Syntax-Directed Translation

Md Shad Akhtar
Assistant Professor
IIIT Dharwad

Compiler Design: Journey so far!!

- Lexical Analysis: Scanning input and generating tokens -- Done
- Syntax Analysis: Validating the input -- Done
- Semantic Analysis: Validating the meaning
  - Issues deeper than the syntax
  - E.g.

```
int func (int x, int y);
int main ()
{
    int list[5], i, j;
    char *str;
    j = 10 + 'b';
    str = 8;
    m = func ("aa", j, list[12]);
    return 0;
}
```

What are the issues with this code snippet?

Beyond Syntax Analysis

- An identifier named x has been recognized
  - Is x a *scalar*, *array* or *function*?
  - What is the *size* of x?
  - If x is a function, *how many* and what *type of arguments* does it take?
  - Is x *declared before being used*?
  - Is the expression x+y *type-consistent*?

- Semantic Analysis is the phase where we collect information about the *types of expressions* and check for *type related errors*

- The more information we can collect at compile time, the less overhead we have at run time

Syntax-Directed Translation (SDT)

- We attach *program fragments* or *rules* to the productions of a grammar that facilitates the semantic analysis.
- These rules get executed when the associated productions are used in the derivation during syntax analysis
  - Therefore, the name syntax-directed translation

- Attributes
  - Any quantity associated with the symbols, e.g., data type, value, count, location, etc.

$E \rightarrow E_1 + T$	{E.val = E_1.val + T.val;}
Production	Program fragment

  - *val* is an attribute of the symbols *E* and *T*

What program fragments can do?

- May perform type checking
- May generate intermediate codes
- May put information into the symbol table
- May issue error messages
- May perform some other activities
- In fact, *they may perform almost any activities!*

Notations for translation

- **Syntax-Directed Definition (SDD)**
  - Production is associated with a set of *semantic rules*, but do not have any prior information about when they will be evaluated
    - Hide many implementation details such as order of evaluation of semantic actions
  - Useful for specification

- **Syntax-Directed Translation schemes**
  - Translation schemes give a little bit information about implementation details
    - Indicate the order of evaluation of *semantic actions* associated with a production rule
  - Useful for implementation

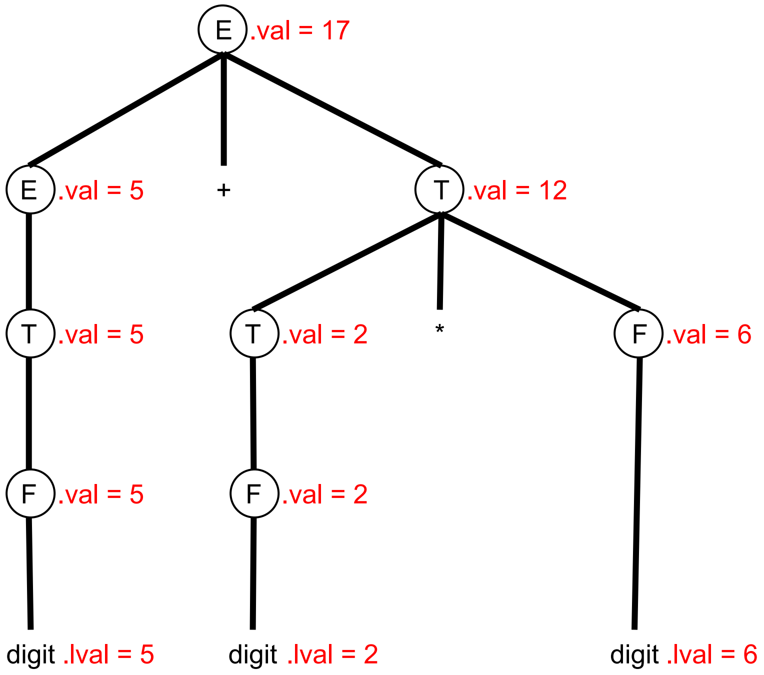
$$E \rightarrow E_1 \ \{print(E_1);\} + T$$

SDD

Input: 5+2*6

- A CFG with attributes and rules
  - Attributes are associated with grammar symbols
  - Rules are associated with productions

Production	**Semantic Rules**
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to digit$	$F.val = digit.lval$



Attributes

- Two types of attributes
  - **Synthesized**
    - A synthesized attribute at node node N is defined only in terms of attribute values of its *children* and *N itself*
  - **Inherited**
    - An inherited attribute at node node N is defined only in terms of attribute values of its *parent*, its *sibling* and *N itself*

Attributes

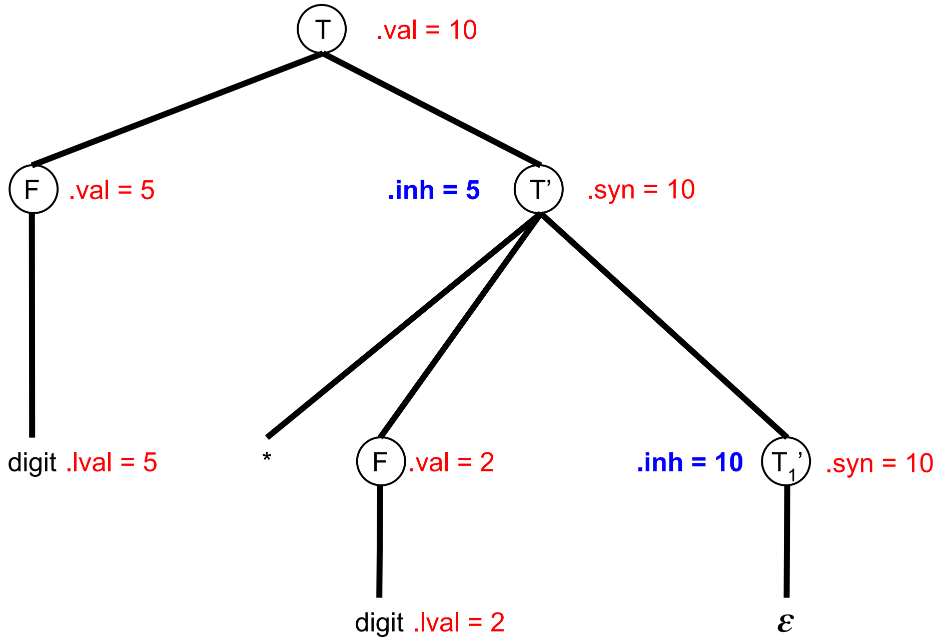
- Let, $b = f(c_1, c_2, \ldots, c_n)$ is a semantic rule for a production $A \rightarrow \alpha_1\alpha_2 \ldots \alpha_n$
  - b is a *synthesize* attribute of A, and $c_1, c_2, \ldots, c_n$ are the attributes of grammar symbols of $A \rightarrow \alpha_1\alpha_2 \ldots \alpha_n$

  - b is an *inherited* attribute of α_i, and $c_1, c_2, \ldots, c_n$ are the attributes of grammar symbols of $A \rightarrow \alpha_1\alpha_2 \ldots \alpha_n$

- Non-terminals can have both synthesized and inherited attributes
- Terminals can have only synthesized attributes
  - Lexical values supplied by the lexical analyser

SDD with inherited attributes

Production	**Semantic Rules**
$T \rightarrow FT'$	$T'.inh = F.val;$
	$T.val = T'.syn$
$T' \rightarrow * \, F \, T_1'$	$T_1'.inh = T'.inh * F.val$
	$T'.syn = T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lval$



Input: 5*2

Annotated Parse Tree

- A parse tree showing values of its attributes is called *annotated parse tree*

- Evaluation of an SDD in the parse tree
    - If all the attributes are synthesized, order of evaluation of attributes is straight-forward
        - Evaluate the attributes of all children before evaluating the attribute of the parent node
            - We can evaluate attributes in *bottom-up order*, i.e., post-order traversal of parse tree

    - If there are both synthesized and inherited attributes, order of evaluation is not fixed
        - There may not even exist any order
        - E.g., $A \rightarrow B$ $\{A.s = B.i; \quad B.i = A.s + 1\}$

Dependency Graph

- Flow of information among the attributes in a parse tree
  - An edge from an attribute to another implying that the first attribute is needed to compute the second

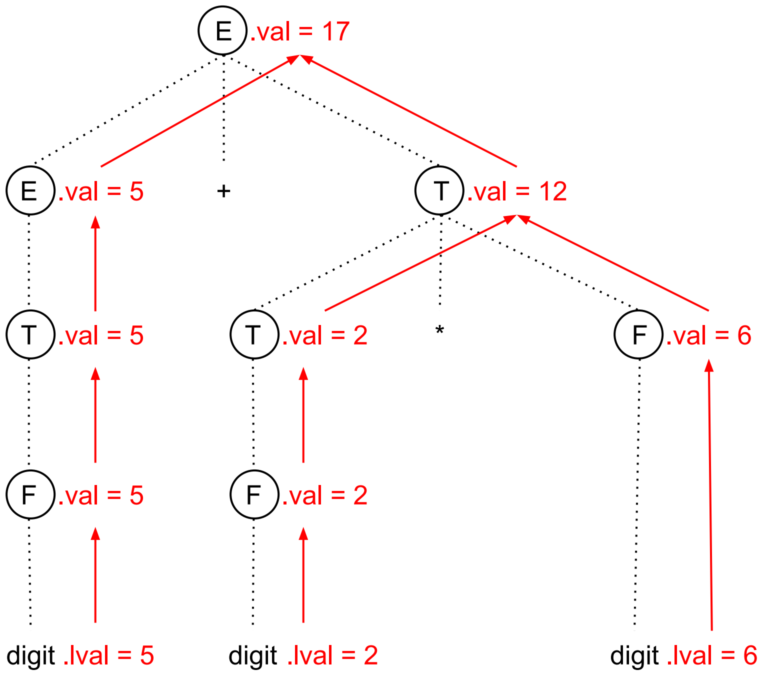
- Annotated parse tree vs. dependency graph
  - Annotated parse tree shows the values of attributes
  - Dependency graph help us determine how the values can be computed

- Gives the order of evaluation of the attributes in a parse-tree
  - If no cycle exists, we have a *topological sort* of the graph
    - A linear ordering of all its node such that if there is an edge (u, v), then u appears before v in the ordering.

