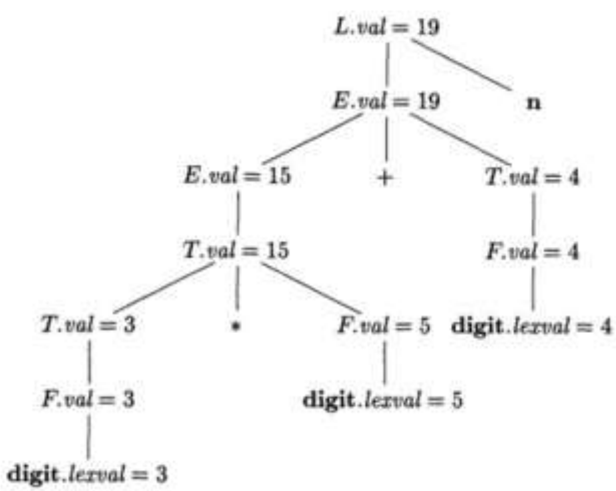
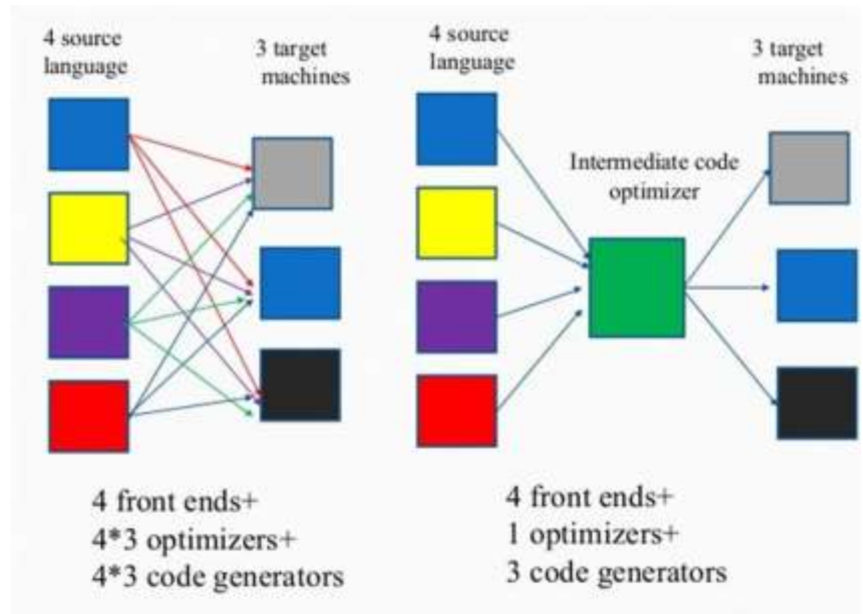
SNS COLLEGE OF TECHNOLOGY

(Autonomous)

