COMPUTER SCIENCE

450 COMPILERS

News & Info

- Scott Hunter
  - Friday 5/20 @ 11:30am SCI III 108
- SoCal Code Camp | UC San Diego, CA 6/25-6/26
  - http://www.socalcodecamp.com/

Administrivia

- Lab 07
  - Solution has been posted
- Lab 08
  - Due Thursday

Abstract Syntax Trees

- So far a parser traces the derivation of a sequence of tokens
- The rest of the compiler needs a structural representation of the program
- Abstract syntax trees
  - Like parse trees but ignore some details
  - Abbreviated as AST

Abstract Syntax Tree continued

- Consider the grammar

  $E \rightarrow \text{int} \mid (E) \mid E + E$

- And the string

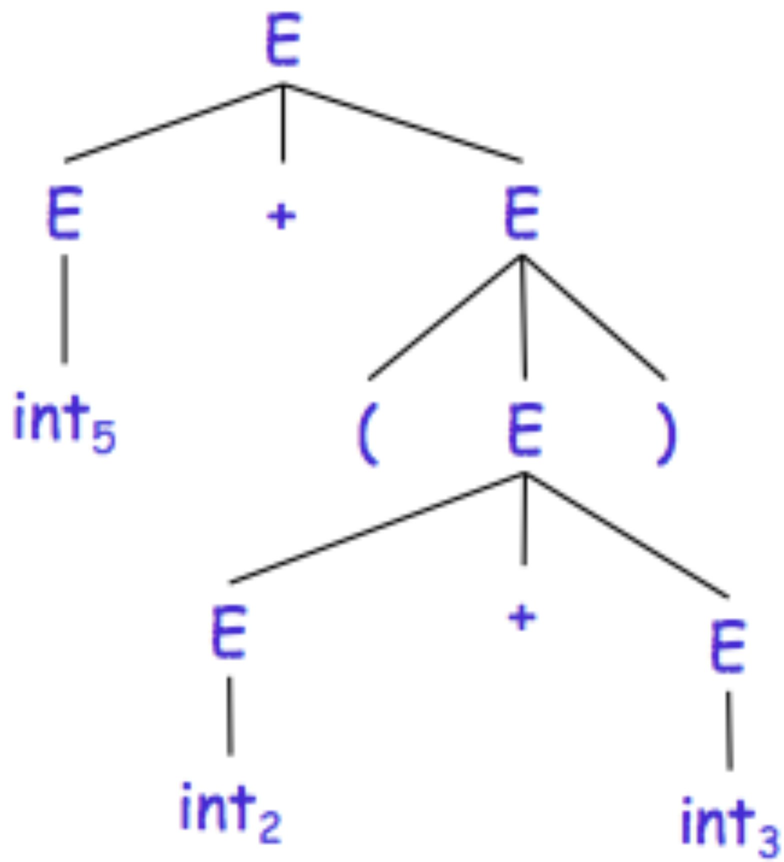
  $5 + (2 + 3)$

- After lexical analysis (a list of tokens)

  int_5 '+' '(' int_2 '+' int_3 ')'

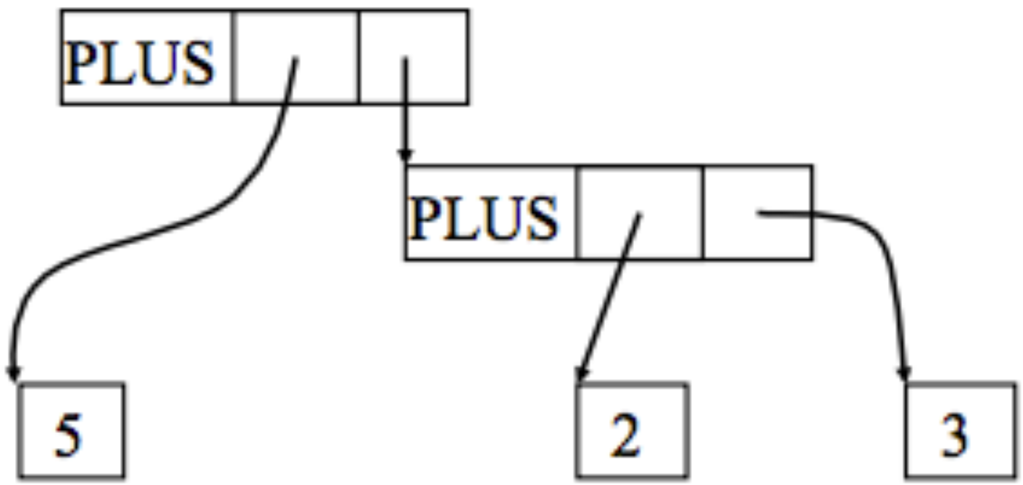
- During parsing we build a parse tree …

