Semantic Analysis

Syntax Directed Definition

- The **grammar** and the set of **semantic rules** constitute the syntax directed definition.
- A **syntax-directed translation** is used to define the translation of a sequence of tokens to some other value, based on a CFG for the input.
- An information is associated as attributes to the grammar symbols
- A syntax directed definition (SDD) specifies the values of attributes associating semantic rules with the grammar productions

Some definitions

- Annotated Parse Trees

  A parse tree showing the attribute values at each node is called an annotated parse tree.

- Synthesized Attributes

  An attribute is said to be synthesized if its value at a parse tree node is determined from attribute values at the children of the node

- Inherited attributes

  An inherited attribute is one whose value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of that node.

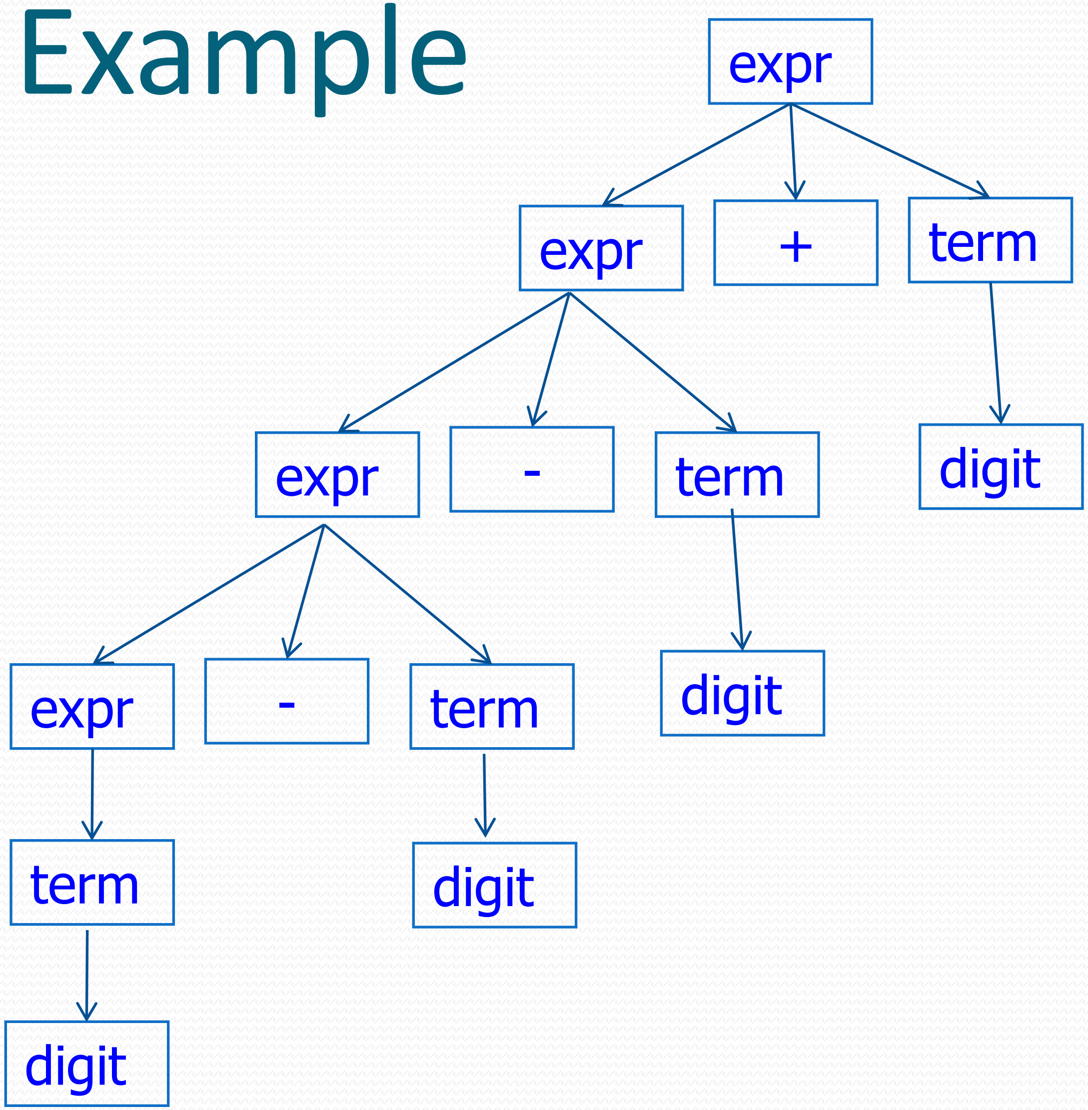
Synthesized Attributes

- **Synthesized attributes** are attributes that are computed purely bottom-up
- Such a grammar plus semantic actions is called an **S-attributed definition**

