Theory of Compilation
236360

Erez Petrank

Lecture 6:
Intermediate Representation and
Attribute Grammars
You are here

Compiler

Source text

Lexical Analysis

Syntax Analysis

Semantic Analysis

Rep. (IR)

Inter. Rep.

Code Gen.

Executable code

Characters

tokens

AST

(Annotated Syntax Tree)

Abstract Syntax Tree

You are here
Intermediate Representation

• “neutral” representation between the front-end and the back-end
  – Abstracts away details of the source language
  – Abstract away details of the target language
• Allows combination of various back-ends with various front-ends.

![Diagram showing various programming languages and platforms connected to a generic intermediate representation]

Languages: Java, C, Ada, Objective C
Platforms: arm, x86, ia64
Intermediate Representation(s)

- Annotated abstract syntax tree
- Three address code
- Postfix notation
- ...

- Sometimes we move between representations. E.g.,
  - Syntax directed translation produces an annotated AST,
  - AST translated to three address code.
Examples: Tree, DAG and Postfix

- Input: \( a := b \times (-c) + b \times (-c) \)

- Tree: DAG representation:

\[
\begin{align*}
\text{Tree:} & & \text{DAG representation:} \\
\text{assign} & & \text{assign} \\
\text{a} & & \text{a} \\
\text{+} & & \text{+} \\
\text{assign} & & \text{assign} \\
\text{a} & & \text{a} \\
\times & & \times \\
\text{b} & & \text{b} \\
\text{uminus} & & \text{uminus} \\
\text{b} & & \text{b} \\
\times & & \times \\
\text{c} & & \text{c} \\
\text{uminus} & & \text{uminus} \\
\text{c} & & \text{c}
\end{align*}
\]

- postfix representation: \( a \ b \ c \ \text{uminus} * \ b \ c \ \text{uminus} * + \text{assign} \)
Last Week: Attribute Grammars

• Adding attributes + actions to a grammar
• Evaluating attributes
  – Build AST
  – Build dependency graph
  – Evaluation based on topological order
  – (works as long as there are no cycles)
• L-attributes, S-attributed grammars
  – Pre-determined evaluation order
  – Can be integrated into parsing
Example: Annotated AST

<table>
<thead>
<tr>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → id := E</td>
</tr>
<tr>
<td>E → E1 + E2</td>
</tr>
<tr>
<td>E → E1 * E2</td>
</tr>
<tr>
<td>E → -E1</td>
</tr>
<tr>
<td>E → (E1)</td>
</tr>
<tr>
<td>E → id</td>
</tr>
</tbody>
</table>
Example: Annotated AST

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → id := E</td>
<td>S.nptr = makeNode('assign', makeLeaf(id,id.place), E.nptr)</td>
</tr>
<tr>
<td>E → E1 + E2</td>
<td>E.nptr = makeNode('+',E1.nptr,E2.nptr)</td>
</tr>
<tr>
<td>E → E1 * E2</td>
<td>E.nptr = makeNode('*',E1.nptr,E2.nptr)</td>
</tr>
<tr>
<td>E → -E1</td>
<td>E.nptr = makeNode('uminus',E1.nptr)</td>
</tr>
<tr>
<td>E → (E1)</td>
<td>E.nptr = E1.nptr</td>
</tr>
<tr>
<td>E → id</td>
<td>E.nptr = makeLeaf(id,id.place)</td>
</tr>
</tbody>
</table>

- makeNode - creates new node for unary/binary operator
- makeLeaf - creates a leaf
- id.place - pointer to symbol table
Memory Representation of an Annotated Tree