Dependency graph: Example 1

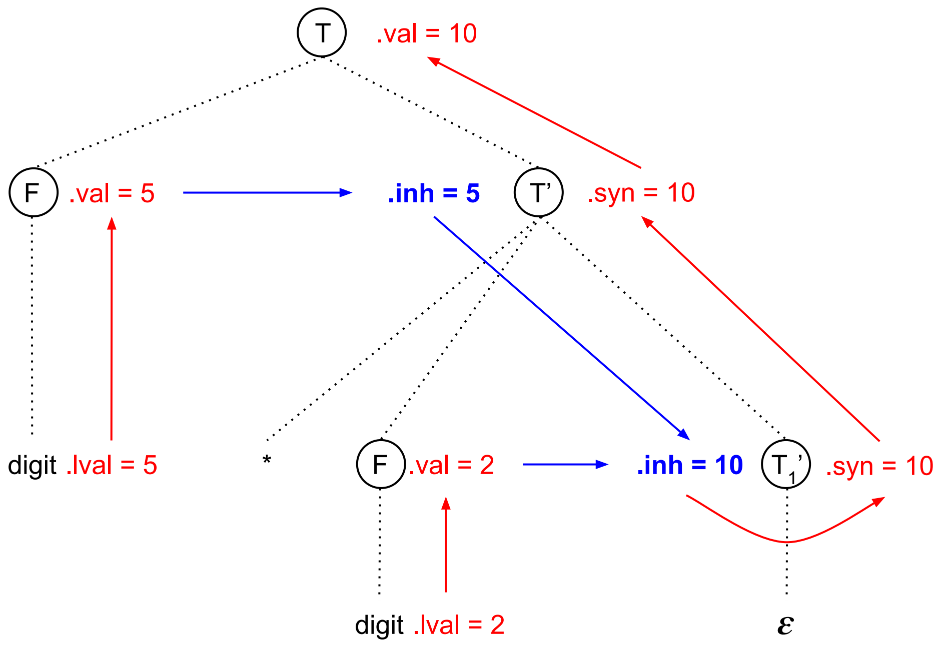
Production	**Semantic Rules**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow digit$	$F.val = digit.lval$

Input: 5+2*6



Dependency graph: Example 2

Production	**Semantic Rules**
$T \rightarrow FT'$	$T'.inh = F.val$
	$T.val = T'.syn$
$T' \rightarrow * F T_1'$	$T_1'.inh = T'.inh * F.val$
	$T'.syn = T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lval$



Input: 5*2

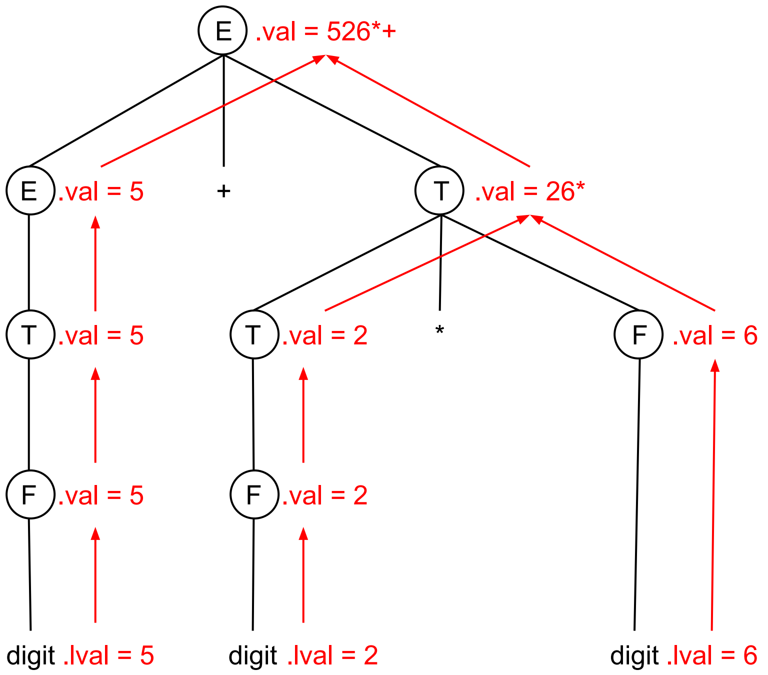
Input: 5+2*6

Output: 526*+

Applications of SDD

- Infix-to-postfix conversion

Production	**Semantic Rules**
$E \rightarrow E_1 + T$	$E.val = strcat(E_1.val, T.val, +)$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = strcat(T_1.val, F.val, *)$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow digit$	$F.val = digit.lval$

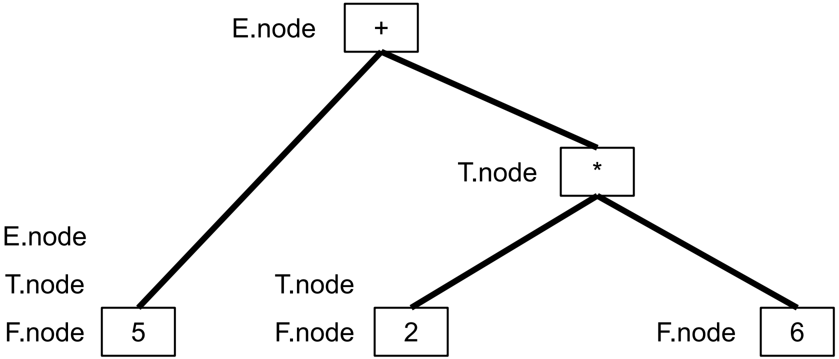


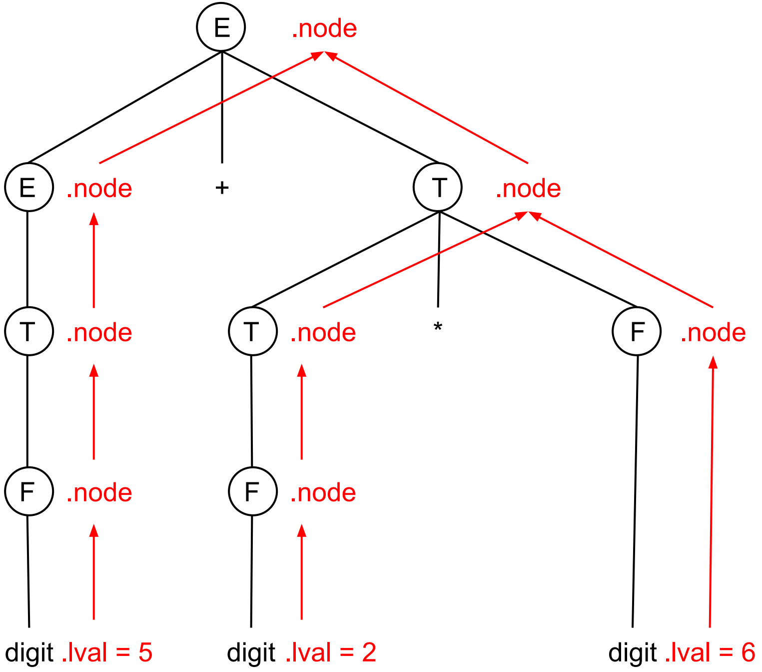
Input: 5+2*6

Applications of SDD

- Syntax Tree

Production	**Semantic Rules**
$E \rightarrow E_1 + T$	E.node = **new** *Node*('+', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow T_1 * F$	T.node = **new** *Node*('*', T_1.node, F.node)
$T \rightarrow F$	T.node = F.node
$F \rightarrow$ digit	F.node = **new** *Leaf*(digit.lval)

