Code Generation
Modern Compiler Project
Different IRs for Different Stages

- Modern compilers use multiple IRs at different stages of code generation

- High-Level IR
  - Examples: Abstract Syntax Tree, Parse Tree
  - Language dependent (a high-level IR for each language)
  - Purpose: Semantic analysis of program

- Low-Level IR
  - Examples: Three address code, Static Single Assignment
  - Essentially an instruction set for an abstract machine
  - Language and machine independent (one common IR)
  - Purpose: Compiler optimizations to make code efficient
    - All optimizations written in this IR is automatically applicable to all languages and machines
Different IRs for Different Stages

- **Machine-Level IR**
  - Examples: x86 IR, ARM IR, MIPS IR
  - Actual instructions for a concrete machine ISA
  - Machine dependent (a machine-level IR for each ISA)
  - Purpose: Code generation / CPU register allocation
    - (Optional) Machine-level optimizations
      - (e.g. strength reduction: $x / 2 \rightarrow x \gg 1$)

- Possible to have one IR (AST) — some compilers do
  - Generate machine code from AST after semantic analysis
  - Makes sense if compilation time is the primary concern
    - (e.g. JIT)

- So why have multiple IRs?
Why Multiple IRs?

- Why multiple IRs?
  - Better to have an appropriate IR for the task at hand
    - Semantic analysis much easier with AST
    - Compiler optimizations much easier with low-level IR
    - Register allocation only possible with machine-level IR
  - Easier to add a new front-end (language) or back-end (ISA)
    - Front-end: a new AST → low-level-IR converter
    - Back-end: a new low-level IR → machine IR converter
    - Low-level IR acts as a bridge between multiple front-ends and back-ends, such that they can be reused
Why Multiple IRs?

Why multiple IRs?

- Better to have an appropriate IR for the task at hand
  - Semantic analysis much easier with AST
  - Compiler optimizations much easier with low-level IR
  - Register allocation only possible with machine-level IR

- Easier to add a new front-end (language) or back-end (ISA)
  - Front-end: a new AST → low-level-IR converter
  - Back-end: a new low-level IR → machine IR converter
  - Low-level IR acts as a bridge between multiple front-ends and back-ends, such that they can be reused

If one IR (AST), and adding a new front-end ...

- Reimplement all compiler optimizations for new AST
- A new AST → machine code converter for each ISA
- Same goes for adding a new back-end
Three Address Code

Generic form is \( X = Y \text{ op } Z \)
where \( X, Y, Z \) can be variables, constants, or compiler-generated temporaries holding intermediate values

- Characteristics
  - Assembly code for an 'abstract machine'
  - Long expressions are converted to multiple instructions
  - Control flow statements are converted to jumps
  - Machine independent
    - Operations are generic (not tailored to specific machine)
    - Function calls represented as generic call nodes
    - Uses **symbolic names** rather than **register names**
      (Actual locations of symbols are yet to be determined)

- Design goal: for easier machine-independent optimization
An example:

\[ x \times y + x \times y \]

is translated to

\[ t1 = x \times y \quad ; \quad t1, t2, t3 \text{ are temporary variables} \]
\[ t2 = x \times y \]
\[ t3 = t1 + t2 \]

- Can be generated through a depth-first traversal of AST
- Internal nodes in AST are translated to temporary variables

Notice: repetition of \( x \times y \)

- Can be later eliminated through a compiler optimization called common subexpression elimination (CSE):
  \[ t1 = x \times y \]
  \[ t3 = t1 + t1 \]

- Using 3-address code rather than AST makes it:
  - Easier to spot opportunities (just find matching RHSs)
  - Easier to manipulate IR (AST is much more cumbersome)
Common Three-Address Statements (I)

- Assignment statement:
  \[ x = y \text{ op } z \]
  where \( \text{op} \) is an arithmetic or logical operation (binary operation)

- Assignment statement:
  \[ x = \text{ op } y \]
  where \( \text{op} \) is an unary operation such as -, not, shift

- Copy statement:
  \[ x = y \]

- Unconditional jump statement:
  \[ \text{goto } L \]
  where \( L \) is label
Common Three-Address Statements (II)

- Conditional jump statement:
  ```c
  if (x relop y) goto L
  ```
  where `relop` is a relational operator such as `=, \neq, >, <`

- Procedural call statement:
  ```c
  param x_1, ..., param x_n, call F, y, n
  ```
  As an example, `foo(x1, x2, x3)` is translated to
  ```c
  param x_1
  param x_2
  param x_3
  call foo, 3
  ```

- Procedural call return statement:
  ```c
  return y
  ```
  where `y` is the return value (if applicable)
Indexed assignment statement:

\[
x = y[i]
\]
or

\[
y[i] = x
\]

where \( x \) is a scalar variable and \( y \) is an array variable.

