Compiler Lecture Notes

Chapter 18: Intermediate Code Generation (Expressions)

April 12, 1994

Preliminary version: not for distribution outside of NYU

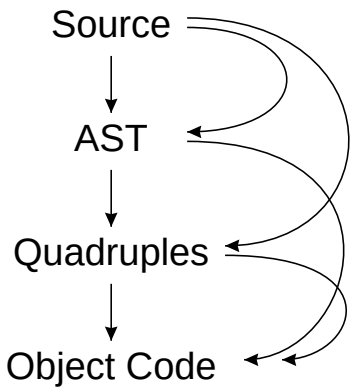
Outline

A. Intermediate Code Generation
   1. How ICG relates to the rest of the compiler
   2. Intermediate Representations
   3. A univeral intermediate representation
B. Three-address code
   1. Quadruples
   2. Quadruple Opcodes (a list)
C. Quadruple code generation from ASTs
   1. The AST-to-Quad Code Generator
   2. GenExpr
   3. GenExpr (implementation)
D. Generating code for binary/unary expressions
E. Generating quads for arrays
   1. Arrays with a lower bound of zero
   2. Arrays with non-zero lower bounds
   3. Virtual origins
   4. A simple indexing optimization
F. Multi-dimensional arrays
   1. Row major form
   2. The address computation
G. Summary

Intermediate Code Generation

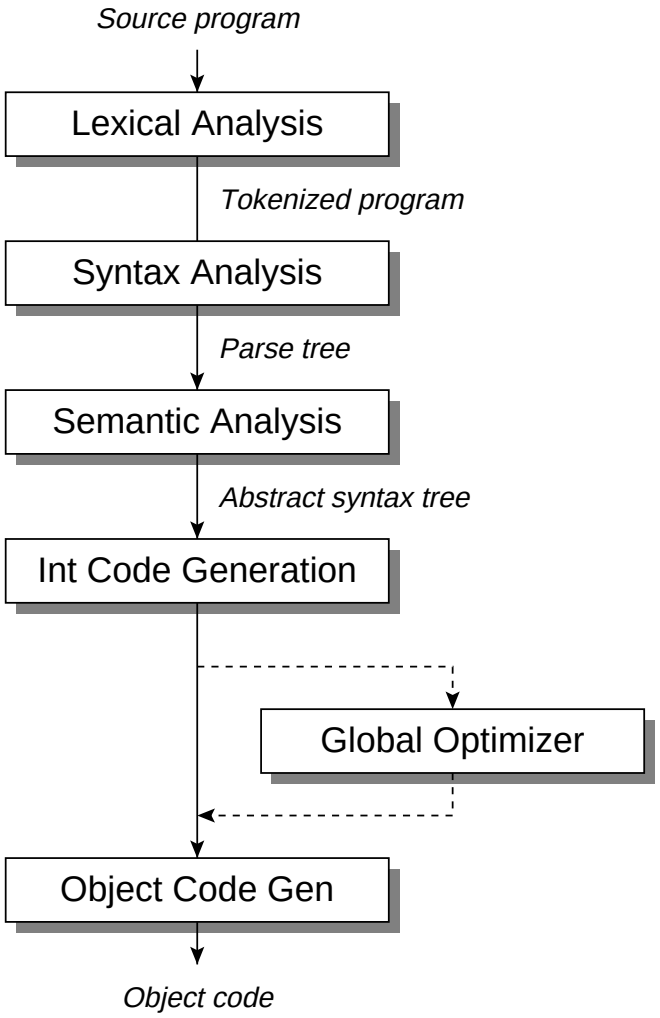
Intermediate code generation is another step in the translation of a source program into an object program.

- Intermediate code generation can occur:
  - From a previously constructed IR such as an abstract syntax tree.
  - During top-down or bottom-up parsing, without some previous IR.
- The two intermediate representations that we will use are abstract syntax trees (ASTs) and quadruples (quads).



How ICG relates to the rest of the compiler

The logical relation between other phases of the compiler and the intermediate code generator are shown below.



