Announcements

- HW1 grades will be up later today
- Quiz 1,2,3 grades up, answers in forum
- Rainbow grades
  - Please check if your grade shows up correctly
- Exam 1 a week from today
  - Links to practice problems on Submitty forum
- HW3 is posted

- Quiz 4 at the end of class

Semantic Analysis

Read: Scott, Chapter 4.1-4.3

Lecture Outline

- *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 grammar is a generalization of the Context-Free Grammar
  - Associate meaning with parse trees
  - Attributes E val E_1 val E_2 val
    - 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

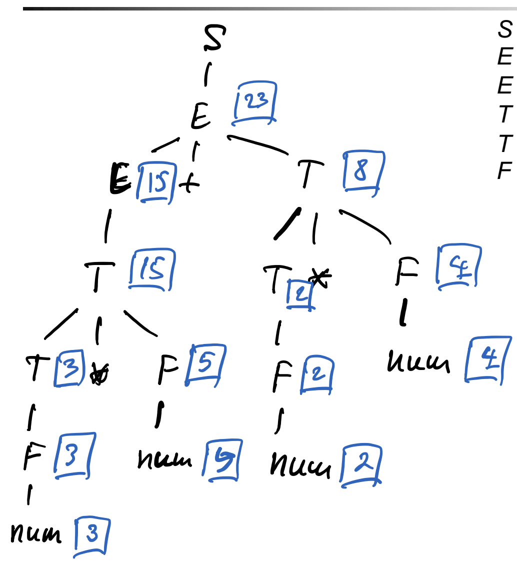
Example: Attribute Grammar to Compute Value of Expression (denote grammar by AG1)

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

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

val: Attributes

Example: Decorated parse tree for input **3*5+2*4**



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

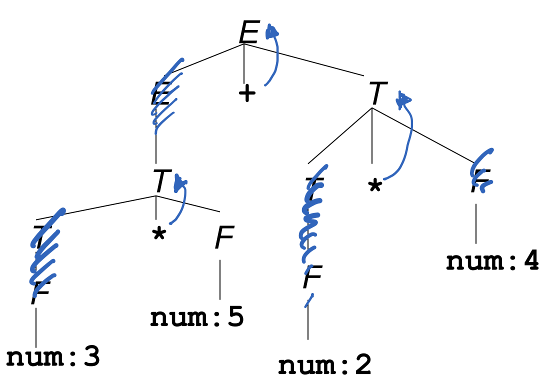
Building an Abstract Syntax Tree (AST)

- An AST is an abbreviated parse tree
  - Operators and keywords do not appear as leaves, but at the interior node that would have been their parent
  - Chains of single productions are collapsed

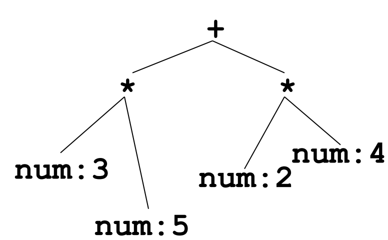
- Compilers work with ASTs

Building ASTs for Expressions

Parse tree for **`3*5+2*4`**



Abstract syntax tree (AST)



How do we construct syntax trees for expressions?

Attribute Grammar to build AST for Expression (denote by AG2)

- An attribute grammar:

Attribute "nodepointer" points to AST

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.nptr$:= **mknode**(+, $E_1.nptr$, $T.nptr$)
$E \rightarrow T$	$E.nptr$:= $T.nptr$
$T \rightarrow T_1 * F$	$T.nptr$:= **mknode**(*, $T_1.nptr$, $F.nptr$)
$T \rightarrow F$	$T.nptr$:= $F.nptr$
$F \rightarrow$ **num**	$F.nptr$:= **mkleaf**(**num**, **num**.*val*)

mknode(op,left,right) creates an operator node with label ***op***, and two fields containing pointers ***left,*** to left operand and ***right,*** to right operand

mkleaf*(num,num.*val*)** creates a leaf node with label ***num, and a field containing the value of the number

