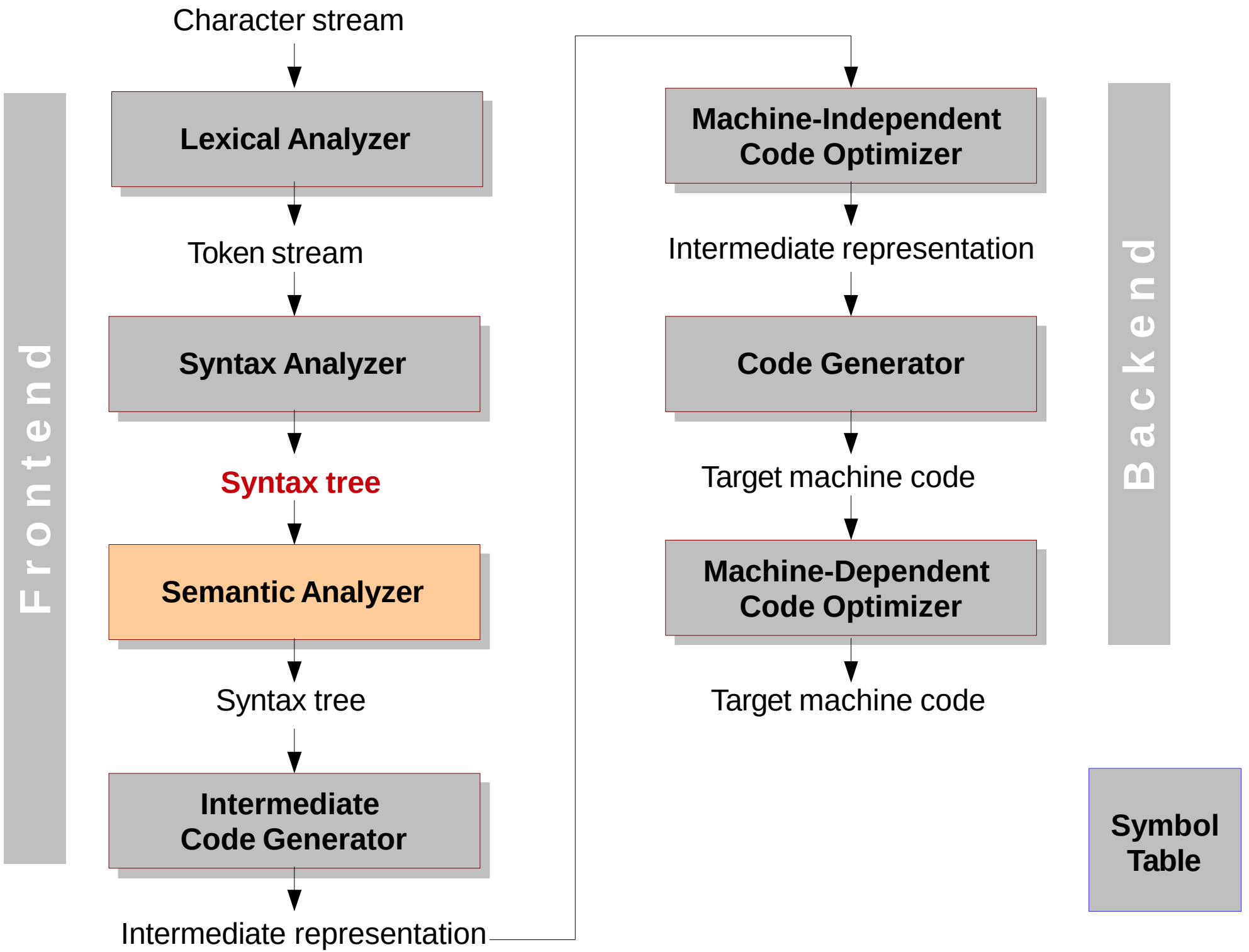
Syntax Directed Translation


Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Machine-Independent Code Optimizer
Intermediate representation
Code Generator
Target machine code
Machine-Dependent Code Optimizer
Target machine code
Frontend
Backend
Symbol Table

Role of SDT

- To associate actions with productions
- To associate attributes with non-terminals
- To create implicit or explicit syntax tree To
- perform semantic analysis

- … essentially, to add life to the skeleton.

Example

$E \rightarrow E + T$

```
$$.code = "";
strcat($$.code, $1.code);
strcat($$.code, $3.code);
strcat($$.code, "+");
```

SDD

Attributes

$E \rightarrow E + T$

```
{ printf("+"); }
```

SDT

SDTs may be viewed as implementations of SDDs and are important from efficiency perspective.