Address and pointer operation statement:

\[
x = \& y \quad ; \text{a pointer} \ x \ \text{is set to location of} \ y
\]
\[
y = * x \quad ; \text{y is set to the content of the address stored in} \ x
\]
\[
* y = x \quad ; \text{object pointed to by} \ y \ \text{gets value} \ x
\]
There are three possible ways to store the code:
- quadruples
- triples
- indirect triples

Using quadruples:

\textbf{op arg1, arg2, result}

- There are four(4) fields at maximum
- Arg1 and arg2 are optional
- Arg1, arg2, and result are usually pointers to the symbol table

Examples:

\begin{align*}
x &= a + b & \Rightarrow & + a, b, x \\
x &= -y & \Rightarrow & - y, , x \\
goto L & \Rightarrow \text{goto} , , L
\end{align*}
Using Triples

- Triple: Quadruple without the result field
- Can refer to results by the positions of instructions that compute them, instead of through temporaries

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

<table>
<thead>
<tr>
<th>Quadruples</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>arg1</td>
</tr>
<tr>
<td>(0)</td>
<td>-</td>
</tr>
<tr>
<td>(1)</td>
<td>*</td>
</tr>
<tr>
<td>(2)</td>
<td>-</td>
</tr>
<tr>
<td>(3)</td>
<td>*</td>
</tr>
<tr>
<td>(4)</td>
<td>+</td>
</tr>
<tr>
<td>(5)</td>
<td>=</td>
</tr>
</tbody>
</table>
More About Triples

- If assigned location is also the result of an expression?
  - Array location (e.g. $x[i] = y$)
  - Pointer location (e.g. $*(x+i) = y$)
  - Struct field location (e.g. $x.i = y$)

- Example: triples for array assignment statement
  
  $x[i] = y$

  is translated to

  $\begin{align*}
  (0) & \ [] x \ i \\
  (1) & = (0) y
  \end{align*}$

  - One or more triples are used to compute the location
Problem with triples

- Compiler optimizations often involve moving instructions
- Hard to move instructions because numbering will change, even for instructions not involved in optimization

<table>
<thead>
<tr>
<th>Quadruples</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>arg1</td>
</tr>
<tr>
<td>(0)</td>
<td>-</td>
</tr>
<tr>
<td>(1)</td>
<td>*</td>
</tr>
<tr>
<td>(2)</td>
<td>-</td>
</tr>
<tr>
<td>(3)</td>
<td>*</td>
</tr>
<tr>
<td>(4)</td>
<td>+</td>
</tr>
<tr>
<td>(5)</td>
<td>=</td>
</tr>
</tbody>
</table>
Using Indirect Triples

Problem with triples

- Compiler optimizations often involve moving instructions
- Hard to move instructions because numbering will change, even for instructions not involved in optimization

<table>
<thead>
<tr>
<th>Quadruples</th>
<th>Tripes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>op</strong></td>
<td><strong>arg1</strong></td>
</tr>
<tr>
<td>(0)</td>
<td>-</td>
</tr>
<tr>
<td>(1)</td>
<td>*</td>
</tr>
<tr>
<td>(2)</td>
<td>+</td>
</tr>
<tr>
<td>(3)</td>
<td>=</td>
</tr>
</tbody>
</table>

Quadruples use 4 operands while Triples use 3 operands.
Using Indirect Triples

- Triples are stored in a triple 'database'
- IR is a listing of pointers to triples in database
- Can reorder listing without changing numbering in database
- Indirection overhead but allows easy code motion

<table>
<thead>
<tr>
<th>Indirect Triples</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ptr to triple database)</td>
<td>op</td>
</tr>
<tr>
<td>(0)</td>
<td>-</td>
</tr>
<tr>
<td>(1)</td>
<td>*</td>
</tr>
<tr>
<td>(2)</td>
<td>-</td>
</tr>
<tr>
<td>(3)</td>
<td>*</td>
</tr>
<tr>
<td>(4)</td>
<td>+</td>
</tr>
<tr>
<td>(5)</td>
<td>=</td>
</tr>
</tbody>
</table>
After Optimization

<table>
<thead>
<tr>
<th>Indirect Triples</th>
<th>Triple Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ptr to triple database)</td>
<td>op</td>
</tr>
<tr>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(2)</td>
<td>(4)</td>
</tr>
<tr>
<td>(3)</td>
<td>(5)</td>
</tr>
<tr>
<td>(4)</td>
<td>(1)</td>
</tr>
<tr>
<td>(5)</td>
<td>(4)</td>
</tr>
</tbody>
</table>

- After optimization, some entries in database can be reused.
  - i.e. Entries in triple database do not have to be contiguous.
After Optimization

- After optimization, some entries in database can be reused.
  - i.e. Entries in triple database do not have to be contiguous.
Static Single Assignment (SSA)

