Semantic Analysis

Read: Scott, Chapter 4.1-4.3

Announcements

- HW 1 grades are up
- Quiz 1,2,3 grades up
  - We will release answers in review lecture
- Rainbow grades
  - Please check if your grade shows up correctly
- Exam 1 a week from today --- Oct 11th
  - Links to practice problems on Submitty forum
- HW3 is posted
  - Due in 10 days

Lecture Outline

- Quiz 4
- Attribute grammars
  - Attributes and rules
  - Synthesized and inherited attributes
  - S-attributed grammars
  - L-attributed grammars

- Attribute evaluation

Attribute Grammars: Foundation for Static Semantic Analysis

- Attribute Grammars: generalization of Context-Free Grammars
  - Associate meaning with parse trees
  - Attributes
    - Each grammar symbol has one or more values called attributes associated with it. Each parse tree node has its own instances of those attributes; attribute value carries the "meaning" of the parse tree rooted at node
  - Semantic rules
    - Each grammar production has associated rule, which may refer to and compute the values of attributes

Example: Attribute Grammar to Compute Value of Expression (denote grammar by AG1)

$S \rightarrow E \qquad E \rightarrow E + T \mid T \qquad T \rightarrow T * F \mid F \qquad F \rightarrow \mathbf{num}$

Production	Semantic Rule
$S \rightarrow E$	print($E.val$)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow \mathbf{num}$	$F.val := \mathbf{num}.val$

val: Attributes

Another Grammar

E stands for *expr*
T stands for *term*
TT stands for *term_tail*

- Now, the right-recursive LL(1) grammar:

$E \rightarrow T\ TT$
$TT \rightarrow$ **-** $T\ TT$
$TT \rightarrow$ **ε**
$T \rightarrow$ **num**

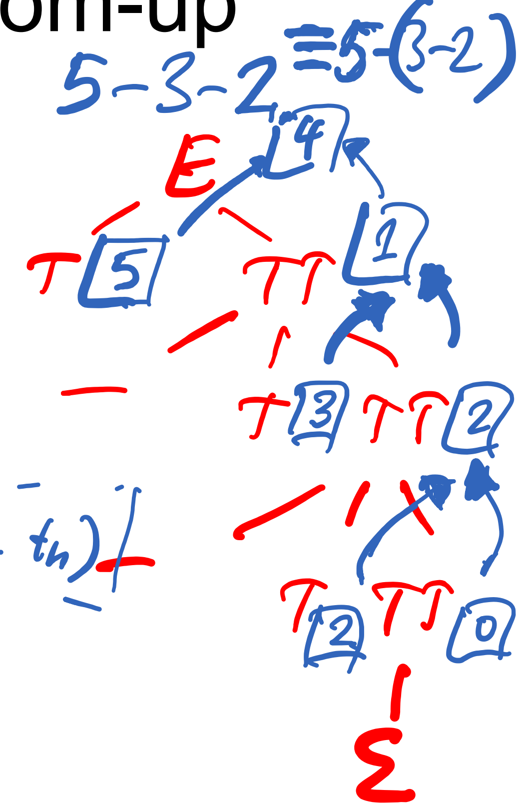
- Goal: construct an attribute grammar that computes the value of an expression
  - Values must be computed "normally", i.e., **5-3-2** must be evaluated as **(5-3)-2**, not as **5-(3-2)**

0

4 ✗ NO

Question

- What happens if we wrote a "bottom-up attribute flow" grammar?



$E \rightarrow T\ TT$	$E.val = T.val - TT.val$
$TT \rightarrow -\ T\ TT_1$	$TT.val = T.val - TT_1.val$
$TT \rightarrow \varepsilon$	$TT.val = 0$
$T \rightarrow \mathbf{num}$	$T.val = \mathbf{num}.val$

<u>A hack:</u>

$E \rightarrow T\ TT$	$E.val = T.val - TT.val$
$TT \rightarrow -\ T\ TT_1$	$TT.val = T.val + TT_1.val$
$TT \rightarrow \varepsilon$	$TT.val = 0$
$T \rightarrow \mathbf{num}$	$T.val = \mathbf{num}.val$

