Syntax Trees

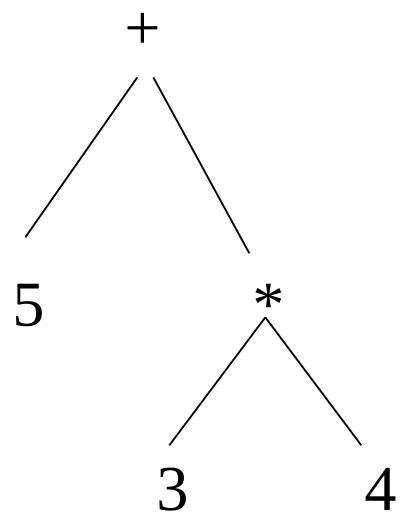
Syntax-Tree

- an intermediate representation of the compiler's input.
- A condensed form of the parse tree.
- Syntax tree shows the syntactic structure of the program while omitting irrelevant details.
- Operators and keywords are associated with the interior nodes.
- Chains of simple productions are collapsed.

Syntax directed translation can be based on syntax tree as well as parse tree.

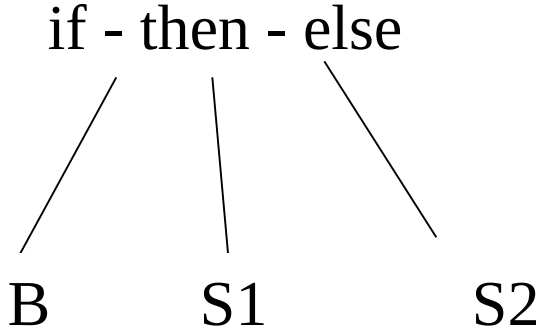
Syntax Tree-Examples

Expression:



- Leaves: identifiers or constants
- Internal nodes: labelled with operations
- Children: of a node are its operands

if B then S1 else S2



Statement:

- Node's label indicates what kind of a statement it is
- Children of a node correspond to the components of the statement

Constructing Syntax Tree for Expressions

- Each node can be implemented as a record with several fields.
- Operator node: one field identifies the operator (called *label of the node*) and remaining fields contain pointers to operands.
- The nodes may also contain fields to hold the values (pointers to values) of attributes attached to the nodes.

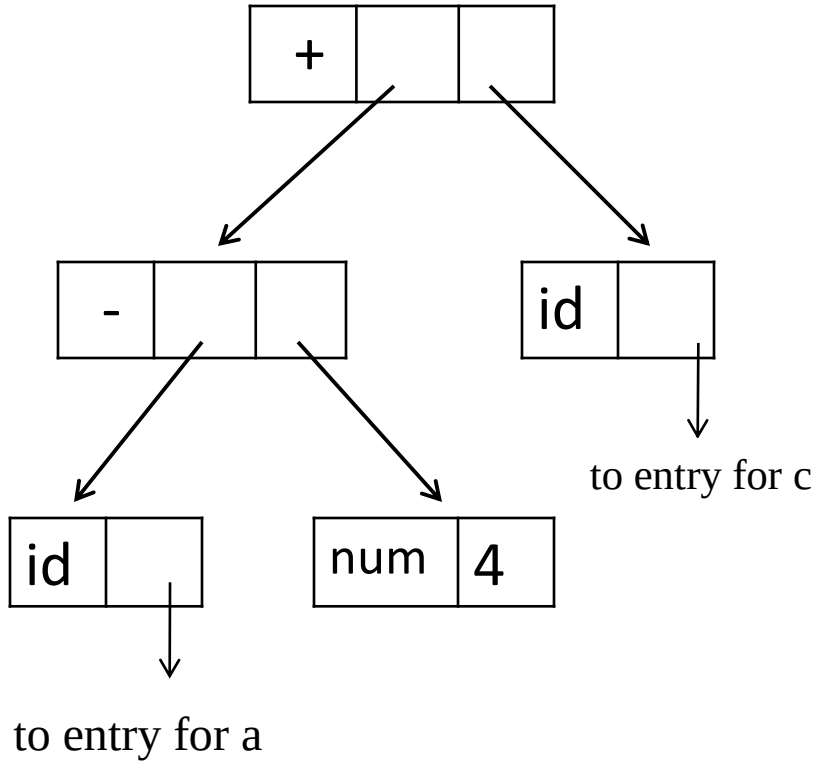
- Functions used to create nodes of syntax tree for expressions with binary operator are given below.
  - mknode(op,left,right)
  - mkleaf(id,entry)
  - mkleaf(num,val)

Each function returns a pointer to a newly created node.