Example of Parse Tree



- Traces the operation of the parser
- Does capture the nesting structure
- But too much info
  - Parentheses
  - Single-successor nodes

Example of AST



- Also captures the nesting structure
- But <u>abstracts</u> from the concrete syntax
  => more compact and easier to use
- An important data structure in a compiler

Semantic Actions

- This is what we'll use to construct ASTs
- Each grammar symbol may have <u>attributes</u>
  - For terminal symbols (lexical tokens) attributes can be calculated by the lexer
- Each production may have an <u>action</u>
  - Written as: $X \rightarrow Y_1 \ldots Y_n \quad \{ \text{action} \}$
  - That can refer to or compute symbol attributes

Semantic Actions: Example

- Consider the grammar

$$E \rightarrow \text{int} \mid E + E \mid (E)$$

- For each symbol X define an attribute X.val
  - For terminals, val is the associated lexeme
  - For non-terminals, val is the expression's value (and is computed from values of subexpressions)

- We annotate the grammar with actions:

$$\begin{array}{ll} E \rightarrow \text{int} & \{ E.val = \text{int}.val \} \\ \quad \mid E_1 + E_2 & \{ E.val = E_1.val + E_2.val \} \\ \quad \mid (E_1) & \{ E.val = E_1.val \} \end{array}$$

Semantic Actions: Example continued

- String: 5 + (2 + 3)
- Tokens: int_5 '+' '(' int_2 '+' int_3 ')'

Productions	Equations
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E_1 \rightarrow int_5$	$E_1.val = int_5.val = 5$
$E_2 \rightarrow (E_3)$	$E_2.val = E_3.val$
$E_3 \rightarrow E_4 + E_5$	$E_3.val = E_4.val + E_5.val$
$E_4 \rightarrow int_2$	$E_4.val = int_2.val = 2$
$E_5 \rightarrow int_3$	$E_5.val = int_3.val = 3$