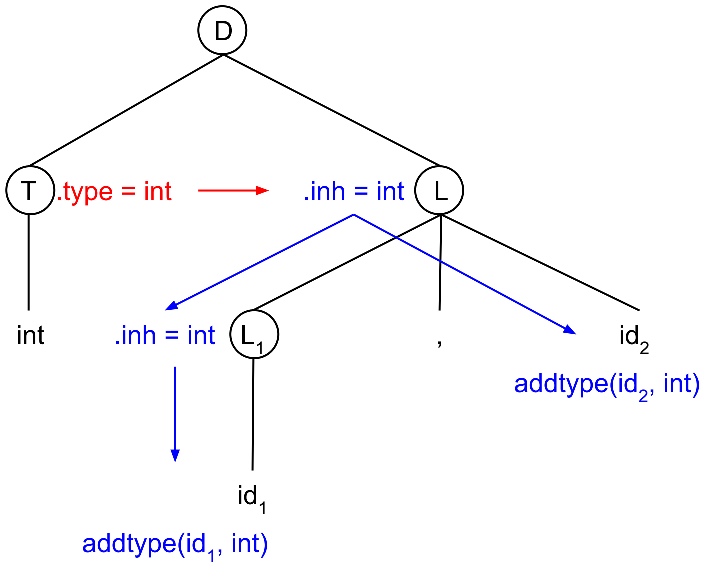




Applications of SDD

- Update the type of variable in symbol table
  - Variable declaration

```
int id₁, id₂
float id₃
```



Production	Semantic Rules
$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.inh = L.inh addtype(id.entry, L.inh)
$L \rightarrow$ id	addtype(id.entry, L.inh)

Evaluation of Semantic rules

- In SDD, the semantic rules can evaluate
  - The values of an attribute, OR
  - May have side-effects, such printing a value
    - E.g., SDD for infix-to-postfix

Production	**Semantic rules**
$E \rightarrow E_1 + T$	print ('+')
$E \rightarrow T$	
$T \rightarrow T_1 * F$	print ('*')
$T \rightarrow F$	
$F \rightarrow digit$	print (digit.lval)

- An SDD without side-effects is called *attribute grammar*

Classes of SDDs

- Given an SDD, can we detect whether there exists any parse tree whose dependency graphs have cycle?
  - While the problem is *decidable*, it is an *NP-hard* problem!

- Classes of SDDs that guarantees no cycle
  - S-Attributed Definitions
    - If every attribute is synthesized
  - L-Attributed Definitions
    - The edges between the sibling in the dependency-graph can only go from *left-to-right*
    - Formally, each attribute must be
      - Synthesized, OR
      - Inherited with the following constraints
        - If $A \rightarrow \alpha_1\alpha_2...\alpha_n$ is a production
          - Inherited attribute of $\alpha_i = f(A, \alpha_1, \alpha_2, ..., \alpha_{i-1})$

Syntax-Directed Translation Schemes

- A CFG with program construct embedded within the production bodies
- Program fragments is called *semantic actions,* and can appear at any position within a production body
    - Convention is to enclosed the semantic actions within a pair of braces to separate it with the grammar symbols
        - $A \to B$ {semantic actions} C

- Syntax-Directed Translation schemes are complementary notations of SDD
    - All applications of SDD can be implemented using SDT

Postfix Translation Schemes

- An SDT with all actions at the right end of the production bodies are called *postfix*-SDT
- An S-attributed SDD for evaluating the expression can be converted into an equivalent postfix-SDT

Production + Semantic Actions

$L \rightarrow E \backslash n$ {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}

$F \rightarrow$ digit {F.val = digit.lval}

- Since the grammar is LR and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps in the parser.

Stack implementation of *postfix* SDT

- Shift the attributes of the grammar symbols along with the symbol itself onto the stack

Symbol stack	\$	X	Y	Z		
Attribute stack	\$	$X.x$	$Y.y$	$Z.z$		

- If the attributes are all synthesized in the postfix-SDT,
  - We can compute the attribute of the head when we reduce the body with the head
    - $A \rightarrow XYZ$

Symbol stack	\$	A				
Attribute stack	\$	$A.a$				

Explicit stack implementation of *postfix* SDT

Production + Semantic Actions

$L \rightarrow E$ \n	{ print(stack[top-1].val);	top = top - 1;	}
$E \rightarrow E_1 + T$	{ stack[top-2].val = stack[top-2].val + stack[top].val);	top = top - 2;	}
$E \rightarrow T$			
$T \rightarrow T_1 * F$	{ stack[top-2].val = stack[top-2].val * stack[top].val);	top = top - 2;	}
$T \rightarrow F$			
$F \rightarrow$ digit			

Input: 2*5 \n

Output: 10

Stack	Stack (Attribute)	Input	Action
$	$	id_1 * id_2 \n $	Shift
$ id_1	$ 2	* id_2 \n $	Reduce by $F \rightarrow$ digit
$ F	$ 2	* id_2 \n $	Reduce by $T \rightarrow F$
$ T	$ 2	* id_2 \n $	Shift
$ T *	$ 2 *	id_2 \n $	Shift
$ T * id_2	$ 2 * 5	\n $	Reduce by $F \rightarrow$ digit
$ T * F	$ 2 * 5	\n $	Reduce by $T \rightarrow T*F$
$ T	$ 10	\n $	Reduce by $E \rightarrow T$
$ E	$ 10	\n $	Shift
$ E \n	$ 10	$	Reduce by $L \rightarrow E$ \n
$ L	$	$	Accept

SDT with actions inside production

- An action may be placed at any position within the body of a production.
- The action is performed after all symbols to its left are processed.
  - For the production $A \rightarrow X \{a\} Y$, action a is performed after
    - Symbol X is recognized, if X is a terminal, OR
    - All the terminals derived from X is recognized, if X is a non-terminal.
  - More precisely,
    - In bottom-up parsing, we perform a as soon as X appears on top of the stack
    - In top-down parsing, we perform a before we attempt to expand Y (for non-terminal) or check for Y on input(for a terminal).

SDTs implementation during parsing

- Classes of SDT that can be implemented during parsing
  - Postfix SDT
  - SDT that implements L-attributed definitions (We will see it soon!)

- Not all SDTs can be implemented during parsing!!
  - E.g., following SDT for infix-to-prefix conversion

Production + Semantic Actions

$E \rightarrow$ {print('+');} E_1 + T

$E \rightarrow T$

$T \rightarrow$ {print('*');} T_1 * F

$T \rightarrow F$

$F \rightarrow$ digit {print(digit.lval);}

- Not possible to implement this SDT during either top-down or bottom-up
  - Because, it has to print the operators '+' or '*' long before it knows whether these symbols will appear on in its input.

General implementation of SDT

- Any SDT can be implemented as follows
  - Ignore the actions during the construction of parse tree (i.e., during parsing)
  - For each interior node N in the parse tree (interior node represents a production $A \rightarrow \alpha$)
    - Add actions of α as the children of node *N*.
  - Perform a pre-order traversal of the tree and perform the action as soon as it is visited

Production + Semantic Actions

E → {print('+');} E_1 + T

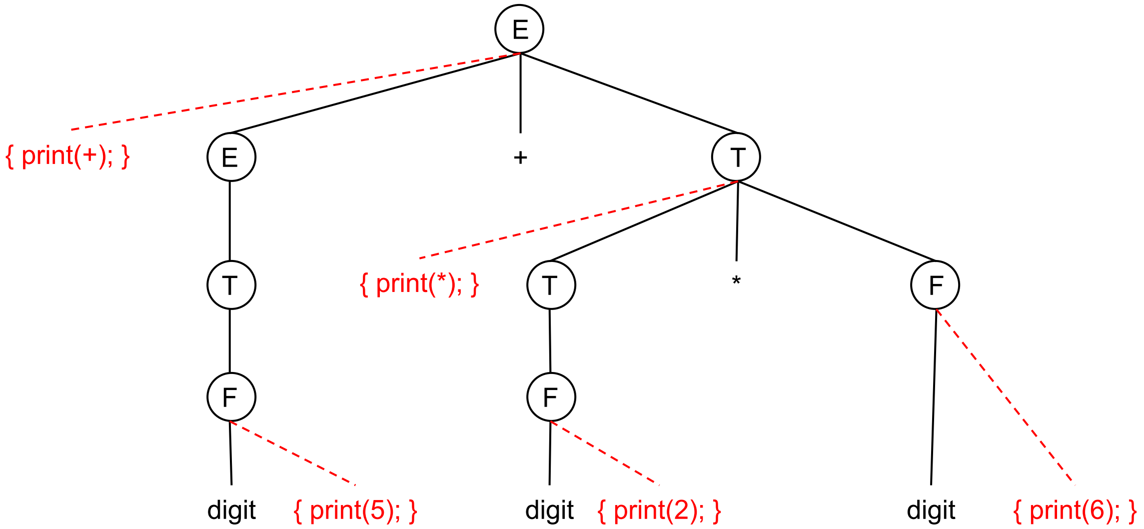
E → T

T → {print('*');} T_1 * F

T → F

F → digit {print(digit.lval);}

Input: 5+2*6



SDT with left-recursive grammar

- A left-recursive grammar can not be parsed by top-down (LL) parser.
- Removing left-recursion in an SDT also requires to handle the actions
- A simple case
  - Assume, we care about the order in which actions are performed
    - E.g., if each action simply prints a string, we care about the order of strings

$$E \rightarrow E + T \ \{print('+');\}$$
$$E \rightarrow T$$

  - While removing left-recursion, we treat the actions as the grammar symbols
    - E.g., α = + T {print('+');}

