Lecture 2: Context-free grammars, recursive descent parsing

David Hovemeyer

August 28, 2024

601.428/628 Compilers and Interpreters

Today

- Context-free grammars, derivations, parse trees
- Recursive descent parsing
- Expression grammars
- Ambiguity
- Operator precedence and associativity

Context-free grammars, parse trees

Context-free grammars

- *Context-free grammars* are the most common way of describing the syntax of a programming language
- If a source module conforms to the language's grammar rules, it is syntactically valid
  - Which doesn't imply that it's *semantically* valid

Context-free grammar concepts

- The *input string* is a sequence of *terminal symbols*
  - For an interpreter or compiler, the terminal symbols are the input tokens scanned by the lexical analyzer
- The grammar is a set of *productions*:
  - One *nonterminal symbol* on the left hand side
  - Sequence of zero or more terminal and/or nonterminal symbols on the right hand side
- The grammar has one nonterminal *start symbol*
- An input string is in the language specified by the grammar if it can be *derived* from the grammar

Example context-free grammar

Nonterminal symbol: $\boxed{\mathsf{E}}$ (start symbol)

Terminal symbols: $\boxed{\mathsf{i\ n\ +\ -\ *\ /\ =}}$
(note that 'i' and 'n' mean 'identifier' and 'number')

Grammar:

$$
\begin{aligned}
\mathsf{E} &\rightarrow + \ \mathsf{E}\ \mathsf{E}\\
\mathsf{E} &\rightarrow - \ \mathsf{E}\ \mathsf{E}\\
\mathsf{E} &\rightarrow * \ \mathsf{E}\ \mathsf{E}\\
\mathsf{E} &\rightarrow / \ \mathsf{E}\ \mathsf{E}\\
\mathsf{E} &\rightarrow = \ \mathsf{i}\ \mathsf{E}\\
\mathsf{E} &\rightarrow \mathsf{i}\\
\mathsf{E} &\rightarrow \mathsf{n}
\end{aligned}
$$

Derivations

Deriving a string means:

- The *working string* initially consists of the start symbol
- Repeatedly:
  - Choose a nonterminal symbol in the working string, and a production with that nonterminal symbol on its left hand side
  - Replace the chosen nonterminal symbol in the working string with the sequence of symbols on the right hand side of the production

The process ends when the working string has no terminal symbols remaining

Example derivation

Input string: $\boxed{+ \text{ - } 4\ 1\ 5}$

(Note that $\boxed{4}$, $\boxed{1}$, and $\boxed{5}$ are occurrences of the 'n' terminal symbol, so really we are deriving $\boxed{+ \text{ - } n\ n\ n}$)

Working string	Production
$\underline{E}$	

Example derivation

Input string: $\boxed{+ \text{ - } 4\ 1\ 5}$

(Note that $\boxed{4}$, $\boxed{1}$, and $\boxed{5}$ are occurrences of the 'n' terminal symbol, so really we are deriving $\boxed{+ \text{ - } n\ n\ n}$)

Working string	Production
$\underline{E}$	$E \rightarrow + E\ E$
$+ \underline{E}\ E$	

Example derivation

Input string: $\boxed{+\ \text{-}\ 4\ 1\ 5}$

(Note that $\boxed{4}$, $\boxed{1}$, and $\boxed{5}$ are occurrences of the 'n' terminal symbol, so really we are deriving $\boxed{+\ \text{-}\ \text{n}\ \text{n}\ \text{n}}$)

Working string	Production
$\underline{\text{E}}$	E $\rightarrow$ + E E
+ $\underline{\text{E}}$ E	E $\rightarrow$ - E E
+ - $\underline{\text{E}}$ E E	

Example derivation

Input string: $\boxed{+ \text{ - } 4\ 1\ 5}$

(Note that $\boxed{4}$, $\boxed{1}$, and $\boxed{5}$ are occurrences of the 'n' terminal symbol, so really we are deriving $\boxed{+ \text{ - } n\ n\ n}$)

