Compiler Theory: Intermediate Code Generation

Marc Moreno Maza
University of Western Ontario
CS447b – Winter 2002

December 1, 2004

1 Intermediate Languages

<u>An intermediate representation</u> (IR) is a language for an abstract machine (or a language that can be easily evaluated by an abstract machine)

- It should not include too much machine specific detail.
- It provides a separation between front and back ends which helps compiler portability. Code generation and assignment of temporaries to registers are clearly separated from semantic analysis.
- It allows optimization independent of the target machine. Intermediate representations are by design more abstract and uniform, so optimization routines are simpler.

Properties of *good* intermediate representations.

- Convenient to produce in the semantic analysis phase.
- Convenient to translate into code for the desired target architecture.
- Each construct must have a clear and simple meaning such that optimizing transformations that rewrite the IR can be easily specified and implemented.

Tree-Based Intermediate Representations. The most obvious way to present the information gained from lexical and syntax analysis is a syntax tree. In C, the representation can be handled using a struct for each node:

```
struct node {
    int nodetype;
    struct node *field1;
    struct node *field2;
    ...................
}
```

- The `nodetype` field contains a small integer to identify the *nonterminal corresponding* to the node.
- The other fields point to subtrees.
- Unused fields may have null pointers.

Here's a C routine for allocating storage for the nodes.

```
struct node *mknode (int type,
                     struct node *f1,
                     struct node *f2,
                     struct node *f3)
{
   struct node *t = (struct node *) malloc(sizeof(struct node));
   t -> nodetype = type;
   t -> field1 = f1;
   t -> field2 = f2;
   t -> field3 = f3;
   return t;
}
```

The routine `mknode` can be used to build the syntax tree using a syntax-directed definition where each non-terminal E has a synthesized attribute $E.ptr$ that points to the root of its syntax tree.

In practice we **do not hold** individual characters (letters or characters) at leaves of the tree:

- i.e. we do not build subtrees for characters of a variable name.

- The lexer groups these together into strings or integers.
- thus we can use string or integer fields in the `struct`.

We can extend our definition of the `struct` node from earlier to include name and value fields

```
struct node {
    int nodetype;
    struct node *field1;
    struct node *field2;
    struct node *field3;
    char *name;
    int value;
}
```

Example 1 A syntax-directed definition to produce syntax trees for assignment statements could be

Production	Semantic Rule
$S \longmapsto \textbf{id} := E$	$S.ptr :=$ make_node('**assign**', make_leaf(**id**, **id**.$entry$), $E.ptr$)
$E \longmapsto E_1 + E_2$	$E.ptr :=$ make_node('+', $E_1.ptr$, $E_2.ptr$)
$E \longmapsto E_1 * E_2$	$E.ptr :=$ make_node('*', $E_1.ptr$, $E_2.ptr$)
$E \longmapsto -E_1$	$E.ptr :=$ make_node('**unimus**', $E_1.ptr$)
$E \longmapsto (E_1)$	$E.ptr := E_1.ptr$
$E \longmapsto \textbf{id}$	$E.ptr :=$ make_leaf(**id**, **id**.$entry$)

<u>POSTFIX NOTATION</u> is a linearized representation of a syntax tree.

- It is a list of the nodes of the tree in which every node appears immediately after its children.
- The edges of a syntax tree do not appear explicitely in postfix notation.
- However they can be recovered from
  - the order in which the nodes appear and
  - the number of operands that the operator at a node expects.
  - The recovery of edges is similar to the evaluation, using a stack, of an expression in postfix notation.

2 Abstract Stack Machines

One popular form of intermediate representation is code for an abstract stack machine.

<u>ABSTRACT STACK MACHINE.</u> This machine has three memory zones.

The instruction memory which holds the instruction of the program being executed.

The data memory which store values. Each value can be accessed

- by its address
- or by the identifier of a variable associated with this address. if any.

A stack which is used to perform arithmetic operations.

The instruction are quite limited and fall into three classes:

- arithmetic operations
- stack manipulation
- control flow.

