Simulation

• Semantic analysis uses a combination of two techniques:

  • **Tabulation**
    Collection of declaration and contextual information into tables for easy lookup

  • **Simulation**
    Calculation of derived attributes by simulating the execution of the program on an ideal machine.

Two basic mechanisms are used to perform the simulation. These are the *Symbol Stack* and the *Type Stack*

Both of these stacks are similar to the run-time Expression Stack, but instead of computing the *value* of expressions, they are used to compute the attributes and types of expressions
The Symbol Stack

• The Symbol Stack mimics the run-time resolution of references to symbols, mirroring the Expression Stack's computation of symbol references such as subscripted variables

• Computes the effective access attributes of the reference, which may be different from the attributes of the declared symbols themselves

Example:
If \( x \) is a variable field of a constant record \( r \), then \( r.x \) is effectively a constant, not a variable (and therefore can't be assigned to)

• Programming languages vary widely in their rules for the effective attributes of complex symbol references - some are simple, some are very complex

• In general, we can say that the rule is that \textit{minimum access} is inherited from left to right in a symbol reference - that is, a field cannot have greater access attributes than its parent
Symbol Stack - Access Calculation Example

Consider the Euclid programming language reference

\[ M.R.X \]

where

- \( M \) is a module (i.e., static class)
- \( R \) is a record variable exported \textbf{readonly} from \( M \) (i.e., is public but not assignable)
- \( X \) is a \textbf{var} field of the record

Minimum inherited access gives us the following:

\[
\begin{array}{ccc}
M & . & R & . & X \\
\text{module} & & \text{readonly} & & \\
\text{readonly} & & \text{readonly} & & \text{var} \\
\text{readonly} & & & & \\
\end{array}
\]
Symbol Stack - Access Calculation Example

- The Symbol Stack records the effective access attributes at each stage as we process the reference.

(Note: the actual references are shown for clarity in the example - the stack does not actually store the reference expressions.)
Symbol Stack - Access Calculation Example 2

\[ F(X).Y \]

where

- \( F \) is a function
- \( X \) is a variable
- \( Y \) is a field of the record (or object) returned by \( F \)

We end up with the following:

\[
\begin{array}{c}
F \quad ( \quad X \quad ) \quad . \quad Y \\
\text{function} \\
\text{var} \\
\text{const} \\
\text{var} \\
\text{const}
\end{array}
\]
(Note: the actual references are shown only for clarity - the stack does not actually store the reference expressions, only their attributes, such as access and memory address)
Symbol Stack – Access Calculation

• Minimum access inheritance is a general rule, and applies well to OO languages such as C++ and Java as well as to modular languages like Turing and Ada

• For example, access permissions of member variables generally follow minimum access inheritance rules - a protected variable of a superclass may not be exported as a public variable by the subclass

• However, minimum access does not apply in all cases

  e.g., If $p$ is a pointer in C, then $*p$, the variable pointed at by $p$, is an assignable variable even if $p$ itself is a constant pointer
Symbol Stack - Declaration Processing

• The Symbol Stack is also used to accumulate declaration information for a symbol until it is complete and can be entered in the Symbol Table.

• Example:

```pascal
var R : record
    var A : integer
    var B : integer
end record
```

```
R var rec 0

A var
R var rec 0
```

```
B var
A var
R var rec 0
```

```
R var rec 2
```
Symbol Stack - Declaration Processing

- Since types and initial values can be expressions, it's possible for the Symbol Stack to be used for both purposes at once.
- Example:

  \[
  \text{var } A : M.T := F(X)
  \]
Symbol Stack - Declaration Processing

- Conceptually, the Symbol Stack has two parts -
  - the *Symbol Declaration Stack* (SDS), which processes declarations
  - the *Symbol Expression Stack* (SES), which processes references

- They don't interfere with one another since we can prove that the SES is always empty when we need to push or pop the SDS

<table>
<thead>
<tr>
<th></th>
<th>SDS</th>
<th>SES</th>
</tr>
</thead>
<tbody>
<tr>
<td>A var</td>
<td>A var</td>
<td></td>
</tr>
<tr>
<td>A var</td>
<td>A var</td>
<td>M var</td>
</tr>
<tr>
<td>X var</td>
<td>F var</td>
<td>F var</td>
</tr>
<tr>
<td>F var</td>
<td>A var</td>
<td>A var</td>
</tr>
</tbody>
</table>
Symbol Declaration – Example

module M 1
  var R : record 2
    var X : N.T 3,4,5,6,7
  ... 8
  end record
end module

where N is a previously declared module exporting type T

1. push new entry for M (SDS)
2. push new entry for R (SDS)
3. push new entry for X (SDS)
4. push reference to module N (SES)
5. push reference to type T (SES)
6. resolve reference to N.T (SES)
7. Enter X into symbol table as type N.T (SES/SDS)
8. Enter R into symbol table with record type (SDS).
Symbol Declaration – Example

M module

R var

M module

X var

R var

M module

N module

T type

N module

X var

R var

N.T type

X var

R var

M module

N.T module

R var

M module
Summary

- **Simulation** in Semantic Analysis
  - We simulate execution like an ES, except compute types and attributes rather than values
  - The **Symbol Stack** computes attributes of symbol declarations and references
  - Because each use (declaration, expression) nests with respect to the other, this stack can be used for both purposes at once

- **Next time**
  - The **Type Stack**, Semantic Mechanisms