Working string	Production
$\underline{E}$	$E \rightarrow +\ E\ E$
$+\ \underline{E}\ E$	$E \rightarrow$ - $E\ E$
$+$ - $\underline{E}\ E\ E$	$E \rightarrow n$
$+$ - $n\ \underline{E}\ E$	

Example derivation

Input string: $\boxed{+ \text{ - } 4\ 1\ 5}$

(Note that $\boxed{4}$, $\boxed{1}$, and $\boxed{5}$ are occurrences of the 'n' terminal symbol, so really we are deriving $\boxed{+ \text{ - } n\ n\ n}$)

Working string	Production
$\underline{E}$	$E \rightarrow + E\ E$
$+ \underline{E}\ E$	$E \rightarrow$ - $E\ E$
$+$ - $\underline{E}\ E\ E$	$E \rightarrow n$
$+$ - $n\ \underline{E}\ E$	$E \rightarrow n$
$+$ - $n\ n\ \underline{E}$	

Example derivation

Input string: $\boxed{+ \text{ - } 4\ 1\ 5}$

(Note that $\boxed{4}$, $\boxed{1}$, and $\boxed{5}$ are occurrences of the 'n' terminal symbol, so really we are deriving $\boxed{+ \text{ - n n n}}$)

Working string	Production
$\underline{E}$	E $\rightarrow$ + E E
+ $\underline{E}$ E	E $\rightarrow$ - E E
+ - $\underline{E}$ E E	E $\rightarrow$ n
+ - n $\underline{E}$ E	E $\rightarrow$ n
+ - n n $\underline{E}$	E $\rightarrow$ n
+ - n n n	

$\boxed{\text{Done!}}$

Parse trees

A *parse tree* is a data structure reflecting the productions applied in a derivation:

- The start symbol is the root
- Applying a production attaches new nodes — the symbols on the right hand side of the production — to the node representing the production's left hand side nonterminal symbol

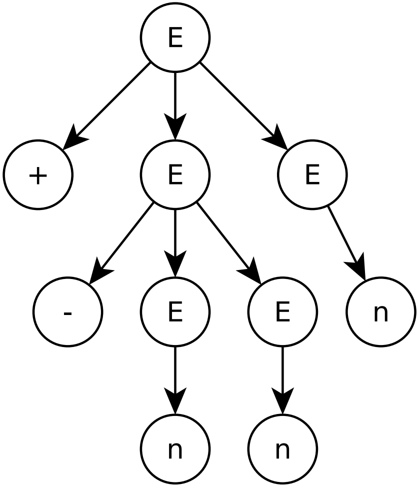
It sounds more complicated than it is, let's do it for the example derivation

Parse tree example

Working string	Production
$\underline{E}$	$E \rightarrow + E E$
$+ \underline{E} E$	$E \rightarrow - E E$
$+ - \underline{E} E E$	$E \rightarrow n$
$+ - n \underline{E} E$	$E \rightarrow n$
$+ - n n \underline{E}$	$E \rightarrow n$
$+ - n n n$	

Parse tree example

Working string	Production
E	E → + E E
+ E E	E → - E E
+ - E E E	E → n
+ - n E E	E → n
+ - n n E	E → n
+ - n n n	



What this all means

OK, so what does any of this have to do with compilers and interpreters?

The idea is that we can carefully design a language's grammar:

- Each nonterminal symbol corresponds to a syntactic construct in the language, e.g., "E" means "prefix expression"
- The structure of the parse tree corresponds to the structure of the program, e.g., when the first child of an "E" node is "$+$", it's an addition

The idea that semantic properties follow from syntax is sometimes referred to as "syntax-directed translation"

Important point: in general many different context-free grammars can describe the same language, but not every grammar will correctly represent the intended meaning of derived strings

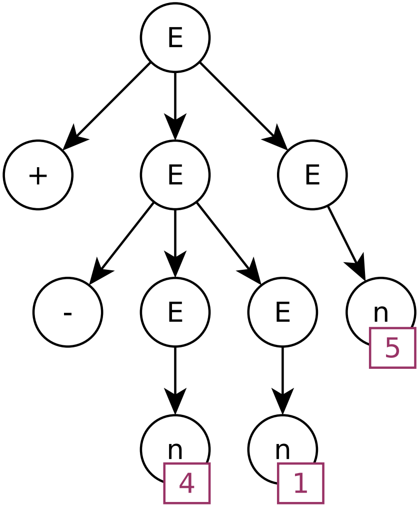
Demonstration that parse trees are useful

