Generating Code for Assignment Statements
— Part II —

Comp 412

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapters 4, 6 & 7 in EaC2e
Expanding the Expression Grammar

The Classic, Left-Recursive, Expression Grammar

<table>
<thead>
<tr>
<th></th>
<th>Rule</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Goal</strong> → <em>Expr</em></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><strong>Expr</strong> → <em>Expr</em> + <em>Term</em></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><em>Term</em> → <em>Term</em> - <em>Term</em></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><em>Term</em> → <em>Term</em></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><strong>Term</strong> → <em>Term</em> <em>Factor</em></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><em>Term</em> → <em>Term</em> / <em>Factor</em></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><em>Factor</em> → <em>Factor</em></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><em>Factor</em> → ( <em>Expr</em> )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><em>Factor</em> → <em>number</em></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><em>Factor</em> → <em>ident</em></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td><em>Factor</em> → <em>ident</em> [ <em>Exprs</em> ]</td>
<td>Array reference</td>
</tr>
<tr>
<td>12</td>
<td><em>Factor</em> → <em>ident</em> ( <em>Exprs</em> )</td>
<td>Function invocation</td>
</tr>
</tbody>
</table>

References to aggregates & functions

- First, the compiler must recognize expressions that refer to arrays, structures, and function calls
- Second, the compiler must have schemes to generate code to implement those abstractions
Array Layout

Row-Major Order
• Lay out as a sequence of consecutive rows
• Rightmost subscript varies fastest
• $A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]$ 

Storage Layout

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
<th>2,1</th>
<th>2,2</th>
<th>2,3</th>
<th>2,4</th>
</tr>
</thead>
</table>

Stride One Access

```c
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    A[i][j] = 0;
```

Declared arrays in C (and most languages)

Stride one access: successive references in the loop are to successive locations in the level one (or L1) data cache. Stride one access maximizes spatial reuse in all levels of cache, as well as the effectiveness of hardware prefetch units.

The Concept

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
<td></td>
</tr>
</tbody>
</table>

COMP 412, Fall 2017
Array Layout

Column-Major Order

• Lay out as a sequence of columns
• Leftmost subscript varies fastest
• A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

Storage Layout

| A | 1,1 | 2,1 | 1,2 | 2,2 | 1,3 | 2,3 | 1,4 | 2,4 |

Stride One Access

```
for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
        A[i][j] = 0;
```

All arrays in FORTRAN

COMP 412, Fall 2017
Array Layout

Indirection Vectors
- Vector of pointers to pointers to ... to values
- Takes much more space, trades indirection for arithmetic
- Not amenable to analysis

Storage Layout

Stride One Access
No reference pattern guarantees stride one access in cache, unless rows are contiguous

Arrays in Java
int **array; in C

COMP 412, Fall 2017
Computing an Address for an Array Element

A[ i ]

• \( \text{@A} + ( i - \text{low} ) \times \text{sizeof(elt)} \)

• In general: base(A) + ( i - low ) x sizeof(A[1])

What about \( A[i_1,i_2] \)?

Row-major order, two dimensions

\( \text{@A} + (( i_1 - \text{low}_{1}) \times (\text{high}_{2} - \text{low}_{2} + 1) + i_2 - \text{low}_{2}) \times \text{sizeof(elt)} \)

Address computation took 3 adds, 3 subtracts, 1 multiply and 1 shift

Cheap in comparison to the register-save costs of a function call.
Array Layout

Row-Major Order
• Lay out as a sequence of consecutive rows
• Rightmost subscript varies fastest
• $A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]$

Storage Layout

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
<th>2,1</th>
<th>2,2</th>
<th>2,3</th>
<th>2,4</th>
</tr>
</thead>
</table>

Address Polynomial for $A[i,j]$

@A + (( $i_1 - low_1$ ) x ( $high_2 - low_2 + 1$ ) + $i_2 - low_2$ ) x sizeof(elt)

Why does C start arrays indices at zero?
To avoid those subtractions

This polynomial follows Horner’s rule for evaluation.
Array Layout

Column-Major Order
• Lay out as a sequence of columns
• Leftmost subscript varies fastest
• A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

Storage Layout

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>2,1</th>
<th>1,2</th>
<th>2,2</th>
<th>1,3</th>
<th>2,3</th>
<th>1,4</th>
<th>2,4</th>
</tr>
</thead>
</table>

Address Polynomial for A[i,j]

@A + ((i₂ - low₂) x (high₁ - low₁ + 1) + i₁ - low₁) x sizeof(elt)

Change in layout order swaps the subscripts ...

