Code Generation
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?

What 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 $\rightarrow$ low-level-IR converter
- Back-end: a new low-level IR $\rightarrow$ 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

\[
\begin{align*}
  t1 &= x \times y \\
  t2 &= x \times y \\
  t3 &= t1 + t2
\end{align*}
\]

- 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):

\[
\begin{align*}
  t1 &= x \times y \\
  t3 &= t1 + t1
\end{align*}
\]

- 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 op is an arithmetic or logical operation (binary operation)

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

- Copy statement:
  \[ x = y \]

- Unconditional jump statement:
  \[ \text{goto } L \]
  where L is label
Conditional jump statement:

\[
\text{if (x relop y) goto L}
\]

where relop is a relational operator such as =, \(\neq\), >, <

Procedural call statement:

\[
\text{param x}_1, \ldots, \text{param x}_n, \text{call F}_y, n
\]

As an example, \(\text{foo(x1, x2, x3)}\) is translated to

\[
\text{param x}_1 \\
\text{param x}_2 \\
\text{param x}_3 \\
\text{call foo, 3}
\]

Procedural call return statement:

\[
\text{return y}
\]

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:

\[ x = \& y \quad ; \text{a pointer } x \text{ is set to location of } y \]
\[ y = *x \quad ; \text{ } y \text{ is set to the content of the address} \]
\[ \quad \quad \quad ; \text{stored in pointer } x \]
\[ *y = x \quad ; \text{object pointed to by } y \text{ gets value } x \]
Implementation of Three-Address Code

- 3 possible ways (and more)
  - quadruples
  - triples
  - indirect triples

- Trade-offs between space, speed, ease of manipulation

- Using quadruples
  \[ \text{op arg1, arg2, result} \]
  - There are four(4) fields at maximum
  - Arg1 and arg2 are optional, depending on op

Examples:

- \( x = a + b \) => + a, b, x
- \( x = -y \) => - y, , x
- \( \text{goto } L \) => goto , , L
Using Triples

- **Triple**: Quadruple without the result field
- **Can refer to results by indices 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

What If LHS of assignment 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. \(x.i = y\))

Compute address of LHS location beforehand

Example: triples for array assignment statement

\[ x[i] = y \]

is translated to

\[ (0) [\] x i \]
\[ (1) = (0) y \]

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>
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></th>
<th>Quadruples</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>op arg1 arg2 result</td>
<td>op arg1 arg2</td>
</tr>
<tr>
<td>(0)</td>
<td>- c t1</td>
<td>- c</td>
</tr>
<tr>
<td>(1)</td>
<td>* b t1 t2</td>
<td>* b (0)</td>
</tr>
<tr>
<td>(2)</td>
<td>+ t2 t2 t5</td>
<td>+ (1) (1)</td>
</tr>
<tr>
<td>(3)</td>
<td>= t5 a</td>
<td>= a (4)</td>
</tr>
</tbody>
</table>
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>(ptr to triple database)</th>
<th>Indirect Triples</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th><strong>op</strong></th>
<th><strong>arg1</strong></th>
<th><strong>arg2</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>-</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>*</td>
<td>b</td>
<td>(0)</td>
</tr>
<tr>
<td>(2)</td>
<td>-</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>*</td>
<td>b</td>
<td>(2)</td>
</tr>
<tr>
<td>(4)</td>
<td>+</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(5)</td>
<td>=</td>
<td>a</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, some entries in database can be reused

- i.e. Entries in triple database do not have to be contiguous
Every variable is assigned to exactly once statically

- Convert original variable name to name\textit{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 ( ...) } \\
  x &= x + 5; \\
  x &= x * 4; \\
  y &= x * 4; \\
  \text{true} & \quad \text{false} \\
\end{align*}

\begin{align*}
  x_1 &= a + b; \\
  y_1 &= x_1 - c; \\
  x_2 &= x_1 - y_1; \\
  \text{if ( ...) } \\
  x_3 &= x_2 + 5; \\
  x_4 &= x_2 * 4; \\
  x_5 &= \phi(x_3, x_4); \\
  y_2 &= x_5 * 4; \\
  \text{true} & \quad \text{false} \\
\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**