Unfortunately, this won't work if we add $TT \rightarrow +\ T\ TT_1$

Attribute Grammar to Compute Value of Expressions (denote by AG3)

$E \to T\ TT \qquad TT \to -T\ TT \mid +T\ TT \mid \boldsymbol{\varepsilon} \qquad T \to$ **num**

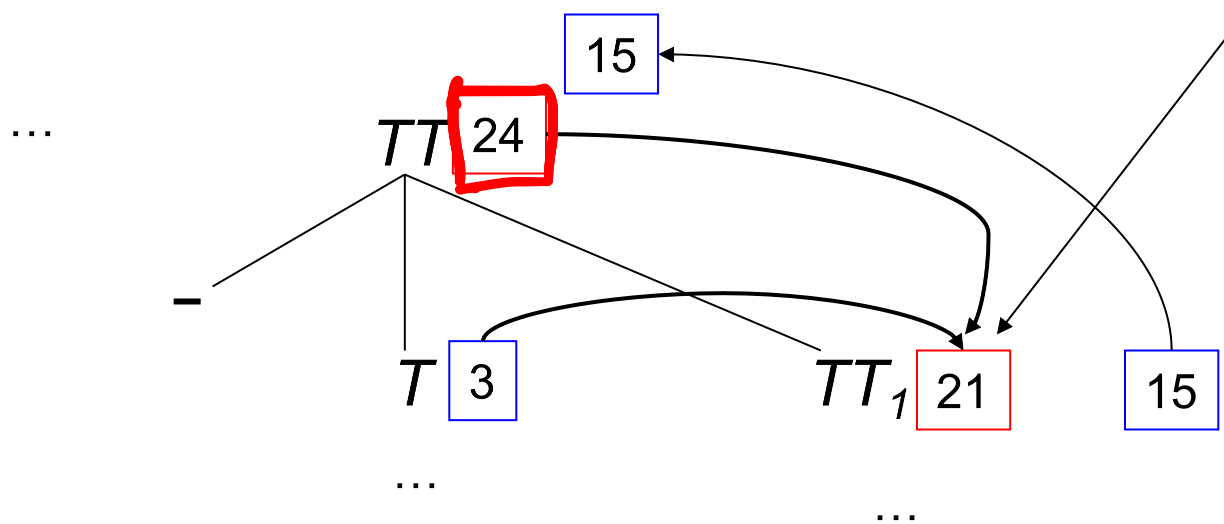
5-3-2 TT.sub=5 TT_1.sub=2 (5-3)

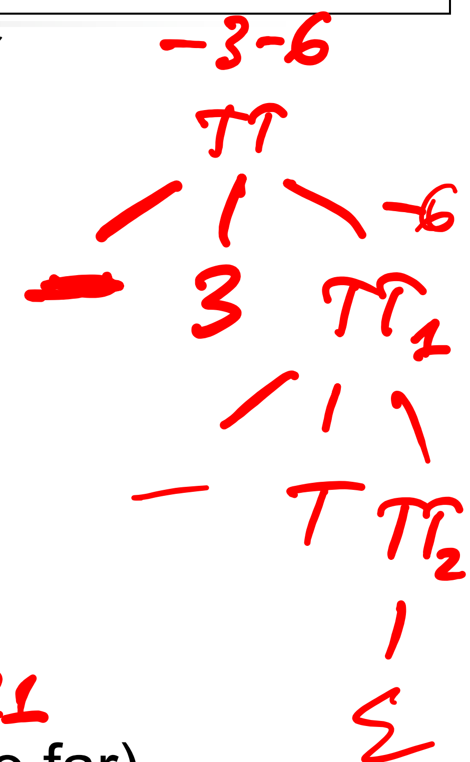
Production	Semantic Rules	
$E \to T\ TT$	(1) $TT.sub := T.val$	(2) $E.val := TT.val$
$TT \to -\ T\ TT_1$	(1) $TT_1.sub := TT.sub - T.val$	(2) $TT.val := TT_1.val$
$TT \to +\ T\ TT_1$	(1) $TT_1.sub := TT.sub + T.val$	(2) $TT.val := TT_1.val$
$TT \to \boldsymbol{\varepsilon}$	(1) $TT.val := TT.sub$	
$T \to$ **num**	(1) $T.val :=$ **num**.val	(provided by scanner)

Attributes flow from parent to node, and from "siblings" to node!

Attribute Flow

Attribute TT_1.*sub*: computed based on parent *TT* and sibling *T*: *TT*.*sub* – *T*.*val*





E.g., `25 – 1 – 3 – 6`

TT holds subtotal **`24`** (for `25 – 1`, computed so far)

T holds value **`3`** (i.e., the value of next term)

TT_1 gets subtotal **`21`** (for `25 – 1 – 3`)

Passed down the tree of TT_1 to next *TT* on chain

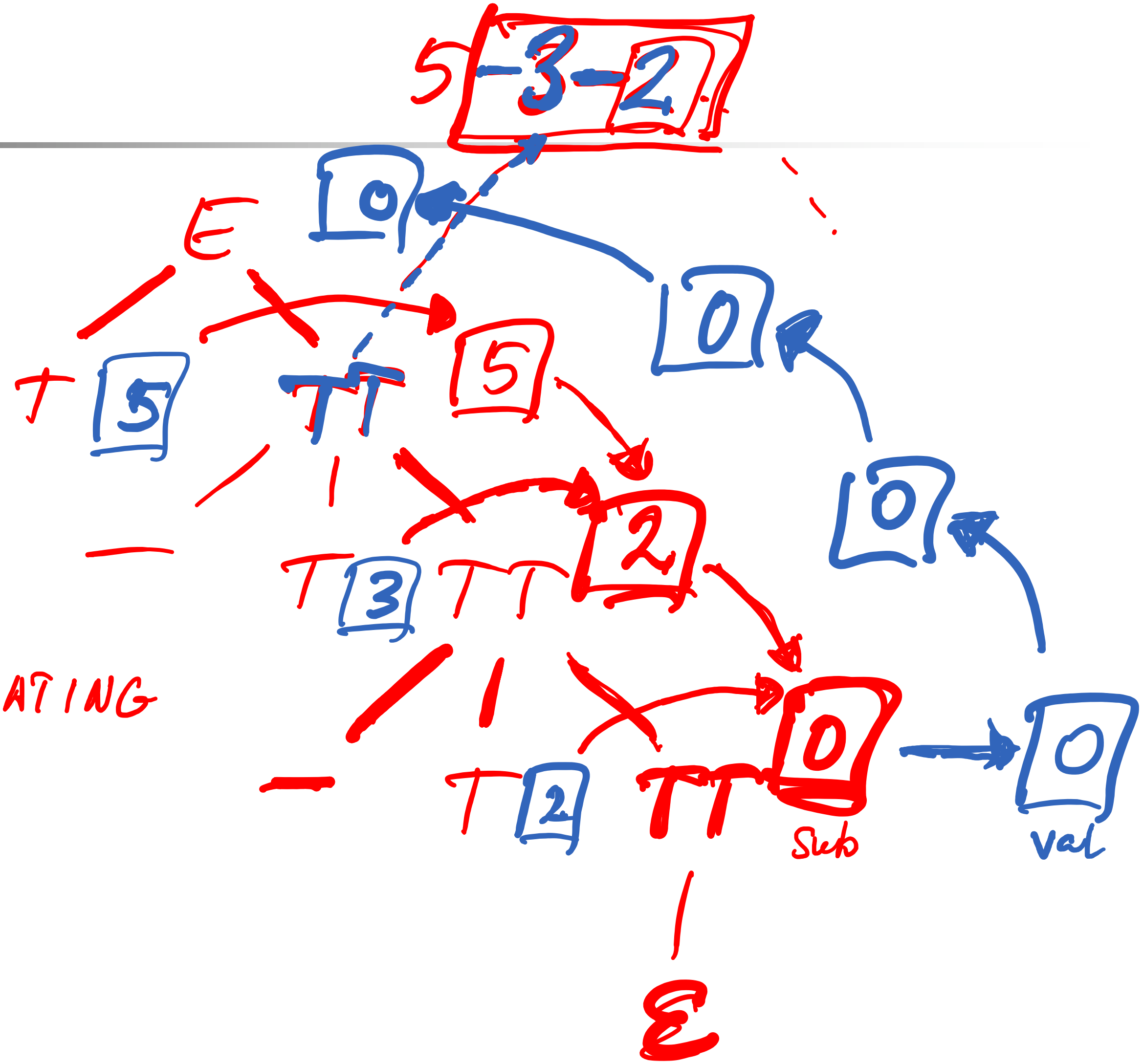
Eventually, we hit $TT \to \varepsilon$ and value gets subtotal **`15`**

Value **`15`** is passed back up

Example

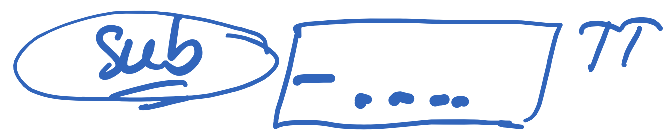
PARSING AND EVALUATING

5-3-2



Attribute Flow

- Attribute *.val* carries the total value
- Attribute *.sub* is the subtotal carried from left



- Rules for nonterminals *E, T* do not perform computation
  - No need for *.sub* attribute
- *T.val* attribute flows to the right
  - In $E \rightarrow T\ TT$: *val* of *T* is passed to sibling *TT*
  - In $TT \rightarrow -T\ TT_1$: *val* of *T* is passed to sibling TT_1

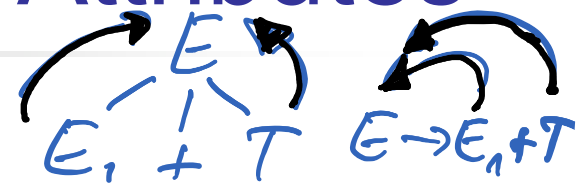
Attribute Flow

- Rules for nonterminal *TT* do perform computation
  - *TT* needs to carry subtotal in *.sub*
    - E.g., in $TT \rightarrow - T\ TT_1$ the subtotal of TT_1 is computed by subtracting the value of *T* from the subtotal of *TT*
- *TT.val* attribute flows up
  - In $E \rightarrow T\ TT$: *val* of *TT* is passed to parent *E*
  - In $TT \rightarrow -T\ TT_1$: *val* of TT_1 is passed to parent *TT*

Lecture Outline

- Quiz 4
- Attribute grammars
  - Attributes and rules
  - Synthesized and inherited attributes
  - S-attributed grammars
  - L-attributed grammars
- Attribute evaluation

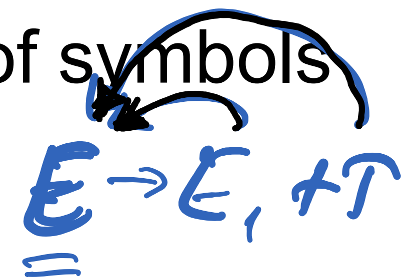
Synthesized and Inherited Attributes



- Synthesized attributes
  - Attribute value computed from attributes of descendants in parse tree or attributes of self
  - E.g., attributes *val* in AG1, *val* in AG3
  - E.g., attributes *nptr* in AG2 (Constructs the AST)
- Inherited attributes
  - Attribute value computed from attributes of parent in tree, or attributes of siblings in tree
  - E.g., attributes *sub* in AG3
    - In order to compute value "normally" we needed to pass sub down the tree (sub is inherited attribute).

S-attributed Grammars

- An attribute grammar for which all attributes are synthesized is said to be S-attributed
  - "Arguments" of rules are attributes of symbols from the production right-hand-side
    - I.e., attributes of children in parse tree
  - "Result" is placed in attribute of the symbol on the left-hand-side of the production
    - I.e., computes attribute of parent in parse tree
  - I.e., attribute values depend only on descendants in tree. They do not depend on parents or siblings in tree!



