

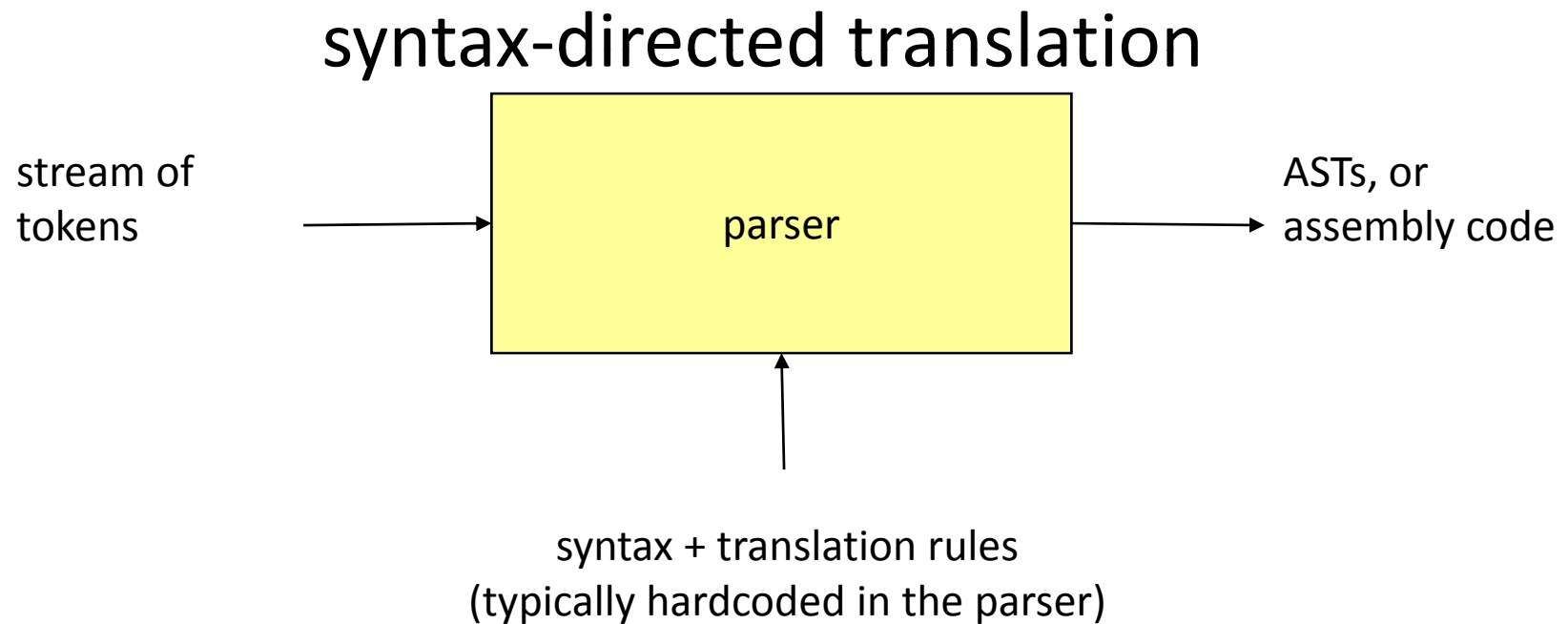
Syntax-Directed Translation

Chapter 5

Bad Translation

- CS581 is a very boring class
 - Use babelfish translator
 - English to Chinese, then Chinese to English
- CS581 is the extremely tasteless kind

Motivation: parser as a translator



Syntax-Directed Definitions

- A **syntax-directed definition (SDD)** is a context free grammar together with attributes and rules
 - Attributes are associated with grammar symbols
 - Rules are associated with productions
- A **synthesized attribute** for A at a parse tree node N is defined only in terms of attributes at the children of N and at N itself
- An **inherited attribute** for B at a parse tree node N is defined only in terms of attributes at the N's parent, N itself, and N's siblings

