

Syntax-Directed Translation

Lecture 14

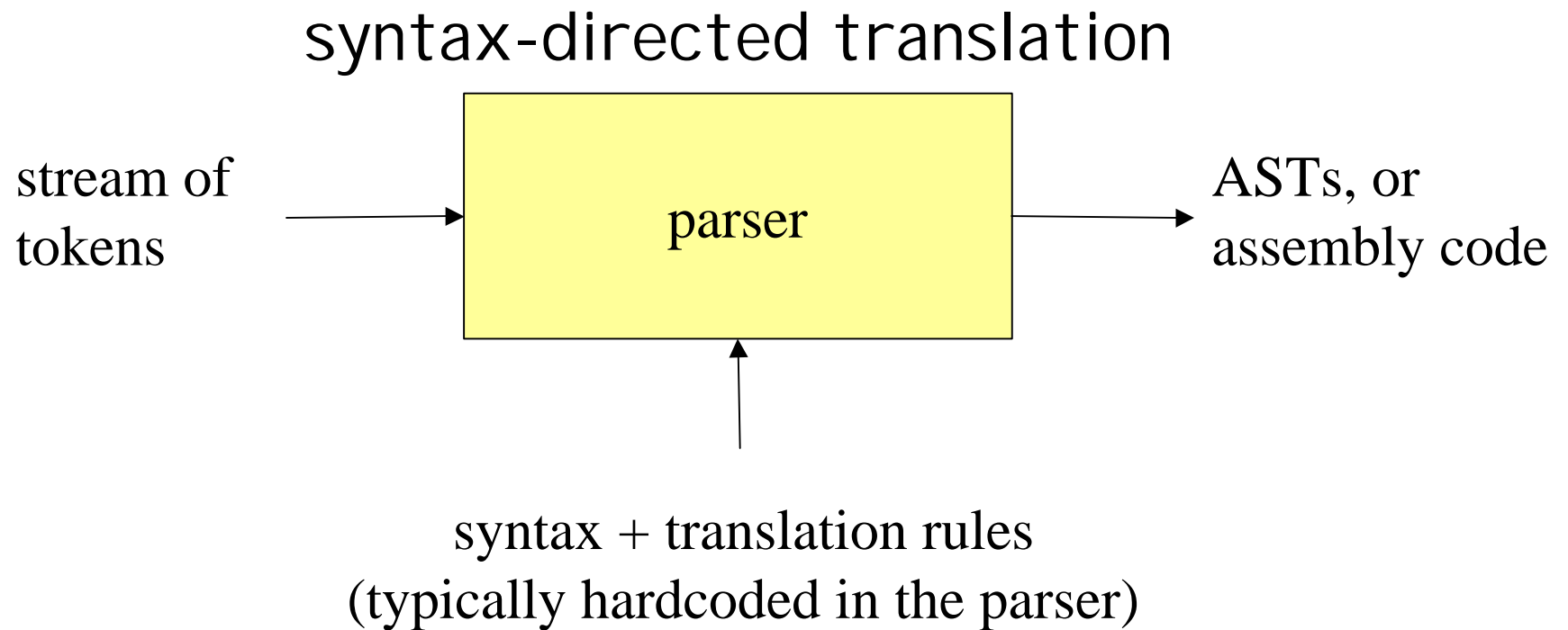
Harry Potter has arrived in China, riding the biggest initial print run for a work of fiction here since the Communist Party came to power 51 years ago. It may take a bit of Hogwarts's magic to make those books disappear.

testing Babelfish:

*English
to German to French
to English
to French to German
to English*

Toepler Harry arrived to China and is here the largest unit of magnetic cards for a work of the invention met, since the communist party came, in order to begin 51 years. It can take few magic Hogwarts, in order to make, around to disappear these books.

