Names and Bindings

Syllabus

Lecture Series (hours)	**Topics**
1-4	Introduction and Motivation, Paradigms
5-10	Syntax and Semantics, BNF, Compilation
11-18	Data Types, Constructs, Functions, Activation Records, Names and Bindings
19-28	Functional PLs, Logical PLs, Lambda Calculus, Event driven programming, Concurrency
29-36	Virtual Machines, Managed Languages, JIT, Case study

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor
- Variables are characterized by attributes
  - To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

- Design issues for names:
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Names (continued)

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN 95: maximum of 31
    - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
    - C#, Ada, and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one

Names (continued)

- Special characters
  - PHP: all variable names must begin with dollar signs
  - Perl: all variable names begin with special characters, which specify the variable’s type
  - Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Names (continued)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in others are not
    - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)

Names (continued)

- Special words
  - An aid to readability; used to delimit or separate statement clauses
    - A *keyword* is a word that is special only in certain contexts, e.g., in Fortran
      - `Real VarName` (`Real` *is a data type followed with a name, therefore* `Real` *is a keyword)*
      - `Real = 3.4` (*Real is a variable)*
  - A *reserved word* is a special word that cannot be used as a user-defined name
  - Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Type - bindings
  - Address/Storage - bindings
  - Value
  - Lifetime and Scope - local/global

Variables Attributes

- Name - not all variables have them
- Address - the memory address with which it is associated
  - A variable may have different addresses at different times during execution
  - A variable may have different addresses at different places in a program
  - If two variable names can be used to access the same memory location, they are called aliases
  - Aliases are created via pointers, reference variables, C and C++ unions
  - Aliases are harmful to readability (program readers must remember all of them)

Variables Attributes (continued)

- *Type* - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision
- *Value* - the contents of the location with which the variable is associated

  - The l-value of a variable is its address

  - The r-value of a variable is its value
- *Abstract memory cell* - the physical cell or collection of cells associated with a variable

The Concept of Binding

A *binding* is an association between an entity and an attribute, such as between a variable and its type or value, or between an operation and a symbol

- *Binding time* is the time at which a binding takes place.

Possible Binding Times

- Language design time -- bind operator symbols to operations
- Language implementation time-- bind floating point type to a representation
- Compile time -- bind a variable to a type in C or Java
- Load time -- bind a C or C++ `static` variable to a memory cell)
- Runtime -- bind a nonstatic local variable to a memory cell

Static and Dynamic Binding

- A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.
- A binding is *dynamic* if it first occurs during execution or can change during execution of the program

Explicit/Implicit Type Declaration

- An *explicit declaration* is a program statement used for declaring the types of variables
- An *implicit declaration* is a default mechanism for specifying types of variables through default conventions, rather than declaration statements
- Fortran, BASIC, Perl, Ruby, JavaScript, and PHP provide implicit declarations (Fortran has both explicit and implicit)
  - Advantage: writability (a minor convenience)
  - Disadvantage: reliability (less trouble with Perl)

Explicit/Implicit Declaration (continued)

- Some languages use type inferencing to determine types of variables (context)
  - C# - a variable can be declared with **`var`** and an initial value. The initial value sets the type
  - Visual BASIC 9.0+, ML, Haskell, F#, and Go use type inferencing. The context of the appearance of a variable determines its type

Dynamic Type Binding

- Dynamic Type Binding (JavaScript, Python, Ruby, PHP, and C# (limited))
- Specified through an assignment statement
  e.g., JavaScript

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

  - Advantage: flexibility (generic program units)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult
    - i and x are integers, y array, what does i = y mean?

Storage binding

- Storage Bindings & Lifetime
  - Allocation - getting a cell from some pool of available cells
  - Deallocation - putting a cell back into the pool
- The lifetime of a variable is the time during which it is bound to a particular memory cell

Categories of Variables by Lifetimes

- Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ `static` variables in functions
  - Advantages: efficiency (direct addressing), history-sensitive subprogram support
  - Disadvantage: lack of flexibility (no recursion), storage cannot be shared

Categories of Variables by Lifetimes

- **Stack-dynamic**--Storage bindings are created for variables when their declaration statements are *elaborated*.