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T \ \{print('+');\}\ E'$$
$$E' \rightarrow \varepsilon$$

Input: X Y Y

Elimination of left-recursion from an SDT

- Eliminating left-recursion from an SDT that compute attributes is not straightforward.
- E.g., a left-recursive S-attributed SDT

  $A \rightarrow A_1\ Y\ \{A.a = A_1.a + Y.y\}$

  $A \rightarrow X\ \{A.a = (X.x)^2\}$

- On applying left-recursion removal technique

  $A \rightarrow X\ \{A.a = (X.x)^2\}\ R$

  $R \rightarrow Y\ \{A.a = A_1.a + Y.y\}\ R_1$

  $R \rightarrow \varepsilon$

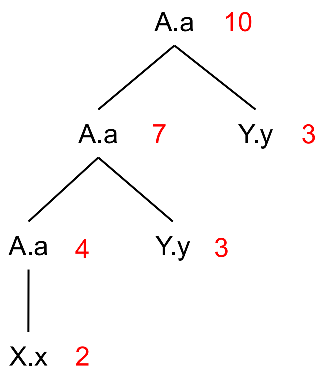
- Observe that new symbol has been introduced, so needs adjustments in the actions as well.

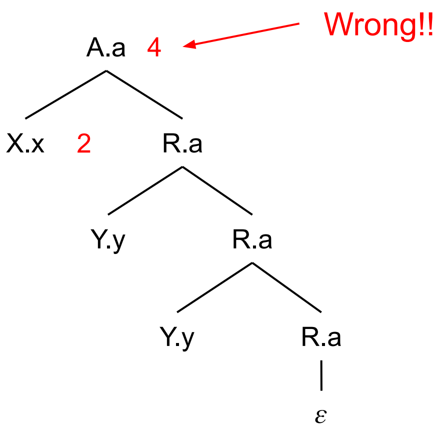
  $A \rightarrow X\ \{A.a = (X.x)^2\}\ R$

  $R \rightarrow Y\ \{R.a = R_1.a + Y.y\}\ R_1$

  $R \rightarrow \varepsilon$

- Now, for the same input





Input: X Y Y

Elimination of left-recursion from an SDT

- A left-recursive S-attributed SDT

  $A \rightarrow A_1\ Y\ \{A.a = A_1.a + Y.y\}$

  $A \rightarrow X\ \{A.a = (X.x)^2\}$

- Left-recursion removal

  $A \rightarrow X\ \{A.a = (X.x)^2\}\ R$

  $R \rightarrow Y\ \{R.a = R_1.a + Y.y\}\ R_1$

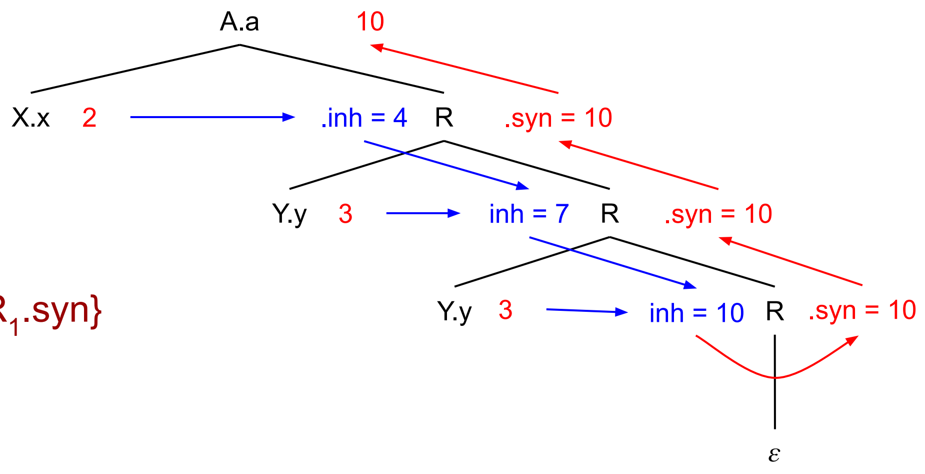
  $R \rightarrow \varepsilon$

- Correcting actions

  $A \rightarrow X\ \{R.inh = (X.x)^2\}\ R\ \{A.a = R.syn\}$

  $R \rightarrow Y\ \{R_1.inh = R.inh + Y.y\}\ R_1\ \{R.syn = R_1.syn\}$

  $R \rightarrow \varepsilon\ \ \{R.syn = R.inh\}$



SDT for L-attributed Definitions

- An L-attributed SDD can be parsed in top-down fashion
  - If not, it is almost impossible to perform the translation by either LL or LR pasers

- A general rule to convert an L-attributed SDD into SDT
  1. Embed the action that computes the inherited attributes for a non-terminal A immediately before that occurance of A in the body of the production
     - E.g., $A \rightarrow B \{C.inh = f(B.syn)\}\ C$
  2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of the production
     - E.g., $A \rightarrow B \{C.inh = f(B.syn)\}\ C \{A.syn = f(B.syn)\}$

SDT for L-attributed Definitions: Example

- SDT for intermediate code generation for a simple `while` loop
  - Generate an intermediate code for facilitating the control flow
- Grammar G:

$$S \rightarrow \text{while}\ (C)\ \ S_1$$

where

S can generate all kinds of statements

C is a conditional expression that evaluates to true or false

<u>Control-flow for a `while` statement - Conceptual</u>

- Evaluate C and if C is `true`
  - The next statement to be executed ⇒ S_1
  - After S_1, the next statement to be executed ⇒ $S_1.next = C$
- Else
  - The next statement to be executed ⇒ whatever comes after the symbol S, i.e., $S.next$

<u>Actual implementation</u>

Label L1: Evaluate C
if C is `false`, goto $S.next$, else goto L2

Label L2: Execute S_1
goto L1

String concatenation: ||

SDT for L-attributed Definitions: Example

- SDD for intermediate code generation for a simple `while` loop

$S \rightarrow$ while (C) S_1

L1 = new Label();
L2 = new Label();
S_1.next = L1;
C.false = S.next;
C.true = L2;
S.code = Label || L1 || C.code || Label || L2 || S_1.code;

Rules for SDD to SDT:

Inherited attributes immediately before the non-terminal

Synthesized attributes at the end of the production

- Corresponding SDT

$S \rightarrow$ **while (** {L1 = new Label(); L2 = new Label(); C.false = S.next; C.true = L2;} ***C*** **)**
{S_1.next = L1;} $\boldsymbol{S_1}$ { S.code = Label || L1 || C.code || Label || L2 || S_1.code; }

Implementing L-attributed SDD

- Generic approach
  - Build the parse tree, add actions, and execute the actions in pre-order.
- Specific approach
  - Translation during recursive-descent parsing
  - Translation during LL parsing
  - Translation during LR parsing

Translation during recursive-descent parsing

- Recall, in recursive-descent parsing, we have one function for each non-terminal

- Translation steps
  - The *inherited attributes* of the non-terminal $\mathcal{A}$ needs to be passed as argument to the associated function
  - The return value of the function $\mathcal{A}$ is the collection of *synthesized attributes* of non-terminal $\mathcal{A}$
  - The body of the function needs to both parse and handle attributes
    - Preserve the values of all attributes in local variables that will be needed to compute
      - The *inherited attributes* of non-terminals in body, OR
      - The *synthesized attribute* of the non-terminal head, i.e., $\mathcal{A}$.
    - Call functions associated with the non-terminals in body and provide them with the proper attributes.
      - Since, its an L-attributed SDD, all the required attributes should have been already computed. (We are doing left-to-right processing!)

Translation during recursive-descent parsing

$S \rightarrow$ **while (** {L1 = new Label(); L2 = new Label(); C.false = S.next; C.true = L2;} ***C*** **)**
{S_1.next = L1;} $\boldsymbol{S_1}$ { S.code = Label || L1 || C.code || Label || L2 || S_1.code; }

```
string S (Label next)
{
      string Scode, Ccode; /*local variables*/
      Label L1, L2; /*local variables*/
      if (current_input == 'while')
      {
            ReadNextInput();
            Match('(') and ReadNextInput();
            L1 = new Label();
            L2 = new Label();
            Ccode = C(next, L2);
            Match(')') and ReadNextInput();
            Scode = S(L1);
            return ("label" || L1 || Ccode || "label" || L2 || Scode);
      }
      else /*Other Statements*/
}
```

Translation during LL parsing

- In addition to the terminals and non-terminals, the LL(1) parser stack will hold
  - Action-record:
    - Actions to be executed
  - Synthesized-record:
    - Holds the synthesized attributes of the non-terminal
- Each non-terminal record will hold its associated inherited attributes
- Action-record for a non-terminal will be placed just above the non-terminal record
  - It will compute the inherited attributes of the non-terminals
  - It contains a pointer to *code to be executed*
- Synthesized attributes for a non-terminal are placed in a separate record immediately below the non-terminal record
  - Synthesized-record can also have *action/code* part, usually, to copy the values

Translation during LL parsing

$S \rightarrow$ **while (** {L1 = new Label(); L2 = new Label(); C.false = S.next; C.true = L2;} ***C*** **)**
{S_1.next = L1;} $\boldsymbol{S_1}$ { S.code = Label || L1 || C.code || Label || L2 || S_1.code; }

top

S	Synthesized S.code
next = x	code = ?

