Semantic Analysis

- Scanner analyzes input as sequence of tokens
- Parser analyzes sequence of tokens according to grammar
  - Just syntax analysis, looking at structure
  - Result is abstract syntax tree
- Semantic analysis checks syntax tree for meaning
  - Static analysis of program – no execution
  - Stores/generates information needed for code generation

Semantic Information

- Symbol Table
  - Needed if variables must be declared
  - Resolves all identifiers
- Type Checking
  - Static analysis of types used in expressions, statements, etc.
  - Add type information to symbol table
Designing the Semantic Analysis

- Scope rules
  - Nested scopes, redefinition, affect symbol table structure
  - Binding times
- Type Rules
  - Allowed types
  - Type requirements on expressions, statements
  - User defined types
  - Type equivalence

Methodology

- Attributes
  - Computable properties of language constructs
  - Used to satisfy language requirements or for code generation
  - Describe attribute computation with equations or algorithms
    - Attribute computation associated with grammar: rules and/or nodes in syntax tree
  - Determine order of attribute computation
Attribute Grammars

- Formal attribute equations
  - Rules for semantics, like rules for syntax
  - Much more complicated than syntax
  - Too complex for automation, but still useful as a design methodology
- Attributes associated with grammar symbols
  - Numeric value, type, readonly, lvalue, ...
- Attributes determined by grammar rules
  - Function arguments, types in assignment, ...
  - Values of attributes for symbols in grammar rules are related (syntax directed semantics)

Attribute Equations (Semantic Rules)

- Attribute of a symbol in a grammar rule is related to attributes of other symbols in the rule
  - Since symbols can appear multiple times in a grammar rule, they must be indexed to distinguish
  - All attributes of all symbols may be related
- General form:
  Grammar Rule: $X_0 \rightarrow X_1X_2 \ldots X_n$
  Attribute equations: $X_i.a_j = f_{ij}(X_0.a_1, \ldots X_n.a_k, \ldots X_{i-1}.a_k)$
**Attribute Grammar Example**

Grammar:

\[
\begin{align*}
\text{exp} & \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term} \\
\text{term} & \rightarrow \text{term} \ast \text{factor} \mid \text{factor} \\
\text{factor} & \rightarrow ( \text{exp} ) \mid \text{number}
\end{align*}
\]

Attribute Grammar for numerical value

<table>
<thead>
<tr>
<th>GRAMMAR RULE</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{exp}_1 \rightarrow \text{exp}_2 + \text{term}</td>
<td>\text{exp}_1.val = \text{exp}_2.val + \text{term}.val</td>
</tr>
<tr>
<td>\text{exp}_1 \rightarrow \text{exp}_2 - \text{term}</td>
<td>\text{exp}_1.val = \text{exp}_2.val - \text{term}.val</td>
</tr>
<tr>
<td>\text{exp} \rightarrow \text{term}</td>
<td>\text{exp}.val = \text{term}.val</td>
</tr>
<tr>
<td>\text{term}_1 \rightarrow \text{term}_2 \ast \text{factor}</td>
<td>\text{term}_1.val = \text{term}_2.val \ast \text{factor}.val</td>
</tr>
<tr>
<td>\text{term} \rightarrow \text{factor}</td>
<td>\text{term}.val = \text{factor}.val</td>
</tr>
<tr>
<td>\text{factor} \rightarrow ( \text{exp} )</td>
<td>\text{factor}.val = \text{exp}.val</td>
</tr>
<tr>
<td>\text{factor} \rightarrow \text{number}</td>
<td>\text{factor}.val = \text{number}.val</td>
</tr>
</tbody>
</table>

---

**Example Notes**

- These attribute rules look a lot like grammar rules
  - E.g., expression value determined from operands
  - Suited for bottom up computation
  - Such an attribute is **synthesized**
- Number tokens get value from lexeme
  - Essentially a precomputed value
- Attribute equations could annotate the parse tree
  - The syntax tree itself is a synthesized attribute
Another Example

Grammar: \( \text{decl} \rightarrow \text{type} \ \text{var-list} \)
\( \text{type} \rightarrow \text{int} \mid \text{float} \)
\( \text{var-list} \rightarrow \text{id} \mid \text{var-list} \ \text{id} \)

Attribute Grammar for data type (e.g., int or float)

