Lecture 3: Recursive descent limitations, precedence climbing

David Hovemeyer

September 6, 2023

601.428/628 Compilers and Interpreters

Today

- Limitations of recursive descent
- Precedence climbing
- Abstract syntax trees
- Supporting parenthesized expressions

Before we begin...

Assume a context-free grammar has the following productions on the nonterminal A:

$A \rightarrow b\ C$
$A \rightarrow d\ E$

(A, C, E are nonterminals; b, d are terminals)

What is the problem with the parse function shown on the right?

```
Node *Parser::parse_A() {
  Node *next_tok = m_lexer->peek();
  if (next_tok == nullptr) {
    SyntaxError::raise("Unexpected end of input");
  }

  std::unique_ptr<Node> a(new Node(NODE_A));
  int tag = next_tok->get_tag();
  if (tag == TOK_b) {
    a->append_kid(expect(TOK_b));
    a->append_kid(parse_C());
  } else if (tag == TOK_d) {
    a->append_kid(expect(TOK_d));
    a->append_kid(parse_E());
  }
  return a.release();
}
```

Limitations of recursive descent

Recall: 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		

Parsing infix expressions

Can we write a recursive descent parser for infix expressions using this grammar?

Parsing infix expressions

Can we write a recursive descent parser for infix expressions using this grammar?

No

Left recursion

Left-associative operators want to have left-recursive productions, but recursive descent parsers can't handle left recursion

Why?

Consider productions for the E nonterminal:

$E \rightarrow E + T$
$E \rightarrow E$ - T
$E \rightarrow T$

Left recursion (continued)

Imagine what the parse function for E would look like:

```
Node *Parser::parse_E() {
  std::unique_ptr<Node> e(new Node(NODE_E));

  if (some condition) {
    // apply E -> E + T production
    e->append_kid(parse_E());
    e->append_kid(expect(TOK_PLUS));
    e->append_kid(parse_T());
  }
```

Left recursion (continued)

Imagine what the parse function for E would look like:

```
Node *Parser::parse_E() {
  std::unique_ptr<Node> e(new Node(NODE_E));

  if (some condition) {
    // apply E -> E + T production
    e->append_kid(parse_E()); <--- problem
    e->append_kid(expect(TOK_PLUS));
    e->append_kid(parse_T());
  }
```

The problem with left recursion

Parse functions are recursive if there are recursive productions on the parse function's nonterminal symbol

In a left-recursive production such as $\boxed{E \rightarrow E + T}$, if the production is chosen,

- No tokens have been consumed prior to the recursive call to `parse_E()`
- So, in the recursive call, the lexer is in the same state as the original call to `parse_E()`

Thus, the recursive call will also attempt to choose the $\boxed{E \rightarrow E + T}$ production

The problem with left recursion

Parse functions are recursive if there are recursive productions on the parse function's nonterminal symbol

In a left-recursive production such as $\boxed{E \rightarrow E + T}$, if the production is chosen,

- No tokens have been consumed prior to the recursive call to `parse_E()`
- So, in the recursive call, the lexer is in the same state as the original call to `parse_E()`

Thus, the recursive call will also attempt to choose the $\boxed{E \rightarrow E + T}$ production

$$\boxed{\text{Infinite recursion}}$$

Observation

For any "reasonable" grammar, each nonterminal symbol must have (at least) one non-recursive production

- If all of the productions on a nonterminal are recursive, then there is no way to ever eliminate that nonterminal from the working string in a derivation

The non-recursive productions on a nonterminal are the base cases for the nonterminal's parse function

Eliminating left recursion

There is a trick for eliminating left recursion!

Given productions

$A \rightarrow A\ \alpha$
$A \rightarrow \beta$

These productions generate a β followed by 0 or more occurrences of α

We can rewrite these productions as

$A \rightarrow \beta\ A'$
$A' \rightarrow \alpha\ A'$
$A' \rightarrow \epsilon$ (note that ϵ designates the empty string)

without changing the language generated by the grammar

Revised infix expression grammar

Grammar (start symbol is A):

$A \rightarrow i = A$
$A \rightarrow E$
$E \rightarrow T\ E'$
$E' \rightarrow + T\ E'$
$E' \rightarrow -\ T\ E'$
$E' \rightarrow \epsilon$

$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T'$
$T' \rightarrow /\ F\ T'$
$T' \rightarrow \epsilon$
$F \rightarrow i$
$F \rightarrow n$

(Again, note that ϵ designates the empty string)

Example derivation

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

Working string	Production
A	

Example derivation

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

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

Example derivation

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

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow T\ E'$
$\underline{T}\ E'$	

