Three major categories

- **Structural**
  - Graphically oriented
  - Heavily used in source-to-source translators
  - Tend to be large

- **Linear**
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange

- **Hybrid**
  - Combination of graphs and linear code

Examples:
- Trees, DAGs
- 3 address code
- Stack machine code
- Control-flow graph
An abstract syntax tree is the procedure’s parse tree with the nodes for most non-terminal nodes removed.

\[ x - 2 * y \]

- Can use linearized form of the tree
  - Easier to manipulate than pointers
    - \( x \ 2 \ y \ * \ - \) in postfix form
    - \( - \ * \ y \ 2 \ x \) in prefix form

- S-expressions are (essentially) ASTs (remember functional languages such as Scheme or Lisp!)
A directed acyclic graph (DAG) is an AST with a unique node for each value.

- Makes sharing explicit
- Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!

Expression: $z \leftarrow x - 2 \times y$

Expression: $w \leftarrow x / 2$

Diagram:

- Node $z$ connected to $x$ and $2$
- Node $-\times$ connected to $x$ and $y$
- Node $w$ connected to $x$
- Node $l$ connected to $w$
Review - Stack Machine Code

Originally used for stack-based computers, now Java

- Example:
  \[ x - 2 \times y \]
  becomes

Advantages

- Compact form
- Introduced names are \textit{implicit}, not \textit{explicit}
- Simple to generate and execute code

Useful where code is transmitted over slow communication links (\textit{the net})
Several different representations of three address code

- In general, three address code has statements of the form:
  \[ x \leftarrow y \ op \ z \]
  
  With 1 operator (\( \textit{op} \)) and, at most, 3 names (\( x, y, z \))

Example:
\[ z \leftarrow x - 2 \times y \]

becomes
\[ t \leftarrow 2 \times y \]
\[ z \leftarrow x - t \]

Advantages:
- Resembles many machines
- Introduces a new set of names
- Compact form
Naïve representation of three address code

- Table of $k \times 4$ small integers
- Simple record structure
- Easy to reorder
- Explicit names

```
load r1, y  
loadI r2, 2 
mult r3, r2, r1
load r4, x  
sub   r5, r4, r3
```

RISC assembly code

<table>
<thead>
<tr>
<th>Quadruples</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 1 Y</td>
</tr>
<tr>
<td>loadI 2 [2]</td>
</tr>
<tr>
<td>mult 3 2 1</td>
</tr>
<tr>
<td>load 4 X</td>
</tr>
<tr>
<td>sub 5 4 2</td>
</tr>
</tbody>
</table>

The original FORTRAN compiler used “quads”
Review - Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load y</td>
</tr>
<tr>
<td>2</td>
<td>loadI 2</td>
</tr>
<tr>
<td>3</td>
<td>mult (1) (2)</td>
</tr>
<tr>
<td>4</td>
<td>load x</td>
</tr>
<tr>
<td>5</td>
<td>sub (4) (3)</td>
</tr>
</tbody>
</table>

Implicit names take no space!
Review - Control-flow Graph (CFG)

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example

```
if (x == y)
    a ← 2
    b ← 5
a ← 3
b ← 4
c ← a * b
```

Basic blocks — Maximal length sequences of straight-line code
Review - Static Single Assignment Form (SSA)

- The main idea: each name defined exactly once in program
- Introduce $\phi$-functions to make it work

### Original

\[
\begin{align*}
x & \leftarrow \ldots \\
y & \leftarrow \ldots \\
\text{while } (x < k) & \\
x & \leftarrow x + 1 \\
y & \leftarrow y + x
\end{align*}
\]

### SSA-form

\[
\begin{align*}
x_0 & \leftarrow \ldots \\
y_0 & \leftarrow \ldots \\
\text{if } (x_0 > k) & \text{ goto next} \\
\text{loop: } x_1 & \leftarrow \phi(x_0, x_2) \\
y_1 & \leftarrow \phi(y_0, y_2) \\
x_2 & \leftarrow x_1 + 1 \\
y_2 & \leftarrow y_1 + x_2 \\
\text{if } (x_2 < k) & \text{ goto loop} \\
\text{next: } & \ldots
\end{align*}
\]