\[ a = b \ast -c + b\ast -c \]

```
a = b * -c + b* -c
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>c</td>
</tr>
<tr>
<td>uminus</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>0 2</td>
</tr>
<tr>
<td>id</td>
<td>b</td>
</tr>
<tr>
<td>id</td>
<td>c</td>
</tr>
<tr>
<td>uminus</td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>4 6</td>
</tr>
<tr>
<td>+</td>
<td>3 7</td>
</tr>
<tr>
<td>id</td>
<td>a</td>
</tr>
<tr>
<td>assign</td>
<td>9 8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
Three Address Code (3AC)

• Every instruction operates on three addresses
  – result = operand1 \texttt{operator} operand2

• Close to low-level operations in the machine language
  – Operator is a basic operation

• Statements in the source language may be mapped to multiple instructions in three address code
Three address code: example

\[
\begin{align*}
t_1 & := - c \\
t_2 & := b * t_1 \\
t_3 & := - c \\
t_4 & := b * t_3 \\
t_5 & := t_2 + t_4 \\
a & := t_5
\end{align*}
\]
Three address code: example instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := y op z</td>
<td>assignment with binary operator</td>
</tr>
<tr>
<td>x := op y</td>
<td>assignment unary operator</td>
</tr>
<tr>
<td>x:= y</td>
<td>assignment</td>
</tr>
<tr>
<td>x := &amp;y</td>
<td>assign address of y</td>
</tr>
<tr>
<td>x:=*y</td>
<td>assignment from deref y</td>
</tr>
<tr>
<td>*x := y</td>
<td>assignment to deref x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto L</td>
<td>unconditional jump</td>
</tr>
<tr>
<td>if x relop y goto L</td>
<td>conditional jump</td>
</tr>
</tbody>
</table>
Array operations

- Are these 3AC operations?

\[
x := y[i] \\
\]

\[
t1 := \&y \quad ; \quad t1 = \text{address-of } y \\
t2 := t1 + i \quad ; \quad t2 = \text{address of } y[i] \\
x := *t2 \quad ; \quad \text{value stored at } y[i]
\]

\[
x[i] := y \\
\]

\[
t1 := \&x \quad ; \quad t1 = \text{address-of } x \\
t2 := t1 + i \quad ; \quad t2 = \text{address of } x[i] \\
*t2 := y \quad ; \quad \text{store through pointer}
\]
int main(void) {
    int i;
    int b[10];
    for (i = 0; i < 10; ++i)
        b[i] = i*i;
}

i := 0                      ; assignment
L1: if i >= 10 goto L2      ; conditional jump
t0 := i*i
    t1 := &b                ; address-of operation
t2 := t1 + i            ; t2 holds the address of b[i]
    *t2 := t0               ; store through pointer
    i := i + 1
    goto L1
L2:

(example source: wikipedia)
Three address code

• Choice of instructions and operators affects code generation and optimization

• Small set of instructions
  – Easy to generate machine code
  – Harder to optimize

• Large set of instructions
  – Harder to generate machine code

• Typically prefer small set and smart optimizer
Creating 3AC

• Assume bottom up parser
  – Covers a wider range of grammars
  – LALR sufficient to cover most programming languages

• Creating 3AC via syntax directed translation using attribute grammars.

• Attributes examples:
  – code - code generated for a nonterminal
  – var - name of variable that stores result of nonterminal

• freshVar() - helper function that returns the name of a fresh variable
Attribute Grammars

• Provide (local) equations for each derivation rule
• When all local equations are correct, the full global information on the derivation is computed.
# Creating 3AC: expressions

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → id := E</td>
<td>S.code := E.code</td>
</tr>
<tr>
<td>E → E1 + E2</td>
<td>E.var := freshVar(); E.code = E1.code</td>
</tr>
<tr>
<td>E → E1 * E2</td>
<td>E.var := freshVar(); E.code = E1.code</td>
</tr>
<tr>
<td>E → - E1</td>
<td>E.var := freshVar(); E.code = E1.code</td>
</tr>
<tr>
<td>E → (E1)</td>
<td>E.var := E1.var E.code = (‘</td>
</tr>
<tr>
<td>E → id</td>
<td>E.var := id.var; E.code = ”</td>
</tr>
</tbody>
</table>

(we use || to denote concatenation of intermediate code fragments)
Generate Code

Output a string (which is the program).

gen(E.var ':=' E1.var '*' E2.var)

t256 := t23 * t124
Creating 3AC: expressions

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S -&gt; id := E</td>
<td>S.code := E.code</td>
</tr>
<tr>
<td>E -&gt; E1 + E2</td>
<td>E.var := freshVar(); E.code = E1.code</td>
</tr>
<tr>
<td>E -&gt; E1 * E2</td>
<td>E.var := freshVar(); E.code = E1.code</td>
</tr>
<tr>
<td>E -&gt; - E1</td>
<td>E.var := freshVar(); E.code = E1.code</td>
</tr>
<tr>
<td>E -&gt; (E1)</td>
<td>E.var := E1.var E.code = '('</td>
</tr>
<tr>
<td>E -&gt; id</td>
<td>E.var := id.var; E.code = ''</td>
</tr>
</tbody>
</table>

(we use || to denote concatenation of intermediate code fragments)
example

$$a = b * -c + b * -c$$

```
E.code ='
t1 = -c
t2 = b*t1
t3 = -c
t4 = b*t3
t5 = t2+t4
a = t5'
E.var = a

assign

E.var = a
E.code =”

