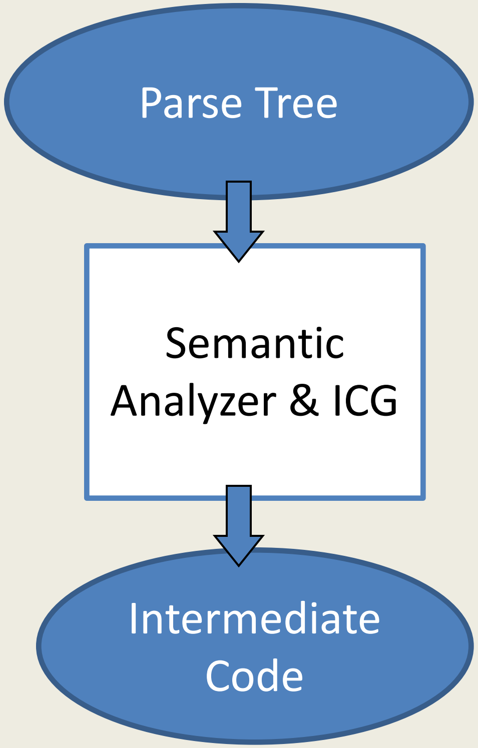
Semantic Analyzer & Intermediate Code Generation

Dinesh Gopalani
dgopalani.cse@mnit.ac.in

Semantic Analyzer & Intermediate Code Generation

Checks whether the code is semantically correct and then transforms into Intermediate Code



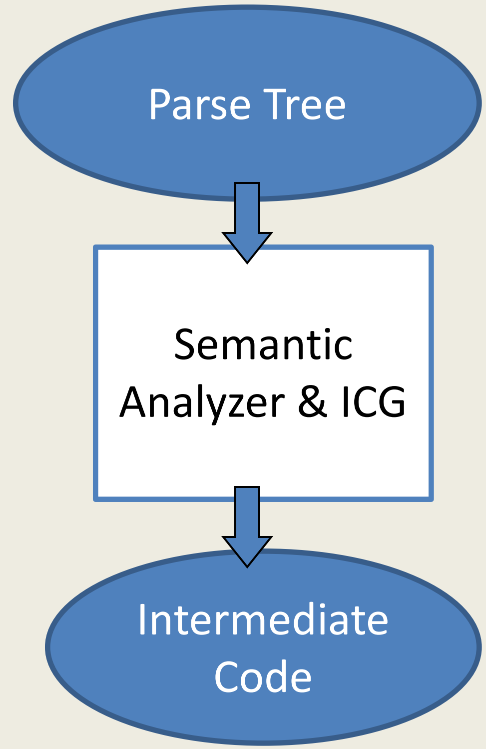
Semantic Analyzer & Intermediate Code Generation

Example :

```
...
...
...
i = 'a';
...
```

Syntax is correct

How about Semantics?

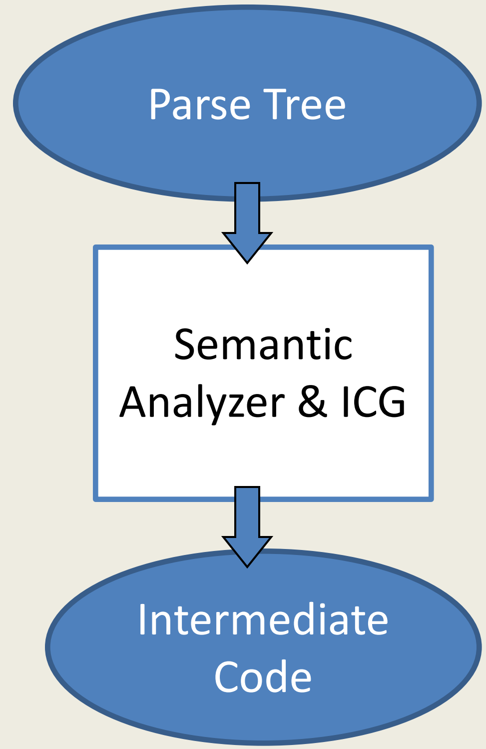


Semantic Analyzer & Intermediate Code Generation

Example :

```
int i;
...
...
i = 'a';
...
```

Context will decide



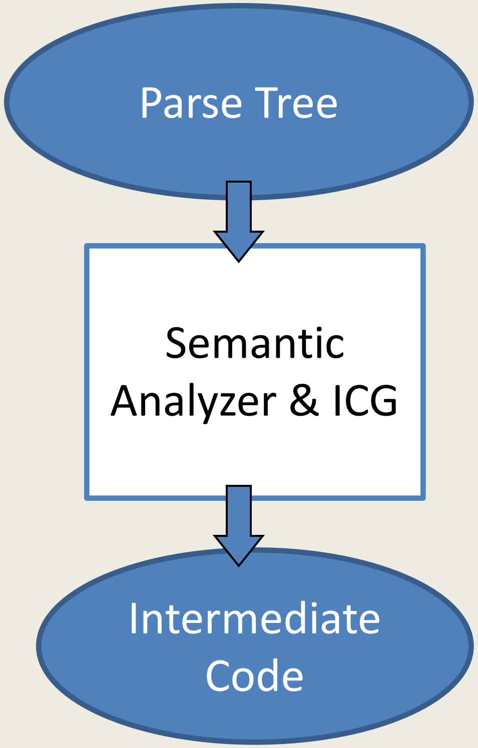
Semantic Analyzer & Intermediate Code Generation

Example :

```
int main()
{
        ...
        ...
        ...
    f(10, 20);
        ...
}
        ...
        ...
```

Syntax is correct

How about Semantics?

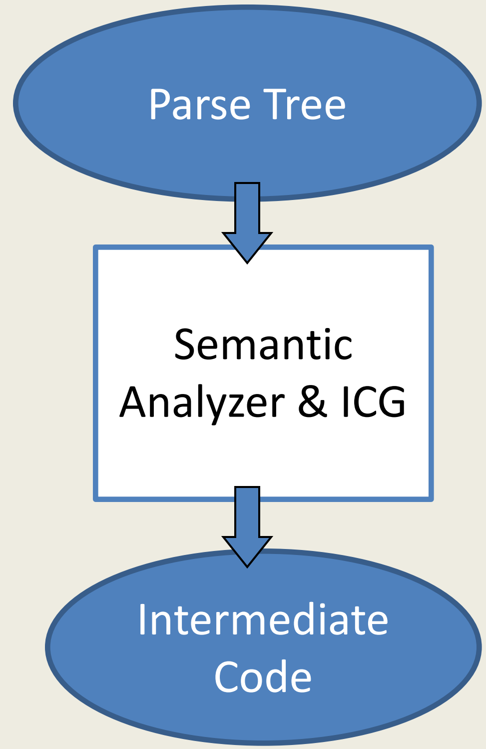


Semantic Analyzer & Intermediate Code Generation

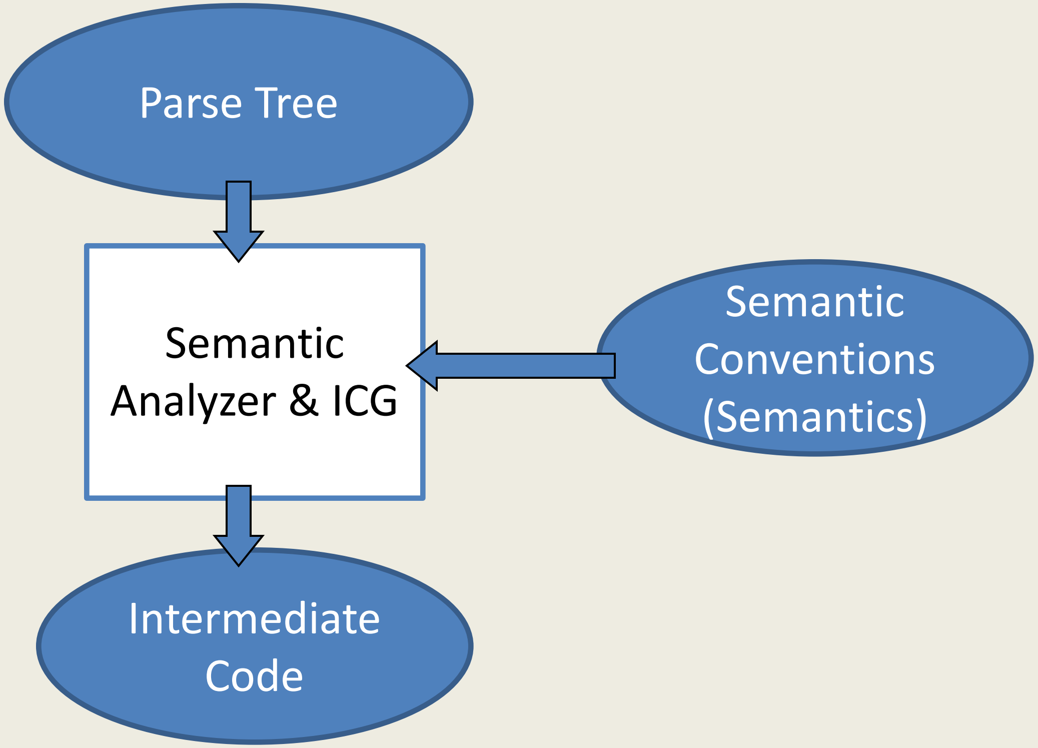
Example :

```
int main()
{
        ...
        ...
        ...
    f(10, 20);
        ...
}
void f(int a)
{
        ...
        ...
}
```