Productions **Actions**

Syntax Directed Definition

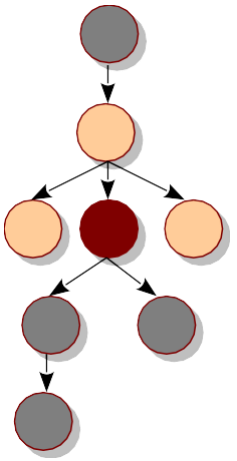
- An SDD is a CFG with attributes and rules.
  - Attributes are associated with grammar symbols.
  - Rules are associated with productions.
- An SDD specifies the semantics of productions.
  - It does not enforce a specific way of achieving the semantics.

Syntax Directed Translation

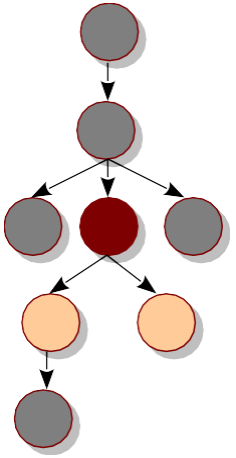
- An SDT is done by attaching rules or program fragments to productions.
- The order induced by the syntax analysis produces a translation of the input program.

Attributes

- Inherited
  - In terms of the attributes of the node, its parent and siblings.
  - e.g., int x, y, z; or
    Ishu's nested scoping



- Synthesized
  - In terms of the attributes of the node and its children.
  - e.g., a + b * c or
    most of the constructs from your assignments

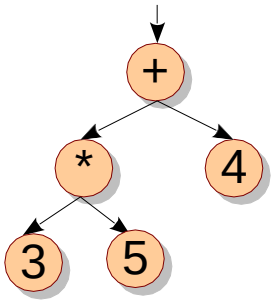


SDD for Calculator

Input string

3 * 5 + 4 $

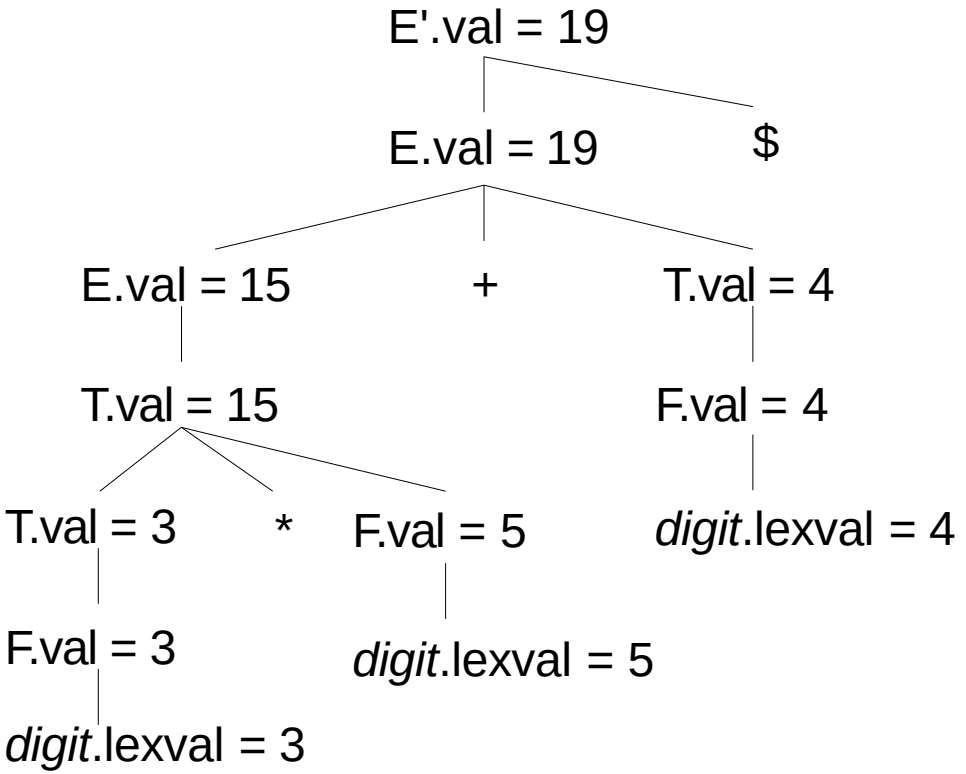
Parse Tree



SDD

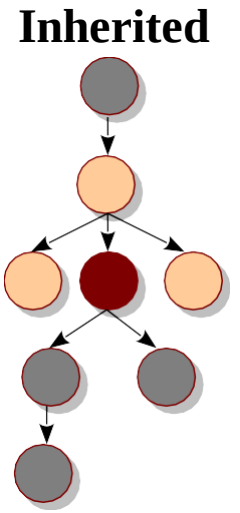
Sr. No.	Production	Semantic Rules
1	E' → E $	E'.val = E.val
2	E → E_1 + T	E.val = E_1.val + T.val
3	E → T	...
4	T → T_1 * F	...
5	T → F	...
6	F → (E)	...
7	F → *digit*	F.val = *digit*.lexval