Motivation: parser as a translator



Mechanism of syntax-directed translation

- syntax-directed translation is done by extending the CFG
 - a *translation rule* is defined for each production

given

$$X \rightarrow d A B c$$

the translation of X is defined in terms of

- translation of nonterminals A, B
- values of attributes of terminals d, c
- constants

To translate an input string:

1. Build the parse tree.
2. Working bottom-up
 - Use the translation rules to compute the translation of each nonterminal in the tree

Result: the translation of the string is the translation of the parse tree's root nonterminal.

Why bottom up?

- a nonterminal's value may depend on the value of the symbols on the right-hand side,
- so translate a non-terminal node only after children translations are available.

Example 1: arith expr to its value

Syntax-directed translation:

the CFG translation rules

$E \rightarrow E + T$ $E_1.trans = E_2.trans + T.trans$

$E \rightarrow T$ $E.trans = T.trans$

$T \rightarrow T * F$ $T_1.trans = T_2.trans * F.trans$

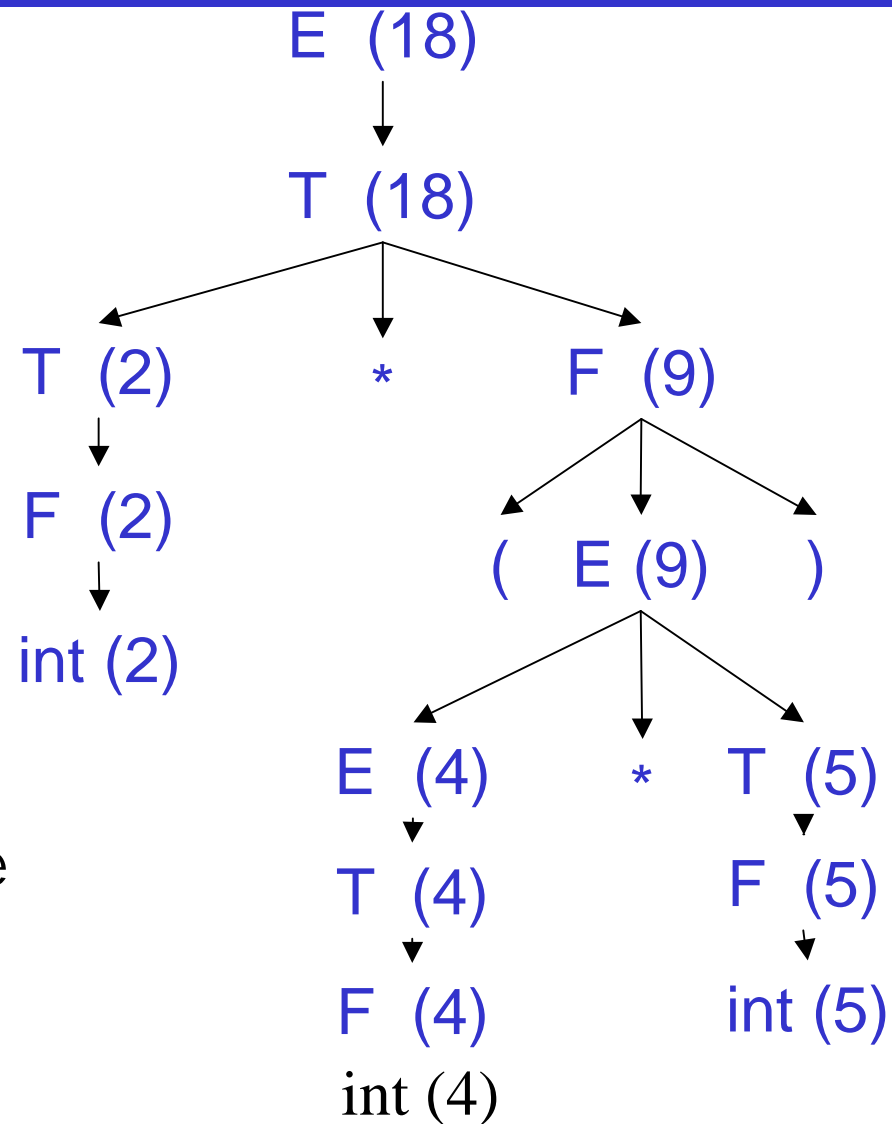
$T \rightarrow F$ $T.trans = F.trans$

$F \rightarrow int$ $F.trans = int.value$

$F \rightarrow (E)$ $F.trans = E.trans$

Example 1 (cont)