Intermediate Representations

An intermediate representation (called IR for short) is a data structure used to represent executable statements in a compiler. Most commonly used IRs fall into one of three categories:

- *Tree-based intermediate IRs*. Abstract syntax trees are an example of a tree-based IR.
- *Linearized IRs*. *Three-address code* is a family of IRs that are characterized as having an operand and a maximum of three addresses. There are several variants of three-address code; the one we will use is called quadruples.
  - Quadruples break down large expressions in single-operator expressions through the use of compiler generated variables called *temporaries*.
- *Stack-based IRs*. Use of stack-based IRs are usually restricted to special languages such as SNOBOL or PostScript that are inherently stack-based.

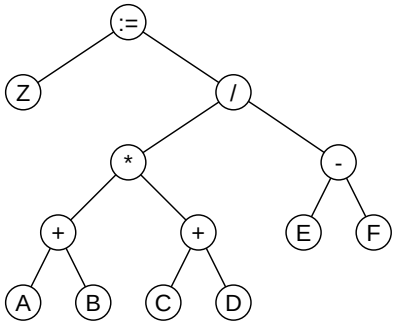
Example

Consider the following statement:

```
z := (a + b) * (c + d) / (e - f)
```

and the way in which it might be represented using the three different forms of IR.

Abstract syntax tree



Quadruples

```
t1 := a + b
t2 := c + d
t3 := t1 * t2
t4 := e - f
t5 := t3 / t4
z  := t5
```

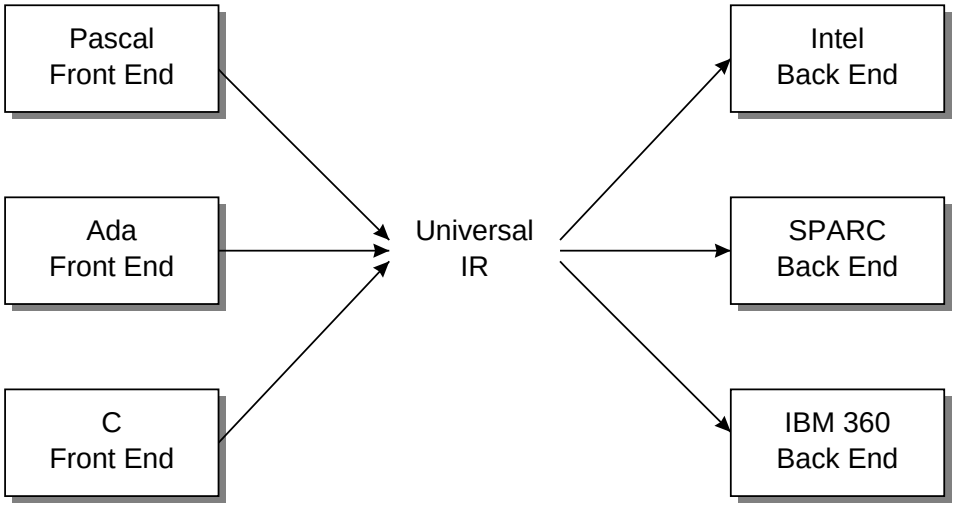
Stack IR

```
a b + c d + * e f - / z :=
```

A univeral intermediate representation

As early as the 1950s, the concept of a *universal intermediate representation* was proposed (the proposal first appeared in CACM, as the IR was called UNCOL).

- The primary idea behind a universal IR is to reduce the effort in building multiple compilers for multiple source languages and multiple target architectures.
- If m is the number of languages you are writing a compiler for, and n is the number of target architectures, then the use of an intermediate language reduces the number of compilers you are writing from m * n compilers to m front ends + n back ends.



Three-address code

Three-address code is an IR in which an IR opcode is followed by several operands, typically 0 to 3.

