Abstract Syntax Trees
&
Top-Down Parsing

Review of Parsing

- Given a language L(G), a parser consumes a sequence of tokens s and produces a parse tree
- Issues:
  - How do we recognize that s ∈ L(G) ?
  - A parse tree of s describes how s ∈ L(G)
  - Ambiguity: more than one parse tree (possible interpretation) for some string s
  - Error: no parse tree for some string s
  - How do we construct the parse tree?

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 Trees (Cont.)

- Consider the grammar

  E → int | (E) | E + E

- And the string

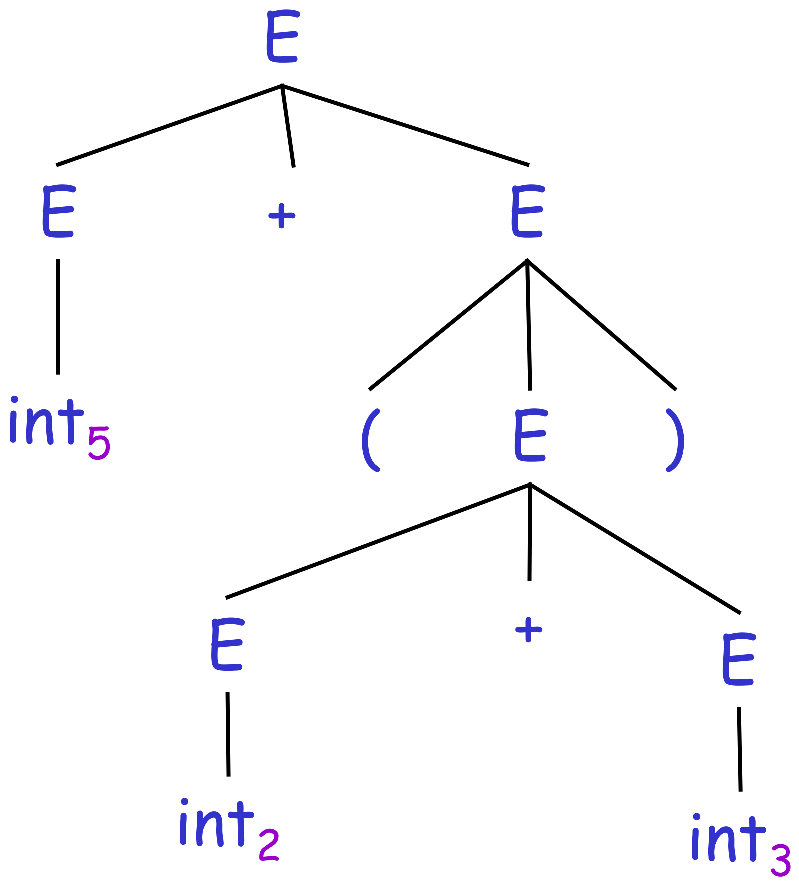
  5 + (2 + 3)

- After lexical analysis (a list of tokens)

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

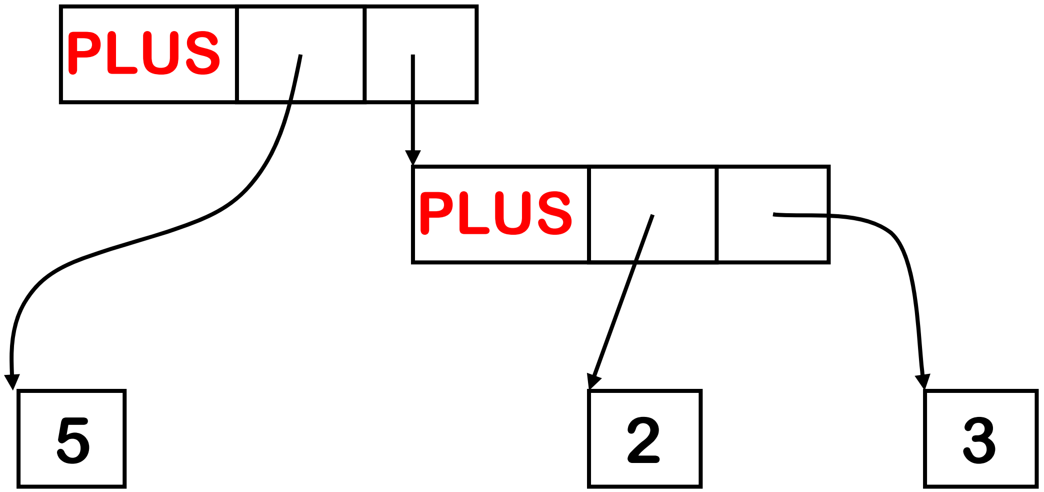
- During parsing we build a parse tree …