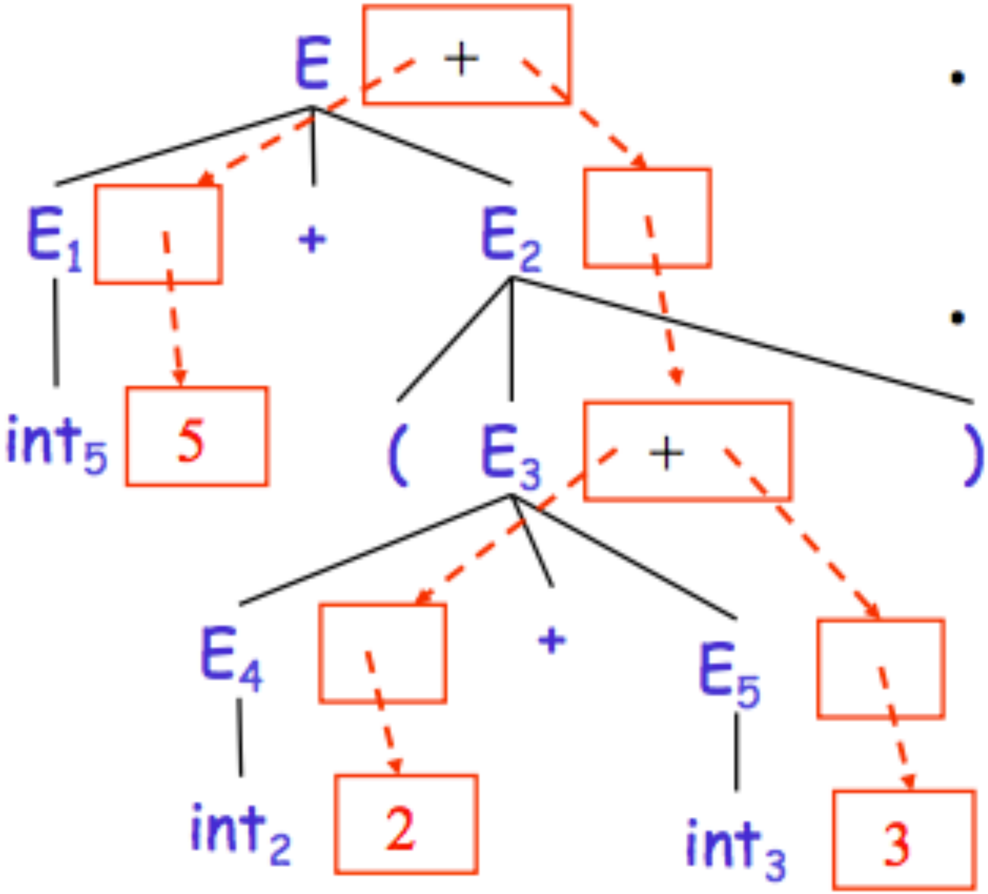
Semantic Actions: Notes

- Semantic actions specify a system of equations
  - Order of resolution is not specified
- Example:

$$E_3.val = E_4.val + E_5.val$$

  - Must compute $E_4.val$ and $E_5.val$ before $E_3.val$