- It is called three-address code because the operands are usually addresses (either symbol table addresses or references back into the three-address code).
- Three address code resembles assembly language, but there are important differences:
  - Although the operators closely correspond to assembly language operators, there is not an exact one-to-one correspondance.
  - Registers are not part of three-address code.
- Types of three-address code:
  - Quadruples. Quadruples use explicit temporaries to represent subexpressions within an expression.
  - Direct triples. No tempoaries are used; instead, subexpressions are specified as references to numbered triples.
  - Indirect triples. One level of indirection is added to the triple references.

Quadruples

Quadruples are probably the most common form of three-address code. They take the following general form:

```
arg1 := arg2 opcode arg3
```

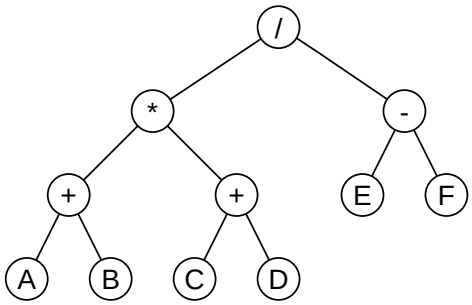
- They are called quadruples because the most general form that they take involves four objects: an operator, and up to three operands:
- The advantages of quadruples are:
  - Access to all arguments is fast because most arguments are just pointers to either the attribute table or to the data structure used to hold the quadruples themselves.
- The disadvantages of quadruples are:
  - The number of temporaries in the symbol table can become quite large.

Example

Consider the expression

```
(a + b) * (c + d) / (e - f)
```

and how it might be represented using different forms of three-address code:



Quadruples

```
t1 := a + b
t2 := c + d
t3 := t1 * t2
t4 := e - f
t5 := t3 / t4
```

Direct Triples

```
(1)  a + b
(2)  c + d
(3)  (1) + (2)
(4)  e - f
(5)  (3) / (4)
```

Indirect Triples

```
(1) 21   (21) a + b
(2) 22   (22) c + d
(3) 23   (23) (21)+(22)
(4) 24   (24) e + f
(5) 25   (25) (23)+(24)
```

Quadruple Opcodes (a list)

Here is a more or less complete listing of the quadruple operators necessary for a simple language such as Pascal.

Quad Type	Standard listing	QuadType
binary assignment	lhs := op1 bop op2	`QuadAdd, QuadSub, QuadMul, etc.`
unary assignment	lhs := uop op1	`QuadUnaryMinus, QuadUnaryPlus`
copy	lhs := op1	`QuadAssign`
array access	lhs := array[index]	`QuadOf`
array assignment	array[index] := value	`QuadSOf`
unconditional jump	goto x	`QuadGoto`
conditional jump	if x > y goto label	`QuadIfEq, QuadIfNE, etc.`
procedure call	call p	`QuadCall`
parameter	param x	`QuadParam`
proc return with value	return x	`QuadReturn`
label x	x:	`QuadLabel`

For the implementation, we use a type called QuadType:

```
type QuadType = (
  QuadAdd, QuadSub, QuadMul, QuadDiv,
  QuadUnaryMinus, QuadUnaryPlus,
  QuadCopy, QuadOf, QuadSOf, QuadGoto,
  QuadIfEq, QuadIfNE, QuadIfLT, QuadIfLE,
```

```
QuadIfGT, QuadIfGE,
QuadCall, QuadParam, QuadReturn, QuadLabel );
```

Example

The Pascal statement

```
for i := 1 to 10 do
  a[i] := i * 5;
```

would translated into the following quadruples:

```
     i := 1
L1:
     if i > 10 goto L2
     t1 := i * 5          ; compute expr
     t2 := t1 * 2         ; scale the index
     a[i] := t2
     i := i + 1
     goto L1
L2:
```

Quadruple code generation from ASTs

We will take a very simple approach in generating quadruples from an AST.

- We do a one-pass traversal of the AST, guided at each point by the kind of node it is (this is given by the AstClass field in the AST node).
- In many places, there are special optimizations that can be done to generate better quadruples (which in turn result in better object code). For example:
  - Optimizations for array references.
  - Optimizations for for loops.
  - Optimization of boolean expressions.