	Instruction			Stack	Data value	and address
1	`push 5`			16	0	1
2	`rvalue 2`		*top* →	7	11	2
3	`+`				7	3
4	`rvalue 3`				...	4
5	`*`	← *pc*				
6	...					

<u>ARITHMETIC OPERATIONS.</u> The abstract stack machine code for an arithmetic expression E simulates the evaluation of a **postfix** representation of E using a stack.

- The evaluation proceeds by processing the postfix representation from left to right.
- Each operand is pushed onto the stack as it is encountered.

- When a k-ary operator is reached,
  - its leftmost argument is $k-1$ positions below the top of the stack and
  - its rightmost argument is at the top of the stack.
- The evaluation applies the operator to its k arguments, pop them and push the result onto the stack.

In our intermediate language,

- all values are integers
- 0 corresponds to `false` and
- all nonzero integers corresponds to `true`.

L-VALUES AND R-VALUES. In assignments like

```
x := a
x := x + 1
```

- the right side specifies an integer value,
- the left side specifies where the value should be stored

The terms *L-values* and *R-values* refer to values that are appropriate on the left and rigth sides of an assignment, respectively.

STACK MANIPULATION. We have the following instructions.

`push` v which pushes v onto the stack.

`rvalue` ℓ which pushes the **content** of **data location** ℓ.

`lvalue` ℓ which pushes the **address** of **data location** ℓ.

`pop` which throws away the value on top of the stack.

`:=` which places the R-value on top in the L-value below it, and both are popped.

`copy` which pushes a copy of the top value on the stack.

Example 2 The translation of the PASCAL-like statement

```
x := (a + b) * (c mod 5)
```

in our abstract stack machine is

```
lvalue x
rvalue a
rvalue b
+
rvalue c
push 5
mod
*
:=
```

<u>CONTROL FLOW.</u> The stack machine executes instructions in numerical order unless told to do otherwise by a conditional or unconditional jump statement. We assume that our machine supports symolic labels, that is we can decorate any instruction with a name, its *label*. Then the control-flow instructions for our abstract stack machine are

`label` ℓ the current statement receives the label ℓ,

`goto` ℓ the next instruction is taken from statement with label ℓ,

`gofalse` ℓ **pop** the top value, say v, and **jump** to label ℓ if $v = 0$,

`gotrue` ℓ **pop** the top value, say v, and **jump** to label ℓ if $v \neq 0$.

`label` stop the execution.

Example 3 The stack machine code for an alternative

$$\textbf{if } expr_1 \textbf{ then } action_1 \textbf{ else } action_2$$

looks like

code for $expr_1$
`gofalse else`
code for $action_1$
`goto out`
`label else`
code for $action_2$
`label out`

Example 4 A boolean disjonction

$$expr_1 \textbf{ or } expr_2$$

can be viewed as an alternative:

$$\textbf{if } expr_1 \textbf{ then true else } expr_2$$

Hence, the stack machine code for it looks like

code for $expr_1$
```
copy
gotrue out
pop
```
code for $expr_2$
```
label out
```

- Recall that `gotrue` and `gofalse` pop the value on top of the stack in order to simplify code generation for conditional and while statements.
- By copying the value of $expr_1$, we ensure that the value on top of the stack is **true** if the **gotrue** instruction leads to a jump.

A boolean conjonction

$$expr_1 \textbf{ and } expr_2$$

would be treated similarly as the alternative:

$$\textbf{if } expr_1 \textbf{ then } expr_2 \textbf{ else false}$$

3 Three-Address Code

Three-address codes are a form of IR similar to assembler for an imaginary machine. Each three address code instruction has the form

```
x := y op z
```

- `x,y,z` are names (identifiers), constants, or temporaries (names generated by the compiler)
- `op` is an operator, from a limited set defined by the IR.

Observe that

- Complicated arithmetic expressions are represented as a sequence of 3-address statements, using temporaries for intermediate values. For instance the expression `x + y + z` becomes

  ```
  t1 := y + z
  t2 := x + t1
  ```

- Three address code is a linearized representation of a syntax tree, where the names of the temporaries correspond to the nodes.
- The use of names for intermediate values allows three-address code to be easily rearranged which is convenient for optimization. Postfix notation does **not** have this feature.
- The reason for the term *three-address code* is that each statement usually contain three addresses, two for the operands and one for the result.

