Intermediate Representations

- A variety of intermediate representations are used in compilers

- Most common intermediate representations are:
  - Abstract Syntax Tree
  - Directed Acyclic Graph (DAG)
  - Three-Address Code
  - Code for a simplified stack-based virtual machine (Example: P-code)

- Abstract Syntax is adequate representation of the source code
  - However, it is not linear and does not resemble target code
  - High-level constructs, such as `if` and `while`, should be translated into jumps

- Directed acyclic graph is an optimization of a syntax tree
Three-Address Code

- Generalized assembly code for a virtual 3-address machine
- 3-address code represents a linearization of the syntax tree
- 3-address code can be:
  * High level: representing all operations as abstractly as a syntax tree
  * Low level: closely resembling target code
- Basic 3-address instruction consists of an operator and 3 addresses
  * Two addresses for the two operands and one address for the result
  * General form $x := y \text{ op } z$
  * $op$ is an operator code
  * $x$, $y$, and $z$ are typically implemented as pointers to symbols
  * $x$ is either an identifier or a temporary symbol
  * $y$ and $z$ can be an identifier, a literal, or a temporary symbol
Types of 3-Address Instructions

- Arithmetic, logical, and shift instructions are of the form:
  \[ x := y \text{ op } z \]
  
  *Binary Operator*

  \( op \) can be any of the following operators:
  
  - **Binary Arithmetic**
    - \text{ADD, SUB, MUL, DIV, MOD,}
    - \text{AND, OR, XOR, SHL, SHR, SHRA,}
    - \text{EQ, NE, LT, LE, GT, GE}
  
  The above operators are **generic**

  Type information can be also added to each operator
  - To distinguish between integer and floating-point operations

- Unary \( op \) instructions are of the form:
  \[ x := \text{ op } y \]
  
  *Unary Operator*

  \( op \) can be **PLUS, MINUS, or NOT** operator

  \( op \) can also be conversion operators to convert between integer and FP
Example on Translating an Expression

- Consider the translation of \((2 + a \times (b - c / d)) / e\)
  
  \[ \begin{align*}
  t1 & := c / d \\
  t2 & := b - t1 \\
  t3 & := a \times t2 \\
  t4 & := 2 + t3 \\
  t5 & := t4 / e 
  \end{align*} \]

- Compiler **generates temporaries** when translating into 3AC
- \(t1, t2, t3, t4,\) and \(t5\) are generated temporaries
- Temporaries are **identified by number**
- Temporaries are stored in symbols, like identifiers
- Type information is also added to temporary symbols
3-Address Instructions for Copy and Jumps

- Move or copy instruction is of the form:
  \[ x := y \]

- Unconditional jump and label instructions are of the form:
  \[ \text{GOTO Ln} \]
  \[ \text{LABEL Ln} \]
  \[ \text{Ln is a label identified by a number n} \]
  \[ \text{A label is not an instruction; It is the address of the following instruction} \]

- Conditional branch instructions are of the form:
  \[ \text{IF x goto Ln} \]
  \[ \text{Branch if x is true} \]
  \[ \text{IFNOT x goto Ln} \]
  \[ \text{Branch if x is false} \]
  \[ x \text{ is a Boolean variable that evaluate to true or false} \]

- The general conditional branch instruction is of the form:
  \[ \text{if x relop y goto Ln} \]
  \[ \text{Conditional branches are used to implement conditional and loop statements} \]
Example on Translating a While Loop

- Consider the following while loop:

  ```
  sum := 0; i := 1;
  while (i<n) {
    sum := sum + i;
    i := i+1;
  }
  ```

- A translation of the above loop into 3AC is shown below:

  ```
  sum := 0
  i := 1
  goto L2
  label L1
  sum := sum + i
  i := i + 1
  label L2
  if i < n goto L1
  ```
Implementation of 3-Address Code

- A 3-address instruction is implemented as a **quadruple**:
  - An **operator code**
  - Two pointers to **operand symbols**
  - A pointer to **result symbol, goto target, or label number**

- A code sequence can be implemented as an array or a linked list

- A linked list is preferable because it …
  - Facilitates reordering and concatenation of instructions
  - Grows dynamically as required

- A 3-address instruction has a **link** to the next instruction

<table>
<thead>
<tr>
<th>opcode</th>
<th>result target label</th>
<th>first</th>
<th>second</th>
<th>link</th>
</tr>
</thead>
</table>

