Syntax Directed Translation

**AST generation:**

\[
S \rightarrow \text{id} := E \quad \{ S.nptr := \text{mknode}(\text{:=}, \text{id}, \text{idPLACE}, E.nptr) \}
\]

\[
E \rightarrow E_1 + E_2 \quad \{ E.nptr := \text{mknode}(\text{+}, E_1.nptr, E_2.nptr) \}
\]

\[
E \rightarrow E_1 * E_2 \quad \{ E.nptr := \text{mknode}(\text{*}, E_1.nptr, E_2.nptr) \}
\]

\[
E \rightarrow - E_1 \quad \{ E.nptr := \text{mknode}(\text{-}, E_1.nptr) \}
\]

\[
E \rightarrow ( E_1 ) \quad \{ E.nptr := E_1.nptr \}
\]

\[
E \rightarrow \text{id} \quad \{ E.nptr := \text{mkleaf(id, idPLACE)} \}
\]

Intermediate-Code Representations

- **High-level (HIR):** Abstract Syntax Trees.
  - Suitable for some dependence analysis and source code transformations.

- **Medium-level (MIR):** Three address code.
  - Suitable for most optimizations.

- **Low-level (LIR):** Register transfer level (RTL).
  - Suitable for optimizations requiring registers and addressing.

Syntax Directed Translation

**Code Generation**

- **Intermediate code generation:** Abstract (machine independent) code.
- **Code optimization:** Transformations to the code to improve time/space performance.
- **Final code generation:** Emitting machine instructions.

**Interpretation:**

\[
E \rightarrow E_1 + E_2 \quad \{ E.val := E_1.val + E_2.val; \}
\]

**Type Checking:**

\[
E \rightarrow E_1 + E_2 \quad \{
\text{if } E_1.type \equiv E_2.type \equiv \text{int}
\]

\[
E.type = \text{int};
\]

\[
\text{else}
\]

\[
E.type = \text{float};
\]

\}
Types of Three-Address Statements

- Address and pointer assignments:
  - \( x := y \) sets the value of \( x \) to the location of \( y \).
  - \( x := *z \) sets the value of \( x \) to the contents of the location pointed to by \( z \).
  - \( *x := y \) sets the value of the location pointed to by \( x \) to the value of \( y \).

- Jump statements:
  - \( \text{goto } L \) unconditional jump to statement with label \( L \).
  - \( \text{if } x \text{ rel } y \text{ goto } L \) conditional jump.

Types of Three-Address Statements

- Procedure call statements:
  - \( \text{param } x \) push actual parameter \( x \).
  - \( \text{call } p, n \) call procedure \( p \) with \( n \) arguments.
  - \( \text{receive } x \) receive parameter value in \( x \).
  - \( \text{return } y \) return with optional value in \( y \).

- Example:
  - A procedure call \( p(x_1, x_2, \ldots, x_n) \) results in:
    - \( \text{param } x_1 \)
    - \( \text{param } x_2 \)
    - \( \cdots \)
    - \( \text{param } x_n \)
    - \( \text{call } p, n \)

Three-Address Code

- General form: \( x := y \text{ op } z \)
  - \( x, y \) and \( z \) are names, compiler generated temporaries or constants.
  - \( \text{op} \) stands for any arithmetic or logical operator.

- Example: The source language expression \( x \text{ + } y \text{ * } z \) might be translated to:
  - \( t_1 := y \text{ * } z \)
  - \( t_2 := x + t_1 \)

- Intuition: Three-address code is a linearized representation of
  - an AST or DAG:
  - explicit names correspond to interior nodes.

Types of Three-Address Statements

- Simple assignments:
  - \( x := y \text{ op } z \) with binary arithmetic or logical \( \text{op} \).
  - \( x := \text{ op } z \) with unary arithmetic or logical \( \text{op} \).
  - \( x := y \) where the value of \( y \) is assigned to \( x \).

- Indexed assignments:
  - \( x := y[i] \) sets the contents of \( x \) to the value in the location \( i \) units beyond location \( y \).
  - \( x[i] := y \) sets the contents of the location \( i \) units beyond location \( x \) to the value of \( y \).
### Intermediate Forms

Choice depends on convenience of further processing
- Stack code is simplest to generate for expressions,
- Quadruples are most general, permitting most optimizations including code motion,
- Triples permit optimizations such as common subexpression elimination, but code motion is difficult.

### Quadruples

Explicit representation (implementation) of three address code, 
*Records with fields op, arg1, arg2 and result.*

Example: \( a := a + b \times c; \)

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>c</td>
<td>( t_1 )</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>b</td>
<td>( t_1 )</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>a</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>3</td>
<td>:=</td>
<td>( t_3 )</td>
<td>a</td>
</tr>
</tbody>
</table>

### Triples

Representation of three address code with implicit destination argument.

Example: \( a := a + b \times c; \)

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>c</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>:=</td>
<td>a</td>
</tr>
</tbody>
</table>