<u>THREE-ADDRESS STATEMENTS</u> are similar to assembly code. We will consider the following statements.

Assignments. Two possible forms:

- `x:= y op z` where `op` is a binary arithmetic or logical operation,
- `x:= op y` where `op` is a unary operation (minus, negation, conversion operator)

Copy statements. They have the form `x := y`.

Inconditional jumps. They have the form

```
goto L
```

where `L` is a symbolic label of a statement.

inconditional jumps. They have the form

```
if x relop y goto L
```

where statement `L` is executed if `x` and `y` are in relation `relop`.

Procedure calls. They have the form

```
param x1
param x2

param xn
call p, n
```

corresponding to the procdure call `p(x1, x2, ..., xn)`

Return statement. They have the form `return y` where `y` representing a returned value is optional.

Indexed assignments. They have the form `x := y[i]` or `x[i] := y`.

Address assignments. They have the form `x := &y` which sets `x` to the location of `y`.

Pointer assignments. They have the form

- `x := *y` where `y` is a pointer and which sets `x` to the value pointed to by `y`
- `*x := y` which changes the location of the value pointed to by `x`.

SYNTAX-DIRECTED TRANSLATION INTO 3-ADDRESS CODE. We give below a S-definition generating 3-address code from assignments. We use the following attributes

- *S.code* represents the 3-address code for the assignment S,
- *E.place* is the name that will hold the value of E,
- *E.code* represents the sequence of 3-address statements evaluating E.
- **id**.*place* is the name that holds the value of **id** which may be found by consulting the symbol table at **id**.*entry*.

The function newtemp returns a sequence of distinct names and || is used to denote the concatenation of strings.

Production	Semantic Rule
$S \longmapsto \textbf{id} := E$	$S.code := E.code \mid\mid \mathsf{generate}(\textbf{id}.place, \text{'}\mathbf{:=}\text{'}, E.place)$
$E \longmapsto E_1 + E_2$	$E.place := \mathsf{newtemp}$
	$code := \mathsf{generate}(E.place, \text{'}\mathbf{:=}\text{'}, E_1.place, \text{'}+\text{'}, E_2.place)$
	$E.code := E_1.code \mid\mid E_2.code \mid\mid code$
$E \longmapsto E_1 * E_2$	$E.place := \mathsf{newtemp}$
	$code := \mathsf{generate}(E.place, \text{'}\mathbf{:=}\text{'}, E_1.place, \text{'}\mathbf{*}\text{'}, E_2.place)$
	$E.code := E_1.code \mid\mid E_2.code \mid\mid code$
$E \longmapsto -E_1$	$E.place := \mathsf{newtemp}$
	$code := \mathsf{generate}(E.place, \text{'}\mathbf{:=}\text{'}, \text{'}\textbf{unimus}\text{'}, E_1.place)$
	$E.code := E_1.code \mid\mid code$
$E \longmapsto (E_1)$	$E.place := E_1.place$
	$E.code := E_1.code$
$E \longmapsto \textbf{id}$	$E.place := \textbf{id}.place$
	$E.code := ''$

Observe that

- The semantic rules for $E \longmapsto \textbf{id}$ means that in this case we **do not generate a new temporary** for E, instead use the name of the identifier.
- Attributes $E.place$ and $\textbf{id}.place$ are symbol table pointers.

4 Boolean Expressions and Control Flow

<u>Boolean expressions</u> are constructed using boolean operators. We will consider here the following rules.

E	$\longmapsto$	E **or** E
E	$\longmapsto$	E **and** E
E	$\longmapsto$	**not** E
E	$\longmapsto$	(E)
E	$\longmapsto$	**id relop id**
E	$\longmapsto$	**true**
E	$\longmapsto$	**false**

- Boolean expressions are used as conditions for statements changing the flow of control.