S on "while"

Pop S

Push the right-side of the production onto stack

---

top

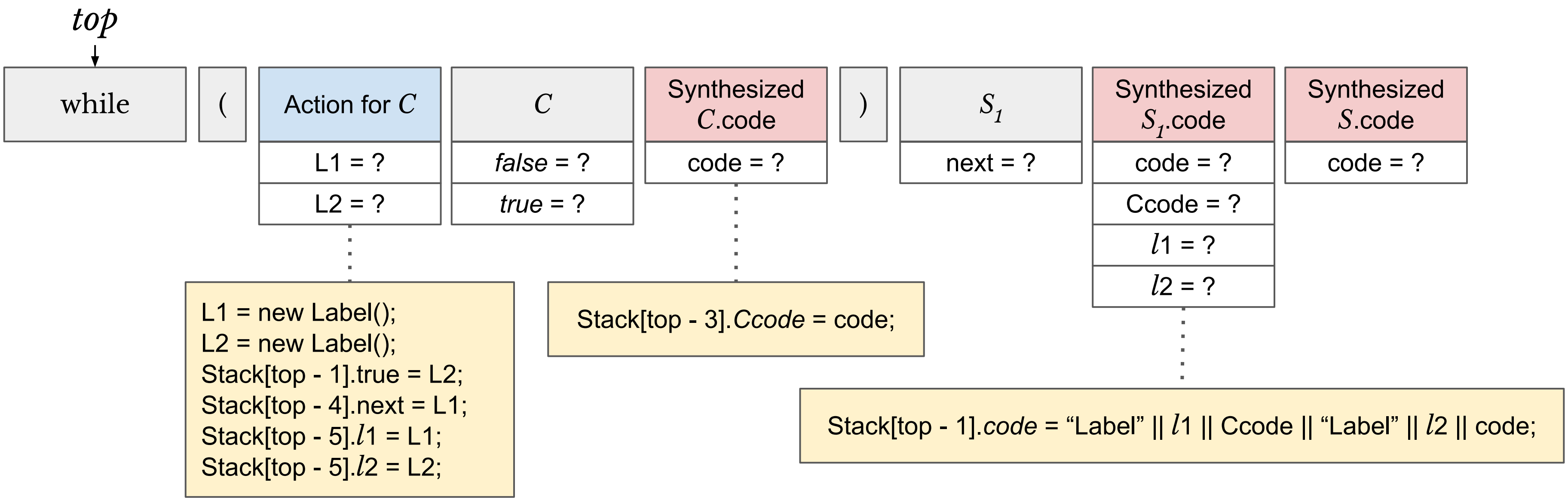
while	(	Action for C	C	Synthesized C.code	)	S_1	Synthesized S_1.code	Synthesized S.code
		L1 = ?	*false* = ?	code = ?		next = ?	code = ?	code = ?
		L2 = ?	*true* = ?				Ccode = ?	
							$l1$ = ?	
							$l2$ = ?	

L1 = new Label();
L2 = new Label();
Stack[top - 1].true = L2;
Stack[top - 4].next = L1;
Stack[top - 5].$l1$ = L1;
Stack[top - 5].$l2$ = L2;

Stack[top - 3].*Ccode* = code;

Stack[top - 1].*code* = "Label" || $l1$ || Ccode || "Label" || $l2$ || code;

Translation during LL parsing



If *top* matched the next input symbol (i.e., “while”)

Pop it

Translation during LL parsing

top

(	Action for C	C	Synthesized C.code	)	S_1	Synthesized S_1.code	Synthesized S.code
	L1 = ?	*false* = ?	code = ?		next = ?	code = ?	code = ?
	L2 = ?	*true* = ?				Ccode = ?	
						$l1$ = ?	
						$l2$ = ?	

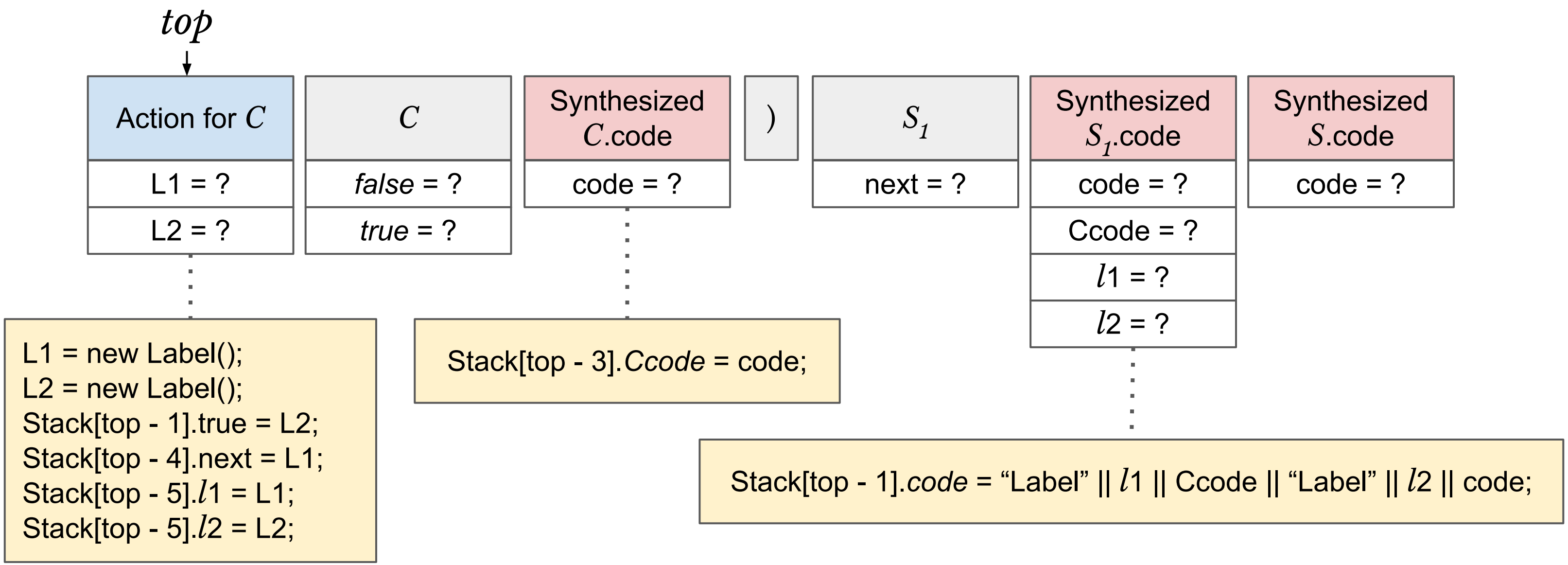
L1 = new Label();
L2 = new Label();
Stack[top - 1].true = L2;
Stack[top - 4].next = L1;
Stack[top - 5].$l1$ = L1;
Stack[top - 5].$l2$ = L2;

Stack[top - 3].*Ccode* = code;

Stack[top - 1].*code* = “Label” || $l1$ || Ccode || “Label” || $l2$ || code;

If *top* matched the next input symbol (i.e., “(”)
Pop it

Translation during LL parsing



If *top* is Action,

Execute the code

Pop

Translation during LL parsing

top

Action for C	C	Synthesized C.code	)	S_1	Synthesized S_1.code	Synthesized S.code
L1 = ?	*false* = ?	code = ?		**next = L1**	code = ?	code = ?
L2 = ?	***true* = L2**				Ccode = ?	
					l1 = L1	
					l2 = L2	

```
L1 = new Label();
L2 = new Label();
Stack[top - 1].true = L2;
Stack[top - 4].next = L1;
Stack[top - 5].l1 = L1;
Stack[top - 5].l2 = L2;
```

Stack[top - 3].*Ccode* = code;

