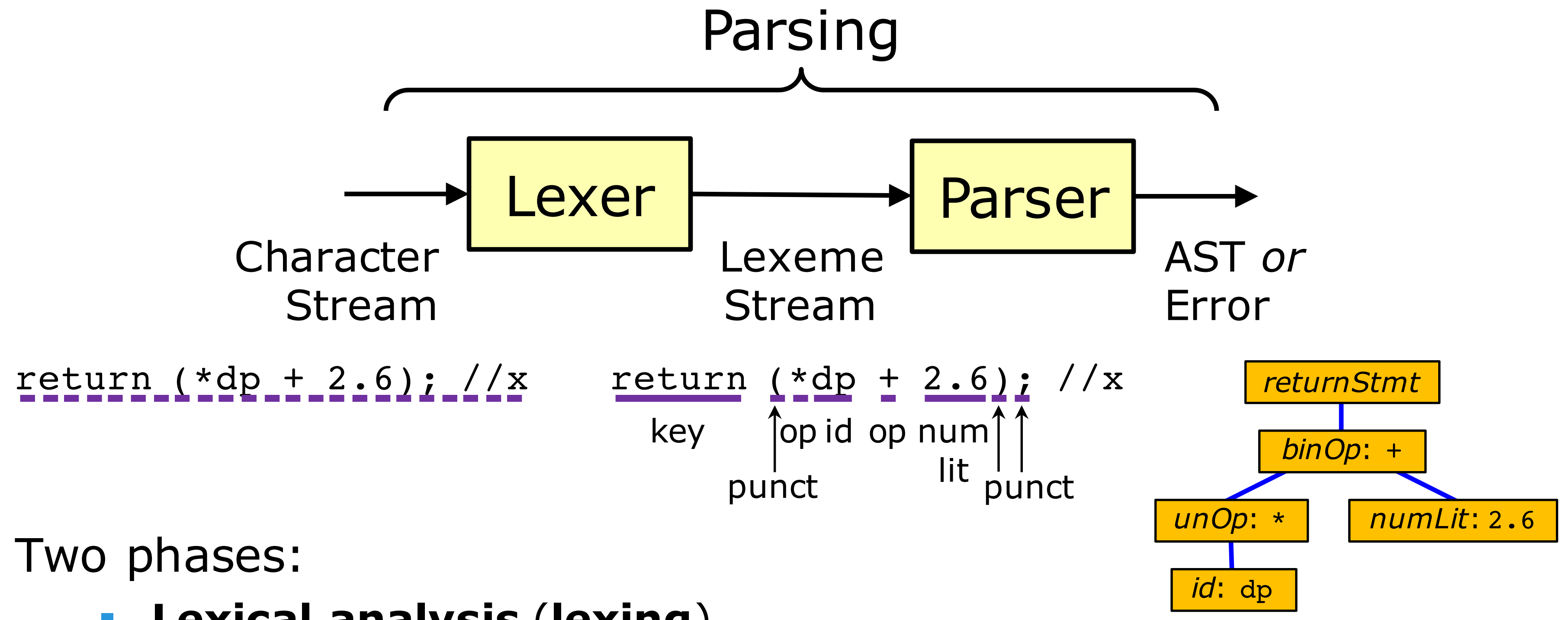
Recursive-Descent Parsing continued

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Wednesday, February 12, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Review

Review
Overview of Lexing & Parsing



Two phases:

- **Lexical analysis** (**lexing**)
- **Syntax analysis** (**parsing**)

The output of a parser is typically an **abstract syntax tree** (**AST**). Specifications of these will vary.

The Basics of Syntax Analysis — Introduction

Syntax analysis is usually based on a context-free grammar (CFG).

Recall the notion of a **derivation**: begin with the start symbol, apply productions one by one, ending with a string of terminals.

CFG (start symbol: *item*)

item → "`(`" *item* "`)`"

item → *thing*

thing → ID

thing → "`%`"

Here, ID is a lexeme category. On the right, the actual string might be something like "`((abc_39))`".

