Code Generation
Modern Compiler Project

Fortran program
- Fortran's Lexer, Parser, and Static Checker
- Intermediate Code Generator
- Code Optimization
- MIPS Code Generator
- MIPS code

C program
- C's Lexer, Parser, and Static Checker
- Intermediate Code Generator
- Code Optimization
- X86 Code Generator
- X86 code

C# program
- C#'s Lexer, Parser, and Static Checker
- Intermediate Code Generator
- Code Optimization
- ARM Code Generator
- ARM code
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 $\rightarrow$ 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 \( X = Y \ op \ Z \)
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
Example

- An example:
  \[ x \times y + x \times y \]
  is translated to
  
  \[
  t_1 = x \times y ; \quad t_1, t_2, t_3 \text{ are temporary variables} \\
  t_2 = x \times y \\
  t_3 = t_1 + t_2
  \]

  - 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):
    
    \[
    t_1 = x \times y \\
    t_3 = t_1 + t_1
    \]

  - 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:

```c
if (x relop y) goto L
```
where relop is a relational operator such as =, ≠, >, <

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:

\[ x = & y \] ; a pointer \( x \) is set to location of \( y \)

\[ y = * x \] ; \( y \) is set to the content of the address

\[ *y = x \] ; object pointed to by \( y \) gets value \( x \)
There are three possible ways to store the code:

- quadruples
- triples
- indirect triples

Using quadruples:

\[ \text{op arg1, arg2, result} \]

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

Examples:

\[
\begin{align*}
  x &= a + b \quad \Rightarrow + a, b, x \\
  x &= -y \quad \Rightarrow - y, , x \\
  \text{goto } L &= \quad \Rightarrow \text{goto , , L}
\end{align*}
\]
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
  (0) [] x i
  (1) = (0) y
  - One or more triples are used to compute the location
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>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>Quadruples</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>op arg1 arg2 result</td>
<td>op arg1 arg2</td>
</tr>
<tr>
<td>(0) - c t1</td>
<td>- c</td>
</tr>
<tr>
<td>(1) * b t1 t2</td>
<td>* b</td>
</tr>
<tr>
<td>(2) + t2 t2 t5</td>
<td>+ (1)</td>
</tr>
<tr>
<td>(3) = t5 a</td>
<td>= a</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>(0)</td>
</tr>
<tr>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(2)</td>
<td>(2)</td>
</tr>
<tr>
<td>(3)</td>
<td>(3)</td>
</tr>
<tr>
<td>(4)</td>
<td>(4)</td>
</tr>
<tr>
<td>(5)</td>
<td>(5)</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
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

\[
\begin{align*}
  &x = a + b; \\
  &y = x - c; \\
  &x = x - y; \\
  &\text{if ( ...)}
\end{align*}
\]

\[
\begin{align*}
  &x = a + b; \\
  &y = x - c; \\
  &x = x - y; \\
  &\text{if ( ...)}
\end{align*}
\]

\[
\begin{align*}
  &x = x + 5; \\
  &x = x * 4;
\end{align*}
\]

\[
\begin{align*}
  &x = x + 5; \\
  &x = x * 4;
\end{align*}
\]

\[
\begin{align*}
  &x_1 = a + b; \\
  &y_1 = x_1 - c; \\
  &x_2 = x_1 - y_1; \\
  &\text{if ( ...)}
\end{align*}
\]

\[
\begin{align*}
  &x_3 = x_2 + 5; \\
  &x_4 = x_2 * 4;
\end{align*}
\]

\[
\begin{align*}
  &x_3 = x_2 + 5; \\
  &x_4 = x_2 * 4;
\end{align*}
\]

\[
\begin{align*}
  &x_5 = \phi(x_3, x_4); \\
  &y_2 = x_5 * 4;
\end{align*}
\]
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
  - ...


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 variables sequentially and 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;
}
```
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 \( x \)
  - `addr(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 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=4</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 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;
}
```

Address

<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`.

- `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=16</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>Addr(c)←8</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 a variable `offset` to get the address of a variable.

- `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=20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0</td>
</tr>
<tr>
<td>0x0004</td>
<td>4</td>
</tr>
<tr>
<td>0x0008</td>
<td>8</td>
</tr>
<tr>
<td>0x000c</td>
<td>16</td>
</tr>
<tr>
<td>0x0010</td>
<td></td>
</tr>
</tbody>
</table>

```c
addr(a) ← 0
addr(b) ← 4
addr(c) ← 8
addr(d) ← 16
```
Allocation alignment

- Enforce \texttt{addr(x) mod sizeof(x.type) == 0}
- Most machine architectures are designed such that computation is most efficient at \texttt{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 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

- Big-endian
  - Memory layout: "... 0A 0B 0C 0D ...
  - Register layout: 31.............................................0