SDD Example 1

Production

Semantic Rules

$E \rightarrow E + T$

$E_1.val = E_2.val + T.val$

$E \rightarrow T$

$E.val = T.val$

$T \rightarrow T * F$

$T_1.val = T_2.val * F.val$

$T \rightarrow F$

$T.val = F.val$

$F \rightarrow \text{int}$

$F.val = \text{int.lexval}$

$F \rightarrow (E)$

$F.val = E.val$

- An SDD that involves only synthesized attributes is called S-attributed

SDD Example 2

Production

Semantic Rules

$T \rightarrow F T'$

$T'.inh = F.val$

$T.val = T'.syn$

$T' \rightarrow *FT'_1$

$T'_1.inh = T'.inh * F.val$

$T'.syn = T'_1.syn$

$T' \rightarrow \epsilon$

$T'.syn = T'.inh$

$F \rightarrow int$

$F.val = int.lexval$

- Annotated parse tree for $3*5$

TEST YOURSELF #1

- A CFG for the language of binary numbers:
 - $B \rightarrow 0$
 - $\rightarrow 1$
 - $\rightarrow B 0$
 - $\rightarrow B 1$
- Define a syntax-directed translation so that the translation of a binary number is its base-10 value.
- Draw the parse tree for 1001 and annotate each nonterminal with its translation.

Evaluation Orders for SDDs

- S-Attributed Definitions

```
postorder(N) {  
    for (each child C of N, from the left)  
        postorder (C);  
    evaluate the attributes associated with N;  
}
```

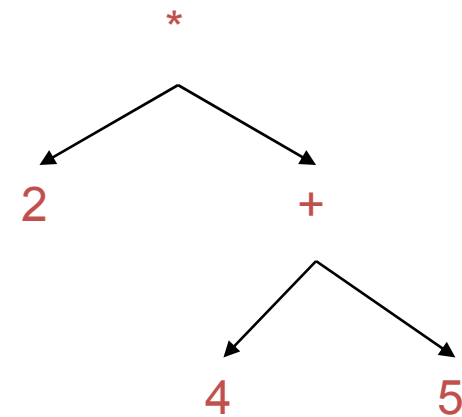
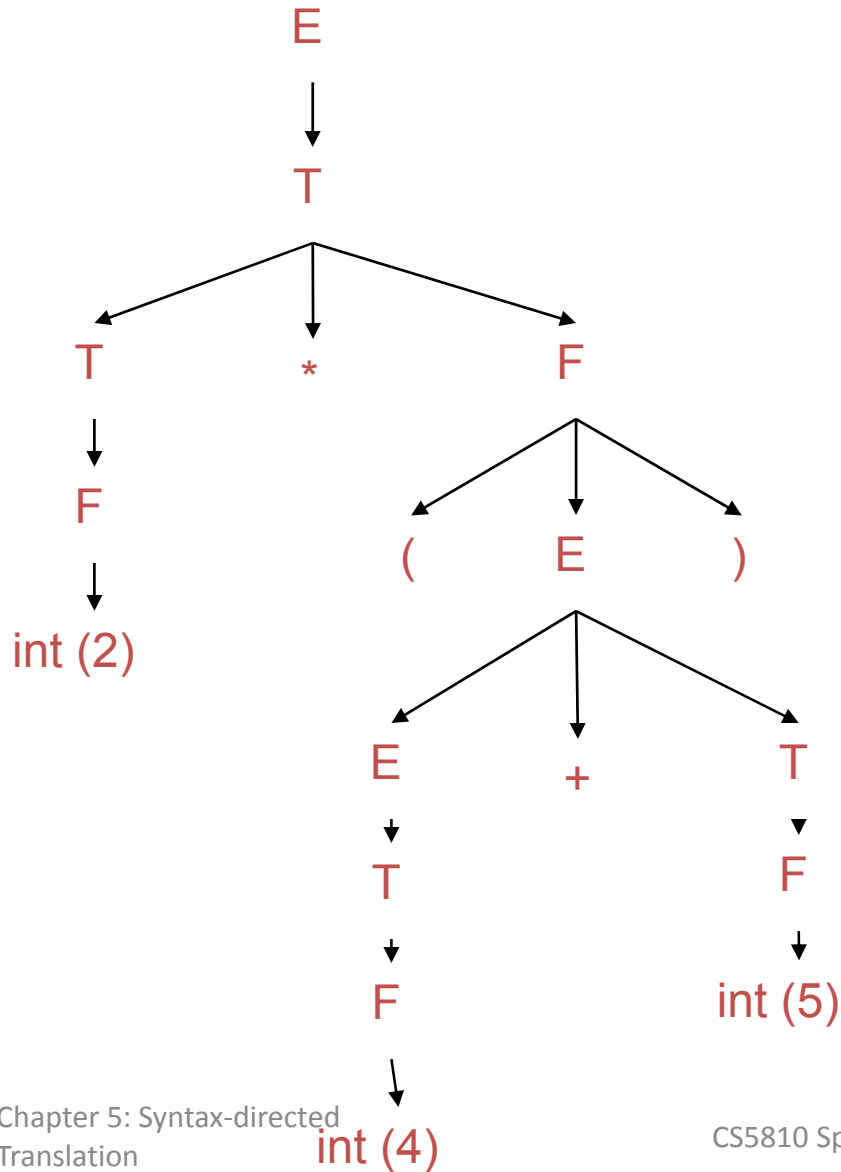
L-Attributed Definitions