Derivation

<u>*item*</u>

`(`<u>*item*</u>`)`

`((`<u>*item*</u>`))`

`((`<u>*thing*</u>`))`

`((`ID`))`

There are many different parsing methods that are based on CFGs. All will go through the steps necessary to produce a derivation; however, they typically will not store or output this derivation.

Review
The Basics of Syntax Analysis — Categories of Parsers

Parsing methods can be divided into two broad categories.

Top-Down Parsers

- Go through derivation top to bottom, **expanding** nonterminals.
- Important subcategory: **LL parsers** (read input **L**eft-to-right, produce **L**eftmost derivation).
- Often hand-coded—but not always.
- Method we look at: **Predictive Recursive Descent**.

Bottom-Up Parsers

- Go through the derivation bottom to top, **reducing** substrings to nonterminals.
- Important subcategory: **LR parsers** (read input **L**eft-to-right, produce **R**ightmost derivation).
- Almost always automatically generated.
- Method we look at: **Shift-Reduce**.

Review
The Basics of Syntax Analysis — Categories of Grammars [1/2]

As a rule, a fast parsing method is *not* capable of handling all CFGs. For each category of parsing methods (LL, LR, etc.), there is an associated category of grammars that such methods can handle.

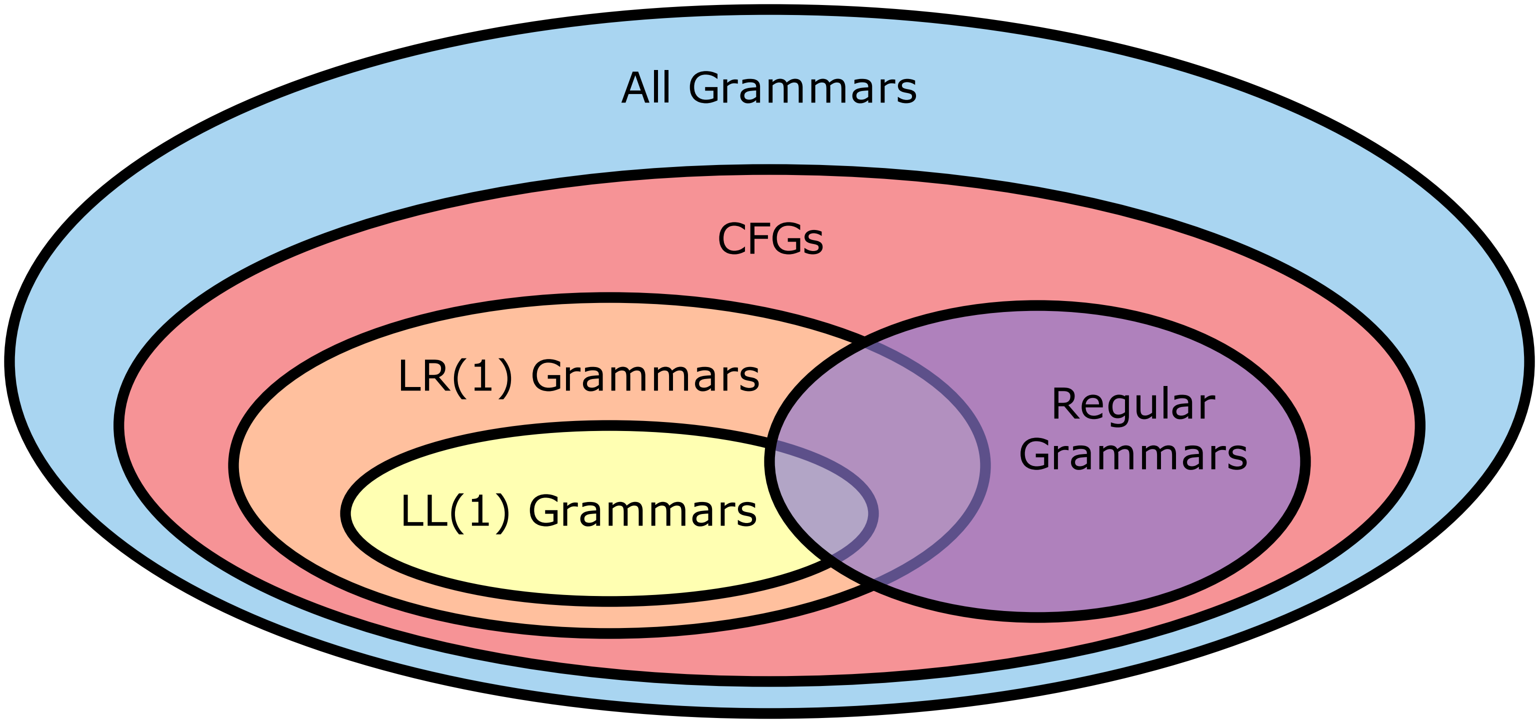
The grammars that an LL parser can handle, if *k* upcoming input symbols (lexemes?) are used to make each decision, are called **LL(*k*) grammars**. Similarly, **LR(*k*) grammars**.

