CSE401:
Compilers vs Interpreters

Larry Ruzzo
Spring 2002

Slides by Chambers, Eggers, Notkin, Ruzzo, and others
© W.L. Ruzzo and UW CSE, 1994-2002

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
  - …

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 for basic operations
  - Like interpreting, but instead generate code to be executed later
- Do optimization across instructions if desired

Compile-time vs. run-time

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

Details are coming

An interpreter for PL/0

- Data structure to represent run-time values: `Value` hierarchy
  - Also useful for `resolve_constant`
  - Value-level analogue of `Type`
- Data structure to store `Value`s for each variable
  - `ActivationRecord` containing `ActivationRecordEntries`
  - Run-time analogue of `SymbolTableScope`
- `eval` method per AST class

```
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

```
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 allocates 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 `Value`s 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 a 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

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

- Disallow 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

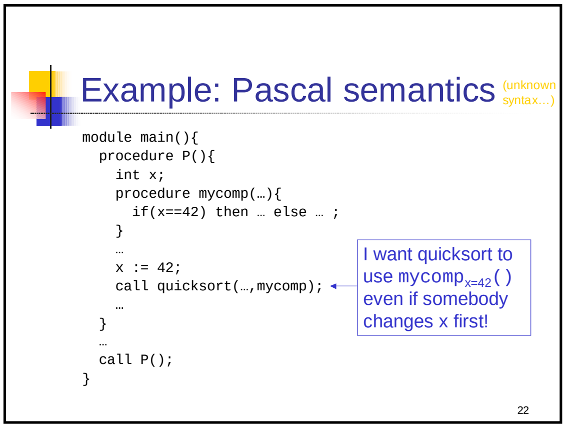
- Allow nesting of procedures
- Allow references to variables of lexically enclosing procedures
- Don't allow procedures to be passed around

⇒ 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*)

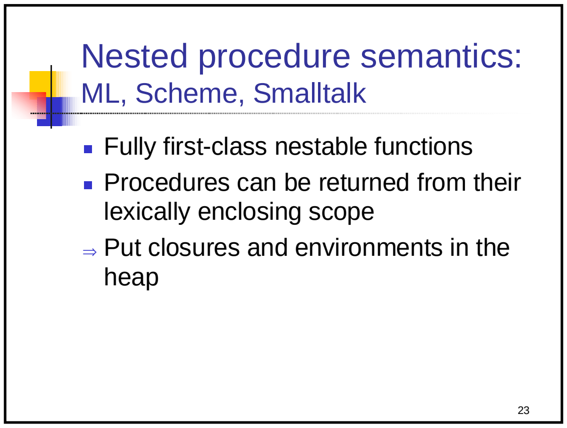


Example: Pascal semantics

(unknown syntax…)

```
module main(){
  procedure P(){
    int x;
    procedure mycomp(…){
      if(x==42) then … else … ;
    }
    …
    x := 42;
    call quicksort(…,mycomp);
    …
  }
  …
  call P();
}
```

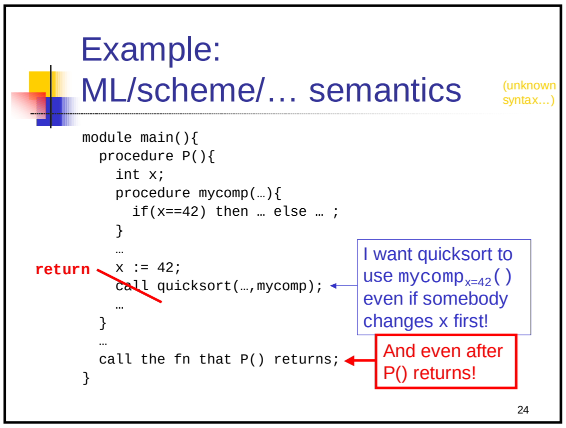
I want quicksort to use `mycomp`$_{x=42}$( ) even if somebody changes x first!



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



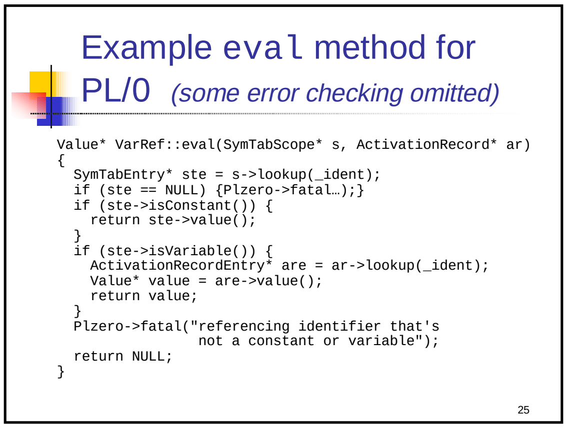
Example: ML/scheme/… semantics

(unknown syntax…)

```
        module main(){
          procedure P(){
            int x;
            procedure mycomp(…){
              if(x==42) then … else … ;
            }
            …
return      x := 42;
            call quicksort(…,mycomp);
            …
          }
          …
          call the fn that P() returns;
        }
```

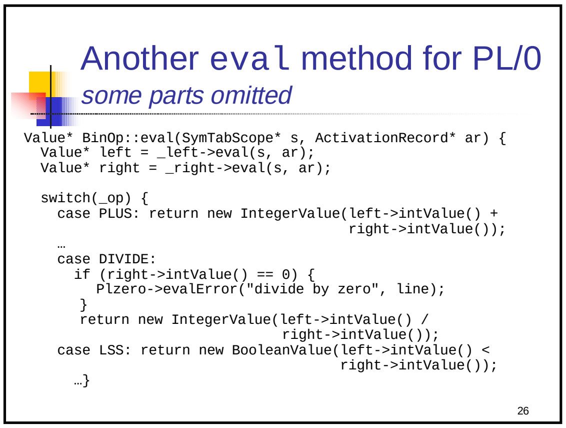
I want quicksort to use `mycomp`$_{x=42}$( ) even if somebody changes x first!

And even after P() returns!



Example `eval` method for PL/0 *(some error checking omitted)*

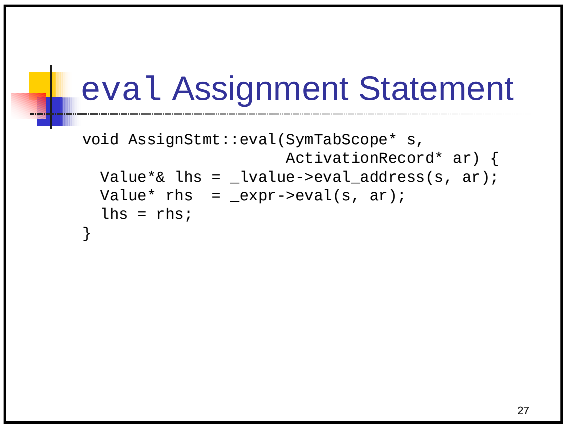
```
Value* VarRef::eval(SymTabScope* s, ActivationRecord* ar)
{
  SymTabEntry* ste = s->lookup(_ident);
  if (ste == NULL) {Plzero->fatal…);}
  if (ste->isConstant()) {
    return ste->value();
  }
  if (ste->isVariable()) {
    ActivationRecordEntry* are = ar->lookup(_ident);
    Value* value = are->value();
    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);
}
```