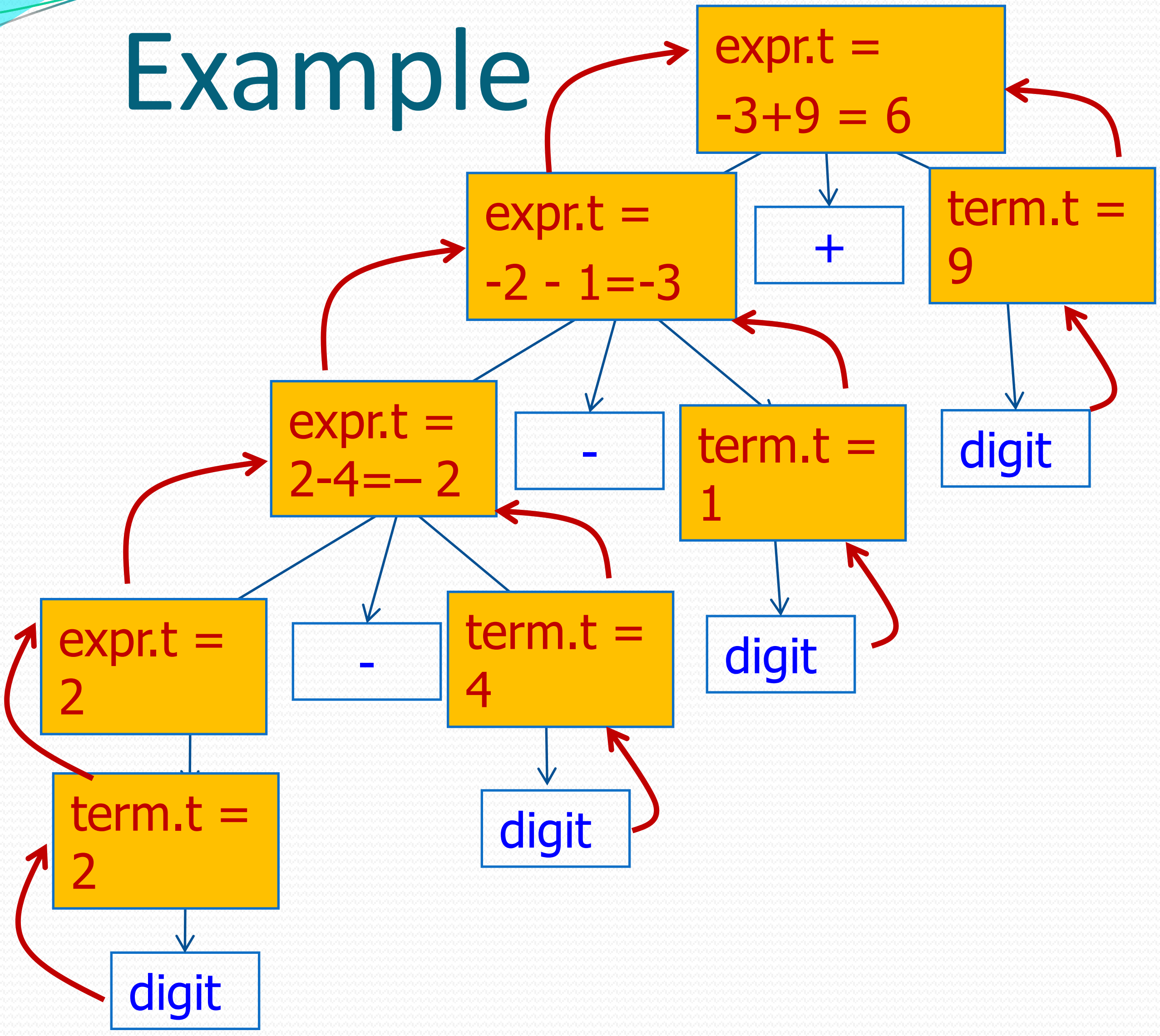
Understanding Translation:
syntax directed definition for evaluating an expression

	Production	Semantic rule
1	expr→ $expr_L$ + term	expr.t:= $expr_L$.t + term.t
2	expr→ $expr_L$ - term	expr.t:= $expr_L$.t - term.t
3	expr→ term	expr.t := term.t
4	term→digit	term.t := digitval (=getValue(ST, digit))

Example


expr
expr
+
term
expr
-
term
digit
expr
-
term
digit
term
digit
digit

Example



expr.t =
-3+9 = 6
expr.t =
-2 - 1=-3
+
term.t =
9
expr.t =
2-4=– 2
-
term.t =
1
digit
expr.t =
2
-
term.t =
4
digit
term.t =
2
digit
digit

Computing The Translation

To compute the s ynthesized attributed translation of a string,

- Build the parse tree,
- Use the translation rules to compute the translation of each nonterminal in the tree, bottom-up
- The translation of the string is the translation of the root nonterminal.

Example : Compute the type of an expression

- Compute the type of an expression that includes both arithmetic and boolean operators. (The type is INT, BOOL, or ERROR.)
- First define CFG Rules
- Determine meaning (semantic) of each rule

Context free grammar rules with SDD

1. exp -> exp + exp

```
if ((exp2.t == INT) and (exp3.t == INT)
then exp1.t = INT
else exp1.t = ERROR
```

2. exp -> exp and exp

```
if ((exp2.t == BOOL) and (exp3.t ==BOOL)
then exp1.t = BOOL
else exp1.t = ERROR
```