- Each attribute must be either
 - Synthesized, or
 - Inherited with limited rules. For $A \rightarrow X_1 X_2 \dots X_n$, and there is an inherited attribute $X_i.a$ computed by a rule. The rule must use only
 - Inherited attributes of head A
 - Either inherited or synthesized attributes of X_1, X_2, \dots, X_{i-1}
 - Either inherited or synthesized attributes of X_i , but only if there are no cycles
- $A \rightarrow BC$ with semantic rules $A.s=B.b$; $B.i=f(C.c, A.s)$ is not L-attributed

AST vs Parse Tree

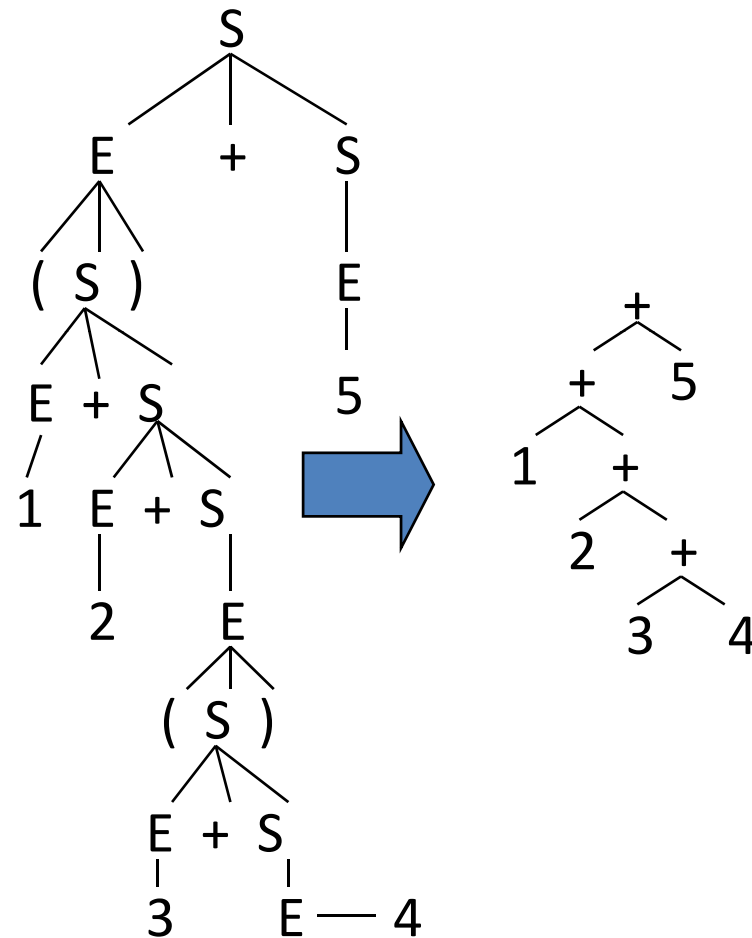
- AST is a better structure for later compiler stages
 - omits details having to do with the source language,
 - only contains information about the essential structure of the program.
 - e.g., parentheses, commas, semi-colons
- AST is condensed form of a parse tree
 - operators appear at *internal* nodes, not at leaves.
 - "Chains" of single productions are collapsed.
 - Lists are "flattened".
 - Syntactic details are ommitted