This polynomial follows Horner’s rule for evaluation.
Array Layout

Indirection Vectors
• Vector of pointers to pointers to ... to values
• Takes much more space, trades indirection for arithmetic
• Not amenable to analysis

Storage Layout

Address Polynomial for A[i,j]
*(A[i_1])[i_2] — where each of the [i]’s is, itself, a 1-d array reference

Sketch of the code

Back when systems supported 1 or 2 cycle “indirect load” operations, this scheme was efficient. It replaces a multiply & an add with an indirect load.

COMP 412, Fall 2017
Array Address Calculations

In scientific codes, array address calculations are a major cost

• Each additional dimension adds more arithmetic

• Efficiency in address calculation is a critical issue for performance
  – A[i+1,j], A[i,j], A[i,j+1] should all have some common terms
  – Horner’s rule evaluation hides the common terms
    → Improving these calculations is a matter of algebra (including distributivity!)

• Improving the efficiency of array address calculations has been a major focus of code optimization (& hardware design) for the last 40 years
  – Generate “better” code
  – Transform it to fit the context
  – Design memory systems that support common array access patterns
    → Optimize hardware for stride one access
    → Include sophisticated prefetch engines that recognize access patterns
Generating Better Code for $A[i,j]$

In row-major order

@A + ((i – low_1) x (high_2 – low_2 + 1) + j – low_2) x w

Can be re-distributed to

@A + (i – low_1) x (high_2 – low_2 + 1) x w + (j – low_2) x w

Which can be re-distributed to

@A + i x (high_2 – low_2 + 1) x w + j x w

– (low_1 x (high_2 – low_2 + 1) x w) - (low_2 x w)

If low_i, high_i, and w are known, the last term is a constant

Define $@A_0$ as

$@A – (low_1 x (high_2 – low_2 + 1) x w - low_2 x w)$

And $len_2$ as $(high_2 – low_2 + 1)$

Then, the address expression becomes

$@A_0 + (i x len_2 + j) x w$

If $@A$ is known, $@A_0$ is a known constant.

Just a couple of operations: 2 adds, 1 multiply, and 1 shift

Why does C start arrays indices at zero?

COLOR CODE:
- Invariant
- Varying

COMPUTE, associative, & distributive rules.
How do we know what to do?

**Accumulated wisdom**

Certain simple optimizations are performed as the **quadruples** are constructed. For example, an integer multiplication by a constant power of two (e.g., \(I \times 4\) or \(I \times 16\)) is replaced by a left-shift operation. An exponentiation involving an integer constant power (e.g., \(A^{**9}\)) is replaced by a series of in-line multiplications. Some operations involving minus signs are converted to simpler forms; for instance, \(-(B-C)\) becomes \(C-B\).  

*Finally, constants employed in a subscript expression [for example, each number 7 in \(A(7,1-7,1+7)\)] are often extracted from the subscript, evaluated as constant offsets from the start of the subscribed array, and combined into an aggregate constant offset which does not require computation during execution.*  

(p. 662; description of the “old” compiler, e.g., Medlock and Lowry)

The standard FORTRAN compiler evaluates subscripted array references in six steps. *First, numerical constants embedded in the subscripts are extracted, evaluated, and combined into an aggregate constant subscript when the program is translated into quadruples. Second, the subscript expression remaining in each dimension is evaluated and converted to integer. Third, each of these evaluated subscripts is multiplied by the span in bytes represented by a unit subscript in the subscripted dimension. Fourth, these products are added together to produce an aggregate computed subscript. Fifth, the constants extracted from the subscript and combined to form the aggregate constant subscript are added to the aggregate computed subscript to produce the aggregate effective subscript.* For an aggregate constant subscript in the range 0-4095, this addition is accomplished implicitly by encoding the constant in the displacement field of an indexed machine instruction.  

*Sixth, the address of the array itself is added to the aggregate effective subscript to produce the address of the subscripted array element.* This addition is always accomplished implicitly by using the base and index registers of an indexed machine instruction.  

