Code Generation
Modern Compiler Project

- Pascal's Lexer, Parser, and Static Checker
- Intermediate Code Generator
- Code Optimization
- MIPS Code Generator
- MIPS code
- C’s Lexer, Parser, and Static Checker
- Intermediate Code Generator
- Code Optimization
- X86 Code Generator
- X86 code
- C#’s Lexer, Parser, and Static Checker
- Intermediate Code Generator
- Code Optimization
- ARM Code Generator
- ARM code
Multiple IRs

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

- High-Level IR
  - Examples: Abstract Syntax Tree, Parse Tree
  - Essentially a tree syntax and semantics of program
  - Need a high-level IR for each language
  - Purpose
    - Semantic analysis of program
    - Language-specific optimizations (e.g. inlining)

- Low-Level IR
  - Examples: Three address code, Static Single Assignment
  - Essentially an instruction set for an abstract machine
  - Tries to be language and machine independent
  - Purpose: Language / machine independent optimizations
Multiple IRs

- **Machine-Level IR**
  - Examples: x86 IR, ARM IR, MIPS IR
  - Actual instructions for a concrete machine ISA
  - Need a machine-level IR for each machine ISA
  - Purpose
    - Machine-specific optimizations (e.g. strength reduction)
    - Register allocation / code generation

- But could just use one IR (high-level IR) — some simple compilers do this

- Why multiple IRs?
Multiple IRs

- Machine-Level IR
  - Examples: x86 IR, ARM IR, MIPS IR
  - Actual instructions for a concrete machine ISA
  - Need a machine-level IR for each machine ISA
  - Purpose
    - Machine-specific optimizations (e.g. strength reduction)
    - Register allocation / code generation

- But could just use one IR (high-level IR) — some simple compilers do this

- Why multiple IRs?
  - Better to have an appropriate IR for the task at hand
  - Machine / language independent IRs enable reuse of code
    - Low-level IR optimizations reused regardless of language / machine
    - Machine IR code generation reused regardless of language
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)

---

Why this form?

- Allows all operations to be handled in a uniform way
- Modifications to IR can be done much more easily
  (Optimizations don’t worry about syntactic structure)
Example

An example:

\[ x \times y + z \div w \]

is translated to

\[
\begin{align*}
t1 &= x \times y &; t1, t2, t3 \text{ are temporary variables} \\
t2 &= z \div w \\
t3 &= t1 + t2
\end{align*}
\]

- Sequential translation of an AST
- Internal nodes in AST are translated to temporary variables
- Can be generated through a depth-first traversal of AST
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:
\[
\text{if (} x \ \text{relop} \ y \ \text{)} \ \text{goto} \ L
\]
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, foo(x1, x2, x3) is translated to
\[
\text{param} \ x_1 \\
\text{param} \ x_2 \\
\text{param} \ x_3 \\
call \ foo, \ 3
\]

