CSE401: Backend (B)

David Notkin
Autumn 2000

Interpreting PL/0

- We're looking at how to take the AST+ST representation and execute it interpretively
- We looked at the basic idea of recursively applying `eval` to the AST
- We looked at activation records and their relationship to symbol tables
- We briefly discussed static links
  - And even more briefly dynamic links

Static linkage

- Connect each activation record to its lexically enclosing activation record
  - This represents the block structure of the program
- When calling a procedure, what activation record to use for the lexically enclosing activation record?

```
module M;
  var x:int;
  proc P(y:int);
    proc Q(y:int);
      begin R(x+y);end Q;
    proc R(z:int);
      begin P(x+y+z);end R;
    begin Q(x+y);end P;
begin
  x := 1;;
  P(2);
end M.
```


Nested procedure semantics: C

- Allow nesting of procedures
- Allow procedures to be passed as regular values, but without referencing variables in the lexically enclosing scope

⇒ Lexically enclosing activation record is always the global scope

Nested procedure semantics: PL/0

- Allow nesting of procedures
- Allow references to variables of lexically enclosing procedures
- Don't allow procedures to be passed around

⇒ Caller can always compute callee's lexically enclosing activation record

Nested procedure semantics: Pascal

- Allow nesting of procedures
- Allow references to variables of lexically enclosing procedures
- Allow procedures to be passed down but not to be returned

⇒ Represent procedure value as a pair of a procedure and an activation record (*closure*)

Nested procedure semantics:
ML, Scheme, Smalltalk

- Fully first-class nestable functions
- Procedures can be returned from their lexically enclosing scope

⇒Put closures and environments in the heap

Example `eval` method for PL/0

some error checking omitted

```
Value* VarRef::eval(SymTabScope* s, ActivationRecord* ar) {
  SymTabEntry* ste = s->lookup(_ident);
  if (ste == NULL) {Plzero->fatal…);}
  if (ste->isConstant()) {
    return ste->value();
  }
  if (ste->isVariable()) {
    ActivationRecordEntry* are = ar->lookup(_ident);
    Value* value = are->value();
    return value;
  }
  Plzero->fatal("referencing identifier that's
                not a constant or variable");
  return NULL;
}
```


Another `eval` method for PL/0

some parts omitted

```
Value* BinOp::eval(SymTabScope* s, ActivationRecord* ar) {
  Value* left = _left->eval(s, ar);
  Value* right = _right->eval(s, ar);

  switch(_op) {
    case PLUS: return new IntegerValue(left->intValue() +
                                      right->intValue());
    …
    case DIVIDE:
      if (right->intValue() == 0) {
        Plzero->evalError("divide by zero", line);
      }
      return new IntegerValue(left->intValue() /
                              right->intValue());
    case LSS: return new BooleanValue(left->intValue() <
                                     right->intValue());
      …}
```


`eval` Assignment Statement

```
void AssignStmt::eval(SymTabScope* s,
                      ActivationRecord* ar) {
  Value*& lhs = _lvalue->eval_address(s, ar);
  Value* rhs  = _expr->eval(s, ar);
  lhs = rhs;
}
```


`eval` while Statement

```
void WhileStmt::eval(SymTabScope* s,
                     ActivationRecord* ar) {
  for (;;) {
    Value* test = _test->eval(s, ar);
    if (test->boolValue()) {
      for (int i = 0; i < _loop_stmts->length(); i++) {
        _loop_stmts->fetch(i)->eval(s, ar);
      }
    } else {
      break;
    }
  }
}
```


Note: recursion

- By now you should understand that recursion is much much more than a cool way to write tiny little procedures in early programming language classes
- If you don't really see this yet, I have a special assignment for you
  - Rewrite either the parser or the interpreter without using recursion
  - Oh, you can do it, for sure…

`eval` declarations

```
void VarDecl::eval(ActivationRecord* ar) {
  for (int i = 0; i < _items->length(); i++) {
    _items->fetch(i)->eval(ar);
  }
}

void VarDeclItem::eval(ActivationRecord* ar) {
  ActivationRecordEntry* varActivationRecordEntry =
    new VarActivationRecordEntry(_name, undefined);
  ar->enter(varActivationRecordEntry);
}
```


`eval` constant declarations

```
void ConstDecl::eval(ActivationRecord* ar) {
  --OK, what goes here?
}
```


`eval` procedure calls

```
void CallStmt::eval(SymTabScope* s, ActivationRecord* ar) {
  ValueArray* argValues = new ValueArray;
  for (int i = 0; i < _args->length(); i++) {
    Value* argValue = _args->fetch(i)->eval(s, ar);
    argValues->add(argValue);
  }
  SymTabEntry* ste = s->lookup(_ident);
  if (ste == NULL) {Plzero->fatal…;}
  ActivationRecord* enclosingAR;
  ActivationRecordEntry* are =
       ar->lookup(_ident, enclosingAR);
  if (are == NULL) {Plzero->fatal…;}
  ProcDecl* callee = are->procedure();
  callee->call(argValues, enclosingAR);
}
```


`eval` procedure calls II

```
void ProcDecl::call(ValueArray* argValues,
                    ActivationRecord* enclosingAR) {
  ActivationRecord* calleeAR =
     new ActivationRecord(enclosingAR);

  for (int i = 0; i < _formals->length(); i++) {
    FormalDecl* formal = _formals->fetch(i);
    Value* actual = argValues->fetch(i);
    formal->bind(actual, calleeAR);
  }
  _block->eval(calleeAR);
}
```


OK, that's most of interpretation

- Next
  - memory layout (data representations, etc.)
  - stack layout, etc.
- Then back to how we compile activation records, etc.
- And generate code, of course