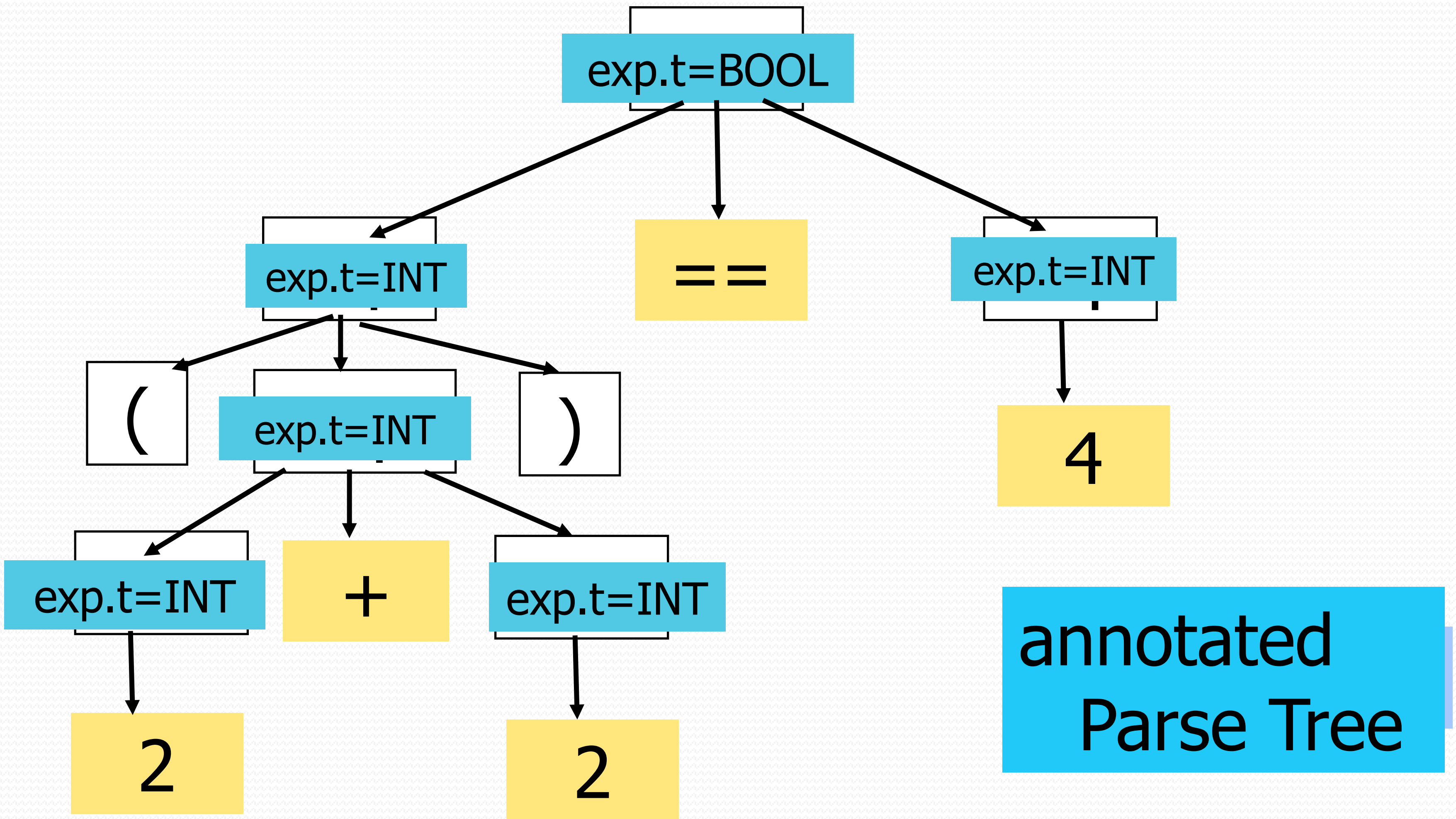
3. exp -> true

```
exp.t = BOOL
```

Context free grammar rules with SDD

4. exp -> false	exp.t = BOOL
5. exp -> int	exp.t = INT
6. exp -> (exp)	exp1.t = exp2.t
7. exp -> exp == exp	if ((exp2.t == exp3.t) and (exp2.t != ERROR)) then exp1.t = BOOL else exp1.t = ERROR

