HAVING A FAMILY MEETING IN A TREE ISN'T NORMAL. BUT ON CATNIP IT IS.
lab discussion

func1:
  pushl  %ebp
  movl  %esp, %ebp
  subl  $24, %esp
  movl  8(%ebp), %eax
  movl  $259, (%eax)
  movl  8(%ebp), %eax
  movl  (%eax), %eax
  movl  $globalvar, 4(%esp)
  movl  %eax, (%esp)
  call  func2
  leave
  ret

func2:
  pushl  %ebp
  movl  %esp, %ebp
  movl  12(%ebp), %eax
  imull  8(%ebp), %eax
  popl  %ebp
  ret
Questions?
Using Nested Arrays

• **Strengths**
  
  – C compiler handles doubly subscripted arrays
  
  – Generates very efficient code
  
  – Avoids multiply in index computation
  
  – (See figure 3.28)

• **Limitation**
  
  – Only works for fixed array size

```c
#define N 16
typedef int fix_matrix[N][N];

/* Compute element i,k of fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```

![Diagram of matrices and index notation](image)
Dynamic Nested Arrays

• **Strength**
  – Can create matrix of any size

• **Programming**
  – Must do index computation explicitly

• **Performance**
  – Accessing single element costly
  – Must do multiplication

```c
int * new_var_matrix(int n)
{
    return (int *)
    calloc(sizeof(int), n*n);
}
```

```c
int var_ele
    (int *a, int i, int j, int n)
{
    return a[i*n+j];
}
```

```assembly
movl 12(%ebp),%eax  # i
movl 8(%ebp),%edx  # a
imull 20(%ebp),%eax  # n*i
addl 16(%ebp),%eax  # n*i+j
movl (%edx,%eax,4),%eax  # Mem[a+4*(i*n+j)]
```
Dynamic Array Multiplication

- Without Optimizations
  - Multiplies: 3
    - 2 for subscripts
    - 1 for data
  - Adds: 4
    - 2 for array indexing
    - 1 for loop index
    - 1 for data

```c
/* Compute element i,k of variable matrix product */
int var_prod_ele
  (int *a, int *b,
   int i, int k, int n)
{
  int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result +=
      a[i*n+j] * b[j*n+k];
  return result;
}
```
Optimizing Dynamic Array Multiplication

- **Optimizations**
  - Performed when set optimization level to -O2

- **Code Motion**
  - Expression \(i \times n\) can be computed outside loop

- **Strength Reduction**
  - Incrementing \(j\) has effect of incrementing \(j \times n + k\) by \(n\)

- **Operations count**
  - 4 adds, 1 mult

- **Compiler can optimize regular access patterns**

```c
{ 
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}

{ 
    int j;
    int result = 0;
    int iTn = i*n;
    int jTnPk = k;
    for (j = 0; j < n; j++) {
        result +=
            a[iTn+j] * b[jTnPk];
        jTnPk += n;
    }
    return result;
}
Structures

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

- **Concept**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

- **Accessing Structure Member**

```c
void set_i(struct rec *r, int val)
{
    r->i = val;
}
```

**IA32 Assembly**

```
# %eax = val
# %edx = r
movl %eax, (%edx)  # Mem[r] = val
```
Generating Pointer to Structure Member

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

<table>
<thead>
<tr>
<th>i</th>
<th>a[3]</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>16 20</td>
</tr>
</tbody>
</table>

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time

```c
int *find_a
    (struct rec *r, int idx)
{
    return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```
Structure Referencing (Cont.)

• C Code

```c
struct rec {
    int i;
    int a[3];
    int *p;
};

void set_p(struct rec *r) {
    r->p = &r->a[r->i];
}
```

```c
# %edx = r
movl (%edx),%ecx       # r->i
leal 0(,%ecx,4),%eax   # 4*(r->i)
leal 4(%edx,%eax),%eax  # r+4+4*(r->i)
movl %eax,16(%edx)      # Update r->p
```