Example of Parse Tree



- Traces the operation of the parser
- Captures the nesting structure
- But too much information
  - Parentheses
  - Single-successor nodes

Example of Abstract Syntax Tree



- Also captures the nesting structure
- But abstracts from the concrete syntax
  - ↦ more compact and easier to use
- An important data structure in a compiler

Semantic Actions

- This is what we will use to construct ASTs

- Each grammar symbol may have attributes
  - An attribute is a property of a programming language construct
  - For terminal symbols (lexical tokens) attributes can be calculated by the lexer
- Each production may have an action
  - Written as: $X \rightarrow Y_1 \ldots Y_n$ { action }
  - That can refer to or compute symbol attributes

Semantic Actions: An 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 (which is computed from values of subexpressions)
- We annotate the grammar with actions:

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

Semantic Actions: An Example (Cont.)

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

Productions	Equations
$E \rightarrow E_1 + E_2$	$E.\text{val} = E_1.\text{val} + E_2.\text{val}$
$E_1 \rightarrow \text{int}_5$	$E_1.\text{val} = \text{int}_5.\text{val} = 5$
$E_2 \rightarrow (E_3)$	$E_2.\text{val} = E_3.\text{val}$
$E_3 \rightarrow E_4 + E_5$	$E_3.\text{val} = E_4.\text{val} + E_5.\text{val}$
$E_4 \rightarrow \text{int}_2$	$E_4.\text{val} = \text{int}_2.\text{val} = 2$
$E_5 \rightarrow \text{int}_3$	$E_5.\text{val} = \text{int}_3.\text{val} = 3$

Semantic Actions: Dependencies

Semantic actions specify a system of equations

- Order of executing the actions is not specified

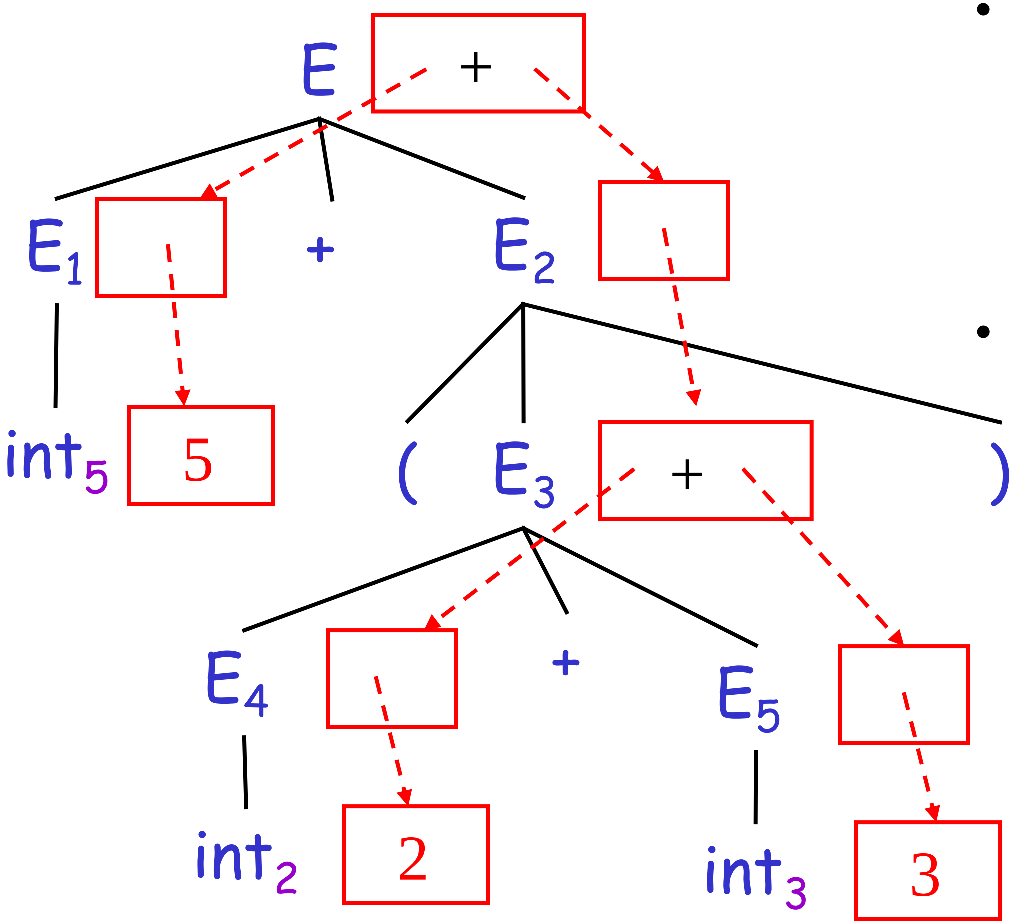
- Example:

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

  - 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 with a non-terminal E has one slot for its val attribute
- Note the dependencies

Evaluating Attributes

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

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

Semantic Actions: Notes (Cont.)

- 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 frequent kinds of grammars

Inherited Attributes

- Another kind of attributes
- Calculated from attributes of the parent node(s) and/or siblings in the parse tree

- Example: a line calculator

A Line Calculator

- Each line contains an expression

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

- Each line is terminated with the = sign

  $L \rightarrow E = \mid + E =$

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

  $P \rightarrow \varepsilon \mid P\,L$

Attributes for the Line Calculator

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

$L \to E = \quad \{\ L.val = E.val\ \}$

$\quad \mid\ + E = \quad \{\ L.val = E.val + L.prev\ \}$

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

Attributes for the Line Calculator (Cont.)

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

$P \to \varepsilon$ { P.val = 0 }

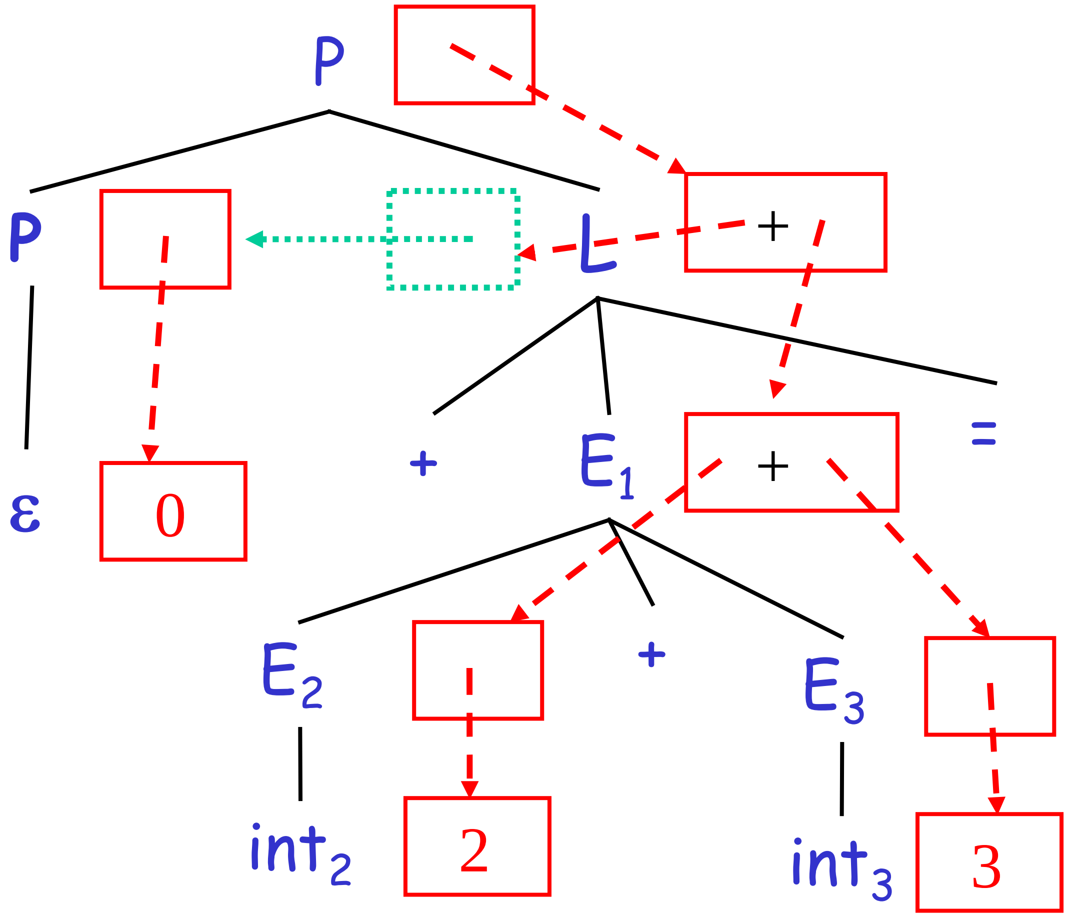
$\mid$ P_1 L { P.val = L.val;

L.prev = P_1.val }

- 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

Semantic Actions: Notes (Cont.)

