During IR generation, do we make use of target machine specifics?
- Addressing modes
- Specific, weird instructions
- Size of data

**No**
Isolate machine dependencies in final code generation
   To retarget the compiler...
   Replace the final code generation
   Don’t need to modify...  IR code generation
   Optimization phase

**Yes**
IR code generator creates more specific code
The optimizer can improve this code

**Example**
\[
\begin{align*}
  a[i] &:= b[i]; \\
  t1 &:= i*4 \\
  t2 &:= b[t1] \\
  t3 &:= i*4 \\
  a[t3] &:= t2 \\
  t1 &:= i*4 \\
  t2 &:= b[t1] \\
  a[t1] &:= t2
\end{align*}
\]

**Our Approach to Variables**
For each routine (and main body)
- Run through the stmts and generate IR quads
  Also compute “`maxArgNumber`”
  When generating IR code for “foo”
  \[
  \begin{array}{cccc}
  \text{...} & \text{bar(aaa, bbb, ccc, ddd)} & \text{...} \\
  1 & 2 & 3 & 4
  \end{array}
  \]
- Assign offset to our variables
  \[
  \begin{array}{l}
  \text{nextOffset} := -4 \\
  \text{for each VarDecl p do} \\
  \quad p.\text{offset} := \text{nextOffset} \\
  \quad \text{nextOffset} := \text{nextOffset} - 4
  \end{array}
  \]
- Assign offset to our formals
  \[
  \begin{array}{l}
  \text{nextFormal} := 68 \\
  \text{for each Formal f do} \\
  \quad f.\text{offset} := \text{nextFormal} \\
  \quad \text{nextFormal} := \text{nextFormal} + 4
  \end{array}
  \]
- Compute “`sizeOfFrame`”
  \[
  \text{body.sizeOfFrame} := \ldots
  \]
Register window
save area (64 bytes)

Display reg save area

Storage for args to callees (e.g., bar)

Optional alignment word

Space for locals and temporaries

Space for our formals

Frame of our caller

procedure foo \( (x_1, x_2, \ldots, x_M) \)
\[
\begin{align*}
\text{var} & \quad y_1, y_2, \ldots, y_N; \\
\text{begin} & \quad \ldots \text{bar}(z_1, z_2, \ldots, z_P) \ldots \\
\text{end};
\end{align*}
\]

Let “\( P \)” be the MaxArgNumber, for all routines that “foo” calls

---

Records (in C-like languages)

\[
\begin{align*}
\text{var} & \quad x: \text{integer};
\end{align*}
\]
Records (in C-like languages)

```plaintext
var x: integer;

type T is record
    f1: ...;
    f2: ...;
    f3: ...;
end;

var y: T;

var p: ptr to T;
```

© Harry H. Porter, 2006
Records in PCAT

```pascal
type T is record
  f1: ...;
  f2: ...;
  f3: ...;
end;

var r: T;
... := r.f2;
r.f3 := ...
```

**IR Code for reading a field “... := r.f2”**

```
t8 := r+4
  t9 := *t8
  ... := t9
```

**IR Code for setting a field “r.f3 := ...”**

```
t11 := r+8
  t12 := ...
  *t11 := t12
```

We will allocate all records and arrays on the heap

Our Approach

When walking the AST...
we will visit all nodes...
whenever we see a “RecordType” (in genRecordType)...

• Walk the list of fields.
  Fill in the offsets.
• Set the “size” of the entire record (in bytes).
• Must recursively handle nested record types.
Translating Assignment Statements

Source:
\[ x := y + 3; \]

Translation:
\[
\begin{align*}
    t_1 &:= \&x \\
    t_2 &:= ... \\
    *t_1 &= t_2
\end{align*}
\]

Most general approach.
Handles complex L-Values
\[ a[ \text{foo(...)} ] := y + 3; \]

Problem:
Difficult to optimize this:
\[
\begin{align*}
    t &:= \&x \\
    ... \\
    *t &= ...
\end{align*}
\]

Solution:
In `genAssignStmt` watch for special case:
L-Value is a simple variable
\[ t_2 := ... \]
\[ x := t_2 \]

**Goal: Reduce temporary usage!**

Goal: Reduce Temp Usage

\[ E_0 \rightarrow E_1 \ast E_2 \]
Generates:
\[
\begin{align*}
    t_5 &= ...E_1... \\
    t_6 &= ...E_2... \\
    t_3 &= t_5 \ast t_6
\end{align*}
\]

Note: \( t_5 \) and \( t_6 \) are never used again.