Curiously, every LL(1) grammar is an LR(1) grammar, while there are LR(1) grammars that are not LL(1) grammars. So LR parsers are more general.

Note, however, that when we write a compiler or interpreter for a programming language, we only need *one* grammar, and if it is an LL(1) grammar, then an LL parser works fine.

The Basics of Syntax Analysis — Categories of Grammars [2/2]

This following diagram shows the relationship between various categories of grammars.



Review
Recursive-Descent Parsing — Introduction, How It Works

Our first parsing method: **Recursive Descent**.

- A top-down parsing method.
- In the LL category, if *backtracking* is not done.
- Almost always hand-coded.
- Has been known for decades. Still in common use.

Writing a Recursive-Descent parser:

- There is one parsing function for each nonterminal.
- A parsing function is responsible for parsing all strings that its nonterminal can be expanded into.
- Code for a parsing function is a translation of the right-hand side of the production for its nonterminal.

Backtracking after choosing the wrong production is too slow. A **predictive** parser must always choose the right production the first time. This restricts the grammars we can use: LL(k) grammars. If we do not do lookahead: LL(1) grammars.

Review
Recursive-Descent Parsing — Example #1: Simple [1/2]

We wrote a Predictive Recursive-Descent parser based on Grammar 1, below, whose start symbol is *item*. We began by combining productions with a common left-hand side.

Grammar 1

item → "`(`" *item* "`)`"
item → *thing*
thing → ID
thing → "`%`"

Grammar 1a

item → "`(`" *item* "`)`"
  | *thing*
thing → ID
  | "`%`"

Our parser does not generate an AST. Return values are Booleans.

See `rdparser1.lua`.

Recursive-Descent Parsing — Example #1: Simple [2/2]

Parsing-function code is a translation of the right-hand side of the production for the nonterminal.

```
function parse_item()
    if matchString("(") then
        if not parse_item() then
            return false
        end
        if not matchString(")") then
            return false
        end
        -- We would construct an AST here
        return true
    elseif parse_thing() then
        -- We would construct an AST here
        return true
    else
        return false
    end
end
```

A nonterminal in the right-hand side becomes a call to its parsing function.

Grammar 1a

item → "(" *item* ")"
  | *thing*

thing → ID
  | "%"

We do not backtrack. If we call a parsing function, and it returns `false`, then this function must return `false`.

A terminal in the right-hand side becomes a check that the input string contains the proper lexeme.

Review
Recursive-Descent Parsing — Handling Incorrect Input [1/3]

As originally written, our parser said that these are both syntactically correct:

- `((x)))`
- `a,b,c`

Clearly, they are not. Why did the parser say that?

Grammar 1a

item	→	"`(`" *item* "`)`"
	|	*thing*
thing	→	ID
	|	"`%`"

A parsing function stops when it has successfully parsed a string of the kind it is aimed at. From a parsing function's point of view, the above are correct strings with extra characters at the end.

Each parsing function is doing its job, but the parser is not giving us the information we need. What can we do about this?

Review
Recursive-Descent Parsing — Handling Incorrect Input [2/3]

One solution is to add another lexeme category: **end of input**. Standard notation for this: $. Then add a new start symbol, and augment the grammar with one more production, of the form *newStartSymbol* → *oldStartSymbol* $.