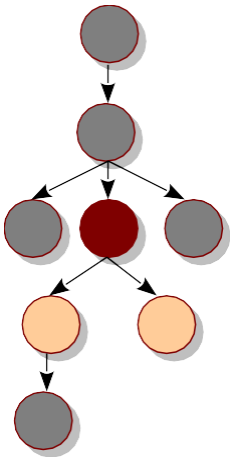
Annotated Parse Tree



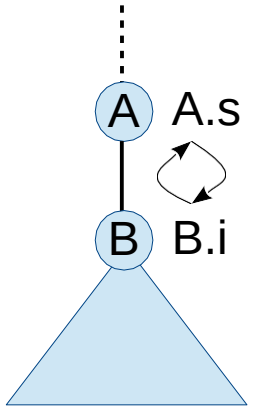
Order of Evaluation

- If there are only synthesized attributes in the SDD, there exists an evaluation order.
- Any bottom-up order would do; for instance, post-order.
- Helpful for LR parsing.
- How about when the attributes are both synthesized as well as inherited?
- How about when the attributes are only inherited?

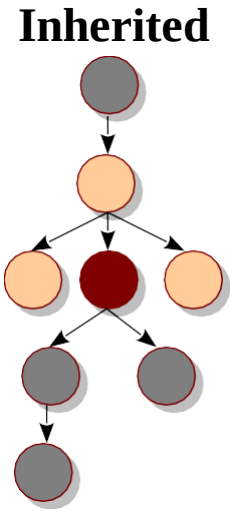




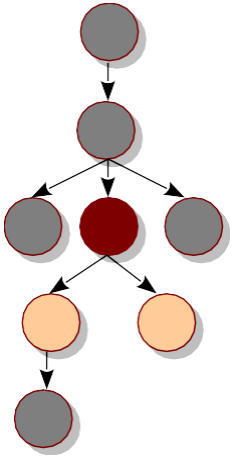
Order of Evaluation



Production	Semantic Rule
$A \rightarrow B$	A.s = B.i; B.i = A.s + 1;



- This SDD uses a combination of synthesized and inherited attributes.
- A.s (head) is defined in terms of B.i (body non-terminal). Hence, it is synthesized.
- B.i (body non-terminal) is defined in terms of A.s (head). Hence, it is inherited.
- There exists a *circular dependency* between their evaluations.
- In practice, subclasses of SDDs required for our purpose do have an order.

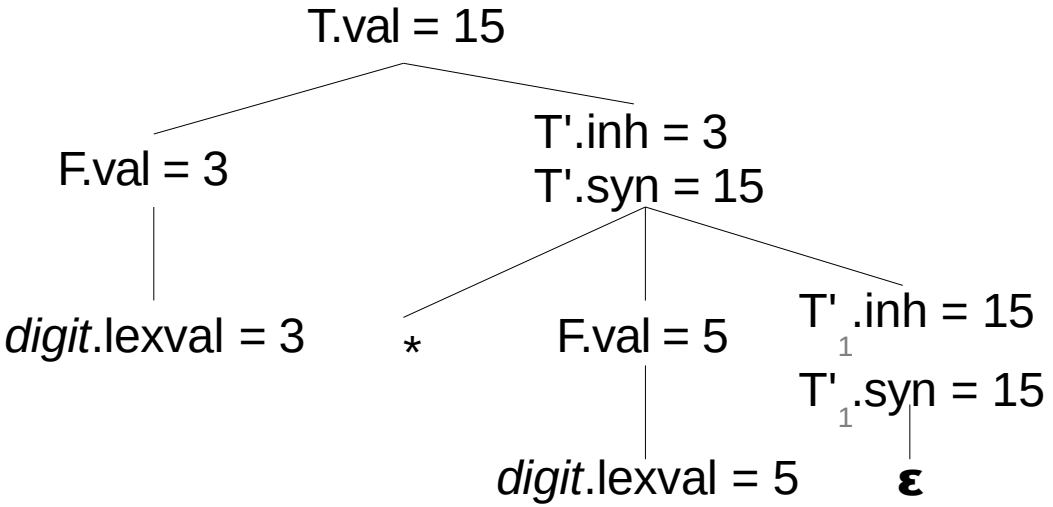


Classwork

- Write semantic rules for the following grammar.
  - It computes terms like 3 * 5 and 3 * 5 * 7.
  - Is * left or right-associative? Can you make it left?
- Now write the annotated parse tree for 3 * 5.