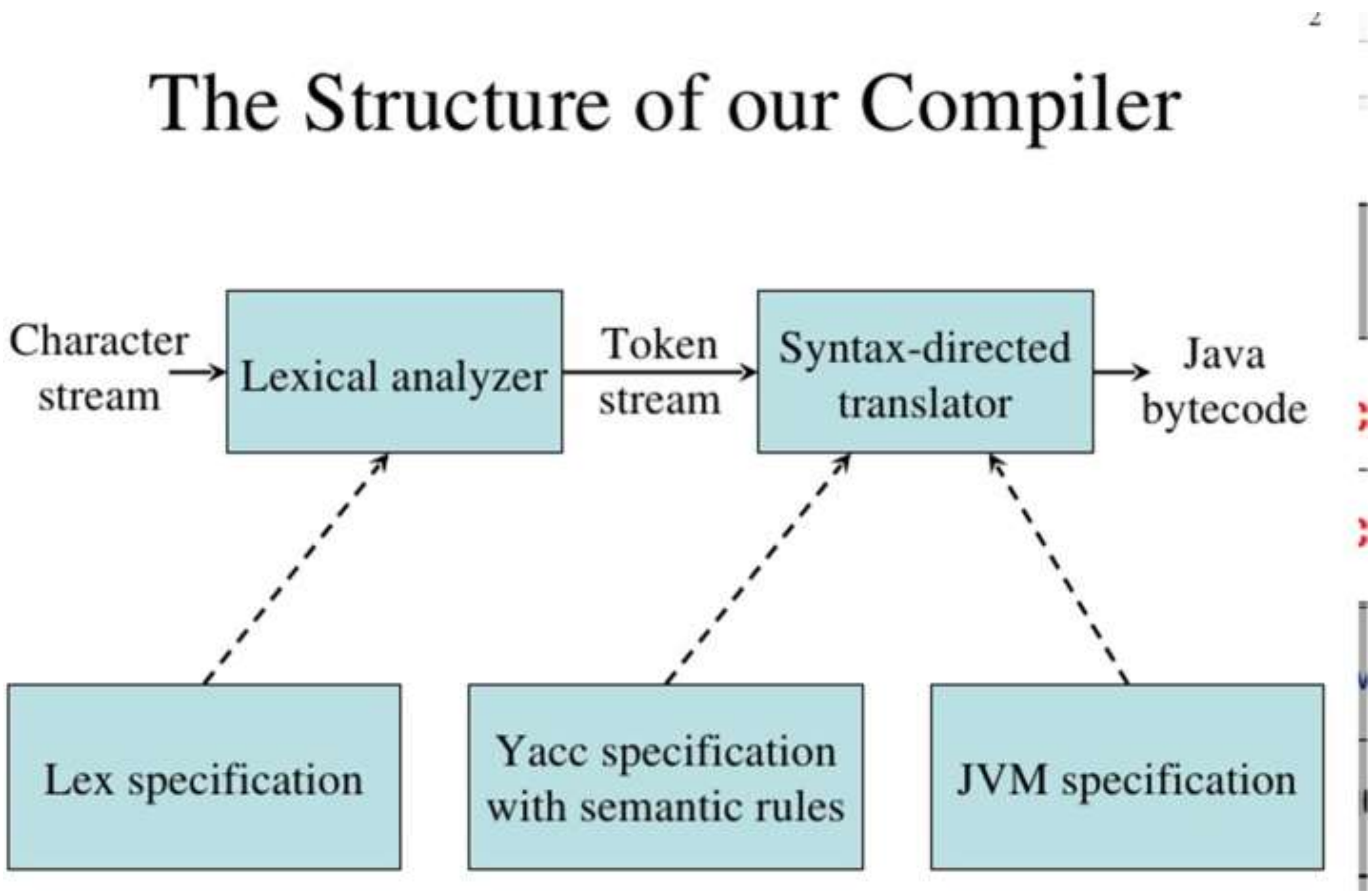
COIMBATORE-35

Syntax Directed Definitions &Intermediate Languages





Syntax Directed Definitions



Syntax Directed Definitions

Syntax-Directed Translation

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - in fact, they may perform almost any activities.
- An attribute may hold almost any thing.
  - a string, a number, a memory location, a complex record.

Syntax Directed Definitions & Translation Scheme

- When we associate semantic rules with productions, we use two notations:
  - **Syntax-Directed Definitions**
  - **Translation Schemes**
- **Syntax-Directed Definitions:**
  - give high-level specifications for translations
  - hide many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
- **Translation Schemes:**
  - indicate the order of evaluation of semantic actions associated with a production rule.
  - In other words, translation schemes give a little bit information about implementation details.

Syntax Directed Definitions

Syntax-Directed Definitions

- A syntax-directed definition is a generalization of a context-free grammar in which:
  - Each grammar symbol is associated with a set of attributes.
  - This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited** attributes of that grammar symbol.
  - Each production rule is associated with a set of semantic rules.
- *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.
- This *dependency graph* determines the evaluation order of these semantic rules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.
- Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Syntax Directed Definitions

- In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

  $b=f(c_1,c_2,\ldots,c_n)$ where f is a function,

  and b can be one of the followings:

  - b is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

  OR

  - b is an inherited attribute one of the grammar symbols in α (on the right side of the production), and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