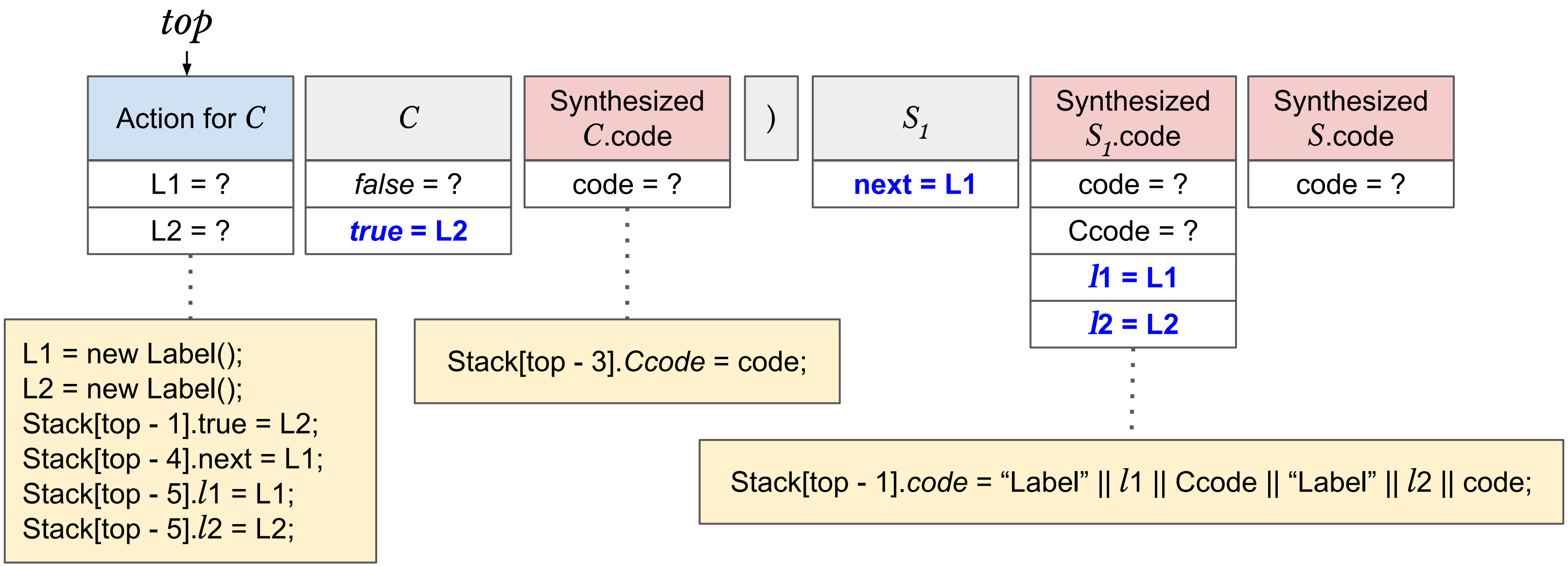
Stack[top - 1].*code* = “Label” || l1 || Ccode || “Label” || l2 || code;

If *top* is Action,

Execute the code

Pop

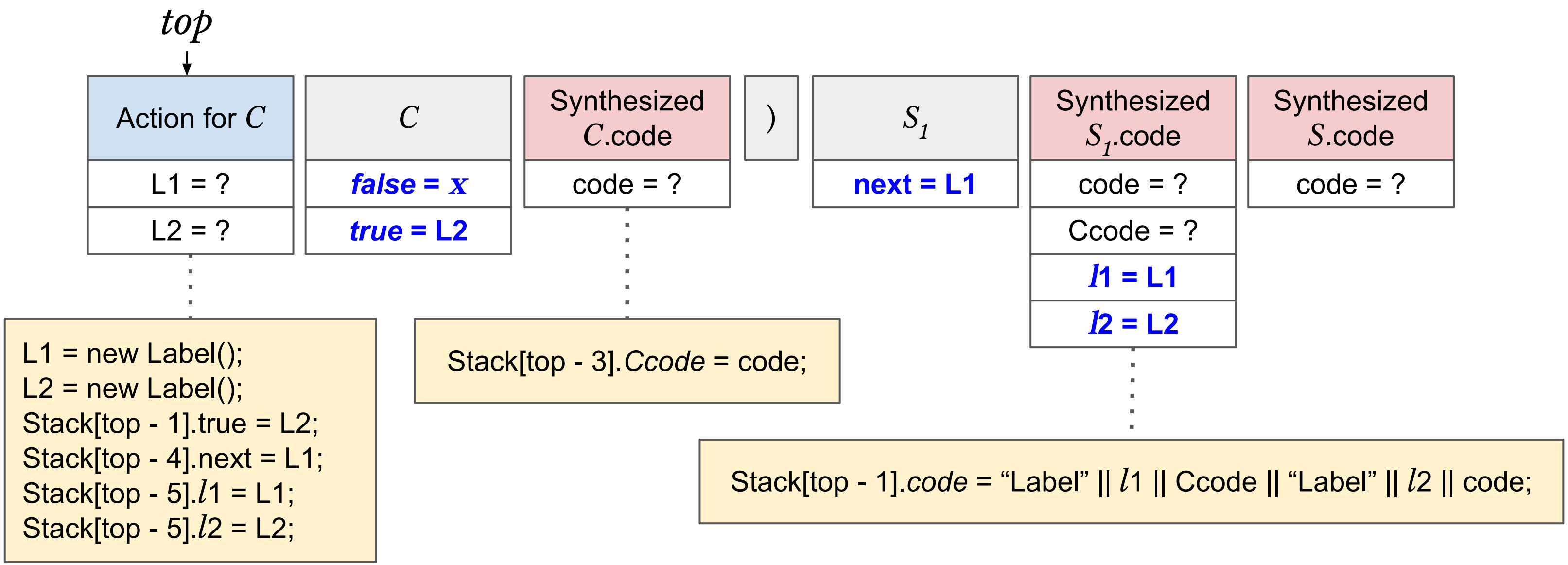
Translation during LL parsing



- Observe, we have computed the inherited attribute of S_1.next in the Action for C
- In the same way, the inherited attribute of C.false was computed during the Action for S
  - C.false = S.next

If *top* is Action,
Execute the code
Pop

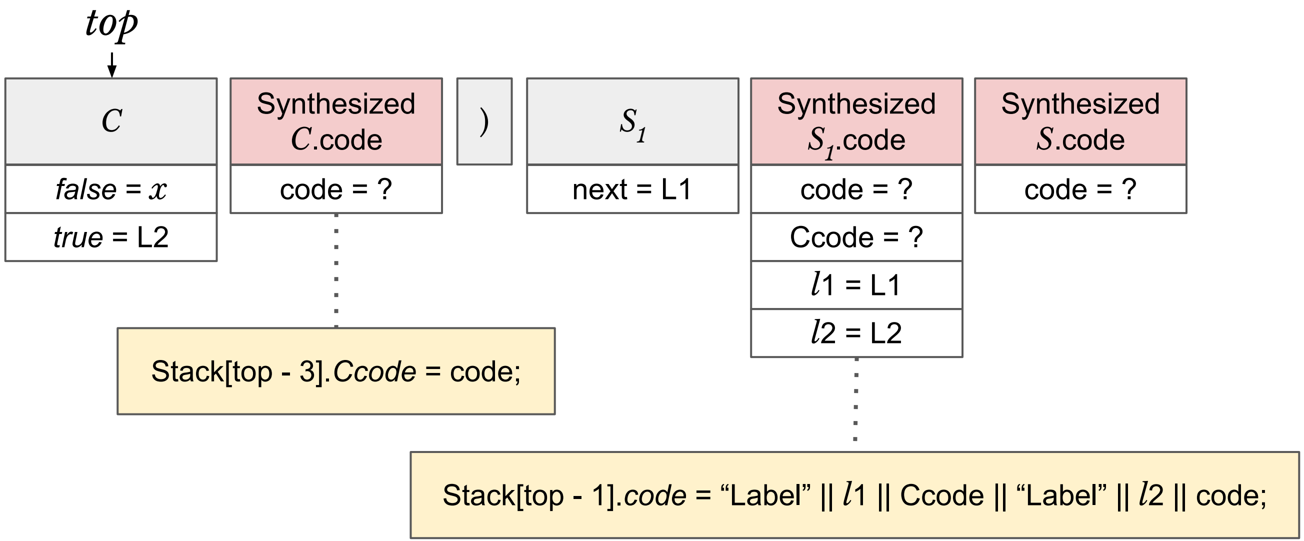
Translation during LL parsing



- Observe, we have computed the inherited attribute of S_1.next in the Action for C
- In the same way, the inherited attribute of C.false was computed during the Action for S
  - C.false = S.next

If *top* is Action,
Execute the code
Pop

Translation during LL parsing

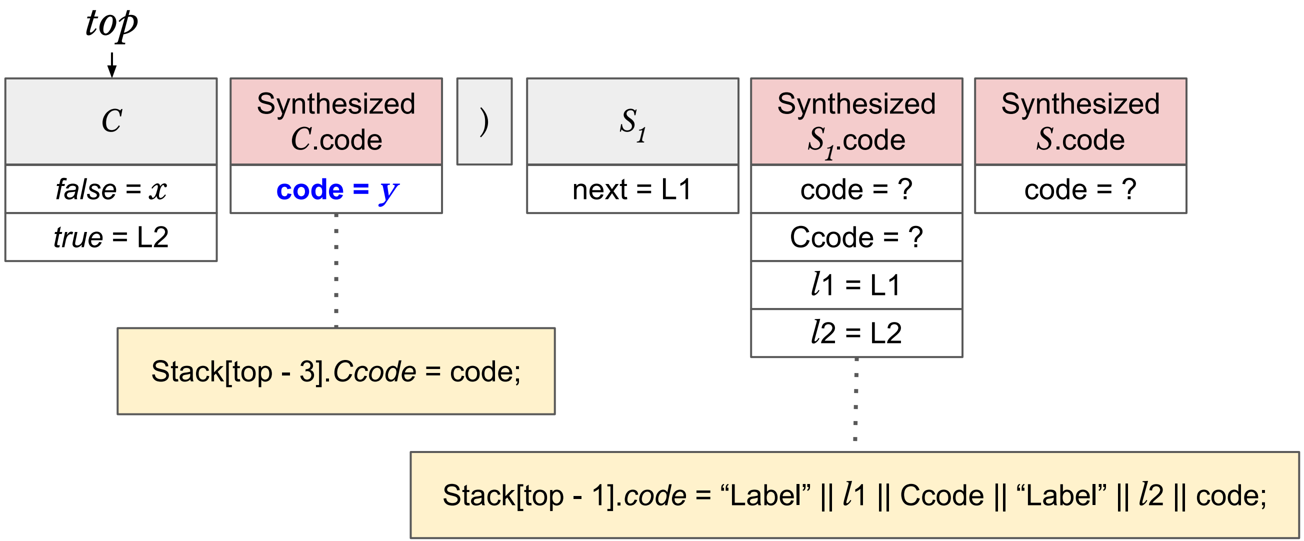


If *top* is a non-terminal, i.e., C,

Expand C and match the condition expression with input.