Idea: Recycle!

newTemp() \rightarrow temp
Create a new temp and return it.

recycle (temp)
Maintain a collection of unused temporaries
(Each routine will start with an empty collection)
recycle ()
Add the temp to collection.
newTemp() Get a temp from the collection.
Create a new temp only when necessary.
Source: \[ x := ((a*b) + (c*d)) - (e*f); \]

Before:
- \( t1 := a * b \)
- \( t2 := c * d \)
- \( t3 := t1 + t2 \)
- \( t4 := e * f \)
- \( t5 := t3 - t4 \)
- \( x := t5 \)

When to recycle?
In expressions, every temp is used exactly once
So call recycle when the temp is used as an operand.

With Recycling:
- \( t1 := a * b \)
- \( t2 := c * d \)
- \( t1 := t1 + t2 \)
- \( t2 := e * f \)
- \( t1 := t1 - t2 \)
- \( x := t1 \)
Recycling Bins

**PCAT**
Only one kind of temp (4 bytes long)

**Other compilers:**
Many kind of temps...
byte, word, double

**Approach:**
Will have a “bin” (i.e., collection) for each kind of temp.

```java
recycleByteTemp (temp)
```
Returns the temp to the recycling bin for “byte” temps

```java
newByteTemp () \rightarrow temp
```
Check the “byte” recycling bin before creating

```java
recycleWordTemp (temp)
```
```java
newWordTemp () \rightarrow temp
```

```java
recycleDoubleTemp (temp)
```
```java
newDoubleTemp () \rightarrow temp
```

---

Arrays in PCAT

Array of N elements

```plaintext
a[0], a[1], ... a[N-1]
```

All arrays will be stored in the heap.

Will be stored in a block of N+1 words.

The first word will contain
the number of elements in the array.

At runtime, we must check for...

“Index out-of-bounds” error
“Uninitialized array” error

```java
var a: array of real := nil;
...
a := {{ 1000 of 123.456 }};
... a[i] ...
```
Generating Code to Access “a[i]”

Source:
...a[expr]...

Code Generated:

```
t1 := &a
  genLValue
  t1

t2 := *t1
  if t2 = 0 goto null_ptr_error

t3 := ...expr...
  genExpr
  t3

  if t3 < 0 goto bounds_error
  t4 := *t2
  if t3 >= t4 goto bounds_error
  t5 := t3 * 4
  t6 := t5 + t2
  t7 := t6 + 4
```

Now t7 contains the address of the word in question

Array Representation

How is an array stored in memory? 
Where is A[i] stored?
Array Representation

How is an array stored in memory?
Where is A[i] stored?

Assumptions:
• Array starts at A[0]
• No other information (e.g., “size”) stored in the array
• No indirection, no pointers

Let:
\[ w = \text{width (in bytes) of each element} \]
\[ \text{base} = \text{address of 1st byte of the array} \]

The address of A[i]
\[ \text{base} + (i \times w) \]
Array Representation

How is an array stored in memory? Where is A[i] stored?

Assumptions:
- Array starts at A[0]
- No other information (e.g., “size”) stored in the array
- No indirection, no pointers

Let:
- \( w \) = width (in bytes) of each element
- \( \text{base} \) = address of 1st byte of the array

The address of \( A[i] \)
\[
\text{base} + (i \times w)
\]

\[
1000 + (3 \times 8) = 1024
\]
Array Representation

Assumption: Array can start anywhere
A[-5], A[-4], ..., A[0], ...
B[6], B[7], ...

Let:
- \( w \) = width of elements
- \( \text{base} \) = starting address of the array
- \( \text{low} \) = smallest legal index (e.g., -5)

Example
\( w = 8 \) bytes
\( \text{base} = 1000 \)
\( \text{low} = -5 \)

<table>
<thead>
<tr>
<th>Address</th>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>A[-5]</td>
</tr>
<tr>
<td>1008</td>
<td>A[-4]</td>
</tr>
<tr>
<td>1016</td>
<td>A[-3]</td>
</tr>
<tr>
<td>1024</td>
<td>A[-2]</td>
</tr>
<tr>
<td>1032</td>
<td>A[-1]</td>
</tr>
<tr>
<td>1040</td>
<td>A[0]</td>
</tr>
<tr>
<td>1048</td>
<td>A[1]</td>
</tr>
<tr>
<td>1056</td>
<td>A[2]</td>
</tr>
</tbody>
</table>

The address of \( A[i] \)

Before:
\[ \text{base} + (i \times w) \]

Now:
\[ \text{base} + ((i - \text{low}) \times w) \]

\[
1000 + ((2 - (-5)) \times 8) = 1056
= 2 \times 8 + (1000 - (-5 \times 8)) = 16 + 1040 = 1056
\]
The Zero-Normalized Base

Address of A[i]:
\[ \text{base} + ((i - \text{low}) \times \text{w}) \]

Rewriting:
\[ i \times \text{w} + (\text{base} - (\text{low} \times \text{w})) \]