Syntax Directed Definitions-Attribute Grammar

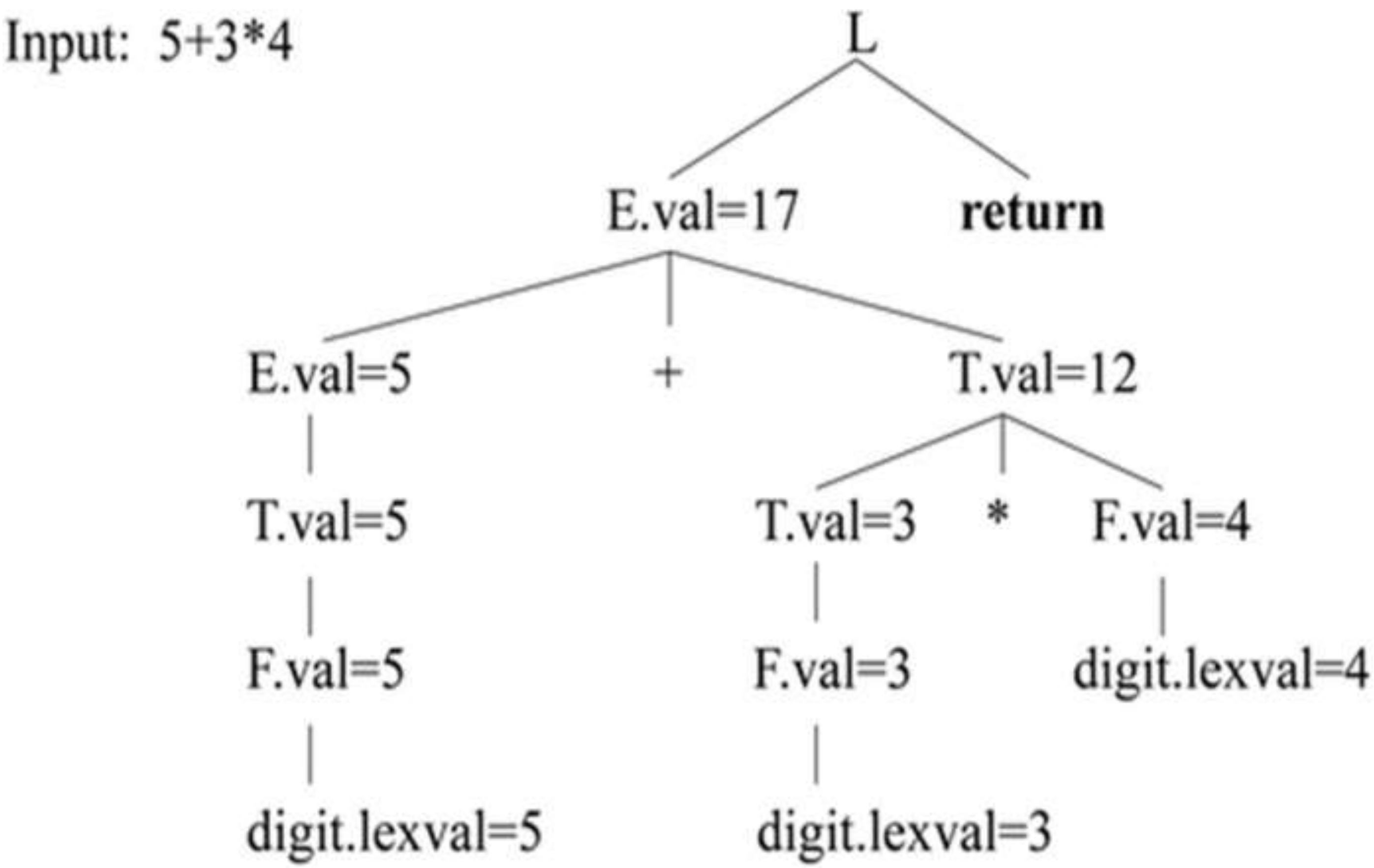
- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends on* attributes $c_1,c_2,\ldots,c_n$.
- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.
- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

