Semantic Analysis

- Semantics
  - American Heritage - *The study of relationships between signs and symbols and what they represent*
  - Webster - *the meaning or relationship of meanings of a sign or set of signs*
- What then is the relationship between syntax and semantics?

- Syntax defines the legal ordering of tokens
- Semantics is the meaning of the tokens

- Is the rule that a variable must be declared before used a syntax or semantics requirement?

Semantic Analysis

- This chapter is entitled Semantic Analysis
  - However, be aware that there are parts that could be considered syntax also
- Context
  - So, we have created an AST
    - We have captured quite a bit of understanding of the meaning of the language – e.g., operator precedence
  - Still have some limitations for CFGs we need to address
    - Declare before use, function parameters
  - We also need to capture the types of the operations required
    - If I have a floating point and integer variable I need to add, what is the type of the result?
    - What type conversions will be required?
    - Are there illegal type conversions implied?

Semantic Analysis

- Semantic analysis cleans up issues like this, completing the analysis phase of compilation, in preparation for the synthesis phase
  - Following semantic analysis, we should have fully captured the meaning of the source program
- An awful lot of research has gone into parser construction
- Semantic analysis is much less developed
  - No theory as developed as CFG or methods of expressing like BNF
  - No tools like YACC/LEX
  - One method we will look at is attribute grammars
    - Just to describe, not to automate (although some have tried
  - Frequently done ad hoc

Semantic Analysis

- Semantic analysis can be done in the same pass as parsing, or done in a subsequent pass
  - Easier if done in separate pass
  - Memory and processing speed now make multi-pass compilers practical, so this is probably the preferred method

- Attribute Grammars
  - An attribute is simply a property of a programming language construct
    - You can think of it as an instance variable of one of your AST classes
    - Examples: line number, data type, value of expression, location in memory, register assigned, assembly code associated with node

Attribute Grammars

- Attributes can be computed and assigned a value either during compilation or at runtime
  - We call this static binding or dynamic binding
    - Most languages you are familiar with (C, Pascal, Java) are statically typed and use static binding

- Types of things a C compiler might do
  - Type checking – assign a type attribute
    - This is what we focus on this chapter
  - Optimizations, such as constant folding – assign value attribute
  - Low-level or assembly code – associate with AST
  - Register assignment

Attribute Grammars

- In syntax-directed semantics, attributes are associated directly with the grammar
  - If I have production A → B C, I can reference an attribute of A as A.type
    - For example, I might say A.type = B.type
  - Two options
    - I can write this code straight into my YACC-type file, and compute attributes during parsing
    - Since the AST is almost a direct implementation of the grammar, I can associate attributes with the grammar, and use syntax-directed semantics to guide the development of a subsequent pass

Attribute Grammars

```
num -> num digit | digit
digit -> [0..9]
```

- Example: unsigned numbers
  - Computing a value attribute
  - Note: this is usually done in scanner, not the parser
  - You may have learned in DLD the trick for computing the value of a number when reading it one digit at a time
    - When you get a new digit, you multiply the present value by 10, and then add the new digit
  - For num → digit, the overall value of the number is simply the value of the digit
    - num.val = digit.val
  - For num -> num digit, the overall value is the old value * 10 plus the new value
    - When the same grammar symbol is repeated in a production, we subscript to differentiated
    - $num_1.val = num_2.val * 10 + digit.val$

Attribute Grammars

- Example: unsigned numbers

```
num -> num digit | digit
digit -> [0..9]
```

  - The productions on digit are simple
    - digit → 0 has an attribute digit.val = 0
- Can now make a table of grammar rules and attributes, which we call the attribute grammar

Grammar Rule	Semantic Rules
num → num digit	num_1.val = num_2.val *10 + digit.val
num → digit	num.val = digit.val
digit → 0	digit.val = 0
digit → 1	digit.val = 1 …

Attribute Grammars

- Example: arithmetic expressions
  - Again, we want to compute value
  - What do parse tree and AST look like?

expr → expr + term | expr – term | term
term → term * factor | factor
factor → (expr) | num

Grammar rule	Semantic Rule
$expr_1 \rightarrow expr_2 + term$	$expr_1.val = expr_2.val + term.val$
$expr_1 \rightarrow expr_2 - term$	$expr_1.val = expr_2.val - term.val$
expr → term	expr.val = term.val
$term_1 \rightarrow term_2 * factor$	$term_1.val = term_2.val * factor.val$
term → factor	term.val = factor.val
factor → (expr)	factor.val = expr.val
factor → num	factor.val = num.val