If “base” and “low” are known at compile-time we can precompute this constant:
\[ \text{base} - (\text{low} \times \text{w}) \]

Example

\[ \text{w} = 8 \text{ bytes} \]
\[ \text{base} = 1000 \]
\[ \text{low} = -5 \]

\[
\begin{array}{c|c}
1000 & A[-5] \\
1008 & A[-4] \\
1016 & A[-3] \\
1024 & A[-2] \\
1032 & A[-1] \\
1040 & A[0] \\
1048 & A[1] \\
\end{array}
\]
The Zero-Normalized Base

\textbf{Address of A[i]}: \[ \text{base} + ((\text{i} - \text{low}) \times \text{w}) \]

\textbf{Rewriting}: \[ \text{i} \times \text{w} + (\text{base} - (\text{low} \times \text{w})) \]

If “base” and “low” are known at compile-time we can precompute this constant:

\[ \text{base} - (\text{low} \times \text{w}) \]
\[ 1000 - (-5 \times 8) \]
\[ = 1040 \]

\textit{This is the address of A[0].}

The “zero-normalized base”

\textbf{Address of A[i]}:
\[ \text{i} \times \text{w} + \text{constant} \]

\textbf{Example}

\begin{itemize}
  \item \text{w = 8 bytes}
  \item \text{base = 1000}
  \item \text{low = -5}
\end{itemize}

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[-5]</td>
<td>1000</td>
</tr>
<tr>
<td>A[-4]</td>
<td>1008</td>
</tr>
<tr>
<td>A[-3]</td>
<td>1016</td>
</tr>
<tr>
<td>A[-2]</td>
<td>1024</td>
</tr>
<tr>
<td>A[-1]</td>
<td>1032</td>
</tr>
<tr>
<td>A[0]</td>
<td>1040</td>
</tr>
<tr>
<td>A[1]</td>
<td>1048</td>
</tr>
</tbody>
</table>

\begin{itemize}
  \item Address of A[2]:
  \[ 2 \times 8 + 1040 \]
  \[ = 1056 \]
\end{itemize}
The Zero-Normalized Base

The “zero-normalized base” works, even if array does NOT contain A[0].

Example:

\[
\begin{align*}
\text{low} &= 3 \\
\end{align*}
\]
The Zero-Normalized Base

The “zero-normalized base” works, even if array does NOT contain A[0].

Example:

low = 3

Address of A[i]:

\[ i*w + (base - (low * w)) \]

i\(\times\)w + \text{constant}

Zero-Normalized Base:

\[ base - (low * w) \]

1000 – (3 \times 8) = 976
The Zero-Normalized Base

The “zero-normalized base” works, even if array does NOT contain A[0].

Example:

\[
\text{low} = 3 \\
\]

Address of A[i]:

\[
i \times w + (\text{base} - (\text{low} \times w)) \\
i \times w + \text{constant}
\]

Zero-Normalized Base:

\[
\text{base} - (\text{low} \times w) \\
1000 - (3 \times 8) \\
= 976
\]

Address of A[5]:

\[
i \times w + \text{constant} \\
5 \times 8 + 976 \\
= 1016
\]

Example

\[
w = 8 \text{ bytes} \\
\text{base} = 1000 \\
\text{low} = 3
\]

Multi-Dimensional Arrays

Two Dimensional Arrays:

\[
\text{var } A: \text{ array}[6..8, 1..4] \text{ of double};
\]

\[
\ldots A[i, j] \ldots
\]

Three Dimensional Arrays:

\[
\text{var } B: \text{ array}[6..8, 1..4, 0..9] \text{ of double};
\]

\[
\ldots B[i, j, k] \ldots
\]

Multi-Dimensional Arrays...

\[
\text{var } C: \text{ array}[6..8, 1..4, \ldots, 0..9] \text{ of } \ldots;
\]

How do we place the array elements in memory?
Row-Major Order

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

1000 6,1
1008 6,2
1016 6,3
1024 6,4
1032 7,1
1040 7,2
1048 7,3
1056 7,4
1064 8,1
1072 8,2
1080 8,3
1088 8,4

Row 6

Row 7

Row 8

Column-Major Order

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

1000 6,1
1016 8,1
1024 8,2
1032 8,3
1040 8,4
1048 6,3
1056 7,3
1064 8,3
1072 6,4
1080 7,4
1088 8,4
Row-Major Order

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

3,687
3,688
3,689
3,690
3,691

Row-Major is most common.
Like decimal numbers
(Last digit varies fastest)

Row-Major Order

3 Dimensional Arrays, Row-Major Order

A[6..7, 1..4, 3..7]
3 Dimensional Arrays, Row-Major Order

A[6..7, 1..4, 3..7]

\[
\begin{array}{cccc}
6,1 & 6,2 & 6,3 & 6,4 \\
6,5 & 6,6 & 6,7 & 6,8 \\
6,9 & 6,10 & 6,11 & 6,12 \\
6,13 & 6,14 & 6,15 & 6,16 \\
\end{array}
\]

Row 6
Col. 2

Row 7
Col. 3

Row 8

Where is A[i] Stored?