- 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`
  - **What is 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 and 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;
}
```
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>
<th>Addr(a)←0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x000c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0010</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 the 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</th>
<th>Addr(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x0004</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x000c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Offset=4
Addr(a)←0
When there are multiple variables defined in a procedure, we layout the variable sequentially. Use variable offset, to get address of \( x \):

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

```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. We 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=16</th>
<th>Addr(a)←0</th>
<th>Addr(b)←4</th>
<th>Addr(c)←8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td></td>
<td></td>
<td></td>
<td></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 and use variable offset, to get address of x:

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

```cpp
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>
<th>Addr(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x000c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offset = 20
More About Storage Layout (I)

Allocation alignment

- Enforce `addr(x) mod sizeof(x.type) == 0`
- Most machine architectures are designed such that computation is most efficient at `sizeof(x.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: **MSB** (most significant byte) in lowest address
- Little endian: **LSB** (least significant byte) in lowest address

```
Memory  Register

...  31.........................0
a:    0A
a+1:  0B
a+2:  0C
a+3:  0D
...  
```

Big-endian
Endianness

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

---

**Big-endian**

- Memory: ... 0A 0B 0C 0D ...
- Register: 0A 0B 0C 0D...
- Big-endian

**Little-endian**

- Memory: ... 0D 0C 0B 0A ...
- Register: 0A 0B 0C 0D...
- Little-endian
More About Storage Layout (III)

Questions still unanswered
  ➢ How are non-local variables laid out?
  ➢ How dynamically allocated memory laid out?

Will be answered in next chapter: Runtime Management
Processing Declarations

Translating the declaration in a single procedure

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

\[
P \rightarrow M \ D
\]

\[
M \rightarrow \epsilon \\
\{ \text{offset}=0; \} /* \text{reset offset before layout} */
\]

\[
D \rightarrow D \ D
\]

\[
D \rightarrow T \ id \\
\{ \text{enter(id.name, T.type, offset); offset += T.width; } \}
\]

\[
T \rightarrow \text{integer} \\
\{ T.type=integer; T.width=4; \}
\]

\[
T \rightarrow \text{real} \\
\{ T.type=real; T.width=8; \}
\]

\[
T \rightarrow T_1[\text{num}] \\
\{ T.type=\text{array}(\text{num.val, T}_1.type); \\
T.width=\text{num.val} \times T_1.width; \}
\]

\[
T \rightarrow \ast T_1 \\
\{ T.type=\text{ptr}(T_1.type); T.width=4; \}
\]
Processing Nested Declarations

- Must maintain current offset 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

```c
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>Symbol</th>
<th>Offset Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>a 0</td>
</tr>
<tr>
<td></td>
<td>b 4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
P1
a 0
b 4
P2
q 0
...

check point #1
check point #2
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;
    } // check point #3
    use q
}
```

Symbol Table Stack

<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>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

Stack

- \( P1 \)
  - \( q \): 0
  - \( \ldots \)
- \( P2 \)
  - \( \ldots \)
- \( P3 \)
  - \( \ldots \)
Nested Declaration Example

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
}

Symbol Table Stack

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

check point #1
check point #2
check point #3
check point #4
void P1() {
    int a;
    int b;
    // check point #1
}

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

void P3() {
    int J;
    // check point #3
    void P4() {
        use a
        // check point #4
    }

    use q
    // check point #5
}

Symbol Table Stack
Offset Stack

<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>

check point #1
check point #2
check point #3
use a
check point #4
check point #5
Syntax directed translation rules

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

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

