1 Part VII: Interpretation (Run-time)

- Run-time 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 Java compiler, this is when the JVM runs the Java bytecode.

2 What do we need to think about?

- How is the program represented at run-time? (e.g. Java Byte Code)
- How is data represented at run-time? (e.g. format for integers)
- How are variables represented at run-time?
- These issues can be very complex (e.g. in a modern JVM we have even multiple code representations).
  - First we interpret bytecode.
  - If a function is executed often, it is compiled to native code.
  - This implies calls between bytecode and native code.
- The characteristics of the running program will depend heavily on these decisions:
  - speed
  - memory consumption
  - interpreter complexity

3 Representing Values

Very simple interpreter (like our expression language):

- Each value is an integer.
- We represent it by a Java **int**.

Typical interpreter:

- We have values of different types.
- We make one abstract superclass Value.
- For each other kind of value we have a subclass IntValue, BoolValue, ObjectValue ...
- The instances of these classes contain the actual values.
- The interpreter-visitor returns Values.

4 Example

```
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; }
    }
    static class ObjectValue extends Value {
        Value[ ] fields;
        public ObjectValue(Value[ ] fields) { this.fields = fields; }
    }
    ...
}
```

5 Interpreting Expressions

```
void caseBinOp(BinOp tree) {
    Value lval = interpret(tree.left);
    Value rval = interpret(tree.right);
    switch(tree.op) {
    case '+':
        val = new IntValue(((IntValue)lval).i + ((IntValue)rval));
        break;
    case '–':
        ...
    }
}
```

This assumes, that types are correct. It only works well if we had a type-checking pass before.

6 Interpreting Expressions (2)

```
void caseBinOp(BinOp tree) {
    Value lval = interpret(tree.left);
    Value rval = interpret(tree.right);
    switch(tree.op) {
    case '+':
        if ((lval instanceof IntValue) && (rval instanceof IntValue)) {
            val = new IntValue(((IntValue)lval).i + ((IntValue)rval));
        } else {
            error();
        }
        break;
    case '–':
        ...
    }
}
```


7 Representing Data for a Compiler

- For a compiler the questions are, how do we layout values during run-time in memory.
- How many bits do integers, booleans, characters have?
- How do we represent strings?
- How do we represent arrays?
- How do we represent objects?
  - Often: ClassId followed by fields.
  - Sometimes: Garbage Collector information.
- Sometimes we are constrained by the language definition (In C strings are **char**-arrays terminated by null)

8 Memory Management

- At run-time we create new objects.
- We have to allocate memory for them.
- If the memory is not used any more, it has to be given back (otherwise we have a space-leak).
- Some languages (C, C++) have manual memory management (malloc, free).
- Others (Java, LISP) have automatic memory management (garbage collection).
- Allocation/deallocation/Garbage Collection have to be supplied with the compiler.

For an interpreter we can sometimes use the memory management of the host language, Java in our case (The Java garbage collector collects our Values as well).

9 How do we represent Functions?

Interpreter:

- Use abstract syntax tree
  - easiest solution
- Generate byte code for each function and interpret this
  - faster execution
  - faster even if we count byte code generation (we might execute some bytecode more than once)
  - used in Perl, Python, Tcl
- Generate native code for each function and execute this (Just-In-Time compiler)
  - fastest solution
  - complex
  - less portable

10 How do we represent Functions? (2)

Compiler:

- Generate native code (C, C++)
  - fast
- Generate bytecode (Java, Original Pascal, Smalltalk)
  - more portable
  - modern bytecode-interpreters can be fast, too (JVM).

11 How do we represent Variables?

Very simple interpreter (no recursion):

- Store the value in the symbol.
- If we interpret an assignment we store it there.
- If we interpret an identifier, we read it from there.
- Unfortunately, this does not work for languages with recursion.
- Modern programming languages have recursion (FORTRAN had no recursion).

