Generating Code for Assignment Statements

— back to work —

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
### Example — Building an Abstract Syntax Tree

<table>
<thead>
<tr>
<th>Line</th>
<th>Grammar Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal → Expr</td>
<td>$$ = $1;$$</td>
</tr>
<tr>
<td>2</td>
<td>Expr → Expr + Term</td>
<td>$$ = \text{MakeAddNode}(1,3);$$</td>
</tr>
<tr>
<td>3</td>
<td>Expr - Term</td>
<td>$$ = \text{MakeSubNode}(1,3);$$</td>
</tr>
<tr>
<td>4</td>
<td>Term</td>
<td>$$ = $1;$$</td>
</tr>
<tr>
<td>5</td>
<td>Term → Term * Factor</td>
<td>$$ = \text{MakeMulNode}(1,3);$$</td>
</tr>
<tr>
<td>6</td>
<td>Term / Factor</td>
<td>$$ = \text{MakeDivNode}(1,3);$$</td>
</tr>
<tr>
<td>7</td>
<td>Factor</td>
<td>$$ = $1;$$</td>
</tr>
<tr>
<td>8</td>
<td>Factor → (Expr)</td>
<td>$$ = $2;$$</td>
</tr>
<tr>
<td>9</td>
<td>number</td>
<td>$$ = \text{MakeNumNode}(\text{token});$$</td>
</tr>
<tr>
<td>10</td>
<td>ident</td>
<td>$$ = \text{MakeIdNode}(\text{token});$$</td>
</tr>
</tbody>
</table>

### Assumptions:
- constructors for each node
- stack holds pointers to nodes

We call $$ = $1;$$ a copy rule.
## Example — Building an Abstract Syntax Tree

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id – num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$ id</td>
<td>– num * id</td>
<td>reduce 10</td>
</tr>
<tr>
<td>$ Factor</td>
<td>– num * id</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$ Term</td>
<td>– num * id</td>
<td>reduce 4</td>
</tr>
<tr>
<td>$ Expr</td>
<td>– num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr –</td>
<td>num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr – num</td>
<td>* id</td>
<td>reduce 9</td>
</tr>
<tr>
<td>$ Expr – Factor</td>
<td>* id</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$ Expr – Term</td>
<td>* id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr – Term *</td>
<td>id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr – Term * id</td>
<td></td>
<td>reduce 10</td>
</tr>
<tr>
<td>$ Expr – Term * Factor</td>
<td></td>
<td>reduce 5</td>
</tr>
<tr>
<td>$ Expr – Term</td>
<td></td>
<td>reduce 3</td>
</tr>
<tr>
<td>$ Expr</td>
<td></td>
<td>accept</td>
</tr>
</tbody>
</table>

**AHSDT Works!**

- Built the AST
- Some reduce actions just copied values; others called constructors.
- Same tree as earlier slide

**Example**

```
Expression: $ id x $ id y

AST:
```

- `-`
  - `*`
    - `id x`
    - `num 2`
    - `id y`

COMP 412, Fall 2017
Example — Emitting ILOC

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal → Expr</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Expr → Expr + Term</td>
<td>$$ = NextRegister(); Emit(add, $1, $3, $$);</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>$$ = NextRegister(); Emit(sub, $1, $3, $$);</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>$$ = $1;</td>
</tr>
<tr>
<td>5</td>
<td>Term → Term * Factor</td>
<td>$$ = NextRegister(); Emit(mult, $1, $3, $$);</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>$$ = NextRegister(); Emit(div, $1, $3, $$);</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>$$ = $1;</td>
</tr>
<tr>
<td>8</td>
<td>Factor → ( Expr )</td>
<td>$$ = $2;</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>$$ = NextRegister(); Emit(loadI,Value(lexeme),$$);</td>
</tr>
<tr>
<td>10</td>
<td>ident</td>
<td>$$ = NextRegister(); EmitLoad(ident,$$);</td>
</tr>
</tbody>
</table>

Assumptions
- NextRegister() returns a virtual register name
- Emit() can format assembly code
- EmitLoad() handles addressability & gets a value into a register

Copy rules on the same productions as in the last example
### Example — Emitting ILOC