The AST-to-Quad Code Generator

- We will describe a one-pass algorithm in which quadruples are generated from an AST during a one-pass traversal.
- The code generator is divided into three mutually recursive routines:
  - GenStmt is responsible for generating code for statement nodes.
  - GenExpr is responsible for generating code for arithmetic expressions.
  - GenBool is responsible for generating code for boolean expressions (typically used to generate code for boolean expressions controlling IF statements and WHILE statements).
- Three other auxiliary routines will be used.
  - GenLabel creates a new label each time one is needed.
  - GenTmp creates a new temporary each time one is needed.
  - Emit actually creates a quadruple in the data structure or file being used to hold the quadruples.

GenExpr

The routine GenExpr generates code for arithmetic expressions by traversing an expression tree.

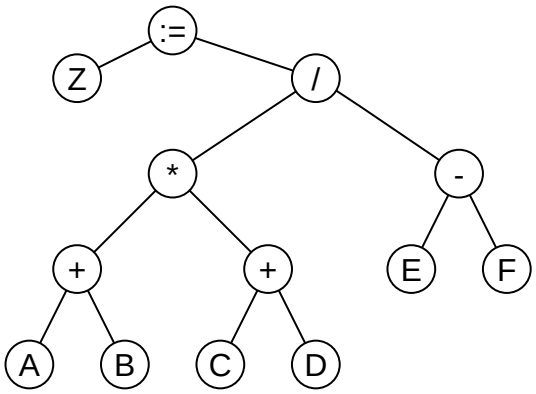
- GenExpr traverses an expression tree in strict left-to-right order generating a quadruple each time it leaves the node for the last time.
- GenExpr takes as an argument a point to the root node of an expression or subexpression in the AST.
- GenExpr returns a pointer to the attribute table entry corresponding to the expression or subexpression.
  - If the expression is a simple variable or constant GenExpr returns a pointer to the symbol table entry for that variable or constant.
  - If the expression is a binary or unary expression GenExpr creates a new temporary in the attribute table and returns a pointer to this entry.

GenExpr (implementation)

A schematic of GenExpr is shown below; note that GenExpr takes a pointer to a node in the AST, and returns a pointer to the attribute table.

```
function GenExpr(p : AstPtrType) : AttTabPtrType;
  ...
begin
  if Ast[p].AstClass is a leaf node then begin
    return Ast[p].AttTabPtr;
  end
  else
    case Ast[p].AstClass of
      ...
      AstAdd   : ... ;
      AstSub   : ... ;
      AstMul   : ... ;
      AstDiv   : ... ;
      AstUmin  : ... ;
      AstIndex : ... ;
      AstField : ... ;
      ...
    end case;
  end if;
end;
```

Generating code for binary/unary expressions



A simple algorithm for generating code for expressions where the operands are only identifiers and literals given below.

```
function GenExpr(p : AstPtrType) : AttTabPtrType;
  var arg1, arg2, arg3 : AttTabPtrType;
begin
  if Ast[p].AstClass is a leaf node then begin
    return Ast[p].AttTabPtr;
  end
  else begin
    case Ast[p].AstClass of
    ...
    AstAdd :
      begin
        arg1 := GenExpr(Ast[p].Child[1]);
        arg2 := GenExpr(Ast[p].Child[2]);
```

```
        arg3 := GenTmp();
        Emit(QuadAdd, arg3, arg1, arg2);
        return arg3;
      end;
    ...
    end {case};
  end {if};
end;
```

Example

For the following abstract syntax tree
the following quads will be generated:

```
t1 := a + b
t2 := c + d
t3 := t1 * t2
t4 := e - f
t5 := t3 / t4
z := t5
```

Generating quads for arrays

The issues involved in generating code for arrays are:

- Understanding the layout of arrays allocated either statically or on the stack.
- Optimizing arrays declared with non-zero lower bounds.