Example derivation

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

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow T\ E'$
$\underline{T}\ E'$	$T \rightarrow F\ T'$
$\underline{F}\ T'\ E'$	

Example derivation

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

Working string	Production
$\underline{\mathsf{A}}$	$\mathsf{A} \to \mathsf{E}$
$\underline{\mathsf{E}}$	$\mathsf{E} \to \mathsf{T}\ \mathsf{E'}$
$\underline{\mathsf{T}}\ \mathsf{E'}$	$\mathsf{T} \to \mathsf{F}\ \mathsf{T'}$
$\underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	$\mathsf{F} \to \mathsf{n}$
$\mathsf{n}\ \underline{\mathsf{T'}}\ \mathsf{E'}$	

Example derivation

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

Working string	Production
$\underline{\mathsf{A}}$	$\mathsf{A} \to \mathsf{E}$
$\underline{\mathsf{E}}$	$\mathsf{E} \to \mathsf{T}\ \mathsf{E'}$
$\underline{\mathsf{T}}\ \mathsf{E'}$	$\mathsf{T} \to \mathsf{F}\ \mathsf{T'}$
$\underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	$\mathsf{F} \to \mathsf{n}$
$\mathsf{n}\ \underline{\mathsf{T'}}\ \mathsf{E'}$	$\mathsf{T'} \to \epsilon$
$\mathsf{n}\ \underline{\mathsf{E'}}$	

Example derivation

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

Working string	Production
$\underline{\mathsf{A}}$	$\mathsf{A} \to \mathsf{E}$
$\underline{\mathsf{E}}$	$\mathsf{E} \to \mathsf{T}\ \mathsf{E'}$
$\underline{\mathsf{T}}\ \mathsf{E'}$	$\mathsf{T} \to \mathsf{F}\ \mathsf{T'}$
$\underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	$\mathsf{F} \to \mathsf{n}$
$\mathsf{n}\ \underline{\mathsf{T'}}\ \mathsf{E'}$	$\mathsf{T'} \to \epsilon$
$\mathsf{n}\ \underline{\mathsf{E'}}$	$\mathsf{E'} \to +\ \mathsf{T}\ \mathsf{E'}$
$\mathsf{n} + \underline{\mathsf{T}}\ \mathsf{E'}$	

Example derivation

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

Working string	Production
$\underline{\mathsf{A}}$	$\mathsf{A} \to \mathsf{E}$
$\underline{\mathsf{E}}$	$\mathsf{E} \to \mathsf{T}\ \mathsf{E'}$
$\underline{\mathsf{T}}\ \mathsf{E'}$	$\mathsf{T} \to \mathsf{F}\ \mathsf{T'}$
$\underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	$\mathsf{F} \to \mathsf{n}$
$\mathsf{n}\ \underline{\mathsf{T'}}\ \mathsf{E'}$	$\mathsf{T'} \to \epsilon$
$\mathsf{n}\ \underline{\mathsf{E'}}$	$\mathsf{E'} \to +\ \mathsf{T}\ \mathsf{E'}$
$\mathsf{n} + \underline{\mathsf{T}}\ \mathsf{E'}$	$\mathsf{T} \to \mathsf{F}\ \mathsf{T'}$
$\mathsf{n} + \underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	

Example derivation

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

Working string	Production
$\underline{\mathrm{A}}$	$\mathrm{A} \to \mathrm{E}$
$\underline{\mathrm{E}}$	$\mathrm{E} \to \mathrm{T}\ \mathrm{E'}$
$\underline{\mathrm{T}}\ \mathrm{E'}$	$\mathrm{T} \to \mathrm{F}\ \mathrm{T'}$
$\underline{\mathrm{F}}\ \mathrm{T'}\ \mathrm{E'}$	$\mathrm{F} \to \mathrm{n}$
$\mathrm{n}\ \underline{\mathrm{T'}}\ \mathrm{E'}$	$\mathrm{T'} \to \epsilon$
$\mathrm{n}\ \underline{\mathrm{E'}}$	$\mathrm{E'} \to +\ \mathrm{T}\ \mathrm{E'}$
$\mathrm{n} + \underline{\mathrm{T}}\ \mathrm{E'}$	$\mathrm{T} \to \mathrm{F}\ \mathrm{T'}$
$\mathrm{n} + \underline{\mathrm{F}}\ \mathrm{T'}\ \mathrm{E'}$	$\mathrm{F} \to \mathrm{n}$
$\mathrm{n} + \mathrm{n}\ \underline{\mathrm{T'}}\ \mathrm{E'}$	

