

CSE401: Compilers vs Interpreters

Andrei Alexandrescu Fall 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
- n 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
 - n ...

2



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
- Mhat are some other example analyses?

3



Implementing a language

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

4



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
 - n An activation record for each called procedure
 - Environment to store local variable bindings
 - n Pointer to calling activation record (dynamic link)
 - n Pointer to lexically-enclosing activation record (static link)

5



Pros and cons of interpretation

- n Pros
 - Simple conceptually, easy to implement
 - Fast turnaround time
 - Good programming environments
 - Easy to support fancy language features
- n Con: slow to execute
 - n Data structure for value vs. direct value
 - ⁿ Variable lookup vs. registers or direct access
 - EVAL overhead vs. direct machine instructions
 - No optimizations across AST nodes



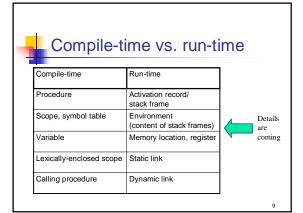
Compilation

- Divide the interpreter's work into two parts
- Compile-time
- Run-time
- n Compile-time does preprocessing
 - Perform some computations at compile-time only
 - 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
 - _n Global memory
 - Format of in-memory data structures (e.g., records, arrays)
- n Generate machine code for basic operations
 - Like interpreting, but instead generate code to be executed later
- Do optimization across instructions if desired





An interpreter for PL/0

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





Example eval

```
Value* UnOp::eval(SymTabScope* s, ActivationRecord* ar)
 Value* arg = _expr->eval(s, ar);
 switch(_op) {
   case MINUS
     return new IntegerValue(- arg->intValue());
       new BooleanValue(arg->intValue()%2 == 1);
      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
- n 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 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
- $_{\rm n}$ In PL/0, this link is implicit in the call structure of the PL/0 $_{\rm eval}$ 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
 - n And we will definitely do this in the compiler itself

13



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

14

module M;
 var res: int;
 procedure
fact(n:int);
 begin
 if n > 0 then
 res := res * n;
 fact(n-1);
 end;
 end fact;
 begin
 res := 1;
 fact(input);
 output := res;
end M.



This stuff is important!

- n So we'll repeat in here (interpreting)
- n And again in compiling

16



Interpreting PL/0

- We're looking at how to take the AST+ST representation and execute it interpretively
- $_{\scriptscriptstyle \rm I\! I\! I}$ We looked at the basic idea of recursively applying $_{\scriptscriptstyle \rm I\! I\! I\! I\! I}$ to the AST
- We looked at activation records and their relationship to symbol tables
- n We briefly discussed static links
 - n And even more briefly dynamic links

17



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

19

Nested procedure semantics:



- 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

20

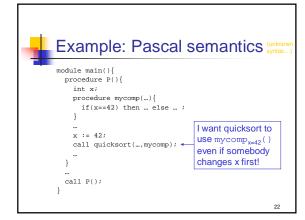
1

Nested procedure semantics:

Pascal

- n 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)

21



-

Nested procedure semantics:

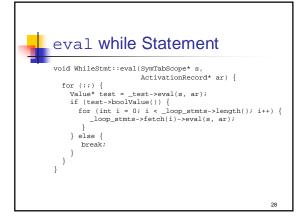
ML, Scheme, Smalltalk

- _n Fully first-class nestable functions
- Procedures can be returned from their lexically enclosing scope
- ⇒ Put closures and environments in the heap

23

Example: ML/scheme/... semantics 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; } And even after P() returns!

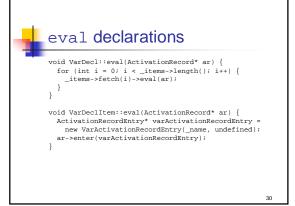
Value* VarRef::eval(SymTabScope* s, ActivationRecord* ar) { SymTabEntry* ste = s->lookup(_ident); if (ste == NULL) {Plzero->fatal...); if (ste->isConstant()); } if (ste->isVariable()) { ActivationRecordEntry* are = ar->lookup(_ident); Value* value = are->value(); return value; } Plzero->fatal(*referencing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

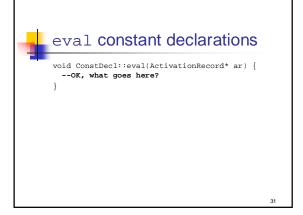


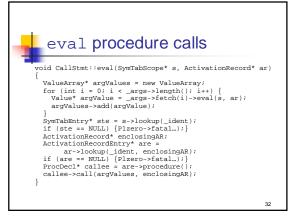


Note: recursion

- 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
 - n Oh, you can do it, for sure...







•

eval procedure calls II

OK, that's most of interpretation

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