### Strengths of SSA-form

- Sharper analysis
- $\phi$-functions give hints about placement
- (sometimes) faster algorithms
A compiler is a lot of fast stuff followed by some hard problems

→ The hard stuff is mostly in code generation and optimization

→ For superscalars, its allocation & scheduling that is particularly important
Conventional wisdom says that we lose little by solving these problems independently

Instruction selection
- Use some form of pattern matching
- Assume enough registers or target “important” values

Instruction scheduling
- Within a block, list scheduling is “close” to optimal
- Across blocks, build framework to apply list scheduling

Register allocation
- Start from virtual registers & map into $k$
- Focus on good priority heuristic
Definition
- All those nebulous properties of the code that impact performance & code “quality”
- Includes code, approach for different constructs, cost, storage requirements & mapping, & choice of operations
- Code shape is the end product of many decisions \((big & small)\)

Impact
- Code shape influences algorithm choice & results
- Code shape can encode important facts, or hide them

Rule of thumb: expose as much derived information as possible
- Example: explicit branch targets in ILOC simplify analysis
An example

- What if $x$ is 2 and $z$ is 3?
- What if $y+z$ is evaluated earlier?

The “best” shape for $x+y+z$ depends on contextual knowledge → There may be several conflicting options

Addition is commutative & associative for integers
Code Shape

An example

- What if $x$ is 2 and $z$ is 3?
- What if $y+z$ is evaluated earlier?

The “best” shape for $x+y+z$ depends on contextual knowledge

- There may be several conflicting options
- $x + (5 + y)*7$

Addition is commutative & associative for integers
Another example -- the case statement

- Implement it as cascaded if-then-else statements
  - Cost depends on where your case actually occurs
  - $O(\text{number of cases})$
- Implement it as a binary search
  - Uniform ($\log n$) cost
- Implement it as a jump table
  - Lookup address in a table & jump to it
  - Uniform (constant) cost

```c
Switch (var) {
    case 1:
        statement 1; break;
    case 2:
        statement 2; break;
    case 3:
        statement 3: break;
    ...
    default:
        ....
}
```
Generating Code for Expressions

The key code quality issue is holding values in registers

• When can a value be safely allocated to a register?
  → When only 1 name can reference its value (no aliasing)
    int *a, b; ....; a = &b; b = ...; ...; c = *a;
  → Pointers, parameters, aggregates & arrays all cause trouble

• When should a value be allocated to a register?
  → When it is both safe & profitable

Encoding this knowledge into the IR (register-register model)

• Use code shape to make it known to every later phase
• Assign a virtual register to anything that can go into one
• Load or store the others at each reference

Relies on a strong register allocator
Generating Code for Expressions

The concept

- Use a simple treewalk evaluator
- Bury complexity in routines it calls
  - base(), offset(), & val()
- Implements expected behavior
  - Visits & evaluates children
  - Emits code for the op itself
  - Returns register with result
- Works for simple expressions
- Easily extended to other operators
- Does not handle control flow

```
expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case ×,÷,+,- :
            t1 ← expr(left child(node));
            t2 ← expr(right child(node));
            result ← NextRegister();
            emit (op(node), t1, t2, result);
            break;
        case IDENTIFIER:
            t1 ← base(node);
            t2 ← offset(node);
            result ← NextRegister();
            emit (loadAO, t1, t2, result);
            break;
        case NUMBER:
            result ← NextRegister();
            emit (loadI, val(node), none, result);
            break;
    }
    return result;
}
```

“Expr” returns virtual register number that will contain result of subtree evaluation at runtime.
Generating Code for Expressions

Example:

\[ + \]
\[ x \rightarrow y \]

Produces:

\[
\text{expr(“x”)} \rightarrow \\
\text{loadI } \ @x \quad \Rightarrow \ r1 \\
\text{loadAO } \ r0, r1 \quad \Rightarrow \ r2 \\
\text{expr(“y”)} \rightarrow \\
\text{loadI} \quad @y \quad \Rightarrow \ r3 \\
\text{loadAO} \quad r0, r3 \quad \Rightarrow \ r4 \\
\text{NextRegister()} \rightarrow \ r5 \\
\text{emit(add,r2,r4,r5)} \rightarrow \\
\text{add} \quad r2, r4 \quad \Rightarrow \ r5
\]
Generating Code for Expressions

