Names, Types, and Functions

standard hue names
Different values are usually used to solve different instances of a problem.

Would it be efficient to build the particular values into the program?

High Level Languages
(Imperative or Procedural Languages)

• Traditionally the concept of a variable with a changeable association between a name and a value is used.

• Assignments are made to the variable usually multiple times.

• In other words, the same name may be associated with different values.

Relation-based Languages
(Functional Languages)

• Traditionally the concept of a function application is used.

• In a function application, a function receives argument values and returns values.

• In other words, a name is only associated with one value.
This Course

Java (Object Oriented)

ACL2 (Propositional Induction)
Algorithmic Information Theory (Information Compression and Randomness) - Kolmogorov Complexity
Orc (Parallel Computing)
GpH (Parallel Computing)
RDF (Horn Clause Deduction, Semantic Web)

High Level Languages

Jython in Java

A Snapshot of Programming Language History
Symbol Table

1. Each time a scope is entered, push a new dictionary onto the stack.

2. Each time a scope is exited, pop a dictionary off the top of the stack.

3. For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack.

4. Given a name reference, search the dictionary on top of the stack:
   
a) If found, return the binding.

b) Otherwise, repeat the process on the next dictionary down in the stack.

c) If the name is not found in any dictionary, report an error.
Scope Concepts

The lifetime of a variable is the time interval during which the variable has been allocated a block of memory.

Disjoint scopes nested in Outer scope

Nested scope

Symbol Table

Lifetime

Dictionaries

Variables are “local” if they are in the “active” dictionary otherwise they’re non-local.
Scope Concepts

The lifetime of a variable is the time interval during which the variable has been allocated a block of memory.

Symbol Table

1. Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14> <sort, 21>
   Lifetime: 1 - 33
2. Function B nested scope: <w, 2> <j, 3> <k, 3>
   Lifetime: 2 - 7
3. Function A nested scope: <x, 8> <y, 8> <i, 9> <j, 9>
   Lifetime: 8 - 13
4. Function main nested scope: <a, 15> <b, 15>
   Lifetime: 15 - 20
5. Function sort nested scope: <a, 21> <size, 21> <i, 22> <j, 22>
   Lifetime: 21 - 33
   <t, 26>

Dictionaries

For static scoping, the referencing environment (the set of statements which can validly reference a name) for a name is its defining scope and all nested subscopes.

Reference to i (4) resolves to <i, 1> in Outer scope

In dynamic scoping, a name is bound to its most recent declaration based on the program’s call history. Since main (17) calls → A (10) sets I, A(11) calls → B, the reference to i (4) resolves to <i, 9> in A. What about i at (5)?
Scope Concepts

Disjoint scopes

1. Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14> <sort, 21> 1 - 33
2. Function B nested scope: <w, 2> <j, 3> <k, 3> 2 - 7
3. Function A nested scope: <x, 8> <y, 8> <i, 9> <j, 9> 8 - 13
4. Function main nested scope: <a, 15> <b, 15> 15 - 20
5. Function sort nested scope: <a, 21> <size, 21> <i, 22> <j, 22> 21 - 33

Nested scope

Dictionaries

A name is visible if its referencing environment (the set of statements which can validly reference a name) includes the reference and the name is not declared in an inner scope.

A name redeclared in an inner scope effectively hides the outer declaration.

Some languages provide a mechanism for referencing a hidden name; e.g.: this.x in C++/Java.
Scope Concepts

Disjoint scopes

1. Outer scope: \(<h, 1> <i, 1> <B, 2> <A, 8> <main, 14> <sort, 21>\)
2. Function B nested scope: \(<w, 2> <j, 3> <k, 3>\)
3. Function A nested scope: \(<x, 8> <y, 8> <i, 9> <j, 9>\)
4. Function main nested scope: \(<a, 15> <b, 15>\)
5. Function sort nested scope: \(<a, 21> <size, 21> <i, 22> <j, 22>\)

Nested scope

Dictionaries

Overloading uses the number or type of parameters to distinguish among identical function names or operators.

public class PrintStream extends FilterOutputStream {
    ...
    public void print(boolean b);  
    public void print(char c);  
    public void print(int i);  
    public void print(long l);  
    public void print(float f);  
    ...
}
Types

A type is a collection of values and operations on those values.
Example: Integer type has values ..., -2, -1, 0, 1, 2, ... and operations +, -, *, /, <, ...