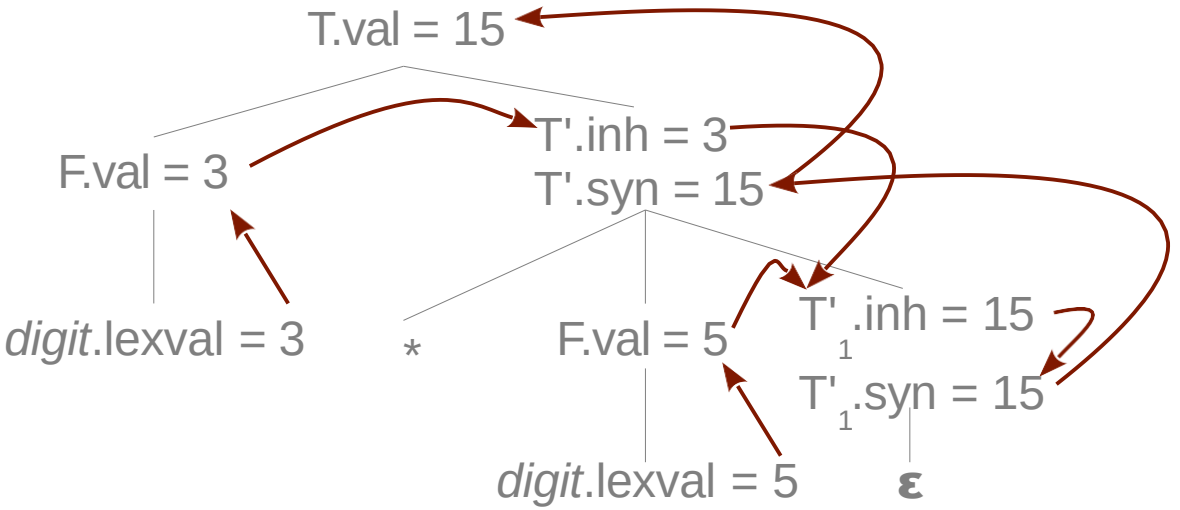
Sr. No.	Production	Semantic Rules
1	$T \rightarrow F\ T'$	T'.inh = F.val T.val = T'.syn
2	$T' \rightarrow *\ F\ T'_1$	T'_1.inh = T'.inh * F.val T'.syn = T'_1.syn
3	$T' \rightarrow \boldsymbol{\varepsilon}$	T'.syn = T'.inh
4	$F \rightarrow$ *digit*	F.val = *digit*.lexval

Classwork



Sr. No.	Production	Semantic Rules
1	T → F T'	T'.inh = F.val T.val = T'.syn
2	T' → * F T'1	T'1.inh = T'.inh * F.val T'.syn = T'1.syn
3	T' → **ε**	T'.syn = T'.inh
4	F → *digit*	F.val = *digit*.lexval

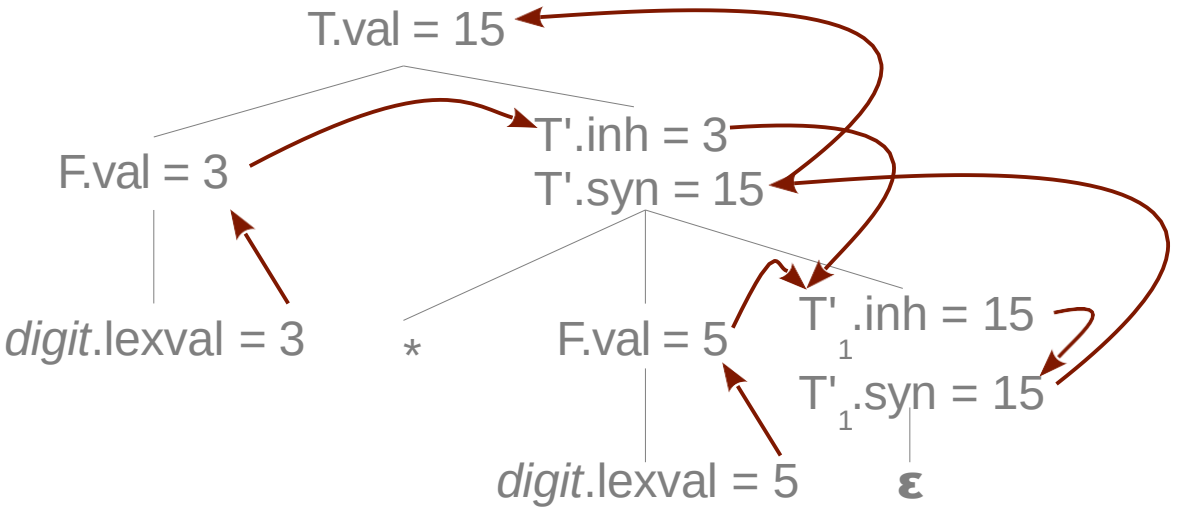
Classwork



What is the order in which rules are evaluated?

