Implementing a Language

Given type-checked AST program representation:
- might want to run it
- might want to analyze program properties
- might want to display aspects of program on screen for user
  - ...

To run program:
- can interpret AST directly
- can generate target program that is then run recursively

Tradeoffs:
- time till program can be executed (turnaround time)
- speed of executing program
- simplicity of implementation
- flexibility of implementation

Interpreters

Create data structures to represent run-time program state
- values manipulated by program
- activation record (a.k.a. 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

Pros and cons of interpretation

+ 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

Compilation

Divide interpreter work into two parts:
- compile-time
- run-time

Compile-time does preprocessing
- perform some computations at compile-time once
- produce an equivalent program that gets run many times

Only advantage over interpreters: faster running programs
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

<table>
<thead>
<tr>
<th>Compile time</th>
<th>Run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure</td>
<td>Activation record/stack frame</td>
</tr>
<tr>
<td>Scope, symbol table</td>
<td>Environment (Contents of stack frame)</td>
</tr>
<tr>
<td>Variable</td>
<td>Memory location, Register</td>
</tr>
<tr>
<td>Lexically-enclosing scope</td>
<td>Static link</td>
</tr>
<tr>
<td>Calling procedure</td>
<td>Dynamic link</td>
</tr>
</tbody>
</table>

An interpreter for MiniJava

In Environment subdirectory:

Data structure to represent run-time values: Value hierarchy
- analogous to ResolvedType hierarchy

Value
- IntValue
- BooleanValue
- ClassValue
- NullValue

Data structure to store Values for each variable: Environment hierarchy
- analogous to SymbolTable hierarchy

Environment
- GlobalEnvironment
- NestedEnvironment
- ClassEnvironment
- CodeEnvironment
- MethodEnvironment

evaluate methods for each kind of AST class

Activation records

Each call of a procedure allocates an activation record
(instance of Environment, somewhat poorly named)

Activation record stores:
- mapping from names to Values, for each formal and local variable in that scope (environment)
- lexically enclosing activation record (static link)

Method activation record: also
- calling activation record (dynamic link)

Class activation record: also
- methods (to support run-time method lookup)
- instance variable declarations, not values
  - values stored in class instances, i.e., ClassValue
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 activation records have same "shape," described by single symbol table

Example

...  

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