CPSC 323 Compilers and Languages

Instructor: Susmitha Padda

Credits : Doina Bein

Chapter 5. Intermediate Code Generation

5.1 Introduction

5.2 Intermediate Representations

a) Abstract Syntax Tree (AST)

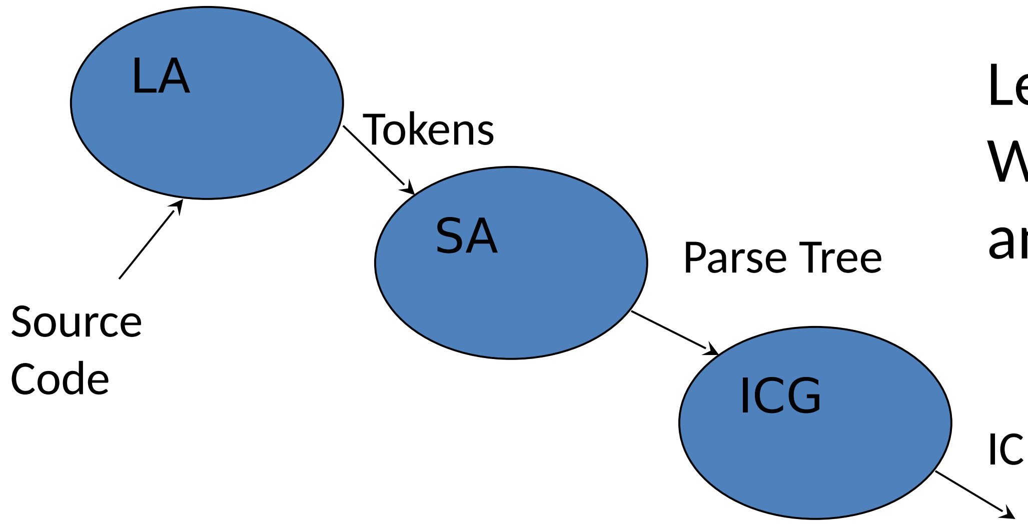
b) Postfix Notation

c) Three Address Code

5.3 Bottom-up Translations

5.4 Top-Down Translations

5.1 Introduction



Let's see where we are now!
We have tokenized the program and parse it.

- We can generate the IC directly from the parse tree or we can produce an intermediate representation and get the IC or code in object language
- The choice depends partially on how much optimization is to be done or whether the writer intends the compiler to be retargetable
- An intermediate representation provides useful information for the optimizer while on object code very little optimization can be done

- A retargetable compiler will have different back ends for different target machines, thus a machine-independent intermediate representation is needed
- There are several intermediate representations and they all based on the syntactic information discovered during the parse
- We can attach a *meaning* to each production so they become *semantic actions*
- Example:

E1 -> E2 + E3 { E1 := E2 + E3 }

E1 -> E2 * E3 { E1 := E2 * E3 }

E1 -> (E2) { E1 := E2 }

E -> i { E := i.lexeme }

- The interpretation E1 := E2 + E3 tells us that to get E1 we must add the things to which E2 and E3 evaluated.
- Because the sequence of productions guides the generation of intermediate code, we call this process *syntax-directed translation*

- Let’s go back to parsing with a stack (Chapter 4.1.1) and we want to parse a+b*c (in the textbook is listed first as i + i* i)

move	stack	Unread input	production	Action
-	$	a+b*c$		
shift	$a	+b*c$	E1 -> i	E1 := a
reduce	$E1	+b*c$		
shift	$E1+	b*c$		
shift	$E1+b	*c$	E2 -> i	E2 := b
reduce	$E1+E2	*c$		
shift	$E1+E2*	c$		
shift	$E1+E2*c	$	E3 -> i	E3 := c
reduce	$E1+E2*E3	$	E4 -> E2*E3	E4 := E2*E3
reduce	$E1+E4	$	E5 -> E1 + E4	E5 := E1 + E4
reduce	$E5	$		

