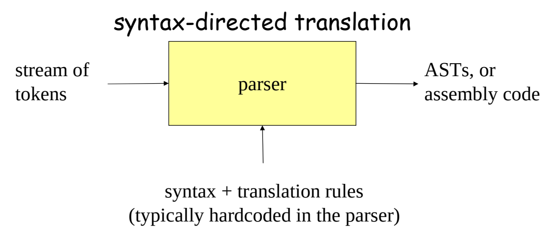
Syntax-Directed Translation

Lecture 10

Motivation: parser as a translator



Outline

- Syntax directed translation: <u>specification</u>
  - translate parse tree to its value, or to an AST
  - typecheck the parse tree
- Syntax-directed translation: <u>implementation</u>
  - during LR parsing
  - during LL parsing

Mechanism of syntax-directed translation

- syntax-directed translation is done by extending the CFG
  - a <u>translation rule</u> is defined for each production

given

X → 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

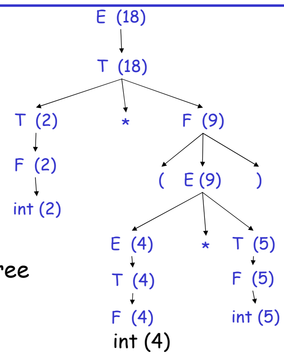
Syntax-directed translation:

the CFG translation rules

E → E + T	E_1.trans = E_2.trans + T.trans
E → T	E.trans = T.trans
T → T * F	T_1.trans = T_2.trans * F.trans
T → F	T.trans = F.trans
F → int	F.trans = int.value
F → (E)	F.trans = E.trans

Example 1 (cont)

Input: 2 * (4 + 5)



Annotated Parse Tree

Example 2: Compute the type of an expression

E -> E + E	if ((E_2.trans == INT) and (E_3.trans == INT)) then E_1.trans = INT else E_1.trans = ERROR
E -> E and E	if ((E_2.trans == BOOL) and (E_3.trans == BOOL)) then E_1.trans = BOOL else E_1.trans = ERROR
E -> E == E	if ((E_2.trans == E_3.trans) and (E_2.trans != ERROR)) then E_1.trans = BOOL else E_1.trans = ERROR
E -> true	E.trans = BOOL
E -> false	E.trans = BOOL
E -> int	E.trans = INT
E -> (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 → 0
  → 1
  → B 0
  → 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.

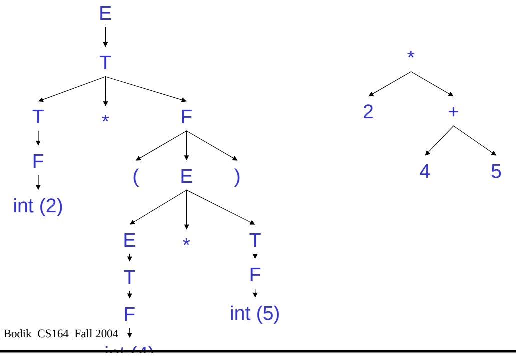
Building Abstract Syntax Trees

- 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



AST-building translation rules

$E_1 \rightarrow E_2 + T$	E_1.trans = new PlusNode(E_2.trans, T.trans)
$E \rightarrow T$	E.trans = T.trans
$T_1 \rightarrow T_2 * F$	T_1.trans = new TimesNode(T_2.trans, F.trans)
$T \rightarrow F$	T.trans = F.trans
$F \rightarrow$ int	F.trans = new IntLitNode(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 LR Parsing

- add semantic stack,
  - parallel to the parsing stack:
    - each symbol (terminal or non-terminal) on the parsing stack stores its value on the semantic stack
  - holds terminals' attributes, and
  - holds nonterminals' translations
  - when the parse is finished, the semantic stack will hold just one value:
    - the translation of the root non-terminal (which is the translation of the whole input).

Semantic actions during parsing

- when shifting
  - push the value of the terminal on the sem. stack
- when reducing
  - pop k values from the sem. stack, where k is the number of symbols on production's RHS
  - push the production's value on the sem. stack

An LR example

Grammar + translation rules:

$E_1 \rightarrow E_2 + (E_3)$	E_1.trans = E_2.trans + E_3.trans
$E_1 \rightarrow$ int	E_1.trans = int.trans

Input:

2 + (3) + (4)

Shift-Reduce Example with evaluations

parsing stack		semantic stack
► int + (int) + (int)$	shift	►

Shift-Reduce Example with evaluations

► int + (int) + (int)$	shift	►
int ► + (int) + (int)$	red. E → int	2 ►

Shift-Reduce Example with evaluations

► int + (int) + (int)$	shift	►
int ► + (int) + (int)$	red. E → int	2 ►
E ► + (int) + (int)$	shift 3 times	2 ►

Shift-Reduce Example with evaluations

► int + (int) + (int)$	shift	►
int ► + (int) + (int)$	red. E → int	2 ►
E ► + (int) + (int)$	shift 3 times	2 ►
E + (int ►) + (int)$	red. E → int	2 '+' '(' 3 ►

Shift-Reduce Example with evaluations

► int + (int) + (int)$	shift	►
int ► + (int) + (int)$	red. E → int	2 ►
E ► + (int) + (int)$	shift 3 times	2 ►
E + (int ►) + (int)$	red. E → int	2 '+' '(' 3 ►
E + (E ►) + (int)$	shift	2 '+' '(' 3 ►