<table>
<thead>
<tr>
<th>GRAMMAR RULE</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{decl} \rightarrow \text{type} \ \text{var-list} )</td>
<td>( \text{var-list}.\text{dtype} = \text{type}.\text{dtype} )</td>
</tr>
<tr>
<td>( \text{type} \rightarrow \text{int} )</td>
<td>( \text{type}.\text{dtype} = \text{integer} )</td>
</tr>
<tr>
<td>( \text{type} \rightarrow \text{float} )</td>
<td>( \text{type}.\text{dtype} = \text{real} )</td>
</tr>
<tr>
<td>( \text{var-list} \rightarrow \text{id} \mid \text{var-list} \ \text{id} )</td>
<td>( \text{id}.\text{dtype} = \text{var-list}.\text{dtype} )</td>
</tr>
<tr>
<td>( \text{var-list} \rightarrow \text{id} )</td>
<td>( \text{var-list}.\text{dtype} = \text{var-list}.\text{dtype} )</td>
</tr>
<tr>
<td>( \text{var-list} \rightarrow \text{id} \mid \text{var-list} \ \text{id} )</td>
<td>( \text{id}.\text{dtype} = \text{var-list}.\text{dtype} )</td>
</tr>
</tbody>
</table>

Example Notes

- Terminal keywords have inherent attribute
  - Non-terminal \text{type} \textit{inherits} this attribute
  - Not suited for bottom up calculation
  - Attributes flow down the tree
- \text{var-list} gets attribute from declaration rule
- \text{dtype} propagated to identifiers
Dependencies

- Attribute rules imply dependencies
  - Must compute attributes in order consistent with dependencies
- Attributes of children computed before parent for synthesized attributes
  - Post order traversal of tree always sufficient
- Inherited attributes may require different orders
  - Pre-order if attributes flow down
  - But may depend on siblings in tree

Data type dependencies (by grammar rule):

\[ decl \rightarrow type \ var-list: \]
\[ var-list.dtype = type.dtype \]
\[ \]
\[ var-list \rightarrow id, \ var-list: \]
\[ id.dtype = var-list_1.dtype \]
\[ var-list_2.dtype = var-list_1.dtype \]
\[ \]
\[ decl \rightarrow type \ var-list: \]
\[ var-list.dtype = type.dtype \]
\[ \]
\[ var-list \rightarrow id, \ var-list: \]
\[ id.dtype = var-list_1.dtype \]
\[ var-list_2.dtype = var-list_1.dtype \]
Inherited Attribute Dependencies

Three basic mechanisms

(a) Inheritance from parent to siblings

(b) Inheritance from sibling to sibling via the parent

(c) Sibling inheritance via sibling pointers

Terminology

- Grammar is **S-attributed** if all attributes are synthesized
  - All dependencies point from child to parent
- Grammar is **L-attributed** if all inherited attributes for a symbol only depend on attributes of symbols to left in a grammar rule
  - Dependencies point down or to right
Implementation Patterns

- Inherited attributes passed as parameters
  - From parent nodes to children nodes
  - From sibling to sibling
  - L-attributed possible during top down parse
- Synthesized attributes as return values
  - Passed up the tree in postorder bottom up traversal
  - S-attributed suited to bottom up parse or post order traversal

Symbol Table

- Major data structure (after syntax tree)
  - Used to implement scoping rules
  - Used to check use against declaration
  - Critical for type checking
- Keeps track of all identifiers
  - Variables, types, functions
- Built after syntax tree created
  - But might be built during parse (or even scanning)
Symbol Table Operations

- **Insert**
  - Add a symbol, e.g., when declaration seen

- **Lookup**
  - Find a symbol in table, e.g., when variable used in expression
  - Must take into account scoping rules, redefinition rules, etc.

- **Delete**
  - Remove entry from table, e.g., at end of scope

Symbol Table Concerns

- **Efficiency**
  - Frequent checks during semantic analysis
  - Economical storage
  - Lookup must be fast
  - Various strategies

- **Dynamic**
  - Changes frequently during semantic analysis

- **Complications**
  - Namespaces (Java, C++)
  - Class scope, e.g., data members and methods
  - Overloading
  - Forward references (classes in C++, Java)
Symbol Table Lookup

- Linear search may be too slow for real compilers
- Use hash table
  - Fast algorithm to map names to small space
  - Must operate in constant time
- Hashing
  - Must distribute evenly, especially taking typical variable naming conventions into account (common prefixes, etc.)
  - Example: sum of ASCII char values mod a prime
  - Should take character position into account, too
- Collision Resolution
  - Open addressing – put collision in a successive bucket
  - Separate chaining – each bucket is a stack, so fast

