Semantic Analysis

Read: Scott, Chapter 4.1-4.3

Lecture Outline

- Syntax vs. static semantics
- Static semantics vs. dynamic semantics

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

Static Semantics

- Earlier we considered syntax analysis
  - Informally, syntax deals with the form of programming language constructs
- We now look at static semantic analysis
  - Semantics deals with the meaning of programming language constructs
- The distinction between the two is fuzzy
  - In practice, anything that is not expressed in terms of certain CFG (LALR(1), in particular) is considered semantics

Static Semantics vs. Dynamic Semantics

- Static semantic analysis (compile-time)
  - Informally, reasons about program properties statically, before program execution
    - E.g., determine static types of expressions, detect certain errors
- Dynamic semantic analysis (run-time)
  - Reasons about program properties dynamically, during program execution
    - E.g., could expression `a[i]` index out of array bounds, etc.?

The Role of Semantic Analysis

- Detect errors in programs!
- Static semantic analysis
  - Detect as many errors as possible early, before execution
    - Type inference and type checking
- Dynamic semantic analysis
  - Detect errors by performing checks during execution
    - Again, detect errors as early as possible. E.g., flagging an array-out-of-bounds at assignment `a[i] = …` is useful
    - Tradeoff: dynamic checks slow program execution
- Languages differ greatly in the amount of static semantic analysis and dynamic semantic analysis they perform

Examples of Static Semantic Errors

- Type mismatch:
  - `x = y+z+w`: type of left-hand-side does not "match" type of right-hand-side
  - `A a; … ; a.m()`: `m()` cannot be invoked on a variable of type `A`
- Definite assignment check in Java: a local variable must be assigned before it is used

Examples of Dynamic Semantic Errors

- Null pointer dereference:
  - `a.m()` in Java, and `a` is null (i.e., uninitialized reference)
  - What happens?
- Array-index-out-of-bounds:
  - `a[i], i` goes beyond the bounds of `a`
  - What happens in C++? What happens in Java?
- Casting an object to a type of which it is not an instance
  - C++? Java?
- And more…