12 A Recursive Function

This is a recursive function, which computes the factorial:

```
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 space for more than one n, m, but there is only one Symbol.

13 Environments

- In an Environment we store all values of parameters and local variables of a function for one specific call.
- We create a new Environment when we call a function.
  - We store the actual arguments of the call in the environment.
  - We initialize local variables.
- Once we returned from the function, we don't need it any more.
- We will typically represent an environment by an array of Values

```
class Environment {
    Environment outer;
    Value[ ] values;
}
```

- In a compiler Environments are implemented on the call-stack.

14 Offsets

- Each local variable has a fixed offset, which tells us at which position to find it in the array.
- The offset is stored in the Symbol of the local variable.
- The offset is usually computed during name analysis.
- The same holds for parameters.

```
void caseIdent(Tree.Ident tree) {
    val = tree.sym.load(currentEnv);
}

class Symbol {
    ...
    abstract Value load(Environment env);
    abstract void store(Environment env, Value val);
}
```


15 Local variables

```
class LocalSymbol extends Symbol {
    ...
    int offset;
    Value load(Environment env) {
        return env.values[offset];
    }
    void store(Environment env, Value val) {
        env.values[offset] = val;
    }
}
```

- It is convenient to use abstract methods in class Symbol.
- This saves us an **instanceof** LocalSymbol.

16 Global Variables

Global variables can be stored in the symbol or in a global environment.

```
class GlobalSymbol extends Symbol {
    ...
    Value val;
    Value load(Environment env) {
        return val;
    }
    void store(Environment env, Value val) {
        this.val = val;
    }
}
```

17 The Interpreter-Visitor

In general the interpreter visitor is another name for evaluator. However, function calls are more complicated.

```
void caseFunCall {
        // call interpret on function arguments (keep results)
        // create a new Environment
        // store the computed arguments in the new environment
        // call interpret on the body of the called function
        //        using the new environment
        // restore old environment
}
```


18 The Difference: Static and Dynamic

- There is an analogy between Value and Symbol:
  - Both have information about an identifier.
  - Symbol stores information about the identifier which is static, that is it does not change over time.
  - Value stores information about the identifier which is dynamic, an identifier has different values over time.
- Symbol is a static concept (it is fixed during interpretation).
- Value is a dynamic concept (it changes during interpretation).
- Scopes are static (Each block has only one scope over time).
- Environments are dynamic (For each call to a function we create a new environment).

19 The Difference: Static and Dynamic(2)

- The abstract syntax tree contains static information.
- Most compiler passes work recursively on the abstract syntax tree and compute static information.
- Only the interpreter computes dynamic information.
- The interpreter is not a simple recursion on the abstract syntax tree. In a function call it jumps from the call to its definition to execute the function body.
- Type checking is sometimes done statically (part of analysis), sometimes dynamically (dynamically) [sometimes not at all].
- Some early interpreters (LISP) did name analysis during interpretation. This can lead to very surprising behaviour.

20 Summary: Compilation

Scanner

- Converts a character stream into a token stream (breaks up the input into words).
- Can bew written by hand or with a scanner generator.

Parser

- Converts the token stream into an abstract syntax tree (analyzes the structure)
- Recognizes syntax errors.
- Can be written by hand or with a parser generator.

21 Summary: Compilation

Analysis

- Associates uses and definitions by attaching Symbols to the abstract syntax tree.
- Uses Scopes to store Symbols intermediately.
- Recognizes undefined variables.
- (Attaches types to the abstract syntax tree)
- (Recognizes Type errors)

Interpretation

- Runs the user program.

22 Other Topics in Compilation

- Code Generation.
  - How to generate machine code?
  - Register allocation.
- Optimization
  - Constant folding $\Rightarrow$ Dataflow analysis.
  - Loop invariants $\Rightarrow$ Loop optimizations.
- Object oriented languages.
  - How to implement objects?
  - How to translate dynamic binding?
- Run-time
  - Threads, how to do synchronization fast?
  - Memory management, garbage collection.