Syntax Directed Definitions -Example

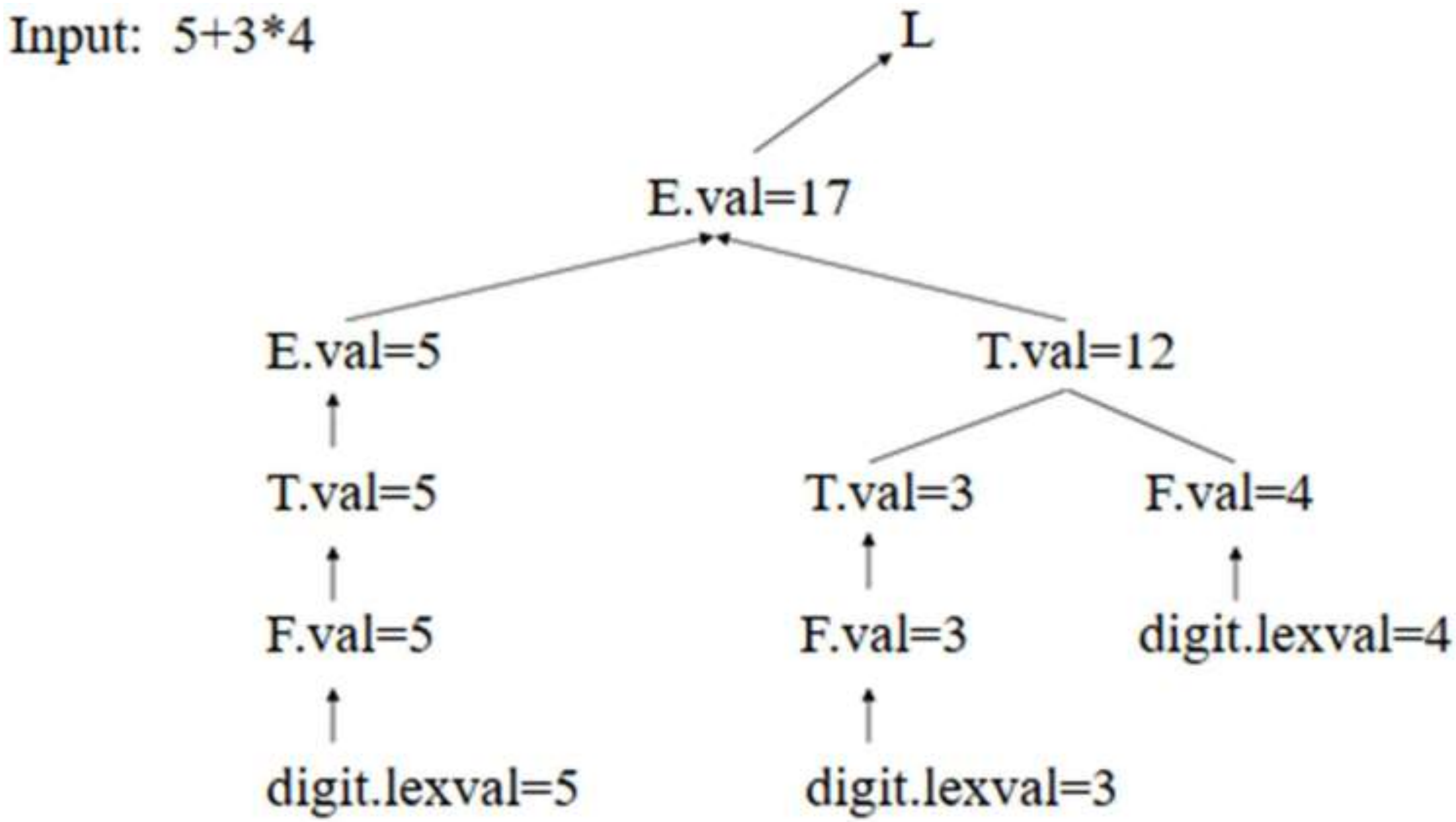
Production	**Semantic Rules**
$L \rightarrow E$ **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	F.val = **digit**.lexval

- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

Syntax Directed Definitions
Annotated Parse Tree Example

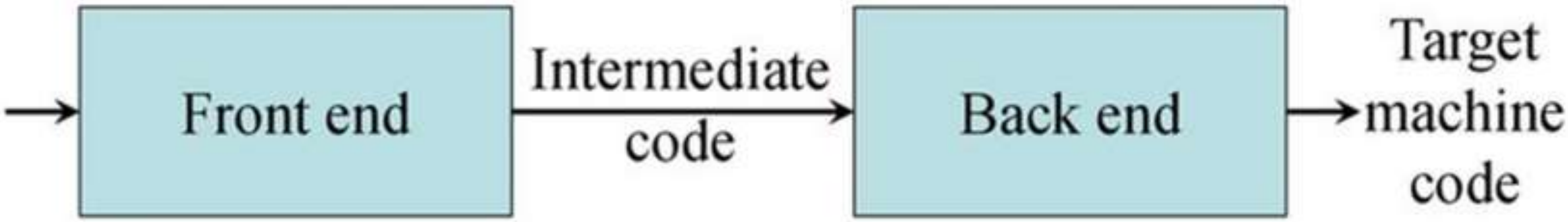


Syntax Directed Definitions
Dependency Graph



Intermediate Code Generation

- Facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end



- Enables machine-independent code optimization

Intermediate Representations

- *Graphical representations* (e.g. AST)
- *Postfix notation*: operations on values stored on operand stack (similar to JVM bytecode)
- *Three-address code*: (e.g. *triples* and *quads*)

  $x := y \text{ op } z$
- *Two-address code*:

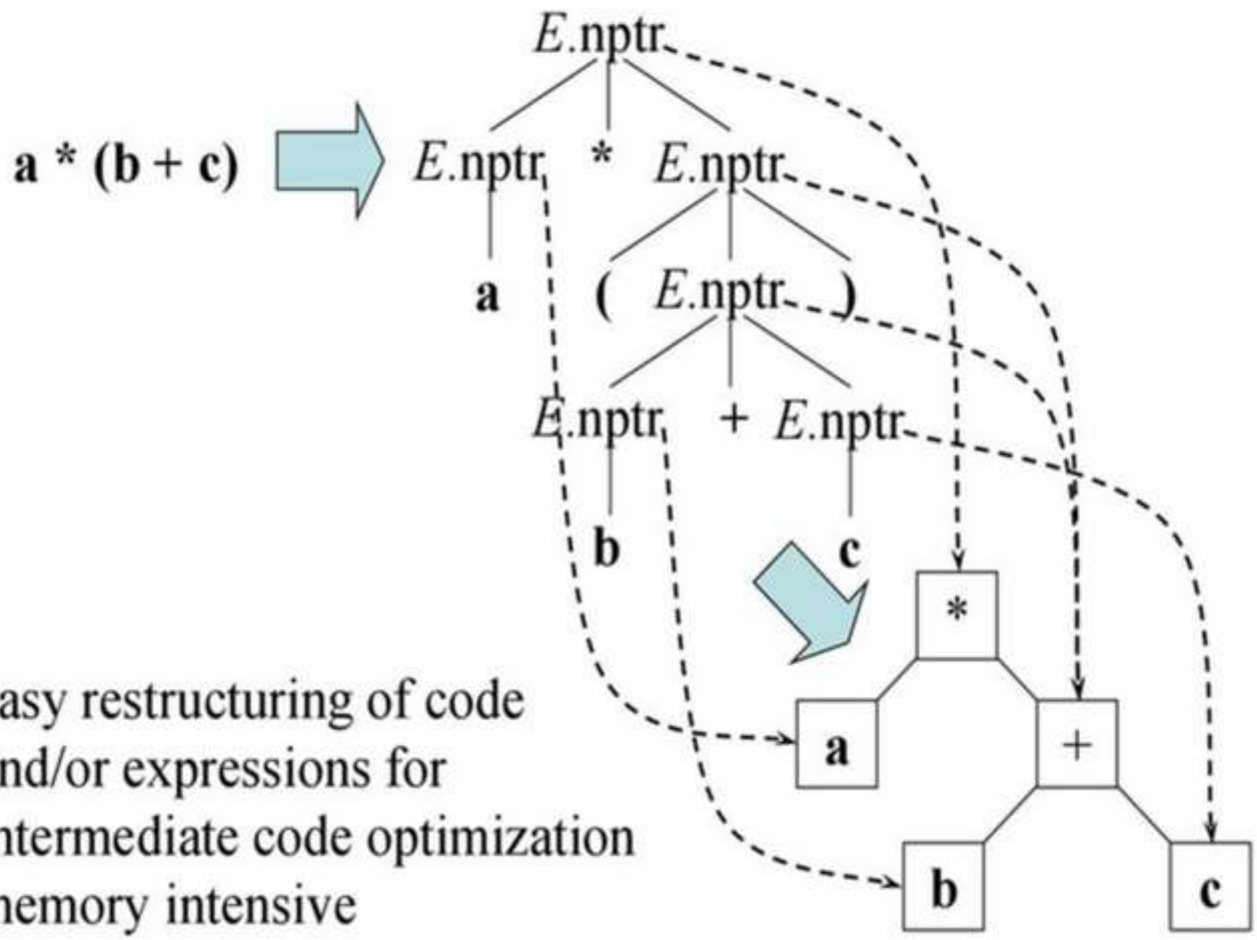
  $x := \text{op } y$

  which is the same as $x := x \text{ op } y$

Syntax Directed Translation of Abstract Syntax Trees

Production	Semantic Rule
$S \rightarrow \mathbf{id} := E$	S.nptr := *mknode*(':=' , *mkleaf*(**id**, **id**.entry), E.nptr)
$E \rightarrow E_1 + E_2$	E.nptr := *mknode*('+' , E_1.nptr, E_2.nptr)
$E \rightarrow E_1 * E_2$	E.nptr := *mknode*('*' , E_1.nptr, E_2.nptr)
$E \rightarrow - E_1$	E.nptr := *mknode*('uminus' , E_1.nptr)
$E \rightarrow (E_1)$	E.nptr := E_1.nptr
$E \rightarrow \mathbf{id}$	E.nptr := *mkleaf*(**id**, **id**.entry)