- The sequence of instructions that result are:

E1 := a

E2 := b

E3 := c

E4 := E2 * E3

E5 := E1 + E4

Recall the parse tree:

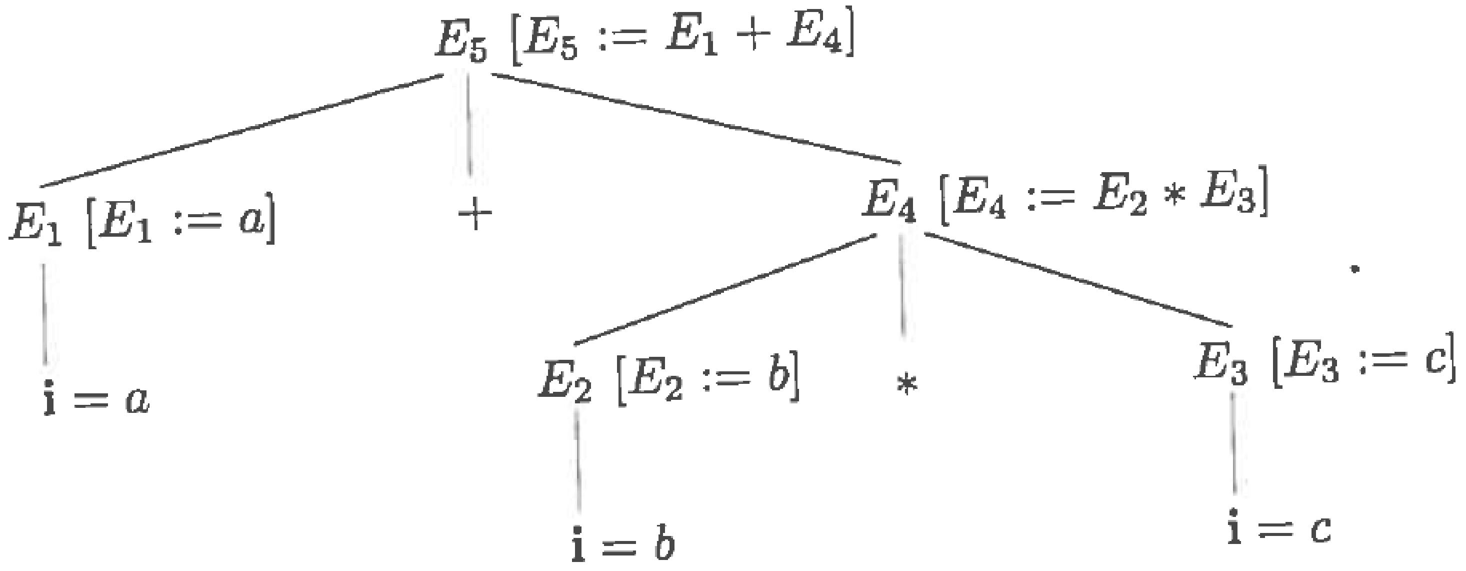


Figure 5.1

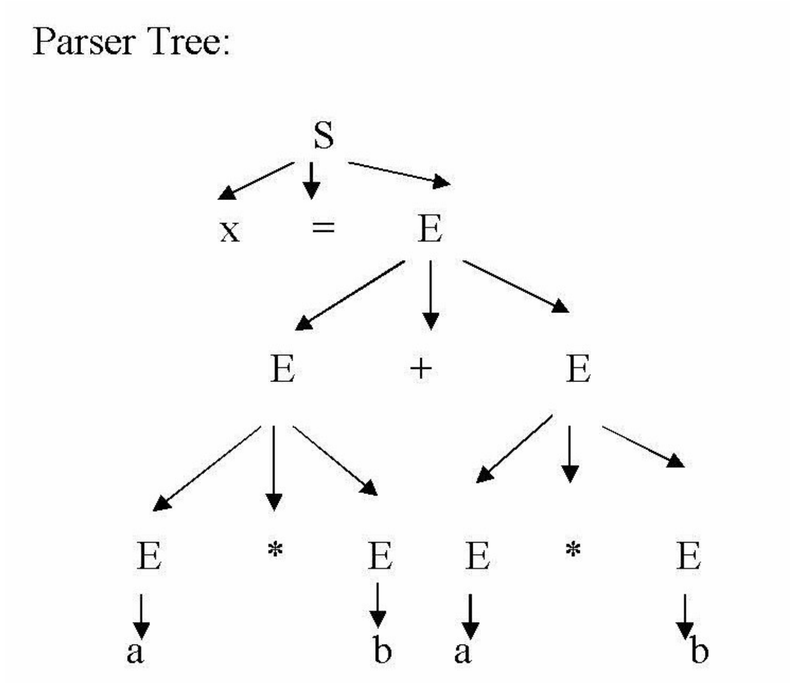
- The computations or other operations attached to the productions impute a meaning to each production, so these operations are called *semantic actions*
- The information obtained by the semantic actions is associated with the symbols of the grammar and is normally placed in the records associated with the symbols in the symbol table; these fields are called *attributes*