  (A declaration is elaborated when the executable code associated with it is executed)
- If scalar, all attributes except address are statically bound
  - local variables in C subprograms (not declared **`static`**) and Java methods
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Categories of Variables by Lifetimes

- **Explicit heap-dynamic** -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
- Referenced only through pointers or references, e.g. dynamic objects in C++ (via **`new`** and **`delete`**), all objects in Java
- Advantage: provides for dynamic storage management
- Disadvantage: inefficient and unreliable

Categories of Variables by Lifetimes

- **Implicit heap-dynamic--**Allocation and deallocation caused by assignment statements
  - all variables in APL; all strings and arrays in Perl, JavaScript, and PHP
- Advantage: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is visible
- The *local variables* of a program unit are those that are declared in that unit
- The *nonlocal variables* of a program unit are those that are visible in the unit but not declared there
- *Global variables* are a special category of nonlocal variables
- The scope rules of a language determine how references to names are associated with variables

Static Scope

- Based on program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*
- Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Common LISP, Scheme, Fortran 2003+, F#, and Python)

Scope (continued)

- Variables can be hidden from a unit by having a "closer" variable with the same name
- Ada allows access to these "hidden" variables
  - E.g., **`unit.name`**

Blocks

- A method of creating static scopes inside program units--from ALGOL 60
- Example in C:

```
void sub() {
  int count;
  while (...) {
    int count;
    count++;
    ...
  }
  …
}
```

- Note: legal in C and C++, but not in Java and C# - too error-prone

Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear
  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
  - In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
    - However, a variable still must be declared before it can be used

The LET Construct

- Most functional languages include some form of **`let`** construct
- A let construct has two parts
  - The first part binds names to values
  - The second part uses the names defined in the first part
- In Scheme:

```
(LET (
  (name₁ expression₁)
  …
  (nameₙ expressionₙ)
)
```

The LET Construct (continued)

- In ML:

```
let
  val name₁ = expression₁
  …
  val nameₙ = expressionₙ
in
 expression
end;
```

- In F#:
  - First part: **`let`** left_side = expression
  - (left_side is either a name or a tuple pattern)
  - All that follows is the second part

Declaration Order (continued)

- In C++, Java, and C#, variables can be declared in `for` statements
  - The scope of such variables is restricted to the `for` construct

Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file
  - These languages allow variable declarations to appear outside function definitions
- C and C++have both declarations (just attributes) and definitions (attributes and storage)
  - A declaration outside a function definition specifies that it is defined in another file

Global Scope (continued)

- PHP
  - Programs are embedded in HTML markup documents, in any number of fragments, some statements and some function definitions
  - The scope of a variable (implicitly) declared in a function is local to the function
  - The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions
    - Global variables can be accessed in a function through the `$GLOBALS` array or by declaring it `global`

Global Scope (continued)

- Python
  - A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be `global` in the function

Evaluation of Static Scoping

- Works well in many situations
- Problems:
  - In most cases, too much access is possible
  - As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

Scope Example

```
function big() {
    function sub1()
       var x = 7;
    function sub2() {
      var y = x;
    }
   var x = 3;
}
```

```
big calls sub1
sub1 calls sub2
sub2 uses x
```

- Static scoping
  - Reference to `x` in `sub2` is to `big`'s `x`
- Dynamic scoping
  - Reference to `x` in `sub2` is to `sub1`'s `x`

Scope Example

- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - *Disadvantages:*
    1. While a subprogram is executing, its variables are visible to all subprograms it calls
    2. Impossible to statically type check
    3. Poor readability- it is not possible to statically determine the type of a variable

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a **`static`** variable in a C or C++ function

Referencing Environments

- The *referencing environment* of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is active if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

