Names, Scopes, and Bindings

ORGANIZATION OF PROGRAMMING LANGUAGES

JUCHEOL MOON

Name, Scope, and Binding

- A name is exactly what you think it is
  - Most names are identifiers
- A binding is an association between two things, such as a name and the thing it names
- The scope of a binding is the part of the program (textually) in which the binding is active

Binding Time

- language design time (3)
- language implementation time (4)
- program writing time (2)
- compile time (1)

1. static data in memory
2. function (method) names
3. possible data type
4. maximum size of heap and stack

Binding Time

- link time (3)
- load time (1)
- run time (2)

1. physical addresses to instructions
2. dynamic data in memory
3. object in another module (library)

- The terms static and dynamic are generally used to refer to things bound before run time and at run time, respectively

Binding Time

- In general, early binding times are associated with greater (**efficiency / flexibility**)
- Later binding times are associated with greater (**efficiency / flexibility**)
- Compiled languages tend to have (**early / later**) binding times
- Interpreted languages tend to have (**early / later**) binding times

Lifetime

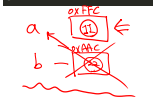
- Key events
  - creation of objects
  - creation of bindings
  - references to variables (which use bindings)
  - (temporary) deactivation of bindings
  - reactivation of bindings
  - destruction of bindings
  - destruction of objects

Lifetime

- The period of time from creation to destruction is called the LIFETIME of a binding
  - If object outlives binding it's garbage
  - If binding outlives object it's a dangling reference
- The textual region of the program in which the binding is *active* is its scope

```
void garbage() {
    int* a = new int(11);
    cout << *a << endl;
    int b = 22;
    a = &b;
    cout << *a << endl;
    delete a;
}

int* dangling() {
    int* c = new int(33);
    return c;
}
```



Storage Management

- Storage allocation mechanisms
  - static
  - stack
  - Heap
- Static allocation for
  - arguments
  - local variables
  - temporaries

Recursive Subroutine

- Factorial Function

```
int fact (int n)
{
  if (n == 1) return 1;
  else return n * fact(n - 1);
}
```

Recursive Subroutine

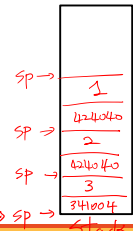
- $a0=3 and the instruction "0x341000 jal 0x424000"

PC	Code
0x424000	addi $sp, $sp, -8
0x424004	sw $ra, 4($sp)
0x424008	sw $a0, 0($sp)
0x424012	bgt $a0, $1, 0x424032
0x424020	addi $v0, $zero, 1
0x424024	addi $sp, $sp, 8
0x424028	jr $ra
0x424032	addi $a0, $a0, -1
0x424036	jal 0x424000
0x424040	lw $a0, 0($sp)
0x424044	lw $ra, 4($sp)
0x424048	addi $sp, $sp, 8
0x424052	mul $v0, $a0, $v0
0x424056	jr $ra

$ra = 0x341004 → 0x424040

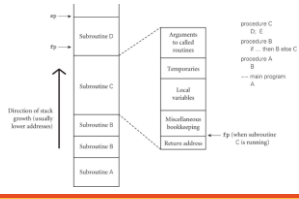
$a0 = 3 → 2 → 1

$v0 = 6



Storage Management

- Contents of a stack frame
  - Assigned at compile time
  - arguments
  - local vars.
  - temporaries
  - return addr.



Storage Management

- Maintenance of stack is responsibility of calling subroutines
  - How can we (compilers) write **time efficient** and **space efficient** assembly codes?

↑ using stack

↓ using stack

Leaf Procedure Example

- MIPS code:

```
leaf_example:
  addi $sp, $sp, -4
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero
  lw   $s0, 0($sp)
  addi $sp, $sp, 4
  jr   $ra
```

- C code:

```
int leaf_example (int g, h, i, j) {
  int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, …, j in $a0, …, $a3
- f in $s0 (hence, need to save $s0 on sta
- Result in $v0

Name Declarations

- Some constructs must first be introduced by explicit declarations
  - int javaNum;
- Some constructs can be introduced by implicit declarations
  - pyNum = someNum + 10

Name Scope

- Once a name is declared, how long is that declaration valid?
  - ___file___?
  - ___program___?
  - ___function___?
- C/C++ scoping example
  - Declarations are valid in ___block___ that they are declared
  - Declarations not in a block are global, unless the `static` keywords is used, in which case the declaration is valid in that file only

C++ example

```
#include <iostream>
using namespace std;
string x = "Global";

int main()
{
    {
        string x = "Block 1";
        cout << x << endl;
    }
    {
        string x = "Block 2";
        cout << x << endl;
    }
}
```

string x = "what?";

Scope Rules

- A scope is a program section of maximal size in which no bindings change, or at least in which no ___re-declaration___ are permitted.
- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - ___create___ bindings for new local variables,
  - ___deactivate___ bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope)

Scope Rules

- On subroutine exit:
  - ___destroy___ bindings for local variables
  - ___reactive___ bindings for global variables that were deactivated