Consider our example parse tree

Note that we've annotated the 'n' terminal nodes with their lexemes (recall that the original string was $\boxed{+ \text{ - } 4\ 1\ 5}$)

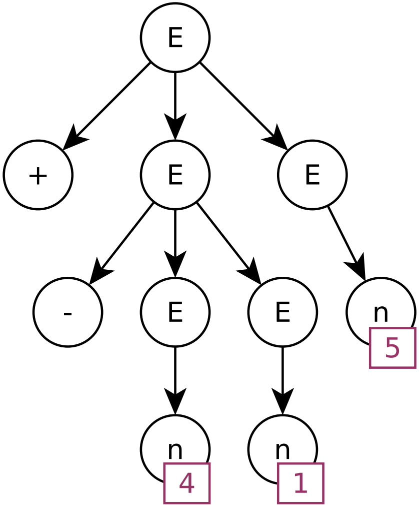
Two ideas:

- The 'n' nodes are literal values
- We can propagate values up towards the root, applying operations, until we know the value of the root node



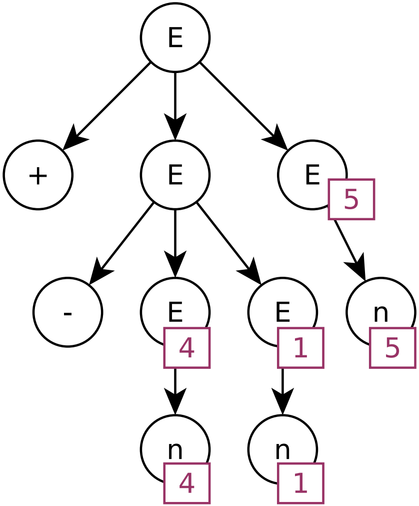
Interpretation

Start



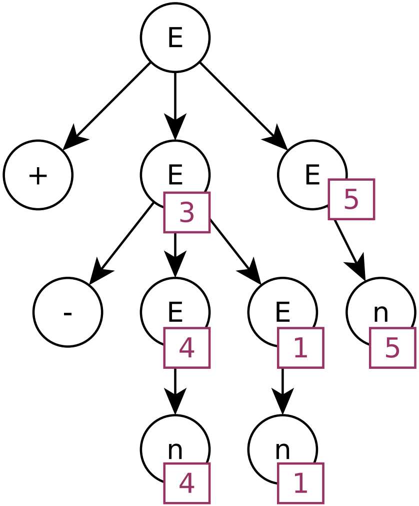
Interpretation

Propagate literal values



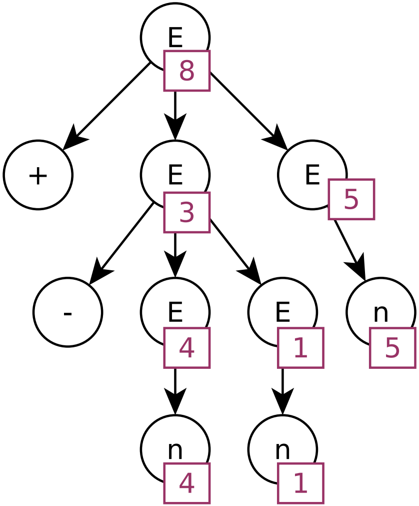
Interpretation

Do the subtraction



Interpretation

Do the addition



Parsing, recursive descent

Parsing

Parsing is the process of finding a derivation for an input string

Since compilers and interpreters are programs, we will need a *parsing algorithm* to automate this

Today we'll introduce *recursive descent* parsing, an incredibly useful and fairly easy ad-hoc parsing technique

Recursive descent parsing

Basic ideas:

- Each nonterminal symbol has a *parse function*
- The goal of a parse function is to apply one production with its nonterminal on the left hand side
  - E.g., the parse function for the E nonterminal will try to apply a production with E on the left hand side
- Applying a production means, for each symbol on the right hand side of the production:
  - If it's a terminal symbol, use the lexer to consume it (advancing to the next input token); if the wrong kind of terminal is consumed, or if the lexer has reached end of input, report an error
  - If it's a nonterminal symbol, call its parse function

Predictive parsing

How does a parse function choose which production to apply?

