Where are we?

- Lectures:
  - Overview, lexical, syntactic analysis
  - Next: semantic analysis
- Project:
  - Modify lexer, define EBNF for extended PL/0
  - Next: implement syntactic analysis

Semantic analysis

- Perform final checking of legality of input program
  - Properties not checked by lexical and syntactic checking
    - Ex: type checking, ensuring break statement is in a loop, etc.
  - “Understand” program well enough to do the back-end activity of synthesis
    - Ex: relate declarations to references of particular variable

Symbol tables

- Key data structure
  - Produced (and used) during semantic analysis
  - Used during code generation
  - Stores information about names used in the program
    - Declarations add entries to the symbol table
    - Uses of names lookup appropriate symbol table entry

What information about names?

- Kind of declaration
  - var, const, proc, etc.
- Type
- Value, if it’s a const
- Place allocated in memory, if var
  - Not computed initially, but later on
- Call-by-value or call-by-ref, if formal parameter

Example: a PL/0 DeclList

```plaintext
var x : int;
var q : array[20] of bool;
procedure foo(a : int); begin ... end foo;
const z : int = 10;
```
PL/0 symbol table entries

class SymTabEntry {
  public:
    char* name();
    Type* type();
    virtual bool isConstant();
    virtual bool isVariable();
    virtual bool isFormal();
    virtual bool isProcedure();
    virtual int value(); // constants only
    virtual int offset(SymTabScope* s);
};

More in a lecture or two

SymTab subclasses

class VarSTE : public SymTabEntry { … };
class FormalSTE : public VarSTE { … };
class ConstSTE : public SymTabEntry { … };
class ProcSTE : public SymTabEntry { … };

Nested scopes

procedure foo(x:int, w:int);
  var z:int;
  const y:bool = true;
  procedure bar(x:array[5] of bool);
    var y:int;
    begin
      …
      x[y] := z;
    end;
  end;
  begin
    while z do
      …
      y := x * z;
      …
    end;
    output := x + y;
  end;
}

How to handle nested scopes?

- What happens when the same name is declared in different scopes?
- This is first a question of language design: what is the defined semantics?
- Two standard choices
  - Lexical (static) scoping: use the block structure of the program
  - Do you remember choice #2 from 341?

Lexical/static scoping

- The syntactic (block) structure of the program defines how names are resolved
- Given a name in a block
  - Start looking in that block for a declaration of that name; if found, that’s the declaration
  - Otherwise, look in the next outermost enclosing block; until a block is found with a declaration for that name
  - If it’s not found, then it’s an error

Dynamic scoping
Lexical scoping and symbol tables

- Each scope has its own symbol table
- Logically, for a block-structured program, there is a tree of symbol tables

Tree of symbol tables

```plaintext
procedure foo(int, wint);
  var x:bool;
  const y:bool = true;
  procedure bar[x:array[5] of bool];
  var y:int;
  begin
      x[y] := 2;
  end bar;
begin
  while x do
      var z:int, y:int;
      y := z * x;
  end;
end;

output := x + y;
end foo;
```

Tree ⇒ Stack

- In PL/0 and in many compilers, we don’t want to manage the full tree of symbol tables at all times
- Observation: we process the program scope by scope
  - When we are resolving names in a particular scope, we’ve already processed all enclosing scopes
  - Furthermore, we don’t need any other scopes in the program
- So, instead, we can at any point use a stack to represent the pertinent symbol tables
  - The stack, at a given time, represents the static nesting structure of the program w.r.t. the scope being processed

Nested scope operations

- When we encounter a new scope during semantic analysis
  - Create a new, empty scope
  - Push it on top of symbol table stack
- When encounter a declaration
  - Add entry to the scope on top of the stack
  - Check for duplicates in the scope only (why?)
- When encounter a use
  - Search scopes for declaration, beginning with top of stack
- When exiting a scope
  - Pop top scope off stack

PL/0 symbol table interface

```plaintext
class SymTabScope {
public:
    SymTabScope(SymTabScope* enclosingScope);
    void enter(SymTabEntry* newSymbol);
    SymTabEntry* lookup(char* name);
    SymTabEntry* lookup(char* name,
       SymTabScope*& retScope);
    ...
}
```

Next lecture

- We’ll start looking at the implementation issues in symbol tables
  - For instance, how to efficiently manage references to outer scopes
  - With a particular focus on how PL/0 does it