+  

*  

b

uminus

c

E.var = t2
E.code =‘t1 = -c
t2 = b*t1
E.code =‘t1 = -c

*  

b

uminus

c

E.var = t1
E.code = ‘t3 = -c
t3 = -c
E.code = ‘t3 = -c

*  

b

uminus

c

E.var = t4
E.code =‘t3 = -c
t4 = b*t3
t5 = t2+t4’
E.code =‘t3 = -c
```
Creating 3AC: control statements

• 3AC only supports conditional/unconditional jumps
• Add labels
• Attributes
  – begin - label marks beginning of code
  – after - label marks end of code

• Helper function freshLabel() allocates a new fresh label
Creating 3AC: control statements

\[ S \rightarrow \text{while } E \text{ do } S_1 \]

**S.begin:**

- \( E \text{.code} \)
- \( \text{if } E \text{.var} = 0 \text{ goto } S \text{.after} \)
- \( S_1 \text{.code} \)
- \( \text{goto } S \text{.begin} \)

**S.after:**

- \( \cdots \)

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
</table>
| \( S \rightarrow \text{while } E \text{ do } S_1 \) |\begin{align*} & S \text{.begin} := \text{freshLabel}(); \\
& S \text{.after} := \text{freshLabel}(); \\
& S \text{.code} := \\
& \quad \text{gen}(S \text{.begin `:'}) \ || \ E \text{.code} \ || \\
& \quad \text{gen(`if' } E \text{.var `=' `0' `goto' } S \text{.after}) \ || \\
& \quad S_1 \text{.code} \ || \ \text{gen(`goto' } S \text{.begin}) \ || \ \text{gen}(S \text{.after `:'}) \end{align*} |
Allocating Memory

- Type checking helped us guarantee correctness
- Also tells us
  - How much memory allocate on the heap/stack for variables
  - Where to find struct fields (based on offsets)
  - Compute address of an element inside array (size of stride based on type of element)
## Declarations and Memory Allocation

- Global variable “offset” with memory allocated so far

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>P → D</td>
<td>{ offset := 0 }</td>
</tr>
<tr>
<td>D → D D</td>
<td></td>
</tr>
<tr>
<td>D → T id;</td>
<td>{ enter(id.name, T.type, offset); offset += T.width }</td>
</tr>
<tr>
<td>T → integer</td>
<td>{ T.type := int; T.width = 4 }</td>
</tr>
<tr>
<td>T → float</td>
<td>{ T.type := float; T.width = 8 }</td>
</tr>
<tr>
<td>T → T1[num]</td>
<td>{ T.type = array (num.val,T1.Type); T.width = num.val * T1.width; }</td>
</tr>
<tr>
<td>T → *T1</td>
<td>{ T.type := pointer(T1.type); T.width = 4 }</td>
</tr>
</tbody>
</table>
Allocating Memory