Generate annotated parse tree for (2+2)==4



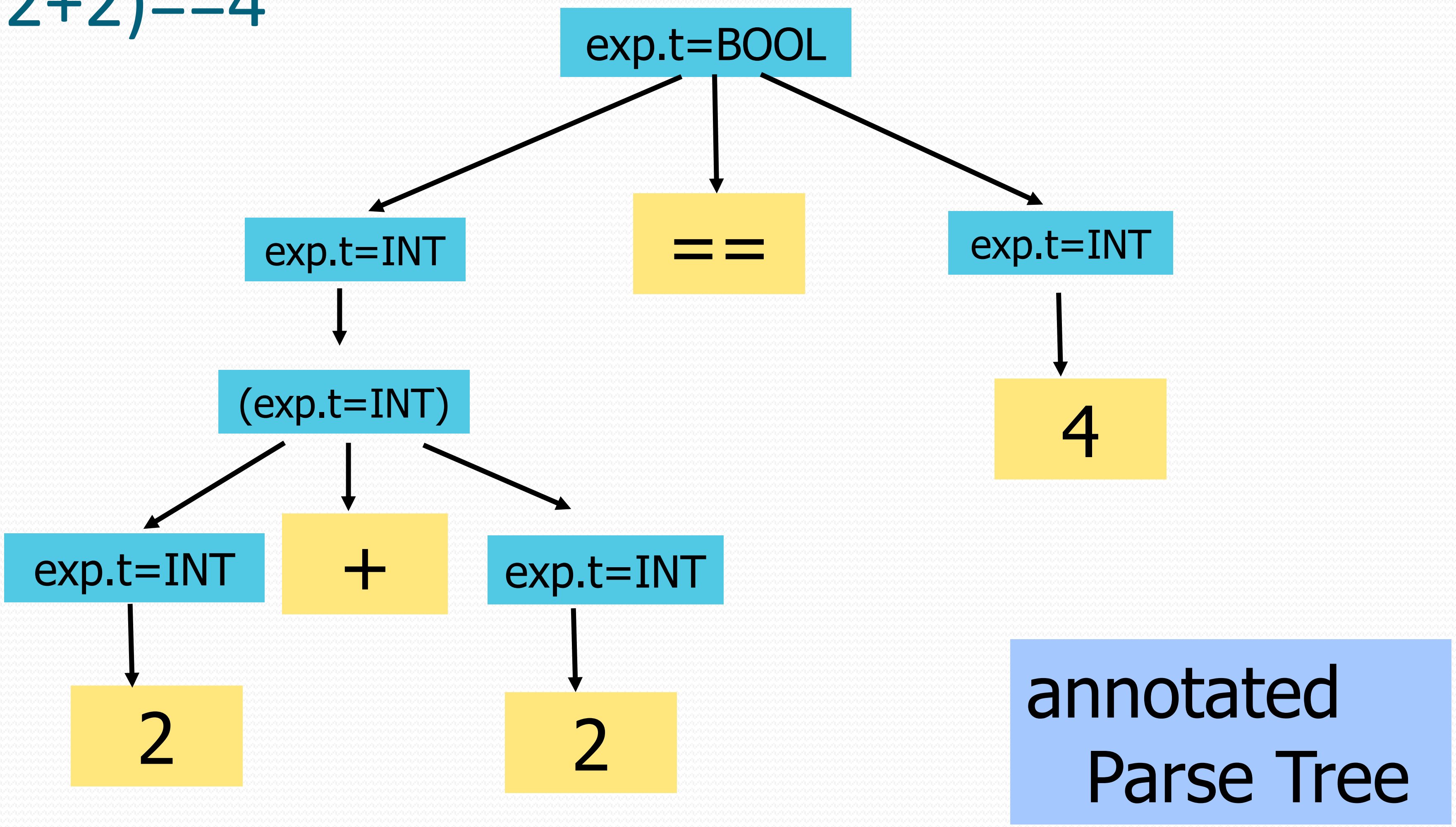
Translation Schemes

- A translation scheme is a context free grammar embedded with semantic actions
- Semantic actions are the program fragments embedded within the right sides of productions

Annotated Parse Tree

- A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node bottom up, from the leaves to the root.

Example of annotated parse tree for (2+2)==4



How to update the symbol table to keep the record of the type of the variables

Lexeme	Token	type	……….	……
Int	INT			
num1	id			
num2	id			
char	CHAR			
c	id			

Use of inherited attributes evaluates the type

Lexeme	Token	type	………	……
Int	INT	integer		
num1	id	integer		
num2	id	integer		
char	CHAR	character		
c	id	character		

Define Grammar rules for declaration construct (sample)

1. D→TL
2. T→int
3. T→char
4. L→L,id
5. L→id

Attributes for the nonterminals

- Terminals={int, char, id, ‘,’}
- Non terminals={D,T,L}
- Attribute for T:type (T.type represents the type of the declared variables)
- Attribute for L: in (L.in represents the inherited attribute information)

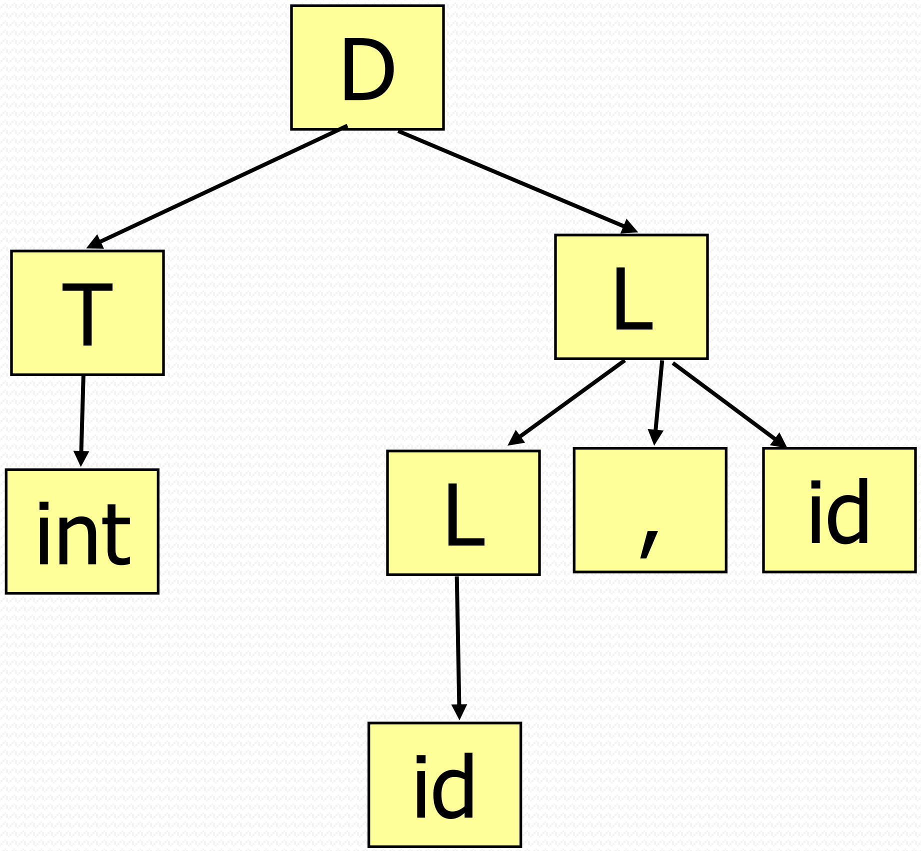
Grammar rules and associated semantic rules