The following would be our new grammar, with start symbol *all*.

Grammar 1b

all	→	*item* $
item	→	"`(`" *item* "`)`"
	|	*thing*
thing	→	ID
	|	"`%`"

This idea was *not* used in our parser.

Review
Recursive-Descent Parsing — Handling Incorrect Input [3/3]

Another solution is to add an extra check at the end of parsing, to see whether all lexemes have been processed. The grammar is unchanged, and the parsing functions are the same.

A correct parse of the entire input then requires two conditions:

- The parsing function for the start symbol indicates a correct parse.
- All lexemes have been read.

The above solution works better with the interactive environment that you will use with your interpreter. So I will be using this solution in all of our Recursive-Descent parsers.

Our parser now implements the above idea. The program that uses it has been modified accordingly.

See `rdparser1.lua` & `userdparser1.lua`.

Recursive-Descent Parsing

continued

Recursive-Descent Parsing
Example #2: More Complex [1/2]

Now let's write a parser for the following more complex grammar, whose start symbol is still *item*.

Grammar 2

item → "`(`" *item* "`)`"
  | *thing*
thing → ID { ("`,`" | "`:`") ID }
  | "`%`"
  | ["`*`" "`-`"] "`[`" *item* "`]`"

Recall:

Braces mean *optional, repeatable* (0 or more).

Brackets mean *optional* (0 or 1).

Note the difference:

["`[`"

All strings in the old language are also in the new language. But Grammar 2 also generates strings like these:

- `((a,b,c:d))`
- `((*-[([%])]))`

Recursive-Descent Parsing
Example #2: More Complex [2/2]

Grammar 2

item → "(" *item* ")" | *thing*

thing → ID { ("," | ":") ID } | "%" | ["*" "-"] "[" *item* "]"

In a parsing function:

[…] Brackets (optional: 0 or 1) become a *conditional* (if).

- Check for the possible initial lexemes inside the brackets. If found, parse everything inside the brackets. Otherwise skip the brackets.

{ … } Braces (optional, repeatable: 0 or more) become a *loop*.

- Loop body: Check for the possible initial lexemes inside the braces. If not found, then exit the loop, moving to just after the braces. If found, parse everything inside the braces, and then REPEAT.

TO DO

- Write a Predictive Recursive-Descent parser based on Grammar 2.

Done. See `rdparser2.lua`*.*

Recursive-Descent Parsing
Example #3: Expressions [1/5]

Now we raise our standards. We wish to parse arithmetic expressions in their usual form, with variables, numeric literals, binary `+`, `–`, `*`, and `/` operators, and parentheses. When given a syntactically correct expression, our parser should return an **abstract syntax tree** (**AST**).

All operators will be binary and left-associative: "`a + b + c`" means "`(a + b) + c`".

Precedence will be as usual: "`a + b * c`" means "`a + (b * c)`".

These may be overridden using parentheses: "`(a + b) * c`".

Due to the limitations of our in-class lexer, the expression "`k–4`" will need to be written as "`k – 4`".

Recursive-Descent Parsing
Example #3: Expressions [2/5]

We begin with the following grammar, with start symbol *expr*.

Grammar 3

expr → *term*
  | *expr* ("+" | "–") *term*
term → *factor*
  | *term* ("*" | "/") *factor*
factor → ID
  | NUMLIT
  | "(" *expr* ")"

Recall: an **expression** is something that has a value.

When several things are added, each is a **term**.

Three terms:

37 – (3+*x*) + 2**x***y*

When several things are multiplied, each is a **factor**.

Three factors:

42(3+*x*)(7–2**x*)

Grammar 3 encodes our associativity and precedence rules, and it allows us to use parentheses to override them.

Recursive-Descent Parsing
Example #3: Expressions [3/5]

To the right is part of a parsing function for nonterminal *expr*.