- Every variable is assigned to exactly once statically
  - Convert original variable name to name_{version}
    - e.g. \( x \rightarrow x_1, x_2 \) for each distinct assignment of \( x \)
  - Same version is guaranteed to contain same value
  - On a control flow merge, use \( \phi \)-function to combine two versions of same variable

\[
\begin{align*}
  x &= a + b; \\
  y &= x - c; \\
  x &= x - y; \\
  \text{if ( ...)} \\
\end{align*}
\]
\[
\begin{align*}
  x_1 &= a + b; \\
  y_1 &= x_1 - c; \\
  x_2 &= x_1 - y_1; \\
  \text{if ( ...)} \\
\end{align*}
\]
\[
\begin{align*}
  x &= x + 5; \\
  y &= x \times 4; \\
\end{align*}
\]
\[
\begin{align*}
  x_3 &= x_2 + 5; \\
  x_4 &= x_2 \times 4; \\
\end{align*}
\]
\[
\begin{align*}
  x_5 &= \phi(x_3, x_4); \\
  y_2 &= x_5 \times 4; \\
\end{align*}
\]
Benefits of SSA

SSA can assist compiler optimizations

- Previously, easier to spot that two instances of \( x \times 4 \) do not compute the same value, hence CSE cannot be applied
- Easier to do other optimizations such as dead code elimination (DCE)

\[
\begin{align*}
x &= a + b; \\
x &= c - d; \\
y &= x \times b;
\end{align*}
\]

\[
\begin{align*}
x_1 &= a + b; \\
x_2 &= c - d; \\
y_1 &= x_2 \times b;
\end{align*}
\]

.... \( x_1 \) is defined but never used, it is safe to remove

- Will discuss more in compiler optimization phase
- Intuition: Makes data dependency relationships between instructions more apparent in the IR
Generating IR using Syntax Directed Translation
Our next task is to translate **language constructs** to IR using **syntax directed translation scheme**

- What language structures do we need to translate?
  - Declarations
    - variables, procedures (need to enforce static scoping), ...
  - Assignment statement
  - Flow of control statement
    - if-then-else, while-do, for-loop, ...
  - Procedure call
  - ...

Generating IR
Attributes to Evaluate in Translation

- **Statement S**
  - `S.code` — a synthesized attribute that holds IR code of S

- **Expression E**
  - `E.code` — a synthesized attribute that holds IR code for computing E
  - `E.place` — a synthesized attribute that holds the location where the result of computing E is stored

- **Variable declaration:**
  - `T V` e.g. int a,b,c;
  - Type information `T.type T.width`
  - Variable information `V.type, V.offset`
Attributes to Evaluate in Translation

- **Statement S**
  - S.code — a synthesized attribute that holds IR code of S

- **Expression E**
  - E.code — a synthesized attribute that holds IR code for computing E
  - E.place — a synthesized attribute that holds the location where the result of computing E is stored

- **Variable declaration:**
  - T V e.g. int a,b,c;
  - Type information T.type T.width
  - Variable information V.type, V.offset

..... What is V.offset?
When there are multiple variables defined in a procedure, we layout the variable sequentially use variable **offset**, to get address of \( x \)

- \( \text{address}(x) \leftarrow \text{offset} \)
- \( \text{offset} += \text{sizeof}(x.\text{type}) \)

```c
void foo() {
    int a;
    int b;
    long long c;
    int d;
}
```
When there are multiple variables defined in a procedure, we layout the variable sequentially.

- Use variable `offset`, to get address of `x`.
  - `address(x) ← offset`
  - `offset += sizeof(x.type)`

```c
void foo() {
    int a;
    int b;
    long long c;
    int d;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
</tr>
<tr>
<td>0x000c</td>
<td></td>
</tr>
<tr>
<td>0x0010</td>
<td></td>
</tr>
</tbody>
</table>
When there are multiple variables defined in a procedure,

- we layout the variable sequentially
- use variable **offset**, to get address of **x**
  - `address(x) ← offset`
  - `offset += sizeof(x.type)`

```c
void foo() {
    int a;
    int b;
    long long c;
    int d;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Addr(a)←0</td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
</tr>
<tr>
<td>0x000c</td>
<td></td>
</tr>
<tr>
<td>0x0010</td>
<td></td>
</tr>
</tbody>
</table>
When there are multiple variables defined in a procedure, we layout the variable sequentially. Use variable `offset`, to get address of `x`:

- `address(x) ← offset`
- `offset += sizeof(x.type)`

```c
void foo() {
    int a;
    int b;
    long long c;
    int d;
}
```

Address:

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Addr(a)←0</td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
</tr>
<tr>
<td>0x000c</td>
<td></td>
</tr>
<tr>
<td>0x0010</td>
<td></td>
</tr>
</tbody>
</table>
When there are multiple variables defined in a procedure,