(p. 665; describing the new compiler)

How do we know what to do?

Accumulated wisdom

Certain simple optimizations are performed as the quadruples are constructed. For example, an integer multiplication by a constant power of two (e.g., I*4 or I*16) is replaced by a left-shift operation. An exponentiation involving an integer constant power (e.g., A**9) is replaced by a series of in-line multiplications. Some operations involving minus signs are converted to simpler forms; for instance, \(-B + C\) becomes \(C - B\).

Finally, constants employed in a subscript expression [for example, each \([7,1+7]\)] are often extracted from the subscript, evaluated as constant offsets for the subscripts of the subscripted array, and combined into an aggregate constant offset during execution. (p. 662; description of the "old" compiler, e.g., Medlock and Lowry)

The standard FORTRAN compiler evaluates subscripted array references in six steps. First, numerical constants embedded in the subscript when the program is translated into quadruples are combined into an aggregate constant subscript when the program is translated into quadruples. Second, the subscript expression remaining in each dimension is evaluated and converted to integer. Third, each of these evaluated subscripts is multiplied by the span in bytes represented by a unit subscript in the subscripted dimension. Fourth, these products are added together to produce an aggregate computed subscript. Fifth, the constants extracted from the subscript and combined to form the aggregate constant subscript are added to the aggregate computed subscript to produce the aggregate effective subscript. For an aggregate constant subscript in the range 0-4095, this addition is accomplished implicitly by encoding the constant in the displacement field of an indexed machine instruction. Sixth, the address of the array itself is added to the aggregate effective subscript to produce the address of the subscripted array element. This addition is always accomplished implicitly by using the base and index registers of an indexed machine instruction. (p. 665, describing the new compiler)

Array References

What about variable-sized arrays? (*dynamically-sized arrays*)

- The compiler cannot know, at runtime, the dimension information
  - Must build a runtime descriptor, called a **dope vector**
  - Create the dope vector when the array is allocated
  - Addressing code references the values in the dope vector

- Compiler can store parameters for naïve polynomial, or for the optimized polynomial
  - Must decide, up front, the code shape issue
Array References

What about variable-sized arrays? (dynamically-sized arrays)

- The compiler cannot know, at runtime, the dimension information
  - Must build a runtime descriptor, called a dope vector
  - Create the dope vector when the array is allocated
  - Addressing code references the values in the dope vector

- Compiler can store parameters for naïve polynomial, or for the optimized polynomial
  - Must decide, up front, the code shape issue

- A variable-sized array may cost more to access than a fixed-size array
  - Different costs for textually similar references

- The dope vector presents opportunity for optimizer
  - Common subexpressions in address polynomials
  - Contents of dope vector are fixed between allocations
    - In many situations, they are provably fixed across the entire invocation
  - Good optimizer can recover much of the lost ground

Violates the principle of least astonishment

Array References

Array-valued parameters pose a similar problem to variable-sized arrays

• Called procedure can receive different size arrays from different calls
• Called procedure must work with any array it is passed

The Solution: Pass the array as a dope vector

• Generate references to elements of array valued parameters as if they were references to variable-sized arrays
  – Array-valued parameters have a different cost than a local or global array
• Build dope vector in caller
• Pass pointer to dope vector as parameter

int A[0:100,0:100];
int B[2:20,0:10000];
... call fee(A);
... call fee(B);
Structures & Records

Structures and records present two complications

Each declared structure has a set of fields

• Each has a size and an offset
• To access a field, need to compute base + offset for the field
• Use the size to choose load width & register width

Structures and records can have dimensions

• Arrays of structures
• Fields in a structure that are arrays or arrays of structures
• Need to intermix array address calculations, as necessary

        → Remember all of those sizeof(elt) terms in the array address polynomials?

Structures and records require compile-time support in the form of a table that maps field names to <offset, size> tuples.

From one jaded view, objects are just instances of a structure implicitly declared as a “class”.

Data-area Layout?
Representing & Manipulating Strings

Strings are fundamentally different from scalars, arrays & records

• Fundamental unit is a character
  – Typical character sizes are one or two bytes
  – Target ISA may (or may not) support character-size operations

• Set of PL supported operations on character & string data is limited
  – Assignment, length, concatenation, & (sometimes) translation

• Efficient string operations are hard to implement from basic operations
  – Particularly difficult on RISC processors
  – Implications for the IR, the procedure linkage convention, & source language design

The IBM 370 had a machine instruction that did a byte-by-byte translation of a string through a 256-byte table, at high speed.

