Code Generation
Modern Compiler Project

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

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

- C# program
  - C#'s Lexer, Parser, and Static Checker
  - Intermediate Code Generator
  - ARM Code Generator
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
  - Compiler optimizations much easier with low-level IR
  - Register allocation much easier with machine-level IR

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

- Why multiple IRs?
  - Better to have an appropriate IR for the task at hand
    - Compiler optimizations much easier with low-level IR
    - Register allocation much easier 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
Example

An example:

\[ \text{x * y + x * y} \]

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:

if (x relop y) goto L

where relop is a relational operator such as =, ≠, >, <

Procedural call statement:

param x_1, ..., param x_n, call F_y, n

As an example, foo(x1, x2, x3) is translated to

param x_1
param x_2
param x_3
call foo, 3

Procedural call return statement:

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:

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

- 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:

- \( x = a + b \) => + a, b, x
- \( x = -y \) => - y, , x
- goto L => goto , , L
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

What if result field stores into a variable memory location instead of a temporary?

Example: triples for array assignment statement
\[ x[i] = y \]
is translated to
\[
(0) \ [x] i \\
(1) = (0) y
\]

That is, one statement is translated to two triples
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>
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>
</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 \( \text{version} \)
    - e.g. \( x \rightarrow x_1, x_2 \) for each distinct assignment of \( x \)
  - Same version is guaranteed to contain same value
  - On a control flow merge, use \( \phi \)-function to combine two versions of same variable

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

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

\[
\begin{align*}
x_3 &= x_2 + 5; \\
x_4 &= x_2 * 4; \\
x_5 &= \phi(x_3, x_4); \\
y_2 &= x_5 * 4;
\end{align*}
\]
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)

```
x = a + b;
x = c - d;
y = x \times b;
```

```
x_1 = a + b;
x_2 = c - d;
y_1 = x_2 \times b;
```

.... $x_1$ is defined but never used, it is safe to remove

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

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

---

**Generating IR**

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**
  - \texttt{S.code} — a synthesized attribute that holds IR code of S

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

- **Variable declaration:**
  - \texttt{T V} e.g. \texttt{int a,b,c};
  - Type information \texttt{T.type, T.width}
  - Variable information \texttt{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
- use variable `offset`, to get address of `x`

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

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

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

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

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

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

- we layout the variable sequentially
- use variable \texttt{offset}, to get address of \texttt{x}

- \texttt{address(x) \leftarrow offset}
- \texttt{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>
</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 \texttt{x}
  - address(\texttt{x}) $\leftarrow$ offset
  - offset $\leftarrow$ sizeof(\texttt{x}.type)

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset=8</th>
<th>Addr(a)$\leftarrow$0</th>
<th>Addr(b)$\leftarrow$4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x000c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0010</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 a variable `offset`, to get the address of a variable. Here's an example:

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

The storage layout for the variables would look like this:

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0</td>
<td>0x0004</td>
<td>4</td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
<td>0x0010</td>
<td></td>
</tr>
</tbody>
</table>

where `Addr(a) ← 0`, `Addr(b) ← 4`, and `Addr(c) ← 8`.
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>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>a</td>
</tr>
<tr>
<td>0x0004</td>
<td>b</td>
</tr>
<tr>
<td>0x0008</td>
<td>c</td>
</tr>
<tr>
<td>0x000c</td>
<td>c</td>
</tr>
<tr>
<td>0x0010</td>
<td>d</td>
</tr>
</tbody>
</table>

- `Addr(a) ← 0`
- `Addr(b) ← 4`
- `Addr(c) ← 8`
- `Addr(d) ← 16`

Offset = 20
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($x.\text{type}$) boundaries
  - E.g. Most machines are designed to load integer values at integer word boundaries
  - If not on word boundary, need to load two words and shift & concatenate

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

Memory

<table>
<thead>
<tr>
<th>...</th>
<th>31.........................0</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:</td>
<td>0A</td>
<td>0A 0B 0C 0D</td>
</tr>
<tr>
<td>a+1:</td>
<td>0B</td>
<td></td>
</tr>
<tr>
<td>a+2:</td>
<td>0C</td>
<td></td>
</tr>
<tr>
<td>a+3:</td>
<td>0D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Big-endian
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 diagram]

![Little-endian diagram]
More About Storage Layout (III)

Questions still unanswered

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

Translating the declaration in a single procedure

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

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

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

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

use q

Symbol Table Stack

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>
</tbody>
</table>

...
Nested Declaration Example

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

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

void P3() {
    int J;
}

check point #3

use a

Symbol Table Stack Offset
Table Stack Stack

P1
a 0
b 4
P2
...  