- we layout the variable sequentially
- use variable **offset**, to get address of \( x \)
  - \( \text{address}(x) \leftarrow \text{offset} \)
  - \( \text{offset} += \text{sizeof}(x.\text{type}) \)

```c
void foo() {
    int a;
    int b;
    long long c;
    int d;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Addr(a)←0</td>
</tr>
<tr>
<td>0x0004</td>
<td>Addr(b)←4</td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
</tr>
<tr>
<td>0x000c</td>
<td></td>
</tr>
<tr>
<td>0x0010</td>
<td></td>
</tr>
</tbody>
</table>
When there are multiple variables defined in a procedure,

- we layout the variable sequentially
- use variable **offset**, to get address of \( x \)
  - \( \text{address}(x) \leftarrow \text{offset} \)
  - \( \text{offset} += \text{sizeof}(x.\text{type}) \)

```c
void foo() {
    int a;
    int b;
    long long c;
    int d;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Addr(a)</th>
<th>Addr(b)</th>
<th>Addr(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>16</td>
<td>0</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x000c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When there are multiple variables defined in a procedure,

- we layout the variable sequentially
- use variable `offset`, to get address of `x`
  - address(x) ← offset
  - offset += sizeof(x.type)

```c
void foo() {
    int a;
    int b;
    long long c;
    int d;
}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
<th>Offset</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0x0000</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>0x0004</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>0x0008</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>0x000c</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
```
Allocation alignment

- Enforce \( \text{addr}(x) \mod \text{sizeof}(x.\text{type}) = 0 \)
- Most machine architectures are designed such that computation is most efficient at \text{sizeof}(x.\text{type})\ boundaries
  - E.g. Most machines are designed to load integer values at integer word boundaries
  - If not on word boundary, need to load two words and shift & concatenate

```c
void foo() {
    char a;       // addr(a) = 0;
    int b;        // addr(b) = 4; /* instead of 1 */
    int c;        // addr(c) = 8;
    long long d;  // addr(d) = 16; /* instead of 12 */
}
```
Endianness

- Big endian stores **MSB** (most significant byte) in lowest address
- Little endian stores **LSB** (least significant byte) in lowest address
More About Storage Layout (II)

- **Endianness**
  - Big endian stores **MSB** (most significant byte) in lowest address
  - Little endian stores **LSB** (least significant byte) in lowest address

![Endianness Diagram]

- **Big-endian**
  - Memory: 0A 0B 0C 0D ...
  - Register: 31 0A 0B 0C 0D...
  - a: 0A
  - a+1: 0B
  - a+2: 0C
  - a+3: 0D

- **Little-endian**
  - Memory: 0A 0B 0C 0D ...
  - Register: 31 0A 0B 0C 0D...
  - a: 0A
  - a+1: 0B
  - a+2: 0C
  - a+3: 0D
More About Storage Layout (III)

Questions still unanswered

- How are non-local variables laid out?
- How dynamically allocated variables laid out?
Processing Declarations

Translating the declaration in a single procedure

- enter(name, type, offset) — insert the variable into the symbol table

```
P → M D
M → ε { offset=0; } /* reset offset before layout */
D → D ; D
D → T id { enter(id.name, T.type, offset); offset += T.width; }
T → integer { T.type=integer; T.width=4; }
T → real { T.type=real; T.width=8; }
T → T1[num] { T.type=array(num.val, T1.type);
              T.width=num.val * T1.width; }
T → * T1 { T.type=ptr(T1.type); T.width=4; }
```
Processing Nested Declarations

- Need scope information for each level of nesting.
- When encountering a nested procedure declaration...
  1. Create a new symbol table
     - `mktable();` — returns pointer to new table
  2. Suspend processing of outer symbol table
     - Push new table in the **active symbol table stack**
     - Push offset 0 into the **offset stack**
  3. When done, resume processing of outer symbol table
     - Pop inner table in **active symbol table stack**
     - Pop inner procedure offset from **offset stack**
     - Now the outer procedure is at the top of both stacks
  4. Store inner procedure name in outer symbol table
     - `enterproc(outer_table_ptr, proc_name, proc_addr);` — enters symbol for inner procedure in outer symbol table
     - `Proc_addr`: address of code generated for proc
Nested Declaration Example

void P1() {
    int a;
    int b;

    check point #1
}

void P2() {
    int q;
}

void P3() {
    void P4() {
        use a
    }

    int J;
}

use q

Symbol Table Stack

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

P1

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
</tbody>
</table>
Nested Declaration Example

```c
void P1() {
    int a;
    int b;
    check point #1
    void P2() {
        int q;
        check point #2
    }
}

void P3() {
    void P4() {
        use a
    }
    int J;
}

use q
```

Symbol Table Stack

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>a 0</td>
</tr>
<tr>
<td></td>
<td>b 4</td>
</tr>
</tbody>
</table>

Stack

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>q 0</td>
</tr>
</tbody>
</table>