1. $D \rightarrow TL$
2. $T \rightarrow int$
3. $T \rightarrow char$
4. $L \rightarrow L_1,id$
5. $L \rightarrow id$

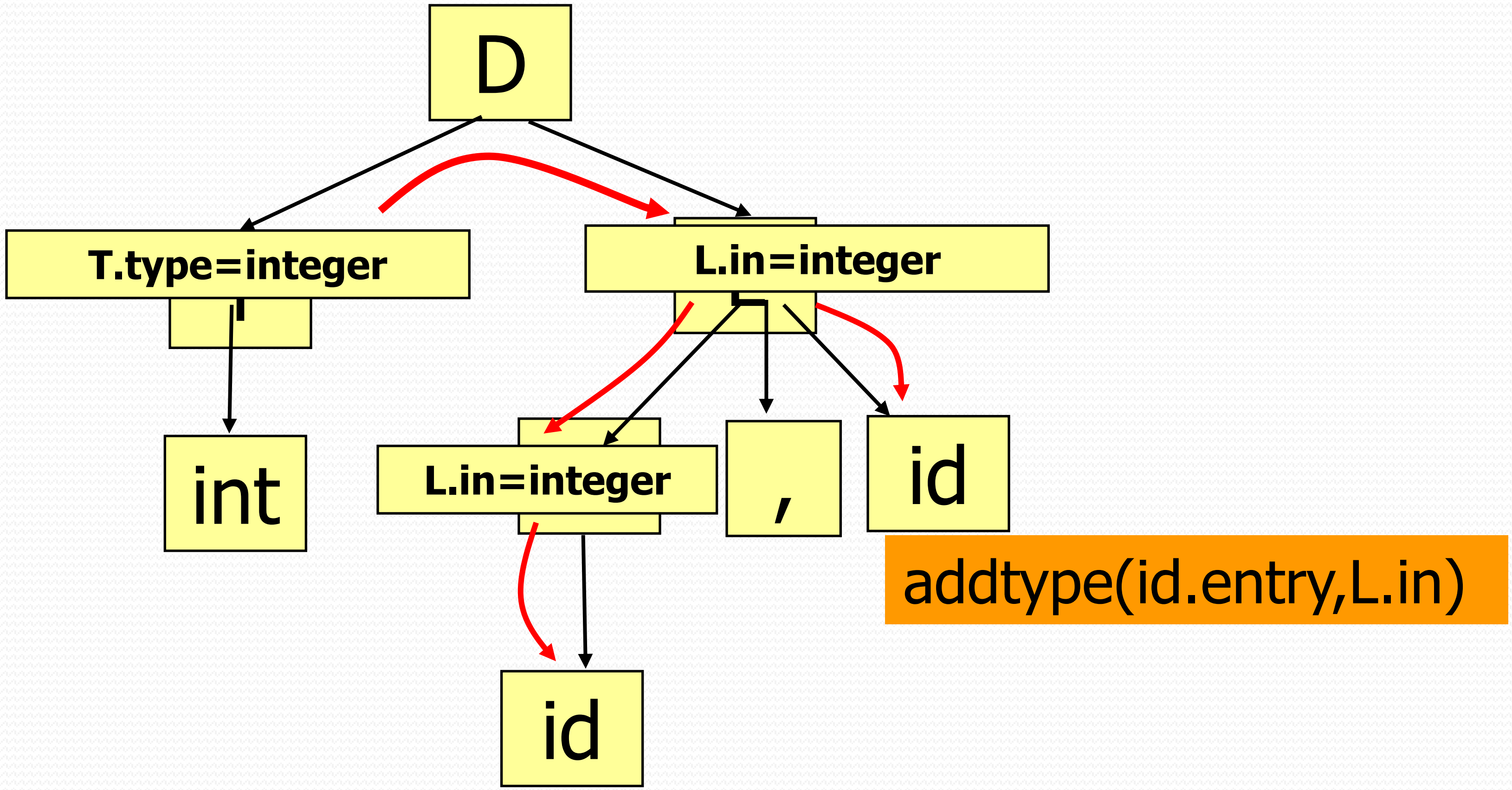
1. L.in=T.type
2. T.type=integer
3. T.type=character
4. L_1.in=L.in
   addtype(id.*entry*,L.*in*)
5. *addtype*(id.*entry*,L.*in*)