Intermediate Code Generation – 7

Compiler Design – © Muhammed Mudawwar
Three-Address Instruction Structure

The structure of a 3-address instruction is given below:

```c
struct Inst {
    Inst(OpType op, Symbol* r=0, Symbol* f=0, Symbol* s=0);
    Inst(OpType op, Inst* t, Symbol* f=0, Symbol* s=0);
    Inst(OpType op, unsigned l);
    OpType opcode;       // Operation Code
    union {
        Symbol* result;  // Either a symbol pointer, or
        Inst*  target;   // Target label used with GOTO
        unsigned label;  // Label number used with LABEL
    };
    Symbol* first;       // First operand symbol
    Symbol* second;      // Second operand symbol
    Inst*   link;        // Link to next instruction
};
```
Generating Temporaries and Labels

- `newtemp()` allocates and returns a new temporary symbol
- The static variable `num` ensures a unique number for every call
  ```c
  Symbol* newtemp() {
    static int num = 1;
    Symbol* temp = new Symbol(TEMP,num);
    num++;  
    return temp;
  }
  ```
- `newlabel()` allocates and returns a new label instruction
  ```c
  Inst* newlabel() {
    static int num = 1;
    Inst* label = new Inst(LABEL,num);
    num++; 
    return label;
  }
  ```
Concatenating Instructions and Code

- A code sequence is a linear linked list of instructions
- We identify the **first** and **last** instructions in a code sequence

```c
struct Code { // Code sequence structure
    Inst* first; // First instruction in code sequence
    Inst* last;  // Last instruction in code sequence
};
```

- The + operator is overloaded to mean concatenation
  - Four + operators will concatenate code sequences and instructions
  - The result of concatenation is a code sequence

```c
Code operator+(Code a, Code b);
Code operator+(Code c, Inst* i);
Code operator+(Inst* i, Code c);
Code operator+(Inst* i, Inst* j);
```
Concatenating Two Code Sequences

- The + operator links the pointers of two code sequences:

```c
Code operator+(Code a, Code b) {
    Code c;
    if (a.first == 0) { // Code a is NULL
        c = b;
    }
    else if (b.first == 0) { // Code b is NULL
        c = a;
    }
    else { // General Case
        c.first = a.first;
        c.last = b.last;
        a.last->link = b.first;
    }
    return c;
}
```
## Translating Expressions into 3-Address Code

### Synthesized Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E.code$</td>
<td>Code sequence evaluating $E$</td>
</tr>
<tr>
<td>$E.sym$</td>
<td>Symbol representing value of $E$</td>
</tr>
<tr>
<td>$addop.op$</td>
<td>Addition operator: ADD or SUB</td>
</tr>
<tr>
<td>$mulop.op$</td>
<td>Multiplication operator: MUL, DIV, or MOD</td>
</tr>
</tbody>
</table>

### Grammar Rules

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Semantic Rule</th>
</tr>
</thead>
</table>
| $E \rightarrow E^1 \text{ addop } E^2$ | $E.sym := \text{newtemp}()$;  
AddInst := new Inst(addop.op, $E.sym$, $E^1.sym$, $E^2.sym$);  
$E.code := E^1.code + E^2.code + \text{AddInst}$; |
| $E \rightarrow E^1 \text{ mulop } E^2$ | $E.sym := \text{newtemp}()$;  
MulInst := new Inst(mulop.op, $E.sym$, $E^1.sym$, $E^2.sym$);  
$E.code := E^1.code + E^2.code + \text{MulInst}$; |
| $E \rightarrow \text{UnaryOp } E^1$ | $E.sym := \text{newtemp}()$;  
UnaryInst := new Inst(UnaryOp.op, $E.sym$, $E^1.sym$);  
$E.code := E^1.code + \text{UnaryInst}$; |
Synthesized Attributes

- **UnaryOp.op**: Unary operator: PLUS, MINUS, NOT
- **id.name**: Identifier name
- **num.sym**: Literal symbol holding number value

Grammar Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \to ( \ E^1 \ )$</td>
<td>$E.sym := E^1.sym$; $E.code := E^1.code$;</td>
</tr>
<tr>
<td>$E \to id$</td>
<td>$E.sym := idTable.lookup(id.name);$ $E.code.first := E.code.last = 0;$</td>
</tr>
<tr>
<td>$E \to num$</td>
<td>$E.sym := num.sym$; $E.code.first := E.code.last = 0;$</td>
</tr>
<tr>
<td>$UnaryOp \to addop$</td>
<td>if addop.op = ADD then $UnaryOp.op := PLUS$ else $UnaryOp.op := MINUS$</td>
</tr>
<tr>
<td>$UnaryOp \to not$</td>
<td>$UnaryOp.op := NOT$</td>
</tr>
</tbody>
</table>
Translating If-Statement into 3-Address Code