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 scalable 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

```plaintext
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:

- `x = a + b` => `+ a, b, x`
- `x = - y` => `- y, , x`
- `goto L` => `goto , , L`
To avoid putting temporaries into the symbol table, we can refer to temporaries by the positions of the instructions that compute them.

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>
Triples for array statements

\[ x[i] = y \]

is translated to

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

That is, one statement is translated to two triples.
## Using Indirect Triples

### Problem with triples

- Cannot move code around because instruction numbers will change

<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

- Cannot move code around because instruction numbers will change

<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>
</tbody>
</table>
Using Indirect Triples

- IR is a listing of pointers to triples instead of triples themselves
- Triples are stored in a separate triple 'database'
- Can modify listing as long as the database does not change
- Slightly more overhead but allows optimizations

<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

<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

**Indirect Triples**

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

**Triple Database**

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>-c</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>*b</td>
<td>(0)</td>
</tr>
<tr>
<td>(2)</td>
<td></td>
<td>(empty)</td>
</tr>
<tr>
<td>(3)</td>
<td>(storing new triple)</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>+</td>
<td>(1)</td>
</tr>
<tr>
<td>(5)</td>
<td>=</td>
<td>a</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.
Developed by R. Cytron, J. Ferrante, et al. in 1980s

- Every variable is assigned exactly once i.e. one DEF
- Convert original variable name to name_{version}
  e.g. \( x \rightarrow x_1, x_2 \) in different places
- Use \( \phi \)-function to combine two DEFs of same original variable on a control flow merge

\[
\begin{align*}
x &= a + b; \\
y &= x - c; \\
x &= x - y; \\
\text{if} \ (\ldots) \\
x &= x + 5; \\
y &= x * 4;
\end{align*}
\]

\[
\begin{align*}
x_1 &= a + b; \\
y_1 &= x_1 - c; \\
x_2 &= x_1 - y_1; \\
\text{if} \ (\ldots) \\
x_3 &= x_2 + 5; \\
x_4 &= x_2 * 4;
\end{align*}
\]

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

- SSA can assist compiler optimizations
  - e.g. remove dead code
  
  \[
  \begin{align*}
  x &= a + b; \\
  x &= c - d; \\
  y &= x \ast b;
  \end{align*}
  \]

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

  .... $x_1$ 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**
Our next task is to translate **language constructs** to IR using **syntax directed translation scheme**

- What is our parsing scheme?
  - Bottom-up LR/LALR parsing
    - Natural to translate synthesized attributes
    - Hack to translate L-attributed inherited attributes
Generating IR

Our next task is to translate **language constructs** to IR using **syntax directed translation scheme**

- What is our parsing scheme?
  - Bottom-up LR/LALR parsing
    - Natural to translate synthesized attributes
    - Hack to translate L-attributed inherited attributes
Generating IR

Our next task is to translate **language constructs** to IR using **syntax directed translation scheme**

- **What is our parsing scheme?**
  - Bottom-up LR/LALR parsing
    - Natural to translate synthesized attributes
    - Hack to translate L-attributed inherited attributes

- **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 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;
}
```
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;
}
```

<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 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;
}
```

<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;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>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>

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

```c
#include <stdio.h>
#include <stdlib.h>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>0x0004</td>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>0x0008</td>
<td>c</td>
<td>8</td>
</tr>
<tr>
<td>0x000c</td>
<td>d</td>
<td>12</td>
</tr>
<tr>
<td>0x0010</td>
<td></td>
<td>16</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)`

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

```
<table>
<thead>
<tr>
<th>Address</th>
<th>0x0000</th>
<th>0x0004</th>
<th>0x0008</th>
<th>0x000c</th>
<th>0x0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Offset=16
- `Addr(a) ← 0`
- `Addr(b) ← 4`
- `Addr(c) ← 8`
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>Variable</th>
<th>Address</th>
<th>Offset</th>
<th>Address Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0x0000</td>
<td>0</td>
<td>Addr(a) ← 0</td>
</tr>
<tr>
<td>b</td>
<td>0x0004</td>
<td>4</td>
<td>Addr(b) ← 4</td>
</tr>
<tr>
<td>c</td>
<td>0x0008</td>
<td>8</td>
<td>Addr(c) ← 8</td>
</tr>
<tr>
<td>c</td>
<td>0x000C</td>
<td>16</td>
<td>Addr(d) ← 16</td>
</tr>
<tr>
<td>d</td>
<td>0x0010</td>
<td>20</td>
<td>Offset = 20</td>
</tr>
</tbody>
</table>
More About Storage Layout (I)

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 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 */
}
```
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

![Diagram showing memory and register layout for Big-endian storage.](attachment:image.png)
Endianness

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

![Diagram showing big-endian and little-endian storage layout.](image-url)
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; }
```
Need scope information for each level of nesting.

When encountering a nested procedure declaration...