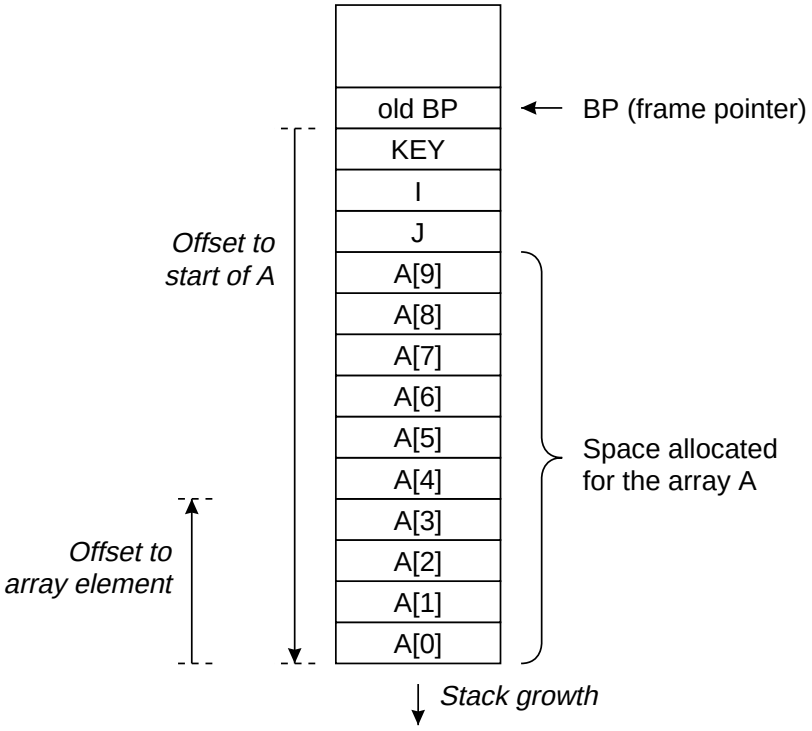
For an array reference A[I], the quads that would be generated to load that value into an index register would be:

```
t1 := I * 2
t2 := a[t1]
```

The assembly code that would be generated from these quadruples would be:

```
mov  di,[bp-4]
shl  di,1
mov  ax,[bp-26][di]
```

The offset to the first element of the array is given by [bp-26] since we assume two-byte integers.



  - When the lower bound of an array is *non-zero* we need to take some special steps to avoid generating additional (and somewhat costly) code.
- Representation of multi-dimensional arrays.
- Other miscellaneous optimizations.
  - When the index expression in an array reference includes simple expressions involving constants, there is an important optimization that can be done.

Arrays with a lower bound of zero

Generating quads for arrays with a lower bound of zero is quite straightforward. However, the issues that need to be addressed are:

- To access an element of an array one must computer the address of that element.
- The address of an elment of a one-dimensional array is described by the following equation:

```
Address( A[I] ) = Base( A ) + ( I * Stride( A) )
```