P2
q 0  
...  

P3
...  

...
Nested Declaration Example

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

check point #4

Symbol Table Stack Offset 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>
<tr>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td></td>
</tr>
</tbody>
</table>

P1
a 0
b 4

P2
q 0

P3

P4

...
Nested Declaration Example

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

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

Symbol Table Stack
Offset Stack

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></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 #5
Processing Nested Declarations

Syntax directed translation rules

\[
P \rightarrow M \ D & \quad \{ \ \text{pop}(\text{active}); \ \text{pop}(\text{offset}); \ \} \\
M \rightarrow \varepsilon & \quad \{ \ t=\text{mktable}(); \ \text{push}(t, \text{active}); \ \text{push}(0,\text{offset}); \ \} \\
D \rightarrow D; \ D & \\
D \rightarrow \text{void } \text{pid()} \ { N \ D; \ S } & \quad \{ \ \text{pop}(\text{active}); \ \text{pop}(\text{offset}); \\
& \quad \text{enterproc}(\text{top}(\text{active}), \ \text{pid}, \ \text{pid\_addr}); \ \} \\
N \rightarrow \varepsilon & \quad \{ \ t=\text{mktable}(); \ \text{push}(t, \text{active}); \ \text{push}(0, \ \text{offset}); \\
D \rightarrow \text{T } \text{id;} & \quad \{ \ \text{enter}(\text{top}(\text{active}), \ \text{id}, \ \text{T.type}, \ \text{top}(\text{offset})); \\
& \quad \text{top}(\text{offset}) = \text{top}(\text{offset})+ \ \text{T.width}; \ \} \\
\]
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 $\rightarrow$ id = E  
E $\rightarrow$ E1 + E2  
E $\rightarrow$ E1 * E2  
E $\rightarrow$ - E1  
E $\rightarrow$ ( E1 )  
E $\rightarrow$ id  

{ P=lookup(id); if (P==nil) perror(...); else emit(P ’=’ E.place); }
{ E.place = newtemp(); emit(E.place ’=’ E1.place ’+’ E2.place); }
{ E.place = newtemp(); emit(E.place ’=’ E1.place ’*’ E2.place); }
{ E.place = newtemp(); emit(E.place ’=’ ’-’ E1.place); }
{ E.place = E1.place; }
{ 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 * width \quad a_1 = i_1
2-dimension: A_2 = a_2 * width \quad a_2 = a_1 * N_2 + i_2
3-dimension: A_3 = a_3 * width \quad a_3 = a_2 * N_3 + i_3
...
k-dimension: A_k = a_k * width \quad a_k = a_{k-1} * N_k + i_k
Processing Array References

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

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

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

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

\[
L \rightarrow L1[E] \quad \begin{cases}
    L\.base = L1.base; \\
    L\.width = L1.width; \\
    L\.dim = L1\.dim + 1; \\
    L\.place = \text{newtemp}(); \\
    \text{emit}(L\.place \ ']=' L1\.place \ '*' L\.bounds[L\.dim]); \\
    \text{emit}(L\.place \ ']=' L\.place \ '+' E\.place);
\end{cases}
\]
Processing Boolean Expressions

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

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

   - Computed just like any other arithmetic expression:
     \[ E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f) \equiv \]
     \[ t_1 = a < b \]
     \[ t_2 = c < d \]
     \[ t_3 = e < f \]
     \[ t_4 = t_2 \&\& t_3 \]
     \[ t_5 = t_1 \|\| t_4 \]
Processing Boolean Expressions

2. Languages with short circuiting

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

  - 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
  \[
  S \rightarrow \text{if } E \text{ then } S1 \mid \text{if } E \text{ then } S1 \text{ else } S2 \mid \text{while } E \text{ do } S1
  \]
Processing Boolean Expressions

2. Languages with short circuiting (cont’d)

- SDD for if statement:
  \[ E \rightarrow \text{id1 relop id2} \{ \begin{align*}
  E.\text{code} &= \text{emit(‘if’ id1 ‘relop’ id2 ‘goto’ E.true)} \mid \\
  &\quad \text{emit(‘goto’ E.false);} \\
  E.\text{true} &= \text{address of code to execute on ‘true’ (inherited)} \\
  E.\text{false} &= \text{address of code to execute on ‘false’ (inherited)} \\
  S &= \text{if E then S1} \{ \begin{align*}
  E.\text{true} &= S1.\text{label}; \ E.\text{false} &= S.\text{next}; \\
  S.\text{code} &= E.\text{code} \mid \text{emit(S1.\text{label}:’)} \mid S1.\text{code}; \}
  \end{align*} \}
  \end{align*} \]