Example derivation

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

Working string	Production
$\underline{\mathsf{A}}$	$\mathsf{A} \to \mathsf{E}$
$\underline{\mathsf{E}}$	$\mathsf{E} \to \mathsf{T}\ \mathsf{E'}$
$\underline{\mathsf{T}}\ \mathsf{E'}$	$\mathsf{T} \to \mathsf{F}\ \mathsf{T'}$
$\underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	$\mathsf{F} \to \mathsf{n}$
$\mathsf{n}\ \underline{\mathsf{T'}}\ \mathsf{E'}$	$\mathsf{T'} \to \epsilon$
$\mathsf{n}\ \underline{\mathsf{E'}}$	$\mathsf{E'} \to +\ \mathsf{T}\ \mathsf{E'}$
$\mathsf{n} + \underline{\mathsf{T}}\ \mathsf{E'}$	$\mathsf{T} \to \mathsf{F}\ \mathsf{T'}$
$\mathsf{n} + \underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	$\mathsf{F} \to \mathsf{n}$
$\mathsf{n} + \mathsf{n}\ \underline{\mathsf{T'}}\ \mathsf{E'}$	$\mathsf{T'} \to *\ \mathsf{F}\ \mathsf{T'}$
$\mathsf{n} + \mathsf{n} * \underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	

Example derivation

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

Working string	Production
$\underline{A}$	$A \to E$
$\underline{E}$	$E \to T\ E'$
$\underline{T}\ E'$	$T \to F\ T'$
$\underline{F}\ T'\ E'$	$F \to n$
$n\ \underline{T'}\ E'$	$T' \to \epsilon$
$n\ \underline{E'}$	$E' \to +\ T\ E'$
$n + \underline{T}\ E'$	$T \to F\ T'$
$n + \underline{F}\ T'\ E'$	$F \to n$
$n + n\ \underline{T'}\ E'$	$T' \to *\ F\ T'$
$n + n * \underline{F}\ T'\ E'$	$F \to n$
$n + n * n\ \underline{T'}\ E'$	

Example derivation

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

Working string	Production
$\underline{A}$	$A \to E$
$\underline{E}$	$E \to T\ E'$
$\underline{T}\ E'$	$T \to F\ T'$
$\underline{F}\ T'\ E'$	$F \to n$
$n\ \underline{T'}\ E'$	$T' \to \epsilon$
$n\ \underline{E'}$	$E' \to +\ T\ E'$
$n + \underline{T}\ E'$	$T \to F\ T'$
$n + \underline{F}\ T'\ E'$	$F \to n$
$n + n\ \underline{T'}\ E'$	$T' \to *\ F\ T'$
$n + n * \underline{F}\ T'\ E'$	$F \to n$
$n + n * n\ \underline{T'}\ E'$	$T' \to \epsilon$
$n + n * n\ \underline{E'}$	

Example derivation

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

Working string	Production
$\underline{A}$	$A \to E$
$\underline{E}$	$E \to T\ E'$
$\underline{T}\ E'$	$T \to F\ T'$
$\underline{F}\ T'\ E'$	$F \to n$
$n\ \underline{T'}\ E'$	$T' \to \epsilon$
$n\ \underline{E'}$	$E' \to +\ T\ E'$
$n + \underline{T}\ E'$	$T \to F\ T'$
$n + \underline{F}\ T'\ E'$	$F \to n$
$n + n\ \underline{T'}\ E'$	$T' \to *\ F\ T'$
$n + n * \underline{F}\ T'\ E'$	$F \to n$
$n + n * n\ \underline{T'}\ E'$	$T' \to \epsilon$
$n + n * n\ \underline{E'}$	$E' \to \epsilon$
$n + n * n$	

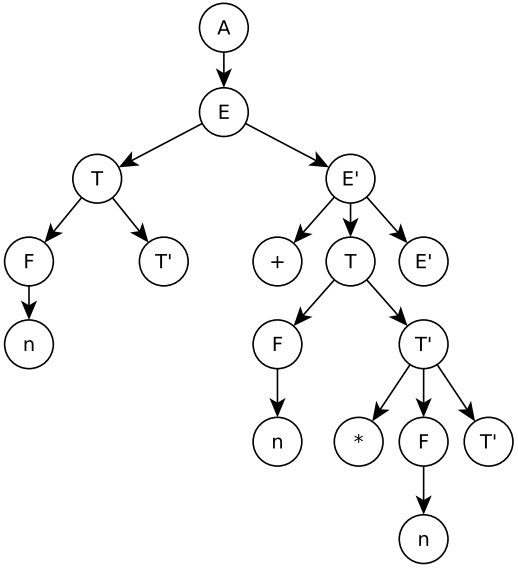
Done!

Example parse tree

Working string	Production
$\underline{\mathsf{A}}$	$\mathsf{A} \to \mathsf{E}$
$\underline{\mathsf{E}}$	$\mathsf{E} \to \mathsf{T}\ \mathsf{E'}$
$\underline{\mathsf{T}}\ \mathsf{E'}$	$\mathsf{T} \to \mathsf{F}\ \mathsf{T'}$
$\underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	$\mathsf{F} \to \mathsf{n}$
$\mathsf{n}\ \underline{\mathsf{T'}}\ \mathsf{E'}$	$\mathsf{T'} \to \epsilon$
$\mathsf{n}\ \underline{\mathsf{E'}}$	$\mathsf{E'} \to +\ \mathsf{T}\ \mathsf{E'}$
$\mathsf{n} + \underline{\mathsf{T}}\ \mathsf{E'}$	$\mathsf{T} \to \mathsf{F}\ \mathsf{T'}$
$\mathsf{n} + \underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	$\mathsf{F} \to \mathsf{n}$
$\mathsf{n} + \mathsf{n}\ \underline{\mathsf{T'}}\ \mathsf{E'}$	$\mathsf{T'} \to *\ \mathsf{F}\ \mathsf{T'}$
$\mathsf{n} + \mathsf{n} * \underline{\mathsf{F}}\ \mathsf{T'}\ \mathsf{E'}$	$\mathsf{F} \to \mathsf{n}$
$\mathsf{n} + \mathsf{n} * \mathsf{n}\ \underline{\mathsf{T'}}\ \mathsf{E'}$	$\mathsf{T'} \to \epsilon$
$\mathsf{n} + \mathsf{n} * \mathsf{n}\ \underline{\mathsf{E'}}$	$\mathsf{E'} \to \epsilon$
$\mathsf{n} + \mathsf{n} * \mathsf{n}$	

Example parse tree

Working string	Production
$\underline{A}$	$A \rightarrow E$
$\underline{E}$	$E \rightarrow T\ E'$
$\underline{T}\ E'$	$T \rightarrow F\ T'$
$\underline{F}\ T'\ E'$	$F \rightarrow n$
$n\ \underline{T'}\ E'$	$T' \rightarrow \epsilon$
$n\ \underline{E'}$	$E' \rightarrow +\ T\ E'$
$n + \underline{T}\ E'$	$T \rightarrow F\ T'$
$n + \underline{F}\ T'\ E'$	$F \rightarrow n$
$n + n\ \underline{T'}\ E'$	$T' \rightarrow *\ F\ T'$
$n + n * \underline{F}\ T'\ E'$	$F \rightarrow n$
$n + n * n\ \underline{T'}\ E'$	$T' \rightarrow \epsilon$
$n + n * n\ \underline{E'}$	$E' \rightarrow \epsilon$
$n + n * n$	



Observations

- Operator precedence and associativity are preserved
- Operators end up in strange places in the parse tree
  - Makes the parse tree a bit difficult to reason about

Precedence climbing

Another observation

After applying the left recursion elimination refactoring to the productions for additive (+, -) operators:

Productions:
$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow -\ T\ E'$$
$$E' \rightarrow \epsilon$$

What this means:

Another observation

After applying the left recursion elimination refactoring to the productions for additive (+, -) operators:

Productions:
$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow -\ T\ E'$$
$$E' \rightarrow \epsilon$$

What this means:

An additive expression (E) is

- a single *term* (T, expression with only multiplicative or higher-precedence operators),
- followed by 0 or more pairs of additive operator and term

Generalizing infix expressions

All infix expressions have the form (X=operand, $\oplus$=operator)

X

$X \oplus X$

$X \oplus X \oplus X$

etc...

Generalizing infix expressions

All infix expressions have the form (X=operand, $\oplus$=operator)

X

$X \oplus X$

$X \oplus X \oplus X$

etc...

