Implementing a Language

Given type-checked AST program representation:

- might want to run it
- might want to analyze program properties
- might want to display aspects of program on screen for user
- ...

To run program:

- can interpret AST directly
- can generate target program that is then run recursively

Tradeoffs:

- time till program can be executed (turnaround time)
- speed of executing program
- simplicity of implementation
- flexibility of implementation

Interpreters

Create data structures to represent run-time program state

- values manipulated by program
- 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/environment (**static link**)

EVAL loop executing AST nodes

Pros and cons of interpretation

+ simple conceptually, easy to implement
+ fast turnaround time
+ good programming environments
+ easy to support fancy language features

- 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 interpreter work into two parts:

- compile-time
- run-time

Compile-time does preprocessing

- perform some computations at compile-time once
- produce an equivalent program that gets run many times

Only advantage over interpreters: faster running programs

Compile-time processing

Decide representation of run-time data values

Decide where data will be stored

- registers
- format of stack frames
- global memory
- format of in-memory data structures (e.g. records, arrays)

Generate machine code to do basic operations

- just like interpreting expression,
  except generate code that will evaluate it later

Do optimizations across instructions if desired

Compile time vs. run time

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

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 `Values` for each variable:
`ActivationRecord` holding
`ActivationRecordEntries`

- run-time analogue of SymbolTableScope

`eval` methods for each kind of AST class

Activation records

Each call of a procedure allocates an **activation record** (instance of `ActivationRecord`)

Activation record stores:

- mapping from names to `Values`,
  for each formal and local variable in that scope (**environment**)
  - no need to store values of constants; they're in the symbol table
- lexically enclosing activation record (**static link**)
- need to store entries for procedures, too (more later)

Calling activation record (**dynamic link**) represented implicitly in calling stack of PL/0 `eval` functions

- some interpreters represent this link explicitly,
  in order to implement procedure return

Also use `ActivationRecord` instances for any other scope contents, e.g. the top-level module variables

Activation records vs. 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,
can have several activation records for same procedure
on stack simultaneously

All activation records have same "shape,"
described by single symbol table

Example

```
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.
```


Static linkage

Connect each activation record to its lexically enclosing a.r.

When calling a procedure, what activation record to use for
lexically enclosing a.r.?

- caller's activation record?
- activation record enclosing callee procedure's declaration?
  - how to find activation record of a procedure,
    in face of nesting of procedures in procedures?

```
module M;
   var x:int;
   procedure P(y:int);
      procedure Q(y:int);
         begin R(x+y); end Q;
      procedure 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,
and allow procedures to be passed around as regular values,
but no references to variables in lexically enclosing procs

- lexically enclosing activation record is always global scope

PL/0: allow nesting of procedures,
and references to variables of lexically enclosing procs,
but don't allow procedures to be passed around

- caller can always compute
  callee's lexically enclosing activation record

Pascal: allow nesting of procedures,
and references to variables of lexically enclosing procs,
and allow procedures to be passed down but not returned

- represent procedure value as a pair of
  a procedure and an activation record (a **closure**)

ML, Scheme, Smalltalk: fully first-class nestable functions

- procedures can be returned from their lexically enclosing
  scope ⇒ put closures and environments in heap memory

Example `eval` methods for PL/0

(Omitting internal error handling)

```
Value* IntegerLiteral::eval(
      SymTabScope* s, ActivationRecord* ar) {
  return new IntegerValue(_value);
}

Value* VarRef::eval(
      SymTabScope* s, ActivationRecord* ar) {
  SymTabEntry* ste = s->lookup(_ident);
  if (ste->isConstant()) {
    return ste->value();
  }
  // ste->isVariable()
  ActivationRecordEntry* are =
    ar->lookup(_ident);
  Value* value = are->value();
  if (value->isUndefined()) {
    Plzero->evalError("...");
  }
  return value;
}
```


A more complicated expression

```
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");
    }
    return new IntegerValue(
      left->intValue() / right->intValue());
   case LSS:
    return new BooleanValue(
      left->intValue() <  right->intValue());
   ...
  }
}
```


Some statements

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

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


Some declarations

```
void VarDecl::eval(ActivationRecord* ar) {
  for (int i = 0; i < _items->length(); i++) {
    _items->fetch(i)->eval(ar);
  }
}
void VarDeclItem::eval(ActivationRecord* ar) {
  ar->enter(
    new VarActivationRecordEntry(
      _name, new UndefinedValue));
}

void ConstDecl::eval(ActivationRecord* ar) {
}

void ProcDecl::eval(ActivationRecord* ar) {
   ar->enter(
    new ProcActivationRecordEntry(
      _name, this, ar));
}
```

Procedure calls

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



```
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);
}
```