```
enter(count, int, 0)
offset = offset + 4

enter(money, float, 4)
offset = offset + 4
```

```
D1:
T1: int
id: count
id.name = count
T1.type = int
T1.width = 4

D2:
T2: float
id: money
id.name = money
T2.type = float
T2.width = 4

D3:
T3: [num]
balances
T4: int
num: 42
```

```
D6:
P
D4:
```

```
enter(count, int, 0)
offset = offset + 4

enter(money, float, 4)
offset = offset + 4
```

```
T1.type = int
T1.width = 4
T2.type = float
T2.width = 4
```
Adjusting to bottom-up

- On a top-down parsing, we can zero the offset initially. But for bottom-up, we don’t know what the first instruction is...
- A standard trick: add a marker and a rule that will always be first.

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>P → M D</td>
<td></td>
</tr>
<tr>
<td>M → ε</td>
<td>{ offset := 0}</td>
</tr>
<tr>
<td>D → D D</td>
<td></td>
</tr>
<tr>
<td>D → T id;</td>
<td>{ enter(id.name, T.type, offset); offset += T.width }</td>
</tr>
<tr>
<td>T → integer</td>
<td>{ T.type := int; T.width = 4 }</td>
</tr>
<tr>
<td>T → float</td>
<td>{ T.type := float; T.width = 8}</td>
</tr>
<tr>
<td>T → T1[num]</td>
<td>{ T.type = array (num.val,T1.Type); T.width = num.val * T1.width; }</td>
</tr>
<tr>
<td>T → *T1</td>
<td>{ T.type := pointer(T1.type); T.width = 4}</td>
</tr>
</tbody>
</table>
Allocating Memory

enter(count, int, 0)
offset = offset + 4

T₁.type = int
T₁.width = 4
id.name = count

T₁
  id
    count

D1
T2
  id
    float
    money

D2

T3
  id
    balances

D3

T4
  [ num ]

D4

P

M

D6

D5

T₁
  int
  42

offset = 0

offset = offset + 4
Generating IR code

- **Option 1**
  accumulate code in AST attributes

- **Option 2**
  emit IR code to a file/buffer during compilation
  - If for every production rule the code of the left-hand-side is constructed from a concatenation of the code of the right-hand-side in some fixed order
Expressions and assignments

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → id := E</td>
<td>{ p:= lookup(id.name); if p ≠ null then <strong>emit</strong>(p ‘:=‘ E.var) else error }</td>
</tr>
<tr>
<td>E → E1 op E2</td>
<td>{ E.var := freshVar(); <strong>emit</strong>(E.var ‘:=‘ E1.var op E2.var) }</td>
</tr>
<tr>
<td>E → - E1</td>
<td>{ E.var := freshVar(); <strong>emit</strong>(E.var ‘:=‘ ‘uminus’ E1.var) }</td>
</tr>
<tr>
<td>E → ( E1 )</td>
<td>{ E.var := E1.var }</td>
</tr>
<tr>
<td>E → id</td>
<td>{ p:= lookup(id.name); if p ≠ null then E.var := p else error }</td>
</tr>
</tbody>
</table>

- Lookup returns the variable location in memory.
- Allocate space for a variable at first sight.
### Boolean Expressions

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E1 op E2</td>
<td>{ E.var := freshVar(); emit(E.var ‘:=’ E1.var op E2.var) }</td>
</tr>
<tr>
<td>E → not E1</td>
<td>{ E.var := freshVar(); emit(E.var ‘:=’ ‘not’ E1.var) }</td>
</tr>
<tr>
<td>E → ( E1)</td>
<td>{ E.var := E1.var }</td>
</tr>
<tr>
<td>E → true</td>
<td>{ E.var := freshVar(); emit(E.var ‘:=’ ‘1’) }</td>
</tr>
<tr>
<td>E → false</td>
<td>{ E.var := freshVar(); emit(E.var ‘:=’ ‘0’) }</td>
</tr>
</tbody>
</table>

- Represent true as 1, false as 0
- Wasteful representation, creating variables for true/false
Boolean expressions via jumps

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
</table>
| $E \rightarrow \text{id1 op id2}$ | {  
E.var := freshVar();  
emit('if' id1.var relop id2.var 'goto' nextStmt+2);  
emit( E.var := '0');  
emit('goto ' nextStmt + 1);  
emit(E.var := '1')  
}  

This is useful for short circuit evaluation. Let us start with an example.
Boolean Expressions: an Example

E
--or--
E
  
  a < b

E
--and--
E
  
  c < d
e < f
Boolean Expressions: an Example

100: if a < b goto 103
101: T_1 := 0
102: goto 104
103: T_1 := 1
Boolean Expressions: an Example

100: if a < b goto 103
101: \( T_1 := 0 \)
102: goto 104
103: \( T_1 := 1 \)

104: if c < d goto 107
105: \( T_2 := 0 \)
106: goto 108
107: \( T_2 := 1 \)
Boolean Expressions: an Example

100: if $a < b$ goto 103
101: $T_1 := 0$
102: goto 104
103: $T_1 := 1$

104: if $c < d$ goto 107
105: $T_2 := 0$
106: goto 108
107: $T_2 := 1$

108: if $e < f$ goto 111
109: $T_3 := 0$
110: goto 112
111: $T_3 := 1$
112:
Boolean Expressions: an Example

100: if $a < b$ goto 103
101: $T_1 := 0$
102: goto 104
103: $T_1 := 1$

104: if $c < d$ goto 107
105: $T_2 := 0$
106: goto 108
107: $T_2 := 1$

108: if $e < f$ goto 111
109: $T_3 := 0$
110: goto 112
111: $T_3 := 1$
112: $T_4 := T_2 \text{ and } T_3$
113: 
Boolean Expressions: an Example

100: if $a < b$ goto 103
101: $T_1 := 0$
102: goto 104
103: $T_1 := 1$

104: if $c < d$ goto 107
105: $T_2 := 0$
106: goto 108
107: $T_2 := 1$

108: if $e < f$ goto 111
109: $T_3 := 0$
110: goto 112
111: $T_3 := 1$
112: $T_4 := T_2$ and $T_3$
113: $T_5 := T_1$ or $T_4$
Short circuit evaluation

- Second argument of a boolean operator is only evaluated if the first argument does not already determine the outcome.

- $(x \text{ and } y)$ is equivalent to: “if $x$ then $y$ else false”;  
- $(x \text{ or } y)$ is equivalent to: “if $x$ then true else $y$”;  

- Is short circuit evaluation equivalent to standard evaluation?  
- We use it if the language definition dictates its use.
Note Difference