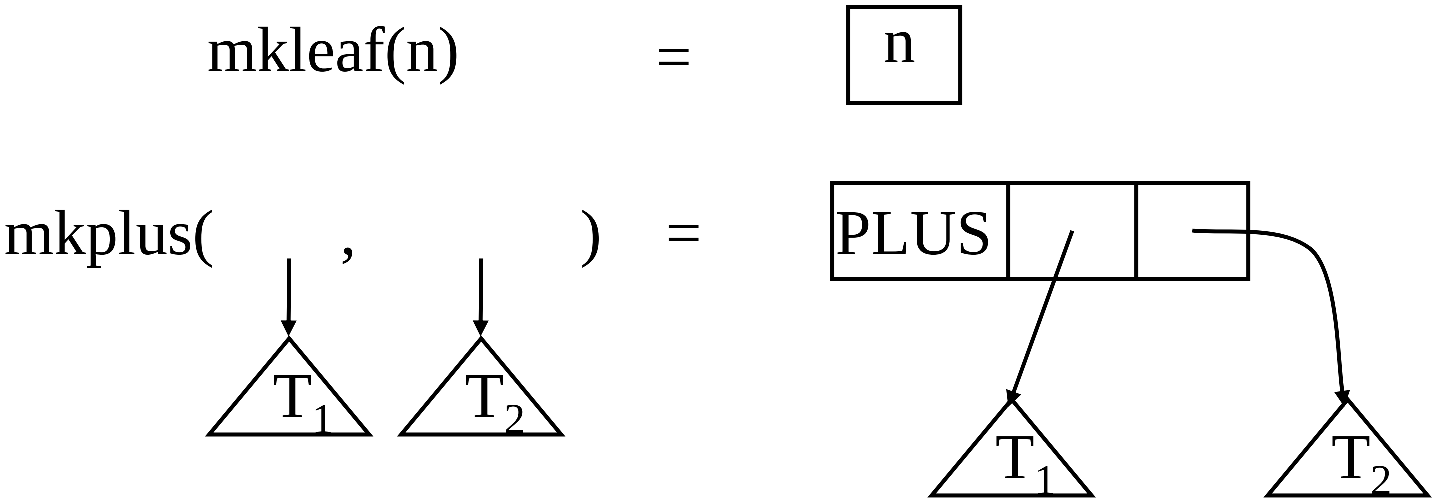
- Semantic actions can be used to build ASTs

- 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
- Consider an abstract tree type with two constructors:



mkleaf(n) = n

mkplus(,) = PLUS

Constructing a Parse Tree

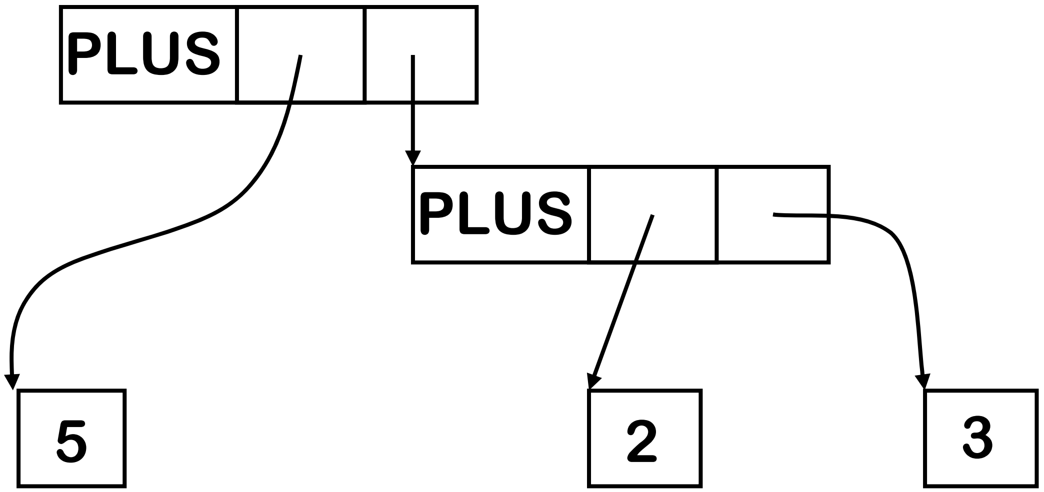
- We define a synthesized attribute ast
  - Values of ast values are ASTs
  - We assume that int.lexval is the value of the integer lexeme
  - Computed using semantic actions