- Evaluation of boolean expressions can be optimized if it is sufficient to evaluate a part of the expression that determines its value.

- When translating Boolean expressions into three-address code, we can use two different methods.

  Numerical method. Assign numerical values to **true** and **false** and evaluate the expression analogously to an arithmetic expression. This is convenient for boolean expressions which are not involved in flow of control constructs.

  Jump method. Evaluate a boolean expression E as a sequence of conditional and unconditional jumps to location $E.true$ (if E is **true**) or to $E.false$. To be detailed shortly.

<u>WHILE STATEMENTS WITH THE NUMERICAL METHOD.</u>

The following syntax-directed translation generates code for a while statement.

Production	Semantic Rule
$S \longmapsto$ **while** E **repeat** S_1	$S.begin :=$ **newlabel** $S.after :=$ **newlabel** $code_1 :=$ generate($S.begin$ ':') $\|\|$ $E.code$ $\|\|$ $code_2 :=$ generate('**if** ' $E.place$ '**= 0** ' '**goto**' $S.after$) $\|\|$ $S_1.code$ $code_3 :=$ generate('**goto**' $S.begin$) $\|\|$ generate($S.after$) $S.code := code_1 \,\|\|\, code_2 \,\|\|\, code_3$

With respect to the previous syntax-directed translation

- we have added two new synthesized attributes $S.begin$ and $S.after$.

- When the value of E becomes zero, control leaves the while statement

<u>FLOW OF CONTROL STATEMENTS WITH THE JUMP METHOD.</u> We will consider here the following rules.

S	$\longmapsto$	**if** E **then** S_1
S	$\longmapsto$	**if** E **then** S_1 **else**S_2
S	$\longmapsto$	**while** E **repeat** S_1

- In each of these productions, E is the boolean expression to be translated.
- The boolean expression E is associated with two labels (that are **inherited** attributes in the following semantic rules)
  - $E.true$ the label to which control flows if E is **true**,
  - $E.false$ the label to which control flows if E is **false**.
- In each of these productions, S is a flow of control statement associated with two attributes
  - $S.next$ which is a label that is attached to the first 3-address statement to be executed after the code for S , $S.next$ is an **inherited** attribute,
  - $S.code$ is the translation code for S, as usual it is a **synthesized** attribute.

Production	Semantic Rule
$S \longmapsto$ **if** E **then** S_1	$E.true :=$ newlabel
	$E.false := S.next$
	$S_1.next := S.next$
	$S.code := E.code \;\|\|$ generate($E.true$ '**:**') $\|\| \; S_1.code$
$S \longmapsto$ **if** E **then** S_1 **else** S_2	$E.true :=$ newlabel
	$E.false :=$ newlabel
	$S_1.next := S.next$
	$S_2.next := S.next$
	$code_1 := E.code \;\|\|$ generate($E.true$ '**:**') $\|\| \; S_1.code$
	$code_2 :=$ generate('**goto**' $S.next$) $\|\|$
	$code_3 :=$ generate($E.false$ '**:**') $\|\| \; S_2.code$
	$S.code := code_1 \;\|\|\; code_2 \;\|\|\; code_3$
$S \longmapsto$ **while** E **repeat** S_1	$S.begin :=$ newlabel
	$E.true :=$ newlabel
	$E.false := S.next$
	$S_1.next := S.begin$
	$code_1 :=$ generate($S.begin$ '**:**') $\|\| \; E.code$
	$code_2 :=$ generate($E.true$ '**:**') $\|\| \; S_1.code$
	$code_3 :=$ generate('**goto**' $S.begin$)
	$S.code := code_1 \;\|\|\; code_2 \;\|\|\; code_3$

Warning! The above is a syntax-directed definition: It provides formulas for the computation of the attributes $S.code$ (via the computations of the other attributes).

- Since several attributes are inherited and since each action above appears after its associated production, **this is not a translation scheme**.
- However it is an L-attributed definition.
- Then its conversion into a translation scheme is obvious.
- From now on, we may present a translation scheme as a syntax-directed definition if the latter is an L-attributed definition.
- The reason is to make large translation schemes easier to read.

