Class Information

► Homework 4 due this Friday 11:55pm.
► Project 1 is posted, due Sunday 10/23 11:55pm.
Project 1: Overview

**Compiler**
- TinyL program
  - Example: `test1`
- Compile
  - Example: `compile test1`
- TinyL.out
  - Output always "tinyL.out"

**Optimizer**
- RISC machine code
  - Example: `tinyL.out`
- Optimize
  - Example: `optimize < tinyL.out`
- RISC machine code
  - Output to stdout

**Virtual Machine**
- RISC machine code
  - Example: `tinyL.out`
- Run
  - Input and output of execution
Redundant Code Elimination: Eliminate code without changing the semantics of the program. If the execution of an operation or instruction does not contribute to the input/output behavior of the program, the instruction is considered dead code and therefore can be eliminated.

Example:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Optimized Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOADI Rx #c1</td>
<td>LOADI Rx #c1</td>
</tr>
<tr>
<td>LOADI Ry #c2</td>
<td>LOADI Ry #c2</td>
</tr>
<tr>
<td>LOADI Rz #c3</td>
<td>ADD R1 Rx Ry</td>
</tr>
<tr>
<td>ADD R1 Rx Ry</td>
<td>STORE a R1</td>
</tr>
<tr>
<td>MUL R2 Rx Ry</td>
<td>WRITE a</td>
</tr>
<tr>
<td>STORE a R1</td>
<td></td>
</tr>
<tr>
<td>WRITE a</td>
<td></td>
</tr>
</tbody>
</table>

See project description for more details.
Review: Storage Management

- Static objects are given an absolute address that is retained throughout program’s execution.

- Stack objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns.

- Heap objects may be allocated and deallocated at arbitrary times.
Maintaining Free List

- **allocate**: continuous block of memory; remove space from free list (here: linked list).

- **free**: return to free list after coalescing with adjacent free storage (if possible); may initiate compaction.
Review: What went wrong?

“Aliasing” and freeing memory

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int *a = NULL; int *b = NULL; int *c = NULL;

    a = (int *) malloc(sizeof(int));
    b = a; *a = 12;
    printf("%x %x: %d\n", &a, a, *a);
    printf("%x %x: %d\n", &b, b, *b);
    free(a);
    printf("%x %x: %d\n", &b, b, *b);

    c = (int *) malloc(sizeof(int));
    *c = 10;
    printf("%x %x: %d\n", &c, c, *c);
    printf("%x %x: %d\n", &b, b, *b);
}
```

```
> a.out
effff60c 209d0: 12
effff608 209d0: 12
effff608 209d0: 12
effff604 209d0: 10
effff608 209d0: 10
```
What went wrong?

Use a subroutine to create an object

```c
#include <stdio.h>
#include <stdlib.h>

/* TYPE DEFINITION */
typedef struct cell listcell;
struct cell
{ int num;
  listcell *next;
};

listcell *head = NULL;

listcell *create_listcell() {
  listcell new;
  new.num = -1; new.next = NULL;
  return &new;
}

int main (void) {
  head = create_listcell();
  printf("head->num = %d\n", head->num);
}

> gcc stack.c
~ stack.c: In function ‘create_listcell’:
~ stack.c:17: warning: function returns address of local variable
> ./a.out
head->num = -1
```
What went wrong?

Use a subroutine to create an object: malloc

```c
#include <stdio.h>
#include <stdlib.h>

/* TYPE DEFINITION */
typedef struct cell listcell;
struct cell
{
    int num;
    listcell *next;
};

listcell *head = NULL;

listcell *create_listcell() {
    listcell *new;
    new = (listcell *) malloc(sizeof(listcell));
    new->num = -1; new->next = NULL;
    return new;
}

int main (void) {
    head = create_listcell();
    printf("head->num = %d\n", head->num);
}
```

> gcc heap.c
> ./a.out

head→num = -1
Zheng Zhang
Pointers and arrays are similar in C:

- array name is pointer to a[0]:
  
  ```c
  int a[10];
  int *pa;
  pa = &a[0];
  ```

  pa and a have the same semantics

- pointer arithmetic is array indexing
  
  pa+1 and a+1 point to a[1]

- exception: an array name is not a variable
  
  a++ is ILLEGAL
  
  a=pa is ILLEGAL (pa=a is LEGAL!)
What’s in a name? — each name “means” something!

- denotes a programming language construct
- has associated “attributes” (e.g.: type, memory location, read/write permission, storage class, access restrictions, etc.)
- has a meaning, i.e., represents a semantic object (e.g.: a type description, an integer value, a function value, a memory address, etc.)
Names, Bindings, and Memory

**Binding** – association of a name with the thing it “names” (e.g., a name and a memory location, a function name and its “meaning”, a name and a value)

- **Compile time** – during compilation process – static (e.g.: macro expansion, type definitions)
- **Link time** – separately compiled modules/files are joined together by the linker (e.g., adding the standard library routines for I/O (stdio.h), external variables)
- **Run time** – when program executes – dynamic

Compiler needs bindings to know meaning of names during translation (and execution).
Binding Times - Choices

- **Early binding** times – more efficient (faster) at runtime
- **Late binding** times – more flexible (postpone binding decision until more “information” is available)

Examples of static binding (early):
- functions in C
- types in C

Examples of dynamic binding (late):
- method calls in Java
- dynamic typing in JavaScript, Scheme

Note: dynamic linking is somewhat in between static and dynamic binding; the function signature has to be known (static), but the implementation is linked and loaded at run time (dynamic).
How to Maintain Bindings

- symbol table: maintained by compiler during compilation
  names ⇒ attributes

- environment: maintained by compiler generated code during program execution
  names ⇒ memory locations

Questions

- How long do bindings for a name hold in a program?
- What initiates a binding?
- What ends a binding?
**Scope Example**

Algol-like Programming Languages

```plaintext
program L;
    var n: char;    {n declared in L}

    procedure W;
    begin
        write(n); {n referenced in W}
    end;

    procedure D;
    var n: char; {n declared in D}
    begin
        n:= 'D'; {n referenced in D}
        W
    end;

begin
    n:= 'L'; {n referenced in L}
    W;
    D
end.
```

Zheng Zhang 15 eddy.zhengzhang@cs.rutgers.edu
Lexical Scope

- Non-local variables are associated with declarations at compile time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

Example:
- The reference to $n$ in $W$ is associated with the declaration of $n$ in $L$
- The output is?
Dynamic Scope

- Non-local variables are associated with declarations at run time
- Find the most recent, currently active run-time stack frame containing a declaration of the variable

Example:
  - The reference to \( n \) in \( W \) is associated with two different declarations at two different times
  - The output is?
Procedure Activations

- Begins when control enters activation (call)
- Ends when control returns from activation

Example:
Run-time stack contains frames for main program and each active procedure.

Each stack frame includes:

1. Pointer to stack frame of caller (control link for stack maintainance and dynamic scoping)
2. Return address (within calling procedure)
3. Mechanism to find non-local variables (access link for lexical scoping)
4. Storage for parameters, local variables, and final values
Next Lecture

Things to do:
Start working on the project. Due Sunday October 23.

Read Scott: Chap. 3.1 - 3.4, 9.1 - 9.3 ;

Next time:

▶ Access link and display, parameter passing styles and their implementation.