1. Create a new symbol table when encountering a sub-procedure declaration
   - `mktable(ptr);` — ptr points back to its parent table

2. Store procedure name in parent symbol table, with a pointer pointing to the new table
   - `enterproc(parent_table_ptr, proc_id, child_table_ptr)`

3. Suspend the processing of parent symbol table
   - Push new table in the active symbol table stack
   - Push the current offset in the offset stack

4. When done, resume the processing of parent symbol table
   - Pop entries in active symbol table stack, offset stack for nested procedure
   - Restore current offset from the offset stack
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 Table</th>
<th>Offset Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>8</td>
</tr>
<tr>
<td>nil</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td></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>P1</th>
<th>Offset Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>8</td>
</tr>
</tbody>
</table>

Stack

P1
- q
- nil
- a
- b
- P2
void P1() {
    int a;
    int b;
    
    void P2() {
        int q;
        
        void P3() {
            void P4() {
                use a
            }
            int J;
        }
        
        check point #1
    }
    
    check point #2
    
    use q
}

void P2() {
    int q;
}

void P3() {
    int J;
}

void P4() {
    use a
}

check point #3

symbol table stack

offset

stack

P1
nil
a
b
P2
q
...
P3
Nested Declaration Example

```c
void P1() {
    int a;
    int b;

    void P2() {
        int q;
    }

    void P3() {
        int J;
    }

    void P4() {
        use a
    }
}

use q

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

Symbol Table Stack

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

Stack

P1
| nil |
| ... |

P2
| q   |
| ... |

P3
| J   |
| ... |

P4
| ... |
```
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
        }
        use q
        check point #4
    }
    int J;
    check point #5
}

Symbol Table Stack Offset Stack

P1 | nil | 8
P2 | q   |
    | ... |
P3 | P2  |
P4 |
    | P3  |
    | P4  |
    | ... |

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

\[
P \rightarrow M \ D \\
M \rightarrow \varepsilon \\
D \rightarrow D_1; D_2 \\
D \rightarrow \text{void pid() \{ N D_1; S \}} \\
D \rightarrow T \ id; \\
N \rightarrow \varepsilon
\]

\[
\{ \text{pop(tblptr); pop(offset);} \} \\
\{ t=\text{mktable(nil); push(t, tblptr); push(0, offset);} \} \\
\{ t=\text{top(tblptr); pop(tblptr); pop(offset); enter proc(top(tblptr), pid, t);} /* new symbol table */ \} \\
\{ \text{enter(top(tblptr), id, T.type, top(offset));} \\
\text{top(offset) = top(offset) + T.width; } \} \\
\{ \text{t=mktable(top(tblptr));} \\
\text{push(t, tblptr); push(0, offset);} \}
\]
Processing Statements

- Statements are processed sequentially after processing declarations
  
  - useful 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

- 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 \text{a}_1 = i_1
  2-dimension: A_2 = a_2 \times \text{width} \quad \text{a}_2 = \text{a}_1 \times \text{N}_2 + i_2
  3-dimension: A_3 = a_3 \times \text{width} \quad \text{a}_3 = \text{a}_2 \times \text{N}_3 + i_3
  ...
  k-dimension: A_k = a_k \times \text{width} \quad \text{a}_k = \text{a}_{k-1} \times \text{N}_k + i_k
Processing Array References

- Processing an array assignment (e.g. A[i] = B[j];)

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

- where op can be <, >, >=, &&, ||, ...

1. Compute just like any other arithmetic expression

- Good for languages with no *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

- Without short circuiting, entire expression is evaluated as usual

\[ S \rightarrow \text{id} = E \quad \equiv \quad \text{lookup(id)} = E\text{.place} \]
\[ E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f) \quad \equiv \quad t_1 = a < b \]
\[ t_2 = c < d \]
\[ t_3 = e < f \]
\[ t_4 = t_2 \text{ && } t_3 \]
\[ E\text{.place} = t_1 \text{ || } t_4 \]
Processing Boolean Expressions

2. Implement as a series of jumps

- For languages with short circuiting (e.g. C/C++), evaluations sometimes have to be 'jumped'
- Processing a boolean expression:
  \[ S \rightarrow \text{if } E \text{ then } S1 \]
  \[ E \rightarrow a < b \quad \equiv \quad E\.true = S1\.label; \]
  \[ E\.false = S\.next \]
  \[ \text{if } E \text{ goto } E\.true \]
  \[ \text{goto } E\.false \]

