolValue` for the two types of values we need.

Design choices

- How is the program represented?
  - As an Abstract Syntax Tree (AST) with the class `Tree`.
- How is data represented?
  - We have different types of values, integers and Booleans.
  - The value of each expression is either an `IntValue` or a `BoolValue`, subclasses to `Value`.
- How are variables represented?
  - With a symbol table where each symbol can have a value.

The Implementation

- The Interpreter itself can be implemented by a Visitor on the AST.
- We need the Value class:

```
class Value {
  static class IntValue extends Value {
    int i;
    public IntValue(int i) { this.i = i; }
  }
  static class BoolValue extends Value {
    boolean b;
    public BoolValue(boolean b) { this.b = b; }
  }
}
```


Interpreting Expressions

```
public void caseOp(Op tree) {
  switch (tree.op) {
  case TRUE:
    result = new BoolVal(true);
    break;
  case FALSE:
    result = new BoolVal(false);
    break;
  case PLUS:
    IntValue lval = (IntValue) interpret(tree.left);
    IntValue rval = (IntValue) interpret(tree.right);
    result = new IntValue(lval.i + rval.i);
    break;
  …
```


Semantic Analysis Needed

- This assumes that types are correct, it only works if we have a type check pass first.
- We could do the type analysis at the same time.

Analyzing While Interpreting

```
public void caseOp(Op tree) {
  switch (tree.op) {
  case PLUS:
    Value lval = interpret(tree.left);
    Value rval = interpret(tree.right);
    if ((lval instanceof IntValue) &&
       (rval instance of IntValue)) {
       result = new IntValue(
            ((IntValue)lval).i +
          ((IntValue)rval).i);
    } else error();
    break;
…
```


Control Flow

- Now we can try to interpret a control flow construct.
- It turns out to be very easy, since we are writing our interpreter in Java which supports the same control flow constructs.
- It becomes a bit complicated if the type analysis has to be done at the same time.

While (assuming type analysis)

```
public void caseWhile(While tree){
  while(((BoolValue)
    interpret(tree.cond)).b {
      interpret(tree.body);
  }
}
```

Interpreting While, While Analyzing

```
public void caseWhile(While tree) {
  Value cond=interpret(tree.cond);
  while((cond instanceof BoolValue)
        && ((BoolValue) cond).b) {
     interpret(tree.body);
     cond=interpret(tree.cond);
  }
}
```


Variables

- We can to reuse the techniques from the analyzer to handle variables.
- We can use the same symbol table as in the analyzer, and we can add variables to it in the same way.
- For the interpreter we also need to add the value of an expression stored in a variable to the symbol.
- If we interpret an assignment we store the value in the symbol.
- If we interpret an identifier we read the value from the symbol.

Variables

```
// Identifier
  public void caseIdent(Ident tree) {
    // Try to find the symbol.
    Symbol sym = lookup(tree.name);
    if (sym == null) { // Not found.
       error();
    } else {
     if (sym.value == null) { // Not defined.
       error();
    } else {
      result = sym.value; }
  }
```


Functions

- These techniques can handle the language “Expressive”, but most programming languages have functions.
- Functions can be seen as pieces of code with a name.
- Functions can usually take arguments and return a result.
- Functions have their own scope.
- Functions can be recursive.

Recursive Functions

```
int factorial(int n) {
  if(n == 0) { return 1; }
  else {
    int m;
    m = factorial(n-1);
    return m * n;
  }
}
```

- If we compute factorial(2) we need to store the variables n and m more than once.

Environments

- In an *Environment* we store all values of parameters (arguments) and local variables of a function for one specific call.
- We create a new environment when we call a function.
  - We store actual arguments of the call in the environment.
  - We initialize local variables.

Part VII

Environments

- After returning from a function the environment is not needed any more.
- An environment can be represented as an array of Values.

```
class Environment {
  Environment outer; // For nested scope.
  Value[] values;
}
```

- In a compiler environments are implemented on the call-stack (They are called *activation records*).

Part VII

Offsets

- Each local variable has a fixed offset, which tells us at which position in the array its value is at.
- We can store the offset in the Symbol of the local variable.
- The offset can be computed during name analysis.
- Parameters can be seen as local variables and treated the same way.

Part VII

Interpreting Functions

- In the last lecture we talked about interpretation, and we had come to the problem of interpreting functions.
- Functions need an environment where they can store the values of arguments and local variables.