- Little-endian
  - Memory layout: "... 0D 0C 0B 0A ...
  - Register layout: 31.............................................0

- Example:
  - Big-endian: `a: 0A`, `a+1: 0B`, `a+2: 0C`, `a+3: 0D`
  - Little-endian: `a: 0A`, `a+1: 0B`, `a+2: 0C`, `a+3: 0D`
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

\[
P \rightarrow M \ D
\]
\[
M \rightarrow \varepsilon \quad \{ \text{offset=0; } \} /* \text{reset offset before layout } */
\]
\[
D \rightarrow D \ D
\]
\[
D \rightarrow T \ id \quad \{ \text{enter(id.name, T.type, offset); offset += T.width; } \}
\]
\[
T \rightarrow \text{integer} \quad \{ \text{T.type=integer; T.width=4; } \}
\]
\[
T \rightarrow \text{real} \quad \{ \text{T.type=real; T.width=8;} \}
\]
\[
T \rightarrow T1[\text{num}] \quad \{ \text{T.type=array(num.val, T1.type); T.width=num.val * T1.width; } \}
\]
\[
T \rightarrow * \ T1 \quad \{ \text{T.type=ptr(T1.type); T.width=4; } \}
\]
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

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Stack</td>
</tr>
<tr>
<td></td>
<td>P1</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>Stack</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</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

8
4
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
    }
    use a
}
check point #3
void P4() {
    use a
}
}
```

Symbol Table Stack

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
</tr>
</tbody>
</table>

Stack

<table>
<thead>
<tr>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>

P2

P3
Nested Declaration Example

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

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

void P4() {
    use a
    check point #4
}

Symbol Table Stack

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

---

check point #1
check point #2
check point #3
check point #4
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
        check point #3
        check point #4
    }
    int J;
    use q
    check point #5
}

check point #1
check point #2
check point #3
check point #4
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

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

Check point #1: P1 begins with int a and int b.
Check point #2: P2 begins with int q.
Check point #3: void P4 begins with use a.
Check point #4: void P4 ends with check point #4.
Check point #5: P3 begins with int J and uses variable q.

The symbol table and stack show the state of variables at different points in the code.
Processing Nested Declarations

Syntax directed translation rules

\[ P \rightarrow M \; D \quad \{ \text{pop(\text{active}); pop(\text{offset});} \} \]
\[ M \rightarrow \varepsilon \quad \{ t=\text{mktable(); push(t, \text{active}); push(0,\text{offset});} \} \]
\[ D \rightarrow D; \; D \]
\[ D \rightarrow \text{void pid() \{ N \; D; \; S \}} \quad \{ \text{pop(\text{active}); pop(\text{offset});} \]
\[ \quad \quad \text{enterproc(\text{top(\text{active})}, \text{pid, pid_addr});} \} \]
\[ N \rightarrow \varepsilon \quad \{ t=\text{mktable(); push(t, \text{active}); push(0, \text{offset});} \} \]
\[ D \rightarrow T \; \text{id;} \quad \{ \text{enter(\text{top(\text{active})}, \text{id, T.type, top(\text{offset})});} \]
\[ \quad \quad \text{top(\text{offset}) = top(\text{offset})+ T.width;} \} \]
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; }
Recall generalized row/column major addressing

For example:
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]

Row major: addressing a k-dimension array item 
(low_i = base = 0)
1-dimension: \( A_1 = a_1 \times \text{width} \quad a_1 = i_1 \)
2-dimension: \( A_2 = a_2 \times \text{width} \quad a_2 = a_1 \times N_2 + i_2 \)
3-dimension: \( A_3 = a_3 \times \text{width} \quad a_3 = a_2 \times N_3 + i_3 \)
...
k-dimension: \( A_k = a_k \times \text{width} \quad a_k = a_{k-1} \times N_k + i_k \)
Processing an array assignment (e.g. A[i] = B[j];)

\[
\begin{align*}
S & \rightarrow L = E & \{ t = \text{newtemp}(); \text{emit}( t '=' L.\text{place} '*' L.\text{width}); \\
& & \text{emit}(t '=' L.\text{base } '+' t); \text{emit}(t '=' E.\text{place}); \} \\
E & \rightarrow L & \{ E.\text{place} = \text{newtemp}(); t = \text{newtemp}(); \\
& & \text{emit}( t '=' L.\text{place } '*' L.\text{width}); \text{emit}( E.\text{place } '=' (L.\text{base } '+' t)); \} \\
L & \rightarrow \text{id } [ E ] & \{ L.\text{base} = \text{lookup(id).base}; L.\text{width} = \text{lookup(id).width}; \\
& & L.\text{bounds} = \text{lookup(id).bounds}; L.\text{dim}=1; L.\text{place} = E.\text{place}; \} \\
L & \rightarrow L1 [ E ] & \{ L.\text{base} = L1.\text{base}; L.\text{width} = L1.\text{width}; L.\text{dim} = L1.\text{dim} + 1; \\
& & L.\text{place} = \text{newtemp}(); \\
& & \text{emit}( L.\text{place } '=' L1.\text{place } '*' L.\text{bounds}[L.\text{dim}] ); \\
& & \text{emit}( L.\text{place } '=' L.\text{place } '+' E.\text{place}); \}
\end{align*}
\]
Boolean expression: \( a \ op \ b \)

where \( op \) 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:
   \[ E \rightarrow (a < b) \ or \ (c < d \ and \ e < f) \equiv \]
   \[ t_1 = a < b \]
   \[ t_2 = c < d \]
   \[ t_3 = e < f \]
   \[ t_4 = t_2 \ && \ t_3 \]
   \[ t_5 = t_1 \ || \ t_4 \]
Processing Boolean Expressions

2. Languages with short circuiting

- Implemented via a series of jumps:

  \[ E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f) \equiv \]
  
  ```
  if (a<b) goto E.true
goto L1
  
