Semantic Analysis

Recall:

- Two main problems solved by Semantic Analysis
  - Validation - *Is the program legal and meaningful?*
  - Annotation - *What does the program mean?*

- Two main techniques used to solve these problems
  - Tabulation - *Collection of information into look-up tables for easy access*
  - Simulation - *Simulate ideal execution to determine meaning, using tables to find meanings of symbols*

- Last time we looked at Simulation, using the Symbol Stack and Type Stack - but where is the information needed about symbols and types actually stored?

Tabulation

- Tabulation collects all information about declared symbols in a table for easy lookup when analyzing references to symbols
- The main table used is usually called the Symbol Table
- Some basic rules to make tabulation easier continue to influence the design of most (not all) procedural and OO programming languages - most obviously,
  1. Declaration before use - *a symbol must be declared before it can be used*
  2. Uniqueness of reference - *two different things cannot have the same name in the same scope*
- Symbol Table structures have evolved over the years to adapt to richer and more complex language concepts - this is a good way to understand how they came to be structured as they are

Basic Simple Symbol Table

- Assemblers (early 1950's) were the first language processors to require a symbol table (not a surprise, since they were also the first time that symbolic information was used to produce programs)
- Characteristics of Assembler Programs
  - one global scope which contains all defined symbols
  - no masking or overrides, once a symbol is defined, it is defined
  - limited number of types: bytes, words, characters, strings, labels (addresses) and constants (no user-defined types)
  - limited number of attributes: size (number of bytes), representation (binary, packed decimal, character, ...)
- Original BASIC is another language with these simple characteristics

Basic Simple Symbol Table - Example

- Example of Assembly Language:

```
FOO       .word 0
BAR       .asciz "hello world"
MAIN      mov   #5,FOO
          mov   FOO,R1
          mov   #BAR,R2
```

FOO	word
BAR	string
MAIN	code

- Symbol table can be a simple, unordered list of defined symbols and their attributes
- Finding the attributes of a symbol consists of simply linearly searching the table to find it

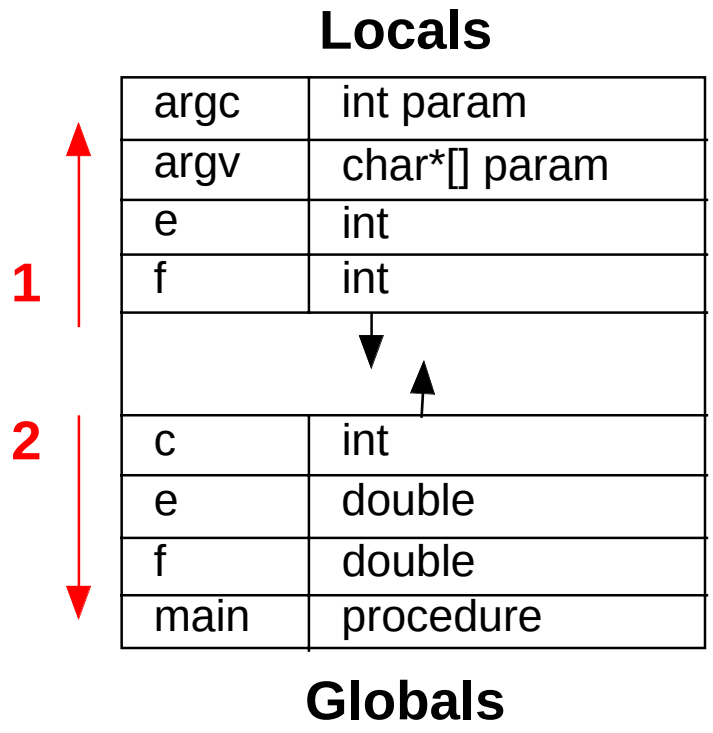
Local Variables - FORTRAN

- FORTRAN (1956 or so) was the first programming language to provide local scopes that divided variables into two kinds:
  - *local* (inside a procedure), and
  - *global* (outside any procedures)
- To provide a symbol table that can handle both local and global variables, two simple tables can be used, one for local symbols, one for globals
- The local table is emptied after each procedure is processed
- FORTRAN still has only a fixed small number of distinct types, plus arrays - no user-defined types - so type information can still be stored dir

