Lesson 19

• Static Scoping
  – Declarations and Definitions
  – Modules
  – Local symbol tables during compilation
  – Syntax-Directed Translation of three-address code in Scope
  – LeBlanc & Cook data structure and lookup function
Declaration order and use of bindings

• Scope of a binding
  1) In the whole block where it is defined
  2) From the declaration to the end of the block

• Use of binding
  a) Only after declaration
  b) In the scope of declaration

• Many languages use 2–a. Java uses 1–b for methods in a class. Modula uses 1–b also for variables!

• Some combinations produce strange effects: Pascal uses 1) – a).

```plaintext
const N = 10;
...
procedure foo;
const
  M = N; (* static semantic error! *)
var
  A : array [1..M] of integer;
  N : real; (* hiding declaration *)
```

Reported errors: “N used before declaration”
“N is not a constant”
• “Use after declaration” would forbid mutually recursive definitions (procedures, data types)
• The problem is solved distinguishing *declaration* and *definition* of a name, as in C
• **Declaration**: introduces a name
• **Definition**: defines the binding

```c
struct manager;       // Declaration only
struct employee {
    struct manager *boss;
    struct employee *next_employee;
    ...
};
struct manager {
    // Definition
    struct employee *first_employee;
    ...
};
```
Nested Blocks

- In several languages local variables are declared in a block or compound statement
  - At the beginning of the block (Pascal, ADA, …)
  - Anywhere (C/C++, Java, …)
- Blocks can be considered as subroutines that are called where they are defined
- Local variables declared in nested blocks in a single function are all stored in the subroutine frame for that function (most programming languages, e.g. C/C++, Ada, Java)
Out of Scope

• Non-local objects can be hidden by local name-to-object bindings

• The scope is said to have a hole in which the non-local binding is temporarily inactive but not destroyed

• Some languages, like Ada, C++ and Java, use qualifiers or scope resolution operators to access non-local objects that are hidden
  – P1.X in Ada to access variable X of P1
  – ::X to access global variable X in C++
  – this.x or super.x in Java
Out of Scope Example

- P2 is nested in P1
- P1 has a local variable X
- P2 has a local variable X that hides X in P1
- When P2 is called, no extra code is executed to inactivate the binding of X to P1

```pascal
procedure P1;
var X: real;
  procedure P2;
  var X: integer
  begin
    ... (* X of P1 is hidden *)
  end;
begin
  ...
end
```
Modules

• Modules are the main feature of a programming language that supports the construction of large applications
  – Support *information hiding* through *encapsulation*: explicit import and export lists
  – Reduce risks of *name conflicts*; support *integrity of data abstraction*

• Teams of programmers can work on separate modules in a project

• No language support for modules in C and Pascal
  – Modula-2 *modules*, Ada *packages*, C++ *namespaces*
  – Java *packages*
Module Scope

• Scoping: modules encapsulate variables, data types, and subroutines in a package
  – Objects inside are visible to each other
  – Objects inside are not visible outside unless **exported**
  – Objects outside are visible [open scopes], or are not visible inside unless **imported** [closed scopes], or are visible with “qualified name” [selectively open scopes] (eg: **B.x**)  

• A module interface specifies exported variables, data types and subroutines

• The module implementation is compiled separately and implementation details are hidden from the user of the module
Module Types, towards Classes

• Modules as abstraction mechanism: collection of data with operations defined on them (sort of *abstract data type*)

• Various mechanism to get module *instances*:
  – Modules as manager: instance as additional arguments to subroutines (*Modula-2*)
  – Modules as types (*Simula*, *ML*)

• Object-Oriented: Modules (classes) + inheritance

• Many OO languages support a notion of Module (packages) independent from classes
Syntax-directed translation of three-address code with names and scopes

• The three-address code generated by the syntax-directed definitions shown in a previous lesson is simplistic

• It assumes that the names of variables can be resolved by the back-end in global or local variables, which is unrealistic

• We need *local symbol tables* to record global declarations as well as local declarations in procedures, blocks, and records (structs) to resolve names
Implementing Static Scoping

• The language implementation must keep trace of current bindings with suitable data structures:
  – Static scoping: symbol tables at compile time
• **Symbol table** main operations: *insert*, *lookup*
  – because of nested scopes, the compiler must handle several bindings for the same name with LIFO policy
  – new scopes (not LIFO) should be created for records and classes
  – Other operations: *enter_scope*, *leave_scope*
• The symbol table might be needed at runtime for **symbolic debugging**
  – The debugger must resolve names in high-level commands by the user
  – Symbol table are saved in portion of the target program code
Symbol Tables for Scoping

struct S
{
    int a;
    int b;
}
s;

void swap(int& a, int& b)
{
    int t;
    t = a;
    a = b;
    b = t;
}

void somefunc()
{
    ...
    swap(s.a, s.b);
    ...
}
Offset and Width for Runtime Allocation

```c
struct S
{
    int a;
    int b;
} s;

void swap(int& a, int& b)
{
    int t;
    t = a;
    a = b;
    b = t;
}

void somefunc()
{
    ...
    swap(s.a, s.b);
    ...
}
```

The fields `a` and `b` of struct `S` are located at offsets 0 and 4 from the start of `S`.

