8.1 Intermediate Languages

- **Three-address code**: sequence of statements of form
  \[ x := y \text{ op } z \]

  where
  - \( x, y, z \) are names, constants, or compiler-generated temporaries (\( t_1, t_2, \text{ etc.} \)) - ignore register/memory distinction
  - \( \text{op} \) is any operator (arithmetic: +, *, logical and, or)

  - No buildup allowed:
    \[
    \begin{align*}
    w &:= x + y \ast z \\
    t_1 &:= y \ast z \\
    t_2 &:= x + t_1 \\
    w &:= t_2
    \end{align*}
    \]

Intermediate Code Generation

- Theoretically possible to translate directly from source language (ML) to target language (MIPS)
- But machine-independent intermediate representation is better because
  - It makes retargeting easier: single front-end with a back-end for each new target architecture
  - We can do machine-independent optimizations on intermediate code

Three-Address Code

- Like assembly language:
  - Can have symbolic labels, representing statement's address = index in an array of statements
  - Labels are translated to indices as in an assembler
  - Support for flow of control (conditionals, jumps)
### Types of Three-Address Code

- **Dyadic-op assignment:** \( x := y \ op \ z \) (where \( \text{op} = +, \and \))
- **Copy:** \( x := y \)
- **Monadic-op assignment:** \( x := \text{op} \ z \) (\( \text{op} = -, \not= \))
- **Unconditional jump:** \( \text{goto L} \)
- **Conditional jump:** if \( x \ rel \ op \ y \) \( \text{goto L} \) (\( \text{op} = <, \geq \))
- **Parameter-passing procedure calls:**
  - `call` must be told value of \( n \) to support nested procedure calls:
  - `param x1`  
  - `param x2`
  - `\ldots`
  - `param xn`
  - `call p, n`

### Syntax-Directed Translation into Three-Address Code

- Compiler makes up temporary names (\( t1, t2, \ldots \)) for interior nodes of syntax tree.
- Add some attributes to nodes to support code generation
  - `place` : variable name (\( x, y, t1, \ldots \))
  - `code` : generated code
- Add semantic rules (actions) to yacc (CUP) grammar rules:

### Compiling Loops

```c
i := n;
while i do i := i - 1 // non-zero means true
```

```c
i := n
L1: if i = 0 goto L2
    tl = i - 1
    i := tl
    goto L1
L2:
```
Compiling Loops

$S \rightarrow \textbf{while} \ E \ 	extbf{do} \ S \ 	extbf{end}$

$S.begin := \text{newlabel};$
$S.after := \text{newlabel};$
$S.code := \text{gen}(S.begin ':') \parallel$
$E.code$
$\text{gen}(\text{if } E.place = 0 \text{ goto } S.after) \parallel$
$S.code \parallel$
$\text{gen}(\text{goto } S.begin) \parallel$
$\text{gen}(S.after ':')$

Implementations of Three-Address Statements

- Abstractly,
  - Each three-address statement is a record (object),
    with fields (instance vars) for operators and operands
  - Intermediate code program is an array of these records
- E.g., $a := b \times -c + b \times -c$

<table>
<thead>
<tr>
<th>op</th>
<th>arg 1</th>
<th>arg 2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>minus</td>
<td>c</td>
<td>t1</td>
</tr>
<tr>
<td>(1)</td>
<td>$*$</td>
<td>b</td>
<td>t2</td>
</tr>
<tr>
<td>(2)</td>
<td>minus</td>
<td>c</td>
<td>t3</td>
</tr>
<tr>
<td>(3)</td>
<td>$*$</td>
<td>b</td>
<td>t4</td>
</tr>
<tr>
<td>(4)</td>
<td>$+$</td>
<td>t2</td>
<td>t5</td>
</tr>
<tr>
<td>(5)</td>
<td>:=</td>
<td>t5</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>arg 1</th>
<th>arg 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>minus</td>
<td>c</td>
</tr>
<tr>
<td>(1)</td>
<td>$*$</td>
<td>b</td>
</tr>
<tr>
<td>(2)</td>
<td>minus</td>
<td>c</td>
</tr>
<tr>
<td>(3)</td>
<td>$*$</td>
<td>b</td>
</tr>
<tr>
<td>(4)</td>
<td>$+$</td>
<td>(1)</td>
</tr>
<tr>
<td>(5)</td>
<td>:=</td>
<td>a</td>
</tr>
</tbody>
</table>
Implementations of Three-Address Statements

- **Indirect Triple** implementation:
  - Array of pointers to triples, rather than array of triples themselves

<table>
<thead>
<tr>
<th>Statement</th>
<th>Instruction</th>
<th>addr</th>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Often want to move statements around when we optimize (Chapter 10)
- Quadruple implementation supports movement, because results are stored in symbol table (temporaries)
- Triple implementation does not support movement, because results are stored as absolute addresses (statement #s)
- Indirect triple implementation supports movement, because moving statements doesn’t move location of results
- So use quadruple or indirect triple implementation in your compiler (basically, just linked list with pointers / objects)