Sr. No.	Production	Semantic Rules
1	$T \rightarrow F\ T'$	T'.inh = F.val T.val = T'.syn
2	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh * F.val$ $T'.syn = T'_1.syn$
3	$T' \rightarrow \varepsilon$	T'.syn = T'.inh
4	$F \rightarrow digit$	F.val = *digit*.lexval

Dependency Graph

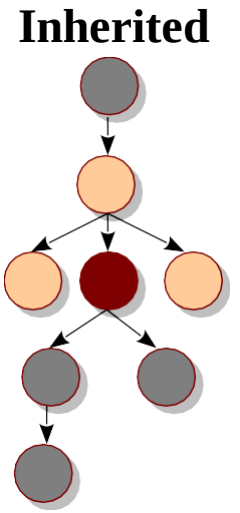


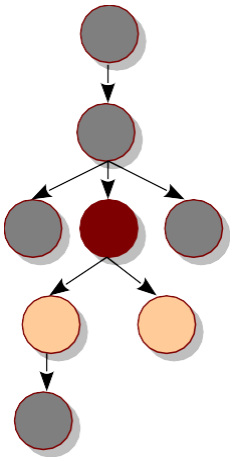
- A dependency graph depicts the flow of information amongst attributes.
- An edge *attr1* → *attr2* means that the value of *attr1* is needed to compute *attr2*.
- Thus, allowable evaluation orders are those sequences of rules N1, N2, …, Nk such that if Ni → Nj, then i < j.
  - What are such allowable orders?
  - Topological sort
  - What about cycles?

Order of Evaluation

- If there are only synthesized attributes in the SDD, there exists an evaluation order.
- Any bottom-up order would do; for instance, post-order.
- Helpful for LR parsing.
- How about when the attributes are both synthesized as well as inherited?
- How about when the attributes are only inherited?

S-attributed





SDD for Calculator

Input string

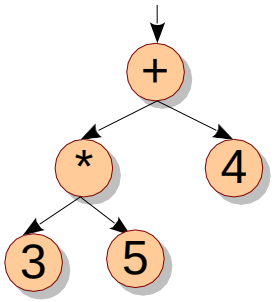
3 * 5 + 4 $

S-attributed

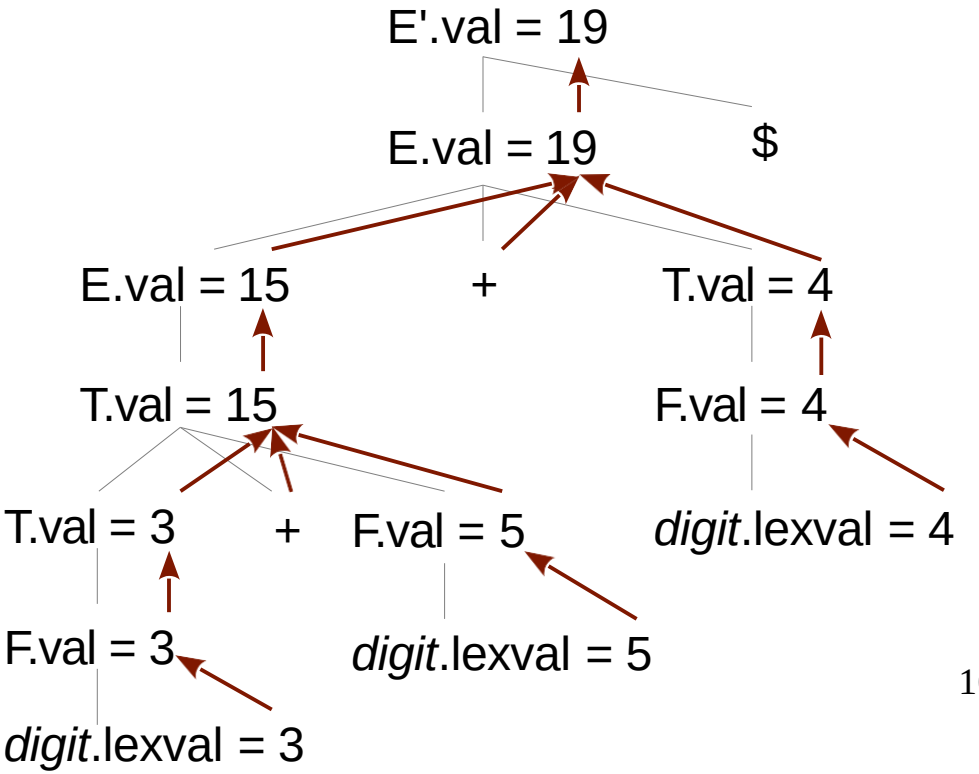
SDD

Sr. No.	Production	Semantic Rules
1	E' → E $	E'.val = E.val
2	E → E_1 + T	E.val = E_1.val + T.val
3	E → T	...
4	T → T_1 * F	...
5	T → F	...
6	F → (E)	...
7	F → *digit*	F.val = *digit*.lexval

Parse Tree



Annotated Parse Tree



S-attributed SDD

- Every attribute is synthesized.
- A topological evaluation order is well-defined.
- Any bottom-up order of the parse tree nodes.
- In practice, preorder is used.

```
preorder(N) {
    for (each child C of N, from the left) preorder(C)
    evaluate attributes of N
}
```

Can we allow more orderings?