Local Variables - Example

- Example Symbol Table for FORTRAN and original C

```
int c;
double e,f;
void main(int argc, char *argv[])
{
  int e,f;
  ...
  e = 42;
  ...
}
```



- Two simple tables serve - one for locals, one for globals
- To find the attributes of a symbol, we look for it first in the local symbol table, and then in the global table
- We can optimize to one table by implementing the local table at one end of the table array and the globals at the other

ALGOL – Nested Scopes

- The first language to have the nested scopes of modern languages like Ada, Pascal, Modula, Turing, Java and so on was ALGOL (1960)
- Nested scopes are static, which means that symbol visibility is determined only by the nesting of blocks in the source program
- Declarations are inherited into nested scopes, but are not visible outside of a symbol's scope of definition

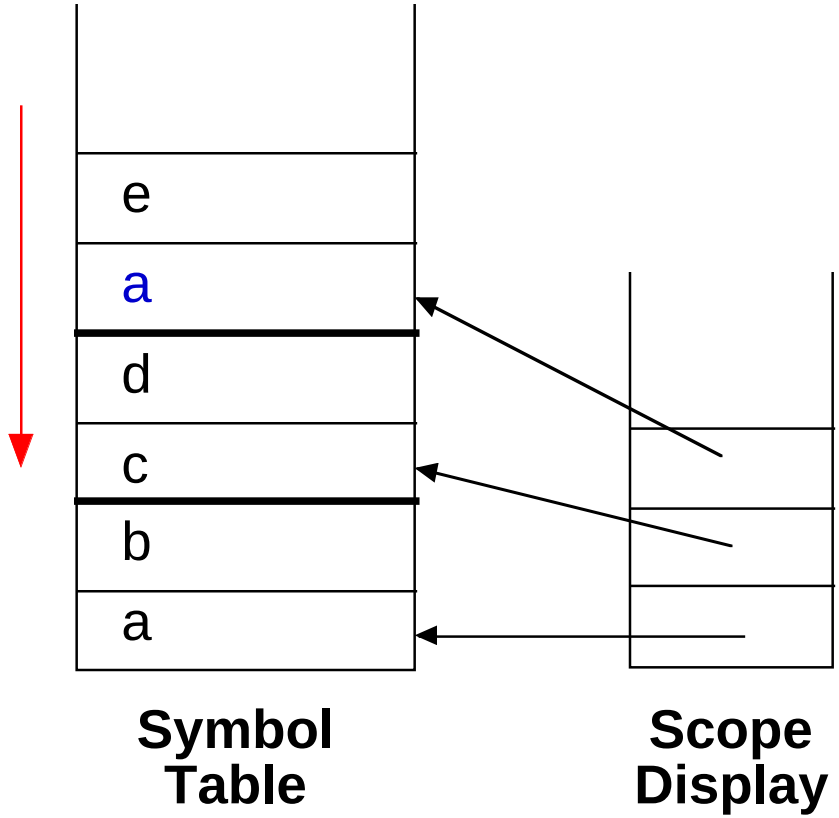
```
var a,b: int
begin
    var c,d: int
    begin
        var a,e : int
        ... here b,c,d,e and nested a visible ...
    end
    ... here a,b,c,d visible (e and nested a not visible) ...
end
... here only a,b visible ...
```

ALGOL – Nested Scopes

- To resolve a reference to a symbol, we start in the scope containing the reference and look for the symbol, then proceed to the next enclosing scope and look there, and so on until we find the symbol or reach the outermost scope
- To model this, we can organize the symbol table into a stack in which we push a new scope onto the stack when we start processing a scope, and pop it when we finish
- This stack model should remind you of the Run Stack and the Display, which after all we designed to fit the semantics of Pascal symbol visibility (and is where that model came from!)