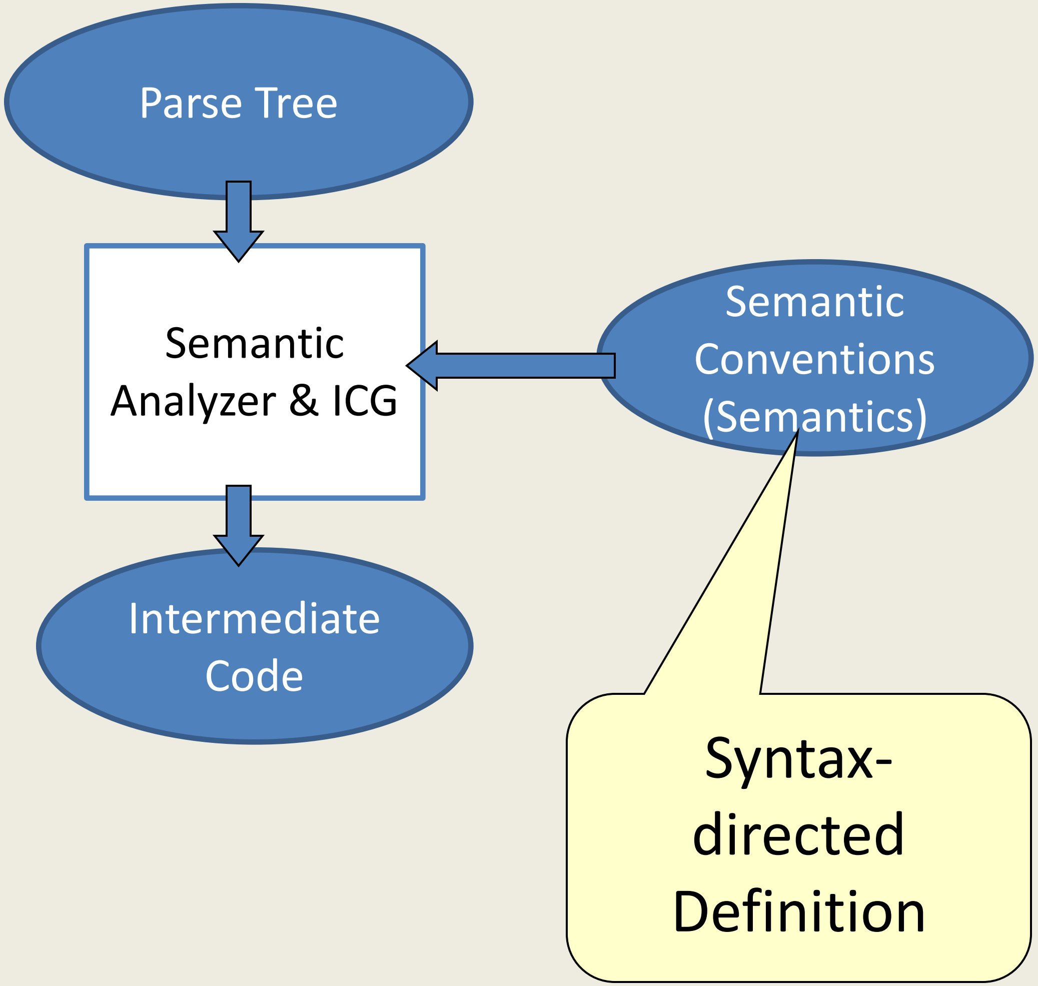
Context will decide



How Semantic Analyzer & ICG works?



How Semantic Analyzer & ICG works?



Syntax-directed Definition

- Extension to CFG.
- with each grammar symbol, a set of Attributes are defined.
- With each production, a Semantic Action is associated.
- A Semantic Action computes the values of attributes associated with the symbols appearing in that production.

Syntax-directed Definition

Example:

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow \text{digit}$

$E \rightarrow E^{(1)} + E^{(2)}$	$\{ E . Val = E^{(1)} . Val + E^{(2)} . Val \}$
$E \rightarrow E^{(1)} - E^{(2)}$	$\{ E . Val = E^{(1)} . Val - E^{(2)} . Val \}$
$E \rightarrow \text{digit}$	$\{ E . Val = \text{digit} . Lexval \}$

Syntax-directed Definition

Grammar G:

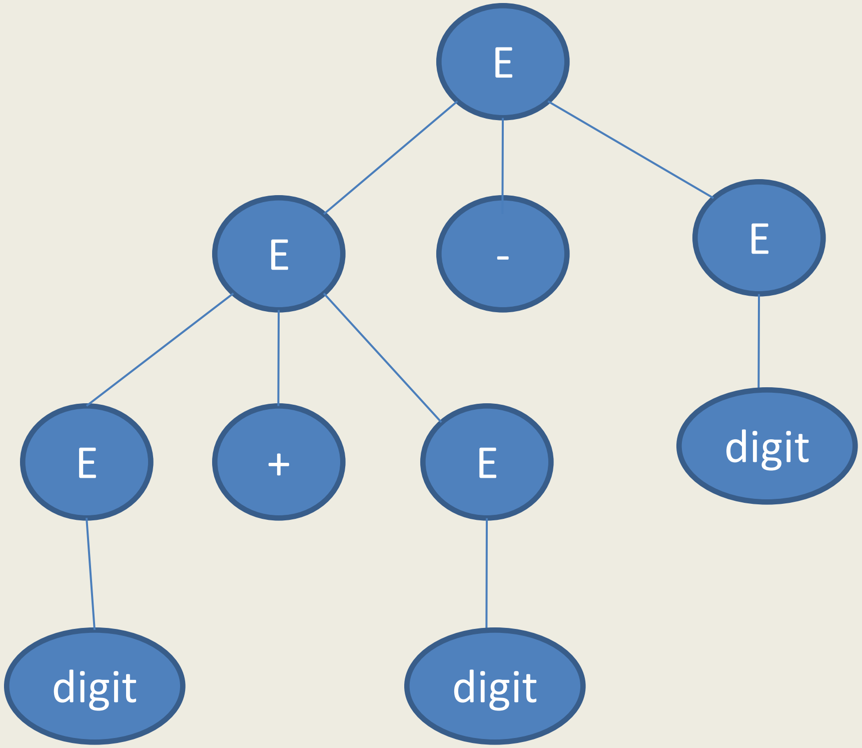
$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$

Sentence:

3 + 5 - 6



Syntax-directed Definition

Grammar G:

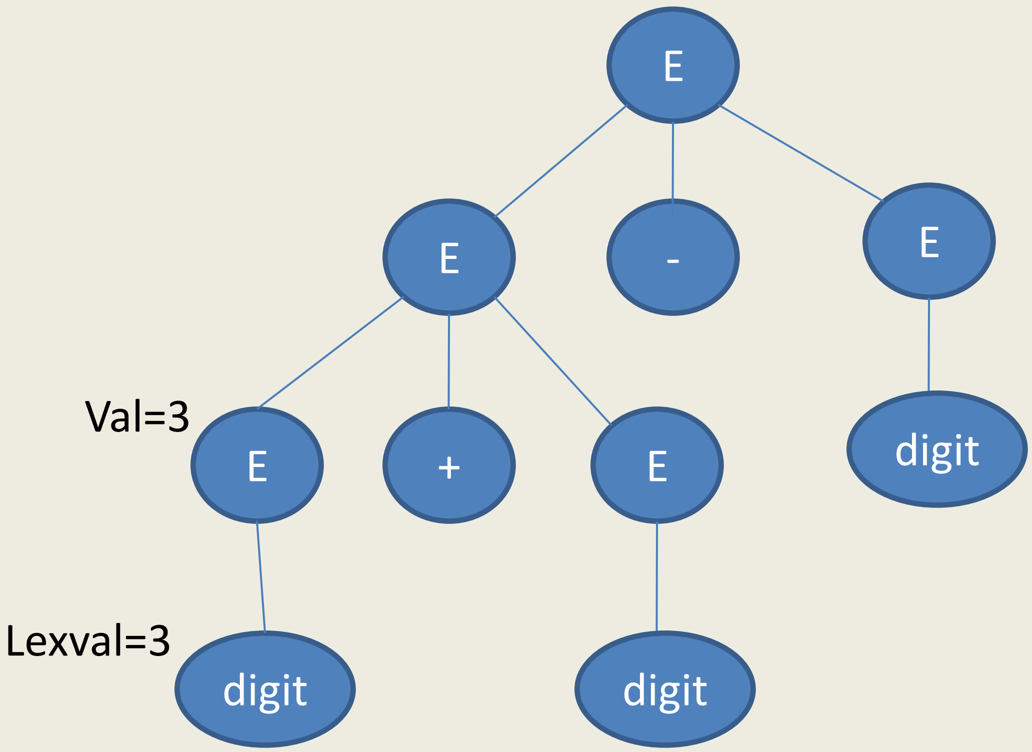
$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$

Sentence:

3 + 5 - 6



Syntax-directed Definition

Grammar G:

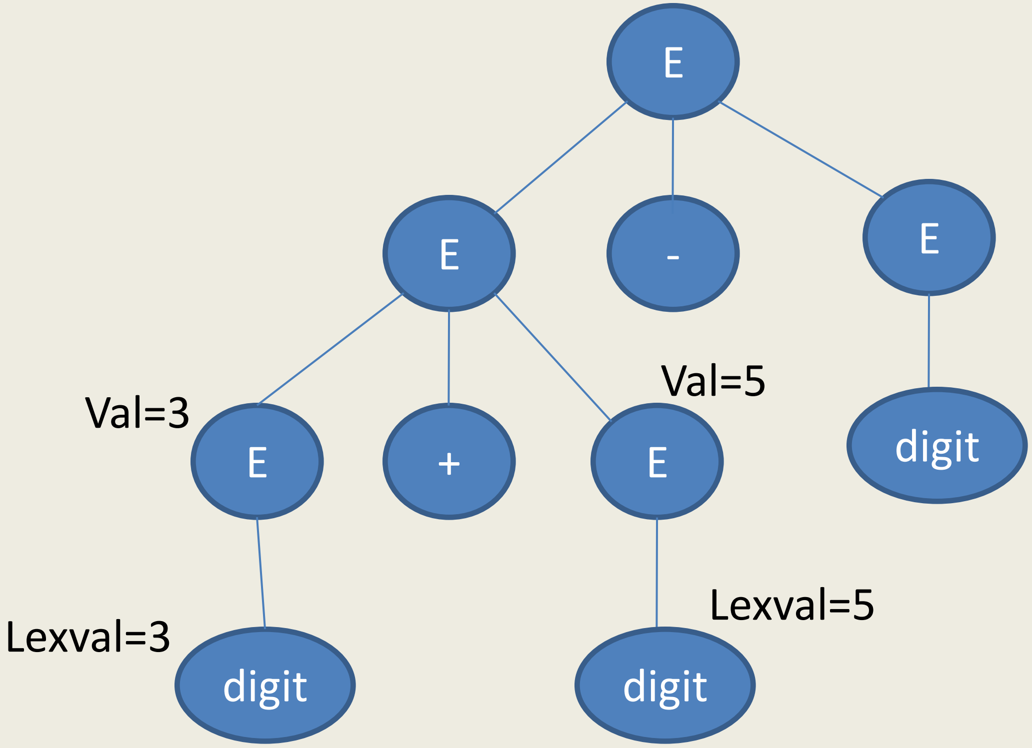
$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$

Sentence:

3 + 5 - 6



Syntax-directed Definition

Grammar G:

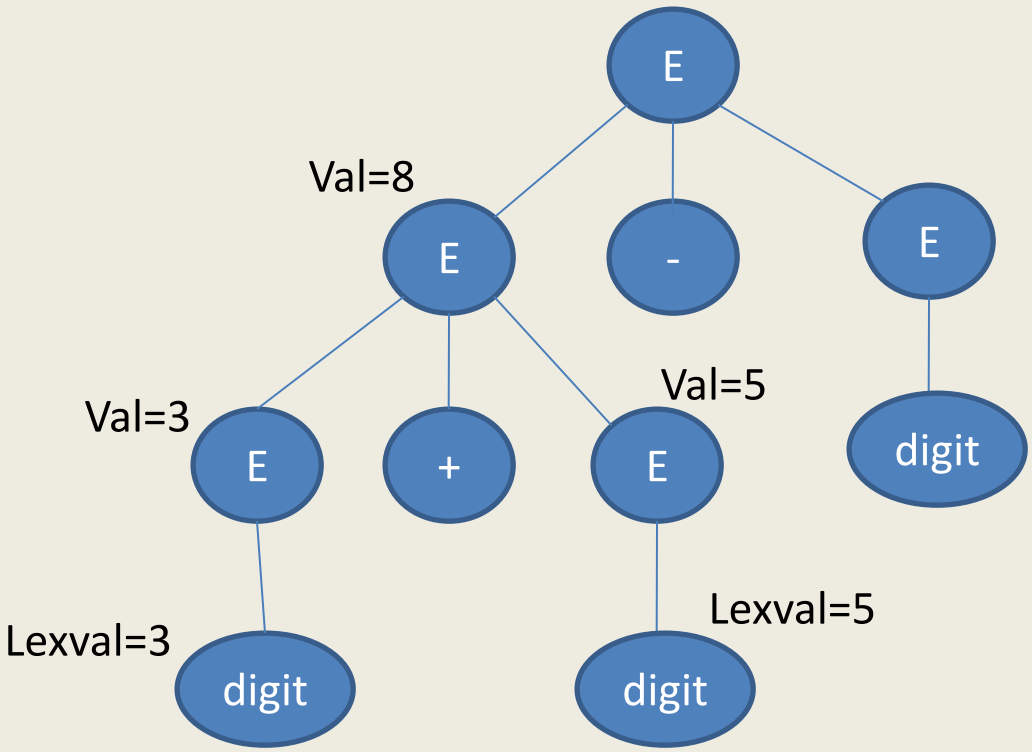
$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$

Sentence:

3 + 5 - 6



Syntax-directed Definition

Grammar G:

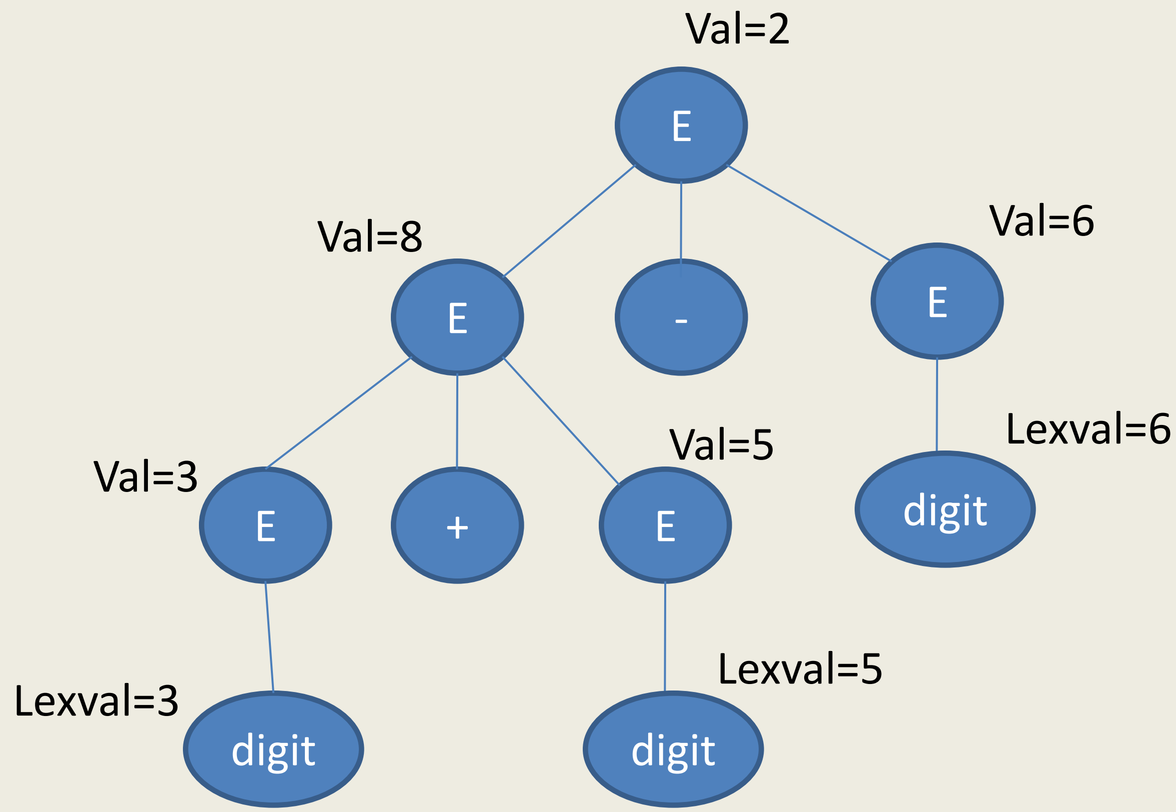
$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$

Sentence:

3 + 5 - 6



Attribute

1. Synthesized Attribute
2. Inherited Attribute

Synthesized Attribute

If semantic action associated with the production $A \rightarrow \alpha$ is of the form

$$b = f (c_1, c_2, \ldots, c_k)$$

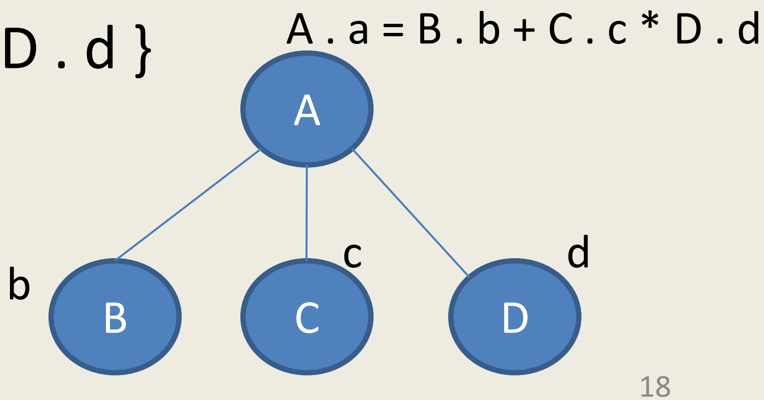
Here b is an attribute of A and c_1, c_2, …, c_k are attributes belonging to the grammar symbols on α.

Synthesized Attribute

Synthesized attribute is one whose value at a node in a parse tree is defined in terms of attributes at their children of that node.