Parse tree for

int id,id



Annotated Parse tree for
INT id,id



Symbol table is updated for both token ‘id’ with type of these as integers

Attributes dependency

- Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears.
- If an attribute *b* at a node in a parse tree depends on an attribute *c*, then the semantic rule for *b* at that node must be evaluated after the semantic rule that defines *c*.
- Dependency graph depicts the interdependencies among the inherited and synthesized attributes at a node

Semantic Analyzer

- The principal job of the semantic analyzer is to enforce static semantic rules
  - Constructs a syntax tree (usually first)
  - Information gathered is then needed by the code generator
- An attribute's value at a given node depends on those of other predecessor or successor nodes

Methods for Evaluating Semantic Rules

- Parse Tree Methods
  - Most Flexible
  - But fail if a cycle exists in the dependency graph
- Rule Based Methods
  - Analyze rules at compiler-generation time
  - Determine a static ordering at that time
  - Evaluate nodes in that order at compile time
- Oblivious Methods
  - Ignore the parse tree and grammar
  - Choose a convenient order and use it

Parse-tree methods

1. Build the parse tree
2. Build the Abstract Syntax Tree (AST) to get attribute dependence
3. Build the dependency graph
4. Topological sort the graph
5. Evaluate it

Multi-Pass Approach

- Separate AST construction from semantic checking phase
- Traverse the AST and perform semantic checks (or other actions) only after the tree has been built and its structure is stable
- This approach is less error-prone and is better when efficiency is not a critical issue
- Attribute evaluation proceeds as tree-walk of the AST

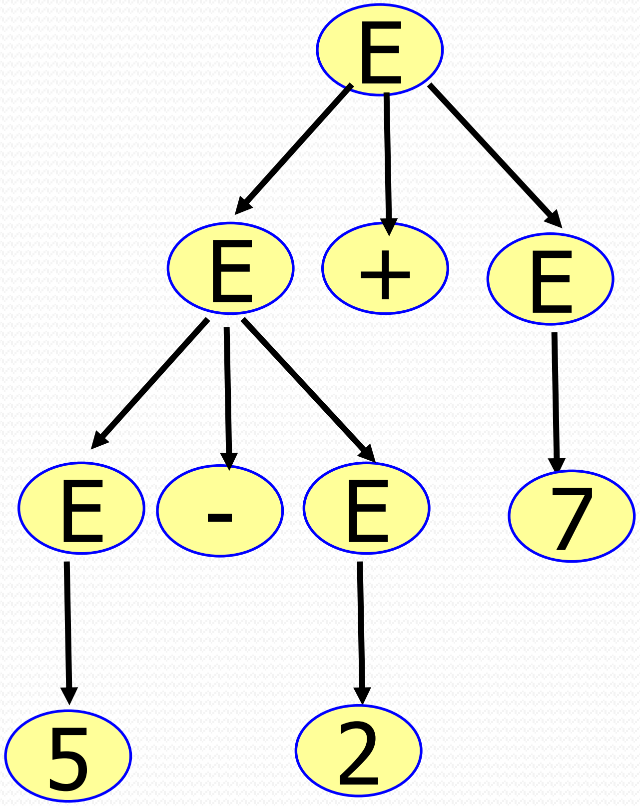
Examples of semantic rules

- Variables must be defined before being used
- A variable should not be defined multiple times
- In an assignment stmt, the variable and the expression must have the same type
- The test expression of an 'if statement' must have boolean type

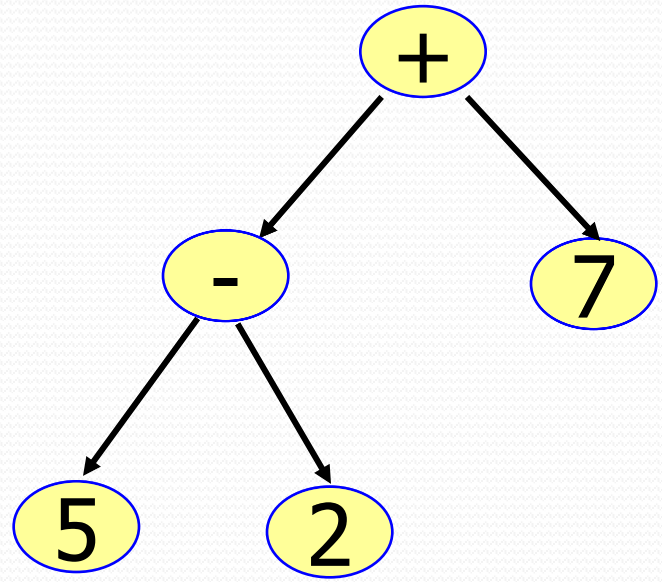
Abstract Syntax Trees

- An Abstract Syntax Tree is a condensed form of parse tree.
- It is useful for representing language constructs
- The keywords and operators do not appear as the leaves
- The syntax directed translation can be based on AST as well as the parse trees
- The attributes can be attached to the nodes in similar way as is done in parse trees