- The association of meanings and productions solves the following problems:
  - Making sure that the variables are declared before use
  - Type checking
  - Actual and formal parameters are properly matched
- This aspect of intermediate code generation is called *semantic analysis*

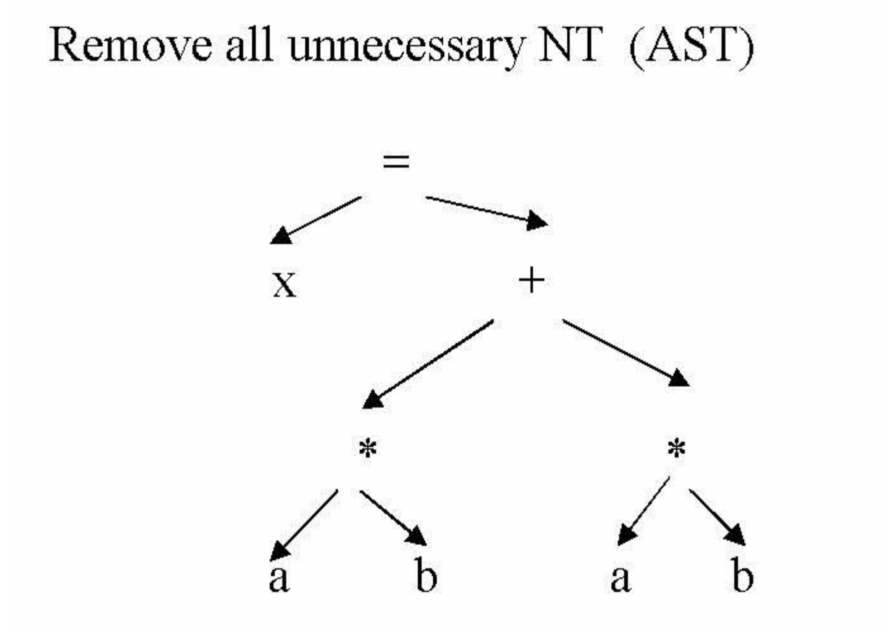
5.2 Intermediate Representations

- The most common representations are:
  - Abstract Syntax Tree (AST)
  - Directed acyclic graphs (DAGs)
  - Postfix Notation
  - Three Address Code (3AC)