```c
int A[N];
A[i] ++;
```

- base — address of the first element
- width — width of each element

Addressing an array element:

\[
\text{address}(A[i]) = \text{base} + i \times \text{width}
\]
Laying out n-dimensional array in 1-dimensional memory

\[
\text{int } A[N_1][N_2]; /* int } A[0..N_1-1][0..N_2-1]; */ \\
A[i_1][i_2]++; \\
A[0,0]
\]

\[
N_2 \\
i_1
\]

\[
A[N_1-1,N_2-1]
\]

\[
N_1
\]

\[
i_2
\]
C arrays are organized row major — stored row by row

Blue items come before $A[i_1, i_2]$ in linear memory

$$\text{address}(A[i_1, i_2]) = \text{base} + (i_1 \times N_2 + i_2) \times \text{width}$$
Arrays can have arbitrary 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]

Offset of k-dimension array item can be defined recursively (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

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];)

\[
\begin{align*}
S &\rightarrow L = E \\
&\quad \{ t = \text{newtemp}(); \text{emit}( t \ '==' \ L.\text{place} \ '' \ L.\text{width}); \\
&\quad \quad \text{emit}(t \ '==' \ L.\text{base} \ '+' \ t); \text{emit} (''t \ '==' \ E.\text{place}); \}
\end{align*}
\]

\[
\begin{align*}
E &\rightarrow L \\
&\quad \{ E.\text{place} = \text{newtemp}(); t= \text{newtemp}(); \\
&\quad \quad \text{emit}( t \ '==' \ L.\text{place} \ '' \ L.\text{width}); \text{emit} (E.\text{place} \ '==' \ (L.\text{base} \ '+' \ t)); \}
\end{align*}
\]

\[
\begin{align*}
L &\rightarrow \text{id} \ [ \ E \ ] \\
&\quad \{ \text{L.base} = \text{lookup(id).base}; \text{L.width} = \text{lookup(id).width}; \\
&\quad \quad \text{L.bounds} = \text{lookup(id).bounds}; \text{L.dim=1}; \text{L.\text{place} = E.\text{place}}; \}
\end{align*}
\]

\[
\begin{align*}
L &\rightarrow L1 \ [ \ E \ ] \\
&\quad \{ \text{L.base} = L1.\text{base}; \text{L.width} = L1.\text{width}; \text{L.dim} = L1.\text{dim} + 1; \\
&\quad \quad \text{L.\text{place} = newtemp}(); \\
&\quad \quad \text{emit}( L.\text{place} \ '==' \ L1.\text{place} \ '' \ \text{L.bounds[L.dim]}); \\
&\quad \quad \text{emit}( L.\text{place} \ '==' \ L.\text{place} \ '+' \ E.\text{place}); \}
\end{align*}
\]
Boolean expression: \( a \ op \ b \)
- where \( op \) can be \(<\), \( >\), \( >=\), \&\&, ||, ...

Short-circuit evaluation
- To skip evaluation of parts of a boolean expression that do not contribute to the boolean value
- In expression \( A \ &\& \ B \), not evaluating \( B \) when \( A \) is false
- In expression \( A \ || \ B \), not evaluating \( B \) when \( A \) is true

Has semantic (as well as performance) implications
- In the following C code:
  ```c
  if (flag || foo()) { ... };
  ```
- If \( \text{flag} \) is true, \( \text{foo()} \) should never execute
- If \( \text{foo()} \) were to execute ...
  - Not only would the program run slower
  - But \( \text{foo()} \) may have side-effects (e.g. updating a global var)
Boolean Expressions (w/o Short-Circuiting)

- Computed just like any other arithmetic expression:
  \[ E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f) \equiv t1 = a < b \]
  \[ t2 = c < d \]
  \[ t3 = e < f \]
  \[ t4 = t2 \text{ and } t3 \]
  \[ t5 = t1 \text{ or } t4 \]

- Then used in control flow statements:
  \[ S \rightarrow \text{if } E \text{ then } S1 \equiv \text{if } (! t5) \text{ goto } S.next \]
  \[ S1.label: \text{(code for } S1) \]
  \[ S.next: ... \]

- Can be applied to all types of control flow statements
  \[ S \rightarrow \text{if } E \text{ then } S1 \mid \text{if } E \text{ then } S1 \text{ else } S2 \mid \text{while } E \text{ do } S1 \]
Boolean Expressions (w/o Short-Circuiting)

- Syntax directed translation rules
  
  E → E1 relop E2 { E.place = newtemp();

  emit(E.place '==' E1.place 'relop' E2.place);
  }

  S → if E then S1 { S.code = emit("if !" E.place 'goto' S.next) ||

  emit(S1.label':') || S1.code || emit(S.next':'); }

  S1.label: label for code generated for 'S1'

  S.next: label for code generated after 'S'

  S → S1; S2 { S1.next = S2.label; S2.next = S.next;

  S.code = S1.code || emit(S2.label':') || S2.code; }