Example: $2 * (4 + 5)$: parse tree vs AST



Abstract Syntax Tree (AST)

- Derivation = sequence of applied productions
 - $S \rightarrow E+S \rightarrow 1+S \rightarrow 1+E \rightarrow 1+2$
- Parse tree = graph representation of a derivation
 - Doesn't capture the order of applying the productions
- AST discards unnecessary information from the parse tree



AST Construction

- We want to explicitly construct the AST during the parsing phase

Constructing Syntax Tree

Production

Semantic Rules

$E \rightarrow E + T$

$E_1.\text{node} = \text{new Node}('+', E_2.\text{node}, T.\text{node})$

$E \rightarrow T$

$E.\text{node} = T.\text{node}$

$T \rightarrow T * F$

$T_1.\text{node} = \text{new Node}('*', T_2.\text{node}, F.\text{node})$

$T \rightarrow F$

$T.\text{node} = F.\text{node}$

$F \rightarrow \text{int}$

$F.\text{node} = \text{new leaf}(\text{int}, \text{int.val})$

$F \rightarrow (E)$

$F.\text{node} = E.\text{node}$

TEST YOURSELF #2

- Illustrate the syntax-directed translation defined above by
 - drawing the parse tree for $2 + 3 * 4$, and
 - annotating the parse tree with its translation
 - i.e., each nonterminal X in the parse tree will have a pointer to the root of the AST subtree that is the translation of X .

Constructing Syntax Tree

Production

Semantic Rules

$T \rightarrow F T'$

$T.\text{node} = T'.\text{syn}$

$T'.\text{inh} = F.\text{node}$

$T' \rightarrow *FT'_1$

$T'_1.\text{inh} = \text{new node}('*', T'.\text{inh}, F.\text{node})$

$T'.\text{syn} = T'_1.\text{syn}$

$T' \rightarrow \epsilon$

$T'.\text{syn} = T'.\text{inh}$

$F \rightarrow \text{int}$

$F.\text{node} = \text{new leaf}(\text{int}, \text{int.val})$

- Annotated parse tree for $3*5$

Example 2: Compute the type of an expression

$E \rightarrow E + E$ if $((E_2.trans == INT) \text{ and } (E_3.trans == INT))$
 then $E_1.trans = INT$
 else $E_1.trans = ERROR$

$E \rightarrow E \text{ and } E$ if $((E_2.trans == BOOL) \text{ and } (E_3.trans == BOOL))$
 then $E_1.trans = BOOL$
 else $E_1.trans = ERROR$

$E \rightarrow E == E$ if $((E_2.trans == E_3.trans) \text{ and } (E_2.trans != ERROR))$
 then $E_1.trans = BOOL$
 else $E_1.trans = ERROR$

$E \rightarrow \text{true}$ $E.trans = BOOL$

$E \rightarrow \text{false}$ $E.trans = BOOL$

$E \rightarrow \text{int}$ $E.trans = INT$

$E \rightarrow (E)$ $E_1.trans = E_2.trans$

Syntax-Directed Translation Schemes (SDT)

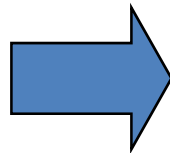
- SDT is a CFG with program fragments embedded within production bodies
 - Any SDT can be implemented by first building a parse tree, but not efficient
 - Typically SDT's are implemented during parsing
 - LR-grammar, and the SDD is S-attributed
 - LL-grammar, and the SDD is L-attributed

AST Construction - LL

LL parsing: extend procedures for non-terminals

$S \rightarrow ES'$
$S' \rightarrow \varepsilon \mid +S$
$E \rightarrow \text{num} \mid (S)$

```
void parse_S() {  
    switch (token) {  
        case num: case '(':  
            parse_E();  
            parse_S'();  
            return;  
        default:  
            ParseError();  
    }  
}
```



```
Expr parse_S() {  
    switch (token) {  
        case num: case '(':  
            Expr left = parse_E();  
            Expr right = parse_S'();  
            if (right == NULL) return left  
            else return new Node(left,right);  
        default:  
            ParseError();  
    }  
}
```

AST Construction - LR

- We again need to add code for explicit AST construction
- AST construction mechanism
 - Store parts of the tree on the stack
 - For each nonterminal symbol X on stack, also store the sub-tree rooted at X on stack
 - Whenever the parser performs a reduce operation for a production $X \rightarrow \gamma$, create an AST node for X