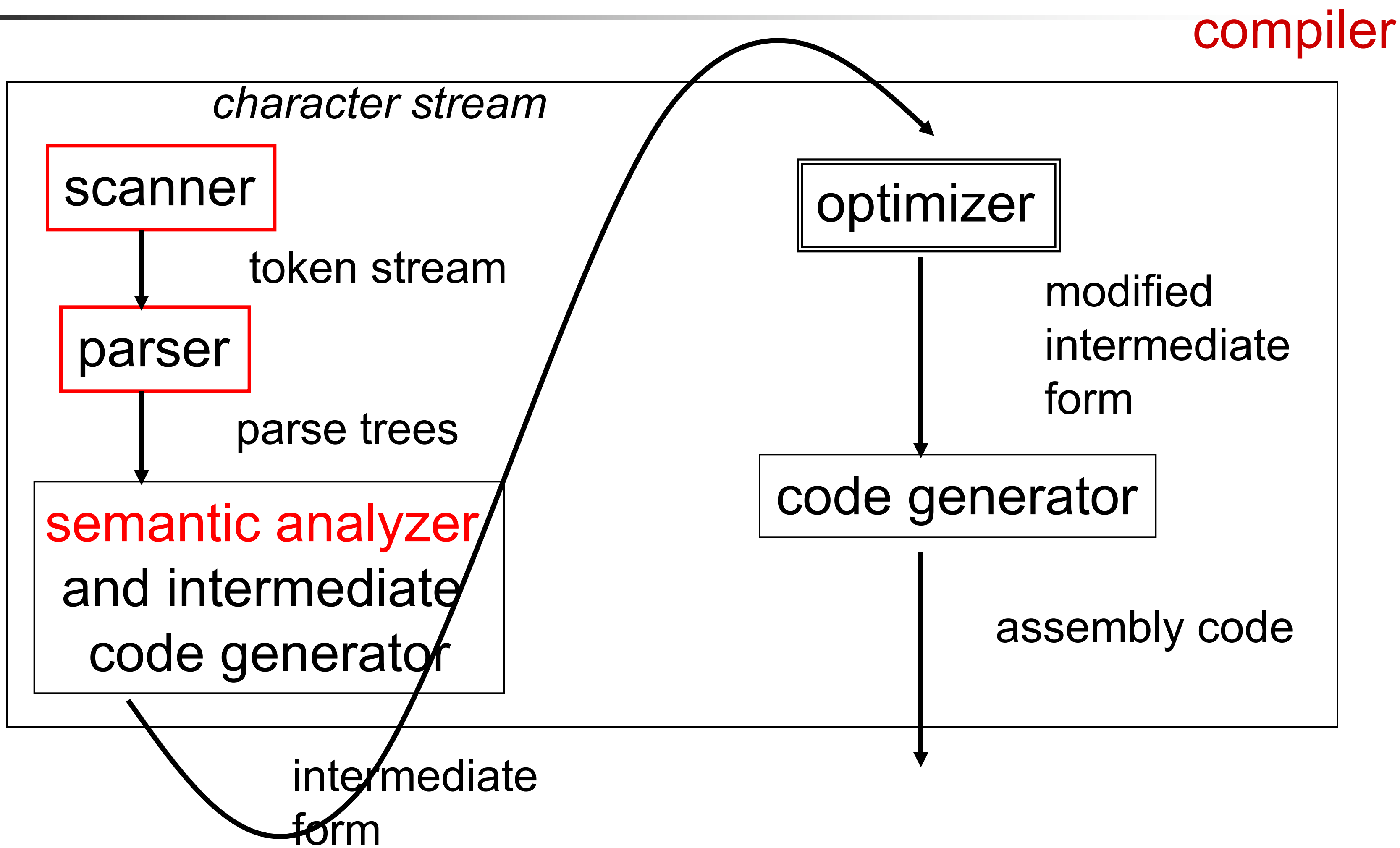
Static Semantics vs. Dynamic Semantics

- Again, distinction between the two is fuzzy
- For some programs, the compiler **can** predict run-time behavior by using static analysis
  - E.g., there is no need for a nullness check:

```
x = new X();
x.m(); // x is non-null
```

- In general, the compiler cannot predict run-time behavior
  - Static analysis is limited by the halting problem

Semantic Analyzer



Semantic analyzer performs static semantic analysis on parse trees and ASTs.
Optimizer performs static semantic analysis on intermediate 3-address code.

Lecture Outline

- Syntax vs. static semantics
- Static semantics vs. dynamic semantics

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

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)

