Lecture #18

Semantic Analysis – I

Semantic Analysis

- **Lexical analysis**: Source code to tokens
- **Parsing**: Validate token conformity with grammar

- Now we need to check if the code makes sense (have correct meaning)
    - Lexically and syntactically correct programs may still contain other errors – correct usage of variables, objects, functions, ...

- **Semantic analysis:** Ensure that the program satisfies a set of rules regarding the usage of programming constructs (variables, objects, expressions, statements)

Semantic Analysis

- Examples of semantic rules
  - Variables must be defined before being used
  - A variable should not be defined multiple times
  - In an assignment statement, the variable and the expression must have the same type
  - The test expression of an if statement must have boolean type
- Two major semantic analysis categories
  - Semantic rules regarding *types*
  - Semantic rules regarding *scopes*

Types

- **Type**
  - A set of **values**.
    - E.g., “int x” in C means –2^31 <= x < 2^31
  - A set of **operations** on those values.

- Certain operations are **legal** for values of each type.
  - It doesn’t make sense to add a function pointer and an integer in C.
  - It does make sense to add two integers.
  - *Both have the same assembly language implementation!*

Types

- *Base types*
  - **int, float, double, char, bool, etc.**
  - **primitive** types provided directly by the underlying hardware.
  - User-defined variants on the base types (such as C **enums).**
- *Compound types*
  - **arrays, pointers, records, structs, unions, classes, and so on**
  - Constructed as aggregations of the base types and simple compound types.
- *Complex types (ADTs)*
  - **lists, stacks, queues, trees, heaps, tables, etc.**

Declarations

- Statements that communicate to compiler information necessary to:
  - Establish name and type of any data object
  - Sometimes visibility and lifetime (**private** in Java; **static** in C).
- Example:

```
double calculate(int a, double b); // function prototype
int x = 0;        // global variables available throughout
double y;         // the program

int main() {
   int m[3];      // local variables available only in
   char *n;       // main
   ...
}
```

Scope

- The **scope** of an identifier is the portion of a program in which that identifier is accessible.
  - Defined by program text enclosed by basic delimiters
    - {} in C, **begin-end** in Pascal
- **Nested Scopes** – Scopes defined within other scopes
- Current Scope, Open Scopes and Closed Scopes

- *The same identifier name may refer to different things in different parts of the program.*

Variable Scope

- Scope of variables in statement blocks:

```
{ int a;
    ...
   {int b;
   ...
   }
    ....
}
```

scope of variable a

scope of variable b

- Scope of **global** variables: current file
- Scope of **external** variables: whole program

Scope Checking

- Determine if an identifier is accessible at a given point in the program
- Throw error message if variable declared in one function is accessed in another
  - Because variables declared in the current scope and in the open scopes containing the current scope are only accessible

```
int a;
void Binky(int a) {
   int a;
   a = 2;
   ...
}
```

Which **a** *gets assigned?*

Type Systems

- A language's **type system** specifies which operations are valid for which types.
- The goal of type checking is to ensure that **operations** are used with the **correct types**.
  - Enforces intended interpretation of values, because nothing else will!
- **Type Errors**: Improper or inconsistent operations during program execution
- **Type-safety**: Absence of type errors

Type Checker Design

- ***Identify the types that are available in the language***

- Example
  - Base types (**int, double, bool, string**)
  - Compound types (arrays, classes, interfaces)
  - An array can be made of any type (including other arrays)
  - ADTs

Type Checker Design

- ***Identify language constructs that have types associated with them***
- Language Construct examples
  - **Constants**
    - Obviously, every constant has an associated type.
    - A scanner tells us these types as well as the associated lexeme.
  - **Variables**
    - Declared with one of the base / compound types.
  - **Functions**
    - Have a return type; also each parameter / argument has a type.
  - **Expression**
    - Can be a constant, variable, function call, or some operator applied to expressions.

Type Checker Design

- ***Identify the semantic rules for the language***
  - Govern what types are allowable in the various language constructs
- Examples:
  - Operand to a unary minus must either be **double or int**
  - The expression used in a loop test must be of **bool** type
  - General rules:
    - All variables must be declared, all classes are global, etc.
- These three things together (the types, the relevant constructs, and the rules) define a *type system for a language.*
- Once we have a type system, we can implement a type checker

Strong Vs. Weak Typing

- Refer to how much type consistency is enforced
- **Strongly typed languages**
  - Every type error is detected during compilation
- **Weakly typed languages**
  - Allow programs which contain type errors
- Type checking can be done compilation, during execution, or divided across both

Strong vs. Weak Typing

	Weak Typing	Strong Typing

Pseudocode	`a = 2`
	`b = "2"`
	
	`concatenate(a, b) # Returns "22"`
	`add(a, b)         # Returns 4`
		`a = 2`
		`b = "2"`
		
		`concatenate(a, b)      # Type Error`
		`add(a, b)              # Type Error`
		`concatenate(str(a), b) # Returns "22"`
		`add(a, int(b))         # Returns 4`

Languages	BASIC, JavaScript, Perl, PHP, Rexx	ActionScript 3, C++, C#, Java, Python, OCaml

Static vs Dynamic Checking

- Static type checking
  - Performed at **compile time**
- Dynamic type checking
  - Performed at **run time**, as the program executes
- Examples of dynamic checking
  - Array bounds checking
  - Null pointer dereferences

Static Type Checking

- Performed at **compile time**
- Information the type checker needs is obtained via declarations and stored in a master symbol table
- Achieving strong typing is *difficult*:
  - If **a** and **b** are of type **int** and we assign very large values to them, **a * b** may not be in the acceptable range of **ints**,
  - An attempt to compute the ratio between two integers may raise a division by zero

Next Lecture

Syntax Directed Translation