Assumption: Two-Dimensions, Row-Major Order

var A: array [0..8, 0..4] of real;

Let:

\( w = \) width of elements \( 8 \)

\( \text{base} = \) starting address of the array \( 1000 \)

\( \text{high}_1 = \) highest row index \( A[0..8, 0..4] \)

\( \text{high}_2 = \) highest column index \( A[0..8, 0..4] \)

Compute:

\( N_1 = \) number of rows

\( = \text{high}_1 + 1 \)

\( = 8 + 1 = 9 \)

\( N_2 = \) number of columns

\( = \text{high}_2 + 1 \)

\( = 4 + 1 = 5 \)

\( = \) size of each row!

\( A[i,j]: \)

\( \text{base} + (i * N_2 + j) * w \)
Where is A[i] Stored?

**Assumption:** Two-Dimensions, Row-Major Order

```latex
\texttt{var A: array [6..8, 1..4] of real;}
```

**Let:**
- \( w \) = width of elements = 8
- \( \text{base} \) = starting address of the array = 1000
- \( \text{low}_1 \) = lowest row index = \( A[6..8, 1..4] \)
- \( \text{high}_1 \) = highest row index = \( A[6..8, 1..4] \)
- \( \text{low}_2 \) = lowest column index = \( A[6..8, 1..4] \)
- \( \text{high}_2 \) = highest column index = \( A[6..8, 1..4] \)

**Compute:**
- \( N_1 \) = number of rows
  \[ N_1 = \text{high}_1 - \text{low}_1 + 1 \]
  \[ = 8 - 6 + 1 = 3 \]
- \( N_2 \) = number of columns
  \[ N_2 = \text{high}_2 - \text{low}_2 + 1 \]
  \[ = 4 - 1 + 1 = 4 \]

**A[i,j]:**
- \( \text{base} + ( (i - \text{low}_1) * N_2 + (j - \text{low}_2) ) * w \)

(Repeating...)

**A[i,j] is stored at:**
- \( \text{base} + ( (i - \text{low}_1) * N_2 + (j - \text{low}_2) ) * w \)

6 operations

*Can we compute any of this at compile-time?*
(Repeating...)

\[ A[i,j] \text{ is stored at:} \]
\[ \text{base } + ( (i - \text{low}_1) \times N_2 + (j - \text{low}_2) ) \times w \]
6 operations

Can we compute any of this at compile-time?

Rewriting
\[ ((i \times N_2) + j) \times w + (\text{base} - (\text{low}_1 \times N_2 + \text{low}_2) \times w) \]

• Assume the array bounds are fixed at compile-time.
• Assume the base address is known at compile-time
\[ ((i \times N_2) + j) \times w + (\text{base} - (\text{low}_1 \times N_2 + \text{low}_2) \times w) \]
4 operations

The “Zero-Normalized Base”
The address of \([0,0]\]

Compile-time constant: Pre-compute it!!!

© Harry H. Porter, 2006
Accessing Multi-Dimensional Arrays

Assume all indexes start at zero: \( A[0..\text{high}_1, 0..\text{high}_2, \ldots, 0..\text{high}_K] \)

\( A[i, j] \) is stored at:
\[
\text{base} + (i \times N_2 + j) \times w
\]

The general case: \( A[\text{low}_1..\text{high}_1, \text{low}_2..\text{high}_2, \ldots, \text{low}_K..\text{high}_K] \)

\( A[i, j] \):
\[
\text{base} + ((i - \text{low}_1) \times N_2 + (j - \text{low}_2)) \times w
\]

\( A[i_1, i_2, i_3] \):
\[
\text{base} + ((i_1 \times N_2 + i_2) \times N_3 + i_3) \times w
\]

Number of dimensions = \( K \)

© Harry H. Porter, 2006
Accessing Multi-Dimensional Arrays

Assume all indexes start at zero: \( A[0..\text{high}_1, 0..\text{high}_2, ... , 0..\text{high}_K]\)

\( A[i, j] \) is stored at:
- \( \text{base} + (i \times N_2 + j) \times w \)
- \( A[i_1, i_2, i_3] \):
  - \( \text{base} + ((i_1 \times N_2 + i_2) \times N_3 + i_3) \times w \)
- \( A[i_1, i_2, i_3, i_4] \):
  - \( \text{base} + (((i_1 \times N_2 + i_2) \times N_3 + i_3) \times N_4 + i_4) \times w \)

Number of dimensions = \( K \)