Check points:
- #1
- #2
Nested Declaration Example

void P1() {
    int a;
    int b;
    
    check point #1

    void P2() {
        int q;
        
        check point #2
    }

    void P3() {
        void P4() {
            use a
        }
        int J;
        
        use q
    }
}

Symbol Table Stack Offset Stack

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

check point #3

P1
a 0
b 4
P2
q 0
...
### Nested Declaration Example

```c
void P1() {
    int a;
    int b;
    // check point #1
    void P2() {
        int q;
        // check point #2
    }
}

void P3() {
    int J;
    // use q
}

void P4() {
    // use a
    // check point #3
}
```

#### Symbol Table Stack

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Offset Stack

- 0
- 0
- 8

---

**Check Points:**

1. P1
2. P2
3. P3
4. P4
Nested Declaration Example

```c
void P1() {
    int a;
    int b;
    check point #1
    void P2() {
        int q;
        check point #2
    }
    check point #3
    void P3() {
        void P4() {
            use a
            check point #4
            int J;
        }
        use q
        check point #5
    }
}
```

Symbol Table Stack

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>a 0</td>
</tr>
<tr>
<td></td>
<td>b 4</td>
</tr>
<tr>
<td>P2</td>
<td>q 0</td>
</tr>
<tr>
<td>P3</td>
<td>J 0</td>
</tr>
<tr>
<td>P4</td>
<td></td>
</tr>
</tbody>
</table>

Stack

```plaintext
P1
P2
P3
P4
```
Processing Nested Declarations

Syntax directed translation rules

\[
P \rightarrow M \ D \\
M \rightarrow \varepsilon \\
D \rightarrow D; \ D \\
D \rightarrow \text{void pid()} \{ \ N \ D; \ S \} \\
N \rightarrow \varepsilon \\
D \rightarrow T \ id; \\
\]

\{ pop(active); pop(offset); \}
\{ t=mktable(); push(t, active); push(0,offset); \}
{ pop(active); pop(offset); enterproc(top(active), pid, pid_addr); }
{ t=mktable(); push(t, active); push(0, offset); }
{ enter(top(active), id, T.type, top(offset)); top(offset) = top(offset)+ T.width; }
Processing Statements

Statements are processed sequentially after declarations

- Attributes:
  - **E.place** — name of location to store value of expression E

- Helper functions:
  - **lookup (id)** — search id in symbol table, return nil if none
  - **emit()** — print three address IR
  - **newtemp()** — get a new temporary variable

S → id = E   { P=lookup(id); if (P=nil) perror(...); else emit(P ’=’ E.place); }
E → E1 + E2   { E.place = newtemp(); emit(E.place ’=’ E1.place ’+’ E2.place); }
E → E1 * E2   { E.place = newtemp(); emit(E.place ’=’ E1.place ’*’ E2.place); }
E → - E1     { E.place = newtemp(); emit(E.place ’=’ ’-’ E1.place); }
E → ( E1 )   { E.place = E1.place; }
E → id       { P=lookup(id); E.place=P; }
Arrays can have one or more dimensions:
1-dimension: int x[100]; ..... x[i_1]
2-dimension: int x[100][200]; ..... x[i_1][i_2]
3-dimension: int x[100][200][300]; ..... x[i_1][i_2][i_3]

Calculating offset of a k-dimension array item
(Where N_k = bounds of dimension k)
1-dimension: a_1 = i_1
2-dimension: a_2 = i_1*N_2 + i_2 = a_1*N_2 + i_2
3-dimension: a_3 = i_1*N_2*N_3 + i_2*N_3 + i_3 = a_2*N_3 + i_3
...
k-dimension: a_k = a_{k-1}*N_k + i_k

Calculating address of a k-dimension array item
(Where width = item width, base = array base address)
k-dimension: A_k = a_k * width + base
Processing an array assignment (e.g. A[i] = B[j];)

S → L = E
   { t = newtemp(); emit( t '=' L.place '*' L.width);
     emit(t '=' L.base '+' t); emit ('*'t '=' E.place); }

E → L
   { E.place = newtemp(); t= newtemp();
     emit( t '=' L.place '*' L.width); emit ( E.place '=' (L.base '+' t ) ); }

L → id [ E ]
   { L.base = lookup(id).base; L.width = lookup(id).width;
     L.bounds = lookup(id).bounds; L.dim=1; L.place = E.place; }

L → L1 [ E ]
   { L.base = L1.base; L.width = L1.width; L.dim = L1.dim + 1;
     L.place = newtemp();
     emit( L.place '=' L1.place '*' L.bounds[L.dim]);
     emit( L.place '=' L.place '+' E.place); }
Processing Boolean Expressions

- **Boolean expression**: \( a \text{ op } b \)
  - where \( \text{op} \) can be \(<\), \(>\), \(\geq\), \(\&\&\), \(||\), ...

