

#### Larry Snyder Autumn 2003

Slides by Chambers, Eggers, Notkin, Ruzzo, Snyder and others © L. Snyder and UW CSE, 1994-2003



- n ...what to do now that we have this wonderful AST+ST representation
- <sup>n</sup> We'll look mostly at interpreting it or compiling it
  - But you could also analyze it for program properties
  - n Or you could "unparse" it to display aspects of the program on the screen for users
  - n ...

2



#### **Analysis**

- n 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
- <sub>n</sub> What are some other example analyses?



#### Implementing a language

- 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)
- $_{\rm n}\,$  Tradeoffs between this include
  - Time until the program can be executed (turnaround time)
  - $_{\scriptscriptstyle \rm n}$  Speed of executing the program
  - n Simplicity of the implementation
  - n 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
  - n Values manipulated by the program
  - $_{\scriptscriptstyle \rm n}$  An activation record for each called procedure
    - n Environment to store local variable bindings
    - n Pointer to calling activation record (dynamic link)
    - Pointer to lexically-enclosing activation record (static link)

## Pros and cons of interpretation

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



- n Divide the interpreter's work into two parts
  - n Compile-time
  - n Run-time

Calling procedure

- n 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

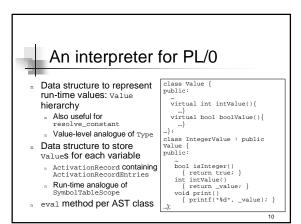


#### Compile-time processing

- Decide on representation and placement of run-time values
  - n Registers
  - n Format of stack frames
  - n Global memory
  - Format of in-memory data structures (e.g., records, arrays)
- n Generate machine code for basic operations
  - $_{\rm n}\,$  Like interpreting, but instead generate code to be executed later
- n Do optimization across instructions if desired

Compile-time vs. run-time Compile-time Run-time Procedure Activation record/ stack frame Scope, symbol table Environment Details (content of stack frames) are Variable Memory location, register coming Lexically-enclosed scope Static link

Dynamic link





#### 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
- $_{\rm n}\,$  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
  - n Lexically enclosing activation record (static link)
    - h 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
  - <sub>n</sub> 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

- n 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
- $_{\rm n}$  We looked at the basic idea of recursively applying  $_{\rm eval}$  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

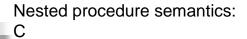
17



#### 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.
```



- 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: PL/0

- n 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

### 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)

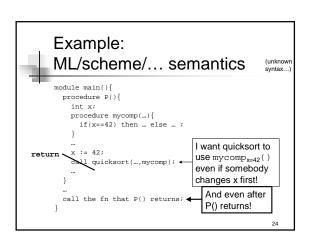
21

# 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<sub>x=42</sub>() even if somebody changes x first!

#### Nested procedure semantics:

ML, Scheme, Smalltalk

- <sub>n</sub> Fully first-class nestable functions
- Procedures can be returned from their lexically enclosing scope
- ⇒ Put closures and environments in the heap



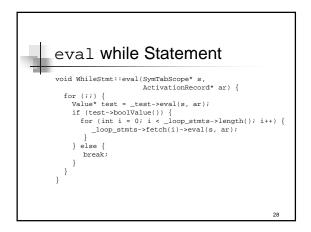
## Example eval method for PL/O (some error checking omitted) Value\* VarRef::eval(SymTabScope\* s, ActivationRecord\* ar) { SymTabEntry\* ste = s->lookup(\_ident); if (ste == NULL) {Plzero->fatal...); if (ste->isConstant()) { return ste->value(); } if (ste->isVariable()) { ActivationRecordEntry\* are = 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 DIVIDE:
    if (right->intValue());
    case DIVIDE:
    if (right->intValue() == 0) {
        Plzero->evalError("divide by zero", line);
    }
    return new IntegerValue(left->intValue());
    case LSS: return new BooleanValue(left->intValue());
    case LSS: return new BooleanValue(left->intValue());
    case LSS: return new BooleanValue(left->intValue());
    case LSS: return new BooleanValue(left->intValue());
```

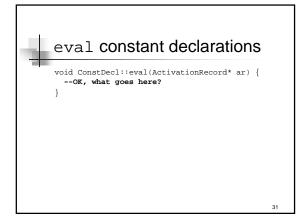


### Note: recursion

- 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
  - n Rewrite either the parser or the interpreter without using recursion
  - n Oh, you can do it, for sure...

29

# 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); }



```
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;
   ActivationRecord*dentry* are =
      ar->lookup(_ident, enclosingAR);
   if (are == NULL) {Plzero->fatal...);
   Procbecl* callee = are->procedure();
   callee->call(argValues, enclosingAR);
}
```

## OK, that's most of interpretation

- $_{\rm n}$  Next
  - n memory layout (data representations, etc.)
  - n stack layout, etc.
- <sup>n</sup> Then back to how we compile activation records, etc.
- n And generate code, of course