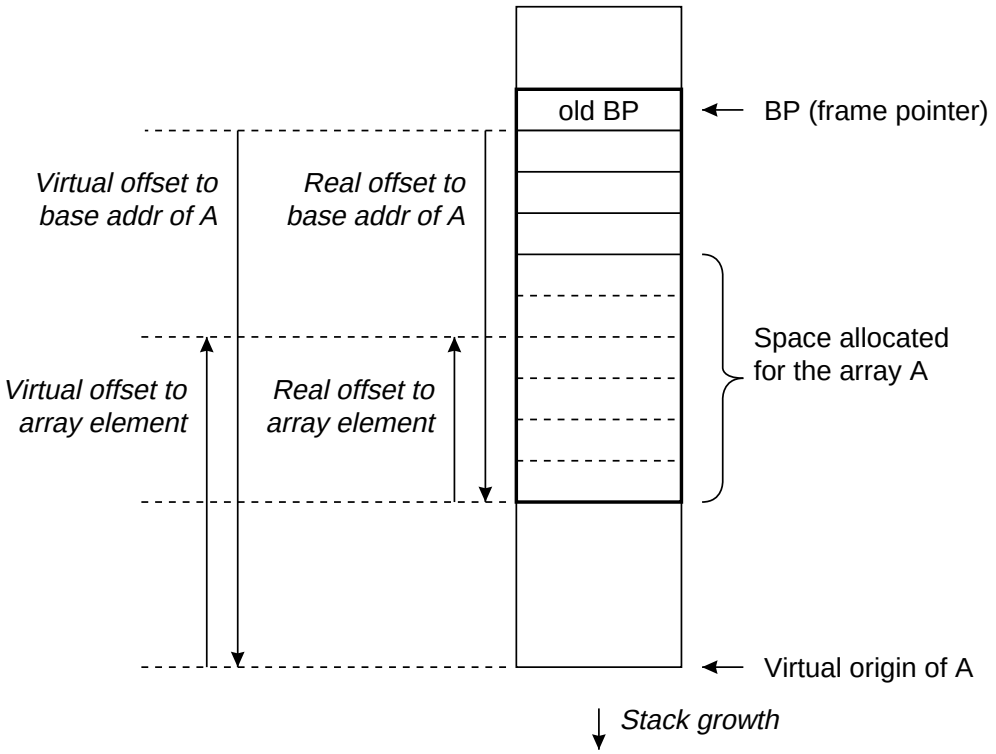
- How the base address of A is computed depends upon whether A is statically or globally allocated, but we do *not* have to consider this during intermediate code generation.

Example

Consider the declarations used in the insertion sort program:

```
key  : integer;
i, j : integer;
a    : array[0..9] of integer;
```

The stack frame for this procedure and code are shown below.



Generating code for arrays

Shown is the portion of GenExpr for generating code for an array reference.

```
function GenExpr(p : AstPtrType) : AttTabPtrType;
  var arg1, arg2, arg3, arg4 : AttTabPtrType;
begin
  if Ast[p].AstClass is a leaf node then begin
    return Ast[n].AttTabPtr;
  end
```

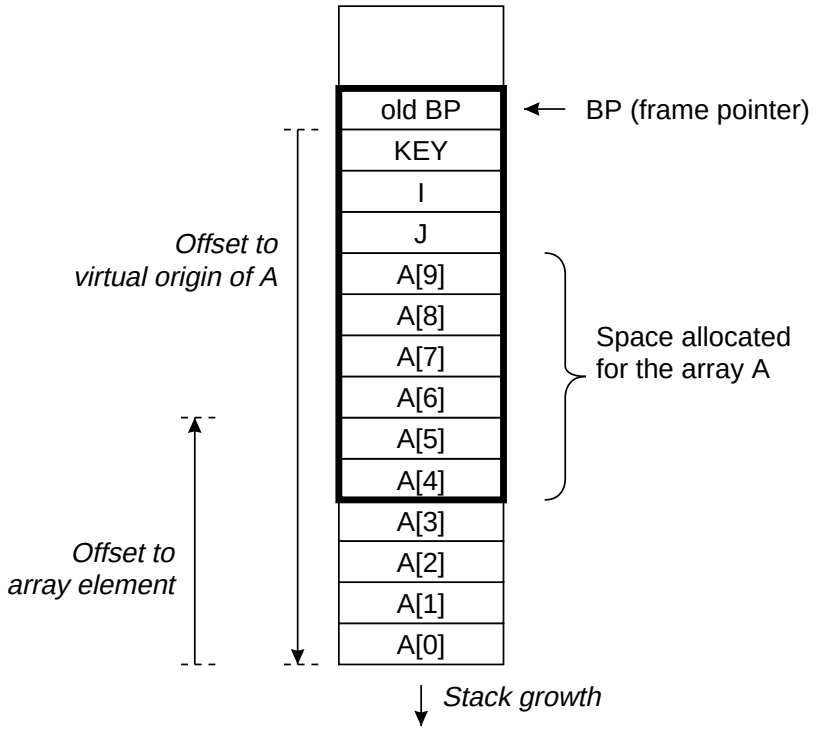
For an array reference A[I], the quads that would be generated to load that value into an index register would be:

```
t1 := I * 2
t2 := a[t1]
```