1. **Languages without short circuiting**
   - **Short circuiting**:
     - In expression \( A \&\& B \), not evaluating \( B \) when \( A \) is false
     - In expression \( A || B \), not evaluating \( B \) when \( A \) is true
     - Can have not only performance but semantic implications
       (e.g. In C++: would \( ++x \) execute in "false && ++x > 0"? )

   - **Computed just like any other arithmetic expression**:
     \[ E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f) \equiv \begin{align*}
     t1 &= a < b \\
     t2 &= c < d \\
     t3 &= e < f \\
     t4 &= t2 \&\& t3 \\
     t5 &= t1 || t4
     \end{align*} \]
2. Languages with short circuiting

- Implemented via a series of jumps:
  \[ E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f) \equiv \]
  \[
  \text{if } (a<b) \text{ goto } E.\text{true} \\
  \text{goto } L1 \\
  \text{L1: if } (c<d) \text{ goto } L2 \\
  \text{goto } E.\text{false} \\
  \text{L2: if } (e<f) \text{ goto } E.\text{true} \\
  \text{goto } E.\text{false}
  \]

- E.true: code to execute on 'true'
- E.false: code to execute on 'false'

- Each relational op converted to two gotos (true and false), chained together
  - Remaining operators skipped when result known in middle

- Applied to all types of control flow statements
  \[ S \rightarrow \text{if } E \text{ then } S1 \mid \text{if } E \text{ then } S1 \text{ else } S2 \mid \text{while } E \text{ do } S1 \]
2. Languages *with short circuiting* (cont’d)

- SDD for if statement:
  
  \[
  E \rightarrow \text{id1 \ relop \ id2} \{
  \text{E.code} = \text{emit(‘if’ id1 ’relop’ id2 ’goto’ E.true) || emit(‘goto’ E.false);}
  \}
  
  E.true: address of code to execute on ’true’ (inherited)
  E.false: address of code to execute on ’false’ (inherited)

- \[
  S \rightarrow \text{if E then S1} \{
  \text{E.true} = \text{S1.label}; \text{E.false} = \text{S.next};
  \text{S.code} = \text{E.code || emit(S1.label’:’)} || \text{S1.code};
  \}
  
  S1.label: label created at the beginning of code S1
  S.next: address of code that comes after S (inherited)

- \[
  S \rightarrow \text{S1; S2} \{
  \text{S1.next} = \text{S2.label}; \text{S2.next} = \text{S.next};
  \text{S.code} = \text{S1.code || emit(S2.label’:’)} || \text{S2.code};
  \}
  
  S2.label: label created at the beginning of code S2

- **Problem: E.true, S1.next are non-L-attributes**
  - **E.true**: Address of S1.label known only when S1 emitted
  - **S1.next**: Address of S2.label known only when S2 emitted
2. Languages with short circuiting (cont’d)

- SDD for && and ||:
  
  $E \rightarrow E_1 \&\& E_2$  
  $\{ E_1.true = E_2.label; E_1.false = E.false; $  
  $E.code = E_1.code \| emit(E_2.label':') \| E_2.code; \}$

  $E \rightarrow E_1 \| E_2$  
  $\{ E_1.false = E_2.label, E_1.true = E.true; $  
  $E.code = E_1.code \| emit(E_2.label':') \| E_2.code; \}$

- Problem: $E_1.true$, $E_1.false$ are non-L-attributes
  - $E_1.true$: Address of $E_2.label$ known only when $E_2$ emitted
  - $E_1.false$: Address of $E_2.label$ known only when $E_2$ emitted

- Do non-L-attributes preclude single pass SDTS?  
  - Both LL and LR SDTS rely on L-attributed grammars
Solutions: two methods

- Two pass approach — process the code twice
  - Generate code with non-address-mapped labels in 1st pass
    (When generated, map labels to addresses in a hashtable)
  - Replace labels with addresses in 2nd pass
    (By now, all labels are mapped to addresses in hashtable)

- One pass approach
  - Generate holes when address is needed but unknown
  - Maintain a list of holes for that address
  - Fill in holes when addresses is known later on
  - Finish code generation in one pass
Two-Pass Based Syntax Directed Translation Scheme

- Attributes for two pass based approach
  - Statement $S \rightarrow \text{if } E \text{ then } S_1$
    - inherited attributes: $E\cdot\text{false}, S_1\cdot\text{next}$
    - non-L inherited attributes: $E\cdot\text{true}$
  - Statement $S \rightarrow S_1 S_2$
    - inherited attributes: $S_2\cdot\text{next}$
    - non-L inherited attributes: $S_1\cdot\text{next}$

- Given rule $S \rightarrow \text{if } E \text{ then } S_1$, the two passes are:
  1. Generate $E\cdot\text{code}$ using unmapped label $E\cdot\text{true}$
     When $S_1\cdot\text{code}$ is generated, map $E\cdot\text{true}$
  2. Replace label $E\cdot\text{true}$ with actual address of $S_1$