  L1: if (c<d) goto L2
goto E.false
  
  L2: if (e<f) goto E.true
goto E.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:**
  \[
  E \rightarrow id1 \text{ relop id2} \ \{ \ E.\text{code} = \text{emit(‘if’ id1 ’relop’ id2 ’goto’ E.true) || emit(’goto’ E.false)}; \ \}
  \]
  
  **E.true:** address of code to execute on ’true’ (inherited)
  
  **E.false:** address of code to execute on ’false’ (inherited)

- **S → if E then S1:**
  \[
  \text{S.true} = \text{S1.label; \ E.false = S.next;} \ \
  \text{S.code = E.code || emit(S1.label’:’) || S1.code; }
  \]
  
  **S1.label:** label created at the beginning of code S1

- **S → S1; S2:**
  \[
  \text{S1.next} = \text{S2.label;} \ \text{S2.next} = \text{S.next;} \ \
  \text{S.code = S1.code || emit(S2.label’:’) || S2.code; }
  \]

  **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 ||:
  \[ E \rightarrow E_1 \&\& E_2 \quad \{ \ E_1.\text{true} = E_2.\text{label}; \ E_1.\text{false} = E.\text{false}; \ \\
  \quad E.\text{code} = E_1.\text{code} \ |\| \text{emit}(E_2.\text{label}':') \ |\| E_2.\text{code}; \ \} \]
  
  \[ E \rightarrow E_1 \| E_2 \quad \{ \ E_1.\text{false} = E_2.\text{label}; \ E_1.\text{true} = E.\text{true}; \ \\
  \quad E.\text{code} = E_1.\text{code} \ |\| \text{emit}(E_2.\text{label}':') \ |\| E_2.\text{code}; \ \} \]

- Problem: \textbf{E}_1.\text{true}, \textbf{E}_1.\text{false} are non-L-attributes
  - \textbf{E}_1.\text{true}: Address of \textbf{E}_2.\text{label} known only when \textbf{E}_2 emitted
  - \textbf{E}_1.\text{false}: Address of \textbf{E}_2.\text{label} known only when \textbf{E}_2 emitted

- Do non-L-attributes preclude single pass SDTS?
  - Both LL and LR SDTS rely on L-attributed grammars
Syntax Directed Translation

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
Attributes for two pass based approach

- Statement $S \rightarrow \text{if } E \text{ then } S_1$
  - Inherited attributes: $E$.false, $S_1$.next
  - Non-L inherited attributes: $E$.true

- Statement $S \rightarrow S_1 \ S_2$
  - Inherited attributes: $S_2$.next
  - Non-L inherited attributes: $S_1$.next

Given rule $S \rightarrow \text{if } E \text{ then } S_1$, the two passes are:

1. Generate $E$.code using unmapped label $E$.true
   When $S_1$.code is generated, map $E$.true
2. Replace label $E$.true with actual address of $S_1$

Given rule $S \rightarrow S_1 \ S_2$, the two passes are:

1. Generate $S_1$.code using unmapped label $S_1$.next
   When $S_2$.code is generated, map $S_1$.next
2. Replace label $S_1$.next with actual address of $S_2$
Two Pass based Rules

\[ S \rightarrow \text{if } E \text{ then } S_1 \]
\[
\begin{align*}
&\{ \ E.\text{true} = \text{newlabel}; \\
&\quad E.\text{false} = S.\text{next}; \\
&\quad S_1.\text{next} = S.\text{next}; \\
&\quad S.\text{code} = E.\text{code} \ || \ \text{emit}(E.\text{true}') \ || \ S_1.\text{code}; \}
\end{align*}
\]

\[ S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2 \]
\[
\begin{align*}
&\{ \ S_1.\text{next} = S_2.\text{next} = S.\text{next}; \\
&\quad E.\text{true} = \text{newlabel}; \\
&\quad E.\text{false} = \text{newlabel}; \\
&\quad S.\text{code} = E.\text{code} \ || \ \text{emit}(E.\text{true}') \ || \\
&\quad \quad \quad S_1.\text{code} \ || \ \text{emit}('\text{goto } ' S.\text{next}) \ || \\
&\quad \quad \quad \text{emit}(E.\text{false}') \ || \ S_2.\text{code}; \}
\end{align*}
\]
More Two Pass based SDT Rules

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

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

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

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

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

\[ E \rightarrow \text{false} \]
\[
\{ E\text{.code }= \text{emit(‘goto’ E.false); }\}
Problem

- Try this at home. Refer to textbook Chapter 6.6.
- Write SDT rule (two pass) for the following statement

\[ S \rightarrow \text{while } E_1 \text{ do}
\]

\[ \quad \text{if } E_2 \]

\[ \quad \text{then } S_2 \]

\[ \quad \text{endif} \]

\[ \text{endwhile} \]
Backpatching

- 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 $E \rightarrow E1 \| E2$
  - 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)
Attributes for one-pass based approach

- Expressions $E \rightarrow E_1 \parallel E_2$, $E \rightarrow \text{id1 relop id2}$ ...
  — Synthesized: $E.holes\_true$, $E.holes\_false$
- Statements $S \rightarrow \text{if } E \text{ then } S_1$, $S \rightarrow S_1 S_2$ ...
  — 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

\[
E \rightarrow E_1 \text{ or } M \ E_2 \\
{ \text{ backpatch}(E_1.holes\_false, M.quad); } \\
{ \text{ E.holes\_true} = \text{ merge}(E_1.holes\_true, E_2.holes\_true); } \\
{ \text{ E.holes\_false} = \text{ E2.holes\_false}; } \\
\]

\[
E \rightarrow E_1 \text{ and } M \ E_2 \\
{ \text{ backpatch}(E_1.holes\_true, M.quad); } \\
{ \text{ E.holes\_false} = \text{ merge}(E_1.holes\_false, E_2.holes\_false); } \\
{ \text{ E.holes\_true} = \text{ E2.holes\_true}; } \\
\]

\[
M \rightarrow \epsilon \\
{ \text{ M.quad} = \text{ nextquad}; } \\
/* \text{ nextquad is index of next quadruple to be generated } */\]
More One Pass SDT Rules

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

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

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

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

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

- $E \rightarrow (a<b) \text{ or } M1 \ (c<d \text{ and } M2 \ e<f)$

- When reducing $(a<b)$ to $E1$, we have
  100: if $(a<b)$ goto ___
  101: goto ___
  $E1.\text{holes\_true}=(100)$
  $E1.\text{holes\_false}=(101)$

- When reducing $e$ to $M1$, we have
  $M1.\text{quad} = 102$

- When reducing $(c<d)$ to $E2$, we have
  102: if $(c<d)$ goto ___
  103: goto ___
  $E2.\text{holes\_true}=(102)$
  $E2.\text{holes\_false}=(103)$

- When reducing $e$ to $M2$, we have
  $M2.\text{quad} = 104$

- When reducing $(e<f)$ to $E3$, we have
  104: if $(e<f)$ goto ___
  105: goto ___
  $E3.\text{holes\_true}=(104)$
  $E3.\text{holes\_false}=(105)$
When reducing (E2 and M2 E3) to E4, we \texttt{backpatch((102), 104)};
\begin{verbatim}
100: if(a<b) goto ___
101: goto ___
102: if(c<d) goto 104
103: goto ___
104: if(e<f) goto ___
105: goto ___
\end{verbatim}
E4.holes_true=(104)
E4.holes_false=(103,105)

When reducing (E1 or M1 E4) to E5, we \texttt{backpatch((101), 102)};
\begin{verbatim}
100: if(a<b) goto ___
101: goto 102
102: if(c<d) goto 104
103: goto ___
104: if(e<f) goto ___
105: goto ___
\end{verbatim}
E5.holes_true=(100, 104)
E5.holes_false=(103,105)
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

\[ S \rightarrow \text{while } E_1 \text{ do }
\]
\[ \quad \text{if } E_2 
\]
\[ \quad \text{then } S_2 
\]
\[ \quad \text{endif} 
\]
\[ \text{endwhile} \]
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</td>
<td>S.code</td>
</tr>
<tr>
<td>E1.holes_true</td>
<td>S.holes_next</td>
</tr>
<tr>
<td>E1.holes_false</td>
<td></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>
</tbody>
</table>