<u>TRANSLATION OF BOOLEAN EXPRESSIONS WITH THE JUMP METHOD.</u> We will consider here the following rules.

E	$\longmapsto$	E_1 **or** E_2
E	$\longmapsto$	E_1 **and** E_2
E	$\longmapsto$	**not** E_1
E	$\longmapsto$	(E_1)
E	$\longmapsto$	$\mathbf{id}_1$ **relop** $\mathbf{id}_2$
E	$\longmapsto$	**true**
E	$\longmapsto$	**false**

- Here again each symbol E is associated with two inherited attributes
  - $E.true$ the label to which control flows if E is **true**,
  - $E.false$ the label to which control flows if E is **false**.
- The attributes $E.true$ and $E.false$ of E will be definned when the flow of control (where E appears) is translated.

Production	Semantic Rule
$E \longmapsto E_1$ **or** E_2	$E_1.true := E.true$ $E_1.false := newlabel$ $E_2.true := E.true$ $E_2.false := E.false$ $E.code := E_1.code \mid\mid \mathsf{generate}(E_1.false\text{'}\mathbf{:}\text{'}) \mid\mid E_2.code$
$E \longmapsto E_1$ **and** E_2	$E_1.true := newlabel$ $E_1.false := E.false$ $E_2.true := E.true$ $E_2.false := E.false$ $E.code := E_1.code \mid\mid \mathsf{generate}(E_1.true\text{'}\mathbf{:}\text{'}) \mid\mid E_2.code$
$E \longmapsto$ **not** E_1	$E_1.true := E.false$ $E_1.false := E.true$ $E.code := E_1.code$
$E \longmapsto (E_1)$	$E_1.true := E.true$ $E_1.false := E.false$ $E.code := E_1.code$
$E \longmapsto \mathbf{id}_1$ **relop** $\mathbf{id}_2$	$code_1 := \mathsf{generate}(\text{'}\mathbf{if}\text{'}\ \mathbf{id}_1.place\ \mathbf{relop}\ \mathbf{id}_2.place\ \text{'}\mathbf{goto}\text{'}\ E.true)$ $code_2 := \mathsf{generate}(\text{'}\mathbf{goto}\text{'}\ E.false)$ $E.code := code_1 \mid\mid code_2$
$E \longmapsto$ **true**	$E.code := \mathsf{generate}(\text{'}\mathbf{goto}\text{'}\ E.true)$
$E \longmapsto$ **false**	$E.code := \mathsf{generate}(\text{'}\mathbf{goto}\text{'}\ E.false)$

Example 5 Let us consider the expression

```
a < b
```

Assume that

- the attributes *true* and *false* exist for the entire expression
- as labels `Ltrue` and `Lfalse` respectively.

Then the translation is

```
if a < b goto Ltrue
goto Lfalse
```

Observations.

- Of course, this is not optimal and looks funny since the expression to translate is a sentence of the target language!
- But this *jump method* allows us to translate more involved expressions (which are not part of the target language) like those of the following examples.

Example 6 Now consider the expression

`a < b or c < d`

Again assume that the labels `Ltrue` and `Lfalse` have been set for the entire expression. Then the translation is

```
     if a < b goto Ltrue
     goto L1
L1:  if c < d goto Ltrue
     goto Lfalse
```

Example 7 Now consider the expression

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

Then the translation is

```
     if a < b goto Ltrue
     goto L1
L1:  if c < d goto L2
     goto Lfalse
L2:  if e < f goto Ltrue
     goto Lfalse
```

Of course the generated code is not optimal! Indeed the second statement can be eliminated.

Example 8 Finally consider the expression

```
while a < b do
    if c < d then
       x := y + z
    else
       x := y - z
```

Then the translation is

```
L1:     if a < b goto L2
        goto Lnext
L2:     if c < d goto L3
        goto L4
L3:     t1 := y + z
        x := t1
        goto L1
L4:     t2 := y - z
        x := t2
        goto L1
Lnext:
```

5 Optimizations for the generated code

<u>Reducing the number of temporaries.</u> Temporaries are used to hold intermediate values when evaluated expressions need to be stored in the symbol-table.