- Given rule $S \rightarrow S_1 S_2$, the two passes are:
  1. Generate $S_1\cdot\text{code}$ using unmapped label $S_1\cdot\text{next}$
     When $S_2\cdot\text{code}$ is generated, map $S_1\cdot\text{next}$
  2. Replace label $S_1\cdot\text{next}$ with actual address of $S_2$
Two Pass based Rules

S → if E then S1
{ E.true = newlabel;
  E.false = S.next;
  S1.next = S.next;
  S.code = E.code || emit(E.true':') || S1.code; }

S → if E then S1 else S2
{ S1.next = S2.next = S.next;
  E.true = newlabel;
  E.false = newlabel;
  S.code = E.code || emit(E.true':') ||
    S1.code || emit('goto ' S.next) ||
    emit(E.false ':') || S2.code; }
More Two Pass based SDT Rules

\[ E \rightarrow \text{id1 relop id2} \]
\[
{ E.\text{code}=\text{emit('if' id1.place 'relop' id2.place 'goto' E.true)} ||
\text{emit('goto' E.false)}; }
\]

\[ E \rightarrow \text{E1 or E2} \]
\[
{ E1.true = E2.true = E.true; \\
E1.false = \text{newlabel}; \\
E2.false = E.false; \\
E.\text{code} = E1.\text{code} || \text{emit(E1.false ':'')} || E2.\text{code}; }
\]

\[ E \rightarrow \text{E1 and E2} \]
\[
{ E1.false = E2.false = E.false; \\
E1.true = \text{newlabel}; \\
E2.true = E.true; \\
E.\text{code} = E1.\text{code} || \text{emit(E1.true ':'')} || E2.\text{code}; }
\]

\[ E \rightarrow \text{not E1} \]
\[
{ E1.true = E.false; E1.false = E.true; E.\text{code} = E1.\text{code}; }
\]

\[ E \rightarrow \text{true} \]
\[
{ E.\text{code} = \text{emit('goto' E.true)}; }
\]

\[ E \rightarrow \text{false} \]
\[
{ E.\text{code} = \text{emit('goto' E.false)}; }
\]
Try this at home. Refer to textbook Chapter 6.6.

Write SDT rule (two pass) for the following statement:

\[ S \rightarrow \text{while } E_1 \text{ do} \]
\[ \quad \text{if } E_2 \]
\[ \quad \text{then } S_2 \]
\[ \quad \text{endif} \]
\[ \text{endwhile} \]
If grammar contains L-attributes only, then it can be processed in one pass.

However, **we know** non-L attributes are necessary:

- Example: E1.false in E → E1 || E2
- Is there a general solution to this problem?

**Solution:**

- Leave holes for non-L attributes, record their locations in holelists, and fill in holes when values are known.
  - *holes*: synthesized attribute of 'holes’ to be filled in for a particular target value.
  - Holes are filled in one shot when target value is known.
  - All holes can be filled by the end of code generation (Since by then, all target addresses are known).
One-Pass Based Syntax Directed Translation Scheme

- Attributes for one-pass based approach
  - Expressions $E \rightarrow E_1 \lor E_2, E \rightarrow id1 \text{ relop } id2 \ldots$
    - Synthesized: $E.holes\_true, E.holes\_false$
  - Statements $S \rightarrow \text{if } E \text{ then } S_1, S \rightarrow S_1 S_2 \ldots$
    - Synthesized: $S.holes\_next$

- Given rule $S \rightarrow \text{if } E \text{ then } S_1$, below is done in one-pass:
  - Gen $E.code$, making $E.holes\_true, E.holes\_false$
  - Gen $S_1.code$, filling $E.holes\_true$, making $S_1.holes\_next$
  - Merge $E.holes\_false, S_1.holes\_next$ into $S.holes\_next$

- Given rule $S \rightarrow S_1 S_2$, below is done in one-pass:
  - Gen $S_1.code$, making $S_1.holes\_next$
  - Gen $S_2.code$, filling $S_1.holes\_next$, making $S_2.holes\_next$
  - Pass on $S_2.holes\_next$ to $S.holes\_next$
Backpatching Rules for Boolean Expressions

3 functions for implementing backpatching

- makelist(i) — creates a new list out of statement index i
- merge(p1, p2) — returns merged list of p1 and p2
- backpatch(p, i) — fill holes in list p with statement index i

\[
E \rightarrow E_1 \text{ or } M \ E_2
\]

\[
\{ \text{ backpatch}(E_1.\text{holes\_false}, M.\text{quad}); \\
E.\text{holes\_true} = \text{merge}(E_1.\text{holes\_true}, E_2.\text{holes\_true}); \\
E.\text{holes\_false} = E_2.\text{holes\_false}; \}
\]

\[
E \rightarrow E_1 \text{ and } M \ E_2
\]

\[
\{ \text{ backpatch}(E_1.\text{holes\_true}, M.\text{quad}); \\
E.\text{holes\_false} = \text{merge}(E_1.\text{holes\_false}, E_2.\text{holes\_false}); \\
E.\text{holes\_true} = E_2.\text{holes\_true}; \}
\]

\[
M \rightarrow \varepsilon
\]

\[
\{ \text{ M.}\text{quad} = \text{nextquad}; \}
\]