Issues with S-attributed SDD

- It is too strict!
- There exist reasonable non-cyclic orders that it disallows.
  - If a non-terminal uses attributes of its parent only (no sibling attributes)
  - If a non-terminal uses attributes of its left-siblings only (and not of right siblings).
- The rules may use information “from above” and “from left”.

L-attributed

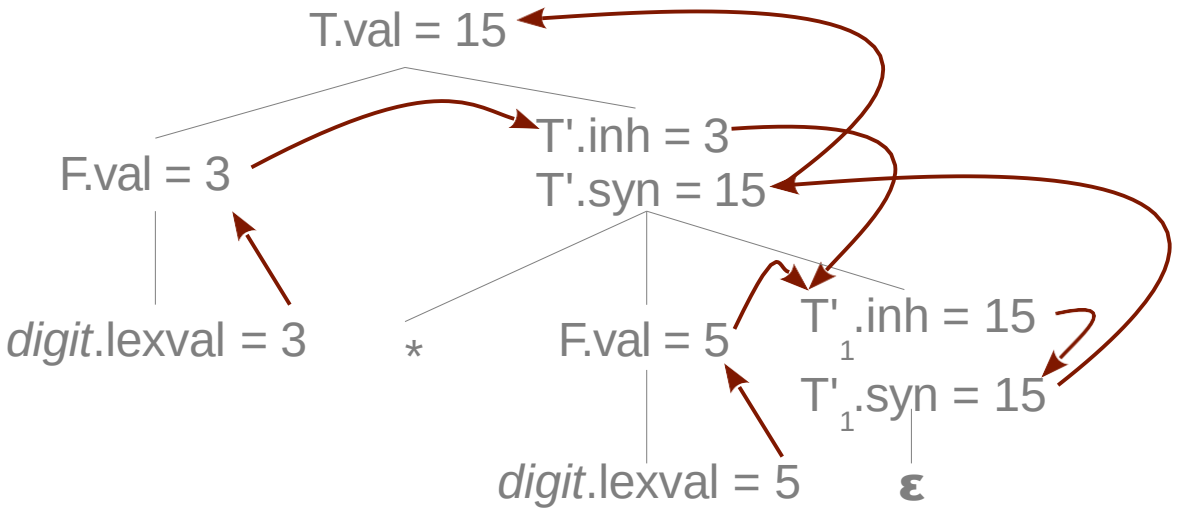
L-attributed SDD

- Each attribute must be either
  - synthesized, or
  - inherited, but with restriction. For production A → X1 X2 … Xn with inherited attributed Xi.a computed by an action; then the rule may use only
    - inherited attributes of A.
    - either inherited or synthesized attributes of X1, X2, …, Xi-1.
    - inherited or synthesized attributes of Xi with no cyclic dependence.
- L is for left-to-right.

Have you seen any such SDD?

Example of L-attributed SDD

Sr. No.	Production	Semantic Rules
1	T → F T'	T'.inh = F.val T.val = T'.syn
2	$T' \rightarrow * F T'_1$	$T'_1.inh = T'.inh * F.val$ $T'.syn = T'_1.syn$
3	T' → **ε**	T'.syn = T'.inh
4	F → *digit*	F.val = *digit*.lexval



Example of non-L-attributed SDD

Production	Semantic rule
$A \rightarrow B\,C$	A.s = B.b; B.i = C.c + A.s

- First rule uses synthesized attributes.
- Second rule has inherited attributes.
- However, B's attribute is dependent on C's attribute, which is on the right.
- Hence, it is not L-attributed SDD.

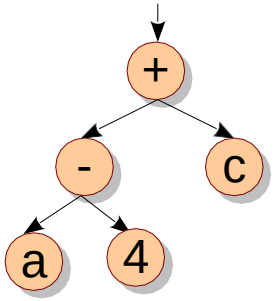
$S \rightarrow L\,.\,L \mid L$
$L \rightarrow L\,B \mid B$
$B \rightarrow 0 \mid 1$

Classwork:

- What does this grammar generate?
- Design L-attributed SDD to compute S.val, the decimal value of an input string.
- For instance, 101.101 should output 5.625.
- Idea: Use an inherited attribute L.side that tells which side (left or right) of the decimal point a bit is on.