  - We say that $E_3.val$ depends on $E_4.val$ and $E_5.val$
- The parser must find the order of evaluation

Dependency Graph



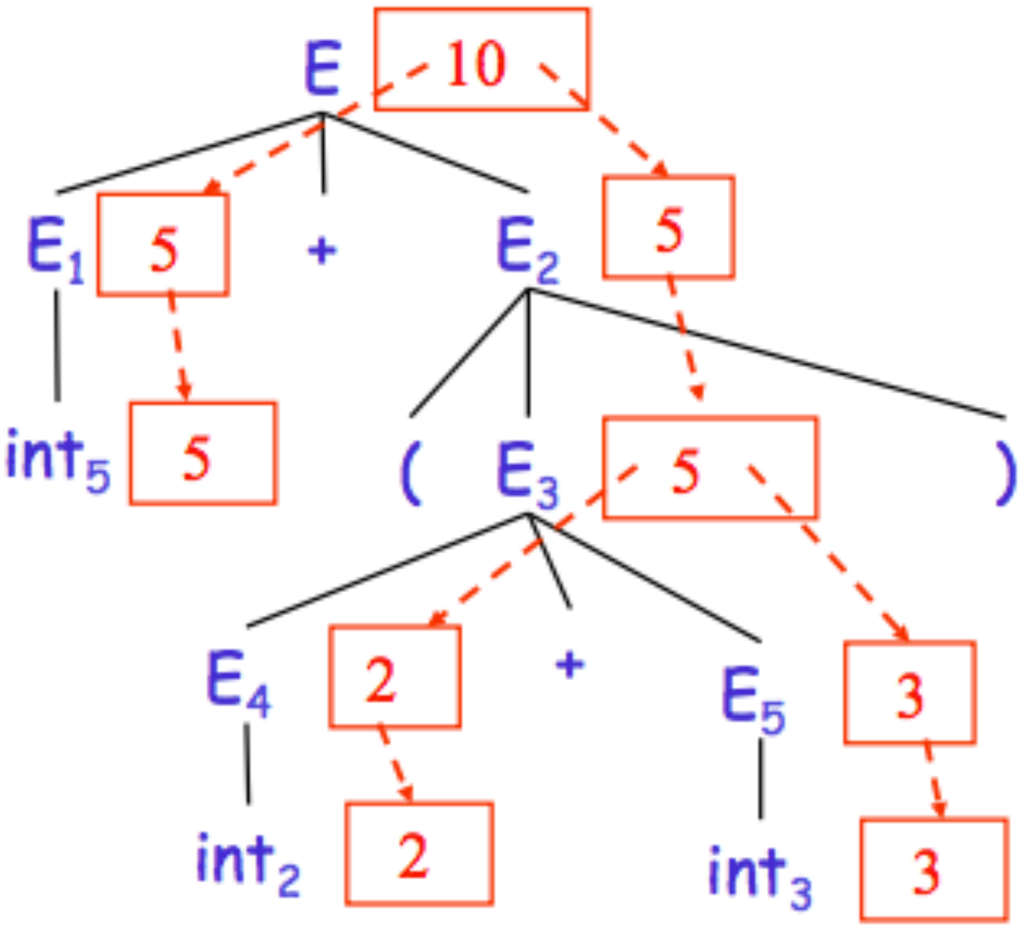
- Each node labeled E has one slot for the val attribute
- Note the dependencies