- If there is only one possible production, apply it unconditionally
- Otherwise, call the lexer's "peek" function to see what the next token will be, and use that to make a decision

Ideal case is when all of the possible productions are distinguished by a unique first terminal symbol on the right hand side

- In this case, the "peek" operation should identify a unique production (or indicate that there is no valid production)

In reality, it's sometimes a bit more complicated

Recursive descent: the reality

In practice, many grammars will require some cleverness:

- Two productions might share a common "prefix" of right hand side symbols
  - In this case, can "partially" apply both productions, until we reach a point where they can be distinguished
- There can be productions with a *nonterminal* symbol as the first right hand side symbol
  - The lexer can only predict what *terminal* symbols appear next in the input
  - "First sets" can allow the parser to make predictions about nonterminals, more on this idea soon
- An *epsilon production* has no symbols on the right hand side
  - The parser should apply an epsilon production only if no other (non-epsilon) production makes sense

Recursive descent parser implementation

From pfxcalc program: https://github.com/daveho/pfxcalc/

Terminal symbols: $\boxed{\text{i n + - * / = ;}}$

Nonterminal symbols: $\boxed{\text{U E}}$

Grammar:

$$
\begin{aligned}
U &\rightarrow E \; ; \; U \\
U &\rightarrow E \\
E &\rightarrow + \; E \; E \\
E &\rightarrow - \; E \; E \\
E &\rightarrow * \; E \; E \\
E &\rightarrow / \; E \; E \\
E &\rightarrow = \; i \; E \\
E &\rightarrow i \\
E &\rightarrow n
\end{aligned}
$$

How pfxcalc works

The pfxcalc program's parser (`Parser` instance) builds a parse tree from the input

Each parse function will return a `Node` instance that is the root of a portion of the parse tree

Once the parse tree is complete, it interprets it directly to compute a result

parse member function

This is the entry point to the parser

```
Node *Parser::parse() {
  // U is the start symbol
  return parse_U();
}
```

parse_U member function

```
Node *Parser::parse_U() {
  std::unique_ptr<Node> u(new Node(NODE_U));

  // U -> ^ E ;
  // U -> ^ E ; U
  u->append_kid(parse_E());
  u->append_kid(expect(TOK_SEMICOLON));

  // U -> E ; ^
  // U -> E ; ^ U
  if (m_lexer->peek() != nullptr) {
    // there is more input, so the sequence of expressions continues
    u->append_kid(parse_U());
  }

  return u.release();
}
```

Things to note about parse_U

- There are two productions on U, but they both start with E, so `parse_E` is called unconditionally
- The `expect` member function consumes a specific token, reporting an error if the expected token is not available
- Comments indicate the productions that are viable, with a caret (`^`) indicating which part of the productions have been applied; this is *super* helpful for reasoning about what a parse function is doing
- After the semicolon is consumed, we're either done, or the second production needs to expand a U to continue recursively (if there are more prefix expressions)
  - The parser assumes that if it hasn't reached end of input, then there are more expressions

parse_E function

```
Node *Parser::parse_E() {
  // read the next terminal symbol
  Node *next_terminal = m_lexer->next();

  std::unique_ptr<Node> e(new Node(NODE_E));

  int tag = next_terminal->get_tag();
```

The function starts by consuming one token, and checking its tag (token kind)

Note that

1. `Lexer::next` throws an exception if the end of input is reached
2. reaching end of input is an error, because there is no epsilon production on E

parse_E function (continued)

```
if (tag == TOK_INTEGER_LITERAL || tag == TOK_IDENTIFIER) {
  // E -> <int_literal> ^
  // E -> <identifier> ^
  e->append_kid(next_terminal);
```