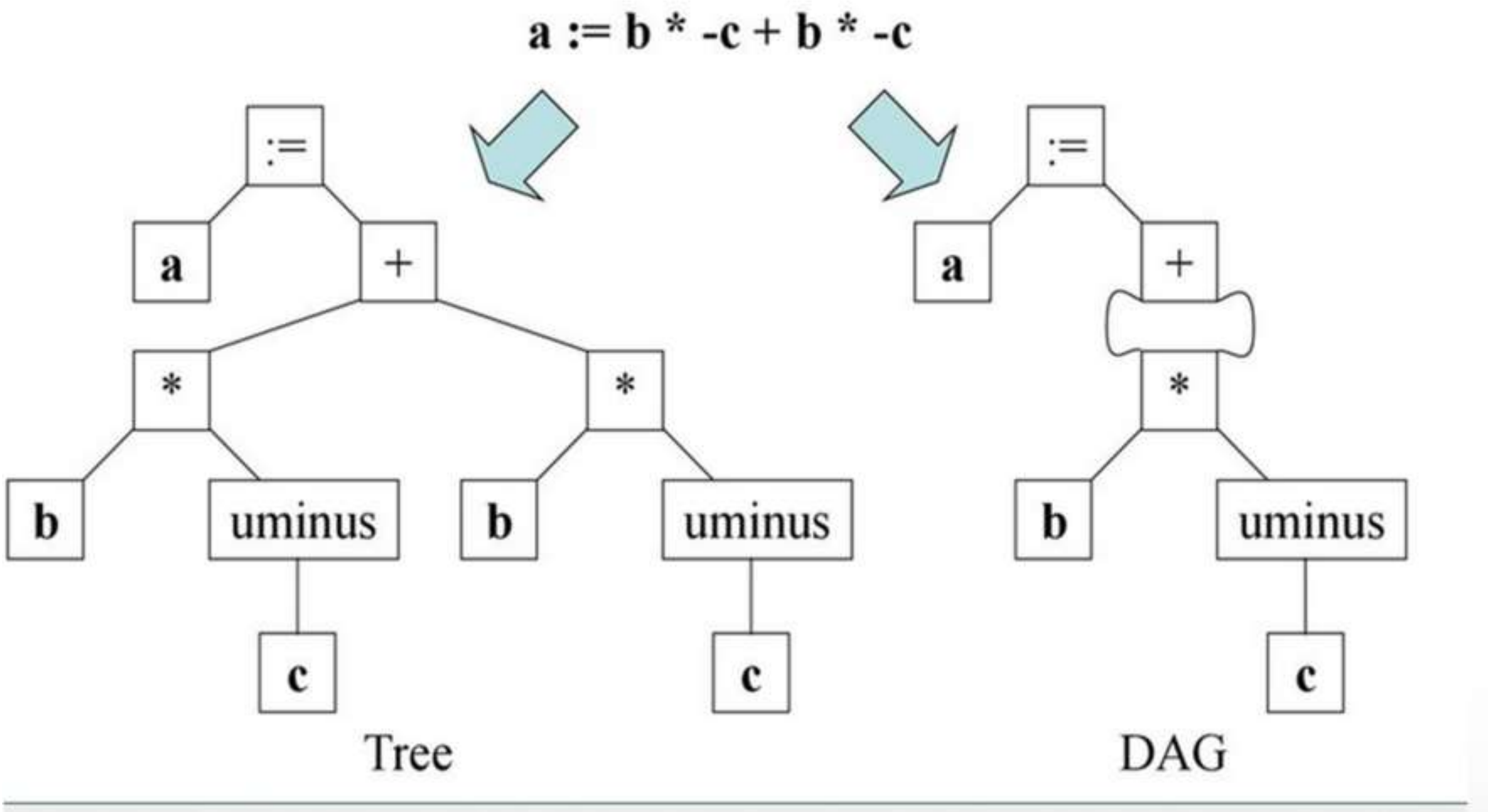
Abstract Syntax Tree



Pro: easy restructuring of code and/or expressions for intermediate code optimization

Cons: memory intensive

Abstract Syntax Tree Versus DAG



Postfix Notation

a := b * -c + b * -c

a b c uminus * b c uminus * + assign

Postfix notation represents operations on a stack

Pro: easy to generate
Cons: stack operations are more difficult to optimize

Bytecode (for example)

```
iload 2        // push b
iload 3        // push c
ineg           // uminus
imul           // *
iload 2        // push b
iload 3        // push c
ineg           // uminus
imul           // *
iadd           // +
istore 1       // store a
```

Three Address Code

a := b * -c + b * -c

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

Linearized representation of a syntax tree

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

Linearized representation of a syntax DAG

Three address Statements

- Assignment statements: $x := y \; op \; z$, $x := op \; y$
- Indexed assignments: $x := y[i]$, $x[i] := y$
- Pointer assignments: $x := \&y$, $x := *y$, $*x := y$
- Copy statements: $x := y$
- Unconditional jumps: `goto` *lab*
- Conditional jumps: `if` *x relop y* `goto` *lab*
- Function calls: `param` *x*... `call` *p*, *n*
  `return` *y*

Syntax Directed Translation into Three address code

Productions

$S \rightarrow$ **id** := E
  | **while** E **do** S
$E \rightarrow E + E$
  | E * E
  | - E
  | (E)
  | **id**
  | **num**

Synthesized attributes:

S.code	three-address code for S
S.begin	label to start of S or nil
S.after	label to end of S or nil
E.code	three-address code for E
E.place	a name holding the value of E

gen(E.place ':=' E_1.place '+' E_2.place)

Code generation → `t3 := t1 + t2`

Implementation of Three address Statements -Quadruples

#	*Op*	*Arg1*	*Arg2*	*Res*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Quads (quadruples)

Implementation of Three address Statements -Triples

#	*Op*	*Arg1*	*Arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Triples

Implementation of Three address Statements –Indirect Triple

#	*Stmt*
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Program

#	*Op*	*Arg1*	*Arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Triple container

Summarization