$S \rightarrow \text{if } E \text{ then } S\text{list}^1 \text{ else } S\text{list}^2 \text{ end ;}$

$S\text{.code}$

$E\text{.code}$

$\text{IFNOT }E\text{.sym goto Else}$

$S\text{list}^1\text{.code}$

$\text{GOTO Next}$

$\text{LABEL Else}$

$S\text{list}^2\text{.code}$

$\text{LABEL Next}$

$S \rightarrow \text{if } E \text{ then } S\text{list}^1 \text{ else } S\text{list}^2 \text{ end ;}$

$S\text{.code}$

$E\text{.code}$

$\text{IFNOT }E\text{.sym goto Else}$

$S\text{list}^1\text{.code}$

$\text{GOTO Next}$

$\text{LABEL Else}$

$S\text{list}^2\text{.code}$

$\text{LABEL Next}$
Translating If-Statement – cont'd

Synthesized Attributes

- \( E.code: \) Code sequence for \( E \)
- \( E.sym: \) Symbol representing \( E \)
- \( S.code: \) Code sequence of a statement
- \( Slist.code: \) Code sequence of a statement list

Grammar Rules

\[
S \rightarrow \text{if } E \text{ then } Slist \text{ end } ; \\
\]

Semantic Rules

\[
Next := \text{newlabel}() ; \\
\text{IfNotNext} := \text{new} \text{Inst(IFNOT, Next, E.sym)} ; \\
S.code := E.code + \text{IfNotNext} + Slist.code + \text{Next} ; \\
\]

\[
S \rightarrow \text{if } E \text{ then } Slist^1 \text{ else } Slist^2 \text{ end } ; \\
\]

\[
\text{Else} := \text{newlabel}() ; \\
Next := \text{newlabel}() ; \\
\text{IfNotElse} := \text{new} \text{Inst(IFNOT, Else, E.sym)} ; \\
\text{GotoNext} := \text{new} \text{Inst(GOTO, Next)} ; \\
S.code := E.code + \text{IfNotElse} + Slist^1.code + \text{GotoNext} + \text{Else} + Slist^2.code + \text{Next} ; \\
\]
Translating While Statement

Possible translation

\[ S \rightarrow \text{while } E \text{ do } S\text{list} \text{ end} ; \]

\[ S.\text{code} \]

\[ \text{LABEL } Expr \]

\[ E.\text{code} \]

\[ \text{IFNOT } E.\text{sym} \text{ goto Next} \]

\[ S\text{List.code} \]

\[ \text{GOTO } Expr \]

\[ \text{LABEL } Next \]
Translating While and Statement Lists

Synthesized Attributes

\[\begin{align*}
E.\text{code} &: \text{Code sequence evaluating } E \\
S.\text{code} &: \text{Code sequence of statement } S \\
Slist.\text{code} &: \text{Code sequence of statement list } Slist \\
E.\text{sym} &: \text{Symbol representing value of } E
\end{align*}\]

Grammar Rules

\[S \rightarrow \text{while } E \text{ do } Slist \text{ end ;} \quad \text{next} := \text{newlabel()}; \]

\[\begin{align*}
\text{Expr} &: \text{newlabel()}; \\
\text{GotoExpr} &: \text{new } \text{Inst(} \text{GOTO}, \text{Expr}); \\
\text{IfNotNext} &: \text{new } \text{Inst(} \text{IFNot}, \text{next}, \text{E.sym}); \\
S.\text{code} &: \text{Expr} + E.\text{code} + \text{IfNotNext} + Slist.\text{code} + \\
&\quad \text{GotoExpr} + \text{next};
\end{align*}\]

Semantic Rules

\[\begin{align*}
\text{Slist} \rightarrow \text{Slist}^1 \ S &\quad \text{Slist.}\text{code} := \text{Slist}^1.\text{code} + S.\text{code}; \\
\text{Slist} \rightarrow \varepsilon &\quad \text{Slist.}\text{code}.\text{first} := 0; \\
&\quad \text{Slist.}\text{code}.\text{last} := 0;
\end{align*}\]