Part of the job of the back end of a compiler is understanding how to allocate and access data in the memory structure of the target machine.

Storage of data in memory is normally considered using a base-displacement model, which subsumes both LL,ON addressing and the addressing on real machines.

Data descriptors are a general formal notation for base-displacement addressing.

The addressing capabilities of a machine can be characterized as the set of data descriptors for which the machine has corresponding addressing modes - these parameterize the code generator.

Storage allocation for any machine can be achieved starting with a simple table of sizes and alignments for the primitive types of the language - a general machine independent algorithm handles all storage allocation from there.
Code Generation

• Code Generation is the translation of the semantics of the program (e.g., as intermediate or abstract machine code) to target machine language or assembly language

• Dependent on target computer (obviously), but mostly language independent

• Primary Goal: good code, but what is good code? e.g., time/space tradeoff

• Example: loop unrolling

```
for i := 1 to 1000
    x[i] := y[i]
end for

Better for space
```

```
x[1] := y[1]
...
x[1000] := y[1000]

Better for time
```

• Fortunately usually, but not always, smaller code is faster - so most compilers concentrate on making smaller code
• Improving the speed or space requirements of a code sequence is called *optimization*

• There are three basic kinds of optimization techniques

  – *Global* optimization – Analysis of the program as a whole for transformation to a more efficient program (but without changing the algorithm). Usually done separately, either before code generation (*source level* global optimization) or after (*machine code* global optimization)

  – *Local* optimization – Analysis of a single expression or statement to make improvements to its generated code. Typically done as part of code generation.

  – *Peephole* optimization – Analysis of the generated machine or assembly code for certain sequences. Examines a small window of the code as it is output (hence the name "peephole").
Global Optimization Techniques

- **Global optimization** involves looking at many statements at once (sometimes entire methods or classes) to look for opportunities to reorder or recode to improve performance.

1. **Code Motion** – move unchanging code sequences out of loops
e.g., subscripts that don't change

```plaintext
for i := 1 to 10
    x[i] := y[j] * i
end for
```

```plaintext
for i := 1 to 10
    t := y[j]
    x[i] := t * i
end for
```
Global Optimization Techniques

2. *Strength reduction* – reduce expensive operations in loops to cheaper ones by using iterative computation

- **Example**: Loops of higher cost instructions can be expressed in terms of lower cost instructions by taking advantage of the loop

\[
\begin{align*}
t &:= y[j] \\
\text{for } i &:= 1 \text{ to } 10 \\
\quad x[i] &:= t \times i \\
\text{end for}
\end{align*}
\]

\[
\begin{align*}
t &:= y[j] \\
tz &:= t \\
\text{for } i &:= 1 \text{ to } 10 \\
\quad x[i] &:= tz \\
\quad tz &:= tz + t \\
\text{end for}
\end{align*}
\]
Global Optimization Techniques

2. **Strength reduction** (cont'd)

• Opportunities for strength reduction are not always directly visible in the source program.

```plaintext
for i := 1 to 100
    for j := 1 to 100
        a[i][j] := 0
    end for
end for
```

• **Example**: Array indexing (especially with multiple indexes) hides a multiplication, which can often be strength reduced

```plaintext
a[i][j] = Memory [addr(a) + (i-loweri) * (upperj-lowerj+1) + j - lowerj]
```
Global Optimization Techniques

2. *Strength reduction* (cont'd)

- **Example:**

```plaintext
for i := 1 to 100
  for j := 1 to 100
    Memory [addr(a) + (i-loweri) * 
             (upperj-lowerj+1) + j - lowerj] := 0
  end for
end for
```

becomes:

```plaintext
t := (upperj-lowerj+1)
tz := (1-loweri) * t
for i := 1 to 100
  for j := 1 to 100
    Memory [addr(a) + tz + j - lowerj] := 0
  end for
tz := tz + t
end for
```
Global Optimization Techniques

3. **Common subexpression elimination** - the same subexpression may be calculated more than once in a loop or sequence of code - optimize by calculating once and using the result

- **Example:**

  \[
  x := y[i+j-5] \times z[i] \\
  \ldots \\
  a := b \times x + z[i+j-5]
  \]

  becomes:

  \[
  t := i+j-5 \\
  x := y[t] \times z[i] \\
  \ldots \\
  a := b \times x + x[t]
  \]

( where \( i, j \) unchanged)
Global Optimization Techniques

- Global optimization includes many other more sophisticated techniques, such as **parallelization**

- Often carried out on the assembly or machine code after code generation, as well as on the intermediate or source code before

- In general **expensive** (in compile time) - so only very high quality compilers implement it (**PT** does **no** global optimization)

- Most effective when speed of result is essential (so cost of execution greatly outweighs cost of compilation) - for example, in **numerical computation** such as weather prediction, physical simulations, and so on
Local Optimization Techniques

• Local optimization involves working to improve the code within a single statement or expression

• Often done as part of code generation (as in PT)
1. **Coalescence** - combine two or more operations into one based on target machine instruction capabilities.

