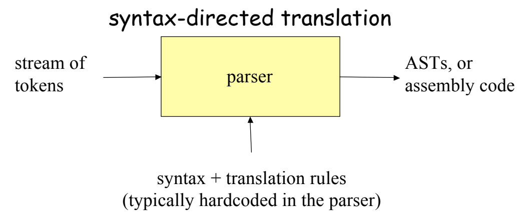
Syntax-Directed Translation

Lecture 14

(adapted from slides by R. Bodik)

Motivation: parser as a translator



Outline

- Syntax-directed translation: *specification*
  - translate parse tree to its value, or to an AST
  - type check the parse tree
- Syntax-directed translation: *implementation*
  - during LR parsing
  - during recursive-descent parsing

Mechanism of syntax-directed translation

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

given

X → d A B c

the translation of X is defined recursively using

- 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: Arithmetic expression to value

Syntax-directed 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: Bison/Yacc Notation

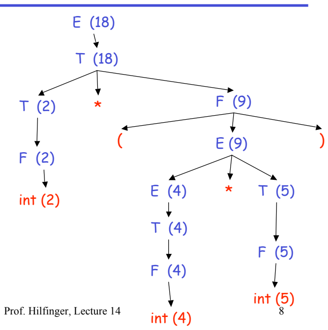
E : E + T	{ $$ = $1 + $3; }
T : T * F	{ $$ = $1 * $3; }
F : int	{ $$ = $1; }
F : '(' E ')'	{ $$ = $2; }

- ***KEY:*** $$: Semantic value of left-hand side

  $n : Semantic value of n^{th} symbol on right-hand side

Example 1 (cont): Annotated Parse Tree

Input: 2 * (4 + 5)

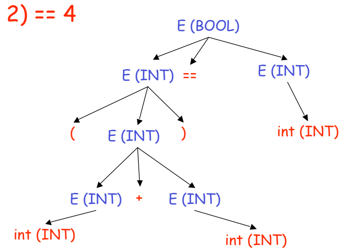


Example 2: Compute the type of an expression

E -> E + E	if $1 == INT and $3 == INT: $$ = INT else: $$ = ERROR
E -> E and E	if $1 == BOOL and $3 == BOOL: $$ = BOOL else: $$ = ERROR
E -> E == E	if $1 == $3 and $2 != ERROR: $$ = BOOL else: $$ = ERROR
E -> true	$$ = BOOL
E -> false	$$ = BOOL
E -> int	$$ = INT
E -> (E)	$$ = $2

Example 2 (cont)

- Input: (2 + 2) == 4



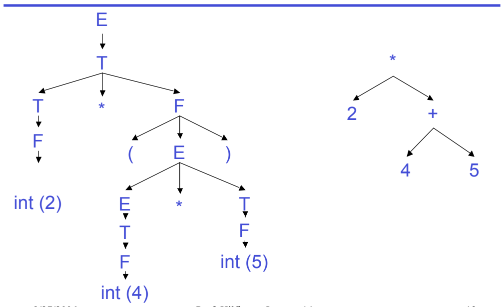
Building Abstract Syntax Trees

- 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 omitted
    - 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.

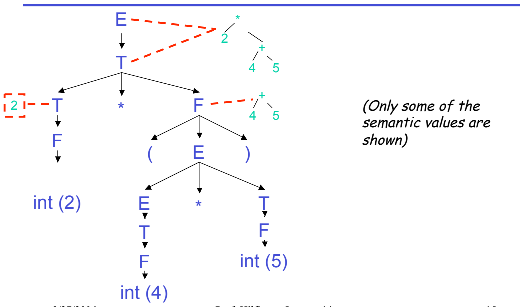
Example: 2 * (4 + 5) Parse tree *vs.* AST



AST-building translation rules

E → E + T	$$ = new PlusNode($1, $3)
E → T	$$ = $1
T → T * F	$$ = new TimesNode($1, $3)
T → F	$$ = $1
F → int	$$ = new IntLitNode($1)
F → (E)	$$ = $2

Example: 2 * (4 + 5): Steps in Creating AST



(Only some of the semantic values are shown)

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

An LR example

Grammar + translation rules:

E → E + (E)	$$ = $1 + $4
E → int	$$ = $1

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 |

Taking Advantage of Derivation Order

- So far, rules have been *functional;* no side effects except to define (once) value of LHS.
- LR parsing produces reverse rightmost derivation.
- Can use the ordering to do control semantic actions with side effects.

Example of Actions with Side Effects

E → E + T	print "+",	*We know that reduction taken after all the reductions that form the nonterminals on right-hand side.*
E → T	pass	
T → T * F	print "*",	*So what does this print for 3+4*(7+1)?*
T → F	pass	
F → int	print $1,	3 4 7 1 + * +
F → (E)	pass	

Recursive-Descent Translation

- Translating with recursive descent is also easy.
- The semantic values (what Bison calls $$, $1, etc.), become *return values of the parsing functions*
- We'll also assume that the lexer has a way to return lexical values (e.g., the scan function introduced in Lecture 9 might do so).

Example of Recursive-Descent Translation

- E → T | E+T   T → P | T*P   P → int | '(' E ')'

```
def E():
   T ()
   while next() == "+":
       scan("+"); T()

def T():
   P()
   while next() == "*":
       scan("*"); P()
```

```
def P():
   if next()==int:
       scan (int)
   elif next()=="(":
       scan("(")
       E()
       scan(")")
   else: ERROR()
```

(we've cheated and used loops; see end of lecture 9)

Example contd.: Add Semantic Values

- E → T | E+T   T → P | T*P   P → int | '(' E ')'

```
def E():
   v = T ()
   while next() == "+":
       scan("+"); v += T()
   return v
def T():
   v = P()
   while next() == "*":
       scan("*"); v *= P()
   return v
```

```
def P():
   if next()==int:
       v = scan (int)
   elif next()=="(":
       scan("(")
       v = E()
       scan(")")
   else: ERROR()
   return v
```


Table-Driven LL(1)

- We can automate all this, and add to the LL(1) parser method from Lecture 9.
- However, this gets a little involved, and I'm not sure it's worth it.
- (That is, let's leave it to the LL(1) parser generators for now!)