  S2.label: label for code generated for 'S2'
Boolean Expressions (with Short-Circuiting)

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

  ➤ E.true: label for code to execute on 'true'
  ➤ E.false: label code to execute on 'false'

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

- Can be applied to all types of control flow statements
  \[ S \rightarrow \text{if } E \text{ then } S1 | \text{if } E \text{ then } S1 \text{ else } S2 | \text{while } E \text{ do } S1 \]
Boolean Expressions (with Short-Circuiting)

Syntax directed translation rules

\[ E \rightarrow id1 \text{ relop } id2 \]
\[ \{ \text{E.code = emit('if' id1 'relop' id2 'goto' E.true) || emit('goto' E.false); } \]  
E.true: label for code to execute on 'true'
E.false: label for code to execute on 'false'

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

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

\[ S \rightarrow \text{if } E \text{ then } S1 \]
\[ \{ \text{E.true = S1.label; E.false = S.next; S.code = E.code || emit(S1.label':') || S1.code; } \]

\[ S \rightarrow S1; S2 \]
\[ \{ \text{S1.next = S2.label; S2.next = S.next; S.code = S1.code || emit(S2.label':') || S2.code; } \]
The Problem with Jumps to Labels

- Labels are names for jump targets
  - Must be associated with a concrete instruction address for final machine code generation
  - Can only be generated along with associated instruction (Instruction address cannot be known before generation)

- Problem with a forward jump to a label
  - Forward jump: a jump to an instruction below you
  - Label for jump target has not yet been generated
  - How can you generate the jump instruction w/o the label?
  - Manifest as non-L-attributes in the attribute grammar

- Labels pose a problem with left-to-right single scan parsers such as LL or LR
Non-L-Attributes marked in red:

1. Languages \textit{without short circuiting}

   \[ S \rightarrow \text{if } E \text{ then } S_1 \{ \text{S.code} = \text{emit}('if !' E.place 'goto' S.next) || \text{emit}(S_1.label':') || \text{S1.code} || \text{emit}(S.next':'); \} \]

   \[ S \rightarrow S_1; S_2 \{ \text{S1.next} = S_2.label; \text{S2.next} = S.next; \]

   \[ \text{S.code} = \text{S1.code} || \text{emit}(S2.label':') || \text{S2.code}; \]

2. Languages \textit{with short circuiting}

   \[ E \rightarrow E_1 \&\& E_2 \{ \text{E1.true} = E_2.label; \text{E1.false} = E.false; \]

   \[ \text{E2.true} = E.true; \text{E2.false} = E.false; \]

   \[ \text{E.code} = \text{E1.code} || \text{emit}(E2.label':') || \text{E2.code}; \]

   \[ S \rightarrow \text{if } E \text{ then } S_1 \{ \text{E.true} = S_1.label; \text{E.false} = S.next; \]

   \[ \text{S.code} = \text{E.code} || \text{emit}(S1.label':') || \text{S1.code}; \]

   \[ S \rightarrow S_1; S_2 \{ \text{S1.next} = S_2.label; \text{S2.next} = S.next; \]

   \[ \text{S.code} = \text{S1.code} || \text{emit}(S2.label':') || \text{S2.code}; \]
Dealing with Non-L-Attribute Labels

Solutions: two methods

Two-pass approach — process the code twice
- Pass 1: Generate code with non-address-mapped labels (Store labels in hashtable with address when later mapped)
- Pass 2: Replace non-address-mapped labels with addresses (Use hashtable built in Pass 1 to map labels to addresses)

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

For a language with short circuiting, we’ll discuss:
1. Syntax Directed Definition for syntax directed translation
2. Syntax Directed Translation Scheme for two pass approach
2. Syntax Directed Translation Scheme for one pass approach
newlabel(): Generates a new non-address-mapped label. Label mapped to address when emitted using emit().

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; }

S → S1 S2
   { S1.next = newlabel();
     S2.next = S.next;
     S.code = S1.code || emit(S1.next':') || S2.code; }
More SDD Rules

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

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

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