9

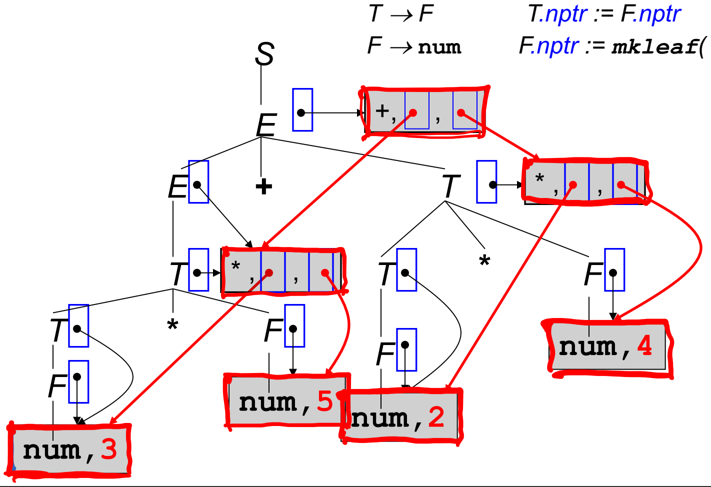
9

Constructing ASTs for Expressions

Input:

3 * 5 + 2 * 4

$E \rightarrow E_1 + T$	$E.nptr$:= **mknode**('+', $E_1.nptr$, $T.nptr$)
$E \rightarrow T$	$E.nptr$:= $T.nptr$
$T \rightarrow T_1 * F$	$T.nptr$:= **mknode**('*', $T_1.nptr$, $F.nptr$)
$T \rightarrow F$	$T.nptr$:= $F.nptr$
$F \rightarrow$ **num**	$F.nptr$:= **mkleaf**('**num**', **num**.*val*)



10

Exercise

- We know that the language $L = \mathbf{a^n b^n c^n}$ is **not** context free. It can be captured, however, with an attribute grammar. Give an underlying CFG and a set of attribute rules that associate an attribute **ok** with the root S of each parse tree, such that S.**ok** is true if and only if the string corresponding to the fringe of the tree is in L.

$L' = a^n b^m c^p$, $n \geq 0, m \geq 0, p \geq 0$

$S \rightarrow ABC$

Exercise

We associate "count" attribute to A, B and C. Counts number of a's, b's and c's, respectively.

$S \rightarrow ABC$ → S.ok = True if A.count == B.count == C.count else False

$A \rightarrow A_1 a$ → $A.count = A_1.count + 1$

$A \rightarrow \varepsilon$ → $A.count = 0$

$B \rightarrow B_1 b$ $B.count = B_1.count + 1$

$B \rightarrow \varepsilon$ $B.count = 0$

$C \rightarrow C_1 c$ $C.count = C_1.count + 1$

$C \rightarrow \varepsilon$ $C.count = 0$

Exercise

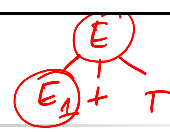
- Consider the expression grammar

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow \mathbf{num} \mid (E)$

Give attribute rules to accumulate into root a count of the depth to which parentheses are nested in the expression. E.g., **((1+2)*3+4)*5+6** has a count of 2.

Exercise

depth



$E \rightarrow E_1 + T$ ----> $E.depth = max(E_1.depth, T.depth)$

$E \rightarrow T$ ----> $E.depth = T.depth$

$T \rightarrow T_1 * F$ ----> $T.depth = max(T_1.depth, F.depth)$

$T \rightarrow F$ ----> $T.depth = F.depth$

$F \rightarrow num$ ----> $F.depth = 0$

$F \rightarrow (E)$ ----> $F.depth = E.depth + 1$

$E \rightarrow E_1 + E_2$

Another Grammar

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

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

same as:

$E \rightarrow T\ TT$	expr → term term_tail
$TT \rightarrow -\ T\ TT$	term_tail → - term term_tail
$TT \rightarrow \varepsilon$	term_tail → ε
$T \rightarrow$ **num**	term → num