Example Parse tree and an AST



Parse Tree

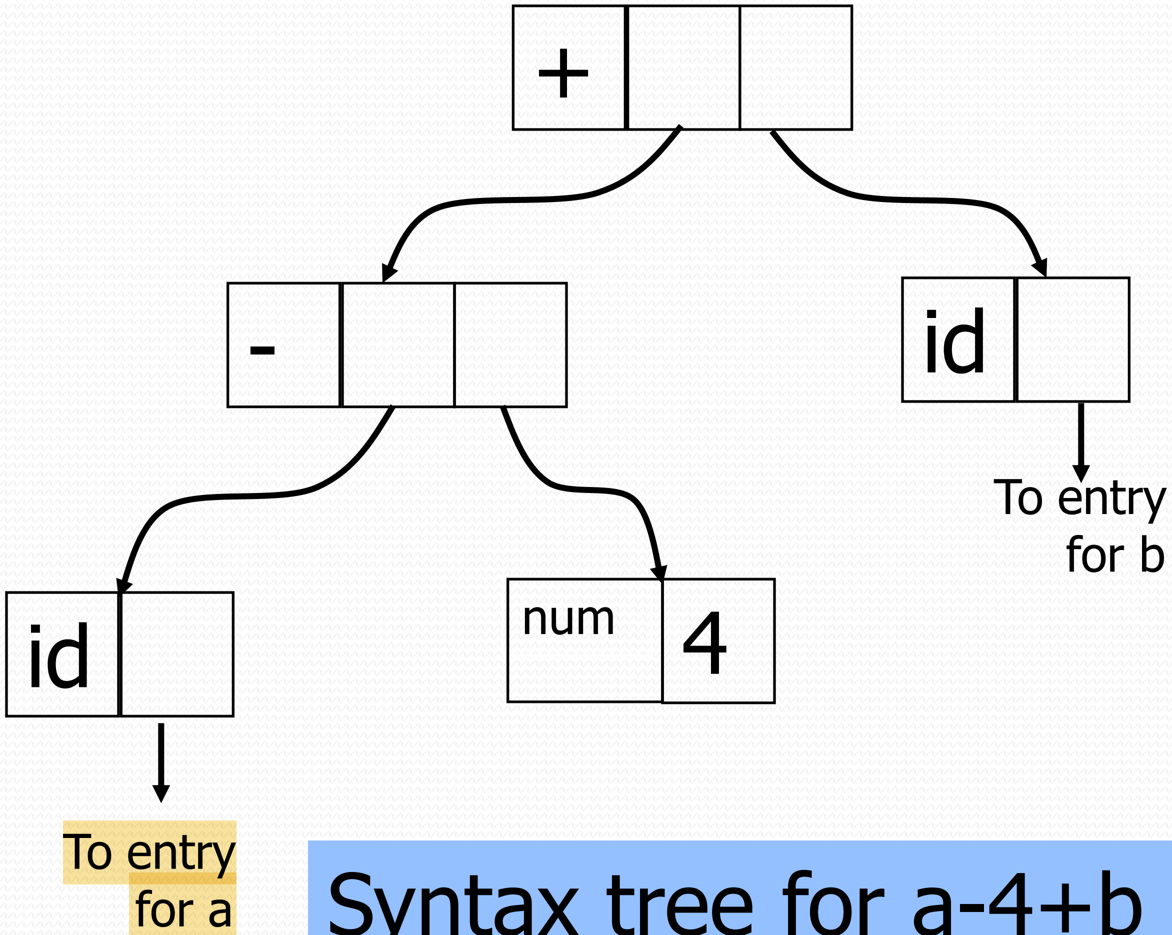


Abstract Syntax Tree

Construction of the Syntax Tree

- Similar to the translation of the infix expression into postfix form
- A node is created for each operator and operand
- The sub-expressions are represented by sub trees whose roots are the children of the operator nodes