Attribute Grammars

- Example: declarations

```
decl → type var_list
type → int | float
var_list → id , var_list | id
```

- Note: data can move either up or down the AST

Grammar rule	Semantic rules
decl → type var_list	var_list.dtype = type.dtype
type → int	type.dtype = integer
type → float	type.dtype = float
var_list_1 → id, var_list_2	id.dtype = var_list_1.dtype $varlist_2$.dtype = var_list_1.dtype
var_list → id	id.dtype = var_list.dtype

Attribute Grammars

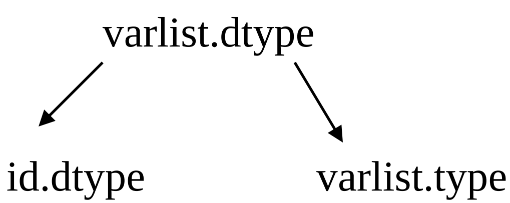
- In the previous examples, we showed operations (+, -, *) on attributes
  - We call these operations the metalanguage of the attribute grammar
  - What operations can you put? How complex can the metalanguage be?
    - Often if-then-else structures are used
    - Can do switch if necessary
    - Can even use functions if helpful
  - Idea is to clearly communicate how attributes are computed from other attributes

Attribute Grammars

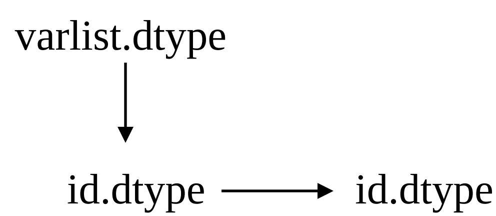
- Notice that, when setting attributes, sometimes data flows up the tree and sometimes down
  - Sometimes data flows to siblings

- The flow of data can be visualized using a dependency graph

$varlist_1 \rightarrow id, varlist_2$	$id.dtype = varlist_1.dtype$ $varlist_2.dtype = varlist_1.dtype$



Parse Tree



AST

Attribute Grammars

- If data is flowing up from children to parent, we say this attribute is synthesized
  - For a walk of the AST, corresponds to a post-order walk
    - Visit your children, then compute your attribute based on their attributes
  - Example: If doing calculator functions, the value attribute

- If data is flowing down from parent to children, these are inherited attributes
  - Inherited also applied where data is passed to siblings
  - Corresponds to a pre-order walk
    - Calculate your attribute value, and then pass it down to child (note: child can't access your attribute if it doesn't have parent pointer, so pass as argument)

Attribute Grammars

- Typically, the values of attributes are stored as an instance variable of each node
  - Example: register allocation – if you assign a temporary register value to the result of a binary operation (like Add), easiest to just store it in BinaryExpression node rather than trying to keep track of it externally
  - However, in some cases it may not be required to store value at the nodes
    - If just doing a calculator, intermediate values may not be interesting
  - Sometimes it's not feasible to store information at the node
    - Most common example is the symbol table
      - Used to store data type information
      - Info generated by declarations, and then needed when the variable is used
        - Variable uses don't have pointers to declaration

Attribute Grammars

- We said that attributes can be computed either during parsing, or as a separate pass
  - Processing speed and memory limitations led researchers to explore computing attributes during parsing
  - Because technology has increased so rapidly, it is feasible, and easier, to compute in a separate pass

- Parsing is limited in the attributes it can compute
  - Can handle synthesized attributes
  - Cannot handle right-to-left passing of attributes
    - Parsers scan left-to-right
  - LR parsers have trouble doing inherited attributes – why?

- Rest of chapter will assume separate pass following parsing

Symbol Table

- The major task during semantic analysis is type checking
  - Actually, it is two separate tasks
    - Type checking – if a type legal the way it is used
      - Correct parameter types, assigning float into int var
    - Type inference (if adding float and int, what is type of result?)
  - Typically these two tasks lumped under the category: type checking

- Three primary data structures support this
  - Symbol table
  - Type attributes in each node
  - A data structure for specifying types
    - int **p,   int [][] getArray()
- We will look first at the symbol table

int x;

=x;

Symbol Table

- A symbol table is simply a Map data structure
  - What's a map?
    - What are its methods?
  - What is the best way to implement a map?