Input: $2 * (4 + 5)$



Annotated Parse Tree

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 $E_1.trans = ERROR$

$E \rightarrow true$ $E.trans = BOOL$

$E \rightarrow false$ $E.trans = BOOL$

$E \rightarrow int$ $E.trans = INT$

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

Example 2 (cont)

- Input: $(2 + 2) == 4$
 1. parse tree:
 2. annotation:

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.

Building Abstract Syntax Trees

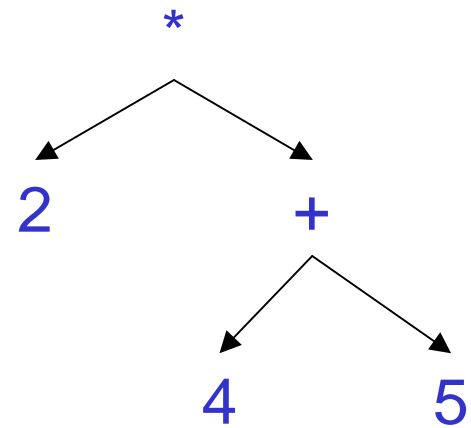
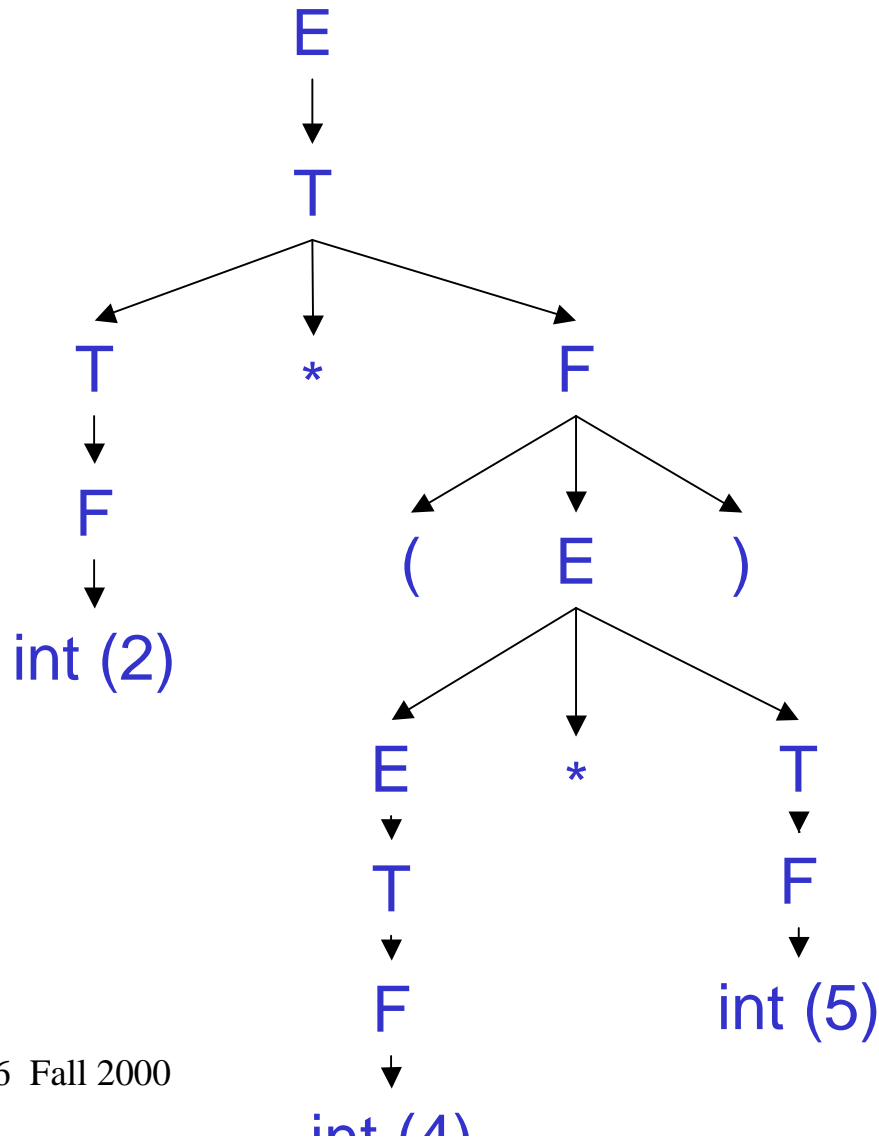
- Examples so far, streams of tokens translated into
 - integer values, or
 - types
- Translating into ASTs is not very different