- They can require a lot of space for their values.
- Moreover wasting space implies wasting time. Indeed they have to be loaded into registers, written back, etc.
- For nowdays compilers, time performance is a crucial issue since space is almost illimited.

Temporaries can be re-used!

- There are a lot of different approaches to doing this
- and a lot of discussion about when during compilation it should be done.

A simple approach is to change *newtemp*.

- Consider the rule $E \longmapsto E_1 + E_2$

- Assume E_1 and E_2 are evaluated into the temporaries t_1 and t_2 respectively.
- From the rules for the synthesized attribute $E.place$ it follows that t_1 and t_2 are not used anywhere else in the program.
- Moreover, if we **do not use DAGS** for arithmetic expressions then the lifetime of each temporary used in the evaluation of E_i is contained in the lifetime of t_i.
- Hence we can modify *newtemp*
  - so that it assigns names for temporaries in a *last-in, first-out* manner.
  - To do so we could store each temporary t together with its scope $s(t)$ such that we can reuse t in another scope from which $s(t)$ is not visible. Scopes are naturally organized as a tree T and it is natural to associate a scope s with its depth $d(s)$ as a node of T (the length of a shortest path from the root of T to s). For most languages (including MOOL, ALLCOT), if the current scope s' (entered after exiting $s(t)$) satisfies $d(s') < d(s(t))$ then we can reuse t.

We will come back to the topic of code optimization in the next chapter.

6 Optimizations for the compiler performances

LOOKING UP SYMBOLS. One should avoid intensive searching for symbols in the symbol table during type checking and translation.

- The syntax-directed definitions shown earlier for simple code generation use names as the attributes **id**.*place*.
- In fact, names should be represented by pointers to symbol-table entries (rather than strings).
- These pointers can be set up during the lexical analysis.
- Hence the value associated with a token `TOK_ID` (for identifiers) returned by LEX should better be a pointer to an entry of the symbol table rather than a string.

Emission rather than concatenation. Instead of attaching a string $S.string$ to each instance of the grammar symbol S encountered during the translation, one should try

- either to write 3-address statements directly into output.
- or to append linked lists of strings (where previously we were concatenating strings).

In the next syntax-directed definitions, the procedure emit writes 3-address statements directly into output.

6.1 Backpatching

When transforming a translation scheme into a YACC program we saw how to forward inherited attriutes (by using markers). This solves the problem of implementing L-attributed syntax-directed definitions in YACC. However we need sometimes to face more general situations.

The general problem. Let $A \longmapsto X_1 \cdots X_n$ be a production used in a parse tree. We consider the case where for a grammar symbol X_i in the rigth side of the production possesses an inherited attriute $X_i.in$ which cannot be given value before the subtree rooted at A is completely visited.

An example and its solution.

- Boolean expressions are usually translated using the jump method since this is convenient for optimization.
- However, more than a single pass may be needed in order to generate code for boolean expressions and flow of control during bottom-up parsing!
- Indeed, when translating forward jumps, at the time we generate the code we do not know the (numerical) address of the label we want to branch to.

The solution is to generate a sequence of branching statements where the addresses of the jumps are temporarily left unspecified.

1. For each boolean expression E we maintain two lists

- $E.truelist$ which is the list of the (addresses of the) jump statements appearing in the translation of E and forwarding to $E.true$.
- $E.falselist$ which is the list of the (addresses of the) jump statements appearing in the translation of E and forwarding to $E.false$.

2. When the label $E.true$ (resp. $E.false$) is eventually defined we can *walk down* the list, patching in the value of its address.

In the **translation scheme** below

- We use emit to generate code that contains *place holders* to be filled in later by the backpatch procedure.
- The attributes $E.truelist$, $E.falselist$ are **synthesized**.
- When the code for E is generated, addresses of jumps corresponding to the values *true* and *false* are left unspecified and put on the lists $E.truelist$ and $E.falselist$, respectively.
- A marker non-terminal M is used to capture the numerical address of a statement we want to branch to.
- nextaddr is a **global variable** that stores the number of the next statement to be generated. In all examples we assume that nextaddr is initialized to `100`.
- makelist is a function which creates a new list from one item.
- merge is a function which creates a sorted list of addresses from two such lists.