If the token was an integer literal (n) or identifier (i) then we've completed a production (integer literal or variable reference)

parse_E function (continued)

```
} else if (tag == TOK_ASSIGN) {
  // E -> = ^ <identifier> E
  e->append_kid(next_terminal);
  e->append_kid(expect(TOK_IDENTIFIER));
  e->append_kid(parse_E());
```

The assignment operator requires an identifier (naming the variable being assigned) followed by an expression (which computes the value being assigned)

parse_E function (continued)

```
} else if (tag == TOK_PLUS || tag == TOK_MINUS ||
           tag == TOK_TIMES || tag == TOK_DIVIDE) {
  // E -> + ^ E E
  // E -> - ^ E E
  // E -> * ^ E E
  // E -> / ^ E E
  e->append_kid(next_terminal);
  e->append_kid(parse_E()); // parse first operand
  e->append_kid(parse_E()); // parse second operand
```

The binary operators require two subexpressions (to compute the operand values)

parse_E function (continued)

```
  } else {
    SyntaxError::raise(next_terminal->get_loc(),
      "Illegal expression (at '%s')", next_terminal->get_str().c_str());
  }

  return e.release();
}
```

If no valid production was found, it is extremely important to report an error rather than continuing!

If a production was successfully applied, the parse node (root of the E subtree) is returned

Is it necessary for the parser to build a parse tree?

- Having the parser build a parse tree is not the only way to make the parser useful
- It could build an *abstract syntax tree* (more about this soon)
- It could do computations immediately, as the input is parsed

Why parse trees are useful

- Our interpreters and compilers will build full parse trees
- They represent the input exactly
- They are important evidence that the parser is working correctly
- They are very useful for debugging

Printing a parse tree

The pfxcalc program has a `treeprint` module for printing a textual representation of a tree

The −p option causes the program to print the parse tree of the input

Example shown on right

This is *very* useful for debugging

```
$ echo "= a 4; * a 5;" | ./pfxcalc -p
U
+--E
|   +--ASSIGN[=]
|   +--IDENTIFIER[a]
|   +--E
|       +--INTEGER_LITERAL[4]
+--SEMICOLON[;]
+--U
    +--E
    |   +--TIMES[*]
    |   +--E
    |   |   +--IDENTIFIER[a]
    |   +--E
    |       +--INTEGER_LITERAL[5]
    +--SEMICOLON[;]
```

Infix expressions

Infix expressions

Prefix expressions are fine, but mathematical notation traditionally uses *infix* notation, where the operator is between the operands

How do we handle these?

Infix expression grammar attempt 1

Grammar:

$$
\begin{aligned}
E &\rightarrow E + E \\
E &\rightarrow E \text{ - } E \\
E &\rightarrow E * E \\
E &\rightarrow E \,/\, E \\
E &\rightarrow i = E \\
E &\rightarrow i \\
E &\rightarrow n
\end{aligned}
$$

Once again, 'i' is an identifier and 'n' is an integer literal

Infix expression derivation

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Infix expression derivation

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Working string	Production
<u>E</u>	

Infix expression derivation

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Working string	Production
$\underline{E}$	$E \rightarrow E + E$
$\underline{E} + E$	

Infix expression derivation

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Working string	Production
$\underline{E}$	$E \rightarrow E + E$
$\underline{E} + E$	$E \rightarrow n$
$n + \underline{E}$	

Infix expression derivation

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Working string	Production
$\underline{E}$	$E \rightarrow E + E$
$\underline{E} + E$	$E \rightarrow n$
$n + \underline{E}$	$E \rightarrow E * E$
$n + \underline{E} * E$	

Infix expression derivation

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Working string	Production
$\underline{E}$	$E \to E + E$
$\underline{E} + E$	$E \to n$
$n + \underline{E}$	$E \to E * E$
$n + \underline{E} * E$	$E \to n$
$n + n * \underline{E}$	

