

The Backend



David Notkin
Autumn 2008

We have...

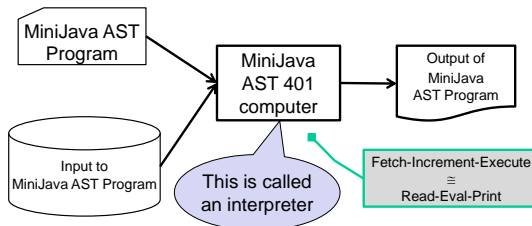
- ...scanned and parsed and type checked and built an abstract syntax tree with a symbol table...
- So we know we have
 - a correct program, and
 - we have a useful representation of that program
- Now what?
 - Generate equivalent code in a lower-level language ... (so we can later run it)
 - Perform further analysis .. (such as?)
 - ...what else?

CSE401 Au08

2

We can execute it immediately...

- To do so, we need to implement a MiniJava-AST computer interpreter

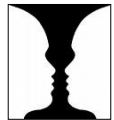


CSE401 Au08

3

Why interpret vs. generate code?

- Time until program can be executed
- Speed of executing program
- Simplicity of implementation
- Flexibility of implementation



TRADEOFFS



Interpreters

- Create data structures to represent run-time program state
 - values manipulated by program
 - activation record (i.e., stack frame) for each called method
 - environment to store local variable bindings
 - pointer to lexically-enclosing activation record/environment (static link)
 - pointer to calling activation record (dynamic link)
- EVAL loop executing AST nodes

An Interpreter for MiniJava:

~readme ([Evaluator](#) subdirectory)

- The main data structure is the **environment**, which keeps track of the values of local variables declared in a given scope, plus some information about declarations in classes.
- **Environments** closely parallel **SymbolTables**
 - "compile-time" information computable before running the program (e.g. declarations and types)
 - "run-time" information representing the program's running state
- Only one symbol table for each program scope, while there can be zero or more environments created for (most) scopes

Continued... ~readme

- There are *environments* for different kinds of scopes (global scope, class scope, and code scope...), as they have different declarations and run-time state.
 - An *activation record* is an instance of an environment
- The (only) global environment maps names of classes to the corresponding class environments...
- A class environment maps the names of locally declared methods to their declarations and the names of locally declared instance variables to their resolved types. Also stores a reference to the environment of its superclass (if any).

CSE401 Au08

7

Continued ~readme

- The values of the instance variables are not stored in the class environment because each instance of the class stores its own values of its instance variables.
- A code environment maps the names of local variables to their current values.
- A method code environment additionally remembers the environment of its caller, for use in printing stack traces during evaluation.
- Each kind of nested environment stores a reference to its lexically enclosing scope's environment.

CSE401 Au08

8

Continued ~readme

- The evaluation values are represented by instances of **Value** classes, organized into a class hierarchy
- Each kind of **ResolvedType** (**Int**, **Boolean**, **Class**, and **Null**) has a corresponding kind of **Value** to use in representing values
- **Int** and **BooleanValues** store their value
- **ClassValues** store the environment for the instantiated class as well as a table that maps instance variable names to the current values for that instance
- **NullValue** represents null pointers.

CSE401 Au08

9

Activation Records

- Each call of a procedure allocates an activation record that stores
 - mapping from names to **Values**, for each formal and local variable in that scope (*environment*)
 - lexically enclosing activation record (*static link*)
- An activation record for a method also stores the calling activation record (*dynamic link*)
- A class activation record also stores
 - methods (to support run-time method lookup)
 - instance variable declarations, not values
 - values stored in class instances (**ClassValues**)

Activation Records vs Symbol Tables

- For each method/nested block scope in a program:
 - exactly one symbol table, storing types of names
 - possibly many activation records, one per invocation, each storing values of names
- For recursive procedures,
 - can have several activation records for same procedure on stack simultaneously
 - All of these activation records have same “shape,” described by single symbol table

Example

```
class Fac {
  public int ComputeFac(int num) {
    int numAux;
    if (num < 1) {
      numAux = 1;
    } else {
      numAux = num * this.ComputeFac(num-1);
    }
    return numAux;
  }
}
```

Interpretation tradeoffs: reprise

- simple conceptually, easy to implement
 - fast turnaround time
 - good programming environments
 - easy to support fancy language features
- slow to execute
 - data structure for value vs. direct value
 - variable lookup vs. registers or direct access
 - EVAL overhead vs. direct machine instructions
 - no optimizations across AST nodes

Compile-time Processing

- Decide representation of run-time data values
- Decide where data will be stored
 - registers
 - format of stack frames
 - global memory
 - format of in-memory data structures (e.g. records, arrays)
- Generate machine code to do basic operations
 - just like interpreting expression, except generate code that will evaluate it later
- Do optimizations across instructions if desired

Compile-time vs Run-time

Compile-time	Run-time
Procedure	Activation record/stack frame
Scope, symbol table	Environment (contents of stack frame)
Variable	Memory location or register
Lexically-enclosing scope	Static link
Calling Procedure	Dynamic link