a) AST

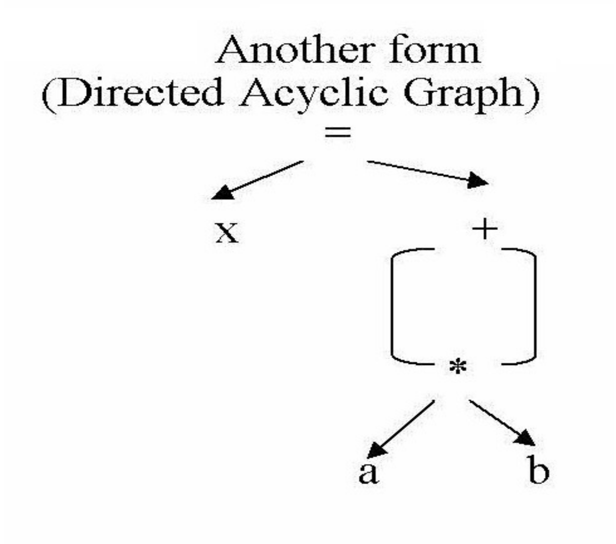
- A concrete syntax tree is a parse tree
- An abstract syntax tree has the same form as a parse tree but where all “unnecessary” parsing info is removed, i.e. the operators take the place of the nonterminals for the internal nodes
- Example: the statement x = a*b + a*b





b) DAGs

- A DAG is a relative of a syntax tree
- The difference is that the nodes for repeated variables or subexpressions are merged into a single node
- A DAG is constructed the same way as a syntax tree, except before constructing a node for anything, check whether such a node already exists
- By generating code from a DAG instead of a syntax tree avoids duplicated code
- Using DAGs to eliminate redundant code is the first instance of optimization (Chapter 6)



Example: x = a*b + a*b, the expression "a*b" needs to be computed one time only and the second time can be reused.

c) Postfix Notation

- Also known as reverse Polish notation (it was devised by the Polish mathematician Jan Lukasiewicz)
- Every expression is rewritten with the operator at the end
- Postorder traversal of a syntax tree produces the postfix notation
- Examples

"a + b"   in postfix notation becomes "a b +"

"a + b * b" in postfix notation becomes "a b b * + "

" x = a*b + a*b" in postfix notation becomes "ab* ab* + x ="

- Postorder traversal applies only to binary trees; if the tree is not necessarily binary, then we have the traversal in which every node is visited after all its subtrees are visited
- Examples:

"- x" in postfix notation becomes "x -"

"if C then A else B" becomes "C A B ?" where ?=if-then-else operator

- An expression in postfix notation can be easily evaluated using a stack:
  - Operands are pushed into the stack
  - Operators pop the required number of operands from the stack, do the operation and push the result back into the stack

d) Three-Address Code (3AC)

- This form breaks the program down into elementary statements having no more than three variables and no more than one operator
- Example 1: the statement “x = a + b*b” translates into the following 3AC statements

```
T := b*b
x := a + T
```

where T is a temporary variable to hold the product

- Example 2: the statement “ x = a*b + a*b” translates into the following 3AC statements

```
T1  = a *b
T2  = a*b
T3  = T1 + T2
X   = T3
```

- This representation will be our focus, since it is close to assembly
- Example 1 in the assembly language of IBM System/370 family of computers become

```
L     3, B
M     2, B   { T := b*b }
ST    3, T
L     3, A
A     3, T   { x := a + T }
ST    3, X
```

Or using General register 3 to hold temporary results:

```
L     3, B
MR    2, 3
A     3, A
ST    3, X
```

- Note that at the machine level A, B, T, and X are *addresses*

Intermediate Languages

- P-code [Nori, 1981] is an intermediate language
- P-code is based on interpretation
- In case of p-code, the machine language is that of an hypothetical stack-based computer
- P-code interpreters for various machines are easy to write, a p-code system is highly portable
- Some of the first Pascal compilers were distributed in p-code
- P-code can be either interpreted or translated
- A Pascal p-compiler is a front-end: does the lexical scan, parses the program, and does the syntax-directed translation into p-code

Generating Intermediate Code

- One way: from the parse tree
- Another way: During the SA, combine the SA and ICG which is called Syntax Directed Translation (SDT)
  - Attach the “ meaning” to each production, so called Semantic action
  - Example:

```
E   ->  E + E     {V (E1) = V(E2)  + V(E3) }
E  ->   id         {V (E) = V(id) }
```