Construction of the Syntax Tree



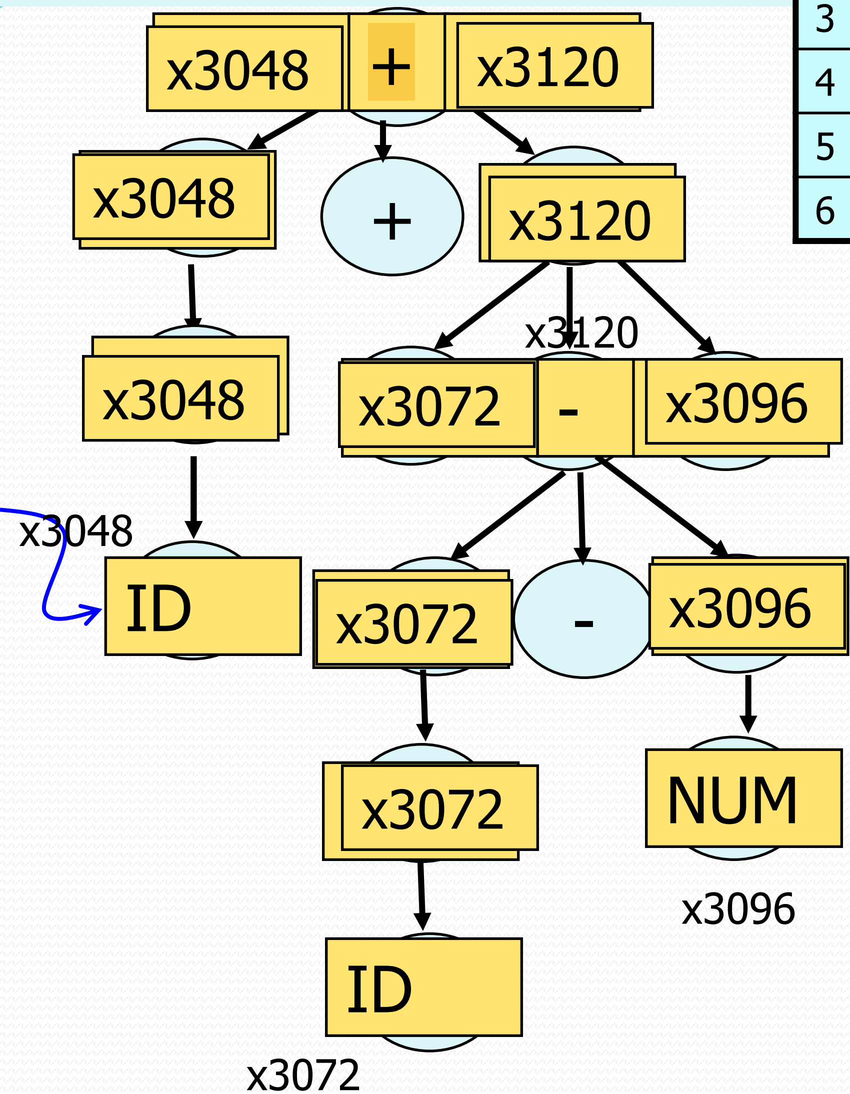
Syntax tree for a-4+b

Syntax Directed Definition To Construct Syntax Tree

Production	Semantic rules
E→E_1+T	E.nptr:=makenode('+',E_1.nptr,T.nptr)
E→E_1-T	E.nptr:=makenode('-',E_1.nptr,T.nptr)
E→T	E.nptr:=T.nptr
T→(E)	T.nptr:= E.nptr
T→id	T.nptr:=makeleaf(id,id.entry)
T→num	T.nptr:=makeleaf(num,num.val)

Constructing syntax tree

Rule no	Production	Semantic rules
1	$E \rightarrow E_1 + T$	E.nptr:=makenode('+',E_1.nptr,T.nptr)
2	$E \rightarrow E_1 - T$	E.nptr:=makenode('-',E_1.nptr,T.nptr)
3	$E \rightarrow T$	E.nptr:=T.nptr
4	$T \rightarrow (E)$	T.nptr:= E.nptr
5	$T \rightarrow$ id	T.nptr:=makeleaf(id,id.entry)
6	$T \rightarrow$ num	T.nptr:=makeleaf(num,num.val)



- Evaluated Bottom up while generating the parser
- The parser keeps the values of S-attributes along with the grammar symbols on its stack

AST vs Parse Tree

- AST is condensed form of a parse tree
- Operators appear at internal nodes, not at leaves.
- Chains of single productions are collapsed.
- Lists are "flattened“ into nodes with arbitrary number of children
- Omits details of concrete syntax (parenthesis, commas, semi-colons etc.)
- AST is a better structure for later compiler stages
- Contains information about essential structure of the program
- Source code is only a linearization of the AST

Source : google search

Semantic Analysis : other applications

- Three Address Code

  The general form is x = y **op** z

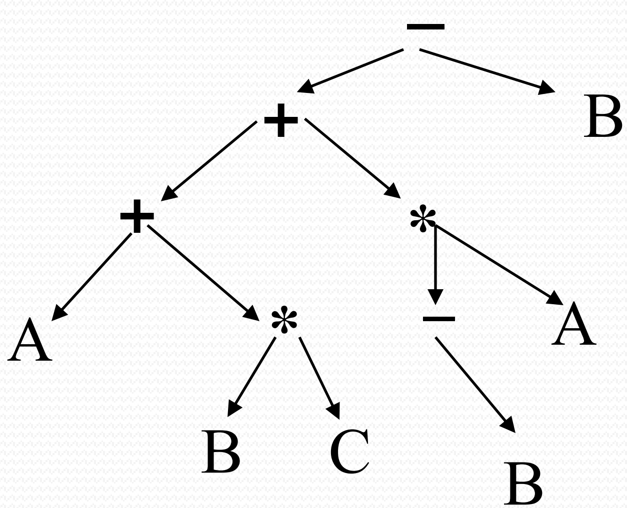
- x,y,and z are names, constants, compiler-generated temporaries
- **op** stands for any operator such as +,-,…
- (x*5)-y is translated as

t1 = x * 5

t2 = t1 - y

Syntax tree vs. Three address code

Expression: (A+B*C) + (-B*A) - B



T1 := B * C
T2 = A + T1
T3 = - B
T4 = T3 * A
T5 = T2 + T4
T6 = T5 – B

Three address code is a linearized representation of a syntax tree in which explicit names correspond to the interior nodes of the graph.

Semantic Analysis Applications: a list

- Generating Intermediate Representation (AST, Intermediate Code etc.)
- Collecting type information
- Type checking and error reporting
- Semantic errors reporting for undeclared variables
- Collecting Scope information
- Expression Evaluation
- Generating Code

Symbol table

- Symbol Table is populated with type and scope information, which in turn associates the offset for each identifier and is required at run time for allocating memory to the identifiers

More on Semantic Analysis

- Type checking, intermediate representation and code generation will be discussed later