Named Constants

- A *named constant* is a variable that is bound to a value only when it is bound to storage
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called *manifest constants*) or dynamic
- Languages:
  - Ada, C++, and Java: expressions of any kind, dynamically bound
  - C# has two kinds, **`readonly`** and **`const`**
    - the values of **`const`** named constants are bound at compile time
    - The values of **`readonly`** named constants are dynamically bound

Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors

Types and Parametric Polymorphism

Type

A type is a collection of computable values that share some structural property

- Examples
  - Integers
  - Strings
  - int → bool
  - (int → int) →bool
- "Non-examples"
  - ∀ {3, true, λx.x}
  - Even integers
  - ∀ {f:int → int | if x>3 then f(x) > x*(x+1)}

Distinction between sets that are types and sets that are not types is language-dependent

Uses for Types

- Program organization and documentation
  - Separate types for separate concepts
    - Represent concepts from problem domain
  - Indicate intended use of declared identifiers
    - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations such as 3 + true - “Bill”
- Support optimization
  - Example: short integers require fewer bits
  - Access record component by known offset

Operations on Typed Values

- Often a type has operations defined on values of this type
  - Integers: + - / * < > …   Booleans: ∧ ∨ ¬ …
- Set of values is usually finite due to internal binary representation inside computer
  - 32-bit integers in C: –2147483648 to 2147483647
  - Addition and subtraction may overflow the finite range, so sometimes **`a + (b + c) ≠ (a + b) + c`**
  - Exceptions: unbounded fractions in Smalltalk, unbounded Integer type in Haskell
  - Floating point problems

Type Errors

- Machine data carries no type information
  - 01000000010110000000000000000000 means…
  - Floating point value 3.375? 32-bit integer 1,079,508,992? Two 16-bit integers 16472 and 0? Four ASCII characters `@ X NUL NUL`?
- A type error is any error that arises because an operation is attempted on a value of a data type for which this operation is undefined
  - Historical note: in Fortran and Algol, all of the types were built in. If needed a type “color,” could use integers, but what does it mean to multiply two colors?

Static vs. Dynamic Typing

- Type system imposes constraints on use of values
  - Example: only numeric values can be used in addition
  - Cannot be expressed syntactically in EBNF
- Language can use static typing
  - Types of all variables are fixed at compile time
  - Example?
- … or dynamic typing
  - Type of variable can vary at run time depending on value assigned to this variable
  - Example?

Strong vs. Weak Typing

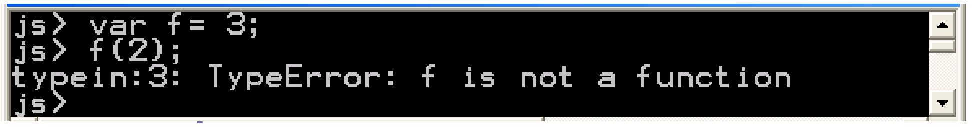
- A language is strongly typed if its type system allows all type errors in a program to be detected either at compile time or at run time
  - A strongly typed language can be either statically or dynamically typed!
- Union types are a hole in the type system of many languages (why?)
- Most dynamically typed languages associate a type with each value

Compile- vs. Run-Time Checking

- Type-checking can be done at compile time
  - Examples: C, ML

f(x) must have f : A → B and x : A

- ... or run time
  - Examples: Perl, Ja



```
js> var f= 3;
js> f(2);
typein:3: TypeError: f is not a function
js>
```

- Java does both
- Basic tradeoffs
  - Both prevent type errors
  - Run-time checking slows down execution
  - Compile-time checking restricts program flexibility
    - JavaScript array: elements can have different types
    - ML list: all elements must have same type

Which gives better programmer diagnostics?