AST vs Parse Tree

- 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
 - e.g., parentheses, commas, semi-colons
- 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.

Example: $2 * (4 + 5)$

parse tree vs AST



Definitions of AST nodes

```
class ExpNode { }
```

```
class IntLitNode extends ExpNode {  
    public IntLitNode(int val) {...}  
}
```

```
class PlusNode extends ExpNode {  
    public PlusNode( ExpNode e1, ExpNode e2 ) { ... }  
}
```

```
class TimesNode extends ExpNode {  
    public TimesNode( ExpNode e1, ExpNode e2 ) { ... }  
}
```

AST-building translation rules

$E_1 \rightarrow E_2 + T$ $E_1.trans = \text{new PlusNode}(E_2.trans, T.trans)$

$E \rightarrow T$ $E.trans = T.trans$

$T_1 \rightarrow T_2 * F$ $T_1.trans = \text{new TimesNode}(T_2.trans, F.trans)$

$T \rightarrow F$ $T.trans = F.trans$

$F \rightarrow \text{int}$ $F.trans = \text{new IntLitNode}(\text{int.value})$

$F \rightarrow (E)$ $F.trans = E.trans$

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 .

Syntax-Directed Translation and LL Parsing

- not obvious how to do this, since
 - predictive parser builds the parse tree top-down,
 - syntax-directed translation is computed bottom-up.
- could build the parse tree (inefficient!)
- Instead, add a **semantic stack**:
 - holds nonterminals' translations
 - when the parse is finished, the semantic stack will hold just one value:
 - the translation of the root nonterminal (which is the translation of the whole input).

How does semantic stack work?

- How to push/pop onto/off the semantic stack?
 - add **actions** to the grammar rules.
- The action for one rule must:
 - Pop the translations of all rhs nonterminals.
 - Compute and push the translation of the lhs nonterminal.
- Actions are represented by **action numbers**,
 - action numbers become part of the rhs of the grammar rules.
 - action numbers pushed onto the (normal) stack along with the terminal and nonterminal symbols.
 - when an action number is the top-of-stack symbol, it is popped and the action is carried out.

Keep in mind

- action for $X \rightarrow Y_1 Y_2 \dots Y_n$ is pushed onto the (normal) stack when the derivation step $X \rightarrow Y_1 Y_2 \dots Y_n$ is made, but
- the action is performed only after complete derivations for all of the Y 's have been carried out.

Example: Counting Parentheses

$E_1 \rightarrow \varepsilon$

$E_1.\text{trans} = 0$

$\rightarrow (E_2)$

$E_1.\text{trans} = E_2.\text{trans} + 1$

$\rightarrow [E_2]$

$E_1.\text{trans} = E_2.\text{trans}$

Example: Step 1

- replace the translation rules with **translation actions**.
 - Each action must:
 - Pop rhs nonterminals' translations from the semantic stack.
 - Compute and push the lhs nonterminal's translation.
- Here are the translation actions:

$E \rightarrow \varepsilon$	<code>push(0);</code>
$\rightarrow (E)$	<code>exp2Trans = pop();</code> <code>push(exp2Trans + 1);</code>
$\rightarrow [E]$	<code>exp2Trans = pop();</code> <code>push(exp2Trans);</code>