	Production	Semantic Rule
1	$E \longmapsto E_1$ **or** $M\ E_2$	{ backpatch($E_1.falselist$, $M.quad$) $E.truelist$:= merge($E_1.truelist$, $E_2.truelist$) $E.falselist := E_2.falselist$ }
2	$E \longmapsto E_1$ **and** $M\ E_2$	{ backpatch($E_1.truelist$, $M.quad$) $E.truelist := E_2.truelist$ $E.falselist$:= merge($E_1.falselist$, $E_2.falselist$) }
3	$E \longmapsto$ **not** E_1	{ $E.truelist := E_1.falselist$ $E.falselist := E_1.truelist$ }
4	$E \longmapsto (E_1)$	{ $E.truelist := E_1.truelist$ $E.falselist := E_1.falselist$ }
5	$E \longmapsto$ **id**$_1$ **relop id**$_2$	{ $E.truelist$:= makelist(nextaddr) nextaddr := nextaddr + 1 $E.falselist$:= makelist(nextaddr) nextaddr := nextaddr + 1 emit('**if**' **id**$_1.place$ **relop id**$_2.place$ '**goto** __') emit('**goto** __') }
6	$E \longmapsto$ **true**	{ $E.truelist$:= makelist(nextaddr) }
7	$E \longmapsto$ **false**	{ $E.falselist$:= makelist(nextaddr) }
8	$M \longmapsto \varepsilon$	{ $M.quad$:= nextaddr }

Example 9 Let us consider again the expression

`a < b`

We only need to apply the above Production 5 leading to

```
100  if a < b goto __
101  goto __
```

Moreover this expression, say E_1, is associated with the lists

$$E_1.truelist = [100]$$

$$E_1.falselist = [101]$$

and nextaddr has value 102.

Example 10 Consider again the expression

`a < b or c < d`

During a bottom-up parsing we will apply twice Production 5 and one Production 1. Before performing the actions of Production 1 the generated code is

```
100  if a < b goto __
101  goto __
102  if c < d goto __
103  goto __
```

and the subexpressions $E_1 = \texttt{a < b}$ and $E_2 = \texttt{c < d}$ are associated with the lists

$$E_1.truelist \ = \ [100],\ E_1.falselist \ = \ [101]$$

$$E_2.truelist \ = \ [102],\ E_2.falselist \ = \ [103]$$

Moreover

- nextaddr has value 104
- $M.quad$ has value 102.

After performing the actions of Production 1 the generated code is

```
100  if a < b goto __
101  goto 102
102  if c < d goto __
103  goto __
```

Moreover the expression $E = \texttt{a < b or c < d}$ is associated with

$$E.truelist \ = \ [100, 102]$$

$$E.falselist \ = \ [103]$$

and nextaddr has value 104.

Example 11 Now consider the expression

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

During a bottom-up parsing we will apply

- three times Production 5
- one Production 2
- and finally once Production 1.

Before performing the actions of Production 1 the generated code is

```
100 if a < b goto __
101 goto __
102 if c < d goto 104
103 goto __
104 if c < d goto __
105 goto __
```

and the subexpressions E_1 = `a < b` and E_4 = `c < d and e < f` are associated with the lists

$$E_1.truelist \;=\; [100],\; E_1.falselist \;=\; [101]$$

$$E_4.truelist \;=\; [104],\; E_2.falselist \;=\; [103, 105]$$

Moreover

- nextaddr has value 106
- $M.quad$ has value 102.

After performing the actions of Production 1 the generated code is

```
100 if a < b goto __
101 goto 102
102 if c < d goto 104
103 goto __
104 if c < d goto __
105 goto __
```

Moreover the expression E = `a < b or (c < d and e < f)` is associated with

$$E.truelist \;=\; [100, 104]$$

$$E.falselist \;=\; [103, 105]$$