The general case: \( A[\text{low}_1..\text{high}_1, \text{low}_2..\text{high}_2, ... , \text{low}_K..\text{high}_K]\)

\( A[i, j] \):
- \( \text{base} + ((i - \text{low}_1) \times N_2 + (j - \text{low}_2)) \times w \)
- \( A[i_1, i_2, i_3] \):
  - \( \text{base} + ((i_1 - \text{low}_1) \times N_2 + (i_2 - \text{low}_2) \times N_3 + (i_3 - \text{low}_3)) \times w \)
- \( A[i_1, i_2, i_3, i_4] \):
  - \( \text{base} + (((i_1 - \text{low}_1) \times N_2 + (i_2 - \text{low}_2) \times N_3 + (i_3 - \text{low}_3)) \times N_4 + (i_4 - \text{low}_4)) \times w \)

Number of dimensions = \( K \)
3 Dimensional Arrays

\[ A[6..7, 1..4, 3..7] \]

Row 6

\[
\begin{array}{c}
6,1 \\
6,2 \\
6,3 \\
6,4 \\
7,1 \\
7,2 \\
7,3 \\
7,4 \\
8,1 \\
8,2 \\
8,3 \\
8,4 \\
\end{array}
\]

Row 7

Row 8

Col. 2

Col. 3

© Harry H. Porter, 2006
Precomputing the Zero-Normalized Base

\[ A[i_1, i_2, \ldots, i_K] \text{ is stored at:} \]
\[ \text{base} + ( \ldots ( (i_1 - \text{low}_1) \cdot N_2 + (i_2 - \text{low}_2) \cdot N_3 + (i_3 - \text{low}_3) \ldots \cdot N_K + (i_K - \text{low}_K) ) \ldots ) \cdot w \]

Factoring out the constant...
\[ \ldots ( (i_1 \cdot N_2 + i_2) \cdot N_3 + i_3 \ldots \cdot N_K + i_K ) \cdot w \]
\[ + \text{base} - ( \ldots ( (\text{low}_1 \cdot N_2 + \text{low}_2) \cdot N_3 + \text{low}_3) \ldots \cdot N_K + \text{low}_K ) \cdot w \]

Performing the computation at runtime...
\[ i_1 \]
Precomputing the Zero-Normalized Base

$A[i_1, i_2, \ldots, i_K]$ is stored at:

$\text{base} + ( \ldots ((i_1 - \text{low}_1) \times N_2 + (i_2 - \text{low}_2)) \times N_3 + (i_3 - \text{low}_3)) \ldots \times N_K + (i_K - \text{low}_K)) \times w$

Factoring out the constant...

$(\ldots ((i_1 \times N_2 + i_2) \times N_3 + i_3) \ldots \times N_K + i_K)) \times w$

Performing the computation at runtime...

$i_1$

$i_1 \times N_2 + i_2$
Precomputing the Zero-Normalized Base

A[i_1, i_2, \ldots, i_K] is stored at:
base + ( \ldots ((i_1 - low_1) * N_2 + (i_2 - low_2)) * N_3 + (i_3 - low_3)) \ldots * N_K + (i_K - low_K) ) \ast w

Factoring out the constant...
( \ldots ((i_1 * N_2 + i_2) \ast N_3 + i_3) \ldots * N_K + i_K) * w
+ base - ( \ldots ((low_1 * N_2 + low_2) \ast N_3 + low_3) \ldots * N_K + low_K) * w

Performing the computation at runtime...

i_1
i_1 \ast N_2 + i_2
(i_1 \ast N_2 + i_2) \ast N_3 + i_3
\ldots ((i_1 \ast N_2 + i_2) \ast N_3 + i_3) \ldots * N_K + i_K

\ldots ((i_1 \ast N_2 + i_2) \ast N_3 + i_3) \ldots * N_K + i_K) * w
Precomputing the Zero-Normalized Base

\[ A[i_1, i_2, \ldots, i_K] \text{ is stored at:} \]
\[ \text{base} + \left( \ldots \left( (i_1 - \text{low}_1) \cdot N_2 + (i_2 - \text{low}_2) \right) \cdot N_3 + (i_3 - \text{low}_3) \ldots \right) \cdot N_K + (i_K - \text{low}_K) \cdot w \]

Factoring out the constant...

\[ \left( \ldots \left( (i_1 \cdot N_2 + i_2) \cdot N_3 + i_3 \ldots \right) \cdot N_K + i_K \right) \cdot w \]
\[ + \text{base} - \left( \ldots \left( (\text{low}_1 \cdot N_2 + \text{low}_2) \cdot N_3 + \text{low}_3 \ldots \right) \cdot N_K + \text{low}_K \right) \cdot w \]

Performing the computation at runtime...