    Boolean type has values true and false and operations $\land$, $\lor$, $\neg$.

A type system is a precise definition of the bindings between the type of a variable, its values and the possible operations on those values. It provides a basis for detecting type errors.

A type error is any error that arises because an operation is attempted on a data type for which it is undefined.

Computer types have a finite number of values due to fixed size allocation; problematic for numeric types.
Exception - Haskell type Integer represents unbounded integers.
Types

A language is *statically typed* if the types of all variables are fixed when they are declared at compile time.

A language is *dynamically typed* if the type of a variable can vary at run time depending on the value assigned.

Some languages perform type checking at compile time (eg, C). Other languages (eg, Perl) perform type checking at run time. Still others do both (eg, Java - downcasting - `stringRef = (String) objRef;` ).
Types

A *type conversion* is a narrowing conversion if the resulting type permits fewer bits, thus potentially losing information.

Otherwise it is termed a *widening* conversion.

An operator or function is *overloaded* when its meaning varies depending on the types of its operands or arguments or result.
Implicit Type Conversion

• Assignment supports implicit widening conversions
• We can transform the abstract syntax tree to insert explicit conversions as needed.
• The types of the target variable and source expression govern what to insert.

Example:
Suppose we have an assignment

\[ f = i - \text{int}(c); \]

\( (f, i, \text{ and } c \text{ are float, int, and char variables}). \)

The abstract syntax tree is:
Example continued:
So an implicit widening is inserted to transform the tree to:

Here, **c2i** denotes conversion from char to int, and **itof** denotes conversion from int to float.

Note: **c2i** is an explicit conversion given by the operator int() in the program.
Variables and Memory Addresses

L-value - use of a variable name to denote its address.
   Ex: L-value = …

R-value - use of a variable name to denote its value.
   Ex: … = R-value

Some languages support/require explicit dereferencing.
   Ex: x := !y + 1

Address could be in global, heap (dynamic memory), or stack (e.g., runtime stack – see function discussion) storage.
C Pointer Example

```c
int main ( ) {
    int x, y;
    int *p;
    x = *p;
    *p = 123;
    printf("x: %d, y: %d, p: %d", x, y, p);
}
```

```c
int main ( ) {
    int x = 101, y = 202;
    int *p;
    printf("x: %d, y: %d, p: %d\n", x, y, p);
    p = &y;
    printf("x: %d, y: %d, p: %d\n", x, y, p);
    x = *p;
    *p = 123;
    printf("x: %d, y: %d, p: %d\n", x, y, p);
}
```
Functions and Recursion
Definitions

An *argument* is an expression that appears in a function call.

A *parameter* is an identifier that appears in a function declaration.

In the code on the right the call `A(a, b)` has arguments `a` and `b`.

The function declaration `A` has parameters `x` and `y`.

```c
int h, i;
void B(int w) {
    int j = 1, k = 2;
    i = 2*w;
    w = w+1;
    printf("In B - w, j, k, h, i: %d, %d, %d, %d, %d\n", 
            w, j, k, h, i);
}
void A(int x, int y) {
    float i = 1.1, j = 2.2;
    B(h);
    printf("In A - x, y, i, j, h: %d, %d, %f, %f, %d\n", 
            x, y, i, j, h);
}
int main() {
    int a, b;
    h = 5; a = 3; b = 2;
    A(a, b);
    printf("In Main  a, b, h, i: %d, %d, %d, %d\n", 
            a, b, h, i);
}
```
Parameter Passing Mechanisms

• By value
• By reference
• By value-result
• By name
Compute the value of the argument at the time of the call and assign that value to the parameter.