Expressiveness vs. Safety

- In JavaScript, we can write function like

  ```
  function f(x) { return x < 10 ? x : x(); }
  ```

  Some uses will produce type error, some will not

- Static typing always conservative

  ```
  if  (big-hairy-boolean-expression)
        then  f(5);
        else   f(10);
  ```

  Cannot decide at compile time if run-time error will occur, so can’t define the above function

Relative Type Safety of Languages

- Not safe: BCPL family, including C and C++
  - Casts, pointer arithmetic
- Almost safe: Algol family, Pascal, Ada
  - Dangling pointers.
    - Allocate a pointer p to an integer, deallocate the memory referenced by p, then later use the value pointed to by p
    - No language with explicit deallocation of memory is fully type-safe
- Safe: Lisp, ML, Smalltalk, JavaScript, and Java
  - Lisp, Smalltalk, JavaScript: dynamically typed
  - ML, Java: statically typed

Enumeration Types

- User-defined set of values
  - enum day {Monday, Tuesday, Wednesday,
                                Thursday, Friday, Saturday, Sunday};
    enum day myDay = Wednesday;
  - In C/C++, values of enumeration types are represented as integers: 0, ..., 6
- More powerful in Java:
  - for (day d : day.values())
          System.out.println(d);

Pointers

- C, C++, Ada, Pascal
- Value is a memory address
  - Remember r-values and l-values?
- Allows indirect referencing
- Pointers in C/C++
  - If T is a type and ref T is a pointer:
    & : T → ref T   * : ref T → T   *(&x) = x
- Explicit access to memory via pointers can result in erroneous code and security vulnerabilities

Arrays

- Example: float x[3][5];
- Indexing [ ]
  - Type signature: T[ ] x int → T
  - In the above example, type of x: float[ ][ ], type of x[1]: float[ ], type of x[1][2]: float
- Equivalence between arrays and pointers
  - a = &a[0]
  - If either e1 or e2 is type: ref T,

    then   e1[e2] = *((e1) + (e2))
  - Example: a is float[ ] and i int, so a[i] = *(a + i)

Strings

- Now so fundamental, directly supported by languages
- C: a string is a one-dimensional character array terminated by a NULL character (value = 0)
- Java, Perl, Python: a string variable can hold an unbounded number of characters
- Libraries of string operations and functions
  - Standard C string libraries are unsafe!

Structures

- Collection of elements of different types
  - Not in Fortran, Algol 60, used first in Cobol, PL/I
  - Common to Pascal-like, C-like languages
  - Omitted from Java as redundant

```
struct employeeType {
    char name[25];
    int age;
    float salary;
};
struct employeeType employee;
...
employee.age = 45;
```

Unions

- union in C, case-variant record in Pascal
- Idea: multiple views of same storage

```
type union =
        record
                case b : boolean of
                        true : (i : integer);
                        false : (r : real);
        end;
var  tagged : union;
begin tagged := (b => false, r => 3.375);
      put(tagged.i);  -- error
```

Functions as Types

- Pascal example:

```
function newton(a, b: real; function f: real): real;
```

  - Declares that f returns a real value, but the arguments to f are unspecified

- Java example:

```
public interface RootSolvable {double
   valueAt(double x);}
public double Newton(double a, double b,
   RootSolvable f);
```

Type Equivalence

- Pascal Report:

  "The assignment statement serves to replace the current value of a variable with a new value specified as an expression ... The variable (or the function) and the expression must be of identical type"

- Nowhere does it define identical type
  - Which of the following types are equivalent?

```
struct complex { float re, im; };
struct polar { float x, y; };
struct { float re, im; } a, b;
struct complex c,d;   struct polar e;   int f[5], g[10];
```