```
int denom = 0;
if (denom && nom/denom) {
    oops_i_just_divided_by_zero();
}
```

```
if ( (file=open("c:\grades") or (die) )  printfile(file,...);
```
Our previous example

a < b or (c<d and e<f)

100: if a < b goto 103
101: T₁ := 0
102: goto 104
103: T₁ := 1
104: if c < d goto 107
105: T₂ := 0
106: goto 108
107: T₂ := 1
108: if e < f goto 111
109: T₃ := 0
110: goto 112
111: T₃ := 1
112: T₄ := T₂ and T₃
113: T₅ := T₁ and T₄

naive

100: if a < b goto 105
101: if ! (c < d) goto 103
102: if e < f goto 105
103: T := 0
104: goto 106
105: T := 1
106:

Short circuit evaluation
How Are Boolean Expressions Used?

• A possible use:

\[ \text{Cond} := a < b \text{ or } (c < d \text{ and } e < f) \]

• A more frequent use: in control structures:

\[
\text{if } (a < b \text{ or } (c < d \text{ and } e < f)) \text{ then print("happy");}
\]

\[
\text{while } (a < b \text{ or } (c < d \text{ and } e < f))
\{
\text{call beHappy}(a,b,c);
\}
\]
Control Structure: if, else, while.

- Consider the conditional jumps:
  \[ S \rightarrow \text{if } B \text{ then } S_1 \]
  \[ | \text{if } B \text{ then } S_1 \text{ else } S_2 \]
  \[ | \text{while } B \text{ do } S_1 \]

- Option 1: create code for B, create code for \( S_1 \), create a jump: beginning of \( S \) or end of \( S_1 \) according to B’s value.

- More efficient: create code that jumps to the correct location immediately when the value of B is discovered.
Control Structure: if, else, while.

- The problem: we do not know where to jump to...
- While parsing B’s tree for “if B then S”, we don’t know S’s code and where it starts or ends.
Control Structure: if, else, while.

- The problem: we do not know where to jump to...
- While parsing B's tree for “if B then S”, we don't know S's code and where it starts or ends.
- Solution: for each expression B keep labels: B.trueLabel and B.falseLabel. Jump to these labels according to B's value.
- Also, for each statement S, keep a label S.nextLabel with the address of the code that follows the statement S.

- The semantic equation: B.falseLabel = S.next.
- For B.true, we create a new label between B's code and S's code.
• While parsing statement $S$, generate the code with its next label.

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
</table>
| $P \rightarrow S$ | $S$.next = freshLabel();
| | $\text{P.code} = S\.code \text{ ++ \ label(S\.next)}$
| $S \rightarrow S1 S2$ | $S1\.next = freshLabel()$;
| | $\text{S2.code} = \text{S\.next}$;
| | $\text{S.code} = S1\.code \text{ ++ \ label(S1\.next)} \text{ ++ \ S2.code}$

• Is $S\.next$ inherited or synthesized?
• Is $S\.code$ inherited or synthesized?
• The (string value of the) label $S\.next$ will only be known after we parse $S$. 
### Control Structures: conditional

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
</table>
| S → if B then S1 | B.trueLabel = freshLabel();  
|   | B.falseLabel = S.next;  
|   | S1.next = S.next;  
|   | S.code = B.code || gen (B.trueLabel ‘:) || S1.code |

- “gen” creates the instruction and returns it as a string. (Unlike emit.)
- Are S1.next, B.falseLabel inherited or synthesized?
- Is S.code inherited or synthesized?
### Control Structures: conditional

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → if B then S1 else S2</td>
<td>B.trueLabel = freshLabel(); B.falseLabel = freshLabel(); S1.next = S.next; S2.next = S.next; S.code = B.code</td>
</tr>
</tbody>
</table>

- B.trueLabel and B.falseLabel considered inherited