So passing by value doesn’t normally allow the called function to modify an argument’s value.

All arguments in C and Java are passed by value.

But references can be passed to allow argument values to be modified. E.g., void swap(int *a, int *b) { … }

```
int h, i;
void B(int w) {
    int j = 1, k = 2;
    i = 2*w;
    w = w+1;
    printf("In B - w, j, k, h, i: %d, %d, %d, %d, %d\n",
           w, j, k, h, i);
}
void A(int x, int y) {
    float i = 1.1, j = 2.2;
    B(h);
    printf("In A - x, y, i, j, h: %d, %d, %f, %f, %d\n",
           x, y, i, j, h);
}
int main() {
    int a, b;
    h = 5; a = 3; b = 2;
    A(a, b);
    printf("In Main  a, b, h, i: %d, %d, %d, %d\n",
           a, b, h, i);
}
```

$ ./a
In B - w, j, k, h, i: 6, 1, 2, 5, 10
In A - x, y, i, j, h: 3, 2, 1.100000, 2.200000, 5
In Main  a, b, h, i: 3, 2, 5, 10
Compute the **address** of the argument at the time of the call and assign it to the parameter.

Since h is passed by reference, its value changes during the call to B.
```c
int h, i;
void B(int w) {
    int j = 1, k = 2;
i = 2*w;
w = w + 1;
printf("In B - w, j, k, h, i: %d, %d, %d, %d, %d\n", w, j, k, h, i);
}
void A(int x, int y) {
    float i = 1.1, j = 2.2; B(h);
printf("In A - x, y, i, j, h: %d, %d, %f, %f, %d\n", x, y, i, j, h);
}
int main() {
    int a, b;
h = 5; a = 3; b = 2;
A(a, b);
printf("In Main  a, b, h, i: %d, %d, %d, %d\n", a, b, h, i);
}
```

```
$ ./a
In B - w, j, k, h, i: 6, 1, 2, 5, 10
In A - x, y, i, j, h: 3, 2, 1.100000, 2.200000, 5
In Main  a, b, h, i: 3, 2, 5, 10
```
Pass by Value-Results

Pass by value at the time of the call and/or copy the result back to the argument at the end of the call – also called copy-in-copy-out.
Pass by Name

Textually substitute the argument for every instance of its corresponding parameter in the function body. Originated with Algol 60 (Jensen’s device), but was dropped by Algol’s successors -- Pascal, Ada, Modula.

\[
\text{real procedure Sum}(j, \text{lo}, \text{hi}, \text{Ej});
\]
\[
\text{value lo, hi; integer j, lo, hi; real Ej; begin}
\]
\[
\text{real } S; \ S := 0;
\]
\[
\text{for } j := \text{lo} \text{ step 1 until } \text{hi} \text{ do}
\]
\[
S := S + \text{Ej};
\]
\[
\text{Sum := S}
\]
\[
\text{end;
}\]

Exemplifies \textit{late binding}, since evaluation of the argument is delayed until its occurrence in the function body is actually executed.

Associated with \textit{lazy evaluation} in functional languages (e.g., Haskell).
A function that can call itself, either directly or indirectly, is a recursive function.

```c
$ cat test.c
int factorial (int n) {
    int i;
    if (n < 2)
    {
        printf("factorial returning 1\n");
        return 1;
    }
    else
    {
        i = n * factorial(n-1);
        printf("factorial returning %d\n", i);
        return i;
    }
}

int main() {
    printf("factorial(3) returns: %d\n", factorial(3));
}

$ ./a
factorial returning 1
factorial returning 2
factorial returning 6
factorial(3) returns: 6
```
A stack of activation records:

- Each new call pushes an activation record, and each completing call pops the topmost one.
- So, the topmost record is the most recent call, and the stack has all active calls at any run-time moment.
A block of information associated with each function call, which includes:

- **parameters and local variables**
- **Return address**

“We call this constructed value a *closure* because it “closes” the function body over the substitutions that are waiting to occur. When the interpreter encounters a function application, it must ensure that the function’s pending substitutions aren’t forgotten. It must, however, ignore the substitutions pending at the location of the invocation, for that is precisely what led us to dynamic instead of static scope. It must instead use the substitutions of the invocation location to convert the function and argument into values, hope that the function expression evaluated to a closure, then proceed with evaluating the body employing the repository of deferred substitutions stored in the closure.”

