Formal Languages and Compilers
Lecture X—Intermediate Code Generation

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science – POS Building, Room: 2.03
artale@inf.unibz.it
http://www.inf.unibz.it/~artale/

Formal Languages and Compilers — BSc course

2019/20 – Second Semester
Summary of Lecture X

- Three-Address Code
- Code for Assignments
- Boolean Expressions and Flow-of-Control Statements
An intermediate code is generated as a program for an abstract machine.

1. The intermediate code should be easy to translate into the target program.
2. A machine-independent Code Optimizer can be applied before generating the target code.

As intermediate code we consider the three-address code, similar to assembly: sequence of instructions with at most three operands such that:

1. There is at most one operator, in addition to the assignment (we make explicit the operators precedence).
2. The general form is: \( x := y \ op \ z \)
   where \( x, y, z \) are called addresses, i.e., either identifiers, constants or compiler-generated temporary names.

   - Temporary names must be generated to compute intermediate operations.
   - Addresses are implemented as pointers to their symbol-table entries.
Types of Three-Address Statements

Three-Address statements are akin to assembly code: Statements can have *labels* and there are statements for flow-of-control.

1. **Assignment Statements**: \( x := y \ op \ z \).
2. **Unary Assignment Statements**: \( x := \ op \ y \).
3. **Copy Statements**: \( x := y \).
4. **Unconditional Jump**: \( \text{goto L} \), with \( L \) a label of a statement.
5. **Conditional Jump**: \( \text{if x relop y goto L} \).
6. **Procedure Call**: param x, and call p,n for calling a procedure, p, with n parameters. With return y the returned value of the procedure is indicated.

```
param x1
param x2
...
param xn
call p,n
```

7. **Indexed assignments**: x := y[i] or x[i] := y.

**Note**: x[i] denotes the value in the location i memory units beyond the location x.

8. **Pointer Assignments**: x := &y, x := *y, or *x := y; where &y stands for the address of y, and *y for the value stored at y.
Summary

- Three-Address Code
- Code for Assignments
- Boolean Expressions and Flow-of-Control Statements
The following S-attributed definition generates three-address code for assignments.

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} ::= E$</td>
<td>$p := \text{lookup(id.name)}$; if $p \neq \text{nil}$ then $\text{emit}(p ::= E.addr)$ else error</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>$E.addr := \text{newtemp}()$; $\text{emit}(E.addr ::= E_1.addr + E_2.addr)$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 \times E_2$</td>
<td>$E.addr := \text{newtemp}()$; $\text{emit}(E.addr ::= E_1.addr \times E_2.addr)$</td>
</tr>
<tr>
<td>$E \rightarrow -E_1$</td>
<td>$E.addr := \text{newtemp}()$; $\text{emit}(E.addr ::= \text{'uminus' } E_1.addr)$</td>
</tr>
<tr>
<td>$E \rightarrow (E_1)$</td>
<td>$E.addr := E_1.addr$</td>
</tr>
<tr>
<td>$E \rightarrow \text{id}$</td>
<td>$p := \text{lookup(id.name)}$; if $p \neq \text{nil}$ then $E.addr := p$ else error</td>
</tr>
<tr>
<td>$E \rightarrow \text{num}$</td>
<td>$E.addr := \text{newtemp}()$; $E.addr := \text{num.val}$</td>
</tr>
</tbody>
</table>
The function `emit()` output to a file a three-address code such that:

1. Everything quoted is taken literally;
2. The rest is evaluated.

Temporary names are generated for intermediate computations.

- The function `newtemp()` generates distinct temporary names $t_1, t_2, \ldots$.

Expressions have a synthesized attribute:

- $E.addr$: Temporary name holding the value of $E$;

Names/addresses stand for pointers to their symbol table entries: other info are needed for the final code generation (in particular, the storage address).

- **Note.** Under this assumption Temporary Names must be also entered into the symbol table as they are created by the `newtemp()` function.

The function `lookup(id.name)` return `nil` if the entry is not found in the symbol table, otherwise a pointer to the entry is returned.

- The `lookup(id.name)` can be easily modified to account for scope: If `name` does not appear in the current symbol table the enclosing symbol table is checked (see the Lecture on “Symbol Table”).
Given the assignment \( a := b \times -c + d \) the code generated by the above grammar is:

\[
S \\
\quad := \ E t_3 \\
\quad \ E \ t_2 \ + \ E \\
\quad \ E \ * \ E \ t_1 \ id \\
\quad \ E \ - \ id \\
\quad \ E \ b \\
\quad \ E \ c
\]

\[
t_1 := \text{uminus} \ c \\
t_2 := b \times t_1 \\
t_3 := t_2 + d \\
a := t_3
\]
Summary

- Three-Address Code
- Code for Assignments
- Boolean Expressions and Flow-of-Control Statements
Boolean Expressions are used to either compute logical values or as conditional expressions in flow-of-control statements.