\[ \ldots \left( \ldots \left( (i_1 \cdot N_2 + i_2) \cdot N_3 + i_3 \ldots \right) \cdot N_K + i_K \right) \cdot w \]
\[ (\ldots (i_1 \cdot N_2 + i_2) \cdot N_3 + i_3) \ldots \cdot N_K + i_K \right) \cdot w + \text{constant} \]

Checking Array Limits

1-Dimensional

- Check \( i \) before computation of address
  \[ \text{low} \leq i \leq \text{high} \]

- Perform the address computation
  \[ p = \text{base} + (i - \text{low}) \cdot w \]
  Check that address is within array
  \[ \text{base} \leq p < (\text{base} + \text{sizeInBytes}) \]
Checking Array Limits

1-Dimensional
- Check i before computation of address
  \[ \text{low} \leq i \leq \text{high} \]
- Perform the address computation
  \[ p = \text{base} + (i - \text{low}) \times w \]
  Check that address is within array
  \[ \text{base} \leq p < (\text{base} + \text{sizeInBytes}) \]

Multi-Dimensional
- Check each index individually
  \[ a[i,j,k] \quad \text{low}_1 \leq i \leq \text{high}_1 \]
  \[ \text{low}_2 \leq j \leq \text{high}_2 \]
  \[ \text{low}_3 \leq k \leq \text{high}_3 \]
- Perform the address computation
  \[ p = \text{base} + \ldots \]
  Check that address is within array
  \[ \text{base} \leq p < (\text{base} + \text{sizeInBytes}) \]

Faster, but flawed!

Example: \( A[6,10] \)

Not in array

Perform address calculation

\[
\begin{align*}
\text{base} + ((i-\text{low}_1)\times N_2 + (j-\text{low}_2))\times w \\
= \text{base} + (9)\times w \\
= \text{base} + ((8-6)\times 4 + (2-1))\times w \\
= \text{base} + (9)\times w
\end{align*}
\]

The access is still within the array!
Arrays in PCAT

Always 1 dimensional.
Always start at zero.

Multi-dimensional arrays?

```pascal
var a: array of integer;
... a[5] ...
... (a[5]) ...
```

In Heap or on Stack

In The HEAP

© Harry H. Porter, 2006
Arrays in PCAT

Always 1 dimensional.
Always start at zero.

**Multi-dimensional arrays?**

```plaintext
var a: array of array of array of integer;
... a[5][7] ... 
... ((a[5])[7])[9] ...
```

In Heap or on Stack

In The HEAP

© Harry H. Porter, 2006
Arrays can have different sizes and some elements may be NIL!

Arrays in PCAT

Always 1 dimensional.
Always start at zero.

Multi-dimensional arrays?

```
var a: array of array of array of integer;
... a[5][7][9] ...
... (((a[5]) [7]) [9]) ...
```

In Heap or on Stack

NIL
NIL
NIL
NIL

Arrays can have different sizes and some elements may be NIL!

Switch Statements

```
switch expr
  case value_1: Stmt-List_1
  case value_2: Stmt-List_2
  ...
  case value_N: Stmt-List_N
  default: Stmt-List_{N+1}
endSwitch
```

In C/C++/Java
Stmt-List_i will fall through to Stmt-List_{i+1}
Must use "break"

AKA: "Case Statements"
Switch Statements

```plaintext
switch expr
  case value_1: Stmt-List_1
  case value_2: Stmt-List_2
  ...
  case value_N: Stmt-List_N
  default: Stmt-List_{N+1}
endSwitch
```

In C/C++/Java
Stmt-List_i will fall through to Stmt-List_{i+1}
Must use “break”

The “value”s must be constants (i.e., statically known)
“Statically Executable”
“Statically Evaluatable”
“Statically Computable”

```plaintext
static final int MAX = 100;
...
  case 43*(17+MAX): Stmt-List_i
```

AKA: “Case Statements”

### “Switch Expression”

“Case Arms”
“Case Clauses”

“Default Case” is optional (If missing, fall through)

Three Implementation Techniques:

1. **Sequence of N explicit tests**

2. **Precompute a table of N entries**
   - Generate code to quickly search this table

3. **Direct Jump Table**
   - Generate a vector of N addresses
   - Use the switch value as an offset into this table
   - Execute a “Jump-Indirect” through the table
(1) Sequence of N Tests

Treat the switch statement exactly like a sequence if-then-else statements.

```
switch expr
  case value_1: Stmt-List_1
  case value_2: Stmt-List_2
  ...
  case value_N: Stmt-List_N
  default: Stmt-List_{N+1}
endSwitch
```

t := expr
  if t = value_1 then
    Stmt-List_1
  elseIf t = value_2 then
    Stmt-List_2
  ...
  elseIf t = value_N then
    Stmt-List_N
  else
    Stmt-List_{N+1}
endIf
(1) Sequence of N Tests

...code for Expr...

if t ≠ Value₁ goto Lab₁
...code for Stmt-List₁...
goto endLabel
Lab₁:

if t ≠ Value₂ goto Lab₂
...code for Stmt-List₂...
goto endLabel
Lab₂:

...

if t ≠ Valueₙ goto Labₙ
...code for Stmt-Listₙ...
goto endLabel
Labₙ:

...code for Stmt-Listₙ₊₁...

endLabel:

This code is easy to generate. But, it is difficult to recognize it as a “switch statement.” We want to do that during the Optimization phase.