/* nextquad is index of next quadruple to be generated */
More One Pass SDT Rules

\[
E \rightarrow \text{true} \quad \{ \ E.holes\_true = \text{makelist}(\text{nextquad}); \\
\quad \text{emit('goto ___'); } \}
\]

\[
E \rightarrow \text{false} \quad \{ \ E.holes\_false = \text{makelist}(\text{nextquad}); \\
\quad \text{emit('goto ___'); } \}
\]

\[
E \rightarrow \text{id1 \ relop \ id2} \quad \{ \ E.holes\_true = \text{makelist}(\text{nextquad}); \\
\quad E.holes\_false = \text{makelist}(\text{nextquad}+1); \\
\quad \text{emit('if id1.place 'relop' id2.place 'goto ___');} \\
\quad \text{emit('goto ___'); } \}
\]

\[
E \rightarrow \text{not \ E1} \quad \{ \ E.holes\_true = E1.holes\_false; \\
\quad E.holes\_false = E1.holes\_true; \}
\]

\[
E \rightarrow (\text{E1}) \quad \{ \ E.holes\_true = E1.holes\_true; \\
\quad E.holes\_false = E1.holes\_false; \}
\]
Backpatching Example

- E → (a<b) or M1 (c<d and M2 e<f)

- When reducing (a<b) to E1, we have
  100: if(a<b) goto ___  E1.holes_true=(100)
  101: goto ___  E1.holes_false=(101)

- When reducing ε to M1, we have
  M1.quad = 102

- When reducing (c<d) to E2, we have
  102: if(c<d) goto ___  E2.holes_true=(102)
  103: goto ___  E2.holes_false=(103)

- When reducing ε to M2, we have
  M2.quad = 104

- When reducing (e<f) to E3, we have
  104: if(e<f) goto ___  E3.holes_true=(104)
  105: goto ___  E3.holes_false=(105)
Backpatching Example (cont.)

When reducing (E2 and M2 E3) to E4, we \texttt{backpatch((102), 104)};
\begin{itemize}
  \item 100: if(a<b) goto ___
  \item 101: goto ___
  \item 102: if(c<d) goto 104
  \item 103: goto ___
  \item 104: if(e<f) goto ___
  \item 105: goto ___
\end{itemize}

When reducing (E1 or M1 E4) to E5, we \texttt{backpatch((101), 102)};
\begin{itemize}
  \item 100: if(a<b) goto ___
  \item 101: goto 102
  \item 102: if(c<d) goto 104
  \item 103: goto ___
  \item 104: if(e<f) goto ___
  \item 105: goto ___
\end{itemize}
Why do I still have holes in E5?

- The true and false branches have not yet been generated
- e.g. Given $S \rightarrow$ if $E5$ then $S1$ else $S2$, $E5.t$ and $E5.f$ filled when $S1$ and $S2$ generated, respectively
Problem

☐ Try this at home. Refer to textbook Chapter 6.6, 6.7.

☐ Write SDT rule (one pass using backpatching) for the following statement

\[ S \rightarrow \text{while } E1 \text{ do} \]

\[ \quad \text{if } E2 \]

\[ \quad \text{then } S2 \]

\[ \quad \text{endif} \]

\[ \text{endwhile} \]
**Solution Hint**

S → while E1 do if E2 then S2 endif endwhile

<table>
<thead>
<tr>
<th>Known Attributes</th>
<th>Attributes to Evaluate/Process</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Two Pass</strong></td>
<td></td>
</tr>
<tr>
<td>E1.code</td>
<td>E1.true, E1.false</td>
</tr>
<tr>
<td>E2.code</td>
<td>E2.true, E2.false</td>
</tr>
<tr>
<td>S2.code</td>
<td>S2.next</td>
</tr>
<tr>
<td>S.next</td>
<td>S.code</td>
</tr>
<tr>
<td><strong>One Pass</strong></td>
<td></td>
</tr>
<tr>
<td>E1.code, E1.holes_true</td>
<td>S.code</td>
</tr>
<tr>
<td>E1.holes_false</td>
<td>S.holes_next</td>
</tr>
<tr>
<td>E2.code, E2.holes_true</td>
<td></td>
</tr>
<tr>
<td>E2.holes_false</td>
<td></td>
</tr>
<tr>
<td>S.code, S.holes_next</td>
<td></td>
</tr>
<tr>
<td>S.code, S.holes_next</td>
<td></td>
</tr>
</tbody>
</table>