While processing C, we can generate the synthesize attribute C.code

Translation during LL parsing

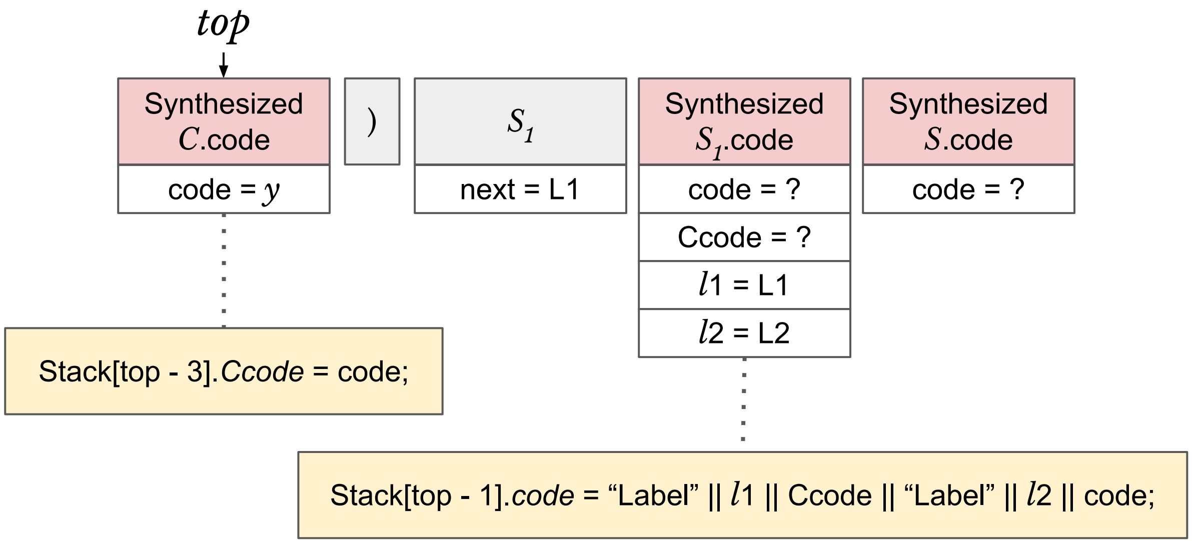


If *top* is a non-terminal, i.e., C,

Expand C and match the condition expression with input.

While processing C, we can generate the synthesize attribute C.code

Translation during LL parsing

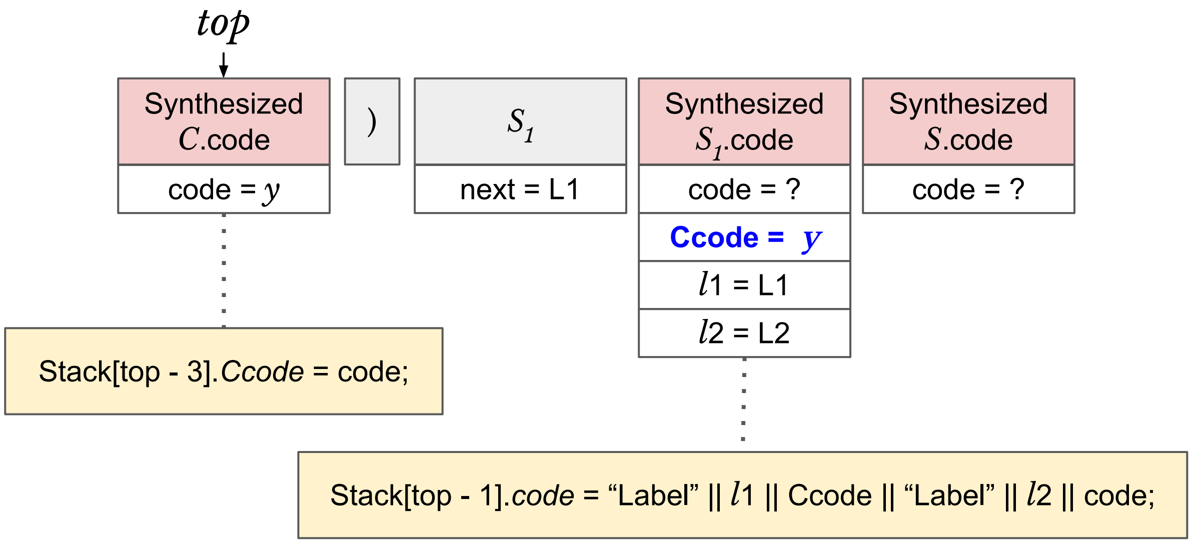


If *top* is synthesized-record

Execute the code, if any

Pop

Translation during LL parsing

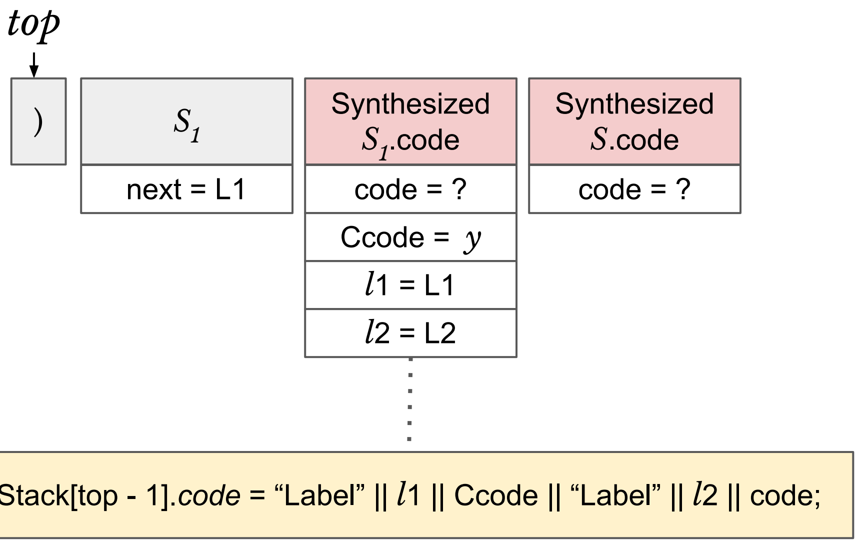


If *top* is synthesized-record

Execute the code, if any

Pop

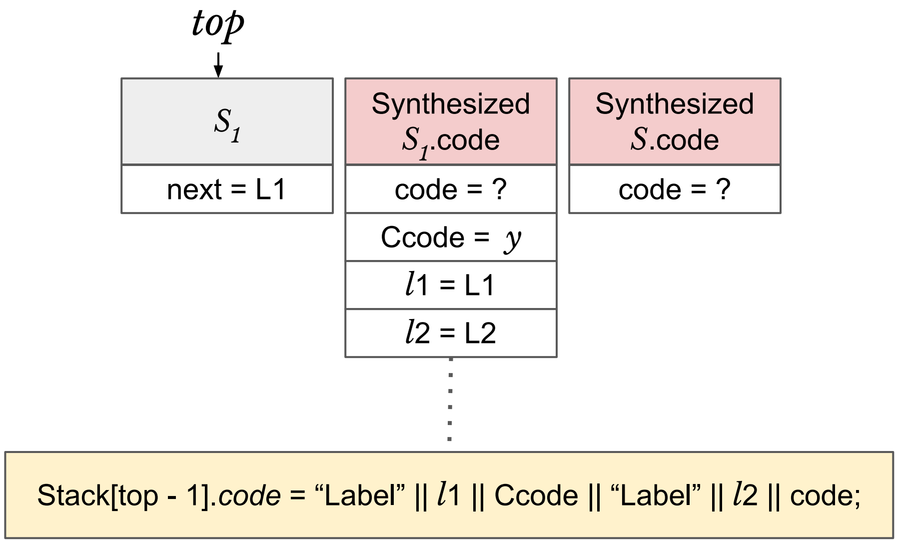
Translation during LL parsing



If *top* is matched with the input, i.e., ‘)’