- An unordered map supports search
  - Answers “Does A exist?” well
  - Not good at “What is value closest to A?”
  - Primary functions are add, remove, and find

- For most applications, a hash table is the best solution
  - This is the approach used most widely in compilers
  - Specifically, the separate chaining (linked list) version

Symbol Table

- We won't say a lot about the hash table
  - Hopefully you remember how they work
  - Issues
    - How large do you make the bucket array?
      - Probably a few hundred locations should work
      - Size should be prime
    - Hash function
      - Recall that you have to be careful hashing ASCII characters – can't just add up ASCII values
        - A decent method is shift and add
        - Perhaps limit to subset of entire string
      - Java String class provides a built-in function
      - Remember, looking for efficiency, not overkill

Symbol Table

- Well, a symbol table probably ought to contain symbols
  - Need a Symbol class
  - We are trying to store attributes related to a particular string
  - Common attributes you might store in symbol table
    - Type – often most important one
      - Will say more about structure of this later
    - Register number
    - Memory location
    - Location of declaration in input file
    - List of uses (may be part of some analysis supporting optimization)
  - Symbol class may also contain pointers to other Symbols as part of table structure

Symbol Table

- Example Symbol
  - What instance variables a particular compiler needs is very implementation dependent

```
public class Symbol {

    private Type type;
    private Location loc;
    private Symbol prev;
    private Symbol next;
    private int RegNum;
    private int MemOffset;

    public Symbol (Type t) {
        type = t;
    }
        // accessor methods
}
```

Symbol Table

- Looks pretty straightforward
  - Well, not so fast
- We have been thinking in terms of storing information on variables
  - Also other things need to think of
    - Constants
      - The string “Please enter data” is a constant string, and if it appears multiple times in program, should only require one storage location
      - Often stored in separate symbol table
    - Labels – may be separate table
    - Function declarations – in C, these are like global constants (but not in Pascal which allows nesting)
    - Type declarations
      - typedef int *myInt;

Symbol Table

- So, we may or may not store different types of information in separate tables
  - But this isn't the big complication
  - The big complication is scope

- For one thing, we want to be able to determine declare before use, so we want to build the symbol table dynamically during semantic analysis
  - When we get to a particular reference, if it is already in symbol table then we know it has been declared

- Block structured languages (Ada, Pascal, C, Java) can declare variables inside a block, and it eventually goes out of scope
  - Must be able to delete it from symbol table when not in scope

Symbol Table

- Many languages also allow the same variable name to be used multiple times
  - Typically use most closely nested rule to determine which is valid

```
int i, j;

int joe (int size) {
    int j, z;

    for (int i=0; i < 10; i++) {
        int z;
        j = 4;
        i = 3;
    }

    z = i;
    j++;
}
```

Symbol Table

- So, we need to dynamically add symbols to the table to help with declare-before-use
- We also need to be able to remove symbols (or otherwise mask them) when we are working (walking a portion of AST) on a section of code where the variable is out of scope
- If the same name exists in multiple places, need to determine which was declared in the closest scope

- Sounds like a pretty complicated symbol table
  - Unfortunately, there are no magic solutions out there
  - Two possible approaches
    - Mark each Symbol with scope
      - Extensive search required for all adds and deletes
    - Use multiple symbol tables, one for each scope

Symbol Table

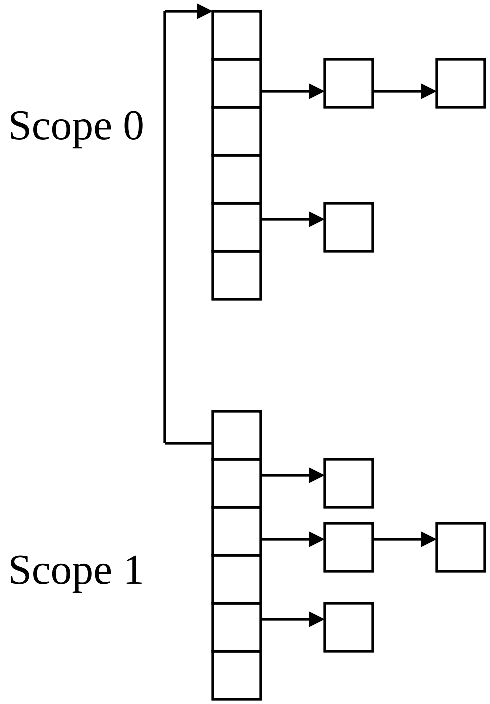
- Using multiple symbol tables
  - We create a symbol table for global variables/functions (scope 0)
  - When we enter a new block (for C, a new function or a compound statement), we create a new symbol table for this scope
    - Also works for languages like Pascal which have nested procedures
  - When we exit a block, we simply delete entire symbol table
    - Our deepest table pointer must be updated
  - Adding symbols – simply add to deepest table
  - Deleting symbols – only do on entire table at a time
  - Finding symbols – Search deepest table; if not there, search next deepest, etc.

