Semantics

- Syntax analysis results in a parse tree
  - Also called abstract syntax tree
- This means the structure of the program is correct according to the grammar
- But does the program make sense?
- What does it mean?

- Need to do semantic analysis

Basic Semantics

- Examples of semantics
  - The type of an expression – is it a double or boolean
    - Important to know if it is the condition of an if statement
  - Is an identifier declared as a structure
    - Must know if we want to access fields
  - How many arguments should a function have?

- Much of the semantic analysis involves checking variables and types
  - Even in a “typeless” language there are underlying types
Variable Attributes

- Attributes of a variable
  - **Name** (identifier) – associates the variable use with declaration, may be handle into a symbol table
  - **Location** (address) – storage associated with variable
    - Two variables with same location are **aliases** of each other
  - **Type** (meaning) – how to interpret the value (or maybe it is a constant)
  - **Value** – the actual value of the variable

- In general, identifiers have some or all of these attributes

Lvalue versus Rvalue

- Not all values get the same treatment
- An identifier on the right side of an assignment just has its value fetched – it is an **rvalue**
  - It could be a constant, i.e., read only
  - The semantics are to fetch only
- A variable appearing on the left side of an assignment is an **lvalue**
  - Its value will be changed – different semantics
- Expressions can be rvalues or lvalues
  - `*p++ = a[i++]`
Binding

- The attributes of a variable are **bound** to the identifier at different points in the program analysis or execution, depending on the language
  - Data type is bound at **compile** time in C, but at **execution** time in Scheme
  - Size of an integer may be bound when the language is **defined** or when the language is **implemented**
  - The location may be determined at **load** time, or execution time
  - A function’s body may be bound at compile time or **link** time or execution time

Symbol Table

- A table is used to keep track of identifier attributes, e.g.,
  - Class names and data layout (recursive types allowed?)
  - Variable names, types, and locations
  - Function names, arg lists and types, return type, bodies
- Depending on the binding time, some information may be determined later
- Symbol table will be used during translation (compile)
- May also be used during execution
  - Then it is called the **environment**
Declarations

- Some languages define variables implicitly
  - First use of variable causes entry into symbol table
  - No explicit declaration required
- Languages like C, C++, Java require declarations
  - Declaration causes entry into symbol table
  - Variable must be in symbol table in order to be used
  - Declaration becomes binding time for many attributes
- Typical attributes of declaration
  - Type, initial value, other modifiers (static, const, register)

Declarations and Definitions

- Declaration associates some attributes with name
  - E.g., Type information only
  - Information necessary to understand the identifier’s use
  - Binding to other attributes, e.g., location may be deferred
- Definition binds all attributes
  - For variables, defines storage, may also provide initial value
- A “declaration” may also be a definition
- But could be separate statements in the language
  - Function declaration in C/C++ often separate from definition
  - Global external variables may have deferred binding
  - Interface in Java like a declaration
Scope

- The scope of a variable declaration is the region of a program in which the variable is known (bound)
  - The lines of code where the variable declared can appear
- Often implicitly indicated by the physical position of the declaration line
- In block structured languages, a common rule is for scope to extend from declaration line forward to the end of the enclosing block
- Scope can extend backwards to beginning of block
  - Class members in C++ and Java
  - Requires multiple passes to build symbol table
  - Keywords (private, public) may modify scope

Lexical vs. Dynamic Scope

- Scope managed by symbol table lookup during translation phase is called **lexical** scope
  - Also called **static** scope since it is the result of static analysis
  - Follows the layout of the file
  - Not affected by execution
- Scope managed by execution is called **dynamic** scope
- Static scope can be managed during execution, but requires extra links to contexts
Lexical vs. Dynamic Scope Example

```java
class Test {
    int x = 2;
    void f(){ System.out.println(x); }
    void g(){
        int x = 3;
        f();
    }
}
```

- Prints the value 2 in Java (static scope)
  - The x in function f is bound to the first x at compile time
- Under dynamic scope, this would print 3 since the x in g is the most recent (closest in execution) declaration of x
- What would be printed under dynamic scope if another call to f were inserted at the beginning of g before the declaration of x?

Dynamic Scope

- Most languages use lexical (static) scope
  - Natural, independent of execution paths
- Some languages do implement dynamic scope
  - Lisp – later considered a “bug”
  - Scheme is Lisp with static instead of dynamic scope
- Dynamic scope makes program analysis harder
  - Logic more difficult to follow since execution path dependent
  - But actually easier to implement
- But dynamic scope is similar to the idea of dynamic binding of methods, so is a useful concept
Holes in Scope

- A declaration in an inner block with the same variable name hides the definition from an outer block
  - Variable’s storage continues to exist, but the name cannot be used to get at it
  - Creates a hole in the scope
- Language may have rules
  - Re-declaration may not be allowed at all
  - May be allowed only in a block
  - May require that the type stays the same
  - Java only allows hiding class variables, C++ permits any
- Variables in blocks are called local scope

Other Scope Issues

- Classes make scope rules more complicated
  - Private, public, protected affect regions of scope
  - Scope qualifier ( :: in C++, and . in Java) used to specify scope
- Function parameters are also declarations, so follow scope rules
- Keyword static on a non-local variable in C/C++ makes scope local to the file
- C++ introduced namespaces to give control over scope blocks, Java has packages and imports
- Name overloading requires extra information for scope resolution
Other Scope Issues

- Order of declaration for local variables
  - C requires declarations at beginning of blocks
  - Java, C++ allow anywhere
  - But, no forward references allowed
- Order of declaration for class variables
  - Does not matter – “forward” references (in class) allowed
  - C++ allows class declarations (not definitions) for forward reference
- Declaration in for loop
  - Is scope just the body and control of the loop?