Constructing Syntax Tree for Expressions-

Example: a-4+c

1. p1:=mkleaf(id,entrya);
2. p2:=mkleaf(num,4);
3. p3:=mknode(-,p1,p2)
4. p4:=mkleaf(id,entryc);
5. p5:= mknode(+,p3,p4);



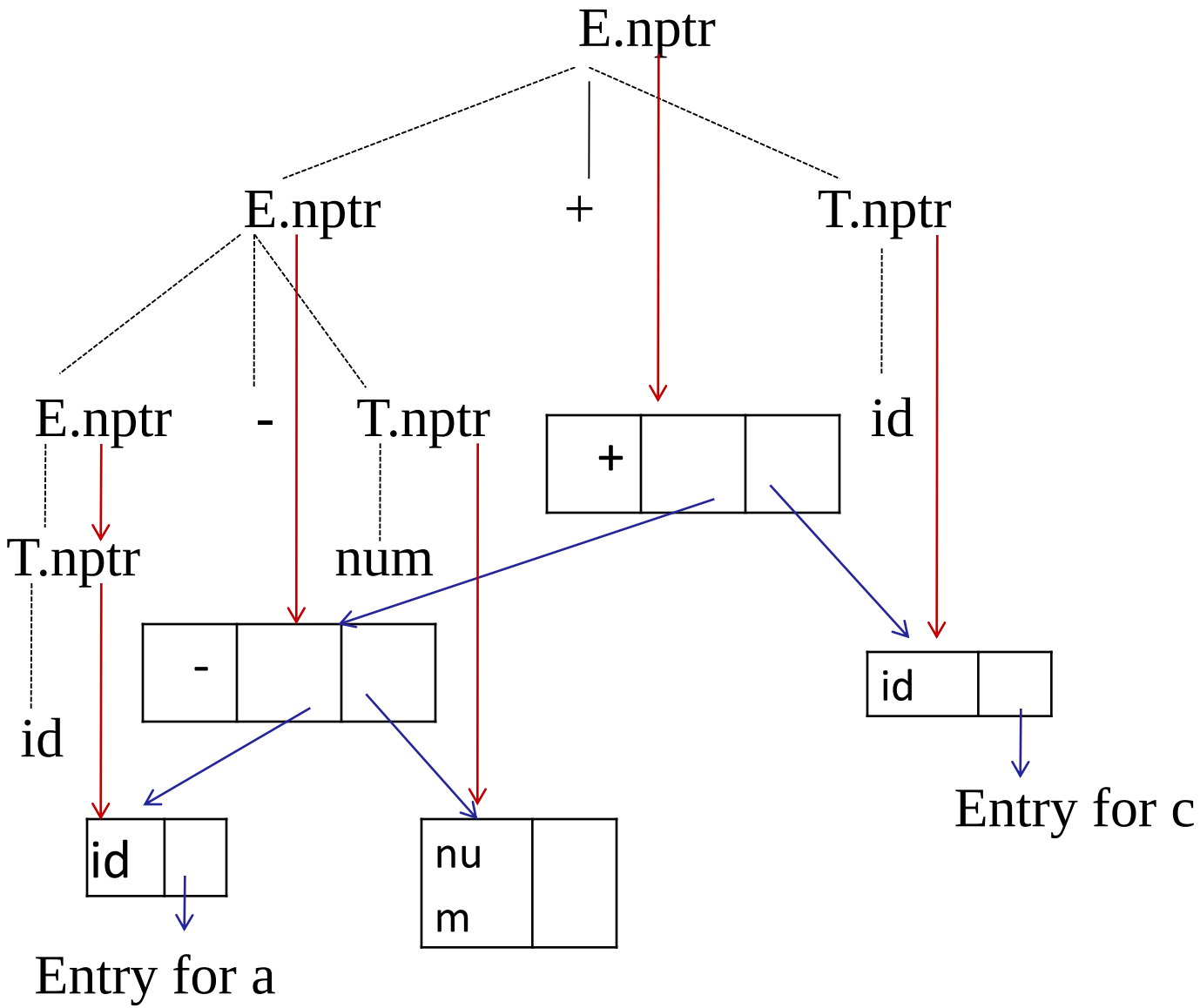
- **The tree is constructed bottom up.**