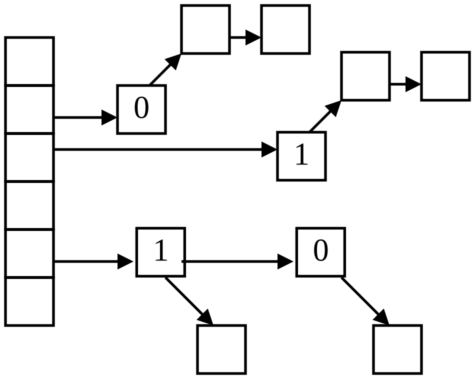
Symbol Table

- Possible data structure for multiple tables
  - Deepest Pointer
  - Each Table has pointer to next scope up
    - If we don't find symbol in deepest table, follow up pointer to next table
      - All hash tables use same hash functions
      - Therefore, already have index into this table
    - Simple to walk up all tables till we hit globals (scope 0)
  - Could even have an up pointer link from each Symbol
    - May be overkill, since path we already have is fairly efficient

Symbol Table

- An alternate structure would store all scopes within the same bucket array
  - But each bucket is a list of lists
  - Could chain all top-level nodes for each scope to aid delete





Symbol Table

- What about object-oriented languages?
  - In C++, what scopes are valid at some point in time?
  - In Java, what scopes are valid?
- In Java, all classes in package and imports (assuming nested classes not allowed) are visible and in scope simultaneously
  - All variables and methods which are declared public are in scope
- When you start executing a method from a class, the scope changes
  - Private objects become visible
  - Protected objects from ancestor classes become visible
  - Package (java) objects become visible
- Within a method, local scope rules still apply as before

Symbol Table

- Object-oriented languages (cont)
  - There is also a concept of what is visible without prefixing the class name and what is visible with a class name prefix
    - Public instance variables of another class are visible, but a prefix is required
  - There is also the concept of static versus instance variables/methods which your table lookup must handle

- Probably the symbol tables themselves don't have to get too much more complicated
  - The find() routine is where much of the complexity will lie
    - It must know what tables are active and which should be searched based on whether a prefix is present

Type Data Structure

- The primary field in a Symbol is the Type of the name being stored
- We have said that we are going to put a Type field within each of the Expression nodes in the AST
- We will now look at how to define a Type data structure

- First we need to understand how types work in languages like C or Java
  - There are a set of basic types
    - char, int, float, double, boolean, short, long, void
  - There is often a set of qualifiers
    - static, extern, register, automatic
  - There are additional symbols which effect type
    - *, [ ], ()

Type Data Structure

- Language types (cont)
  - Basic types can be combined with additional symbols to make new types
    - int A[] array of int
    - int *p pointer to int
    - int *p[] array of pointers to int
    - int **p pointer to pointer to int
    - int *f() function returning pointer to int
- Our Type data structure will correspond to words above
  - It will be recursive, containing a Type reference itself
  - It has a Type Kind field – int, float, pointer, func
  - It has qualifier field – static, extern, etc.
  - May have size and alignment info

Type Data Structure

- Type data structure (cont)
  - For int a;
    - kind = int, child = null
  - For int *a;   (pointer to int)
    - kind = pointer
    - child = {kind=int, child=null}
  - For int *a[5];   (array of pointer to int)
    - kind = array
    - size = 5 (or could be 5 * sizeof(int))
    - child = {kind = pointer, child =
      { kind = int } }

```
public class Type {

    int kind;
    int qualifier;
    Type child;
    int size;
    int align;
    Object misc;

    public Type (int k) {
        kind = k;
    }
}
```

Type Data Structure

- Functions are just another type
  - int *func()
    - kind = function
    - child = {kind = pointer, child =
      { kind = int } }
    - misc = object containing a function parameter list

- Structures (or classes) are just another kind
  - Use misc field to point to list of fields
  - Since we need a list of parameters and list of fields (which are both types, maybe we need a sibling field in Type
  - What would data structure for following look like?

```
struct A {
  int B;
  float *C;
}
```

Type Data Structure

- Classes aren't too much more complicated
  - Methods stored in a method table
  - Like a struct, but has extra pointer to a virtual method table
    - Method table pointer may be new kind
  - What might the data structure for following look like?

```
public class Joe {
    int a;
    float b;

    public Joe (int a1, int b1) {
        a = a1;
        b = b1;
    }
    public eat ( ) {  … }
    public sleep ( ) { … }
}
```