  S1.label: label created at the address of code S1
  S.next: address of code after S
  E.true: code to execute on 'true'
  E.false: code to execute on 'false'

- Processing compound boolean expressions:
  - Chain together multiple of above by updating E.true/E.false
  - \[ E \rightarrow E1 \&\& E2: E1\.true = \text{code for } E2, \ E1\.false = S\.next \]
  - \[ E \rightarrow E1 \| E2: E1\.false = \text{code for } E2, \ E1\.true = S1\.label \]
2. Implement as a series of jumps (cont’d)

- A short circuited compound boolean expression
  \[ E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f) \equiv \]
  
  - `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`

- Can apply to other control flow statements
  \[ S \rightarrow \text{if } E \text{ then } S1 \mid \text{if } E \text{ then } S1 \text{ else } S2 \mid \text{while } E \text{ do } S1 \]

- **Problem:** E.true, E.false, S.next are non-L-attributes
  - Depend on code that has not been generated yet
    - S.next: Only available when code after S is generated
    - E.true: Only available when S1 is generated
A non-L-attributed grammar may preclude a one pass syntax directed translation scheme
- Both top-down and bottom-up SDTS rely on L-attributed grammars

How to handle non-L attributes?
- `E.true`, `E.false`, `S.next`

Solutions: two methods
- Two pass approach — process the code twice
  - Generate labels in the first pass
  - Replace labels with addresses in the second pass
- One pass approach
  - Generate holes when address is needed but unknown
  - Fill in holes when addresses is known later on
  - Finish code generation in one pass
Attributes for two pass based approach

- **Expression E**
  - Synthesized attributes: \( E.\text{code} \)
  - non-L inherited attributes: \( E.\text{true}, E.\text{false} \)

- **Statement S**
  - Synthesized attributes: \( S.\text{code} \)
  - non-L inherited attributes: \( S.\text{next} \)

Evaluation order:

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

1. Generate \( E.\text{code} \) and \( S_1.\text{code} \) making a label for \( E.\text{true} \)
2. Replace label \( E.\text{true} \) with actual address of \( S_1 \)
   (Labels \( E.\text{false} \) and \( S_1.\text{next} \) are inherited from \( S.\text{next} \))

Given rule \( S \rightarrow S_1 \text{ S}_2 \), the two passes are:

1. Generate \( S_1.\text{code} \) and \( S_2.\text{code} \) making a label for \( S_1.\text{next} \)
2. Replace label \( S_1.\text{next} \) with the actual address of \( S_2 \)
   (Label \( S_2.\text{next} \) is inherited from \( S.\text{next} \))
Two Pass based Rules

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

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

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

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

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

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

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

Write SDT rule (two pass) for the following statement

\[ S \rightarrow \text{while (}a<b\text{) do} \]
\[ \quad \text{if (}c<d\text{) then } S \]
\[ \quad \text{endif} \]
\[ \text{endwhile} \]
Backpatching

- If grammar contains L-attributes only, then it can be processed in one pass

- However, we know there are occasions for non-L attributes
  - Example: E.true, E.false, S.next during code generation
  - Is there a general solution to this problem?

Solution:
- Leave holes for non-L attributes, record their locations in holelists, and fill in the holes when we know the target values
  - holelist: 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 replaced at the end of code generation
Attributes for two pass based approach

- Expression \( E \)
  — Synthesized attributes: \( E.\text{code} \)
  \( E.\text{holes\_truelist} \), and \( E.\text{holes\_falselist} \)

- Statement \( S \)
  — Synthesized attributes: \( S.\text{code} \) and \( S.\text{holes\_nextlist} \)

Evaluation order:

Given rule \( S \rightarrow \text{if } E \text{ then } S_1 \), below is done in one-pass:

1. Gen \( E.\text{code} \), making \( E.\text{holes\_truelist} \), \( E.\text{holes\_falselist} \)
2. Gen \( S_1.\text{code} \), filling in \( E.\text{holes\_truelist} \) and merging \( S_1.\text{holes\_nextlist} \) with \( E.\text{holes\_falselist} \)
3. Pass on merged list to \( S.\text{holes\_nextlist} \)