A syntax Directed Definition for Constructing Syntax Tree

1. It uses underlying productions of the grammar to schedule the calls of the functions *mkleaf* and *mknode* to construct the syntax tree
2. Employment of the synthesized attribute *nptr* (pointer) for E and T to keep track of the pointers returned by the function calls.

PRODUCTION	**SEMANTIC RULE**
$E \rightarrow E_1 + T$	E.*nptr* = *mknode*(“+”,E_1.*nptr* ,T.*nptr*)
$E \rightarrow E_1 - T$	E.*nptr* = *mknode*(“-”,E_1.*nptr* ,T.*nptr*)
$E \rightarrow T$	E.*nptr* = T.*nptr*
$T \rightarrow (E)$	T.*nptr* = E.*nptr*
$T \rightarrow id$	T.*nptr* = *mkleaf*(id, id.*lexval*)
$T \rightarrow num$	T.*nptr* = *mkleaf*(num, num.*val*)

Annotated parse tree depicting construction of syntax tree for the expression a-4+c



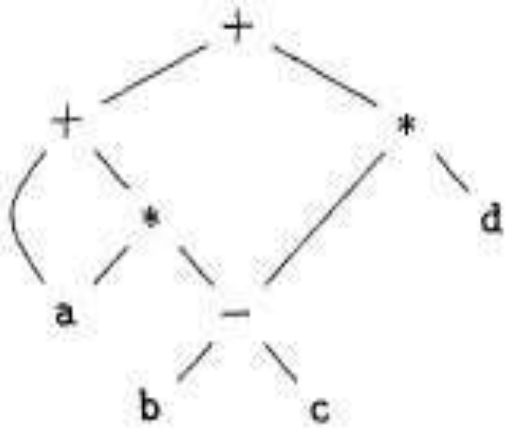
Why intermediate code ?

- Details of the source language are confined to the front end (analysis phase) of a compiler, while details of the target machine are confined to the back-end (synthesis) part.
- This saves a considerable amount of effort since with m front-ends and n back-ends we have m*n compilers
- **Intermediate representations**
- **Syntax Trees :** Code is represented in the form of a tree where nodes represent constructs in the source program; the children of a node represent the meaningful components of a construct.
- **Three-Address Code:** Made up of instructions of the general form x=y op z .
- X, y and z are the three addresses.

Three Address Code (TAC)

- An alternative form of intermediate (lower level) representation.
- x+y*x becomes t 1 = y * z , t 2 = x + t1
- For expressions TAC is very similar to syntax trees.
- For statements it would produce labels and jumps in a similar fashion to machine code

TAC for a+a*(b-c)+(b-c)*d



(a) DAG

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

(b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

Directed Acyclic Graphs

Nodes in a syntax tree represent constructs in the source program
A DAG is used to identify common
sub expressions. e.g. a+a*(b-c)+(b-c)*d
By doing so it gives the compiler important hints on how to generate efficient code to evaluate the expressions

DAG for a+a*(b-c)+(b-c)*d

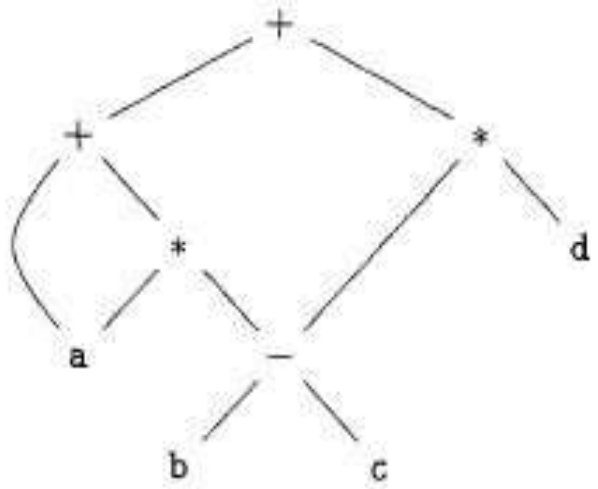


Figure 6.3: Dag for the expression a + a * (b - c) + (b - c) * d

Quadruples and Triples

- **Data structures to hold three address code instructions.**
- A Quadruple has four fields (x = y+z)
  - **Op (+)**
  - Arg1 (y)
  - Arg2 (z)
  - Result (x)

An example ...