$$
\begin{array}{lll}
E \rightarrow \text{int} & & \{\ E.\text{ast} = \text{mkleaf}(\text{int.lexval})\ \} \\
\quad\mid\ E_1 + E_2 & & \{\ E.\text{ast} = \text{mkplus}(E_1.\text{ast},\ E_2.\text{ast})\ \} \\
\quad\mid\ (\ E_1\ ) & & \{\ E.\text{ast} = E_1.\text{ast}\ \}
\end{array}
$$

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))



Review of Abstract Syntax Trees

- We can specify language syntax using CFG.
- The parser answers whether $s \in L(G)$
- … and builds a parse tree
- … which it converts to an AST
- … and passes on to the rest of the compiler.

- In the next "parsing" lectures:
  - How do we answer $s \in L(G)$ and build a parse tree?
- After that: from AST to assembly language.

Second-Half of Lecture: Outline

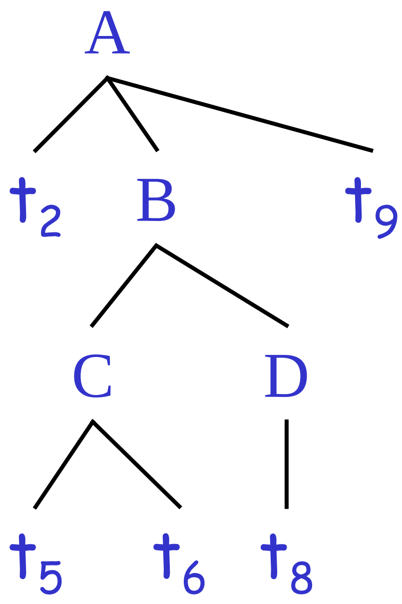
- Implementation of parsers
- Two approaches
  - Top-down
  - Bottom-up
- These slides: Top-Down
  - Easier to understand and program manually
- Next lectures: Bottom-Up
  - More powerful and used by most parser generators

Introduction to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream:

  $t_2 \quad t_5 \quad t_6 \quad t_8 \quad t_9$

- The parse tree is constructed
  - From the top
  - From left to right



Recursive Descent Parsing: Example

- Consider the grammar

  $E \to T + E \mid T$

  $T \to (E) \mid \text{int} \mid \text{int} * T$

- Token stream is: $\text{int}_5 * \text{int}_2$
- Start with top-level non-terminal E

- Try the rules for E in order

Recursive Descent Parsing: Example

Token stream: $\text{int}_5 * \text{int}_2$

- Try $E_0 \rightarrow T_1 + E_2$
- Then try a rule for $T_1 \rightarrow (E_3)$
  - But (does not match input token int_5 ; we backtrack.
- Try $T_1 \rightarrow \text{int}$. Token matches.
  - But + after T_1 does not match input token *
- Try $T_1 \rightarrow \text{int} * T_2$
  - This will match and will consume the two tokens.
    - Try $T_2 \rightarrow \text{int}$ (matches) but + after T_1 will be unmatched.
    - Try $T_2 \rightarrow \text{int} * T_3$ but * does not match with end-of-input.
- We have exhausted all the choices for T_1
  - Backtrack to choice for E_0

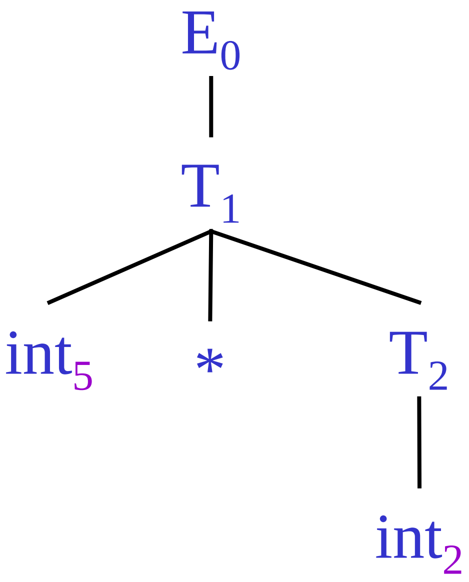
$E \rightarrow T + E \mid T$

$T \rightarrow (E) \mid \text{int} \mid \text{int} * T$

Recursive Descent Parsing: Example

Token stream: int_5 * int_2

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for T_1
  - And succeed with $T_1 \rightarrow \text{int}_5 * T_2$ and $T_2 \rightarrow \text{int}_2$
  - With the following parse tree