Questions

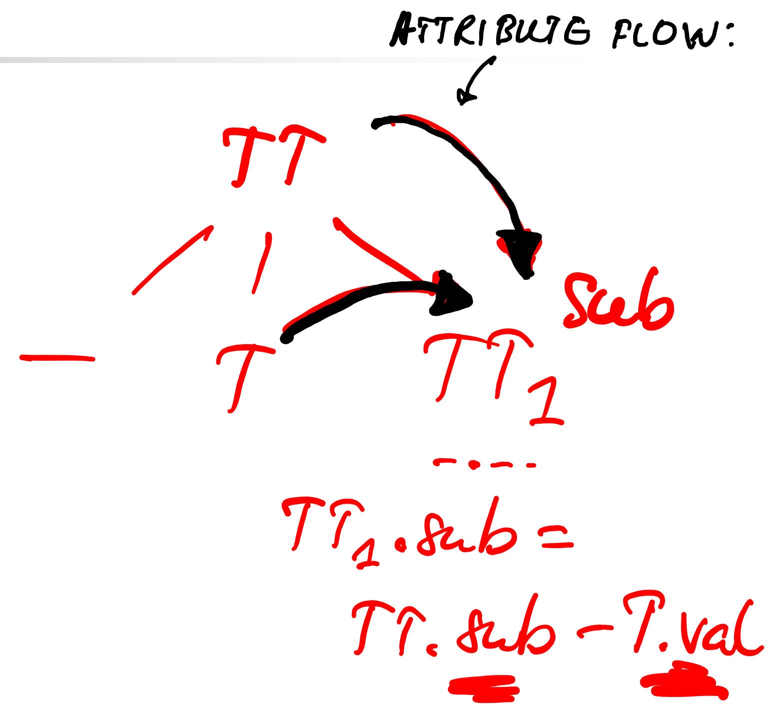
- Can you give examples of S-attributed grammars?
  - Answer: AG1 and AG2
- How can we evaluate S-attributed grammars?
  - I.e., can we evaluate the attributes during a bottom-up parse?
  - Answer: Yes

L-attributed Grammar

- An attribute grammar is L-attributed if each inherited attribute of X_j on the right-hand-side of $A \rightarrow X_1 X_2 \ldots X_{j-1} X_j \ldots X_n$ depends only on
  - (1) the attributes of symbols to the left of X_j: $X_1, X_2, \ldots, X_{j-1}$
  - (2) the inherited attributes of A

Questions

- Can you give examples of L-attributed grammars?
  - Answer: AG3
- How can we evaluate L-attributed grammars?
  - Answer: in a top-down (recursive descent) parse



Attribute Evaluation

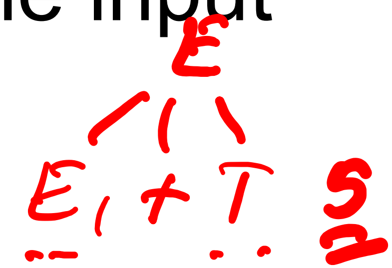
- S-attributed grammars
  - A very special case of attribute grammars
  - Most important case in practice
  - Can be evaluated on-the-fly during a bottom-up (LR) parse
- L-attributed grammars
  - A proper superset of S-attributed grammars
    - Each S-attributed grammar is also L-attributed because restriction applies only to inherited attributes
  - Can be evaluated on-the-fly during a top-down (LL) parse

Bottom-up Parsing

- Also called LR parsing
- LR parsers work with LR(k) grammars
  - L stands for "left-to-right" scan of input
  - R stands for "rightmost" derivation
  - k stands for "need k tokens of lookahead"
- We are interested in LR(0) and LR(1) and variants in between
  LALR(1)
- LR parsing is better than LL parsing!
  - Accepts larger class of languages
  - Just as efficient!

Main Idea

- Stack ← Input
- Stack: holds the part of the input seen so far
  - A string of both terminals and nonterminals
- Input: holds the remaining part of the input
  - A string of terminals
- Parser performs two actions many
  - Reduce: parser pops a "suitable" production right-hand-side off top of stack, and pushes production's left-hand-side on the stack
  - Shift: parser pushes next terminal from the input on top of the stack



OTHER TWO ACTIONS: ERROR, ACCEPT

Example

- Recall the grammar

$$expr \rightarrow expr \; \mathbf{+} \; term \mid term$$
$$term \rightarrow term \; \mathbf{*} \; \mathtt{num} \mid \mathtt{num}$$

  - This is not LL(1) because it is left recursive
  - LR parsers can handle left recursion!