SDT Applications

- Creating an explicit syntax tree.
  - e.g., a – 4 + c

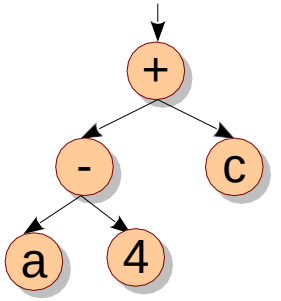


  - p1 = new Leaf(id_a);
  - p2 = new Leaf(num_4);
  - p3 = new Op(p1, '-', p2);
  - p4 = new Leaf(id_c);
  - p5 = new Op(p3, '+', p4);

Production	Semantic Rules
E → E + T	$$.node = new Op($1.node, '+', $3.node)
E → E - T	$$.node = new Op($1.node, '-', $3.node)
E → T	$$.node = $1.node
T → (E)	$$.node = $2.node
T → *id*	$$.node = new Leaf($1)
T → *num*	$$.node = new Leaf($1)

SDT Applications

- Creating an explicit syntax tree.
  - e.g., a – 4 + c

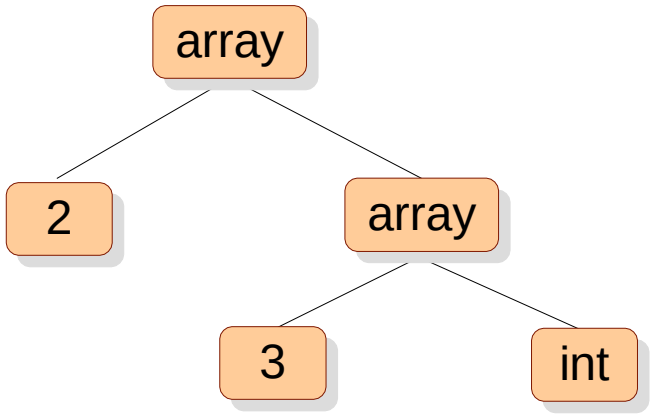


- **Classwork**:
  - Generate syntax tree using the following grammar.

Production	Semantic Rules
E → T E'	$$.node = $2.syn $2.inh = $1.node
E' → + T E'$_1$	$3.inh = new Op($$.inh, '+', $2.node) $$.syn = $3.syn
E' → - T E'$_1$	$3.inh = new Op($$.inh, '-', $2.node) $$.syn = $3.syn
E' → **ε**	$$.syn = $$.inh
T → (E)	$$.node = $2.node
T → *id*	$$.node = new Leaf($1)
T → *num*	$$.node = new Leaf($1)

SDT Applications

- Finding type expressions
  - int a[2][3] is array of 2 arrays of 3 integers.
  - in functional style: *array(2, array(3, int))*



Production	Semantic Rules
T → B id C	T.t = C.t C.i = B.t
B → *int*	B.t = *int*
B → *float*	B.t = *float*
C → [num] C_1	C.t = array(num, C_1.t) C_1.i = C.i
C → **ε**	C.t = C.i

Classwork: Write productions and semantic rules for creating type expressions from array declarations.

SDD for Calculator

Sr. No.	Production	Semantic Rules
1	$E' \rightarrow E\ \$$	E'.val = E.val
2	$E \rightarrow E_1 + T$	E.val = E_1.val + T.val
3	$E \rightarrow T$	...
4	$T \rightarrow T_1 * F$	...
5	$T \rightarrow F$	...
6	$F \rightarrow (E)$	...
7	$F \rightarrow$ *digit*	F.val = *digit*.lexval

SDT for Calculator

Sr. No.	Production	Semantic Rules
1	E' → E $	**print(E.val)**
2	E → E_1 + T	E.val = E_1.val + T.val
3	E → T	...
4	T → T_1 * F	...
5	T → F	...
6	F → (E)	...
7	F → *digit*	F.val = *digit*.lexval

SDT for Calculator

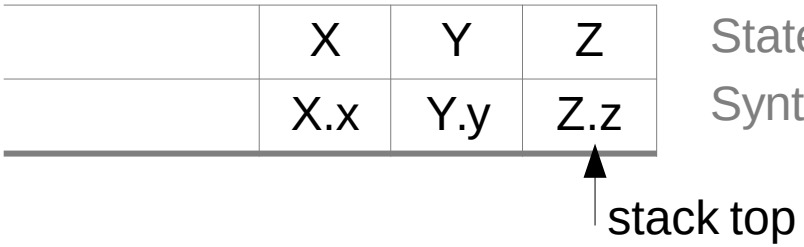
Postfix SDT

E' → E $	**{ print(E.val); }**
E → E_1 +T	{ E.val = E_1.val + T.val; }
E → T	...
T → T_1 *F	...
T → F	...
F → (E)	...
F → *digit*	{ F.val = *digit*.lexval; }

- SDTs with all the actions at the right ends of the production bodies are called *postfix SDTs*.
- Only synthesized attributes are useful here.
- Can be implemented during LR parsing by executing actions when reductions occur.
- The attribute values can be put on a stack and can be retrieved.

Parsing Stack

$A \rightarrow X\ Y\ Z$



State / grammar symbol

Synthesized attribute

Compare with $1, $2, … in Yacc.