Prologue

Case Arms

Code for Default

Code for default statements (optional)
(1) Sequence of N Tests

...code for Expr...

if t = Value_1 goto Lab_1
if t = Value_2 goto Lab_2
...
if t = Value_N goto Lab_N

goto Lab_{N+1}

Lab_1:
...code for Stmt-List_1...
goto endLabel

Lab_2:
...code for Stmt-List_2...
goto endLabel

...

Lab_N:
...code for Stmt-List_N...
goto endLabel

Lab_{N+1}:
...code for Stmt-List_{N+1}...

endLabel:

For C/C++/Java... the “break” statement produces this “goto”

Code for default statements (optional)

Epilogue

Prologue

Perhaps this can be optimized during optimization phase!
(1) Sequence of N Tests

To generate code for a Switch statement...

Prologue
Create a new label “endLabel”
Generate code to evaluate the \texttt{expr} into temporary “t”
Run through all case arms
  Compute Value_1
  Create a new label “Lab_1”
  Generate “if t=\texttt{Value}_1 \texttt{goto Lab}_1”
  Remember each label
  Generate “\texttt{goto Lab}_{N+1}”

Run through all case arms a second time...

To generate code for
  \texttt{case Value}_1 : \texttt{Stmt-List}_1
  Generate label “Lab_1 ;”
  Generate the code for \texttt{Stmt-List}_1
  Generate label “\texttt{goto endLabel}”

Epilogue
Generate label “Lab_{N+1} :”
Generate code for default statements (optional)
Generate label “endLabel :”

Previously:

if t = \texttt{Value}_1 \texttt{goto Lab}_1
if t = \texttt{Value}_2 \texttt{goto Lab}_2
...
if t = \texttt{Value}_N \texttt{goto Lab}_N
\texttt{goto Lab}_{N+1}

Ideas for IR Instructions:

\begin{verbatim}
switch t,Lab_{N+1}
case Value_1,Lab_1
case Value_2,Lab_2
...
case Value_N,Lab_N
\end{verbatim}

Maybe we can generate fast code to search for the right case...
(2) Precompute a Table and Search It

Approach:
Build a table in static storage
Each entry contains a Value and a Label
Generate code to search the table
Upon finding the matching value
The code will jump to the stored label

<table>
<thead>
<tr>
<th>VTAB</th>
<th>LTAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value_1</td>
<td>Lab_1</td>
</tr>
<tr>
<td>Value_2</td>
<td>Lab_2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Value_N</td>
<td>Lab_N</td>
</tr>
</tbody>
</table>

Implementation of the table

Table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>7903</td>
<td>Lab_43</td>
</tr>
<tr>
<td>4067</td>
<td>Lab_44</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>8989</td>
<td>Lab_45</td>
</tr>
</tbody>
</table>

Code to do a linear search

```
p := &Table
Loop:
    if Table[p] = t
    goto *(Table[p+4])
p := p + 8
    goto Loop
```

Dealing with the default case?
The Switch expression value does not match any “Case Clause”

Idea: Use a “Sentinel”
• Add one extra entry to the table.
  Use the label of the default code.
  Will fill in value at runtime.
• Generate code to store the value of “t” into the last entry.
  During the search...
  If no values match, we’ll match the last entry!
Use a Hash Table

Linear search is slow... Use a hash-based search!

At compile-time
We know the number of values.
Determine the optimal hash table size.
Determine the hash function.
Build the table (pre-compile it)

Generate code to:
• Compute the switch expression
• Compute Hash(expr)
• Search the table until...
  Match is found
  Null entry is found
• Jump indirect through the table

Hash Search Example

Source:

```
switch x+17
  case 2004: Stmt-List1
  case 5006: Stmt-List2
  case 4003: Stmt-List3
  case 7009: Stmt-List4
  case 6006: Stmt-List5
  case 3001: Stmt-List6
  default: Stmt-List_{N+1}
endSwitch
```

© Harry H. Porter, 2006
Hash Search Example

Source:

```
switch x+17
  case 2004: Stmt-List₁
  case 5006: Stmt-List₂
  case 4003: Stmt-List₃
  case 7009: Stmt-List₄
  case 6006: Stmt-List₅
  case 3001: Stmt-List₆
  default: Stmt-List₇₊₁
endSwitch
```