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

➢ SDD for && and ||:

\[ E \rightarrow E_1 \&\& E_2 \quad \{ \text{E1.true = E2.label; E1.false = E.false; } \]
\[ \text{E.code = E1.code || emit(E2.label’:) || E2.code; } \}

\[ E \rightarrow E_1 \| E_2 \quad \{ \text{E1.false = E2.label, E1.true = E.true; } \]
\[ \text{E.code = E1.code || emit(E2.label’:) || E2.code; } \}

➢ Problem: \textbf{E1.true, E1.false are non-L-attributes}

- \textbf{E1.true:} Address of \textbf{E2.label} known only when \textbf{E2} emitted
- \textbf{E1.false:} Address of \textbf{E2.label} known only when \textbf{E2} 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
Two-Pass Based Syntax Directed Translation Scheme

Attributes for two pass based approach

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

- Statement $S \rightarrow S_1 \ S_2$
  - inherited attributes: $S_2.\text{next}$
  - non-L inherited attributes: $S_1.\text{next}$

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

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

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

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

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

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

More Two Pass based SDT Rules

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

\[
E \rightarrow E_1 \ \text{or} \ E_2 & \quad \{ \ \text{E1.true = E2.true = E.true;} \\
& \quad \text{E1.false = newlabel;} \\
& \quad \text{E2.false = E.false;} \\
& \quad \text{E.code = E1.code || emit(E1.false ':') || E2.code;} \ \} \\
\]

\[
E \rightarrow E_1 \ \text{and} \ E_2 & \quad \{ \ \text{E1.false = E2.false = E.false;} \\
& \quad \text{E1.true = newlabel;} \\
& \quad \text{E2.true = E.true;} \\
& \quad \text{E.code = E1.code || emit(E1.true ':') || E2.code;} \ \} \\
\]

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

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

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

Write SDT rule (two pass) for the following statement

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

However, **we know** non-L attributes are necessary.
- Example: $E1.\text{false in } E \rightarrow E1 \ || \ E2$
- 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).
One-Pass Based Syntax Directed Translation Scheme

Attributes for one-pass based approach

- Expressions $E \rightarrow E_1 || E_2$, $E \rightarrow \text{id1 \ relop \ id2}$ ...
  --- 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 $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ ε

{ M.quad = nextquad; }

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

$$E \rightarrow \text{true} \quad \{ \text{E.holes_true} = \text{makelist(nextquad)}; \}
\text{emit(‘goto ___’);} \}

$$E \rightarrow \text{false} \quad \{ \text{E.holes_false} = \text{makelist(nextquad)}; \}
\text{emit(‘goto ___’);} \}

$$E \rightarrow \text{id1 relop id2} \quad \{ \text{E.holes_true} = \text{makelist(nextquad)}; \}
\text{E.holes_false} = \text{makelist(nextquad+1)}; \}
\text{emit(‘if’ id1.place ‘relop’ id2.place ‘goto ___’);} \}
\text{emit(‘goto ___’);} \}

$$E \rightarrow \text{not E1} \quad \{ \text{E.holes_true} = \text{E1.holes_false}; \}
\text{E.holes_false} = \text{E1.holes_true}; \}

$$E \rightarrow \text{(E1)} \quad \{ \text{E.holes_true} = \text{E1.holes_true}; \}
\text{E.holes_false} = \text{E1.holes_false}; \}$
Backpatching Example

- $E \rightarrow (a < b)$ or $M_1$ ($c < d$ and $M_2$ $e < f$)

- When reducing $(a < b)$ to $E_1$, we have
  100: if($a < b$) goto ___
  101: goto ___

  $E_1.holes_true = (100)$
  $E_1.holes_false = (101)$

- When reducing $\varepsilon$ to $M_1$, we have
  $M_1.quad = 102$

- When reducing $(c < d)$ to $E_2$, we have
  102: if($c < d$) goto ___
  103: goto ___

  $E_2.holes_true = (102)$
  $E_2.holes_false = (103)$

- When reducing $\varepsilon$ to $M_2$, we have
  $M_2.quad = 104$

- When reducing $(e < f)$ to $E_3$, we have
  104: if($e < f$) goto ___
  105: goto ___

  $E_3.holes_true = (104)$
  $E_3.holes_false = (105)$
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 ___
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 SDT rule (one pass using backpatching) for the following statement

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

- $S \rightarrow \text{while } E1 \text{ do if } E2 \text{ then } S2 \text{ endif endwhile}$

<table>
<thead>
<tr>
<th>Two Pass</th>
<th>Known Attributes</th>
<th>Attributes to Evaluate/Process</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>
</tbody>
</table>

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