Type Data Structure

- The last type-related thing we will look at is the ability to define new types, using something like typedef
  - typedef int myInt;
    - In some languages this creates a whole new type, whose characteristics are similar to ints
    - In C, this essentially just creates a new name for ints
  - typedef struct {

        int a;
        int b;
    } myStruct;
    - In C, this creates a new name for this particular structure

Type Data Structure

- If a typedef creates a whole new type, perhaps our Type object would have a kind = named object
- If typedef (like in C) just creates an alias, we can just enter the actual type when we create the Type for the variable
  - We store alias info in a special Alias table, and look up when needed
  - Alternatively, we could store in standard name table, but use a special kind = alias
    - e.g., myName is an Alias to a Pointer to an int

- OK, one more thing … inheritance
  - Adds the concept that one class is a subset of another
  - Won't worry about details, but adds further complexity which must be handled

Type Checking

- Well, let's put it back in context
  - The reason we want have a Symbol Table and that we defined a Type data structure was to support type checking
- A basic question we must be able to answer is "are these two objects of the same type?"
  - Answer depends upon the language and the way it handles type equivalence
  - Some use structure equivalence
    - Same type if equate to same structure
  - Some languages use name equivalence
    - If do typedef int A, A is not same type as int
    - typedef int feet; typedef int meters   feet != meters
  - Some use declaration equivalence
    - A weaker kind of name equivalence, that allows some aliasing

Type Checking

- C uses declaration equivalence for structures and unions, but structural equivalence for everything else
  - typedef int *p1   - p1 is the same type as an int *
  - typedef struct A1 { … } A;   - A is same type as a struct A1
    - But two structs with the same fields and different names are not the same type, unless the above alias is used

- Why should we care?
  - Consider the C code below – is this a semantic error?

```
typedef int *p1;
int *a;
p1 b;
…
b = a;
```

Type Checking

- How do we implement type equivalence testing?
  - For structural equivalence, we can simply compare the Type data structures (entire recursive structure)
  - For declaration equivalence, when we see a declaration of a variable using an alias type, we look up the actual type and give the variable its actual type
    - Then we can use the structural equivalence test to compare two variables
- Example: for code below, how does type checking work and is this code legal?

```
int **p;
int *q;
…
*p = q;
```

Type Checking

- For some language constructs, we want to do type checking
  - Function parameters, etc;

- For other constructs, we want to do type inference (or type coercion) as well as
  - For example, an assignment statement
    - *p = q; we want to check for legality
    - myFloat = myInt; we test for legality, but also inference
      - We want to cast the integer into a float
      - We actually change the AST, adding a new conversion node

- If types same, OK
- If types difference, need a table to decide if error or coercion

Semantic Analysis

- Semantic analysis (type checking) can be accomplished in a single pass through the code
  - We add a visitSemantic() method to all nodes
  - Depending on the node type, various actions will be performed
    - At variable declarations, names and types added to symbol table
    - At compound statements, scope is updated and new symbol table is created (if any local vars)
    - At end of compound statement walk, scope is decreased and table deleted
    - Statement nodes don't do much except pass on the visitSemantic() call
    - Expression nodes will need to do type checking and inference

Semantic Analysis

- Let's talk a couple examples
  - What happens at each step of the following code segments?

```
int **p;
int *q;
…
*p = q;
```

```
int doStuff (int stuff1, float stuff2) {
   stuff1 = stuff2;
}
…

int x = doStuff (2, 3);
```

Semantic Analysis

- What about overloaded function names
  - That allows you to have multiple functions of the same name, but with different parameters (and maybe return type)
  - How would this affect the symbol table?

- As a minimum, we will have to allow multiple functions to have the same name, but different signature
  - This may be a reason to put functions in a separate table from vars
  - If you get a second definition of a function name, but with a different signature, go ahead and enter into symbol table
  - At call sites, check all table entries of this name for one with correct signature
  - Your low-level code will need to use unique names

Semantic Analysis

- When you start looking at object-oriented languages, semantic analysis obviously becomes somewhat more difficult
  - We've only scratched the surface

- Well, that's it for Semantic Analysis
- That's also it for the Analysis phase of compilation

- There are other passes on the AST you could do (we won't look at)
  - Profiling
  - In-lining
  - High-level optimization

- On to Synthesis and Code Generation