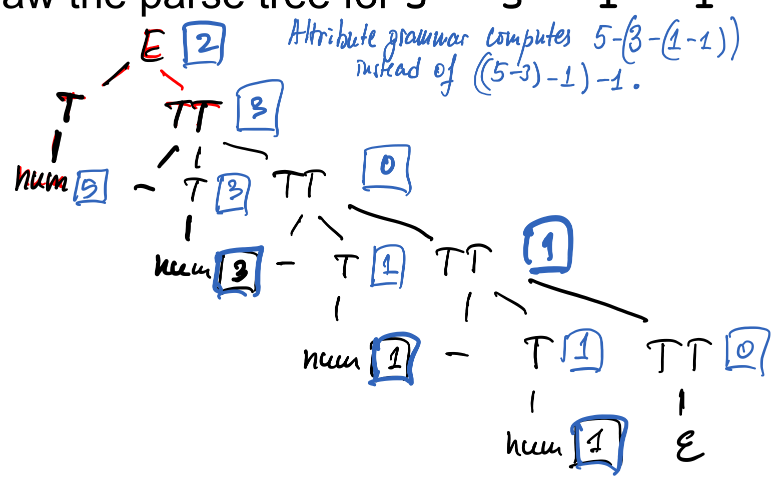
- Goal: construct an attribute grammar that computes the value of an expression
  - Values must be computed "normally", i.e., **`5-3-2`** must be evaluated as **`(5-3)-2`**, not as **`5-(3-2)`**

Exercise

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

- Draw the parse tree for **`5 - 3 - 1 - 1`**

Attribute grammar computes 5-(3-(1-1)) instead of ((5-3)-1)-1.



Question

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

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

A hack: We can "fix" above grammar because $t - t_1 - t_2 - t_3 \ldots - t_k = t - (t_1 + t_2 + t_3 + \ldots t_k)$

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

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

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

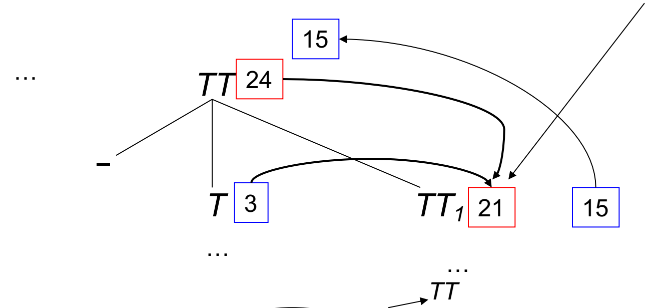
$E \to T\ TT \qquad TT \to -T\ TT \mid +T\ TT \mid \varepsilon \qquad T \to \mathbf{num}$

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) $TT.val := TT_1.val$
$TT \to \varepsilon$	(1) $TT.val := TT.sub$	
$T \to \mathbf{num}$	(1) $T.val := \mathbf{num}.val$	(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

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

- *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
- 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 <u>inherited</u> attribute of X_j on the right-hand-side of $A \to X_1 X_2 \ldots X_{j-1} X_j \ldots X_n$ depends only on
  - (1) the attributes of symbols <u>to the left of</u> X_j: X_1, $X_2, \ldots, X_{j-1}$
  - (2) the <u>inherited</u> 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

29

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
- LR parsing is better than LL parsing!
  - Accepts larger class of languages
  - Just as efficient!

30

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 one of two main actions
  - 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

Example

- Recall the grammar

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

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		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

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

Evaluation 5 + 3*2

Stack	Input	Action
	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

Evaluation 5 + 3*2

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		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.?

Predictive Recursive Descent

$S \rightarrow E$ **$$**
$E \rightarrow T\ TT$ $\quad TT \rightarrow -\ T\ \ TT \mid +\ T\ \ TT \mid \varepsilon$ $\quad T \rightarrow$ **num**

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

int TT(int 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
```


The End