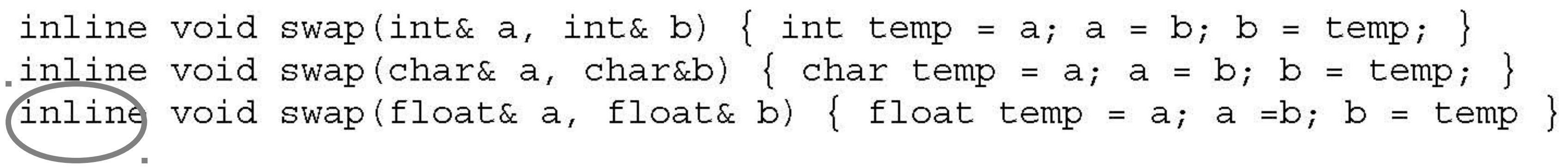
Overloading

- An operator or function is overloaded when its meaning varies depending on the types of its operands or arguments or result
- Examples:
  - Addition: integers and floating-point values
    - Can be mixed: one operand an int, the other floating point
    - Also string concatenation in Java
  - Class PrintStream in Java:

    print, println defined for boolean, char, int, long, float, double, char[ ], String, Object

Function Overloading in C++

- Functions that have the same name but can take arguments of different types



```
inline void swap(int& a, int& b) { int temp = a; a = b; b = temp; }
inline void swap(char& a, char&b) { char temp = a; a = b; b = temp; }
inline void swap(float& a, float& b) { float temp = a; a =b; b = temp }
```

Tells compiler (not preprocessor) to substitute the code of the function at the point of invocation

- Saves the overhead of a procedure call
- Preserves scope and type rules as if a function call was made

Type Checking Expressions

Production	***Semantic Rule***	***Yacc Code***
E → **id**	E.*type* = **id**.*type*	{ $$ = symtab_lookup(id_name); }
E → **intcon**	E.*type* = INTEGER	{ $$ = INTEGER; }
E → E_1 + E_2	E.*type* = result_type(E_1.*type*, E_2.*type*)	{ $$ = result_type($1, $3); }

```
/*  arithmetic type conversions */
Type result_type(Type t1, Type t2)
{
   if (t1 == error || t2 == error) return error;
   if (t1 == t2) return t1;
   if (t1 == double || t2 == double) return double;
   if (t1 == float || t2 == float) return float;
  …
}
```

Return types:

- currently: the type of the expression
- down the road:
  - type
  - location
  - code to evaluate the expression

Type Checking Expressions: cont'd

Arrays:

```
E → id[ E_1 ]  { t_1 = id.type;
                   if (t_1 == ARRAY ^ E_1.type == INTEGER)
                      E.type = id.element_type;
                   else
                      E.type = error;
                 }
```

Type Checking Expressions: cont'd

Function calls:

```
E → id '(' expr_list ')'
        { if (id.return_type == VOID)
              E.type = error;
          else if ( chk_arg_types(id, expr_list) )   /* actuals
  match formals in number, type */
              E.type = id.return_type;
          else
              E.type = error;
        }
```

Type Checking vs. Type Inference

- Standard type checking

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2; };
```

  - Look at the body of each function and use declared types of identifiers to check agreement

- Type inference

  ~~int~~ f(~~int~~ x) { return x+1; };

  ~~int~~ g(~~int~~ y) { return f(y+1)*2; };

  - Look at the code without type information and figure out what types could have been declared

ML is designed to make type inference tractable

Type Inference Summary

- Type of expression computed, not declared
  - Does not require type declarations for variables
  - Find most general type by solving constraints
  - Leads to polymorphism
- Static type checking without type specifications
  - Idea can be applied to other program properties
- Sometimes provides better error detection than type checking
  - Type may indicate a programming error even if there is no type error (how?)

Summary

- Types are important in modern languages
  - Organize and document the program, prevent errors, provide important information to compiler
- Type inference
  - Determine best type for an expression, based on known information about symbols in the expression
- Polymorphism
  - Single algorithm (function) can have many types
- Overloading
  - Symbol with multiple meanings, resolved when program is compiled

Syllabus

Lecture Series (hours)	Topics
1-4	Introduction and Motivation, Paradigms
5-10	Syntax and Semantics, BNF, Compilation
11-18	Data Types, Constructs, Functions, Activation Records, Names and Bindings
19-28	Functional PLs, Logical PLs, Lambda Calculus, Event driven programming, Concurrency
29-36	Virtual Machines, Managed Languages, JIT, Case study