   • **Example**: on two operand machines (such as the x86 family), the general code for addition statements is of the form:

   \[
   x := z + y \\
   \text{load } y, R1 \\
   \text{add } z, R1 \\
   \text{store } R1, x
   \]

   However, by recognizing the special case:

   \[
   x := x + y \\
   \text{load } y, R1 \\
   \text{add } x, R1 \\
   \text{store } R1, x
   \]

   as the coalesced operator \(+=\) ("add-assign"), we get:

   \[
   x += y \\
   \text{load } y, R1 \\
   \text{add } R1, x
   \]

   or on some machines:

   \[
   x += y \\
   \text{add } y, x
   \]
2. **Constant folding** - if the result of an expression can be computed at compile time, then evaluate it at compile time and use the instead of generating code for it

- **Example:**

```plaintext
const a := 10
const b := 33

x := a * 5 + b
```

\[ x := 83 \]
2. *Constant folding* (cont'd)

- While some constant folding can be done at semantic analysis time, some cannot be done until after storage has been allocated.

- **Example:**
  
  ```
  x := a[5]
  x := Memory[addr(a) + elementsize(a) * 4]
  x := Memory[15780 + 8 * 4]
  x := Memory[15812]
  ```

- For this reason, many compilers leave constant folding to the code generator, although some (e.g., *Java*) do it in both phases.
3. **Local strength reduction** – replace expensive operations with cheaper ones where possible. Global strength reduction took advantage of loop iteration to reduce - local strength reduction just looks for special cases where reduction can be done directly

**Example:**

\[
\begin{align*}
x &:= y \times 2 \\
a &:= b \times 16
\end{align*}
\]

\[
\begin{align*}
x &:= y + y \\
a &:= b \ll 4
\end{align*}
\]
4. **Machine idioms** – recognition of opportunities to use special purpose optimized instructions of the target machine

- Example:

```plaintext
x := x + 1  x += 1  inc  x

var s,t : string
s := t  mvc  s(256),t
```

- Other examples include use of special addressing modes specifically designed for case tables (PDP-11), special instructions or addressing modes for subscripting, and so on
Peephole Optimization

• A peephole optimizer acts as a filter on the output code of the code generator - it watches the last few instructions generated to see if it can improve them by recognizing local redundancies, etc.

• The number of instructions it considers at a time is called the peephole "window"

• The peepholer watches the window as instructions are generated looking for instances of instruction patterns it knows how to improve

• Example:

\[
\begin{align*}
\text{mov} & \quad Rn, m \\
\text{mov} & \quad m, Rn \\
\downarrow & \\
\text{mov} & \quad Rn, m
\end{align*}
\]

(but how could that happen?)
Peephole Optimization

- Peephole optimizer (cont'd)

- Example:

  \[
  a := b + c \quad \text{mov} \quad b, R1 \quad \text{mov} \quad Rn, m \\
  \text{add} \quad c, R1 \quad \text{mov} \quad m, Rn \\
  \text{mov} \quad R1, a \\
  x := a + y \quad \text{mov} \quad a, R1 \\
  \text{add} \quad y, R1 \\
  \text{mov} \quad R1, x \\
  \text{mov} \quad Rn, m
  \]
Peephole Optimization

• *Peephole optimizer* (cont'd)

• **Example** - window size 2:

\[
\begin{align*}
  a & := b + c \\
  x & := a + y \\
  \text{mov } b, R1 \\
  \text{add } c, R1 \\
  \text{mov } R1, a \\
  \text{mov } a, R1 \\
  \text{add } y, R1 \\
  \text{mov } R1, x
\end{align*}
\]

\[
\begin{align*}
  \text{mov } Rn, m \\
  \text{mov } m, Rn \\
  \text{mov } Rn, m
\end{align*}
\]

• Peephole optimizers can find things that are hard for the code generator to notice any other way
Summary

• Generated code quality can be significantly improved using **code optimization** techniques

• Three basic kinds:
  - *global* – many statements at a time
  - *local* – within a statement or expression
  - *peephole* – in sequences of adjacent instructions

• **Next time**:
  - Code generation itself

• Semantic Phase **due next Wednesday**