Example:

```
expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case ×,÷,+,- :
            t1 ← expr(left child(node));
            t2 ← expr(right child(node));
            result ← NextRegister();
            emit (op(node), t1, t2, result);
            break;
        case IDENTIFIER:
            t1 ← base(node);
            t2 ← offset(node);
            result ← NextRegister();
            emit (loadAO, t1, t2, result);
            break;
        case NUMBER:
            result ← NextRegister();
            emit (loadI, val(node), none, result);
            break;
    }
    return result;
}
```

Generates:

```
loadl @x    ⇒ r1
loadAO r0, r1 ⇒ r2
loadl 2     ⇒ r3
loadl @y    ⇒ r4
loadAO r0,r4 ⇒ r5
mult r3, r5 ⇒ r6
sub r2, r6  ⇒ r7
```
More complex cases for IDENTIFIER

- **What about values in registers?**
  - Modify the IDENTIFIER case
  - Already in a register ⇒ return the register name
  - Not in a register ⇒ load it as before, but record the fact
  - Choose names to avoid creating false dependences (“fresh” virtual register)

- **What about parameter values?**
  - Many linkages pass the first several values in registers
  - Call-by-value ⇒ just a local variable with “funny” offset
  - Call-by-reference ⇒ needs an extra indirection

- **What about function calls in expressions?**
  - Generate the calling sequence & load the return value
  - Severely limits compiler’s ability to reorder operations
    (We will learn about this later)
Extending the Simple Treewalk Algorithm

Adding other operators (in addition to +, -, *, /)
- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls
- Handle assignment as an operator

Mixed-type expressions
- Insert conversions as needed from conversion table
- Most languages have symmetric & rational conversion tables

<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>
Extending the Simple Treewalk Algorithm

What about evaluation order?
- Can use commutativity & associativity to improve code
- This problem is truly hard

What about order of evaluating operands?
- 1\textsuperscript{st} operand must be preserved while 2\textsuperscript{nd} is evaluated
- Takes an extra register for 2\textsuperscript{nd} operand
- Should evaluate more demanding operand expression first
  (Ershov in the 1950\textquotesingle s, Sethi in the 1970\textquotesingle s)

Taken to its logical conclusion, this creates Sethi-Ullman register allocation scheme (ASU p. 571)
Generating Code in the Parser

Need to generate an initial IR form
- Might generate an AST, use it for some high-level, near-source work (type checking, optimization), then traverse it and emit a lower-level IR similar to ILOC

The big picture
- Recursive expr(node) algorithm really works bottom-up
  - Actions on non-leaves occur after children are done
- Can encode same basic structure into ad-hoc SDT scheme
  - Identifiers load themselves & stack (live in parser stack) virtual register name
  - Operators emit appropriate code & stack resulting VR name
  - Assignment requires evaluation to an lvalue or an rvalue
Ad-hoc SDT versus a Recursive Treewalk

expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case ×,÷,+,−:
            t1 ← expr(left child(node));
            t2 ← expr(right child(node));
            result ← NextRegister();
            emit (op(node), t1, t2, result);
            break;
        case IDENTIFIER:
            t1 ← base(node);
            t2 ← offset(node);
            result ← NextRegister();
            emit (loadAO, t1, t2, result);
            break;
        case NUMBER:
            result ← NextRegister();
            emit (loadI, val(node), none, result);
            break;
    }
    return result;
}

Goal : Expr { $$ = $1; } ;
Expr: Expr PLUS Term
    { t = NextRegister();
    emit(add,$1,$3,t); $$ = t; }
    | Expr MINUS Term {...}
    | Term { $$ = $1; } ;
Term: Term TIMES Factor
    { t = NextRegister();
    emit(mult,$1,$3,t); $$ = t; }
    | Term DIVIDES Factor {...}
    | Factor { $$ = $1; };
Factor: NUMBER
    { t = NextRegister();
    emit(loadI,val($1),none, t );
    $$ = t; }
    | ID
    { t1 = base($1);
    t2 = offset($1);
    t = NextRegister();
    emit(loadAO,t1,t2,t);
    $$ = t; }
More code generation
Read EaC: Chapter 7.1 - 7.5