Implementing AST

Production

Actions

$E \rightarrow E + T$

```
stack[top-2]= new Node('+',stack[top-2].node,stack[top].node);
```

```
top = top - 2;
```

$E \rightarrow T$

$T \rightarrow T * F$

```
stack[top-2]= new Node('*',stack[top-2].node,stack[top].node);
```

```
top = top - 2;
```

$T \rightarrow F$

$F \rightarrow \text{int}$

$F \rightarrow (E)$

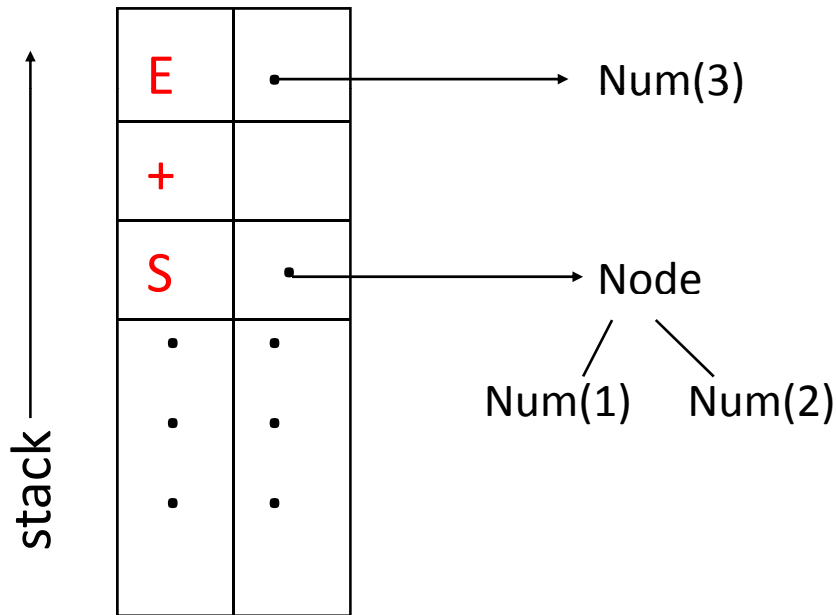
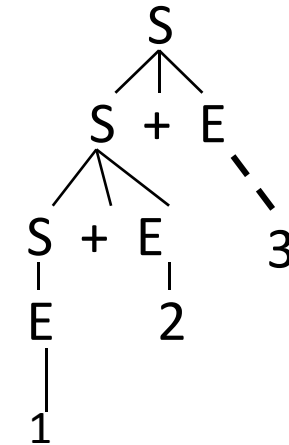
```
stack[top-2]=stack[top].node;
```

```
top = top - 2;
```

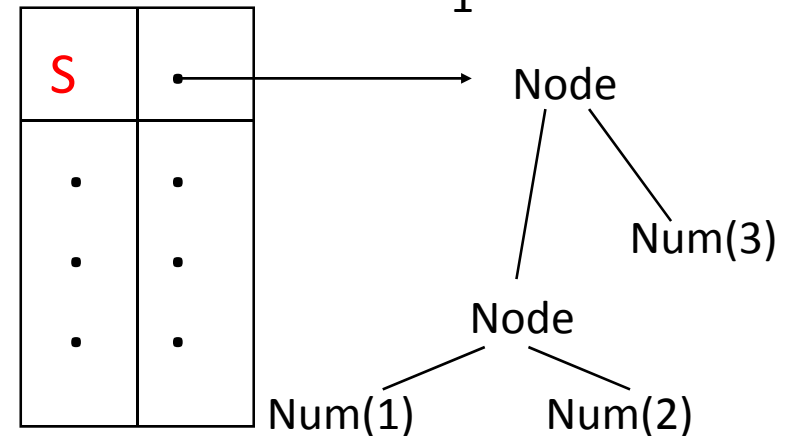
AST Construction for LR - Example

$S \rightarrow S + E \mid E$
$E \rightarrow \text{num} \mid (S)$

input string: "1 + 2 + 3"



Before reduction: $S \rightarrow E + S$



After reduction: $S \rightarrow E + S$