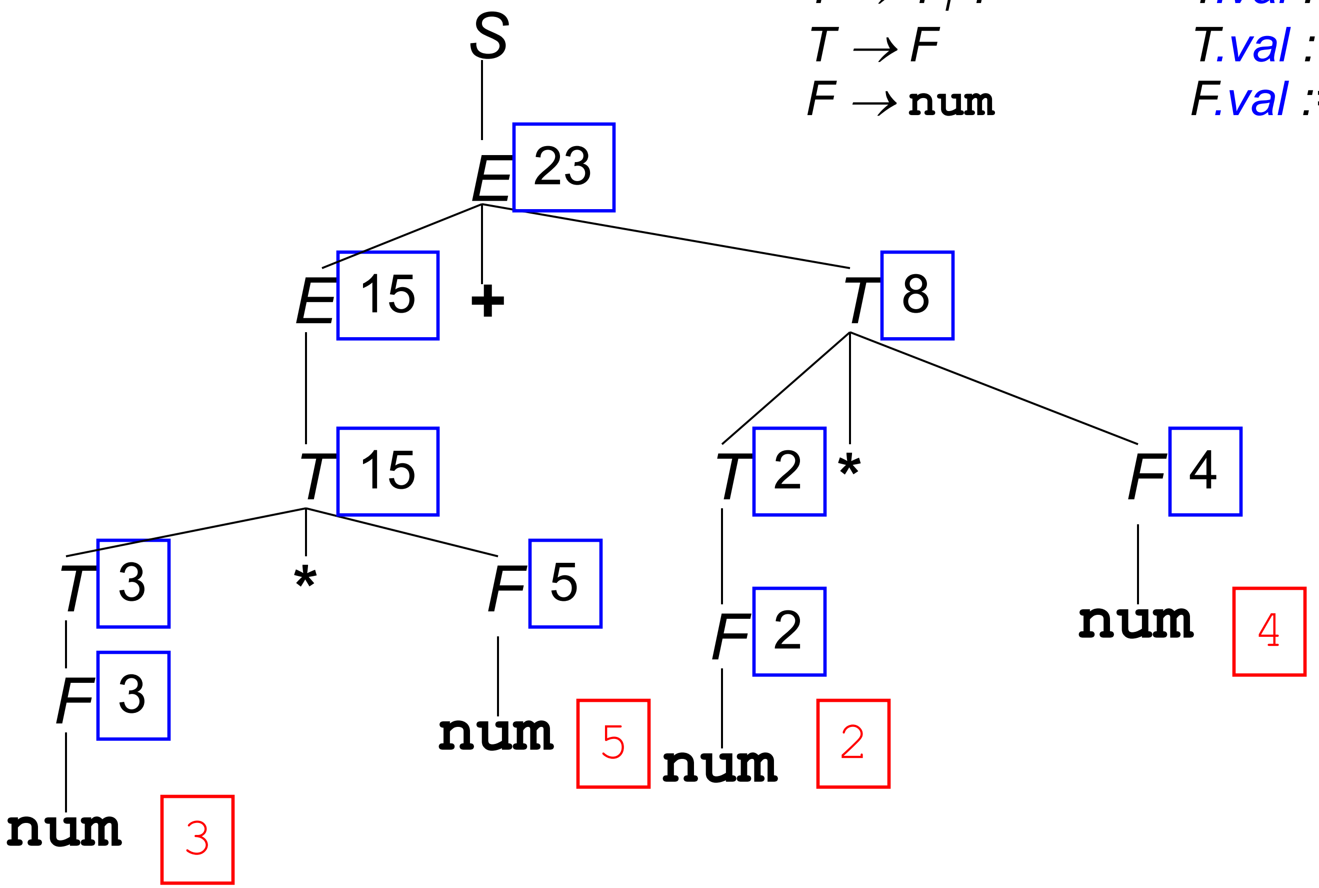
$S \to E \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to \mathbf{num}$

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`

$S \rightarrow E$	print($E.val$)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow$ **num**	$F.val :=$ **num**$.val$



Example

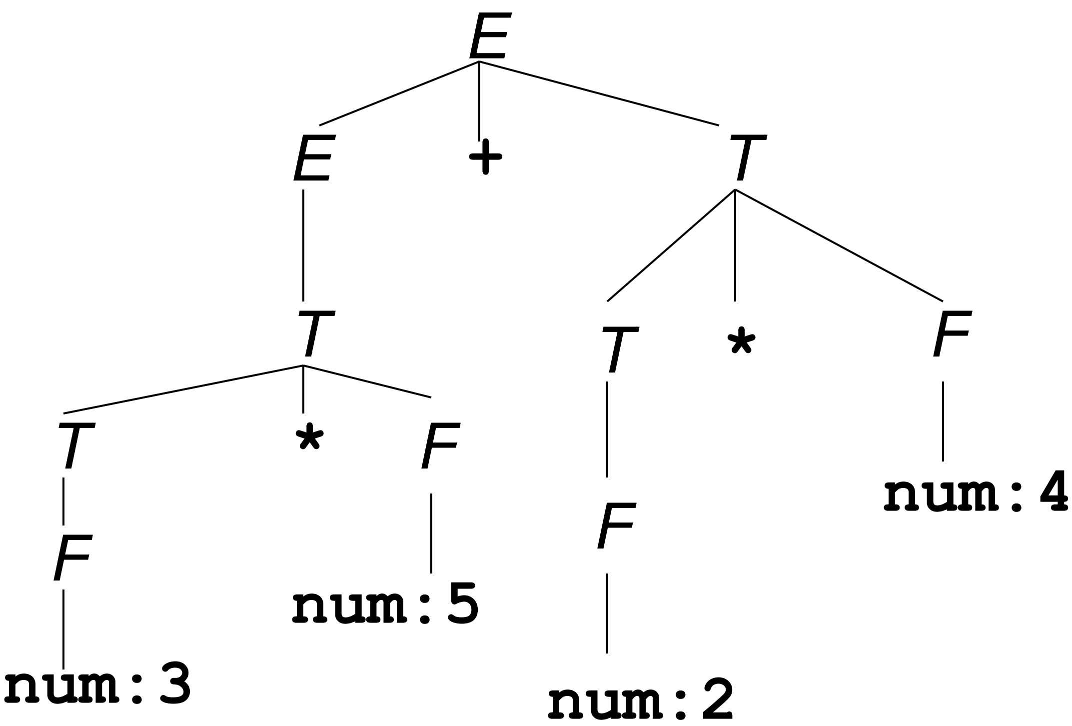
- *val:* Attributes associated to symbols
  - Intuitively, $A.val$ holds the value of the expression, represented by the subtree rooted at A
  - Separate attributes are associated with separate nodes in the parse tree
- Indices are used to distinguish between symbols with same name within same production
  - E.g., $E \rightarrow E_1+T$ $\quad$ $E.val := E_1.val+T.val$
- Attributes of terminals supplied by scanner
  - In example, attributes of **+** and ***** are never used