[Textbook. Pages 46-47]
int h, i;
void B(int w) {
    int j, k;
    i = 2*w;
    w = w+1;
}
void A(int x, int y) {
    bool i, j;
    B(h);
}
int main() {
    int a, b;
    h = 5; a = 3; b = 2;
    A(a, b);
}

• parameters and local variables
• Return address
• Saved registers
• Temporary variables
• Return value
Hmm Runtime Stack for Factorial 3

```c
int factorial(int n) {
    if(n < 1) {
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int number, answer;
    number = 3;
    answer = factorial(number);
    print(answer);
}
```
6.1 A Taxonomy of Functions

- **first-order** Functions are not values in the language. They can only be defined in a designated portion of the program, where they must be given names for use in the remainder of the program. The functions in F1WAE are of this nature, which explains the 1 in the name of the language.

- **higher-order** Functions can [be defined anywhere in a program and] return other functions as values.

- **first-class** Functions are values with all the rights of other values. In particular, they can be supplied as the value of arguments to functions, returned by functions as answers, and stored in data structures. [They can also be defined anywhere in a program.]
Binding

- A binding is an association between an entity (such as a variable) and a property (such as its value).
- A binding is static if the association occurs before run-time.
- A binding is dynamic if the association occurs at run-time.
- Name bindings play a fundamental role.
- The lifetime of a variable name refers to the time interval during which memory is allocated.
## Variables

<table>
<thead>
<tr>
<th></th>
<th>Static Binding</th>
<th>Dynamic Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lifetime</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Static v. Dynamic Scoping

### Static v. Dynamic Typing

### Static v. Dynamic Binding

<table>
<thead>
<tr>
<th></th>
<th>Static Binding</th>
<th>Dynamic Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lifetime</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(with (x 1) (+ x x))
(λx. (+ x x) 1)

(parse ' (with (x 1) (+ x x)))
(app (fun 'x (add (id 'x) (id 'x))) (num 1))

(interp (parse ' (with (x 1) (+ x x))) (mtSub))
(numV 2)

(let ((x 1)) (+ x x))
2

((lambda (x) (+ x x)) 1)
2
(with (x 2) (with (y 1) (+ x y)))
((λx.λy.(+ x y) 2) 1)

(parse '(with (x 2) (with (y 1) (+ x y))))
(app (fun 'x (app (fun 'y (add (id 'x) (id 'y))) (num 1)))
(num 2))

(interp (parse '(with (x 2) (with (y 1) (+ x y)))) (mtSub))
(numV 3)

(let ((x 2)) (let ((y 1)) (+ x y)))
3

(((lambda (x) (lambda (y) (+ x y))) 2) 1)
3
PLAI Chapters 4, 5 and 6

(with (x 5) (with (f (fun (y) (+ x y))) (with (x 3) (f 4)))))
(((λx.λf.λx.(f 4) 5) (λy.(+ x y))) 3)

(λy.(+ x y) 4)

-----
X = 3
-----
f = (λy.(+ x y))
    { x = 5 }
-----
X = 5

Static Scoping:
(interp (parse '(with (x 5) (with (f (fun (y) (+ x y))) (with (x 3) (f 4)))))
(numV 9)

Dynamic Scoping:
(interp (parse '(with (x 5) (with (f (fun (y) (+ x y))) (with (x 3) (f 4)))))
(numV 7)

(let ((x 5)) (let ((f (lambda (y) (+ x y)))) (let ((x 3)) (f 4))))
9

(let ((z 5)) (let ((f (lambda (y) (+ x y)))) (let ((x 3)) (f 4))))
... reference to an identifier before its definition: x
Chapter 4, Page 27 – “To add functions to WAE, we must define their concrete and abstract syntax. In particular, we must both describe a function definition, or declaration, and provide a means for its use, also known as an application or invocation.”