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id – num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$ id</td>
<td>– num * id</td>
<td>reduce 10</td>
</tr>
<tr>
<td>$ Factor</td>
<td>– num * id</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$ Term</td>
<td>– num * id</td>
<td>reduce 4</td>
</tr>
<tr>
<td>$ Expr</td>
<td>– num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr –</td>
<td>num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr –num</td>
<td>* id</td>
<td>reduce 9</td>
</tr>
<tr>
<td>$ Expr –Factor</td>
<td>* id</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$ Expr –Term</td>
<td>* id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr –Term*</td>
<td>id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr –Term * id</td>
<td></td>
<td>reduce 10</td>
</tr>
<tr>
<td>$ Expr –Term * Factor</td>
<td></td>
<td>reduce 5</td>
</tr>
<tr>
<td>$ Expr –Term</td>
<td></td>
<td>reduce 3</td>
</tr>
<tr>
<td>$ Expr</td>
<td></td>
<td>accept</td>
</tr>
</tbody>
</table>

**Emitting ILOC**

- Simple, but clean, code
- *EmitLoad()* hides all of the messy details of naming, addressability, and address modes

**ILOC Examples**

- `loadAl pr0, @x` ⇒ `vr1`
- `loadl 2` ⇒ `vr2`
- `loadAl pr0, @y` ⇒ `vr3`
- `mult vr2, vr3` ⇒ `vr4`
- `sub vr1, vr4` ⇒ `vr5`
What’s Left in Expressions and Assignments?

Assume that we all understand symbol tables & storage mapping

• We hid a lot of detail in “EmitLoad()”
  – How does all of that work?
• What about type information?
  – Every variable & value has a type
  – Operators have a range of types over which they are defined
• What about more operators? Assignment?
• Arrays, strings, structures

If the language does not have a declarations before executables rule, then the compiler must either: (1) derive its knowledge of variables, types, and locations incrementally; or (2) make two passes over the code (e.g., build an AST and re-traverse it).

To accomplish (1), the compiler use a higher level of abstraction in the initial IR to avoid the need for specific addresses (e.g., represent variables with names and expand address computations at a later stage in compilation.)
Handling Identifiers

Identifiers have some subtlety, which we hid in \textit{EmitLoad()}.

The \textbf{compiler} needs to decide which values \textit{can legally} live in registers:

- A value is ambiguous if the compiler can access it via more than one name:
  - Some pointer based values
  - Some call-by-reference parameters
  - Most compilers treat array elements \textit{(e.g., $A[i,j]$)} as ambiguous

- \textbf{Ambiguous} values must live in memory \textit{(load or store at each access)}

- Any \textbf{unambiguous value} is a candidate for a register

The compiler needs to assign offsets to each variable kept in \textbf{RAM}:

- Order them by alignment constraints and assign most constrained objects first \textit{(e.g., double-word, then word, then half-word, then byte, ...)}
  - Reduce padding needed to maintain alignment

\textit{See pages 341-342 in EaC2e for discussion of ambiguity.}
Handling Identifiers

Identifiers have some subtlety, which we hid in EmitLoad()

For a scalar identifier \( x \) at some point \( p \):

- The compiler needs to know
  - Which declaration of \( x \) pertains at \( p \)?
  - And what properties does it declare?

- The various symbol tables resolve this issue

- After storage assignment \( x \) has either
  - A virtual register number, or
  - A \texttt{base + offset} pair to compute an address

\textit{Emitload()} can either return the VR or emit code to compute the address into a register, perform the load, and return the resulting VR

- For aggregates, the address computation is more complex, as we shall see

```c
static int x;

int foo( int y ) {
    int x;
    x = y * 9;
    if(x > 1017)
        x = foo(x) + 17;
    return x;
}

x = foo(12)
```
Extending the Grammar

Adding other operators

• +, -, ×, and ÷ are not enough
• Need to add operators to grammar and generate code for them

Defining the syntax

• Additional productions
• Additional levels of precedence

Code generation follows the scheme for existing operators

• Evaluate the operands, then perform the operation
• Complex operations (sin, log₁₀, 2ˣ) might turn into library calls
• Handle assignment as an operator

See, for example, Fig. 7.7 on p. 351 of EaC2e.
For example, booleans, relationals, and unary operators

\[
\begin{align*}
\text{Boolean} & \rightarrow \text{Boolean} \lor \text{AndTerm} & \text{Expr} & \rightarrow \text{Expr} + \text{Term} \\
& | \quad \text{AndTerm} & & | \quad \text{Expr} - \text{Term} \\
\text{AndTerm} & \rightarrow \text{AndTerm} \land \text{RelExpr} & & | \quad \text{Term} \\
& | \quad \text{RelExpr} & & | \quad \text{Term} \times \text{Value} \\
\text{RelExpr} & \rightarrow \text{RelExpr} < \text{Expr} & & | \quad \text{Term} \div \text{Value} \\
& | \quad \text{RelExpr} \leq \text{Expr} & & | \quad \text{Value} \\
& | \quad \text{RelExpr} = \text{Expr} & & | \quad ! \text{Factor} \\
& | \quad \text{RelExpr} \neq \text{Expr} & & | \quad \text{Factor} \\
& | \quad \text{RelExpr} \geq \text{Expr} & & | \quad ( \text{Expr} ) \\
& | \quad \text{RelExpr} > \text{Expr} & & | \quad \text{number} \\
& | \quad \text{Expr} & & | \quad \text{Reference}
\end{align*}
\]