- Consider string

  `num + num * num`

num + num*num

Stack	Input	Action
EMPTY STACK	num+num*num	shift num
num	+num*num	reduce by *term*→ num
term	+num*num	reduce by *expr*→ *term*
expr	+num*num	shift +
expr+	num*num	shift num
expr+num	*num	reduce by *term* → num

expr → *expr* + *term* | *term*
term → *term* * num | num

num + num*num

expr * ~~num~~

Stack	Input	Action
expr+*term*	***num**	shift *
expr+*term**	**num**	shift **num**
expr+*term**num**		reduce by *term*→*term* ***num**
expr+*term*		reduce by *expr*→*expr*+*term*
expr	EMPTY INPUT	ACCEPT, SUCCESS

expr → *expr* **+** *term* | *term*
term → *term* ***** **num** | **num**

num + num*num

Sequence of reductions performed by parser

num+num*num

term+**num*num**

expr+**num*num**

expr+*term****num**

expr+*term*

expr

- A rightmost derivation in reverse
- The stack (e.g., *expr*) concatenated with remaining input (e.g., **+num*num**) gives a sentential form (*expr*+**num*num**) in the rightmost derivation

$expr \rightarrow expr$ **+** $term \mid term$

$term \rightarrow term$ ***** **num** | **num**

Evaluation 5 + 3*2

Stack	Input	Action
	num+num*num	shift **num**
num	**+num*num**	reduce by *term*→ **num** term.val := num.val
term [5]	**+num*num**	reduce by *expr*→ *term* expr.val := term.val
expr [5]	**+num*num**	shift **+**
expr [5] **+**	**num*num**	shift **num**
expr [5] **+num**	***num**	reduce by *term* → **num**

Evaluation 5 + 3*2

Stack	Input	Action
expr+*term*	*****num**	shift *****
expr+*term**	**num**	shift **num**
expr+*term****num**		reduce by *term*→*term*$_1$***num**
expr+*term*		reduce by *expr*→*expr*+*term*
expr		ACCEPT, SUCCESS

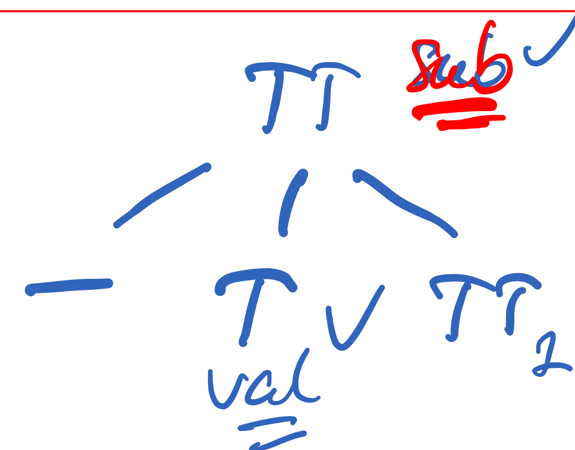
Question

- An attribute grammar is L-attributed if each inherited attribute of X_j on the right-hand-side of $A \rightarrow X_1 X_2 \ldots X_{j-1} X_j \ldots X_n$ depends only on
  - (1) the attributes of symbols to the left of X_j: X_1, $X_2, \ldots, X_{j-1}$
  - (2) the inherited attributes of A
- Why the restriction on siblings and kinds of attributes of parent? Why not allow dependence on siblings to the right of X_j, *e.g.*, X_{j+1}, etc.?

(Top-down) Recursive Descent

$S \rightarrow E$ **$$**

$E \rightarrow T\ TT$ $\qquad TT \rightarrow$ **-** $T\ TT$ | **+** $T\ TT$ | **ε** $\qquad T \rightarrow$ **num**




```
num S()
    case lookahead() of
        num:   val = E(); match($$); return val
        otherwise PARSE_ERROR
num E()
    case lookahead() of
        num:   sub = T(); val = TT(sub); return val
        otherwise PARSE_ERROR

num TT(num sub)
    case lookahead() of
        –  :  match( '–' ); Tval = T(); val = TT(sub – Tval); return val
        +  :  match( '+' ); Tval = T(); val = TT(sub – Tval); return val
        $$ : val = sub; return val
        otherwise: PARSE_ERROR
```

sub is fully evaluated.
Tval is fully evaluated.

The End