Infix expression derivation

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

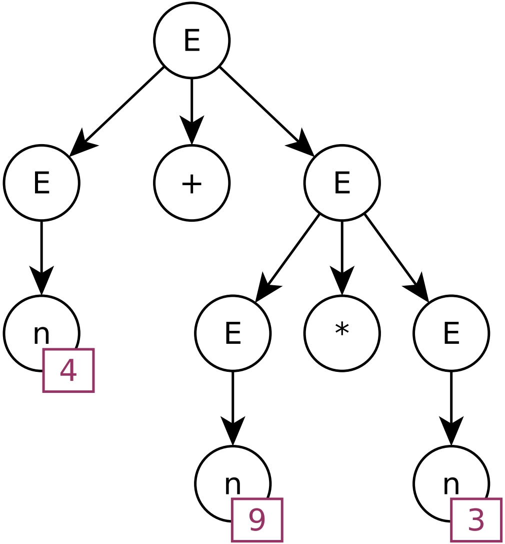
Working string	Production
$\underline{E}$	$E \rightarrow E + E$
$\underline{E} + E$	$E \rightarrow n$
$n + \underline{E}$	$E \rightarrow E * E$
$n + \underline{E} * E$	$E \rightarrow n$
$n + n * \underline{E}$	$E \rightarrow n$
$n + n * n$	

$\boxed{\text{Done!}}$

Parse tree

Derivation for $\boxed{4 + 9 * 3}$
(really, $\boxed{\mathsf{n} + \mathsf{n} * \mathsf{n}}$)

Working string	Production
$\underline{E}$	$E \to E + E$
$\underline{E} + E$	$E \to n$
$n + \underline{E}$	$E \to E * E$
$n + \underline{E} * E$	$E \to n$
$n + n * \underline{E}$	$E \to n$
$n + n * n$	



Infix expression derivation 2

Another derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Infix expression derivation 2

Another derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Working string	Production
<u>E</u>	

Infix expression derivation 2

Another derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Working string	Production
$\underline{E}$	$E \to E * E$
$\underline{E} * E$	

Infix expression derivation 2

Another derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Working string	Production
$\underline{E}$	$E \to E * E$
$\underline{E} * E$	$E \to E + E$
$\underline{E} + E * E$	

Infix expression derivation 2

Another derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Working string	Production
$\underline{E}$	$E \to E * E$
$\underline{E} * E$	$E \to E + E$
$\underline{E} + E * E$	$E \to n$
$n + \underline{E} * E$	

Infix expression derivation 2

Another derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

Working string	Production
$\underline{E}$	$E \to E * E$
$\underline{E} * E$	$E \to E + E$
$\underline{E} + E * E$	$E \to n$
$n + \underline{E} * E$	$E \to n$
$n + n * \underline{E}$	

Infix expression derivation 2

Another derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$)

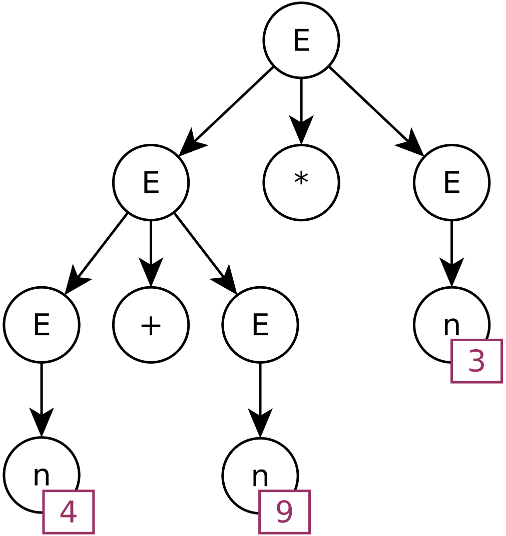
Working string	Production
$\underline{E}$	$E \to E * E$
$\underline{E} * E$	$E \to E + E$
$\underline{E} + E * E$	$E \to n$
$n + \underline{E} * E$	$E \to n$
$n + n * \underline{E}$	$E \to n$
$n + n * n$	