Nested Scopes - Example

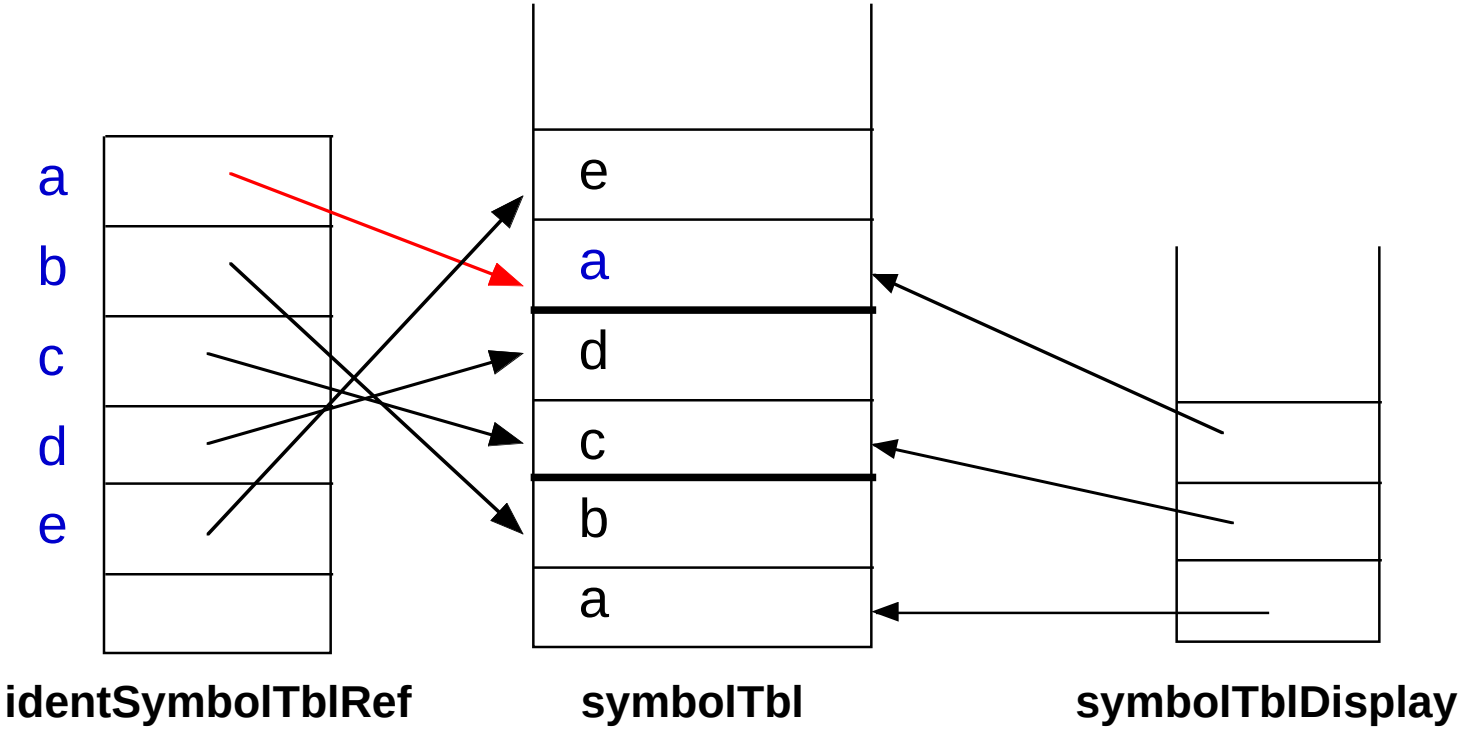
```
var a,b: int
begin
    var c,d: int
    begin
        var a,e : int
          ...
        a := 5
          ...
    end
end
```



- The scopes organize the symbol table into frames - each frame contains the declarations of a scope and is pushed when we start processing the scope, and popped when we finish
- We can find a referenced symbol by searching for it starting at the top of the current table and searching down

