Reminder:
• Nothing covered in lecture or readings from today on will appear on the midterm
• That is, the midterm will cover only front-end issues

Now
• …what to do now that we have this wonderful AST+ST representation
• We’ll look mostly at interpreting it or compiling it
  • But you could also analyze it for program properties
  • Or you could “unparse” it to display aspects of the program on the screen for users
  • …

Analysis
• What kinds of analyses could we perform on the AST+ST representation?
  • The representation is of a complete and legal program in the source language
  • Ex: ensure that all variables are initialized before they are used
  • Some languages define this as part of their semantic checks, but many do not
  • What are some other example analyses?

Implementing a language
• If we want to execute the program from this representation, we have two basic choices
  • Interpret it
  • Compile it (and then run it)
• Tradeoffs between this include
  • Time until the program can be executed (turnaround time)
  • Speed of executing the program
  • Simplicity of the implementation
  • Flexibility of the implementation

Interpreters
• Essentially, an interpreter defines an EVAL loop that executes AST nodes
• To do this, we create data structures to represent the run-time program state
  • Values manipulated by the program
  • An activation record for each called procedure
    – Environment to store local variable bindings
    – Pointer to calling activation record (dynamic link)
    – Pointer to lexically-enclosing activation record (static link)

Pros and cons of interpretation
• Pros
  • Simple conceptually, easy to implement
  • Fast turnaround time
  • Good programming environments
  • Easy to support fancy language features
• Con: 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 the interpreter’s work into two parts
  - Compile-time
  - Run-time
- Compile-time does preprocessing
  - Perform some computations at compile-time only once
  - Produce an equivalent program that gets run many times
- Only advantage over interpreters: faster running programs

Compile-time processing

- Decide on representation and placement of run-time values
  - Registers
  - Format of stack frames
  - Global memory
  - Format of in-memory data structures (e.g., records, arrays)
- Generate machine code to do basic operations
  - Like interpreting, but instead generate code to be executed later
- Do optimization 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</td>
</tr>
<tr>
<td>Scope, symbol table</td>
<td>Environment</td>
</tr>
<tr>
<td>Variable</td>
<td>Memory location, register</td>
</tr>
<tr>
<td>Lexically-enclosed scope</td>
<td>Static link</td>
</tr>
<tr>
<td>Calling procedure</td>
<td>Dynamic link</td>
</tr>
</tbody>
</table>

An interpreter for PL/0

```cpp
class Value {
    public:
        virtual int intValue() {
            ...}
        virtual bool boolValue() {
            ...}
        ...};
class IntegerValue : public Value {
    public:
        bool isInteger() {
            return true;
        }
        int intValue() {
            return _value;
        }
        void print() {
            printf("%d", _value);
        }
    ...};
```

Example eval

```cpp
Value* UnOp::eval(SymTabScope* s, ActivationRecord* ar) {
    Value* arg = _expr->eval(s, ar);
    switch(_op) {
    case MINUS:
        return new IntegerValue(- arg->intValue());
    case ODD:
        return new BooleanValue(arg->intValue() % 2 == 1);
    default:
        Plzero->fatal("unexpected UNOP");
    }
}
```

Activation records

- Each call of a procedure allocated an activation record (instance of ActivationRecord)
  - Basically, equivalent to a stack frame and everything associated with it
- An activation record primarily stores
  - Mapping from names to Values for each formal and local variable in that scope (environment)
    - Don’t store values of constants, since they are in the symbol table
  - Lexically enclosing activation record (static link)
    - Why needed? To find values of non-local variables
Calling procedure

- There must be a logical link from the activation of the calling procedure to the called procedure
- Why? So we can handle returns
- In PL/0, this link is implicit in the call structure of the PL/0 eval functions
- So, when the source program returns from a procedure, the associated PL/0 eval function terminates and returns to the caller
- Some interpreters represent this link explicitly
  - And we will definitely do this in the compiler itself

Activation records & symbol tables

- For each procedure in a program
  - Exactly one symbol table, storing types of names
  - Possibly many activation records, one per call, each storing values of names
- For recursive procedures there can be several activation records for the same procedure on the stack simultaneously
- All activation records for a procedure have the same shape, which is described by the single, shared symbol table

This stuff is important!

- I’ll need some volunteers
  - Symbol tables for M and fact
  - Activation records in executing fact(4)
- So we’ll repeat in here (interpreting)
- And again in compiling