The width of `S` is 8.

Subroutine frame holds arguments `a` and `b` and local `t` at offsets 0, 4, and 8.

The width of the frame is 12.
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void foo()
{ ...
  swap(s.a, s.b);
  ...
}
Hierarchical Symbol Table Operations

- **mktable**(*previous*) returns a pointer to a new (empty) table that is linked to a previous table in the outer scope.
- **enter**(*table*, *name*, *type*, *offset*) creates a new entry in *table*.
- **addwidth**(*table*, *width*) accumulates the total width of all entries in *table*.
- **enterproc**(*table*, *name*, *newtable*) creates a new entry in *table* for procedure with local scope *newtable*.
- **lookup**(*table*, *name*) returns a pointer to the entry in the table for *name* by following linked tables.
Syntax-Directed Translation: Grammar and Attributes

**Productions**

\[
P \rightarrow D ; S \\
D \rightarrow D ; D \\
| \text{id} : T \\
| \text{proc id} ; D ; S \\
T \rightarrow \text{integer} \\
| \text{real} \\
| \text{array [ num ] of } T \\
| ^{T} \\
| \text{record } D \text{ end} \\
S \rightarrow S ; S \\
| \text{id} := E \\
| \text{call id ( } A \text{ ) }
\]

**Productions (cont’d)**

\[
E \rightarrow E + E \\
| E * E \\
| - E \\
| ( E ) \\
| \text{id} \\
| E ^{E} \\
| & E \\
| E . \text{id}
\]

**Synthesized attributes:**

- **T.type**  pointer to type (ex.: ‘integer’, array(2, ‘real’), pointer(record(Table)), …)
- **T.width** storage width of type (bytes)
- **E.place** name of temp holding value of \( E \)

**Global data to implement scoping:**

- **tblptr** stack of pointers to tables
- **offset** stack of offset values
Syntax-Directed Translation of Declarations in Scope

\[ P \rightarrow \{ t := mktable(nil); push(t, tblptr); push(0, offset) \} \]

\[ D \; ; \; S \]

\[ D \rightarrow \text{id : T} \quad \text{\textit{enter}}(\text{table, name, type, offset}) \]
\[ \{ \text{enter(top(tblptr), id.name, T.type, top(offset))}; \]
\[ \text{top(offset) := top(offset) + T.width} \} \]

\[ D \rightarrow \text{proc id ;} \]
\[ \{ t := mktable(top(tblptr)); push(t, tblptr); push(0, offset) \} \]

\[ D_1 \; ; \; S \]
\[ \{ t := top(tblptr); addwidth(t, top(offset)); \]
\[ \text{pop(tblptr); pop(offset); enterproc(table, name, newtable)} \]
\[ \text{enterproc(top(tblptr), id.name, t) } \} \]

\[ D \rightarrow D_1 \; ; \; D_2 \]
Syntax-Directed Translation of Declarations in Scope (cont’d)

\[
T \rightarrow \text{integer} \quad \{ \text{T.type := ‘integer’; T.width := 4} \}
\]
\[
T \rightarrow \text{real} \quad \{ \text{T.type := ‘real’; T.width := 8} \}
\]
\[
T \rightarrow \text{array [ num ] of } T_1
\quad \{ \text{T.type := array(num.val, } T_1.\text{type);} \\
\quad \quad \quad \text{T.width := num.val * } T_1.\text{width} \}
\]
\[
T \rightarrow ^{\wedge} T_1
\quad \{ \text{T.type := pointer( } T_1.\text{type); T.width := 4} \}
\]
\[
T \rightarrow \text{record}
\quad \{ \text{t := mktable(nil); push(t, tblptr); push(0, offset)} \}
\]
\[
D \text{ end}
\quad \{ \text{T.type := record(top(tblptr)); T.width := top(offset);} \\
\quad \quad \quad \text{addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset)} \}
struct S {
    int a;
    int b;
} s;

void swap(int& a, int& b) {
    int t;
    t = a;
    a = b;
    b = t;
}

void foo() {
    ...
    swap(s.a, s.b);
    ...
}
Syntax-Directed Translation of Statements in Scope

\[ S \rightarrow S ; S \]
\[ S \rightarrow \text{id} \:=\ E \]
\[
\{ \ p := \text{lookup} (\text{top} (\text{tblptr}), \text{id}.\text{name}) ; \\
\text{if} \ p = \text{nil} \text{ then} \\
\quad \text{error} () \\
\text{else if} \ p.\text{level} = 0 \text{ then} \ // \text{global variable} \\
\quad \text{emit} (\text{id}.\text{place} \ ':=\ ' \ E.\text{place}) \\
\text{else} \ // \text{local variable in subroutine frame} \\
\quad \text{emit} (\text{fp}[p.\text{offset}] \ ':=\ ' \ E.\text{place}) \}
\]
Syntax-Directed Translation of Expressions in Scope

\[ E \rightarrow E_1 + E_2 \quad \{ \text{E.place := newtemp();} \]
\[ \quad \text{emit(E.place} \, \text{`} :=` \, E_1\text{.place} \, `+` \, E_2\text{.place}) \} \]