```
B.trueLabel:  B.code
              S1.code
              goto S.next
B.falseLabel: S2.code
S.next:       ...
```

→ to B.trueLabel
→ to B.falseLabel
## Boolean expressions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Action</th>
</tr>
</thead>
</table>
| $B \rightarrow B_1 \text{ or } B_2$ | $B_1$.trueLabel = $B$.trueLabel; $B_1$.falseLabel = freshLabel();
$B_2$.trueLabel = $B$.trueLabel; $B_2$.falseLabel = $B$.falseLabel;
$B$.code = $B_1$.code || gen ($B_1$.falseLabel ‘:) || $B_2$.code |
| $B \rightarrow (B_1)$ | $B_1$.trueLabel = $B$.trueLabel; $B_1$.falseLabel = $B$.falseLabel; $B$.code = $B_1$.code; |
| $B \rightarrow \text{id}1 \text{ relop } \text{id}2$ | $B$.code = gen (‘if’ $\text{id}1$.var relop $\text{id}2$.var ‘goto’ $B$.trueLabel) ||
gen (‘goto’ $B$.falseLabel); |
| $B \rightarrow \text{true}$    | $B$.code = gen (‘goto’ $B$.trueLabel) |
| $B \rightarrow \text{false}$   | $B$.code = gen (‘goto’ $B$.falseLabel); |
### Boolean expressions

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
</table>
| B \rightarrow B1 or B2 | B1.trueLabel = B.trueLabel;  
B1.falseLabel = freshLabel();  
B2.trueLabel = B.trueLabel;  
B2.falseLabel = B.falseLabel;  
B.code = B1.code || gen (B1.falseLabel `:`) || B2.code |

- **How can we determine the address of B1.falseLabel?**

- **Only possible after we know the code of B1 and all the code preceding B1**
Example