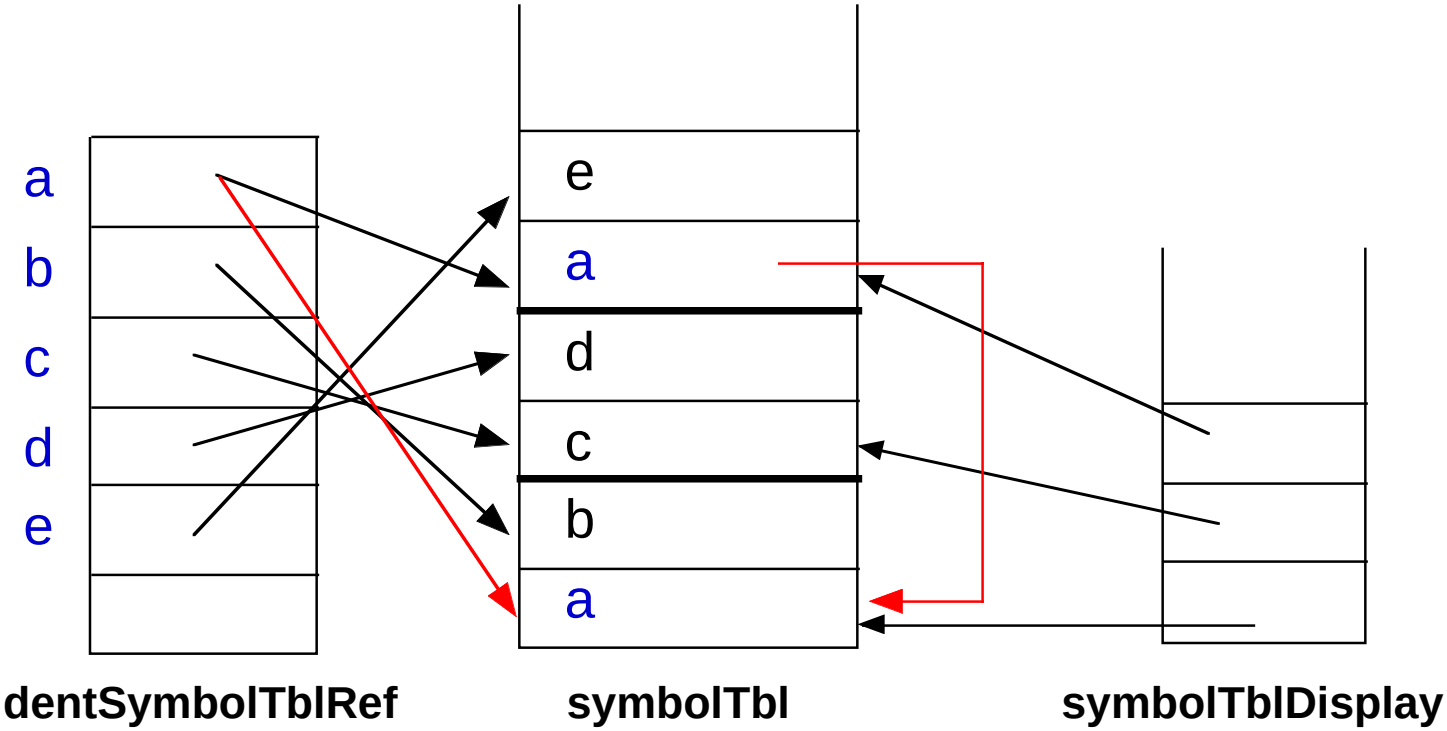
Optimizing Symbol Lookup in PT

- The PT Semantic phase uses exactly the nested symbol table structure for its Symbol Table, but is optimized to avoid lookup of symbols
- Optimization is by a separate array, identSymbolTblRef, indexed by identifier number (assigned by the Scanner/Screener), which references the Symbol Table entry of the currently most local symbol with that identifier



Optimizing Symbol Lookup in PT

- When a new symbol is entered into the table, the reference for its identifier is set in identSymbolTblRef
- The previous reference for that identifier is stored in symbolTblIdentLink, a field of the symbolTbl, so that it can be restored when we pop the scope