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

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

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

Write SDD rule 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:

<table>
<thead>
<tr>
<th>Known Attributes</th>
<th>Updated Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDD</td>
<td>E1.code</td>
</tr>
<tr>
<td></td>
<td>E2.code</td>
</tr>
<tr>
<td></td>
<td>S2.code</td>
</tr>
<tr>
<td></td>
<td>S.next</td>
</tr>
<tr>
<td></td>
<td>E1.true, E1.false</td>
</tr>
<tr>
<td></td>
<td>E2.true, E2.false</td>
</tr>
<tr>
<td></td>
<td>S2.next</td>
</tr>
<tr>
<td></td>
<td>S.code</td>
</tr>
</tbody>
</table>
Pass 1 of two-pass based SDTS

- For $S \rightarrow \text{if } E \text{ then } S_1$:
  - $E\.true = \text{newlabel}(), \ E\.false = S_1\.next = S\.next$
  - Generate $E\.code$ using unmapped labels $E\.true, S\.next$
  - Generate $S_1\.code$ using unmapped label $S\.next$
  - When $S_1$ is emitted, map $E\.true$ to actual address

- For $S \rightarrow S_1 \ S_2$:
  - $S_1\.next = \text{newlabel}(), \ S_2\.next = S\.next$
  - Generate $S_1\.code$ using unmapped label $S_1\.next$
  - Generate $S_2\.code$ using unmapped label $S\.next$
  - When $S_2$ is emitted, map $S_1\.next$ to actual address

Pass 2 of two-pass based SDTS

- For $S \rightarrow \text{if } E \text{ then } S_1$:
  - Replace labels $E\.true, S\.next$ with mapped addresses

- For $S \rightarrow S_1 \ S_2$:
  - Replace labels $S_1\.next, S\.next$ with mapped addresses
If grammar contains L-attributes only, then it can be processed in one pass.

However, we know non-L attributes are necessary:

- Example: $E_1\.false$ in $E \rightarrow E_1 \mid E_2$
- Is there a general solution to this problem?

Solution: Backpatching

- Leave holes in IR for non-L attributes, record their locations in hole lists, and fill in holes when values are known.
  - hole: jump target address that is not yet known
  - Holes for particular jump target organized as a hole list
  - Holes filled when target label is emitted and address known
  - All holes are filled in by the end of code generation (Since by then, all target addresses have been emitted)
One-Pass Based Syntax Directed Translation Scheme

- Attributes for one-pass based approach
  - Expressions (e.g. \( E \rightarrow E_1 || E_2 \), \( E \rightarrow \text{id1 relop id2} \) ...)
    Attributes: \( E.\text{holes}_\text{true} \), \( E.\text{holes}_\text{false} \)
  - Statements (e.g. \( S \rightarrow \text{if E then S1} \), \( S \rightarrow S_1 S_2 \) ...)
    Attributes: \( S.\text{holes}_\text{next} \)

- Given rule \( S \rightarrow \text{if E then S1} \), below is done in one-pass:
  - Gen \( E.\text{code} \), making \( E.\text{holes}_\text{true} \), \( E.\text{holes}_\text{false} \)
  - Gen \( S_1.\text{code} \), patching \( E.\text{holes}_\text{true} \), making \( S_1.\text{holes}_\text{next} \)
  - Merge \( E.\text{holes}_\text{false} \), \( S_1.\text{holes}_\text{next} \) into \( S.\text{holes}_\text{next} \)

- Given rule \( S \rightarrow S_1 S_2 \), below is done in one-pass:
  - Gen \( S_1.\text{code} \), making \( S_1.\text{holes}_\text{next} \)
  - Gen \( S_2.\text{code} \), patching \( S_1.\text{holes}_\text{next} \), making \( S_2.\text{holes}_\text{next} \)
  - Pass on \( S_2.\text{holes}_\text{next} \) to \( S.\text{holes}_\text{next} \)
One-Pass SDTS using Backpatching

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 → E1 or M E2  { backpatch(E1.holes_false, M.quad);
                    E.holes_true = merge(E1.holes_true, E2.holes_true);
                    E.holes_false = E2.holes_false;
                }

E → E1 and M E2 { backpatch(E1.holes_true, M.quad);
                    E.holes_false = merge(E1.holes_false, E2.holes_false);
                    E.holes_true = E2.holes_true;
                }

M → ε           { M.quad = nextquad; }
/* nextquad is index of next quadruple to be generated */
```
One-Pass SDTS using Backpatching