Rather than using a general purpose parsing technique (recursive descent following the productions of a context-free grammar) to parse infix expressions, we could use a *specialized* parsing algorithm optimized for the structure of infix expressions

Precedence climbing

Precedence climbing is a specialized algorithm for parsing infix expressions:

- Arbitrary operators and precedence levels
- Arbitrary associativity (left and right)

One very nice feature of recursive descent parsing is that it is easy to embed specialized parsing algorithms (such as precedence climbing)

I.e., when the recursive descent parser needs to parse an infix expression, it invokes the precedence climbing parser

- Precedence climbing parser returns control after it has parsed one expression

Precedence climbing algorithm

```
parse_expression()
    return parse_expression_1(parse_primary(), 0)
```

Source: https://en.wikipedia.org/wiki/Operator-precedence_parser

Precedence climbing algorithm

```
parse_expression_1(lhs, min_precedence)
    lookahead := peek next token
    while lookahead is a binary operator whose precedence is >= min_precedence
        op := lookahead
        advance to next token
        rhs := parse_primary ()
        lookahead := peek next token
        while lookahead is a binary operator whose precedence is greater
                 than op's, or a right-associative operator
                 whose precedence is equal to op's
            rhs := parse_expression_1 (rhs, lookahead's precedence)
            lookahead := peek next token
        lhs := the result of applying op with operands lhs and rhs
    return lhs
```

Source: https://en.wikipedia.org/wiki/Operator-precedence_parser

Precedence climbing theory of operation

Assume that the goal is to build a tree to represent the parsed expression

- `parse_primary` parses a *primary expression* (i.e., an operand)
  - In our grammar, an occurrence of the F nonterminal is a primary expression
- Recursive calls handle higher-precedence operators, or right-associative operators at the same precedence level
  - These will create subtrees
- In the case of a series of left-associative operators at the same precedence level, tree will grow to the left

Precedence climbing in practice

Precedence climbing (like recursive descent) is a predictive parsing algorithm, so is guided by the lexer

Main concern is to determine whether the lookahead token is an operator

- Also, if a token is an operator, what is its precedence and associativity

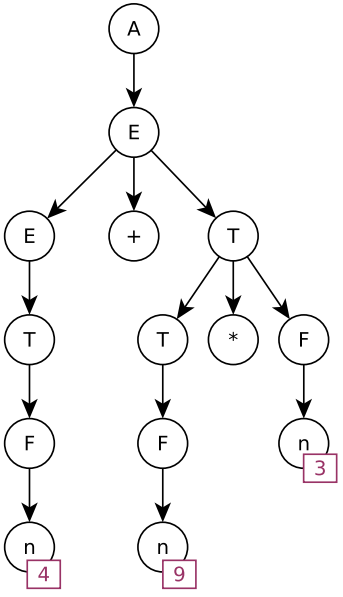
Parsing languages with infix expressions

- Many programming languages use infix expressions for expressing numeric computations
- One approach (which tends to work well in practice):
  - Use a context-free grammar to describe all syntactic constructs *other than* infix expressions
  - Avoid left recursion
  - Use recursive descent for parsing all constructs other than infix expressions
  - Use precedence climbing for infix expressions

Abstract syntax trees

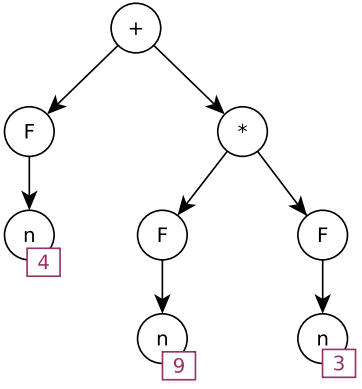
Expression trees

Parse tree for $\boxed{4 + 9 * 3}$ using original context free grammar:



Expression trees

Parse tree for `4 + 9 * 3` using precedence climbing (we're assuming that primary expressions are parsed using the parse function for the F nonterminal):



Abstract syntax trees

- Precedence climbing produces trees in which
  - leaf nodes are primary expressions
  - interior nodes are operators
- This type of representation is called an *abstract syntax tree*, or "AST"
- This is a useful representation of code!
  - ASTs are essentially parse trees where all unnecessary constructs have been eliminated

Creating ASTs

- Precedence climbing automatically produces an AST-like representation
- For more general parsers:
  - Parse tree could be transformed into an AST
  - The parser could create an AST directly, skipping the creation of a full parse tree

Supporting parenthesized expressions

Parenthesized expressions

- For languages using infix expressions, we will want to allow parentheses to explicitly force the order of evaluation
- E.g., in $\boxed{(4 + 9)\ *\ 3}$, the $+$ is done before the *
- This is super easy to support!
- Idea: a parenthesized expression is a kind of *primary expression*

Productions for primary expressions

In our example grammar, the F nonterminal ("factor") represents a primary expression

Productions: $F \rightarrow i$
$F \rightarrow n$

Add an additional production:

$$F \rightarrow (E)$$

Note that E is an arbitrary infix expression