Number of cases: 6
Hash Table size: 10
Hash Function: \( \text{hash}(v) = v \mod 10 \)

2004 \( \rightarrow \) 4
5006 \( \rightarrow \) 6
4003 \( \rightarrow \) 3
7009 \( \rightarrow \) 9
6006 \( \rightarrow \) 6
3001 \( \rightarrow \) 1

© Harry H. Porter, 2006
Hash Search Example

Source:
```
switch x+17
  case 2004: Stmt-List_1
  case 5006: Stmt-List_2
  case 4003: Stmt-List_3
  case 7009: Stmt-List_4
  case 6006: Stmt-List_5
  case 3001: Stmt-List_6
  default: Stmt-List_{N+1}
endSwitch
```

Build Table

<table>
<thead>
<tr>
<th>VTAB</th>
<th>LTAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>3001 Lab_6</td>
</tr>
<tr>
<td>2</td>
<td>0 NULL</td>
</tr>
<tr>
<td>3</td>
<td>4003 Lab_3</td>
</tr>
<tr>
<td>4</td>
<td>2004 Lab_1</td>
</tr>
<tr>
<td>5</td>
<td>0 NULL</td>
</tr>
<tr>
<td>6</td>
<td>5006 Lab_2</td>
</tr>
<tr>
<td>7</td>
<td>6006 Lab_5</td>
</tr>
<tr>
<td>8</td>
<td>0 NULL</td>
</tr>
<tr>
<td>9</td>
<td>7009 Lab_4</td>
</tr>
</tbody>
</table>

Number of cases: 6
Hash Table size: 10
Hash Function: \(\text{hash}(v) = v \mod 10\)

At runtime...

- Compute \(t := x+17\)
- Compute Hash(t)
- Search the table
- If VTAB[p] matches... Jump-Indirect
- If LTAB[p] = null, jump to default Lab_{N+1}

Direct-Jump Table

```
switch x+17
  case 4: Stmt-List_1
  case 6: Stmt-List_2
  case 3: Stmt-List_3
  case 9: Stmt-List_4
  case 7: Stmt-List_5
  case 2: Stmt-List_6
  default: Stmt-List_{N+1}
endSwitch
```

Determine the range of values
Build a table this size.
Each table entry will contain a label
(3) Direct-Jump Table

```plaintext
switch x+17
    case 4: Stmt-List₁
    case 6: Stmt-List₂
    case 3: Stmt-List₃
    case 9: Stmt-List₄
    case 7: Stmt-List₅
    case 2: Stmt-List₆
    default: Stmt-List₇₁
endSwitch
```

Determine the range of values
Build a table this size.
Each table entry will contain a label

Unused table entries?
Fill with \( \text{Lab}_{N+1} \)

© Harry H. Porter, 2006
### (3) Direct-Jump Table

```plaintext
switch x+17
  case 4: Stmt-List_1
  case 6: Stmt-List_2
  case 3: Stmt-List_3
  case 9: Stmt-List_4
  case 7: Stmt-List_5
  case 2: Stmt-List_6
  default: Stmt-List_{N+1}
endSwitch
```

Determine the range of values
Build a table this size.
   Each table entry will contain a label
Unused table entries?
   Fill with Lab_{N+1}

Generate code to...
   Compute the switch expression
   Use t as an index into the table
   Perform Indirect-Jump through the table

LTAB

<table>
<thead>
<tr>
<th>0</th>
<th>Lab_{N+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lab_{N+1}</td>
</tr>
<tr>
<td>2</td>
<td>Lab_6</td>
</tr>
<tr>
<td>3</td>
<td>Lab_3</td>
</tr>
<tr>
<td>4</td>
<td>Lab_1</td>
</tr>
<tr>
<td>5</td>
<td>Lab_{N+1}</td>
</tr>
<tr>
<td>6</td>
<td>Lab_2</td>
</tr>
<tr>
<td>7</td>
<td>Lab_4</td>
</tr>
<tr>
<td>8</td>
<td>Lab_{N+1}</td>
</tr>
<tr>
<td>9</td>
<td>Lab_4</td>
</tr>
</tbody>
</table>

This approach only works when the range of values is “small”.
Otherwise, LTAB is too large.

NOTE: The values can be “shifted”

\[
35,002..35,009 \Rightarrow 0.7
\]
Switch Table Implementation - Recap

(1) Sequence of Explicit Tests
(2) Table plus Search
   Linear Search
   Hash-based search
   Other search (e.g., Binary Search)
(3) Direct Jump Table
   Very Fast!
   ...but, can only use if range is small

Which method is best?

Which method(s) does “gcc” use?
   ...Under what circumstances?

Nice to know how smart/dumb your compiler is...
   ...so you can write efficient code!