Symbol Tables

- Symbol table is the “directory” for variables
- One way to organize is as a stack of contexts
  - New context begun with every block
  - E.g., function definition, control block
  - Context popped at end of block
- Each context keeps stack of declarations
  - Each entry would have name, type, location, etc.
- To resolve name, search back through declaration list from last added to first, then back through contexts
- Could use same strategy for dynamic scope, but building as execution unfolds rather than static translation
Storage (The Environment)

- When to resolve the storage for a variable
  - Could be completely static – Fortran, assembly language
  - Could be completely dynamic – Scheme, ML
  - May be a combination – C, C++, Java
- The storage determines the lifetime of a variable
  - When the value of the variable is in existence
  - Not the scope – the scope is where in the code the variable can be mentioned
  - A variable may exist when it is not in scope
- Storage and scope are (mostly) orthogonal

Storage Types

- **Static**
  - Persistent – entire lifetime of program
  - Think of as existing before main is called until after main returns
  - Global variables, functions in C/C++, local variables labeled “static”
  - Static class variables in Java
- **Automatic**
  - Storage created automatically as execution proceeds
  - **Stack** storage – local variables in functions, blocks
  - Automatically de-allocated on block end / function return
- **Dynamic**
  - Storage created during execution as directed by program
  - **Heap** storage – `malloc` in C, `new` in C++/Java
  - Explicitly allocated so explicit de-allocation required (garbage collection)
Typical Storage Layout

int x;  // static
g() { // pointer x is on stack
    int * x = new int[10]; // 10 ints // on heap
}
f() {
    int x;  // stack
    g();
}
main() {
    int x;  // stack
    f();
}

Storage and Names

- Aliases
  - Two names bound to same storage
  - Pointers in C give indirect aliases
    int n = 10;
    int *pn = & n;  /* (*pn) is same storage as n */
  - All object variables in Java are indirect aliases
  - C++ allows direct aliases
    int n = 10;
    int & m = n;  // m is just another name for n
  - Can be a useful design tool – especially for function parameters
  - May reduce readability or cause confusion
Storage Problems

- **Dangling References**
  - Occurs if object can still be accessed beyond lifetime of storage

    ```c
    int *pn = malloc(sizeof int); /* int on heap */
    *pn = 10; /* OK – assign to heap storage */
    free(pn); /* release the storage */
    *pn += 1; /* Error – ref to freed storage */
    ```
  - Also occurs if pointer to automatic variable is passed out of block
    - Variable’s storage on stack is “released” at end of block
    - Stack storage is reused – no longer “belongs” to variable

- **Garbage**
  - Occurs if object storage is never released

    ```c
    void foo() {
    int * arr = new int[5]; // temp array
    . . . // calculations with arr
    if (. . .) return; // premature return
    . . . // more calculations
    delete arr[]; // Release the space
    return;
    }
    ```
  - Consumes resources that are no longer needed
    - Common programming error, hard to locate
  - C++ auto variables with constructors/destructors help
  - Automatic garbage collection (Java)
    - Compiler reliably arranges for resource release
    - May pose efficiency, timing problems
Data Types

- Type is probably the single most important attribute of identifiers in a program
- Static analysis of types can
  - Detect many programming errors
  - Allow for efficient execution
  - Improve security by ensuring proper use
  - Allow design to be expressed in code

What are types?

- E.g., the primitive types of a language – int, char, double, etc.
- But these are really just names that stand for categories (sets? classes?) of values
- Could define by listing all the values
  - E.g., bool has the values true and false
  - int has values from (-$2^{31}$) to ($2^{31} - 1$)
- But there is more to a type
  - The behavior, e.g., operations that may be performed
Type Construction

- Languages often allow programmer defined types
- Several methods of type construction
  - Cartesian products: tuples in ML, struct in C
  - Unions: actual unions in C, inheritance in C++/Java
    - Discriminated unions tag with names
  - Subset/Subtype: enums, inheritance
  - Arrays and functions: these are new types
  - Vectors, lists, sequences: allowed in some languages
- Not to be confused with class constructors/destructors
  - These create values, not types

Recursive Types

- Types defined in terms of themselves
  - Typically used for list nodes, tree nodes
- More generally we think of recursively defined types
  - A class defined in terms of classes defined in terms of classes…
- May be limitations
  - C/C++ do not allow a class to contain object of same type
    - But it may contain a pointer to such an object
  - Java allows this, but it works since all objects are pointers
Comparing Types

- Analyzing a program often requires comparison of types
- When are two types the same?
- Two ways to compare types:
  - Structural equivalence – essentially the layout of the data is the same, even if two separate definitions
  - Name equivalence – types are the same only if they have the same name
- C/C++ use structural equivalence for primitive types, arrays, pointers. Otherwise name equivalence is used.

Type Checking

- A language is **strongly typed** if its type system guarantees statically (as far as possible) that no data-corrupting errors can occur during execution
  - And all errors that cannot be checked statically are caught at runtime
  - Java and ML are strongly typed, C/C++ and Scheme are not
  - But C++ is much better type checking than C
- Strong type checking guarantees a legal program is safe
- But some illegal programs may be safe!
Type Compatibility

- Different compatibility constraints for assignment, expressions, function calls, etc.
- When are the same types required?
- Assignment – left side must be lvalue, right side must be rvalue compatible with left side’s type
  - Conversion may be required – C/C++ may convert implicitly (e.g., from int to double), but ML requires explicit conversion
- Expressions, functions in C/C++ implicitly convert
  - C++ uses class constructors to convert the data value
  - But constructors may be declared explicit to avoid use for conversions
  - Subclass may be used where superclass is expected
- Implicit conversion increases writability, decreases reliability