```
if B then S1

B1 and B2

true false

B1.trueLabel = freshLabel();
B1.falseLabel = B.falseLabel;
B2.trueLabel = B.trueLabel;
B2.falseLabel = B.falseLabel;
B.code = B1.code || gen(B1.trueLabel ' ::') || B2.code
B.code = gen('goto' B1.trueLabel)
B.code = gen('goto' B2.falseLabel)
```
Example

if B then
  B1 and B2
  true false

S

S1

B.trueLabel = freshLabel();
B.falseLabel = S.next;
S1.next = S.next;
S.code = B.code || gen (B.trueLabel ‘:’) || S1.code

B1.trueLabel = freshLabel();
B1.falseLabel = B.falseLabel;
B2.trueLabel = B.trueLabel;
B2.falseLabel = B.falseLabel;
B.code = B1.code || gen (B1.trueLabel ‘:’) || B2.code

B.code = gen('goto' B1.trueLabel)

B.code = gen('goto' B2.falseLabel)
Computing the labels

- We can build the code while parsing the tree bottom-up, leaving the jump targets as variables to be determined.
- We can compute the values for the labels in a second traversal of the AST
- Can we do it in a single pass?
So far...

- Three address code.
- Intermediate code generation is executed with parsing (via semantic actions).
- Creating code for Boolean expressions and for control statements is more involved.
- We typically use short circuit evaluation, value of expression is implicit in control location.
- We need to compute the branching addresses.
- Option 1: compute them in a second AST pass.
Backpatching (תיקוק לאחור)

• Goal: *generate code in a single pass*

• *Generate code as we did before, but manage labels differently*
• *Keep targets of jumps empty until values are known, and then back-patch them*

• *New synthesized attributes for B*
  – B.truelist - list of jump instructions that eventually get the label where B goes when B is true.
  – B.falselist - list of jump instructions that eventually get the label where B goes when B is false.
Backpatching

• For every label, maintain a list of instructions that jump to this label
• When the address of the label is known, go over the list and update the address of the label

• Previous solutions do not guarantee a single pass
  – The attribute grammar we had before is not S-attributed (e.g., next is inherited), and is not L-attributed (because the inherited attributes are not necessarily from left siblings).
Example: “if-then-else”

- **Difficulty:**
  - computing $S_1.next$ requires the code of $S_1$ and $B$.
  - computing the code of $B$ requires the code of $S_1.next$.
  - It’s cyclic, yet we want to do it in one pass.

- **Solution (backpatching):**
  - compute $B$’s code up to “leaving space” for jump targets ($S_1.next$).
  - maintain a list of all empty spaces that need to jump to $S_1.next$.
  - When we determine $S_1.next$, we go over this list and update the jump targets with $S_1.next$’s value.
Functions for list handling

- `makelist(addr)` - create a list of instructions containing `addr`
- `merge(p1,p2)` - concatenate the lists pointed to by `p1` and `p2`, returns a pointer to the new list
- `backpatch(p,addr)` - inserts `addr` as the target label for each of the instructions in the list pointed to by `p`
Accumulating the code

• Bottom-up parsing, code created from left to right, bottom-up.
• Semantic analysis executed during parsing and the code is accumulated.

• Backpatching: B.truelist and B.falselist: instructions with missing jump target address.
• Also, S.nextlist: all instructions that should jump to after S.
NextInstr()

• Emit:
  – generates an instruction and puts it in CodeBuffer(nextInstr)
  – ++nextInstr

• NextInstr() returns the address of the next instruction.
Computing Boolean expressions

\[ B \rightarrow \text{id}_1 \text{ relop } \text{id}_2 \quad \{ \text{B.truelist} := \text{makelist} (\text{nextinstr}) ; \\
\hspace{1cm} \text{B.falselist} := \text{makelist} (\text{nextinstr} + 1) ; \\
\hspace{1cm} \text{emit} (\text{`if `}\text{id}_1.\text{var relop.op}\text{id}_2.\text{var `goto_`}) ; \\
\hspace{1cm} \text{emit} (\text{`goto_`}) ; \} \]

\[ B \rightarrow \text{true} \quad \{ \text{B.truelist} := \text{makelist} (\text{nextinstr}) ; \text{emit} (\text{`goto_`}) \} \]

\[ B \rightarrow \text{false} \quad \{ \text{B.falselist} := \text{makelist} (\text{nextinstr}) ; \text{emit} (\text{`goto_`}) \} \]

\[ B \rightarrow \text{not } B_1 \quad \{ \text{B.truelist} := B_1.\text{falselist} ; \\
\hspace{1cm} \text{B.falselist} := B_1.\text{truelist} \} \]

\[ B \rightarrow ( B_1 ) \quad \{ \text{B.truelist} := B_1.\text{truelist} ; \\
\hspace{1cm} \text{B.falselist} := B_1.\text{falselist} \} \]
Computing Boolean expressions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| $B \to B_1 \text{ or } M B_2$ | { backpatch ( $B_1$.falselist, M.instr ) ;  
B.truelist := merge ( $B_1$.truelist, $B_2$.truelist ) ;  
B.falselist := $B_2$.falselist } |
| $B \to B_1 \text{ and } M B_2$ | { backpatch ( $B_1$.truelist, M.instr ) ; B.truelist := $B_2$.truelist ;  
B.falselist := merge ( $B_1$.falselist, $B_2$.falselist ) } |
| $M \to \varepsilon$ | { M.instr := nextinstr } |
Using a marker

Example:

- $M.instr = \text{nextinstr}$;
- Use $M$ to obtain the address just before B2 code starts being generated
# Backpatching Boolean expressions

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>B → B1 or \textbf{M} B2</td>
<td>backpatch(B1.falseList,M.instr); B.trueList = merge(B1.trueList,B2.trueList); B.falseList = B2.falseList;</td>
</tr>
<tr>
<td>B → B1 and \textbf{M} B2</td>
<td>backpatch(B1.trueList,M.instr); B.trueList = B2.trueList; B.falseList = merge(B1.falseList,B2.falseList);</td>
</tr>
<tr>
<td>B → not B1</td>
<td>B.trueList = B1.falseList; B.falseList = B1.trueList;</td>
</tr>
<tr>
<td>B → (B1)</td>
<td>B.trueList = B1.trueList; B.falseList = B1.falseList;</td>
</tr>
<tr>
<td>B → id1 relop id2</td>
<td>B.trueList = makeList(nextInstr); B.falseList = makeList(nextInstr+1); emit('if' id1.var relop id2.var 'goto _')</td>
</tr>
<tr>
<td>B → true</td>
<td>B.trueList = makeList(nextInstr); emit('goto _');</td>
</tr>
<tr>
<td>B → false</td>
<td>B.falseList = makeList(nextInstr); emit('goto _');</td>
</tr>
<tr>
<td>\textbf{M} → ε</td>
<td>M.instr = nextInstr;</td>
</tr>
</tbody>
</table>
Example

100: if x< 100 goto _
101: goto _

\[
B.t = \{100\} \\
B.f = \{101\}
\]

\[
B \rightarrow id1 \ relop \ id2
\]

B.trueList = makeList(nextInstr);
B.falseList = makeList(nextInstr+1);
emit ('if' id1.var relop id2.var 'goto _') || emit('goto _');
Example

B.t = \{100\}
B.f = \{101\}

100: if x < 100 goto _
101: goto _

102:
M.i = 102

M \rightarrow \varepsilon
M.instr = nextinstr;
Example

100: if $x < 100$ goto _
101: goto _
102: if $x > 200$ goto _
103: goto _

B → id1 relop id2
B.trueList = makeList(nextInstr);
B.falseList = makeList(nextInstr+1);
emit ('if' id1.var relop id2.var 'goto _') || emit('goto _');
Example

B.t = {100}
B.f = {101}

M.i = 102

B.t = {102}
B.f = {103}

M.i = 104

100: if x < 100 goto _
101: goto _

102: if x > 200 goto _
103: goto _

M ➔ ε
M.instr = nextinstr;
Example

B.t = {100}
B.f = {101}

100: if x< 100 goto _
101: goto _

B.t = {102}
B.f = {103}

102: if x> 200 goto _
103: goto _

M.i = 102
M.i = 104

M.i = 104

B.t = {104}
B.f = {105}

B.t ➞ id1 relop id2
B.trueList = makeList(nextInstr);
B.falseList = makeList(nextInstr+1);
emit (‘if’ id1.var relop id2.var ‘goto _’) || emit(‘goto _’);
Example