Shift-Reduce Example with evaluations

► int + (int) + (int)$	shift	►
int ► + (int) + (int)$	red. E → int	2 ►
E ► + (int) + (int)$	shift 3 times	2 ►
E + (int ►) + (int)$	red. E → int	2 '+' '(' 3 ►
E + (E ►) + (int)$	shift	2 '+' '(' 3 ►
E + (E) ► + (int)$	red. E → E + (E)	2 '+' '(' 3 ')' ►

Shift-Reduce Example with evaluations

► int + (int) + (int)$	shift	►
int ► + (int) + (int)$	red. E → int	2 ►
E ► + (int) + (int)$	shift 3 times	2 ►
E + (int ►) + (int)$	red. E → int	2 '+' '(' 3 ►
E + (E ►) + (int)$	shift	2 '+' '(' 3 ►
E + (E) ► + (int)$	red. E → E + (E)	2 '+' '(' 3 ')' ►
E ► + (int)$	shift 3 times	5 ►

Shift-Reduce Example with evaluations

► int + (int) + (int)$	shift	►
int ► + (int) + (int)$	red. E → int	2 ►
E ► + (int) + (int)$	shift 3 times	2 ►
E + (int ►) + (int)$	red. E → int	2 '+' '(' 3 ►
E + (E ►) + (int)$	shift	2 '+' '(' 3 ►
E + (E) ► + (int)$	red. E → E + (E)	2 '+' '(' 3 ')' ►
E ► + (int)$	shift 3 times	5 ►
E + (int ►)$	red. E → int	5 '+' '(' 4 ►

Shift-Reduce Example with evaluations

► int + (int) + (int)$	shift	►
int ► + (int) + (int)$	red. E → int	2 ►
E ► + (int) + (int)$	shift 3 times	2 ►
E + (int ►) + (int)$	red. E → int	2 '+' '(' 3 ►
E + (E ►) + (int)$	shift	2 '+' '(' 3 ►
E + (E) ► + (int)$	red. E → E + (E)	2 '+' '(' 3 ')' ►
E ► + (int)$	shift 3 times	5 ►
E + (int ►)$	red. E → int	5 '+' '(' 4 ►
E + (E ►)$	shift	5 '+' '(' 4 ►

Shift-Reduce Example with evaluations

► int + (int) + (int)$	shift	►
int ► + (int) + (int)$	red. E → int	2 ►
E ► + (int) + (int)$	shift 3 times	2 ►
E + (int ►) + (int)$	red. E → int	2 '+' '(' 3 ►
E + (E ►) + (int)$	shift	2 '+' '(' 3 ►
E + (E) ► + (int)$	red. E → E + (E)	2 '+' '(' 3 ')' ►
E ► + (int)$	shift 3 times	5 ►
E + (int ►)$	red. E → int	5 '+' '(' 4 ►
E + (E ►)$	shift	5 '+' '(' 4 ►
E + (E) ► $	red. E → E + (E)	5 '+' '(' 4 ')' ►

Shift-Reduce Example with evaluations

► int + (int) + (int)$	shift	►
int ► + (int) + (int)$	red. E → int	2 ►
E ► + (int) + (int)$	shift 3 times	2 ►
E + (int ►) + (int)$	red. E → int	2 '+' '(' 3 ►
E + (E ►) + (int)$	shift	2 '+' '(' 3 ►
E + (E) ► + (int)$	red. E → E + (E)	2 '+' '(' 3 ')' ►
E ► + (int)$	shift 3 times	5 ►
E + (int ►)$	red. E → int	5 '+' '(' 4 ►
E + (E ►)$	shift	5 '+' '(' 4 ►
E + (E) ► $	red. E → E + (E)	5 '+' '(' 4 ')' ►
E ► $	accept	9 ►

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, the **parsing stack** will also contain actions
  - these actions will be delayed: to be executed when popped from the stack
- To simplify the presentation (and to show you a different style of translation), assume:
  - only non-terminals' values will be placed on the sem. stack

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 $X \rightarrow Y_1 Y_2 \ldots Y_n$ is pushed onto the (normal) stack when the derivation step $X \rightarrow Y_1 Y_2 \ldots Y_n$ 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.trans = 0
$\rightarrow (E_2)$	E_1.trans = E_2.trans + 1
$\rightarrow [E_2]$	E_1.trans = E_2.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:

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

Example: Step 2

each action is represented by a unique action number,

- the action numbers become part of the grammar rules:

$E \rightarrow \varepsilon$ #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		replace E with (E) #2
(	(E) #2 EOF		terminal
([	E) #2 EOF		replace E with [E]
([	[E]) #2 EOF		terminal
([]	E]) #2 EOF		replace E with ε #1
([]	#1]) #2 EOF		pop action, do action
([]	]) #2 EOF	0	terminal
([])	) #2 EOF	0	terminal
([]) EOF	#2 EOF	0	pop action, do action
([]) EOF	EOF	1	terminal
([]) 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 → { varDecls stmts }
  - Translation Rule:
    methodBody.trans = varDecls.trans + stmts.trans
  - Translation Action:
    stmtsTrans = pop(); declsTrans = pop();
    push(stmtsTrans + declsTrans);
  - CFG rule with Action:
    methodBody → { 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 → int
  - Translation Rule: F.trans = int.value
  - Translation Action: push(int.value)
  - CFG rule with Action:
    F → #1 int // action BEFORE terminal
    #1: push(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 → E + T
    vs.
    E → T E'
    E' → ε | + 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):
E → E + T #1 | T
T → T * F #2 | F
F → #3 int

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

after removing immediate left recursion:

E → T E'
E' → + T #1 E' | ε
F → #3 int

T → F T'
T' → * F #2 T' | ε

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)

E → E + T | E – T | T
T → T * F | T / F | F
F → int | (E)