Production	**Actions**
E' → E $	{ print(stack[top – 1].val); --top; }
E → E_1 +T	{ stack[top – 2].val += stack[top].val; top -= 2; }
E → T	{ stack[top].val = stack[top].val; }
T → T_1 *F	{ stack[top – 2].val *= stack[top].val; top -= 2; }
T → F	{ stack[top].val = stack[top].val; }
F → (E)	{ stack[top – 2].val = stack[top – 1].val; top -= 2; }
F → *digit*	{ stack[top].val = stack[top].val; }

Actions within Productions

- Actions may be placed at any position within production body. Considered as empty non-terminals called *markers*.
- For production B → X {action} Y, action is performed
  - as soon as X appears on top of the parsing stack in bottom-up parsing.
  - just before expanding Y in top-down parsing if Y is a non-terminal.
  - just before we check for Y on the input in top-down parsing if Y is a terminal.
- SDTs that can be implemented during parsing are
  - Postfix SDTs (S-attributed definitions)
  - SDTs implementing L-attributed definitions

Classwork: Write SDT for infix-to-prefix translation.

Infix-to-Prefix

- What is the issue with this SDT?
- The SDT has shift-reduce conflicts.
- Recall that each marker is an empty non-terminal. Thus, the parser doesn't know whether to shift or shift or reduce on seeing a digit.

E' → E $

E → **{ print '+'; }** E1 + T

E → T

T → **{ print '*'; }** T1 * F

T → F

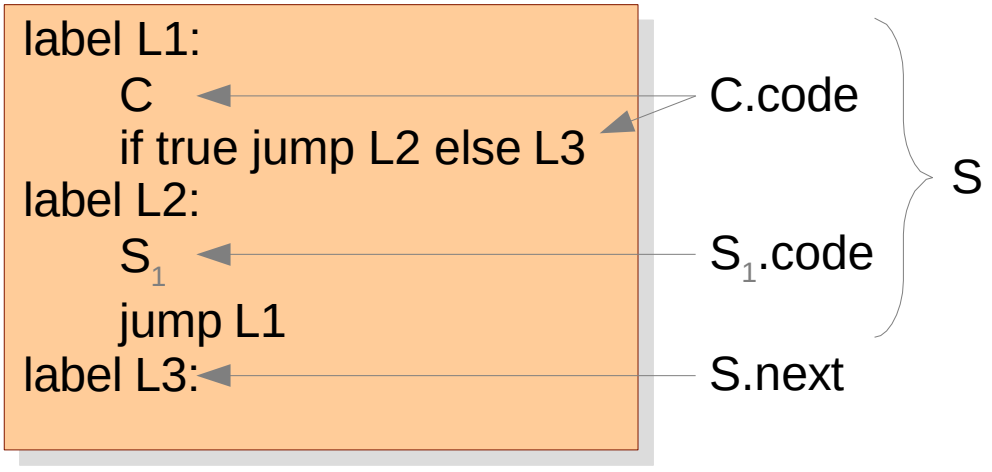
F → (E)

F → digit **{ print digit.lexval; }**

Classwork: Write SDT for infix-to-prefix translation.

Code Generation for *while*

- We want to generate code for while-construct
  - $S \rightarrow$ while (C) S_1
- We assume that code for S_1 and C are available.
- We also (for now) generate a single code string.
- **Classwork**: What all do we require to generate this code?
  - This would give us an idea of what attributes we need and their types.



Code Generation for *while*

- Assume we have the following mechanism.
  - newLabel() returns a new label name.

    You may have used a similar one for temporaries.
  - Each statement has an attribute next, that points to the next statement to be executed.
  - Each conditional has two branches true and false.
  - Each non-terminal has an attribute code.

SDD for *while*

$S \rightarrow$ while (C)
S_1

L1	= newLabel();
L2	= newLabel();
S_1.next	= L1;
C.false	= S.next;
C.true	= L2;
S.code	= “label” + L1 + C.code + ”label” + L2 + S_1.code;

SDT

$S \rightarrow$	while (	{ L1 = newLabel(); L2 = newLabel(); C. false = S.next; C.true = L2; }
	C)	{ S_1.next = L2; }
	S_1	{ S.code = “label” + L1 + C.code + “label” + L2 + S_1.code; }

What is the type of this SDD?

SDD for *while*

$S \rightarrow$ while (C)
S_1

L1	= newLabel();
L2	= newLabel();
S_1.next	= L1;
C.false	= S.next;
C.true	= L2;
S.code	= “label” + L1 + C.code + ”label” + L2 + S_1.code;

SDT

$S \rightarrow$ while ({ L1 = newLabel(); L2 = newLabel(); C. false = S.next; C.true = L2; print(“label”, L1); }
C) { S_1.next = L2; print(“label”, L2); }
S_1

On-the-fly code generation

What is the type of this SDD?

Homework

- Exercises 5.5.5 from ALSU book.