$E \rightarrow T + E \mid T$

$T \rightarrow (E) \mid \text{int} \mid \text{int} * T$

Recursive Descent Parsing: Notes

- Easy to implement by hand

- Somewhat inefficient (due to backtracking)

- But does not always work …

When Recursive Descent Does Not Work

- Consider a production $S \to S\ a$

```
bool S₁() { return S() && term(a);  }
bool S() { return S₁(); }
```

- S() will get into an infinite loop

- We call a grammar left-recursive if it has a non-terminal S

$$S \to^{+} S\alpha \quad \text{for some } \alpha$$

- Recursive descent does not work in such cases – it goes into an infinite loop.

Elimination of Left Recursion

- Consider the left-recursive grammar:

$$S \rightarrow S\ \alpha \mid \beta$$

- Generates all strings starting with a β and followed by any number of α's.

- The grammar can be rewritten using right recursion:

$$S \rightarrow \beta\ S'$$

$$S' \rightarrow \alpha\ S' \mid \varepsilon$$

More Elimination of Left-Recursion

- In general

$$S \to S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1, \ldots, \beta_m$ and continue with several instances of $\alpha_1, \ldots, \alpha_n$
- Rewrite as

$$S \to \beta_1\ S' \mid \ldots \mid \beta_m\ S'$$
$$S' \to \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$$

General Left Recursion

- The grammar

$$S \rightarrow A\ \alpha \mid \delta$$

$$A \rightarrow S\ \beta$$

is also left-recursive because

$$S \rightarrow^{+} S\ \beta\ \alpha$$

- This left-recursion can also be eliminated

[See a Compilers book for a general algorithm]

Summary of Recursive Descent

- Simple and general parsing strategy.
  - Left-recursion must be eliminated first
  - … but that can be done automatically.
- Unpopular because of backtracking.
  - Thought to be too inefficient.
- In practice, backtracking is eliminated by restricting the grammar.

Predictive Parsers

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - k means "predict based on k tokens of lookahead"
- In practice, LL(1) is used

LL(1) Languages

- In recursive-descent, for each non-terminal and input token there may be a choice of productions
- LL(1) means that for each non-terminal and token there is only one production that could lead to success
- Can be specified via 2D tables
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production

Predictive Parsing and Left Factoring

- Recall the grammar for arithmetic expressions

  $E \rightarrow T + E \mid T$

  $T \rightarrow (E) \mid \text{int} \mid \text{int} * T$

- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict

- A grammar must be <u>left-factored</u> before it is used for predictive parsing

Left-Factoring Example

- Recall the grammar

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

- Factor out common prefixes of productions:

$$E \rightarrow T\, X$$
$$X \rightarrow + E \mid \varepsilon$$
$$T \rightarrow (E) \mid \text{int}\, Y$$
$$Y \rightarrow \varepsilon \mid * T$$

- This grammar is equivalent to the original one.

LL(1) Parsing Table Example

- Left-factored grammar

$E \rightarrow T\ X \qquad\qquad X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow (\ E\ ) \mid \text{int}\ Y \qquad\qquad Y \rightarrow *\ T \mid \varepsilon$

- The LL(1) parsing table ($ is the end marker)

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \rightarrow T\ X$ "
  - This production can generate an int in the first place
- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only in a derivation in which $Y \rightarrow \varepsilon$

LL(1) Parsing Tables: Errors

- Blank entries indicate error situations
  - Consider the [E,*] entry
  - “There is no way to derive a string starting with * from non-terminal E”

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal X
  - We look at the next token a
  - And choose the production shown at [X,a]
- We use a stack to keep track of pending non-terminals.
- We reject when we encounter an error state.
- We accept when we encounter end-of-input.

LL(1) Parsing Algorithm

```
initialize stack ← <S $> and next
repeat
  case stack of
    <X, rest> : if T[X,*next] == $Y_1…Y_n$
                then stack ← <$Y_1…Y_n$ rest>;
                else error();
    <t, rest> : if t == *next++
                then stack ← <rest>;
                else error();
until stack == <>
```

LL(1) Parsing Example

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε