CSCE 531  Compiler Construction

Lecture 21
Arrays

Topics

- Arrays
- Test 2 expectations

Readings: 8.2

April 3, 2006
Overview

Last Time

- Final reductions of statement lists in programs/functions/procedures
- Passing Parameters
- Types Revisited – type expression trees
  - Offset within function, widths of types
  - Structural equivalence, name equivalence
  - Type encoding – why and limitations

Today’s Lecture

- Test 2 - comments (7:15 PM 300 Main B213)
- Arrays in expressions
- Code Generation

References: Section 8.3

Next Time: Test 2 - 7:15 PM 300 Main B213
Recap – 2nd thoughts on Last lecture

To handle full complexity of types – type expressions

- Instead of enumerated (int/float and level of project 3)
  use pointer to expression tree and width

- Loops in expression trees

- Structural equivalence by tree traversing

Offsets variable – not attribute but global

- need one for global data and

- one for current function (non-nested)
  - Reinitialize offsetInCurrentFunction = 0 at the start of each function
  - Also start new “symbol table” for this function
Recall: Types within the Declarations

\[ T \rightarrow \text{integer} \quad \{ \ T.\text{type} = \text{INTEGER}; \ T.\text{width} = 4; \ \} \]

\[ \quad | \quad \text{double} \quad \{ \ T.\text{type} = \text{double}; \ T.\text{width} = 8; \ \} \]

\[ \quad | \quad \text{array} \ [\ 'num' \ ] \ of \ T1 \ \{ \]

\[ \quad \quad \quad \quad \quad \ T.\text{type} = \text{array}(\text{num}\.\text{val}, \ T1\.\text{type}); \]

\[ \quad \quad \quad \quad \quad \ T.\text{width} = \text{num}\.\text{val} \times T1\.\text{width}; \]

\[ \quad \}

\[ \quad | \quad \ast \ T1 \quad \{ \ T.\text{type} = \text{pointer}(T1\.\text{type}); \]

\[ \quad \quad \quad \quad \quad \ T.\text{width} = 4; \]

\[ \quad \}

Function that enters the information on this array into the symbol table / type expression tree
Handling Arrays

 Declarations – handled last time

- Pascal-like:
  - var table, x, : array[1..5, -7..3] of integer;
- C-like
  - int intensity[23][80], x;
- As we parse we build the expression tree
- Saving dimensions/limits on indices, widths

Array references in expressions

- \( x = 2 * \text{intensity}[11][23] + 7; \)

- What do we generate?
One Dimensional Arrays

Addressing Example

```c
int a [low .. high];
```

Addressing a[i]

Skip “i – low” elements of width “size”

```c
&a[i] = &a[0] + (i - low) * size
```

base address

Lowest address
Two Dimensional Arrays

Row major order –
Row with index zero, followed by row with index 1, ...

Example

int a[6][3];

Addressing a[i][j]

Skip i rows then skip j elements

&a[i][j] = &a[0][0] + i*numcolumns*size + j*size

Lowest address

Row 0

Row 1

Row 2

Row 4

CSCE 531 Spring 2006
Two Dimensional Arrays with lowest index 1

Row major order –
Row with index one, followed by row with index 2, …

Example

int a[6][3];

Addressing a[i][j]
Skip i – 1 rows then skip j-1 elements
&a[i][j] = &a[0][0]
   + (i-1) * numcolumns * size
   + (j-1) * size

So why start arrays at 0?

Lowest address

Row 1
a[6][2]
a[6][1]
a[6][0]
a[5][2]
a[5][1]
a[5][0]
a[4][2]
a[4][1]
a[4][0]
a[3][2]
a[3][1]
a[3][0]
a[2][2]
a[2][1]
a[2][0]
a[1][2]
a[1][1]
a[1][0]

Row 2
Row 3

Row n
IA 32 support for Array Indexing

The most general address for the Intel 32 bit architecture is

\[
\text{disp}(\text{baseReg, IndexReg, scale})
\]

this specifies the address

\[
\text{disp} + \text{base} + \text{index} \times \text{scale}
\]

As an example

For instance in addressing

\[
\text{double } x[100];
\]

Starting in location 0x200

Addressing \( x[7] \)

\[
\%esi \ 0x200
\]

\[
\%edi \ 7
\]

Then the statement

\[
\text{sum} = \text{sum} + x[i];
\]

might be handled with

\[
\text{addl}(\%esi, \%edi, 8), -8(\%ebp)
\]
IA 32 support for 2-Dimensional Arrays

double d[10][5];

\&d[i][j] = base + i * numColumns * size + j * size

= base + (i * numColumns + j) * size

So

addl (%esi, %edi, 8), %eax

After putting \&d[0][0] in %esi calculate i*numColumns+j
and put this in %edi and set the scale to the width of
the double
**IA 32 Instructions**

`movl addr1, addr2` - move value from `addr1` to `addr2`

`addl addr1, addr2` - add the value @ `addr1` to value @ `addr2`

`subl, imult, incrl, cmpl`

`shll $n, addr` - shift logical left the value @ `addr` `n` places

`leal d(base, index, scale), reg` - “load effective address”
   calculate address `d + base + index*scale` and put this address in `reg`
Arrays/array.c

main(){
    int a[100][7];
    int i, j, r;
    ...
    r=func(a, 100, 7);
}

int func(int array[][7], int i, int j){
    int local[12];
    int temp;
    int m,n;
    for(m=0; m < i; ++m)
        for(n=0; n < j; ++n)
            temp = temp + array[m][n];
    ...
}

Main.s

...pushl $7pushl $100leal -2824(%ebp), %eaxpushl %eaxcall func...

Notes

1. Note 100*7*4 +3*4 =2812 bytes of local variables
2. Three caller-save registers are saved 12 more bytes
3. Note the third arg is pointer to start of the array
4.
array.s/func

func:

    pushl   %ebp
    movl   %esp, %ebp
    subl   $72, %esp
    movl   $0, -64(%ebp)

.L9:
    movl   -64(%ebp), %eax
    cmpl   12(%ebp), %eax
    jge    .L10
    movl   $0, -68(%ebp)

.L12:
    movl   -68(%ebp), %eax
    cmpl   16(%ebp), %eax
    jge    .L11

Notes

1. normal prologue
2. 72 bytes for local storage
3. Initialize local variable to 0 (loop index m)
4. %eax = “m”
5. compare “m” with arg2=“i”
6. Exit outer loop?
7. Initialize local “n” to 0
8. Compare “n” to arg3=“j”
9. Exit inner loop?
array.s/func Page 2

Notes

1. `sall $3, %eax` shift 3 bits = multiply %eax by 8

2. `%eax = 7 * %edx`

3. Why not just multiply by 7?

4. `%edx = %eax * 4`

5. Not very smart here as `incl -64(%ebp)` is better
Use of Pointers to Improve Efficiency

/* Array3.c Assuming summing the full array */

int func(int array[][7], int i, int j)
{
    int local[12];
    int temp;
    int m, n, *p;

    p = &array[0][0];
    for(m=0; m < i; ++m)
        for(n=0; n < j; ++n)
            temp = temp + *p++;
    return(temp);
}
Optimization levels

- man gcc
- Page of optimization options
  - E.g. -funroll-loops
- Optimization Levels
  - O1 – optimize to make code smaller and faster
  - O2 – optimize more
  - O3 – optimize even more
- Optimizer rewrites code for inner loop to use pointers
- gcc –O2 –S array.c -o array.O2.s
gcc –O2 –S array.c –o arrayO2.s

func:

pushl %ebp
movl %esp, %ebp
pushl %edi
pushl %esi
pushl %ebx
subl $92, %esp
movl 12(%ebp), %edx
movl $0, -92(%ebp)
cmpl %edx, -92(%ebp)
movl 16(%ebp), %ebx
jge .L11
movl $0, -76(%ebp)

.L9:
xorl %ecx, %ecx
cmpl %ebx, %ecx
jge .L13
movl -76(%ebp), %esi

Notes:
1. Compiling using the optimizer
2. Callee-save registers
   - %edi, %esi, %ebx
3. 92 bytes of local storage
4. 12(%ebp) = arg 2 ➔ edx
   (outer loop limit)
5. -92(%ebp)
arrayO2.s Page2

Notes
1. edx = edi + esi * 4
2. Inner loop
   - ecx contain for index “n”
3. x
arrayO2.s Page2

.L11:
  addl  $92, %esp
  popl  %ebx
  popl  %esi
  popl  %edi
  popl  %ebp
  ret

Notes
1. Inner loop
   - %ecx contain for index “n”
2. x
Array references in Source Code

\[ \text{sum} = \text{sum} + d[i][j]; \]
- A right hand side value (rval) refers to the value in \( d[i][j] \)

\[ a[i][j] = 32; \]
- A left hand side value (lval) refers to the address of the memory location whose value should be updated by the assignment

Note the variable sum above has lval and an rval also.
Extending Grammar for Expressions

S → L := E
E → E * E | E + E | ( E )
E → L
L → Elist ]
L → id
Elist → Elist , E
Elist → id [ E
Generalizing the Array Address Indexing Formula to 3rd and Higher Dimensions

In this discussion of Addressing \(a[i_1, i_2, \ldots i_m]\) let:

- \(n_k = \text{the number of elements in the } k\text{th dimension}\)
- \(\text{After factoring out width, the width of the element type}\)
- \(\text{Address of } a[i_1, i_2, \ldots i_p] \text{ is } f \times \text{width}\)

- **Dimension = 1**
  \[i_1\]
  \[e_1 = i_1\]

- **Dimension = 2**
  \[(i_1 \times n_2 + i_2)\]
  \[e_2 = e_1 \times n_2 + i_2\]

- **Dimension = 3**
  \[(i_1 \times n_2 + i_2) \times n_3 + i_3\]
  \[e_3 = e_2 \times n_3 + i_3\]

- **Dimension = p**
  \[\ldots((i_1 \times n_2 + i_2) \ldots \times n_p + i_p)\]
  \[e_p = e_{p-1} \times n_p + i_p\]
Attributes for Elist - Index Expr List

- For Elist → Elist, E
  - we need to be able to evaluate the recursion \( e_m = e_{m-1} \times n_m + i_m \)

- For Elist
  - Elist.array - pointer to the symbol table for the array
  - Elist.ndim - number of dimensions
  - Elist.place - place for offset calculation (pointer to the symbol table)

- For L
  - L.place
  - L.offset - place for offset calculation
Elist – Semantic Actions

Elist \rightarrow \text{id} [ \ E \ { \ Elist.array = \text{id}.place; \ Elist.place = \text{E}.place; \ Elist.ndim = 1; \ }

Elist \rightarrow \text{Elist}_1 , \ E \ { \ t = \text{newtemp}(); \ m = \text{Elist}_1.ndim + 1; \text{emit}(t := \text{Elist}_1.place * \text{numcols(Elist}_1.array, m);); \text{emit}(t := t + \text{E}.place;); \text{Elist.array = \text{Elist}_1.array; \ Elist.place = t; \ Elist.ndim = m; \}

**L – Semantic Actions**

\[
L \rightarrow \text{id} \quad \{ \text{L.place} = \text{id.place}; \\
\quad \text{L.offset} = \text{null}; \quad /* \text{non-array} */
\}
\]

\[
L \rightarrow \text{Elist \ ']' \quad \{ \text{L.offset} = \text{newtemp}(); \\
\quad \text{L.offset} = \text{newtemp}(); \\
\quad \text{emit(L.place := Elist.array->base}; \\
\quad \text{emit(L.offset := Elist.place \ast width(Elist.array);}
\}
\]
\[ E \rightarrow L \{ \]

if L.offset = null then
\[
    E.place = L.place;
\]

else{

    E.place = newtemp();

    emit(E.place := L.place[L/offset]);
}

}
Fig 8.18 Annotated Parse Tree $A[y,z]$
One more thing about Arrays

I have been telling you that C started subscripts at zero for simplicity and efficiency.

Actually that is true but compilers can generate code that is just as efficient for more general array references. It's just more work on the compiler!

E.g.   \[a[low1..high1][low2..high2][low3..high3]\]

Static (compile time) versus Dynamic (run time)

\[
\begin{align*}
(...)((i_1 - low) * n_2 + (i_2 - low)) ... * n_p + (i_p - low_p)) + base \\
= (...((i_1 * n_2 + i_2) ... * n_p + i_p) \\
- (...((low_1 * n_2 + low_2) ... * n_p + low_p)) \\
+ base
\end{align*}
\]

So what of this computation is dynamic (run-time) and what is static (compile-time)?
Test 2 - Review

**Coverage**
- Lec12-LRTableConstruction
- Lec13-SemanticActions.pdf
- Lec14-Booleans
- Lec15-ControlFlowB
- Lec16-ControlFlowII
- Lec17-RunTime
- Lec18-Procedures
- Lec19-NestedScope
- Lec20-TypesOffsets

**Sample Test** (actual test from Summer 2004)
What’s Missing from Sample Test2!

- Handling pointers to base types and definitions and references
- Activation records
  - C-like
  - With nested lexical scope (access links)
  - Given C function draw its activation record
  - Given an IA32 assembly language draw activation record and identify local variables/arguments
- Function calls
  - Semantic actions for definitions and
  - Semantic actions for invocations
- Type expressions
Sample Test (actual test from Summer 04)

1. Given the grammar below show the set of LR(0) sets of items. (Earlier Test)

2. Given the grammar below show the sets of LR(1) items.

3. Given the LR(1) sets of items below show the rows of the LR(1) parse table corresponding to states ...

4. Given the LR(1) sets of items below show the LALR(1) parse table.

5. Provide semantic actions for the middle break loop which has the following syntax
   \[ S \rightarrow \text{loop } S1 \text{ until B S2 endloop } ';' \text{ Too easy for you!} \]

6. Explain how in Yacc/Bison or Flex/Lex to accomplish the following: ...

7. For the situation shown below describe the types of the attribute stack and explain what semantic actions are for the "S \rightarrow \text{IF B THEN M S ELSE N M S}"
Other Sample Questions

Problems from the text

- 8.12 – three address code generated from array references
- 8.14 – semantic actions for C for
Exercises from the Online text

* Exercise 6.1.1: Translate the arithmetic expression $a + -(b + c)$ into:

a) A syntax tree.

b) Quadruples.

Exercise 6.1.2: Repeat Exercise 6.1.1 for the following assignment statements:

i. $a = b[i] + c[j]$.

ii. $a[i] = b*c - b*d$.

iii. $x = f(y+1) + 2$.

iv. $x = *p + &y$. 
Figure 6.34: Annotated parse tree for $x < 100 \lor x > 200 \land x \neq y$