The assembly code that would be generated from these quadruples would be:

```
mov  di,[bp-4]
shl  di,1
mov  ax,[bp-26][di]
```

The offset to the virtual origin of the array is still given by [bp-26] since we assume two-byte integers.



```
  else begin
    case Ast[p].AstType of
    ...
    AstIndex :
      begin
        arg1 := ArrayName(Ast[p].Child[1]);
        arg2 := GenExpr(Ast[p].Child[2]);
        arg3 := GenTmp();
        Emit(QuadMul, arg3, arg2, Stride(arg1));
        arg4 := GetTmp();
        Emit(QuadOf,  arg4, arg1, arg3);
        return arg4;
      end;
    ...
    end case;
  end if;
end;
```

Arrays with non-zero lower bounds

The problem: We want array accesses to arrays with non-zero lower bounds to take the same number of instructions as those with lower bounds of zero. Consider a simple array access:

```
procedure P;
  const LowBound = ...; HighBound = ...;
  var   a : array of [LowBound .. HighBound];
begin
  x := a[i];
end;
```

The most straighforward code that could be generated to access an element of an array with a non-zero lower bound would be:

```
t1 := i - LowBound
t2 := t1 * 2
t3 := a[t2]
x  := t3
```

as compared to

```
t1 := i * 2
t2 := a[t1]
x  := t2
```

Note that one extra quadruple is generated using the straightforward approach.

Virtual origins

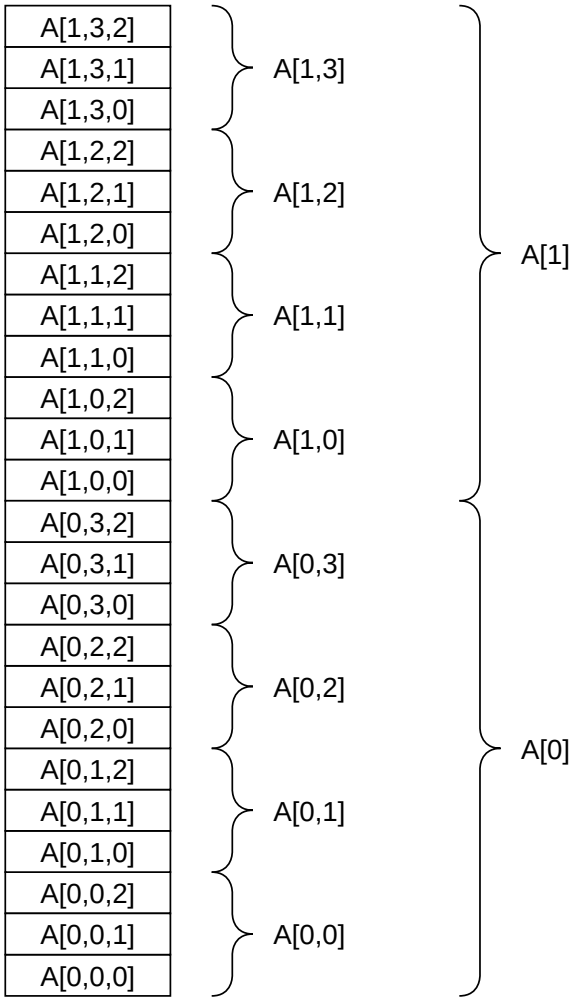
The solution: Address elements of the array as if the array did have a lower bound of zero (even though it really doesn't) by assuming addressing off a base address different from the real base address of the array.

- The virtual origin of a one-dimensional array (with non-zero lower bound) is base address of the array if it had been indexed at 0.

Example

Now consider the same declarations as before, with with the array indexed at a lower bound of 4 instead of 0:

```
key   : integer;
i, j  : integer;
a     : array[4..9] of integer;
```



The situation looks almost the same as before, but there are subtle differences (the virtual origin for A is kept in the attribute table entry for A):

A simple indexing optimization

There are two other standard optimizations that are applied to indexing operations.