See § 7.6 in EaC2e
Representing & Manipulating Strings

Two common representations

• Explicit length field

```
8 a s t r i n g
```

Length field may take more space than a terminator

• Null termination

```
a s t r i n g
```

• This issue is a language design issue
  – Are strings fixed length, or varying length?

String representation is a great case study in the way that one design decision (C, Unix) can have a long term impact on computing (security, ISA, buffer overflow). See the article on the COMP 412 Lectures page, “The most expensive one-byte mistake.”
Each representation has advantages and disadvantages

<table>
<thead>
<tr>
<th>Operation</th>
<th>Explicit Length</th>
<th>Null Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>Straightforward</td>
<td>Straightforward</td>
</tr>
<tr>
<td>Checked Assignment</td>
<td>Checking is $O(1)$</td>
<td>Must count length$^1$</td>
</tr>
<tr>
<td>Length</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Concatenation</td>
<td>Must copy data</td>
<td>Length + copy data</td>
</tr>
</tbody>
</table>

Unfortunately, null termination is almost considered normal

- Hangover from the design of C
- Embedded in various OS and API designs

$^1$Checked assignment requires both a current length for the string and an allocated length for the buffer.
Encoding String Operations

**Single character assignment**

- With character operations
  - Compute address of rhs, load character
  - Compute address of lhs, store character

- With only word operations \((> 1 \text{ char per word})\)
  - Compute address of word containing rhs & load it
  - Shift character to destination position within word
  - Compute address of word containing lhs & load it
  - Mask out \textbf{and} current character & mask in \textbf{or} new character
  - Store lhs word back into place
  - \textit{This style of manipulation gets messy and complicated very quickly}
Encoding String Operations

Multiple character assignment

Two strategies:
1. Wrap a loop around the single character code, or
2. Work up to a word-aligned case, repeat whole word moves, and handle any partial-word end case

(1) Character operations
1. Easy to generate; inefficient use of resources
2. Harder to generate; better use of resources

(2) Only word operations
1. Lots of complication to generate
2. Fold complications into end case; reasonable efficiency

Source & destination aligned differently
⇒ much harder cases for word operations
Encoding String Operations

**Concatenation**

- String concatenation touches every character
  - Need to copy each character to the result string
    → Unless the operation is destructive (b ← b || c)
  - Often, need to compute size of result to allocate space for it

- Exposes representation issues
  - Is string a descriptor that points to text?
  - Is string a buffer that holds the text?
  - Consider a ← b || c
    → Compute b || c and assign descriptor to a?
    → Compute b || c into a temporary & copy it into a?
    → Compute b || c directly into a?

- What about a call to fee( b || c )?
  - Where does b || c go?

Several of you remarked that excessive string concatenation was a source of slowdown in your local allocators.
That is a common problem.
Program with abstractions. Measure performance. Replace the abstractions that cause performance problems.

COMP 412, Fall 2017
Encoding String Operations

Length Computation

• Representation determines cost
  – Explicit length turns \textit{length}(b) into a memory reference
  – Null termination turns \textit{length}(b) into a loop of memory references and arithmetic operations or, as in C, a procedure call to such a loop

• Length computation arises in other contexts
  – Whole-string or substring assignment
  – Checked assignment \textit{(buffer overflow prevention)}
  – Concatenation
  – Evaluating call-by-value actual parameter or concatenation as an actual parameter

And a function call is not cheap ...
Encoding String Operations

As a matter of safety, string operations should be checked

• String assignments and references are potentially dangerous
  – Reading beyond the end of a string can provide access to critical data
  – For example, the return address in the current AR is at a (negative?) offset from a string in the local data area.
  – Buffer overflow attack might write code, then invoke it with the return

• Checking assignments, references, & concatenations is tedious
  – Requires knowledge of the string’s allocated size
  – Adds conditional logic plus potential of an exception

• Checking assignments makes string operations even slower
  – But, compilers and runtimes need to do it

Eliminating such checks is a potential source for significant improvement

All the same statements hold true for array & structure operations

1 “Exception” might be orderly, as in Java, or disorderly, as in C (“segmentation fault”)