Example: Step 2

each action is represented by a unique action number,
- the action numbers become part of the grammar rules:

$E \rightarrow \epsilon \#1$
 $\rightarrow (E) \#2$
 $\rightarrow [E] \#3$

#1: `push(0);`

#2: `exp2Trans = pop(); push(exp2Trans + 1);`

#3: `exp2Trans = pop(); push(exp2Trans);`

Example: example

input so far	stack	semantic stack	action
(E EOF		pop, push "(E) #2"
((E) #2 EOF		pop, scan
([E) #2 EOF		pop, push "[E]"
([[E]) #2 EOF		pop, scan
([]	E]) #2 EOF		pop, push ϵ #1
([]	#1]) #2 EOF		pop, do action
([])]) #2 EOF	0	pop, scan
([])) #2 EOF	0	pop, scan
([]) EOF	#2 EOF		0 pop, do action
([]) EOF	EOF	1	pop, scan
([]) EOF			empty stack: input accepted! translation of input = 1

What if the rhs has >1 nonterminal?

- pop multiple values from the semantic stack:
 - CFG Rule:
methodBody \rightarrow { varDecls stmts }
 - Translation Rule:
methodBody.trans = varDecls.trans + stmts.trans
 - Translation Action:
stmtsTrans = pop(); declsTrans = pop();
push(stmtsTrans + declsTrans);
 - CFG rule with Action:
methodBody \rightarrow { varDecls stmts } #1
#1: stmtsTrans = pop(); declsTrans = pop();
push(stmtsTrans + declsTrans);

Terminals

- Simplification:
 - we assumed that each rhs contains at most one terminal
- How to push the value of a terminal?
 - a terminal's value is available only when the terminal is the "current token":
- put action before the terminal
 - CFG Rule: $F \rightarrow \text{int}$
 - Translation Rule: $F.\text{trans} = \text{int.value}$
 - Translation Action: $\text{push}(\text{int.value})$
 - CFG rule with Action:
 $F \rightarrow \#1 \text{ int} \ // \ \text{action BEFORE terminal}$
 $\#1: \text{push}(\text{currToken.value})$

Handling non-LL(1) grammars

- Recall that to do LL(1) parsing
 - non-LL(1) grammars must be transformed
 - e.g., left-recursion elimination
 - the resulting grammar does not reflect the underlying structure of the program

$E \rightarrow E + T$

vs.

$E \rightarrow T E'$

$E' \rightarrow \varepsilon \mid + T E'$

- How to define syntax directed translation for such grammars?

The solution is simple!

- Treat actions as grammar symbols
 - define syntax-directed translation on the original grammar:
 - define translation rules
 - convert them to actions that push/pop the semantic stack
 - incorporate the action numbers into the grammar rules
 - then convert the grammar to LL(1)
 - treat action numbers as regular grammar symbols

Example

non-LL(1):

$$\begin{aligned} E &\rightarrow E + T \text{ \#1} \\ &\rightarrow T \\ T &\rightarrow T * F \text{ \#2} \\ &\rightarrow F \end{aligned}$$

#1: TTrans = pop(); ETrans = pop(); push Etrans + TTrans;
#2: FTrans = pop(); TTrans = pop(); push Ttrans * FTrans;

after removing immediate left recursion:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T \text{ \#1 } E' \\ &\rightarrow \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F \text{ \#2 } T' \\ &\rightarrow \epsilon \end{aligned}$$

TEST YOURSELF #3

- For the following grammar, give
 - translation rules + translation actions,
 - a CFG with actions so that the translation of an input expression is the value of the expression.
 - Do not worry that the grammar is not LL(1).
- then convert the grammar (including actions) to LL(1)

$$\begin{aligned} E &\rightarrow E + T \quad | \quad E - T \quad | \quad T \\ T &\rightarrow T * F \quad | \quad T / F \quad | \quad F \\ F &\rightarrow \text{int} \quad | \quad (E) \end{aligned}$$