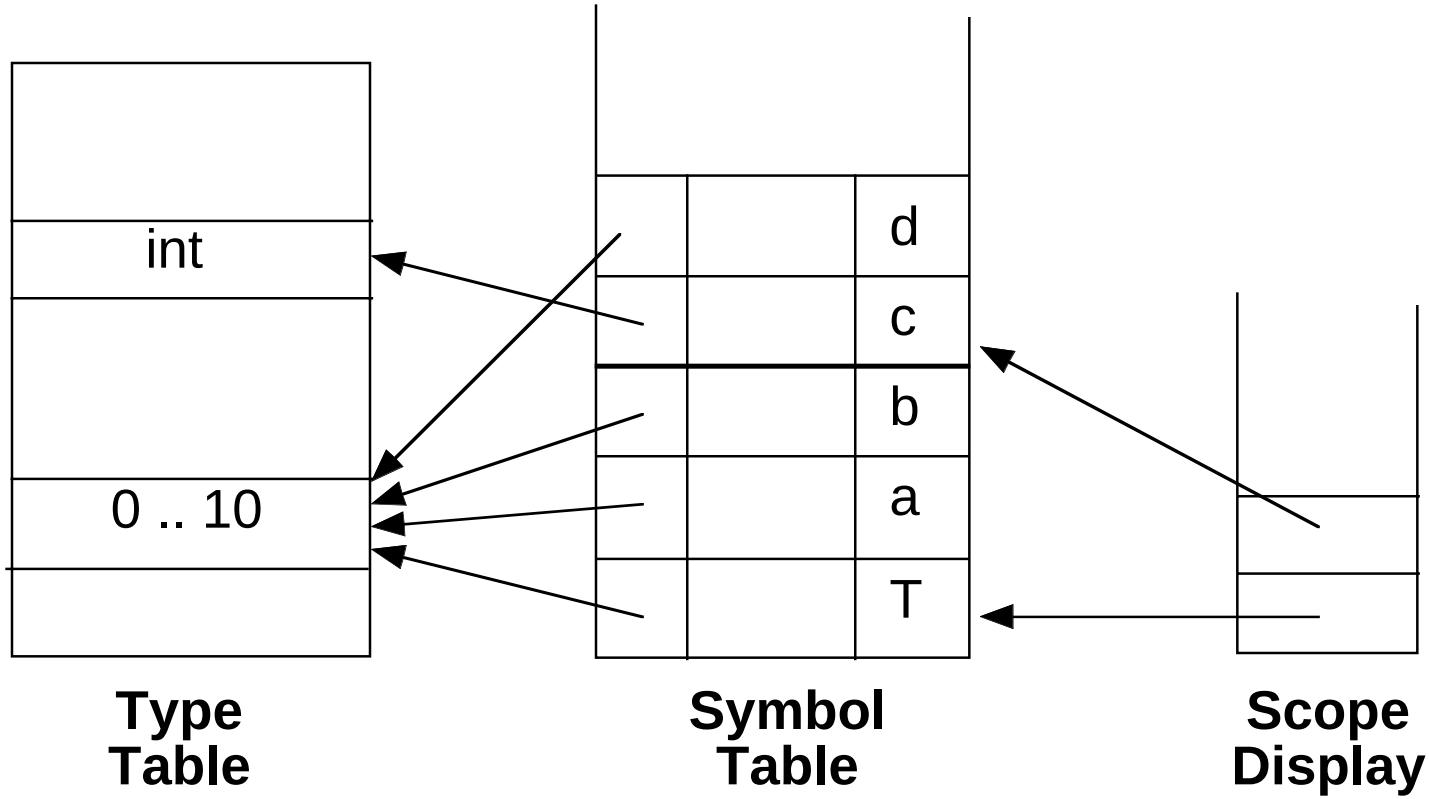


PL/I, Pascal - user defined types

- The introduction of user defined types (e.g. records, subranges, enumerated types, named types, classes) complicates matters
- Instead of a fixed number of known types, the number of different types in the program is now unbounded
- Programmers may introduce as many different types as they want, and each with its own attributes and structure, shared among all variables of the type - thus we can no longer simply store the type attribute of a variable directly in the symbol table
- User defined types do not persist past the semantic phase - they are broken down into their component primitive types for code generation (so that the code generator only has to deal with primitive types)

User defined types – Example

```
type T = 0..10
var a,b: T
begin
    var c: int
    var d: T
    ...        <-- we are here
end
```



Summary

- Symbol Table structures have evolved to reflect the changing semantics of programming languages
- Most modern programming languages designed after ALGOL, using a nested scope structure modelled by a Stack and Display
- A separate Type Table is needed when the language allows users to define their own types
- **Next** :
  - More on tabulation
- **Then** :
  - Implementing the run time model - mapping to real machine structures