We consider Boolean Expressions with the following grammar:

\[ E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid E \text{ relop } E \mid \text{true} \mid \text{false} \]

There are two methods to evaluate Boolean Expressions:

1. **Numerical Representation.** Encode true with '1' and false with '0' and we proceed analogously to arithmetic expressions.
2. **Jumping Code.** We represent the value of a Boolean Expression by a position reached in a program.
Expressions will be evaluated from left to right assuming that: or and and are left-associative, and that or has lowest precedence, then and, and finally not.

**Example 1.** The translation for “a or (b and (not c))” is:

\[
\begin{align*}
  t_1 & := \text{not } c \\
  t_2 & := b \text{ and } t_1 \\
  t_3 & := a \text{ or } t_2
\end{align*}
\]

**Example 2.** A relational expression such as \(a < b\) is equivalent to the conditional statement \(\text{if } a < b \text{ then } 1 \text{ else } 0\). Its translation involves jumps to labeled statements:

\[
\begin{align*}
  100: & \quad \text{if } a < b \text{ goto } 103 \\
  101: & \quad t := 0 \\
  102: & \quad \text{goto } 104 \\
  103: & \quad t := 1 \\
  104: & \quad
\end{align*}
\]
Numerical Representation: The Translation

The following S-Attributed Definition makes use of the global variable `nextstat` that gives the index of the next three-address code statement and is incremented by `emit`.

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
</table>
| $E \rightarrow E_1$ or $E_2$ | $E.addr := \text{newtemp}();$  
|                     | $\text{emit}(E.addr := E_1.addr \text{ or } E_2.addr)$                        |
| $E \rightarrow E_1$ and $E_2$    | $E.addr := \text{newtemp}();$  
|                     | $\text{emit}(E.addr := E_1.addr \text{ and } E_2.addr)$                        |
| $E \rightarrow \text{not } E_1$ | $E.addr := \text{newtemp}();$  
|                     | $\text{emit}(E.addr := \text{not } E_1.addr)$                                 |
| $E \rightarrow (E_1)$     | $E.addr := E_1.addr$                                                          |
| $E \rightarrow E_1 \text{ relop } E_2$ | $E.addr := \text{newtemp}();$  
|                     | $\text{emit}(\text{if } E_1.addr \text{ relop } E_2.addr \text{ goto} \text{ nextstat } + 3);$  
|                     | $\text{emit}(E.addr := \text{'}0\text{'});$                                  
|                     | $\text{emit}(\text{goto nextstat } + 2);$                                    |
|                     | $\text{emit}(E.addr := \text{'}1\text{'});$                                  |
| $E \rightarrow \text{true}$ | $E.addr := \text{newtemp}();$  
|                     | $\text{emit}(E.addr := \text{'}1\text{'});$                                  |
| $E \rightarrow \text{false}$ | $E.addr := \text{newtemp}();$  
|                     | $\text{emit}(E.addr := \text{'}0\text{'});$                                  |
The value of a Boolean Expression is represented by a position in the code.

Consider Example 2: We can tell what value $t$ will have by whether we reach statement 101 or statement 103.

Jumping code is extremely useful when Boolean Expressions are in the context of flow-of-control statements.

We start by presenting the translation for flow-of-control statements generated by the following grammar:

$$S \rightarrow \text{if } E \text{ then } S$$

$$| \text{if } E \text{ then } S_1 \text{ else } S_2$$

$$| \text{while } E \text{ do } S$$
In the translation, we assume that a three-address code statement can have a symbolic label, and that the function `newlabel()` generates such labels.

We associate with $E$ two labels using inherited attributes:

1. $E.true$, the label to which control flows if $E$ is true;
2. $E.false$, the label to which control flows if $E$ is false.

We associate to $S$ the inherited attribute $S.next$ that represents the label attached to the first statement after the code for $S$.

**Note 1.** This method of generating symbolic labels can lead to a proliferation of label: The backpatching method (see the Book) creates labels only when needed and emits directly the code.

**Note 2.** To substitute symbolic labels with actual addresses a second pass is needed: backpatching will avoid also the two-pass translation.
The following figures show how the flow-of-control statements are translated.