- When the index expression is a constant, such as in a[2], the location of the element is known at compile-time.
- When the index expression is a simple expression involving a constant of the form a[i+c] (where c is a constant), an optimization similar to (but not exactly the same as) virtual origins can be applied.

Example

Consider the statement

```
z := a[i+3]
```

Instead of generating the standard set of quads, generate the following quads:

```
t1 := i * 2
t2 := a[t1][+6]
z  := t2
```

Note that the second quad is no longer a real quad! (It has five arguments, and the correction factor +6 must be stored with the quadruple). Assuming the declarations in the next to last example, the assembly code emitted would be:

```
mov  di,i
shl  di,1
mov  ax,[bp-20][di]       ; [bp-(26-6)][di]
```

What has happenned here is that he actual origin of the array (which is 26) has now been “corrected” by subtracting six so that we can index with i instead of i+3.

Multi-dimensional arrays

All multi-dimensional arrays must be represented in linear form since memory is one-dimensional.

- There are two major representations that are used:
  - *Row major form*. The elements of an array are stored in a single linear array in such a way that as one moves through the array the last index varies the fastest.
  - *Column major form*. The elements of an array are stored in a single linear array in such a way that as one moves through the array the first index varies the fastest.
- In both cases, we need a function that will map the logical address of the array element (using its indices) into its location in the one-dimensional representation.

Row major form

Consider the array declared as

```
a : array[0..1,0..3,0..2]
```

The row major representation would be:

The address computation

We need a function that will map the indices of a multi-dimensional array into the location of that element into it's address in the linear array. For the one dimensional case we have:

$$Address(a(i)) = Base(a) + (i * Stride(a))$$

For the two-dimensional case we have:

$$Address(a(i,j)) = Base(a) + [((i * dim2) + j) * Stride(a))]$$

For the n-dimensional case we have:

$$\begin{aligned} Address(a(i_1, i_2, ..., i_n)) = \\ Base(a) \\ + [((i_1 * dim_2 * dim_3 * ... * dim_n) \\ + ((i_2 * dim_3 * dim_4 * ... * dim_n) \\ + \ ... \\ + (i_n)] \ * \ Stride(a) \end{aligned}$$

Here, we assume that dim1, dim2, etc. refer to the total number of elements (not the number of bytes!) in the first, second, etc., dimensions of the array.

Simplifying the address computation

To compute the address of an element of a multi-dimensional array at run-time using the previous expression would be quite expensive, since the product of many of the dimensions is repeated in each term. The original computation

$$
\begin{aligned}
&\text{Address}(a(i_1, i_2, \ldots, i_n)) = \\
&\quad \text{Base}(a) \\
&\qquad + [\ ((i_1 * \text{dim}_2 * \text{dim}_3 * \ldots * \text{dim}_n) \\
&\qquad +\ \ ((i_2 * \text{dim}_3 * \text{dim}_4 * \ldots * \text{dim}_n) \\
&\qquad +\ \ \ldots \\
&\qquad +\ \ (i_n)\ ] \ \ * \ \ \text{Stride}(a)
\end{aligned}
$$

is rearranged as follow:

$$
\begin{aligned}
&\text{Address}(a(i_1, i_2, \ldots, i_n)) = \\
&\quad \text{Base}(a) \\
&\qquad + (\ldots(((i_1 * \text{dim}_2) + i_2) * \text{dim}_3) + \ldots + i_n) * \text{Stride}(a)
\end{aligned}
$$

This computation can be implemented by loading i_1 into a register, the multiplying by dim_2, adding i_2, etc. (all in a single register).

Summary

- There are a number of ntermediate representations (IRs) in use in compilers, but quadruples and abstract syntax trees are two of the most common.
- We have described an intermediate code generator that generates quadruples from abstract syntax trees.
- Several important optimizations can be applied to arrays.