```
Backpatch(B1.trueList,M.instr);
B.trueList = B2.trueList;
B.falseList = merge(B1.falseList,B2.falseList);
```
Example

100: if x < 100 goto _
101: goto 102
102: if x > 200 goto 104
103: goto _
104: if x!=y goto _
105: goto _

B \rightarrow B1 \text{ or } M B2

backpatch(B1.falseList,M.instr);
B.trueList = merge(B1.trueList,B2.trueList);
B.falseList = B2.falseList;
Example

100: if x<100 goto _
101: goto _
102: if x>200 goto _
103: goto _
104: if x!=y goto _
105: goto _

Before backpatching

100: if x<100 goto _
101: goto _
102: if x>200 goto 104
103: goto _
104: if x!=y goto _
105: goto _

After backpatching by the production
B → B1 and M B2

100: if x<100 goto _
101: goto _
102: if x>200 goto 104
103: goto _
104: if x!=y goto _
105: goto _

After backpatching by the production
B → B1 or M B2
# Backpatching for statements

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → if (B) M S1</td>
<td>backpatch(B.trueList,M.instr); S.nextList = merge(B.falseList,S.nextList);</td>
</tr>
<tr>
<td>S → if (B) M1 S1 N else M2 S2</td>
<td>backpatch(B.trueList,M1.instr); backpatch(B.falseList,M2.instr); temp = merge(S1.nextList,N.nextList); S.nextList = merge(temp,S2.nextList);</td>
</tr>
<tr>
<td>S → while M1 (B) M2 S1</td>
<td>backpatch(S1.nextList,M1.instr); backpatch(B.trueList,M2.instr); S.nextList = B.falseList; emit('goto' M1.instr);</td>
</tr>
<tr>
<td>S → { L }</td>
<td>S.nextList = L.nextList;</td>
</tr>
<tr>
<td>S → A</td>
<td>S.nextList = null;</td>
</tr>
<tr>
<td>M → ε</td>
<td>M.instr = nextInstr;</td>
</tr>
<tr>
<td>N → ε</td>
<td>N.nextList = makeList(nextInstr); emit('goto _');</td>
</tr>
<tr>
<td>L → L1 M S</td>
<td>backpatch(L1.nextList,M.instr); L.nextList = S.nextList;</td>
</tr>
<tr>
<td>L → S</td>
<td>L.nextList = S.nextList</td>
</tr>
</tbody>
</table>
Generate code for procedures

- we will see handling of procedure calls in much more detail later

```plaintext
n = f(a[i]);

t1 = i * 4

t2 = a[t1] // could have expanded this as well

param t2

t3 = call f, 1

n = t3
```
Extend grammar for procedures

- **type checking**
  - function type: return type, type of formal parameters
  - within an expression function treated like any other operator
- **symbol table**
  - parameter names

D → define T id (F) { S }
F → ε | T id, F
S → return E; | ...
E → id (A) | ...
A → ε | E, A
Summary

• Three address code.
• Intermediate code generation is executed with parsing (via semantic actions).
• Creating code for Boolean expressions and for control statements is more involved.
• We typically use short circuit evaluation, value of expression is implicit in control location.
• We need to compute the branching addresses.
• Option 1: compute them in a second AST pass.
• Option 2: backpatching (a single pass): maintain lists of incomplete jumps, where all jumps in a list have the same target. When the target becomes known, all instructions on its list are “filled in”. 