... where Reference derives a name, a subscripted name, a structure reference, a string reference, a function call, ...
Mixed-Type Expressions

What if the operands to an operation have different types?

• **Key observation:** the language must define the behavior

• Might simply be an error, detected during semantic elaboration
  – Require the programmer to “fix” it

• Might require the compiler to insert an implicit conversion

### Implicit conversions

• Compiler uses a conversion table to determine type of result
  – Convert arguments to result type & perform operation

• Most languages have symmetric & rational conversion tables

#### Typical Conversion Table for Addition

- **(note the symmetry)**

<table>
<thead>
<tr>
<th></th>
<th>Integer</th>
<th>Real</th>
<th>Double</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>Real</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td>Real</td>
<td>Real</td>
<td>Real</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
</tr>
</tbody>
</table>
Assignment as an Operator

\[ lhs \leftarrow rhs \]

**Strategy**

- Evaluate \( rhs \) to a **value** *(an rvalue)*
- Evaluate \( lhs \) to a **location** *(an lvalue)*
  - \( lvalue \) is a register \( \Rightarrow \) move \( rhs \) into the register
  - \( lvalue \) is an address \( \Rightarrow \) store \( rhs \) into the memory location
- If \( rvalue \) & \( lvalue \) have different types
  - Evaluate \( rvalue \) to its “natural” type
  - Convert that value to the type of \( *lvalue \)

Unambiguous scalars go into registers

Ambiguous scalars or aggregates go into memory

Keeping ambiguous values in memory lets the hardware sort out the addresses.
Handling Assignment

What if the compiler cannot determine the type of the rhs?

• Issue is a property of the language & the specific program
• For type-safety, compiler must insert a run-time check
  – Some languages & implementations ignore safety (a truly bad idea)
• Add a tag field to the data items to hold type information
  – Explicitly check tags at runtime

Code for assignment becomes more complex

```
evaluate rhs
if type(lhs) ≠ rhs.tag
then
  convert rhs to type(lhs) or
  throw an exception
lhs ← rhs
```

Choice between conversion & a runtime exception depends on details of language & type system

Much more complex than static checking, plus costs occur at runtime rather than compile time
Handling Assignment

**Compile-time type-checking**
- Goal is to eliminate the need for both tags & runtime checks
- Determine, at compile time, the type of each subexpression
- Use runtime check only if compiler cannot determine types

**Optimization strategy**
- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation

Can *design* the language so all checks are static
Extending the Schemes

More complex cases for IDENTIFIER

• What about references to aggregate values? (*array or structure elements*)
  – Need a layout for aggregate
  – Need a formula to find the specified element
  – Will (often) generate runtime arithmetic to compute element address

• What about function calls in expressions? *(future lecture)*
  – Generate the calling sequence & load the return value
  – Severely limits compiler’s ability to reorder operations

• What about a reference to a value passed in as a parameter?
  – Many linkages pass the first several values in registers
  – Call-by-value ⇒ just a local variable with a funny offset
  – Call-by-reference ⇒ funny offset, extra indirection

Parameters are typically stored at a negative offset from the local data area. Code will have a pointer to that data area.
Expanding the Expression Grammar

The Classic, Left-Recursive, Expression Grammar

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>( \rightarrow ) Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr</td>
<td>( \rightarrow ) Expr + Term</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>( \mid ) Expr - Term</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>( \mid ) Term</td>
</tr>
<tr>
<td>5</td>
<td>Term</td>
<td>( \rightarrow ) Term * Factor</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>( \mid ) Term / Factor</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>( \mid ) Factor</td>
</tr>
<tr>
<td>8</td>
<td>Factor</td>
<td>( \rightarrow ) ( Expr )</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>( \mid ) number</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>( \mid ) ident</td>
</tr>
</tbody>
</table>

The grammar is great for \( x - 2 * y \)
- Not all programs operate over the domain of \text{ident} and \text{number}
- More complex programs require more complex expressions

No mention of array references, structure references, or function calls
Expanding the Expression Grammar

The Classic, Left-Recursive, Expression Grammar

1. **Goal** → **Expr**
2. **Expr** → **Expr** + **Term**
3. | **Expr** - **Term**
4. | **Term**
5. **Term** → **Term** * **Factor**
6. | **Term** / **Factor**
7. | **Factor**
8. **Factor** → ( **Expr** )
9. | **number**
10. | **ident**
11. | **ident** [ **Exprs** ]
12. | **ident** ( **Exprs** )

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 reference

Function invocation

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
Scheme to Generate Code For A[i,j]?