Given rule \( S \rightarrow S_1 S_2 \), below is done in one-pass:

1. Gen \( S_1.\text{code} \) making \( S_1.\text{holes\_nextlist} \)
2. Gen \( S_2.\text{code} \) filling in \( S_1.\text{holes\_nextlist} \) and making \( S_2.\text{holes\_nextlist} \)
3. Pass on \( S_2.\text{holes\_nextlist} \) to \( S.\text{holes\_nextlist} \)
3 functions for implementing backpatching for IR generation

- makelist(i) — creates a new list with statement index i
- merge(p1, p2) — concatenates list p1 and list p2
- backpatch(p, i) — insert i as target label for each statement in list p

\[
E \rightarrow E_1 \text{ or } M \ E_2 \\
\quad \quad \{ \text{backpatch}(E_1.\text{holes}_\text{falselist}, M.\text{quad}); \} \\
\quad \quad E.\text{holes}_\text{truelist} = \text{merge}(E_1.\text{holes}_\text{truelist}, E_2.\text{holes}_\text{truelist}); \}
\]

\[
E \rightarrow E_1 \text{ and } M \ E_2 \\
\quad \quad \{ \text{backpatch}(E_1.\text{holes}_\text{truelist}, M.\text{quad}); \}
\quad \quad E.\text{holes}_\text{falselist} = \text{merge}(E_1.\text{holes}_\text{falselist}, E_2.\text{holes}_\text{falselist}); \}
\quad \quad E.\text{holes}_\text{truelist} = E_2.\text{holes}_\text{truelist}; \}
\]

\[
M \rightarrow \varepsilon \\
\quad \quad \{ M.\text{quad} = \text{nextquad}; \}
\]
More One Pass SDT Rules

\[ E \rightarrow \text{not } E1 \]
\[ \{ \text{E.holes_truelist} = \text{E1.holes_falselist}; \]
\[ \text{E.holes_falselist} = \text{E1.holes_truelist}; \} \]

\[ E \rightarrow (E1) \]
\[ \{ \text{E.holes_truelist} = \text{E1.holes_truelist}; \]
\[ \text{E.holes_falselist} = \text{E1.holes_falselist}; \} \]

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

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

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

E1.hole_truelist=(100)  
E1.hole_falselist=(101)

E2.hole_truelist=(102)  
E2.hole_falselist=(103)

E3.hole_truelist=(104)  
E3.hole_falselist=(105)
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 ___
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 ___

Are we done?
Backpatching Example (cont.)

- 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 ___

- Are we done?
  ➢ Yes for this expression
E1.t=(100)  
E1.f=(101)  

or  

M1.quad=102

E4.t=(104)  
E4.f=(103,105)  

and  

M2.quad=104

E3.t=(104)  
E3.f=(105)  

ea < b

c < d

e < f
Problem

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

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

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

\[
\text{if } E2
\]

\[
\text{then } S2
\]

\[
\text{endif}
\]

\[
\text{endwhile}
\]
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>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>Two Pass</td>
<td></td>
</tr>
<tr>
<td>E1.hole_truelist</td>
<td>(E1.hole_truelist,E1.hole_falselist)</td>
</tr>
<tr>
<td>E1.hole_falselist</td>
<td>(E2.hole_truelist,E2.hole_falselist)</td>
</tr>
<tr>
<td>One Pass</td>
<td></td>
</tr>
<tr>
<td>E1.code, E1.hole_truelist</td>
<td>S.code</td>
</tr>
<tr>
<td>E1.hole_falselist</td>
<td>S.hole_nextlist</td>
</tr>
<tr>
<td>E2.code, E2.hole_truelist</td>
<td></td>
</tr>
<tr>
<td>E2.hole_falselist</td>
<td></td>
</tr>
<tr>
<td>S.code, S.hole_nextlist</td>
<td></td>
</tr>
</tbody>
</table>