$\boxed{\text{Done!}}$

Parse tree 2

Derivation for $\boxed{4 + 9 * 3}$
(really, $\boxed{n + n * n}$)

Working string	Production
$\underline{E}$	$E \rightarrow E * E$
$\underline{E} * E$	$E \rightarrow E + E$
$\underline{E} + E * E$	$E \rightarrow n$
$n + \underline{E} * E$	$E \rightarrow n$
$n + n * \underline{E}$	$E \rightarrow n$
$n + n * n$	

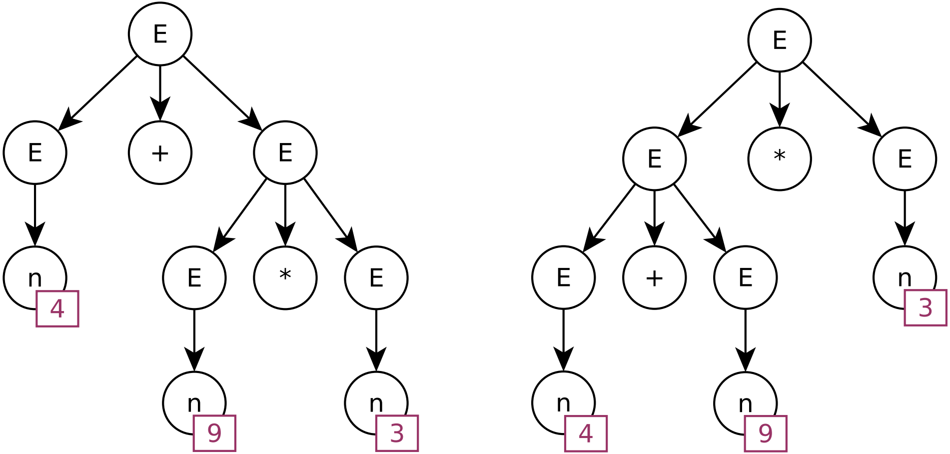


Ambiguity

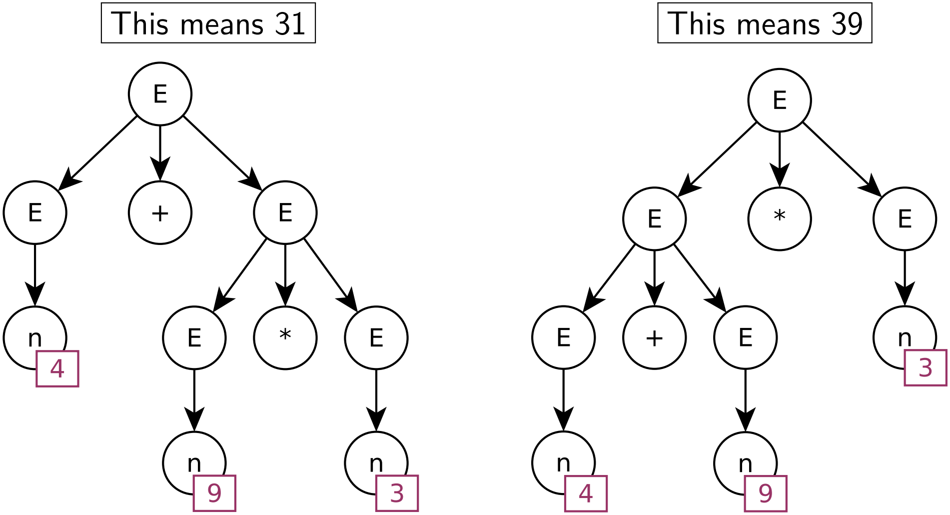
If a grammar can produce more than one parse tree for the same input string, it is *ambiguous*

If we want the parse tree structure to encode meaning, this is bad

Ambiguity leads to multiple meanings



Ambiguity leads to multiple meanings



Correctly parsing infix expressions

To parse infix expressions correctly, we need:

- Correct operator precedence
  - E.g., multiplication happens before addition
- Correct operator associativity
  - E.g., a - b - c means (a - b) - c, not a - (b - c)

Strategies:

- Represent different precedence levels using different nonterminals
- Left recursion yields left associativity, right recursion yields right associativity

A better infix expression grammar

Grammar (start symbol is A):

$A \to i = A$
$A \to E$
$E \to E + T$
$E \to E$ - T
$E \to T$
$T \to T * F$
$T \to T \,/\, F$
$T \to F$
$F \to i$
$F \to n$

Precedence levels:

Nonterminal	Precedence	Meaning	Operators	Associativity
A	lowest	Assignment	$=$	right
E		Expression	$+$ -	left
T		Term	$*$ $/$	left
F	highest	Factor		

Infix expression derivation 3

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Infix expression derivation 3

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Working string	Production
$\underline{A}$	

Infix expression derivation 3

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Working string	Production
$\underline{A}$	$A \to E$
$\underline{E}$	

Infix expression derivation 3

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow E + T$
$\underline{E} + T$	

Infix expression derivation 3

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow E + T$
$\underline{E} + T$	$E \rightarrow T$
$\underline{T} + T$	

Infix expression derivation 3

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow E + T$
$\underline{E} + T$	$E \rightarrow T$
$\underline{T} + T$	$T \rightarrow F$
$\underline{F} + T$	

Infix expression derivation 3

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow E + T$
$\underline{E} + T$	$E \rightarrow T$
$\underline{T} + T$	$T \rightarrow F$
$\underline{F} + T$	$F \rightarrow n$
$n + \underline{T}$	

Infix expression derivation 3

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow E + T$
$\underline{E} + T$	$E \rightarrow T$
$\underline{T} + T$	$T \rightarrow F$
$\underline{F} + T$	$F \rightarrow n$
$n + \underline{T}$	$T \rightarrow T * F$
$n + \underline{T} * F$	

Infix expression derivation 3

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow E + T$
$\underline{E} + T$	$E \rightarrow T$
$\underline{T} + T$	$T \rightarrow F$
$\underline{F} + T$	$F \rightarrow n$
$n + \underline{T}$	$T \rightarrow T * F$
$n + \underline{T} * F$	$T \rightarrow F$
$n + \underline{F} * F$	

Infix expression derivation 3

Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow E + T$
$\underline{E} + T$	$E \rightarrow T$
$\underline{T} + T$	$T \rightarrow F$
$\underline{F} + T$	$F \rightarrow n$
$n + \underline{T}$	$T \rightarrow T * F$
$n + \underline{T} * F$	$T \rightarrow F$
$n + \underline{F} * F$	$F \rightarrow n$
$n + n * \underline{F}$	

Infix expression derivation 3

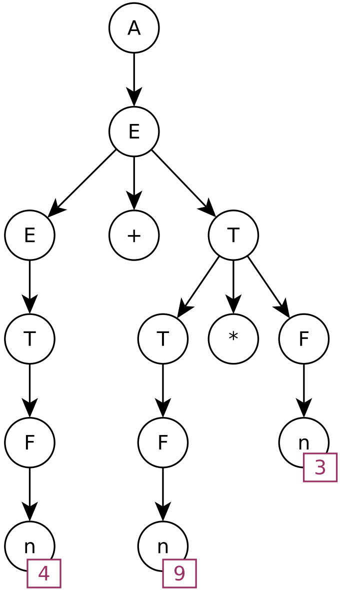
Derivation for $\boxed{4 + 9 * 3}$ (really, $\boxed{n + n * n}$) using improved grammar

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow E + T$
$\underline{E} + T$	$E \rightarrow T$
$\underline{T} + T$	$T \rightarrow F$
$\underline{F} + T$	$F \rightarrow n$
$n + \underline{T}$	$T \rightarrow T * F$
$n + \underline{T} * F$	$T \rightarrow F$
$n + \underline{F} * F$	$F \rightarrow n$
$n + n * \underline{F}$	$F \rightarrow n$
$n + n * n$	

$\boxed{\text{Done!}}$

Parse tree

Parse tree corresponding to the previous derivation:



Next time

- Limitations of recursive descent
- Precedence climbing