**Compiler must generate the runtime address of the element A[i,j]**

- The compiler needs to know where A begins
  - Tie between compile-time knowledge and runtime behavior
  - We will defer this discussion until we discuss procedure calls ...
    → **assume that @A is the address of A’s first element**

- The compiler must have a plan for the internal layout of A
  - Programming language usually dictates array element layout
  - Three common choices
    → Row-major order
    → Column-major order
    → Indirection vectors

- And a formula for calculating the address of an array element
  → **General scheme: compute address, then issue load (rvalue) or store (lvalue)**
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

A

1,1 1,2 1,3 1,4 2,1 2,2 2,3 2,4

Stride One Access
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 & the effectiveness of hardware prefetch units.
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>A</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>

Stride One Access

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

All arrays in FORTRAN

The Concept

<table>
<thead>
<tr>
<th>A</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

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

The General Scheme for an Array Reference

• Compute an address
• Issue a load

A[i]

• \@$A + (i - \text{low}) \times \text{sizeof}(A[1])$
• In general: base(A) + (i - low) \times \text{sizeof}(A[1])

\@$A$ is the base address of A.
Depending on how A is declared, \@$A$ may be
• an offset from the pointer to the procedure’s data area (its ARP),
• an offset from some global label, or
• an arbitrary address.
The first two are compile time constants.

\text{ARP} \equiv \text{Activation Record Pointer} \quad (\text{see next lecture})
Computing an Address for an Array Element

A[ i ]

- \@A + ( i – low ) \times \text{sizeof}(A[1])
- In general: base(A) + ( i – low ) \times \text{sizeof}(A[1])

Almost always a power of 2, known at compile-time
⇒ use a shift for speed

int A[1:10] ⇒ low is 1
Make low be 0 for faster access (saves a – )

Note, for the record, that a naïve vector reference turns into a subtract, a shift, and an address-offset load (e.g., loadAO) — three operations before optimization. This sequence is amazingly cheap.
Computing an Address for an Array Element

\[ A[i] \]

- \( @A + (i - \text{low}) \times \text{sizeof(elt)} \)
- In general: \( \text{base}(A) + (i - \text{low}) \times \text{sizeof}(A[1]) \)

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

*Row-major order, two dimensions*

\[ @A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof(elt)} \]

<table>
<thead>
<tr>
<th>( \text{low}_1 )</th>
<th>Lower bound on row index</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{high}_1 )</td>
<td>Upper bound on row index</td>
<td>5</td>
</tr>
<tr>
<td>( \text{low}_2 )</td>
<td>Lower bound on column index</td>
<td>1</td>
</tr>
<tr>
<td>( \text{high}_2 )</td>
<td>Upper bound on column index</td>
<td>4</td>
</tr>
</tbody>
</table>

\[ \begin{array}{cccc}
1,1 & 1,2 & 1,3 & 1,4 \\
2,1 & 2,2 & 2,3 & 2,4 \\
3,1 & 3,2 & 3,3 & 3,4 \\
4,1 & 4,2 & 4,3 & 4,4 \\
5,1 & 5,2 & 5,3 & 5,4 \\
\end{array} \]

(i-\( \text{low}_1 \)) is 3
(high\(_2 - \text{low}_2 + 1 \)) is 4
3 \times 4 is 12
i\(_2 - \text{low}_2 \) is 2

Combined, the 2 terms take us to the start of \( A[4,3] \)

Address computation took 3 adds, 3 subtracts, 1 multiply and 1 shift
Cheap in comparison to the register-save costs of a function call.

Assume size\( \text{of}(A[1]) \) is 1, for simplicity.
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>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address Polynomial for A[i,j]

\[@A + \left( (i_1 - low_1) \times (high_2 - low_2 + 1) + i_2 - low_2 \right) \times \text{sizeof(elt)} \]

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>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address Polynomial for A[i,j]

\[@A + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times \text{sizeof(elt)}\]

This polynomial follows **Horner’s rule** for evaluation.

Change in layout order swaps the subscripts ...
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

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