(a) if-then

(b) if-then-else

(c) while-do
### Production

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
</table>
| $P \rightarrow S$ | $S.next = \text{newlabel}();$
  
  $P.code := S.code \parallel \text{gen}(S.next \ ':');$
| $S \rightarrow \text{if } E \text{ then } S_1$ | $E.true := \text{newlabel}(); \quad E.false := S.next;$
  
  $S_1.next := S.next;$
  
  $S.code := E.code \parallel \text{gen}(E.true \ ':) \parallel S_1.code$
| $S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$ | $E.true := \text{newlabel}(); \quad E.false := \text{newlabel}();$
  
  $S_1.next := S.next; \quad S_2.next := S.next;$
  
  $S.code := E.code \parallel \text{gen}(E.true \ ':) \parallel S_1.code \parallel$
    
  $\text{gen}('\text{goto'} \ S.next) \parallel$
    
  $\text{gen}(E.false \ ':) \parallel S_2.code$
| $S \rightarrow \text{while } E \text{ do } S_1$ | $E.begin := \text{newlabel}();$
  
  $E.true := \text{newlabel}(); \quad E.false := S.next;$
  
  $S_1.next := E.begin;$
  
  $S.code := \text{gen}(E.begin \ ':) \parallel E.code \parallel$
    
  $\text{gen}(E.true \ ':) \parallel S_1.code \parallel$
    
  $\text{gen}('\text{goto'} \ E.begin)$
Translation Scheme

\[ P \rightarrow \{ S.next = \text{newlabel}(); \} \{ P.code := S.code \parallel \text{gen}(S.next' :') \} \]

\[ S \rightarrow \text{if } \{ E.true := \text{newlabel}(); \ E.false := S.next; \} \ E \text{ then} \]
\[ \{ S_1.next := S.next; \} \ S_1\{ S.code := E.code \parallel \text{gen}(E.true ' :') \parallel S_1.code \} \]

\[ S \rightarrow \text{if } \{ E.true := \text{newlabel}(); \ E.false := \text{newlabel}(); \} \ E \text{ then} \]
\[ \{ S_1.next := S.next; \} \ S_1 \text{ else } \{ S_2.next := S.next; \} \ S_2 \]
\[ \{ S.code := E.code \parallel \text{gen}(E.true ' :') \parallel S_1.code \]
\[ \text{gen('goto' S.next) \parallel gen(E.false ' :') \parallel S_2.code} \]
Boolean Expressions are translated in a sequence of conditional and unconditional jumps to either $E.true$ or $E.false$.

- $a < b$. The code is of the form:
  
  ```
  if $a < b$ goto $E.true$
  goto $E.false$
  ```

- $E_1 \lor E_2$. If $E_1$ is true then $E$ is true, so $E_1.true = E.true$. Otherwise, $E_2$ must be evaluated, so $E_1.false$ is set to the label of the first statement in the code for $E_2$.

- $E_1 \land E_2$. Analogous considerations apply.

- not $E_1$. We just interchange the true and false with that for $E$.

**Note.** Both the *true* and *false* attributes are inherited and the translation is an L-attributed grammar.
<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
</table>
| $E \rightarrow E_1 \text{ or } E_2$ | $E_1.true := E.true; \quad E_1.false := \text{newlabel}();$  
|                 | $E_2.true := E.true; \quad E_2.false := E.false;$  
|                 | $E.code := E_1.code \parallel \text{gen}(E_1.false ':') \parallel E_2.code;$ |
| $E \rightarrow E_1 \text{ and } E_2$ | $E_1.true := \text{newlabel}(); \quad E_1.false := E.false;$  
|                 | $E_2.true := E.true; \quad E_2.false := E.false;$  
|                 | $E.code := E_1.code \parallel \text{gen}(E_1.true ':') \parallel E_2.code;$ |
| $E \rightarrow \text{not } E_1$      | $E_1.true := E.false; \quad E_1.false := E.true;$  
|                 | $E.code := E_1.code;$                                  |
| $E \rightarrow (E_1)$            | $E_1.true := E.true; \quad E_1.false := E.false;$  
|                 | $E.code := E_1.code;$                                  |

(Follows $\rightarrow$)
<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
</table>
| $E \rightarrow E_1 \text{ relop } E_2$ | $E.code := E_1.code \parallel E_2.code$
| | gen('if' $E_1.addr$ relop.op $E_2.addr$ 'goto' $E.true$) $\parallel$
| | gen('goto' $E.false$) |
| $E \rightarrow \text{id}$ | $p = \text{lookup}(\text{id}.name)$;
| | if($p.type = \text{bool}$) then
| | $E.code := \text{gen}('if' \ p = \text{true} \ 'goto' \ E.true)$$\parallel$
| | gen('goto' $E.false$) |
| | else if($p \neq \text{nil}$) then
| | $E.addr = p; E.code ='$
| | else error |
| $E \rightarrow \text{true}$ | $E.code := \text{gen}('goto' E.true)$ |
| $E \rightarrow \text{false}$ | $E.code := \text{gen}('goto' E.false)$ |
Example. Translate the following statement:

while a < b do
    if c or d then
        x := y + z
    else
        x := y - z
Summary of Lecture X

- Three-Address Code
- Code for Assignments
- Boolean Expressions and Flow-of-Control Statements