Evaluating Attributes

- An attribute must be computed after all its successors in the dependency graph have been computed
  - In previous example attributes can be computed bottom-up

- Such an order exists when there are no cycles
  - Cyclically defined attributes are not legal

Dependency Graph



Semantic Actions: Notes

- Synthesized attributes
  - Calculated from attributes of descendents in the parse tree
  - E.val is a synthesized attribute
  - Can always be calculated in a bottom-up order

- Grammars with only synthesized attributes are called S-attributed grammars
  - Most common case

Inherited Attributes

- Another kind of attribute
- Calculated from attributes of parent and/or siblings in the parse tree
- Example: a line calculator

A Line Calculator

- Each line contains an expression

  $E \to \text{int} \mid E + E$

- Each line is terminated with the = sign

  $L \to E = \mid + E =$

- In second form the value of previous line is used as starting value
- A program is a sequence of lines

  $P \to \varepsilon \mid P\ L$

Attributes for the Line Calculator

- Each E has a synthesized attribute val
  - Calculated as before
- Each L has an attribute val

$$
\begin{array}{ll}
L \rightarrow E = & \{ L.val = E.val \} \\
\quad \mid + E = & \{ L.val = E.val + L.prev \}
\end{array}
$$

- We need the value of the previous line
- We use an inherited attribute L.prev

Attributes for the Line Calculator

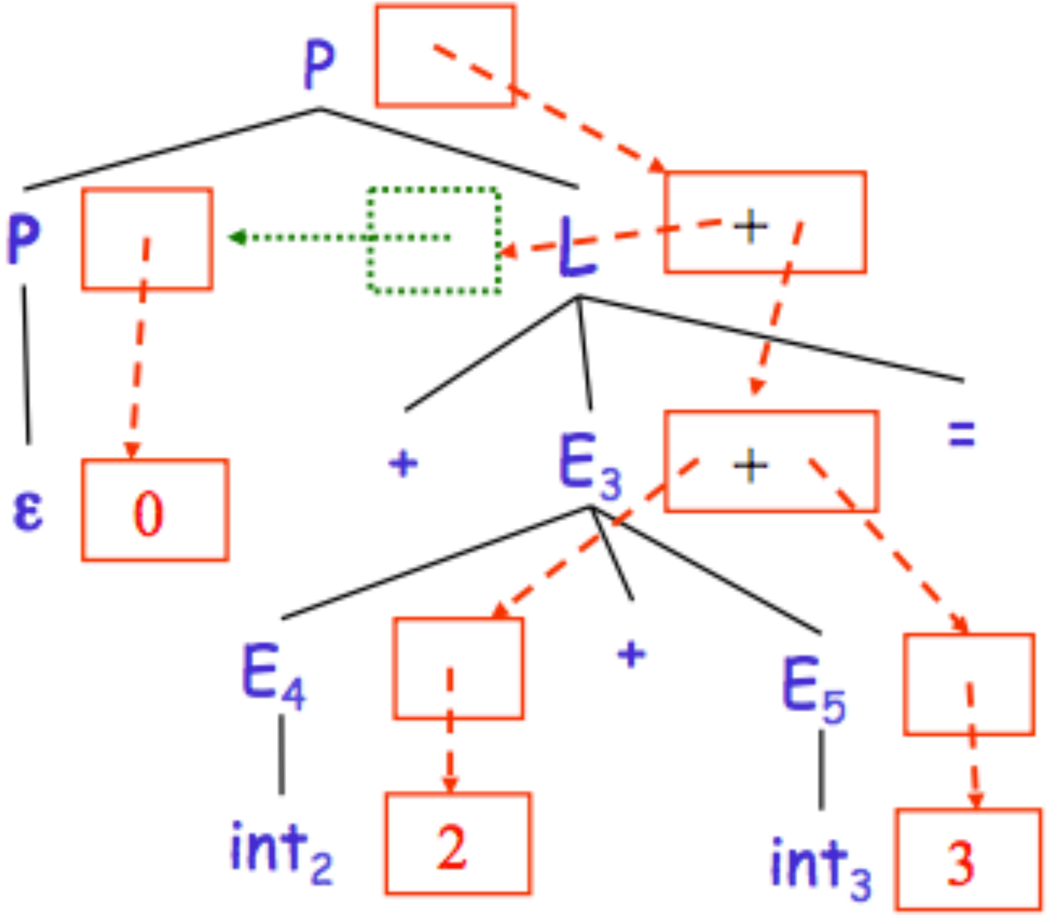
- Each P has a synthesized attribute val
  - The value of its last line

$$
\begin{array}{ll}
P \rightarrow \varepsilon & \{\ P.val = 0\ \} \\
\quad\mid\ P_1\ L & \{\ P.val = L.val; \\
 & \quad L.prev = P_1.val\ \}
\end{array}
$$

  - Each L has an inherited attribute prev
  - L.prev is inherited from sibling P_1.val