Building an Abstract Syntax Tree (AST)

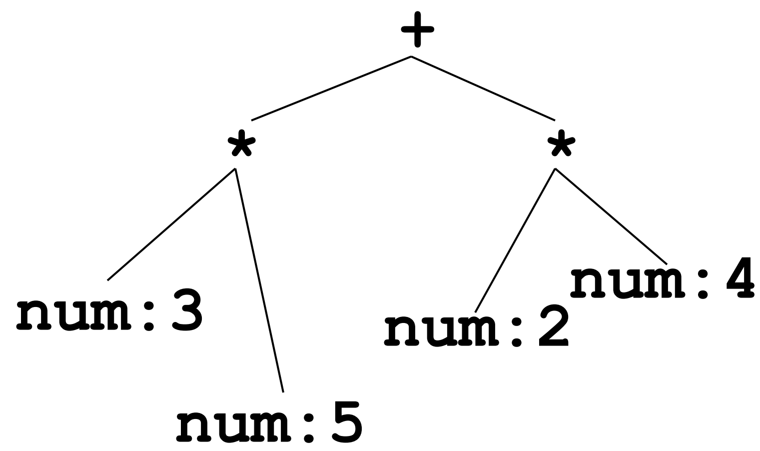
- An AST is an abbreviated parse tree
  - Operators and keywords do not appear as leaves, but at the interior node that would have been their parent
  - Chains of single productions are collapsed
- Compilers typically work with ASTs

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
$E \rightarrow E_1 + T$	$E.nptr := \mathbf{mknode}(+, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow T_1 * F$	$T.nptr := \mathbf{mknode}(*, T_1.nptr, F.nptr)$
$T \rightarrow F$	$T.nptr := F.nptr$
$F \rightarrow \mathbf{num}$	$F.nptr := \mathbf{mkleaf}(\mathbf{num}, \mathbf{num}.val)$

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

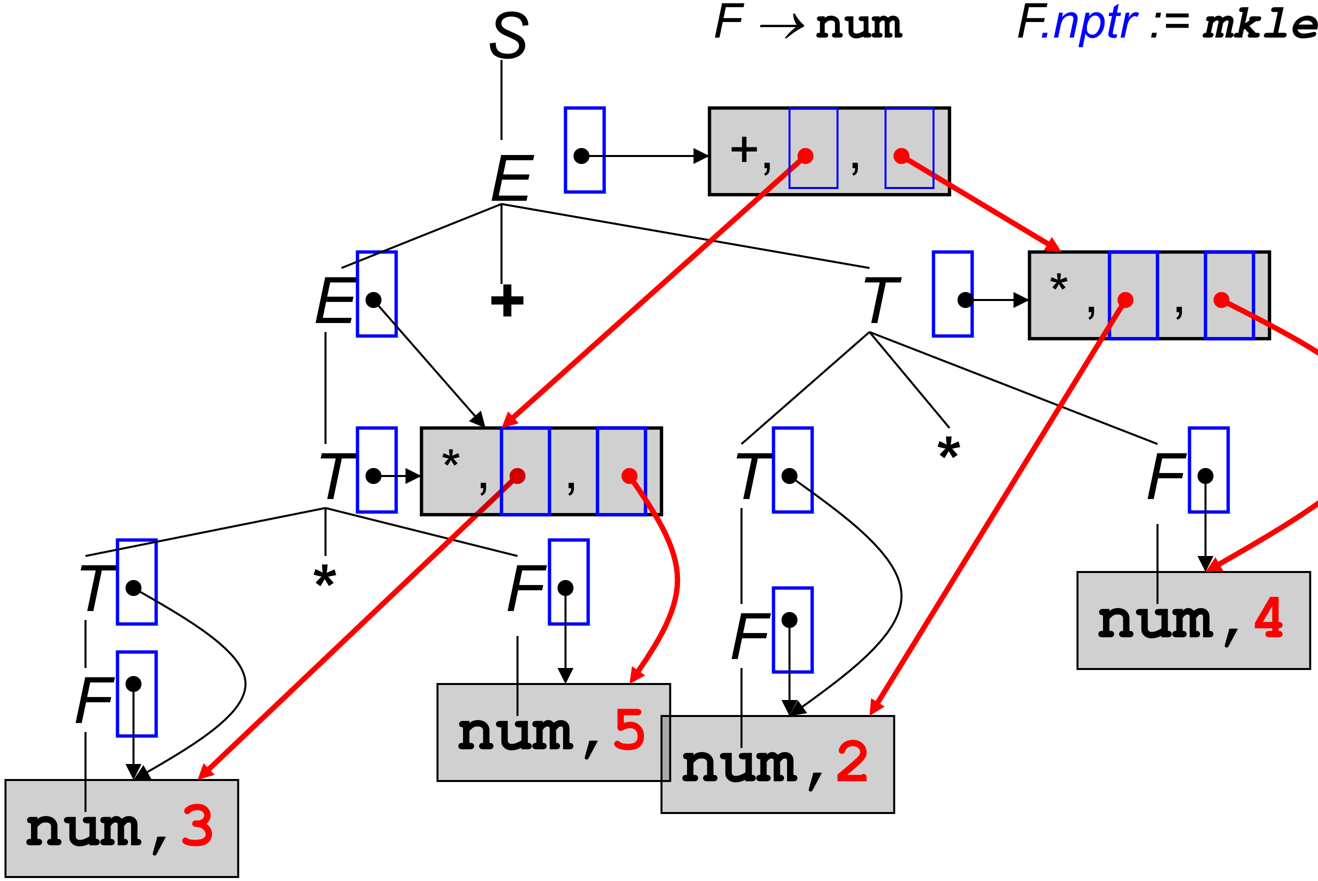
mkleaf(num,num.*val****)*** creates a leaf node with label ***num***, and a field containing the value of the number

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)



Exercise

- We know that the language L = $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ 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.

Exercise

Exercise

- Consider the expression grammar

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

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

Exercise

Another Grammar

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

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

$$E \rightarrow T\ TT$$
$$TT \rightarrow \texttt{-}\ T\ TT$$
$$TT \rightarrow \boldsymbol{\varepsilon}$$
$$T \rightarrow \texttt{num}$$