Example:

$A \rightarrow B\ C\ D \qquad \{ A.a = B.b + C.c * D.d \}$



Inherited Attribute

If semantic action associated with the production $A \rightarrow \alpha$ is of the form

$$b = f(c_1, c_2, \ldots, c_k)$$

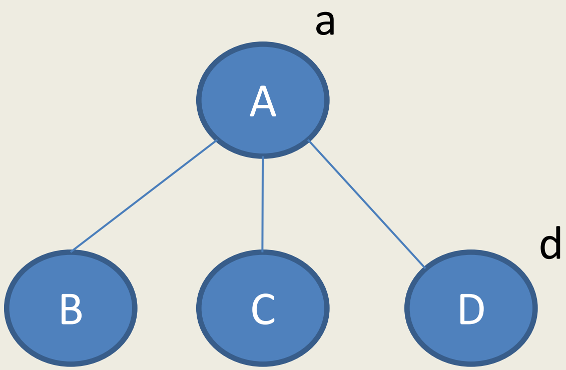
Here b is an attribute of one of the grammar symbols on the right-side of the production and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols the production (may be A also).

Inherited Attribute

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.

Example:

$A \rightarrow B\ C\ D \qquad \{ C.c = A.a + D.d \}$



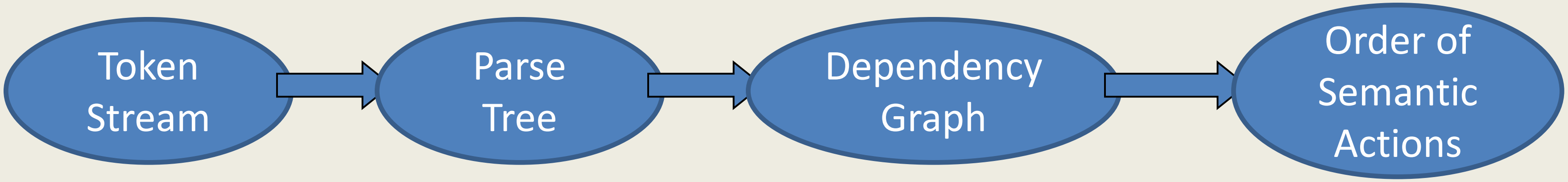
$C.c = A.a + D.d$

Implementation of Syntax-directed Definition

For an arbitrary Syntax-directed Definition can be difficult to build.

Because of dependencies among the synthesized and inherited attributes at the nodes in the parse tree.

Steps:



Implementation of Syntax-directed Definition

There are some classes of useful Syntax-directed Definitions for which implementation is easier.

1. S-attributed Definitions

2. L-attribute Definitions

Implementation of Syntax-directed Definition

There are some classes of useful Syntax-directed Definitions for which implementation is easier.

1. S-attributed Definitions

   (suitable for Bottom-up Parsers)

2. L-attribute Definitions

   (suitable for Top-down Parsers)

S-attributed Definition

A Syntax-directed definition is S-attributed if it involves synthesized attributes only.

Example:

$E \rightarrow E^{(1)} + E^{(2)}$	$\{ E . Val = E^{(1)} . Val + E^{(2)} .Val \}$
$E \rightarrow E^{(1)} - E^{(2)}$	$\{E . Val = E^{(1)} . Val - E^{(2)} .Val \}$
$E \rightarrow digit$	$\{ E . Val = digit . Lexval \}$

L-attributed Definition

A Syntax-directed definition is L-attributed if each inherited attribute of X_j ($j \varepsilon 1..n$) on the right-side of $A \rightarrow X_1\ X_2\ \ldots\ X_n$, depends only on

- the attributes of the symbols $X_1\ X_2\ \ldots\ X_{j-1}$ to the left of X_j in the production
- the inherited attribute of A

Implementation of S-attributed Definition

- Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed.
- The parser can keep the values of synthesized attributes associated with the grammar symbols on its stack.
- In case of reduce action, the values of synthesized attributes associated with the variable on left-side of the production are computed from the attribute values appearing on the stack for the symbols on right-side of the production.

Intermediate Code

1. Postfix Notation
2. Syntax Tree
3. Three-Address Code

Postfix Notation

The postfix notation for an expression E can be defined inductively as follows:

1. If E is a variable or constant, then the postfix notation for E is E itself.

2. If E is an expression of the form E_1 op E_2, then the postfix notation for E is E_1‘ E_2‘ op, where E_1‘ and E_2‘ are postfix notations for E_1 and E_2 respectively.

3. If E is an expression of the form (E_1), then the postfix notation for E is the postfix notation for E_1.

S-attributed Definition Generating Postfix Notation

Example:

$E \rightarrow E^{(1)} \ \text{op} \ E^{(2)}$	{ E . Code = $E^{(1)}$. Code || $E^{(2)}$. Code || "op" }
$E \rightarrow (E^{(1)})$	{E . Code = $E^{(1)}$. Code }
$E \rightarrow \text{id}$	{ E . Code = id.Lexval }

S-attributed Definition Generating Postfix Notation

Grammar G:

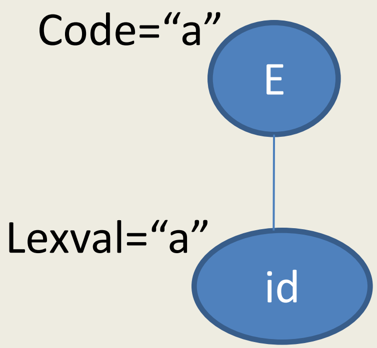
$E \rightarrow E \text{ op } E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

a + b - c



Symbol	Attr
E	“a”

S-attributed Definition Generating Postfix Notation

Grammar G:

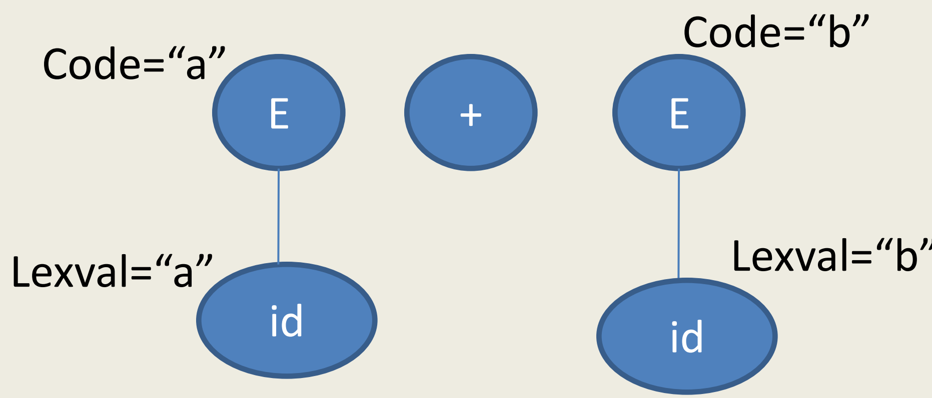
E → E op E

E → (E)

E → id

Sentence:

a + b - c



Symbol	Attr
E	"b"
+	-
E	"a"

S-attributed Definition Generating Postfix Notation

Grammar G:

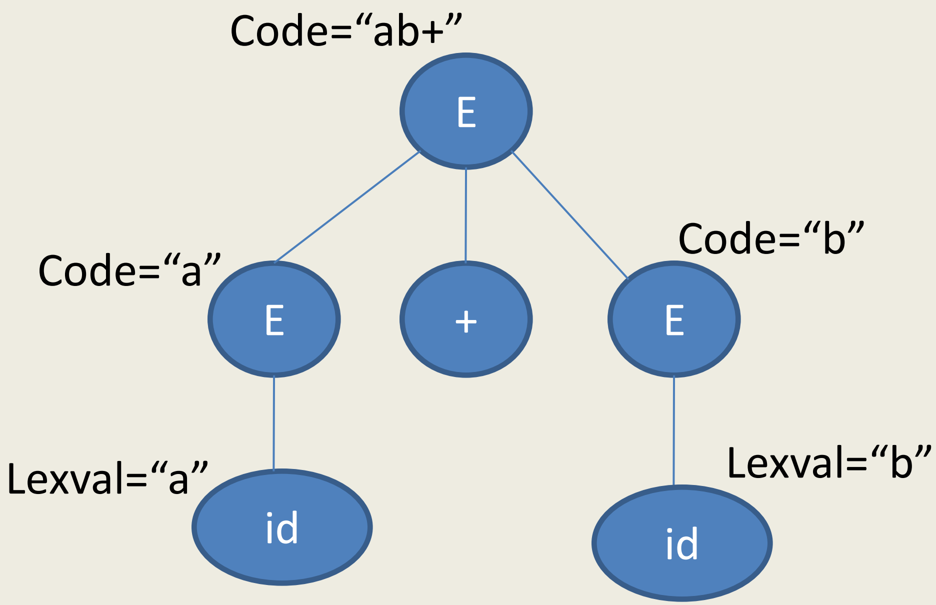
$E \rightarrow E \text{ op } E$

$E \rightarrow (E)$

$E \rightarrow \text{id}$

Sentence:

a + b - c



E	"ab+"
Symbol	Attr

S-attributed Definition Generating Postfix Notation

Grammar G:

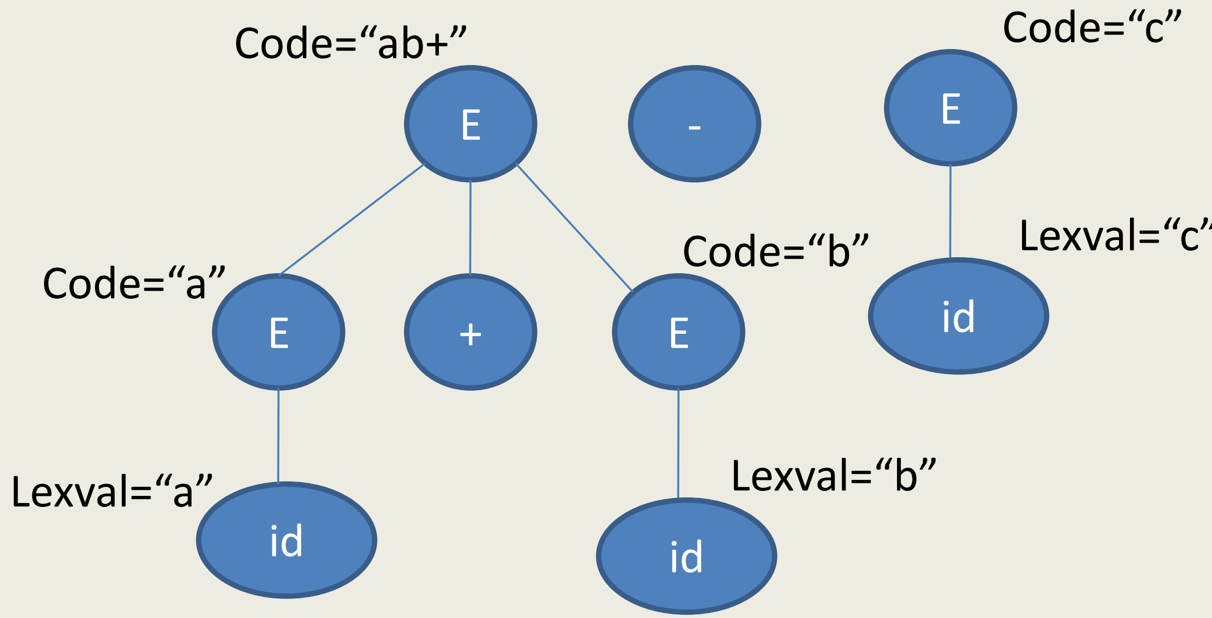
$E \rightarrow E \text{ op } E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

a + b - c



Symbol	Attr
E	"c"
-	-
E	"ab+"

S-attributed Definition Generating Postfix Notation

Grammar G:

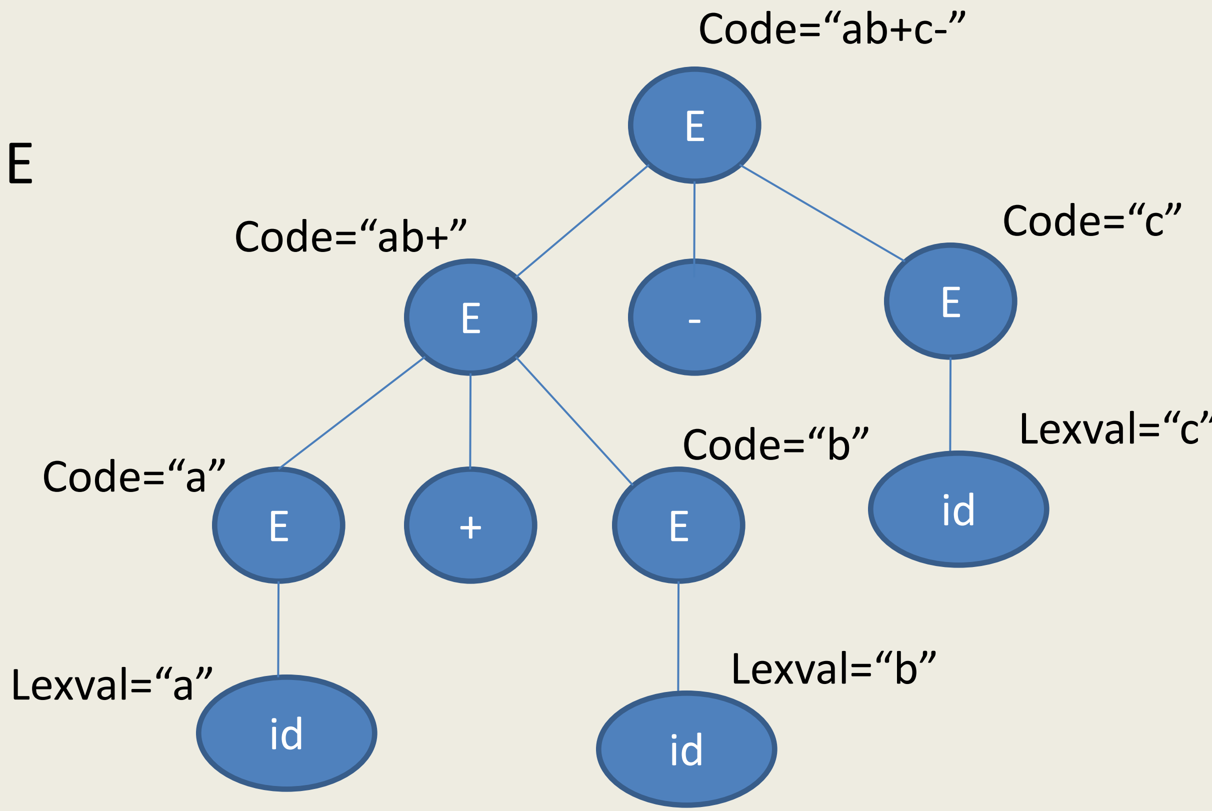
$E \rightarrow E\ op\ E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

a + b - c



Symbol	Attr
E	"ab+c-"

S-attributed Definition Generating Postfix Notation

Example:

$E \rightarrow E^{(1)}$ op $E^{(2)}$	{ if ($E^{(1)}$. Type == "int" AND $E^{(2)}$ .Type == "int") E . Type = "int" ; else Fail(); E . Code = $E^{(1)}$. Code || $E^{(2)}$. Code || "op" }
$E \rightarrow (E^{(1)})$	{E . Type = $E^{(1)}$. Type; E . Code = $E^{(1)}$. Code }
$E \rightarrow$ id	{ E . Type = id . Type ; E . Code = id . Lexval }

S-attributed Definition Generating Postfix Notation

Grammar G:

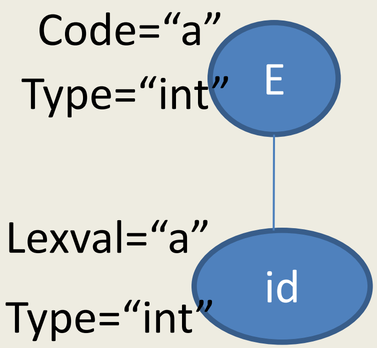
$E \rightarrow E \text{ op } E$

$E \rightarrow (E)$

$E \rightarrow \text{id}$

Sentence:

a + b - c



S-attributed Definition Generating Postfix Notation

Grammar G:

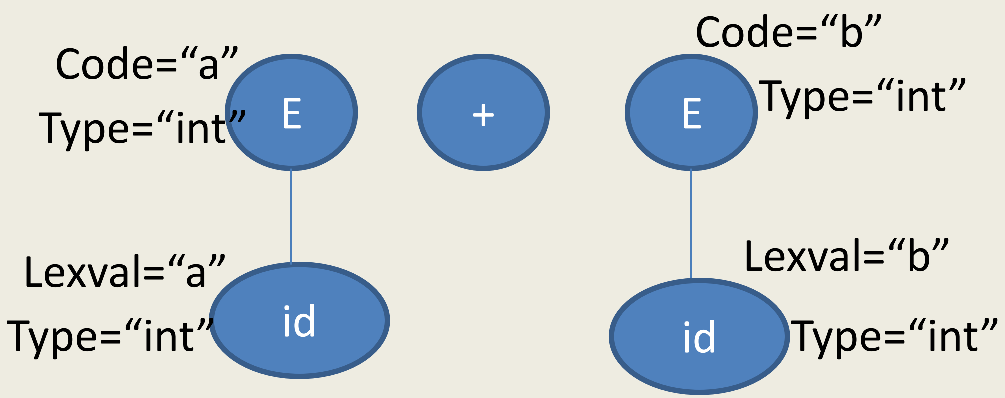
$E \rightarrow E \text{ op } E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

a + b - c



S-attributed Definition Generating Postfix Notation

Grammar G:

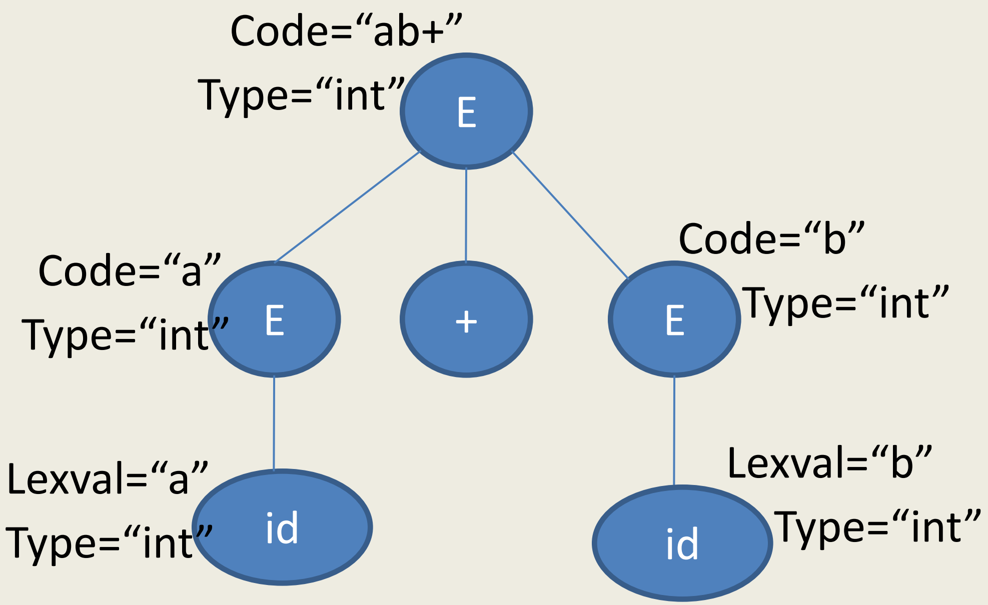
$E \rightarrow E \text{ op } E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

a + b - c



S-attributed Definition Generating Postfix Notation

Grammar G:

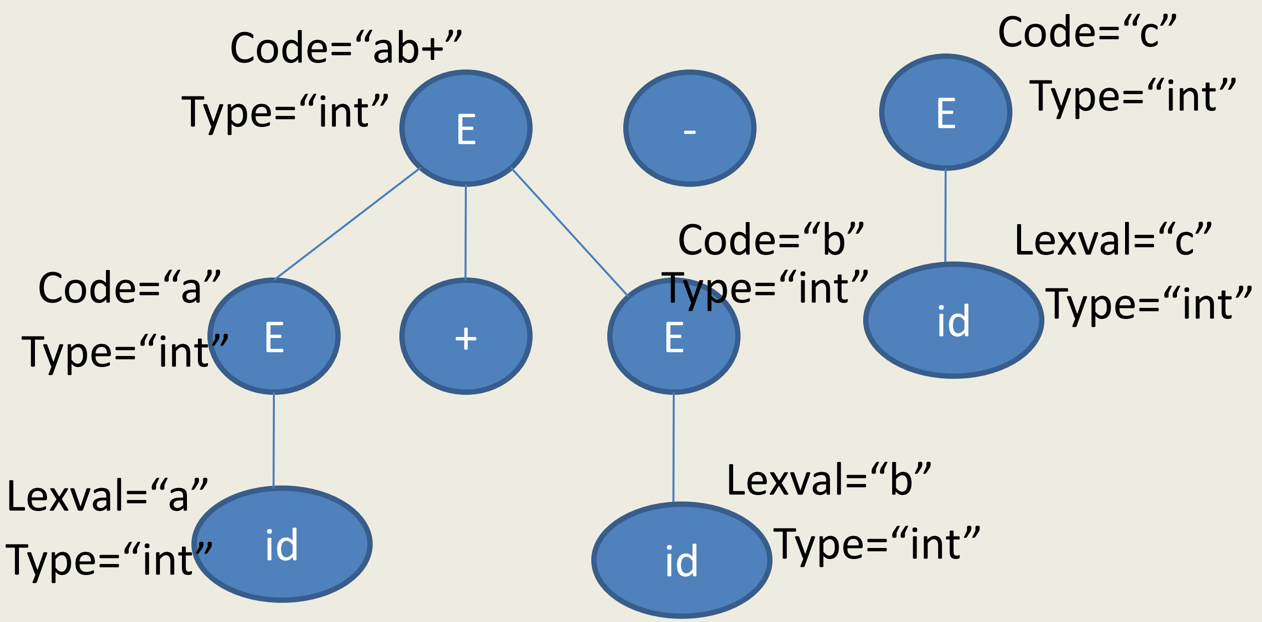
$E \rightarrow E \text{ op } E$

$E \rightarrow (E)$

$E \rightarrow \text{id}$

Sentence:

a + b - c



S-attributed Definition Generating Postfix Notation

Grammar G:

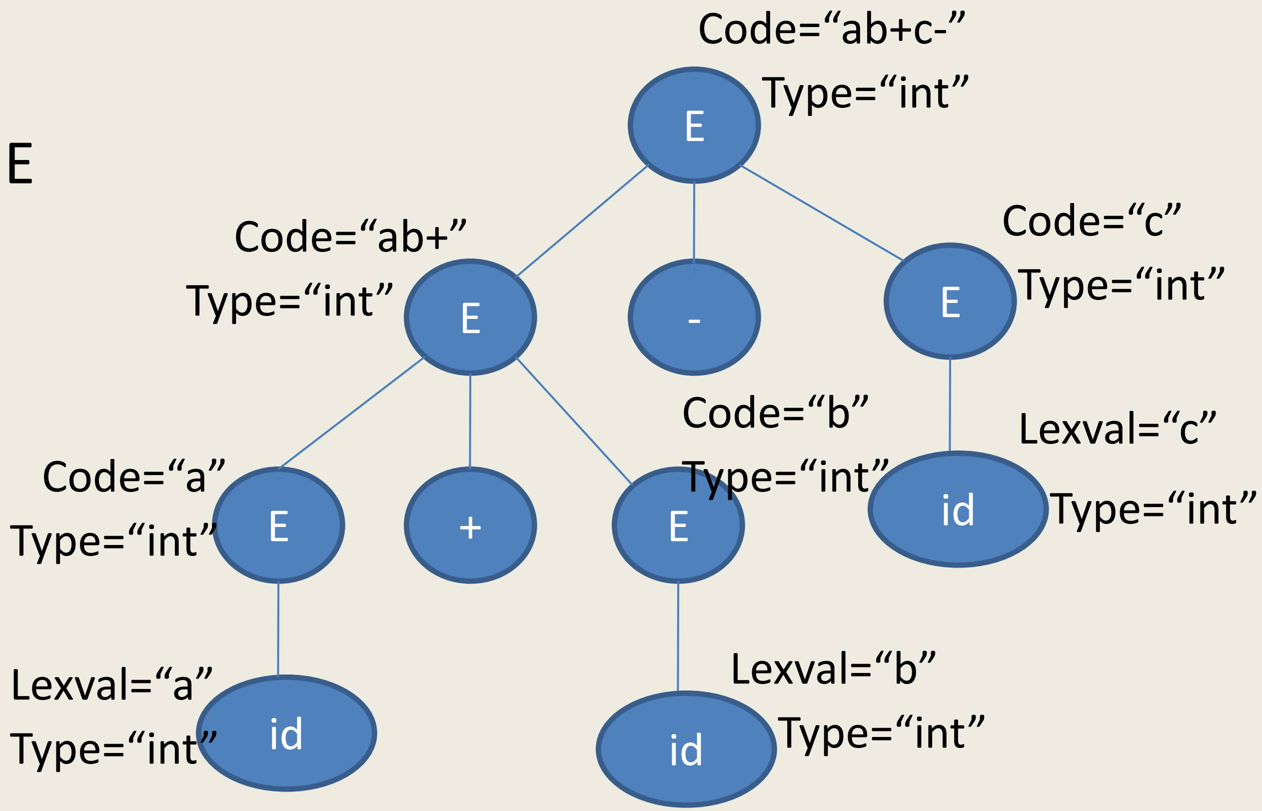
E $\rightarrow$ E op E

E $\rightarrow$ (E)

E $\rightarrow$ id

Sentence:

a + b - c



Three-Address Code

- Most widely used as Intermediate Language.
- It can be used to represent most of the language constructs conveniently.
- It is preferred as it allows the code to be rearranged in a simpler manner.

Three-Address Code

- Assignment Statements:

  A = B op C

  A = op B

  A = B

- Control Statements:

  goto L

  If A rel-op B goto L

Three-Address Code

- Indexed Assignment Statements:

  A = B[I]

  A[I] = B

- Address and Pointer Statements:

  A = addr B

  A = *B

  *A = B

Three-Address Code

- Example:

```
A = -B * ( C + D );
```

Source Code

Three-Address Code

- Example:

Source Code

```
A = -B * ( C + D );
```

Three-Address Code

```
T1 = -B
T2 = C + D
T3 = T1 * T2
A = T3
```

Three-Address Code

- Example:

```
X = A[I][J] ;
```

Source Code

Three-Address Code

- Example:

Source Code

```
X = A[I][J] ;
```

Three-Address Code

```
T1 = A[I]
X = T1[J]
```

S-attributed Definition Generating Three-Address Code

Example:

$S \rightarrow id = E$	{S.Code = E.Code || Gen(id.Name, "=", E.Name) }
$E \rightarrow E^{(1)}\ op\ E^{(2)}$	{ E.Name = NewTemp(); E.Code = $E^{(1)}$. Code || $E^{(2)}$. Code || Gen(E.Name, "=", $E^{(1)}$. Name, "op", $E^{(2)}$. Name) }
$E \rightarrow (E^{(1)})$	{E.Name = $E^{(1)}$. Name ; E.Code = $E^{(1)}$. Code }
$E \rightarrow id$	{ E.Name = id.Lexval ; E.Code = "" }

S-attributed Definition Generating Three-Address Code

Grammar G:

$S \rightarrow id = E$

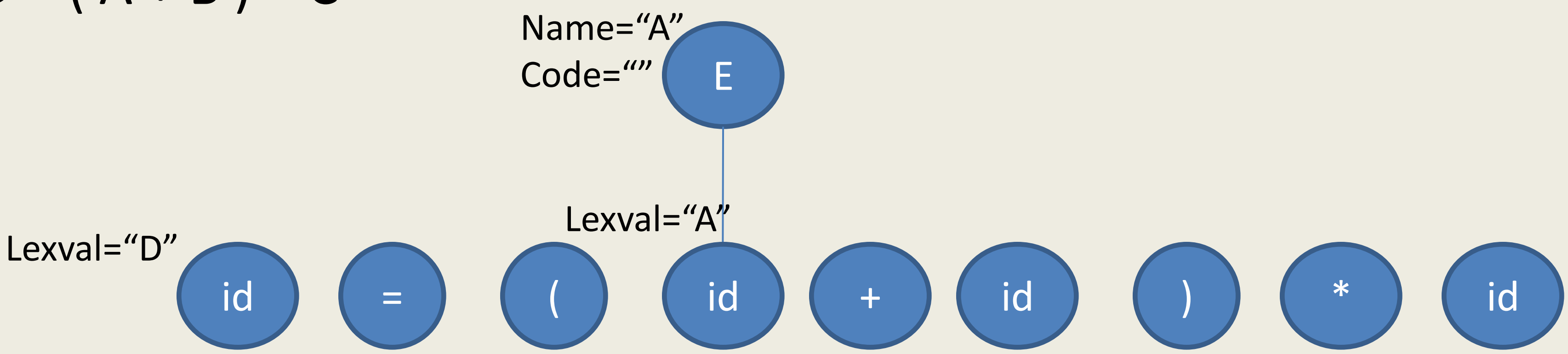
$E \rightarrow E \ op \ E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

D = (A + B) * C



S-attributed Definition Generating Three-Address Code

Grammar G:

$S \rightarrow id = E$

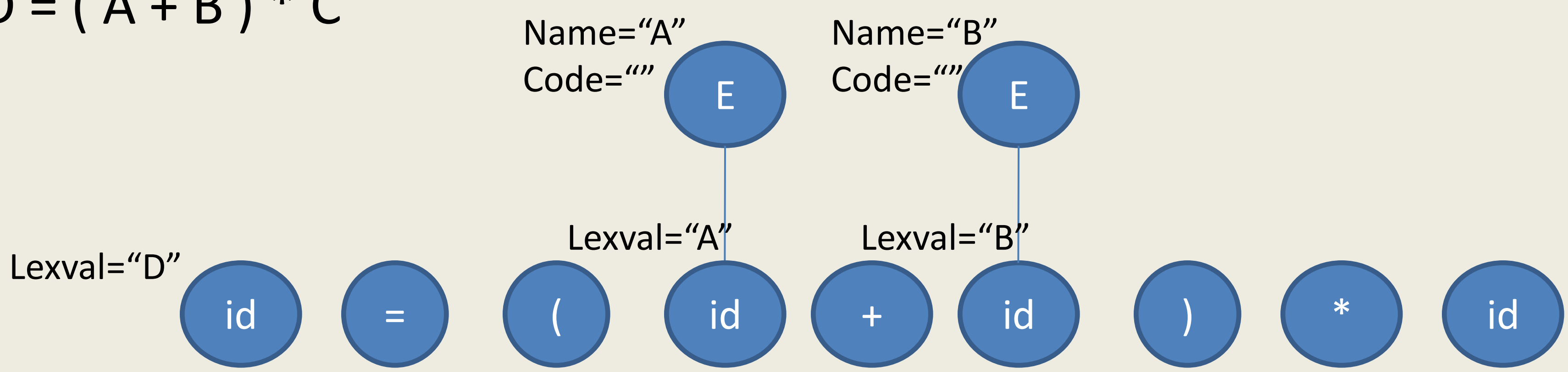
$E \rightarrow E \; op \; E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

D = (A + B) * C



S-attributed Definition Generating Three-Address Code

Grammar G:

$S \rightarrow id = E$

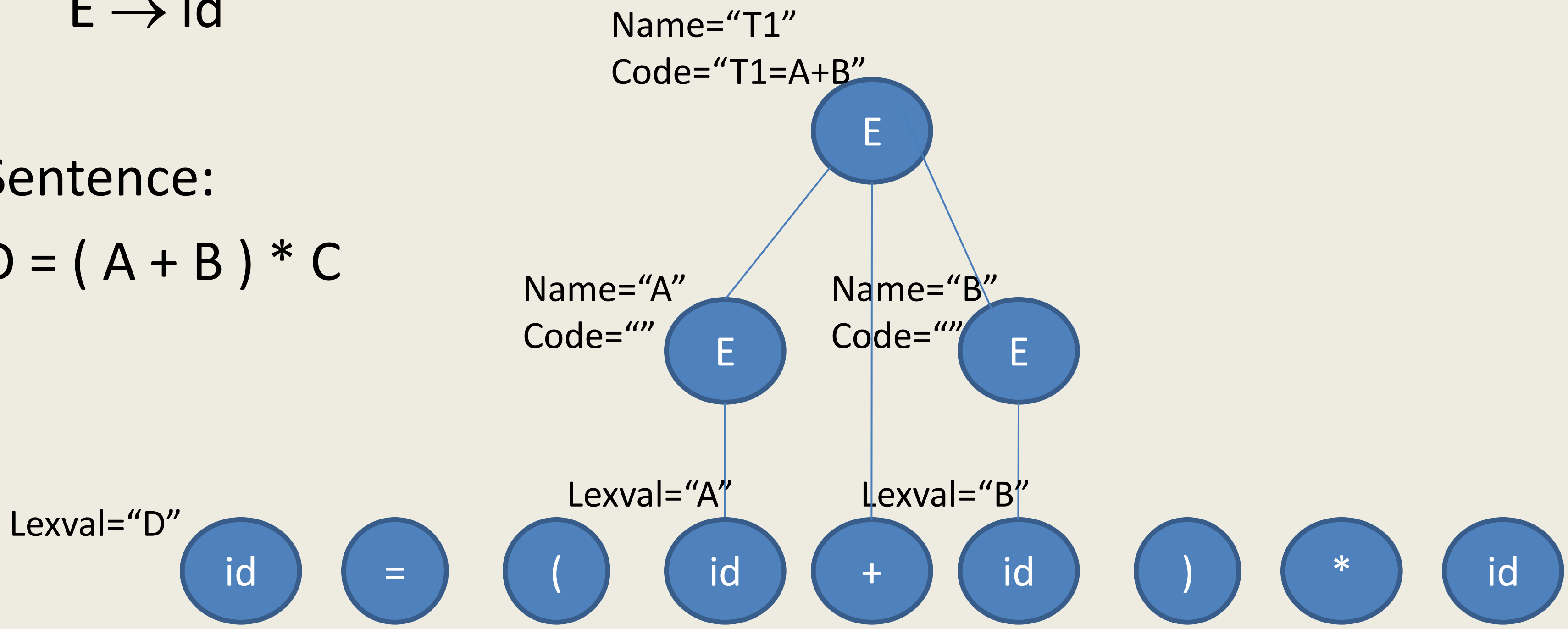
$E \rightarrow E \ op \ E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

D = (A + B) * C



S-attributed Definition Generating Three-Address Code

Grammar G:

$S \rightarrow id = E$

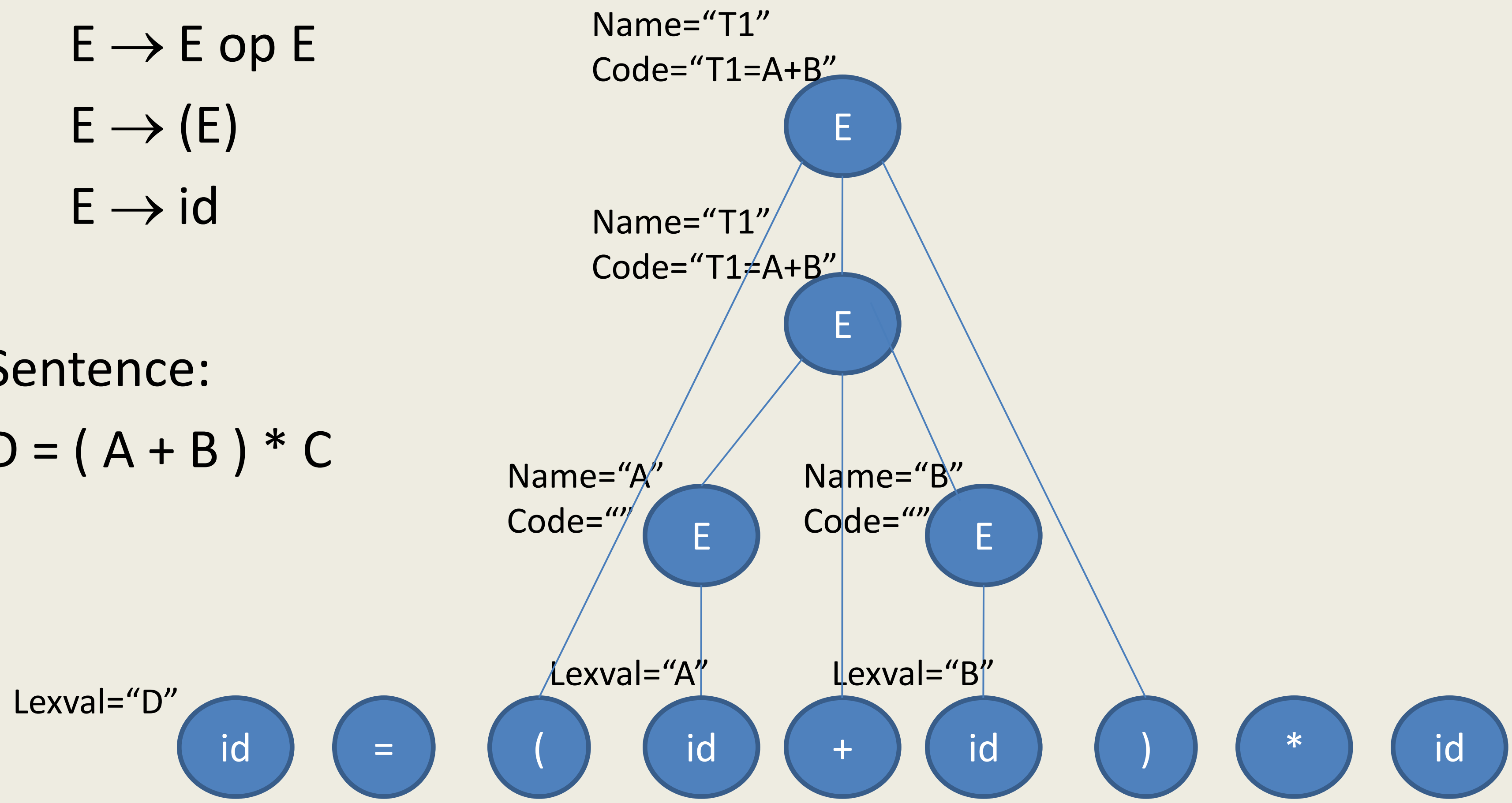
$E \rightarrow E \text{ op } E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