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

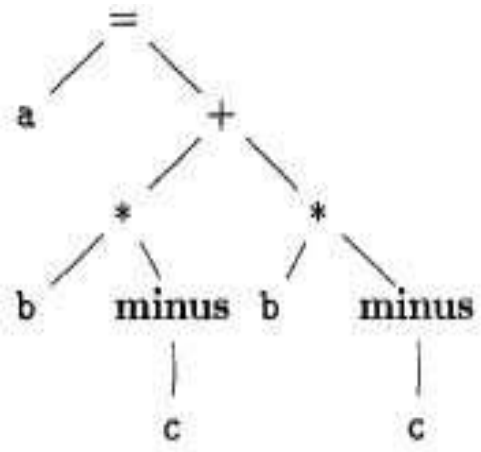
(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

Triples

- Omit result field.
- Instead of a result field we can use pointers to the triple structure itself.
- This makes DAG and triple representation practically identical, since we are pointing to a node.
- In next example (n) indicates position n in the triple structure

An example ...



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

Figure 6.11: Representations of $a + a * (b - c) + (b - c) * d$

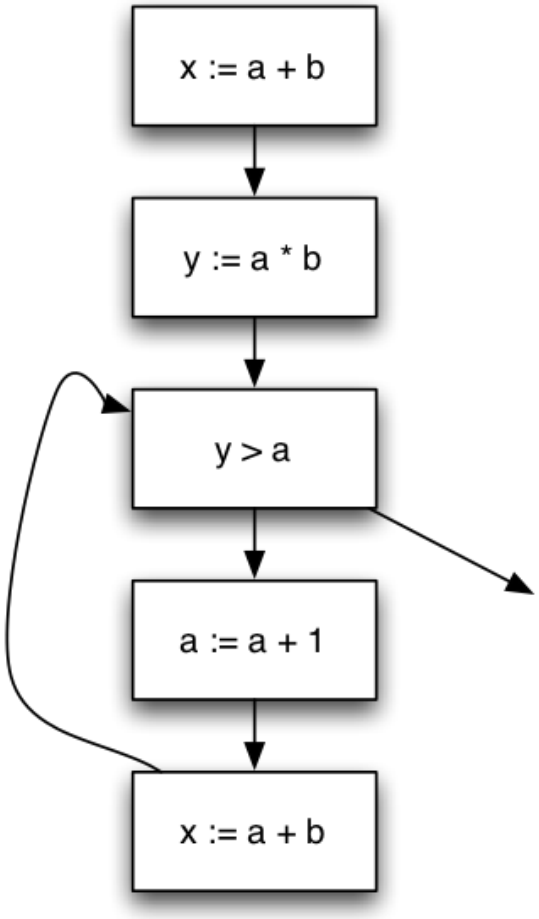
Control Flow

- A directed graph where
  - Each node represents a statement
  - Edges represent control flow

- Statements may be
  - Assignments x = y op z or x = op z
  - Copy statements x = y
  - Branches goto L or if relop y goto L
  - etc

Control-flow Graph Example

```
x := a + b;
y := a * b
While (y > a){
  a := a +1;
   x := a + b
}
```



Back-patching

- The problem in generating three address codes in a single pass is that we may not know the labels that control must go to at the time jump statements are generated.So to get around this problem a series of branching statements with the targets of the jumps temporarily left unspecified is generated.
- **Back Patching** is putting the address instead of labels when the proper label is determined.
- Back patching Algorithms perform three types of operations
- 1) **makelist (i)** – creates a new list containing only **i,** an index into the array of quadruples and returns a pointer to the list it has made.
- 2) **Merge (i, j)** – concatenates the lists pointed to by **i** and **j,** and returns a pointer to the concatenated list.
- 3) **Backpatch (p, i)** – inserts **i** as the target label for each of the statements on the list pointed to by **p.**