  - And whenever we recognize a production, we execute the attached action

5.3 Bottom-up Translation

- It is easier to perform intermediate code generation from bottom-up parsing than top-down parsing: syntax-directed tree using bottom-up parsers
- We must keep track of the attributes of symbols in the grammar
- For an identifier, it will be its address in the symbol table
- For a nonterminal, some appropriate reference to part of the intermediate representation, for example the pointer to the root node (for the starting symbol) or some internal node in the parse tree
- To keep track of these attributes we use the semantic stack
- For bottom up parsing, the semantic stack and the parser move in synchronism:
  - When we pop the parser stack we want to pop the semantic stack
  - When we push something in the parser stack we want to push something into the semantic stack

- We place the attributes in the parser stack itself
- A stack entry (frame) from an LR parser contained a state and optionally a symbol
- We enlarge the frame to contain some sort of reference to the attribute associated with the corresponding nonterminal
- Our stack frame has the form:

```
frame = record
    state: integer;
    symbol: char; { Optional}
    attribute: <some appropriate type>
end;
```

- The attribute field will be renamed as needed for each intermediate representation
  - When we reduce, we carry out the semantic action associated with the production
  - When we pop the handle off the stack, the elements we need for the semantic action will come off the stack with it

a) Translation into AST

- We grew parse trees by planting little trees in the forest and grafting them together as needed
- We grow syntax trees similarly
- The semantic action associated with each production will include planting a tree and taking care of the grafts
- Each tree must contain its operator and pointers to other nodes down the tree
  - The number of pointers depend on the *arity* of the operator, i.e. the number of operands it takes
  - Leaves contain either variables or constants
  - Leaves require a field giving the token type and a pointer to the entry in the symbol table for the token
- The attribute field of the stack frame will contain a pointer to the root of the tree for the expression

- Since nodes can be either internal or leaves, we consider a variant record
- We declare a node as

```
type
      nodekind = (int, leaf);
      nodeptr = ^ treenode;
      treenode = record
                    token: tokentype;
                    case kind: nodekind of
                        int: (left, right: nodeptr);
                        leaf : (loc: integer);
                  end;
```

- `tokentype` can be either a character or a string containing a copy of the token, as ‘+’ or ‘*’, or a numerical code

- To plant a tree, we use the function `MakeTree` that allocates memory for the nodes, fill in their contents, and return a pointer to the root of the tree; it also takes care of grafting our trees
- The function `MakeLeaf` creates a leaf and returns a pointer to it

```
function makeleaf (t : tokentype)
   : nodeptr;
{
    new (leaf);
    with leaf-> do
        token = t;
        left = right = nil;
    endwith
    makeleaf = leaf;
}
```

```
function  maketree  (op: tokentype;
   leftson, rightson: nodeptr): nodeptr;
{
     new (root);
     with root-> do
          token = op;
          left = leftson;
          right = rightson;
      endwith
      maketree = root;
}
```

- For the grammar:

```
S -> i = E
E -> E + E
E -> E * E
E -> (E)
E -> i
```

- We assign semantic actions to the productions:

R1: `S -> i = E` `{S.root := maketree ('=', i.loc, E) }`

R2: E_i `->` E_j `+` E_k `{`E_i`.root = maketree ('+',` E_j`,` E_k `) }`

R3: E_i `->` E_j `*` E_k `{`E_i`.root = maketree ('*',` E_j`,` E_k `) }`

R4: E_i `-> (`E_j`)` `{`E_i`.root =` E_j`.root }`

R5: `E -> i` `{E.root = makeleaf (i, i.loc) }`

where `i.loc` refers to the symbol table address of the token `i`

- Note that parentheses are not put into the syntax tree and are used only to control the action of the parser such that the expression inside the parentheses is evaluated first
- Parentheses are always dropped in generating an intermediate representation

- The semantic actions are part of the parse as follows:

Shift:

```
    If token = i then i.loc = addr(i)
    else E.loc = blank;
```

Reduce:

1. Look up the production number in the list of productions
2. Pop the appropriate number of things off the parse stack and save them in a suitable array; call it the `rhs`
3. If the production is of the form E -> i then

    Pass the token and its symbol-table address to MakeLeaf and put the pointer returned by MakeLeaf into E.root

else if the production is of the form E -> (E) then

    copy the pointer in rhs[2] into E.root

else

    Pass the operator and the pointers from rhs[1] and rhs[3] top MakeTree and put the pointer returned by makeTree into E.root

4. Find the new state from the parse table and put it into E.state. Push E onto the stack

Example: Parsing “x = (a + b) * c” (use Parsing with a Stack from Chapter 4.1, next slide)

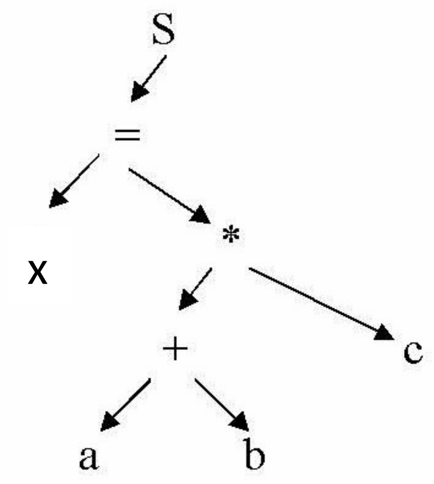
R1: S -> id = E(1)	{E = makeleaf (id); S = maketree (=, E, E(1)) }
R2: E (1) -> E (2) + E (3)	{E(1) = maketree (+, E (2), E(3)) }
R3: E (1) -> E (2) * E (3)	{E(1) = maketree (*, E(2), E(3)) }
R4: E(1) -> (E(2))	{ E(1) = E (2) }
R5: E -> id	{E = makeleaf (id) }

Parsing Stack	Input	Prod. Used	Semantic action	Semantic Stack
$	x= (a+b)*c $			
$ x	= (a+b)*c $			
$x=	(a+b)*c $			
$x = (	a+b)*c $			
$x = (a	+b)*c $	E -> id	E(1)= makeleaf(a);	
$x = (E	+ b)*c$			E (1)
$x =(E +	b)*c$			E(1)
$x= (E+ b	)*c$	E ->id	E(2) = makeleaf(b);	E(1)
$x =(E+E	)*c$	E -> E + E	E(3) = maketree(+, E(1), E(2))	E(2) E(1)
$ x = (E	)*c$			E(3)
$x = (E)	*c$	E -> (E)	E(4) = E(3)	E(4)
$x = E	*c$			E(4)
$x = E*	c$			E(4)
$x = E * c	$	E -> id	E(5) = makeleaf(c)	E(5) E(4)
$x = E * E	$	E -> E*E	E(6) = maketree(*, E(4),E(5))	E(5) E(4)
$x = E	$	S -> id = E	E(7) = makeleaf (x);	E (6)
			S = maketree (=, E(7), E(6))	E(7) E(6)
$ S				S

Parsing with a Stack

- We will push tokens into the stack until at the top of the stack is the right hand side of a production (i.e we have something we can reduce on the top of the stack)
- When we have, we will do it by popping the right-hand side of a production off the stack and pushing the left-hand side of the production on its place
- We will continue pushing and reducing until the input is used up and the stack contains only the starting symbol
- <u>**Handle**</u>: A handle is the ***right-hand side*** of a production which we can reduce to get the preceding step in the RMD
- Additionally, in the rightmost derivation, there are never any nonterminals to the right of a handle

Results in a AST:


$$\begin{tikzcd}
 & & S \arrow[dl] & \\
 & = \arrow[dl] \arrow[dr] & & \\
x & & * \arrow[dl] \arrow[dr] & \\
 & + \arrow[dl] \arrow[dr] & & c \\
a & & b &
\end{tikzcd}$$