Chapter 5, Page 33 – “Well, if the program has size n (measured in abstract syntax tree nodes), then each substitution sweeps the rest of the program once, making the complexity of this interpreter at least O(n²). That seems rather wasteful; surely we can do better. How do we avoid this computational redundancy? We should create and use a repository of deferred substitutions (in the lisp world this is usually called an Environment). Concretely, here’s the idea. Initially, we have no substitutions to perform, so the repository is empty. Every time we encounter a substitution (in the form of a with or application), we augment the repository with one more entry, recording the identifier’s name and the value (if eager) or expression (if lazy) it should eventually be substituted with. We continue to evaluate without actually performing the substitution.”

Chapter 5, Page 36 – “Definition 10 (Static Scope) In a language with static scope, the scope of an identifier’s binding is a syntactically delimited region. A typical region would be the body of a function or other binding construct. In contrast: Definition 11 (Dynamic Scope) In a language with dynamic scope, the scope of an identifier’s binding is the entire remainder of the execution during which that binding is in effect.”
PLAI Chapters 4, 5 and 6

Chapter 5, Page 37 – “Therefore, at the point of invoking a function, our new interpreter must “forget” about the current substitutions . . . That is, we use the empty repository to initiate evaluation of a function’s body, extending it immediately with the formal parameter but no more.”

Chapter 6, Pages 41 & 42 – “first-order Functions are not values in the language. They can only be defined in a designated portion of the program, where they must be given names for use in the remainder of the program. The functions in F1WAE are of this nature, which explains the 1 in the name of the language.

higher-order Functions can return other functions as values.

first-class Functions are values with all the rights of other values. In particular, they can be supplied as the value of arguments to functions, returned by functions as answers, and stored in data structures.
Chapter 6, Page 46 – 
{with {x 3} {with {f {fun {y} {+ x y}}} {with {x 5} {f 4}}} }
(let ((x 3)) (let ((f (lambda (y) (+ x y)))) (let ((x 5)) (f 4)))

. . .

We call this constructed value a closure because it “closes” the function body over the substitutions that are waiting to occur.”

Chapter 6, Page 47 – “Crucially, while we evaluate arg-expr in ds, the repository active at the invocation location, we evaluate the function’s body in its “remembered” repository.

. . .

The C programming language implements a middle-ground by allowing functions (technically, pointers to functions) to serve as values but allowing them to be defined only at the top-level. Because they are defined at the top-level, they have no deferred substitutions at the point of declaration; this means the second field of a closure record is always empty, and can therefore be elided, making the function pointer a complete description of the function. This purchases some small implementation efficiency, at the cost of potentially great expressive power, as the examples in Section 6.5 illustrate.
Lisp Function Definition (Declaration) and Application (Invocation)

Using Scheme8 on the class calendar webpage, evaluate the following:

```
(interp (parse (quote
(with ((z 7))
(with ((z 3) (a 5) (x (fun (x y) (- x (+ y z)))))
(with ((z 10) (a 5)) (x z a)))))
(mtEnv))
```

The (parse (quote (with ((z 7)) . . . evaluates to the following:

```
(with
(list (binding 'z (num 7)))
(with
(list (binding 'z (num 3)) (binding 'a (num 5)) (binding 'x (fun (list 'x 'y) (binop - (id 'x) (binop + (id 'y) (id 'z))))))
(with (list (binding 'z (num 10)) (binding 'a (num 5))) (app (id 'x) (list (id 'z) (id 'a))))))
```

--------------------------------------------------------------------------------------------------------------------------------------------
------

• Where are the assignments?

• Where are the global variables?

• Are there any side affects?