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

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

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

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

\[
E \rightarrow \text{(E1)} \quad \{ \, \text{E.holes\_true} = \text{E1.holes\_true}; \\
\quad \text{E.holes\_false} = \text{E1.holes\_false}; \}
\]
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 ___  
    101: goto ___

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

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

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

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

- E → (a<b) or M1 (c<d and M2 e<f)

- When reducing (E2 and M2 E3) to E4, we backpatch((102), 104);
  100: if(a<b) goto ___
  101: goto ___
  102: if(c<d) goto 104
  103: goto ___
  104: if(e<f) goto ___
  105: goto ___

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

- The true and false branches have not yet been generated
- e.g. Given S → 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 semantic action for SDTS using backpatching for the following statement

\[
S \rightarrow \text{while } E1 \text{ do}
\]
\[
\quad \text{if } E2
\]
\[
\quad \text{then } S2
\]
\[
\quad \text{endif}
\]
endwhile
Solution Hint

S → while E1 do if E2 then S2 endif endwhile

<table>
<thead>
<tr>
<th></th>
<th>Known Attributes</th>
<th>Updated Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDD</td>
<td>E1.code</td>
<td>E1.true, E1.false</td>
</tr>
<tr>
<td></td>
<td>E2.code</td>
<td>E2.true, E2.false</td>
</tr>
<tr>
<td></td>
<td>S2.code</td>
<td>S2.next</td>
</tr>
<tr>
<td></td>
<td>S.next</td>
<td>S.code</td>
</tr>
<tr>
<td>One</td>
<td>E1.code, E1.holes_true</td>
<td>E1.holes_true</td>
</tr>
<tr>
<td>Pass</td>
<td>E1.holes_false</td>
<td>E2.holes_true</td>
</tr>
<tr>
<td>SDTS</td>
<td>E2.code, E2.holes_true</td>
<td>E2.holes_false</td>
</tr>
<tr>
<td></td>
<td>E2.holes_false</td>
<td>S2.holes_next</td>
</tr>
<tr>
<td></td>
<td>S2.code, S2.holes_next</td>
<td>S.holes_next, S.code</td>
</tr>
</tbody>
</table>