\[ E \rightarrow E_1 \times E_2 \quad \{ \text{E.place := newtemp();} \]
\[ \quad \text{emit(E.place} \, \text{`} :=` \, E_1\text{.place} \, `*` \, E_2\text{.place}) \} \]

\[ E \rightarrow - E_1 \quad \{ \text{E.place := newtemp();} \]
\[ \quad \text{emit(E.place} \, \text{`} :=` \, `uminus` \, E_1\text{.place}) \} \]

\[ E \rightarrow ( E_1 ) \quad \{ \text{E.place := E_1\text{.place}} \} \]
\[ E \rightarrow \text{id} \quad \{ p := \text{lookup(top(tblptr), id.name)}; \]
\[ \quad \text{if } p = \text{nil } \text{then error()} \]
\[ \quad \text{else if } p\text{.level} = 0 \text{ then // global variable} \]
\[ \quad \text{emit(E.place} \, \text{`} :=` \, \text{id\text{.place}}) \]
\[ \quad \text{else // local variable in frame} \]
\[ \quad \text{emit(E.place} \, \text{`} :=` \, \text{fp[p.offset]}) \} \} \]
Syntax-Directed Translation of Expressions in Scope (cont’d)

\[ E \rightarrow E_1^\wedge \ { E.\text{place} := \text{newtemp}(); \}
\quad \text{emit}(E.\text{place} \ := \ ‘\star’ \ E_1.\text{place}) \] 
\[ E \rightarrow \& E_1 \ { E.\text{place} := \text{newtemp}(); \}
\quad \text{emit}(E.\text{place} \ := \ ‘\&’ \ E_1.\text{place}) \] 
\[ E \rightarrow \text{id}_1 \cdot \text{id}_2 \ { p := \text{lookup}(\text{top}(\text{tblptr}), \text{id}_1.\text{name}); \}
\quad \text{if } p = \text{nil} \text{ or } p.\text{type} \neq \text{Trec} \text{ then } \text{error}() \}
\quad \text{else}
\quad \quad q := \text{lookup}(p.\text{type}.\text{table}, \text{id}_2.\text{name});
\quad \quad \text{if } q = \text{nil} \text{ then } \text{error}()
\quad \quad \text{else if } p.\text{level} = 0 \text{ then } \text{global variable}
\quad \quad \quad \text{emit}(E.\text{place} \ := \ ‘\text{id}_1.\text{place}[q.\text{offset}])
\quad \quad \text{else} \text{ local variable in frame}
\quad \quad \quad \text{emit}(E.\text{place} \ := \ ‘\text{fp}[p.\text{offset}+q.\text{offset}]) \} \}
LeBlanc & Cook Symbol Table

In the translation just shown, *lookup* of a name may require traversing all enclosing symbol tables, from the current one to the global one. The LeBlanc & Cook Symbol Table implementation for static scoping uses a *hash table* and a *stack*, instead of a tree of symbol tables. This guarantees more efficient lookups.

- Each scope has a **serial number**
  - Predefined names: 0 (*pervasive*)
  - Global names: 1, and so on

- Names are inserted in a **hash table**, indexed by the name
  - Entries contain symbol name, category, **scope number**, (pointer to) type, …

- **Scope Stack**: contains numbers of the currently visible scopes
  - Entries contain scope number and additional info (closed?, …). They are pushed and popped by the semantic analyzer when entering/leaving a scope

- Look-up of a *name*: scan the entries for *name* in the hash table, and look at the scope number *n*
  - If *n* <> 0 (*not pervasive*), scan the Scope Stack to check if scope *n* is visible
  - Stops at first *closed* scope. Imported/Export entries are pointers.
A Modula2 program

type
  T = record
    F1 : integer;
    F2 : real;
  end;
var V : T;
...
module M;
  export I; import V;
  var I : integer;
  ...
procedure P1 (A1 : real;
              A2t: integer) : real;
begin
  ...
end P1;
...
procedure P2 (A3 : real);
var I : integer;
begin
  ...
  with V do
  ...
  end;
  ...
end P2;
end M;

Figure 3.19
LeBlanc-Cook symbol table for an example program in a language like Modula-2. The scope stack represents the referencing environment of the with statement in procedure P2. For the sake of clarity, the many pointers from type fields to the symbol table entries for integer and real are shown as parenthesized (1)s and (2)s, rather than as arrows.
procedure lookup(name)
    pervasive := best := null
    apply hash function to name to find appropriate chain
    foreach entry e on chain
        if e.name = name -- not something else with same hash value
            if e.scope = 0
                pervasive := e
            else
                foreach scope s on scope stack, top first
                    if s.scope = e.scope
                        best := e -- closer instance
                        exit inner loop
                    elseif best != null and then s.scope = best.scope
                        exit inner loop -- won’t find better
                    if s.closed
                        exit inner loop -- can’t see farther
                if best != null
                    while best is an import or export entry
                        best := best.real entry
                    return best
                elseif pervasive != null
                    return pervasive
            else
                return null -- name not found