- 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)`**

Question

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

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

<u>A hack:</u>

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

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

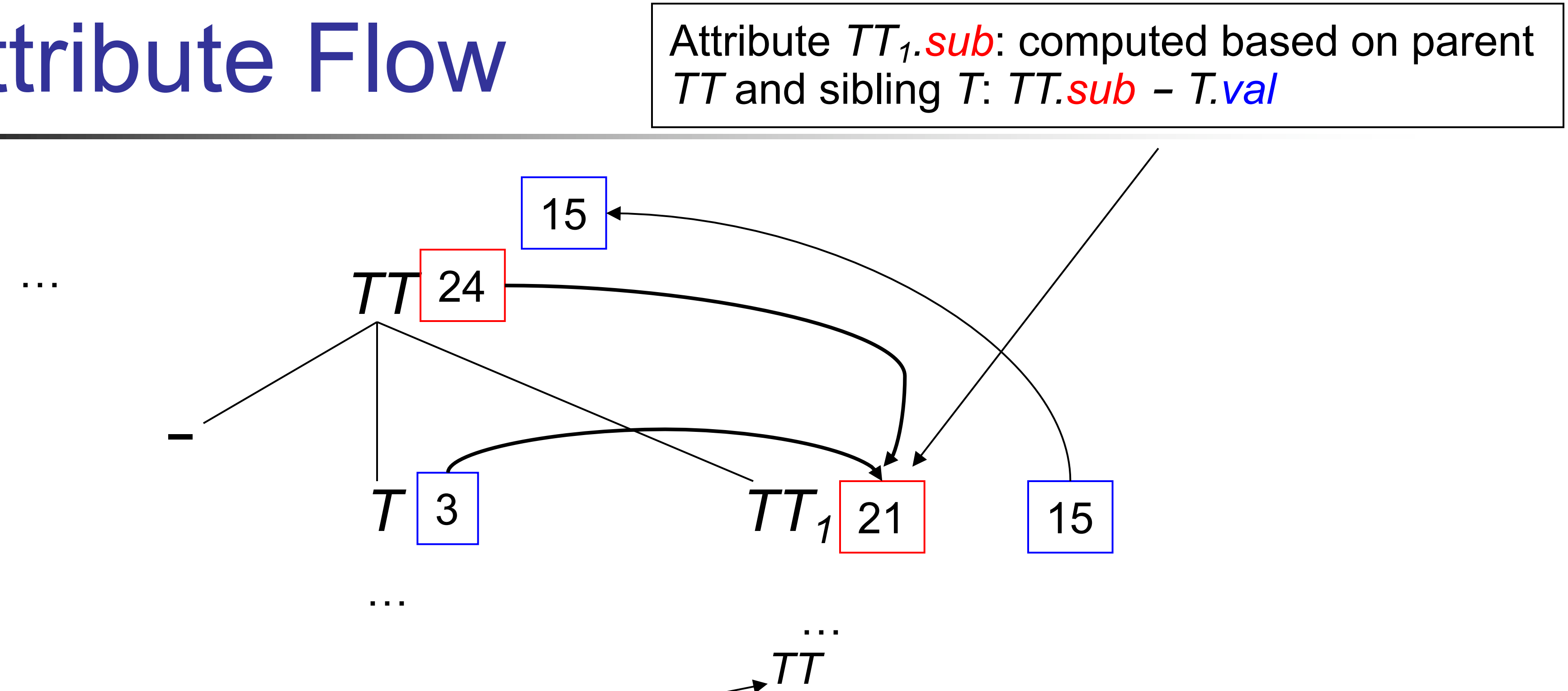
Attribute Grammar to Compute Value of Expressions (denote by AG3)

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

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) $TT.val := TT_1.val$
$TT \to \boldsymbol{\varepsilon}$	(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



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 $TT \rightarrow \varepsilon$ 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
  - *.val* attribute is carried 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*

Lecture Outline

- Syntax vs. static semantics
- Static semantics vs. dynamic semantics

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

Synthesized and Inherited Attributes

- Synthesized attributes
  - Attribute value computed from attributes of descendants in parse tree, and/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 and/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 <u>children</u> in parse tree
  - “Result” is placed in attribute of the symbol on the left-hand-side of the production
    - I.e., computes attribute of <u>parent</u> 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., in what order do we visit nodes of the parse tree and compute attributes, bottom-up or top-down?
  - Answer: bottom-up

L-attributed Grammar

- An attribute grammar is L-attributed if each <u>inherited</u> attribute of X_j on the right-hand-side of $A \rightarrow X_1\, X_2 \ldots X_{j-1} X_j \ldots X_n$ 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?
  - I.e., in what order do we visit the nodes of the parse tree?
  - Answer: top-down

Question

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

Recursive Descent (sketch)

$S \rightarrow E$ **$$**
$E \rightarrow T\ TT$ $\qquad TT \rightarrow -\ T\ \ TT \mid +\ T\ \ TT \mid \varepsilon$ $\qquad T \rightarrow$ **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
```

Evaluating Attributes and Attribute Flow

- 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

The End