Translating into a DAG

- In constructing a DAG there are two complications:

1. Finding how to detect an existing node of the desired type
   - Easiest is to store the nodes in an array and index the array by hashing the contents of the node
   - When we need to create a node, we hash into the array and look for a match
   - If a matching node is found, we return the pointer to the node
   - Otherwise we create a new node and enter it into the array
   - The hashed addresses are known as *value numbers*
2. Occasionally we have to create an apparent redundant node even in DAG (later in Chapter 6)

5.4 Top-down translation

- We need two stacks, one for the parser and one to hold the attributes
  - The semantic actions will be stored in the parser stack
  - The attributes will be stored in a separate stack since they will be stacked in a different sequence from the RHS of the productions
- Top-down is not easy (as straight forward) due to the change of the original grammar to remove the left-recursion.
- Suppose the previous grammar is now changed to remove left-recursion

E -> TQ

Q -> +TQ | epsilon

T -> FR

R -> *FR | epsilon

F -> (E) | I

- This nonrecursive grammar fragments the parse tree in a way that does not help with the intermediate code generation

Example: a + b * c
The parse tree is below

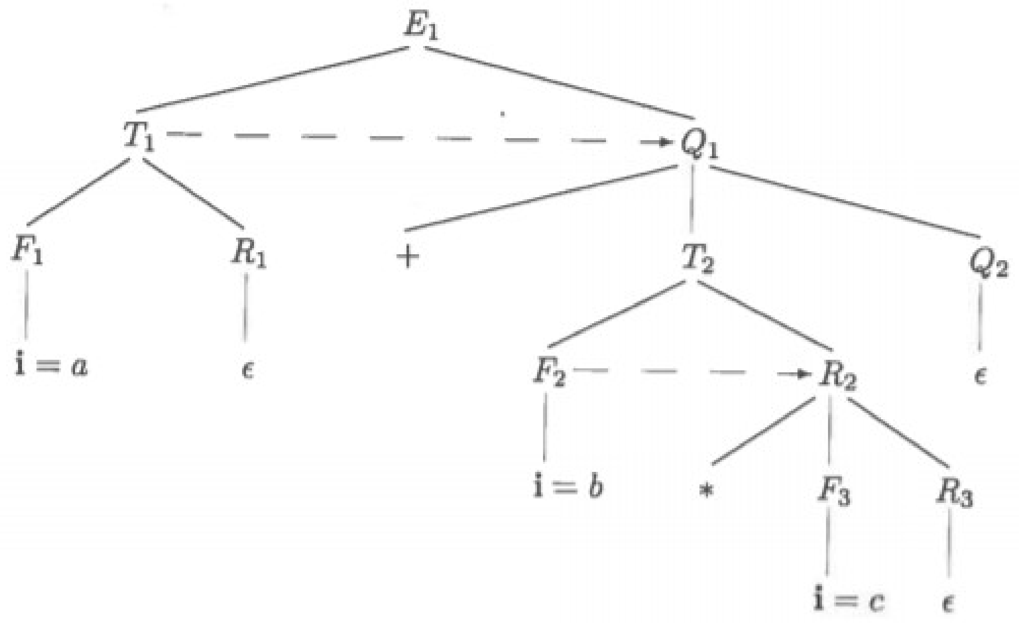


Figure 5.7

First we need to multiply b and c.
We get c from F3 subtree but b is in F2 subtree so we need to get b from F2 (the dotted line)
One level up, we need to get a to add it to the result of the multiplication b*c

Role of Semantic Analysis

- Following parsing, the next two phases of the "typical" compiler are
  - semantic analysis
  - (intermediate) code generation
- The principal job of the semantic analyzer is to enforce static semantic rules
  - constructs a syntax tree (usually first)
  - information gathered is needed by the code generator

Role of Semantic Analysis

- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved
- A common approach interleaves construction of a syntax tree with parsing (no explicit parse tree), and then follows with separate, sequential phases for semantic analysis and code generation

ELSEVIER