D = (A + B) * C



S-attributed Definition Generating Three-Address Code

Grammar G:

$S \rightarrow id = E$

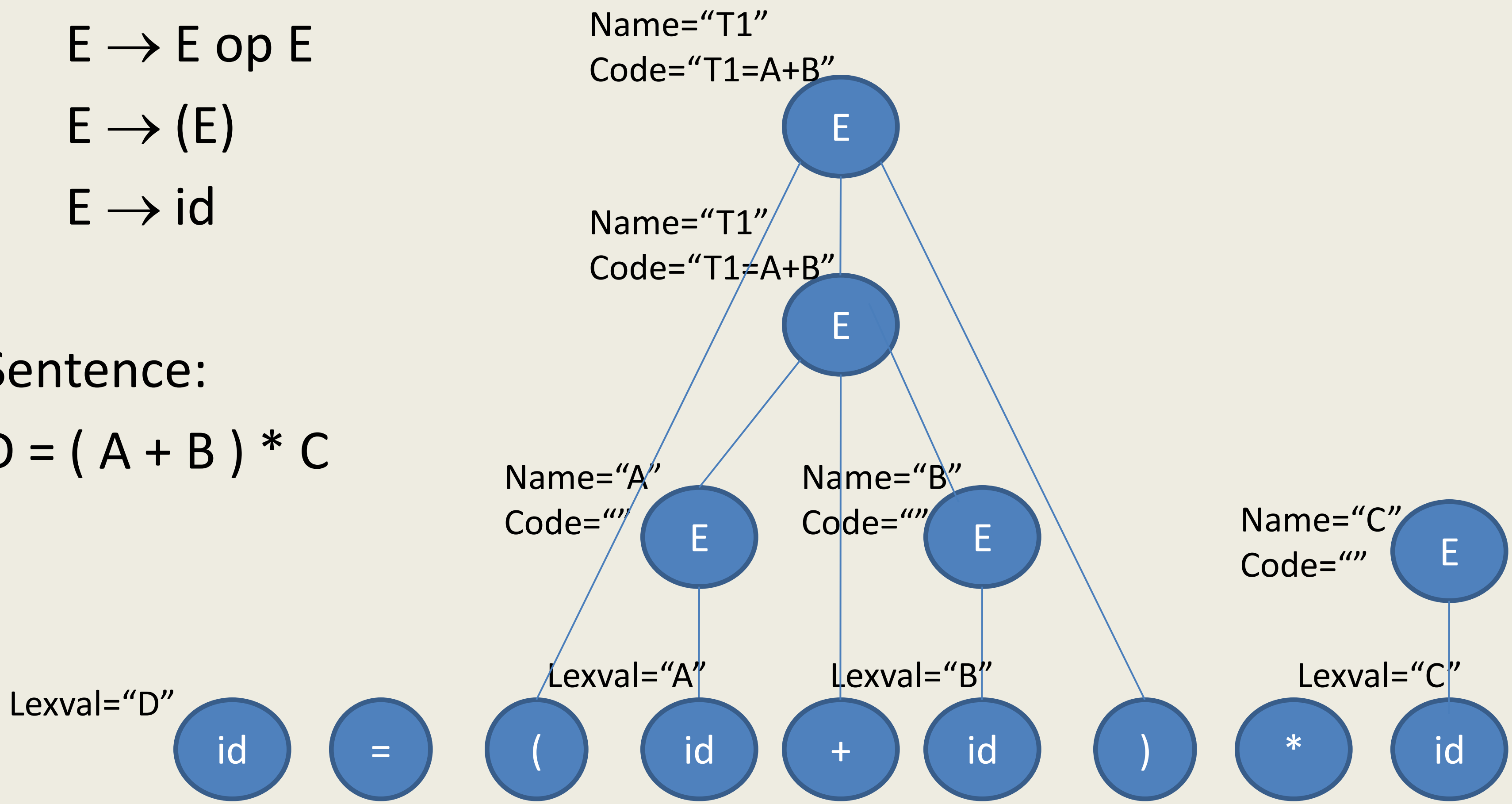
$E \rightarrow E\ op\ E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

D = (A + B) * C



S-attributed Definition Generating Three-Address Code

Grammar G:

S → id = E

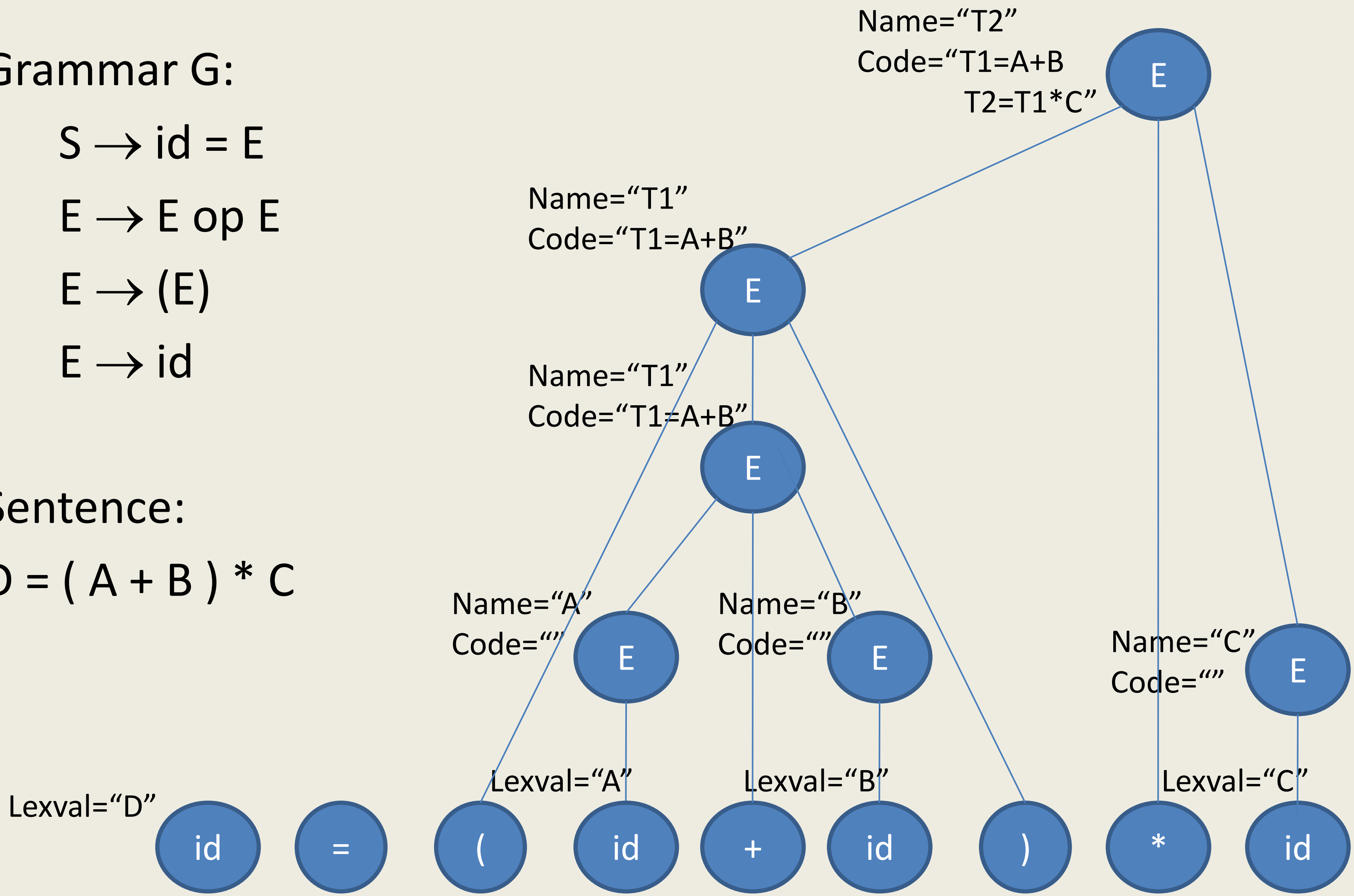
E → E op E

E → (E)

E → id

Sentence:

D = (A + B) * C



S-attributed Definition Generating Three-Address Code

Grammar G:

$S \rightarrow id = E$

$E \rightarrow E \; op \; E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentence:

D = (A + B) * C