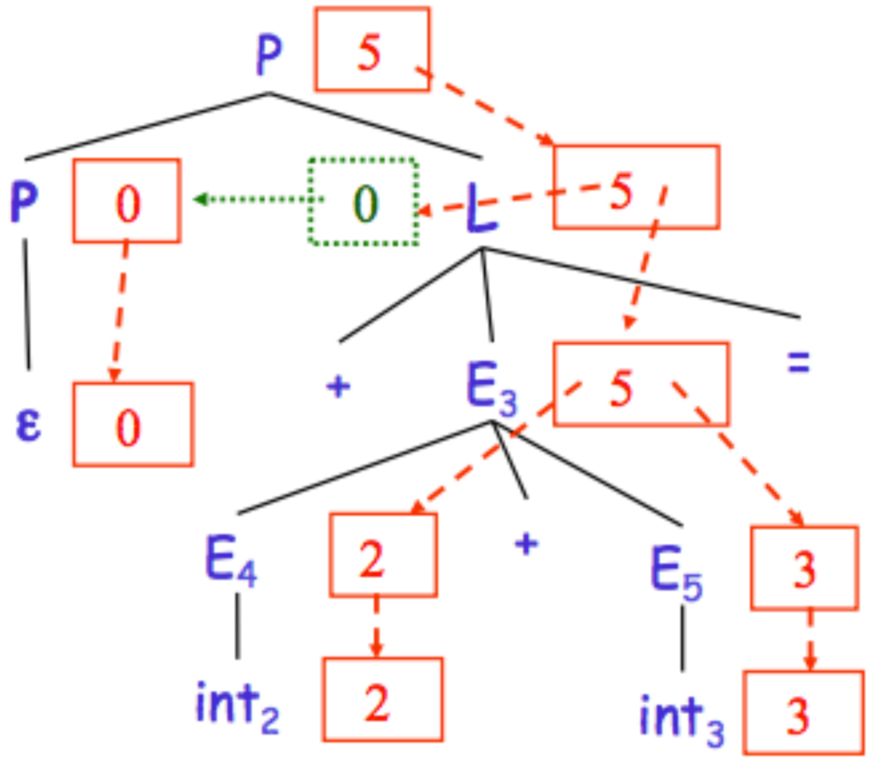
- Example …

Example of Inherited Attributes



- val synthesized
- prev inherited
- All can be computed in depth-first order

Example of Inherited Attributes



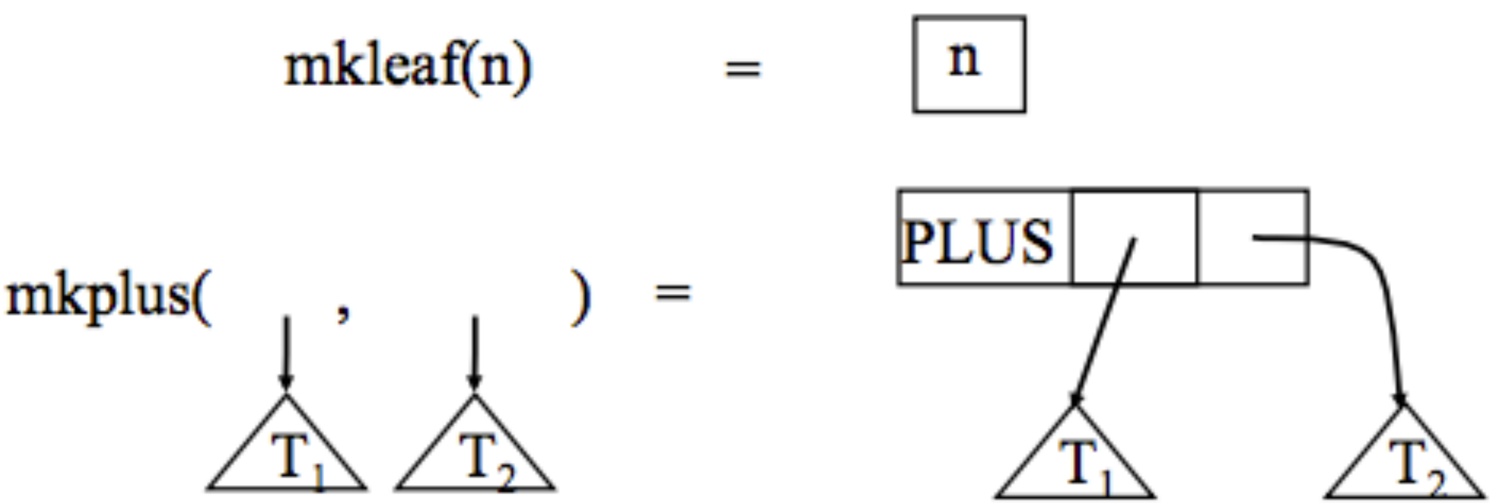
- val synthesized
- prev inherited
- All can be computed in depth-first order

Semantic Actions: Notes

- Semantic actions can be used to build ASTs
- And many other things as well
  - Also used for type checking, code generation, …
- Process is called <u>syntax-directed translation</u>
  - Substantial generalization over CFGs

Constructing An AST

- We first define the AST data type
  - Supplied by us for the project
- Consider an abstract tree type with two constructors:



Constructing a Parse Tree

- We define a synthesized attribute ast
  - Values of ast values are ASTs
  - We assume that int.lexval is the value of the integer lexeme
  - Computed using semantic actions

$E \rightarrow int$	$E.ast = mkleaf(int.lexval)$
$\mid E_1 + E_2$	$E.ast = mkplus(E_1.ast, E_2.ast)$
$\mid (E_1)$	$E.ast = E_1.ast$

Parse Tree Example

- Consider the string int_5 '+' '(' int_2 '+' int_3 ')'
- A bottom-up evaluation of the ast attribute:

  E.ast = mkplus(mkleaf(5),
                                  mkplus(mkleaf(2), mkleaf(3))