Symbol Table Contents

- Attributes
  - Data type, scope, size, location, …
  - Could add these to declaration node in syntax tree and store pointer to node in symbol table
- Scope
  - Entry added to table at declaration point, removed at end of block
  - Possible implementation: symbol table as stack of scope contexts, each a small symbol table
  - Could also use nesting level, or block names
Some Scope Considerations

- Function parameters – separate scope
  - More obvious in syntax of old classic C:
    ```
    void foo(x, y)
    int x, y;
    { }
    ```
- Are redefinitions allowed in nested blocks?
  - Not in Java
  - Type cannot be changed in C/C++
- Can type and variable names overlap?

More Scope Considerations

- Recursive definitions
  - Early insertion into table needed so symbol is valid when used later
  - But C/C++ only allow pointers since definition is incomplete
- Forward references
  - C++ constructor uses data before definition
  - Requires two passes – variables, then methods
**TINY Symbol Table**

- All identifiers global – no scope issues!
  - No deletion
- No declaration – insertion if lookup fails
- Very little information stored
  - All types are int, so no need for type
  - Just location stored (relative offset)
  - Could store line number where defined

**Type Checking**

- Static type checking – main activity of semantic analysis
- Check for consistent use of types, e.g.,
  - Operands for operations
  - Compatibility of combined types
  - Function interfaces
  - Member of structures
- Any language constructs that involve types
Simple Types

- Primitive types of the language
  - Predefined by the language specification
  - int, char, double, bool, void
  - Characteristics are part of language definition
- Language also defines compatibility rules
  - In C, "smaller" integers (e.g., short, char) may be used when larger expected
  - Integers may be used when double expected
  - Need to know type of result
    - $x + y$ is double if $x$ is int and $y$ is double

Type Constructors

- Language mechanisms to create new types
  - Enum, arrays, pointers, structures, unions, functions, inheritance
  - Typedef in C – really just alias of type names
- What information must be stored by compiler?
- What are rules for comparing types?
Arrays

- Array declaration constructs new type from a base type and an integer size
- Should size be stored by compiler?
  - Should size be stored with array itself?
- Array equivalence
  - Assignment permitted?
  - Conversions? (e.g., convert array of ints to array of doubles)
- Multi-dimensional arrays

Structures

- Generalization of arrays – collections of possibly different types
  - C/C++/Java use field names instead of offsets
- Type name for the structure type
- Scope issue – are field names in global namespace? Separate name space from variables? Types?
- Type equivalence
  - Structural equivalence – same fields in same order
  - Recursive algorithm to compute
  - What do C/C++/Java do for structures?
### Recursive Types

- Can structure (or class) contain itself?
  - In C/C++, no, but can contain a pointer
  - In Java, yes (because objects are implicit pointers in Java)
  - In ML, recursive types allowed
- In C, why not allow a struct to contain itself?
  - We can never compute the size of the struct, and so cannot produce static data layout

### Functions

- Lots of types involved
  - Parameter types
  - Return type
  - C++: class of method, also constant or not
- Other characteristics
  - Name (what about overloading?)
  - Scope: static functions, static class methods
  - C++: dynamic binding (always the case in Java)
Type Equivalence

- Structural equivalence
  - Same fields in same order
- Declaration equivalence
  - Came from same original type definition
- Name equivalence
  - Same type name

Equivalence Example

- C Syntax:
  
  ```c
  struct A {};
  typedef struct A A;
  typedef struct {} B;
  struct A x; A y; B z;
  ```

- x, y, z all structurally equivalent
- x, y declaration equivalent, but z is not declaration equivalent to these
- none are name equivalent
Type Equivalence in C

- Combination of structural and declaration equivalence
  - `struct` and `union` use declaration equivalence
  - arrays, pointers, and functions use structural equivalence
  - enums are just int's, so structural equivalence
- In C++, enums are internally typedef'ed, and declaration equivalence is used

Implementation Issues

- Type is a synthesized attribute, so could do bottom up traversal of tree
- In TINY, each ASTnode subclass has its own typecheck method
  - Used to traverse tree
  - For expressions, pass the required type, return the expression type
  - Annotate the ASTnode with the type
- Build the symbol table as we go
Other Issues

- Type definitions and names
  - May need type table – or could add to symbol table
  - Pre-populate with primitive types
- Decide on type equivalence
- Determine where implicit conversions occur