Pop

Translation during LL parsing

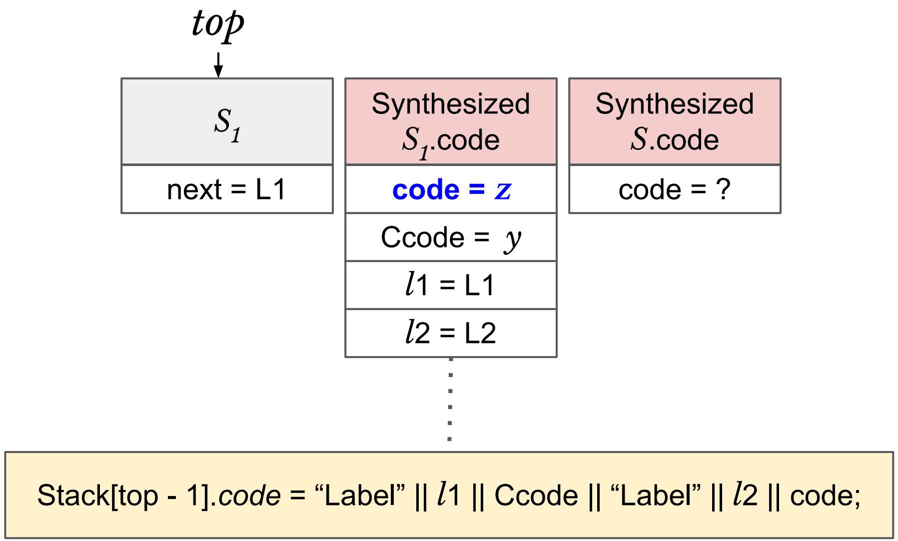


If *top* is a non-terminal, i.e., S_1,

Expand S_1 and process the children

While processing S_1, we can generate the synthesize attribute S_1.code

Translation during LL parsing

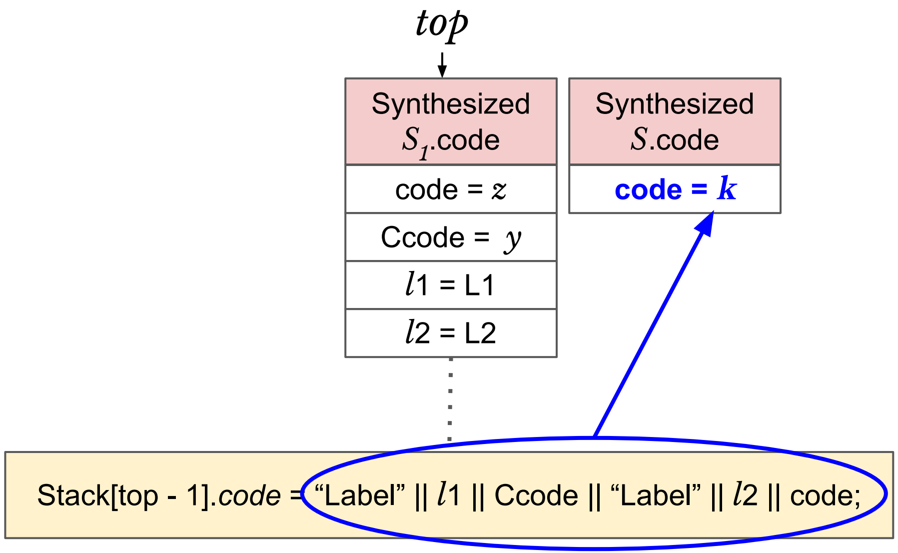


If *top* is a non-terminal, i.e., S_1,

Expand S_1 and process the children

While processing S_1, we can generate the synthesize attribute S_1.code

Translation during LL parsing

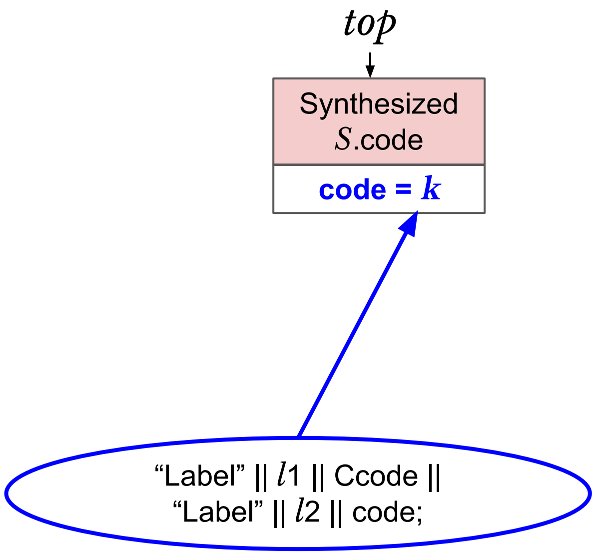


If top is synthesized-record

Execute the code, if any

Pop

Translation during LL parsing



If *top* is synthesized-record

  Execute the code, if any

  Pop

Exercise: Work out on the details

Translation during LR parsing

- Convert the L-attributed SDT into postfix-SDT
  - Move all embedding semantic actions in SDT to the end of the production rules
  - Introduce new non-terminals
  - Copy all inherited attributes into the synthesized attributes (most of the time synthesized attributes of new non-terminals)
- Evaluate all semantic actions during reductions
- Transformation
  - Remove an embedding semantic action S_i, put a new non-terminal M_i instead of that semantic action
  - Put the semantic action S_i into the end of a new production rule $M_i \rightarrow \varepsilon$ for that non-terminal M_i
  - Semantic action S_i will be evaluated when the new production rule is reduced
  - Evaluation order of the semantic rules are not changed by this transformation
- All L-attributed definitions cannot be evaluated during bottom-up parsing
  - The modified grammar is not an LR grammar anymore

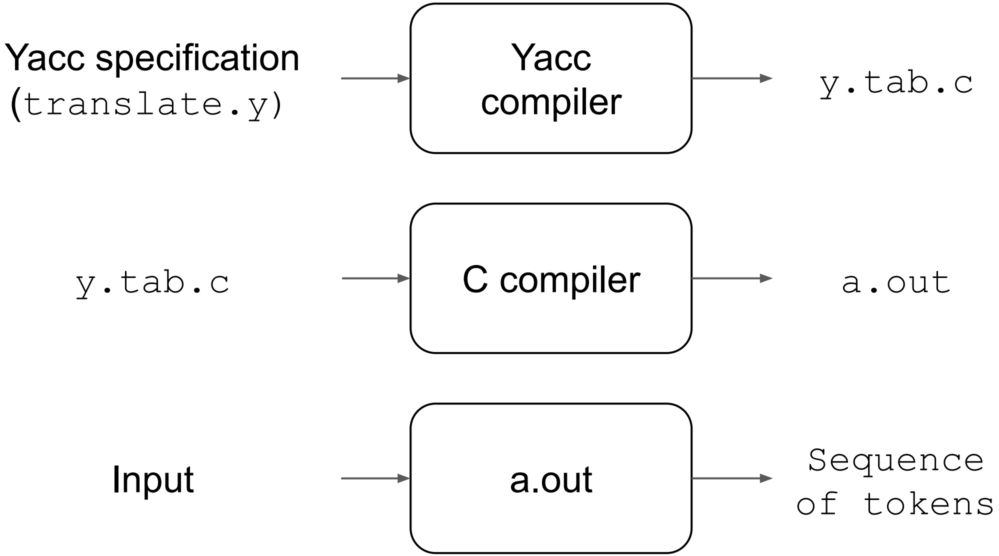
Topics Covered

- Syntax-Directed Definition (SDD) and Syntax-Directed Translation (SDT)
- Inherited and Synthesized Attributes
- Dependency graph, Annotated parse tree, Attributed SDD
- S-Attributed Definitions
- L-Attributed Definitions
- Syntax-Tree
- Implementation of S-Attributed SDD’s
    - Postfix-SDT
- Elimination of left-recursion from SDT’s
- Implementation of L-Attributed SDD’s by Recursive-Descent Parsing
- Implementation of L-Attributed SDD’s by LL Parsing
- Implementation of L-Attributed SDD’s by LR Parsing

Parser Generator

YACC parser generator

- A tool to generate parse tree
- Yet-another-compiler-compiler




```
Declaration
%%
Transition rules
%%
Auxiliary functions
```

A typical yacc file: `translate.y`

YACC parser generator

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line    :       expr '\n'                    {printf("%d", $1);}
        ;
expr    :       expr '+' term                {$$ = $1 + $3;}
        |       term
        ;
term    :       term '*' factor              {$$ = $1 * $3;}
        |       factor
        ;
factor  :       '(' expr ')'                 {$$ = $2;}
        |       DIGIT
        ;
%%
yylex()
{
        int c; c = getchar();
        if (isdigit(c)) { yylval = c; return DIGIT;} else return c;
}
```