Grammar 3

expr → *term*
  | *expr* ("+" | "–") *term*
term → *factor*
  | *term* ("*" | "/") *factor*
factor → ID
  | NUMLIT
  | "(" *expr* ")"

```
function parse_expr()
    if parse_term() then
        …  -- Construct AST
        return true
    elseif parse_expr() then
        …
```

But something is wrong with this code. *See the next slide.*

Recursive-Descent Parsing
Example #3: Expressions [4/5]

```
function parse_expr()
    if parse_term() then
        …  -- Construct AST
        return true
    elseif parse_expr() then
        …
```

What is wrong with this code?

- First, if the call to `parse_term` returns false, then the position in the input may have changed. Fixing this requires backtracking, which we do not do, so our code is incorrect.
- But even if we do backtrack, there is another problem. Suppose `parse_expr` is called with input that does not begin with a valid *term*. What happens? Answer: infinite recursion!

Recursive-Descent Parsing
Example #3: Expressions [5/5]

In fact, it is *impossible* to write a Predictive Recursive-Descent parser based on Grammar 3. It is not an LL(1) grammar—in fact, it is not LL(k) for any k.

Grammar 3

expr → *term*
  | *expr* ("+" | "–") *term*
term → *factor*
  | *term* ("*" | "/") *factor*
factor → ID
  | NUMLIT
  | "(" *expr* ")"

Next we look at what it means to be an LL(1) grammar. We will then return to the expression-parsing problem.

Recursive-Descent Parsing
LL(1) Grammars [1/8]

An **LL(1) grammar** is a CFG that can be handled by an LL parsing method—such as Predictive Recursive Descent—without lookahead.

Recall the origin of the name: these parsers handle their input in a **L**eft-to-right order, and they go through the steps required to generate a **L**eftmost derivation.

Now we look at some of the properties that an LL(1) grammar must have.

Recursive-Descent Parsing
LL(1) Grammars [2/8]

Consider the following grammar.

Grammar A

xx → *xx* "+" "b" | "a"

A parsing function would begin:

```
function parse_xx()
    if parse_xx() then
        …
```

We have recursion without a base-case check.

The trouble lies in the grammar. The right-hand side of the production for *xx* begins with *xx*. This is **left recursion**. It is not allowed in an LL(1) grammar.

Left recursion can be more subtle. Below is a variation on Grammar A.

Grammar A′

$xx \rightarrow yy$ “`b`” | “`a`”

$yy \rightarrow xx$ “`+`”

Grammar A′ also contains left recursion. It is not LL(1).

The grammar below illustrates another problem.

Grammar B

xx → "a" *yy* | "a" *zz*

yy → "*"

zz → "/"

If we do not use lookahead, then we cannot even begin to write a Recursive-Descent parser for Grammar B. How would the code for function `parse_xx` start? Should it take the first or second option? There is no way to tell, without lookahead.

We say the first production in Grammar B is not **left-factored**. An LL(1) grammar can only contain left-factored productions.

Recursive-Descent Parsing
LL(1) Grammars [5/8]

Here is another problematic grammar.

Grammar C

xx → *yy* | *zz*

yy → “” | “a”

zz → “” | “b”

In Grammar C, the empty string can be derived from either *yy* or *zz*. So if we are expanding *xx*, and there is no more input, then there is no basis for deciding between *yy* and *zz*.

One last non-LL(1) grammar.

Grammar D

xx → *yy* "`a`"

yy → "`a`" | ""

The language generated by Grammar D contains two strings: "`a`" and "`aa`". But imagine a Recursive-Descent parser based on Grammar D, attempting to parse these strings. What would happen?

Recursive-Descent Parsing
LL(1) Grammars [7/8]

It turns out that the problems presented by Grammars A–D illustrate *all* the reasons a CFG might not be an LL(1) grammar.

Fact.* Suppose that a context-free grammar G has the following three properties.

I do *not* expect you to memorize this.

1. If $A \to \alpha$ and $A \to \beta$ are productions in G, then there do *not* exist two strings, one derived from α, the other derived from β, that begin with the same (terminal) symbol.
2. If $A \to \alpha$ and $A \to \beta$ are productions in G, then it is *not* the case that the empty string can be derived from both α and β.
3. If $A \to \alpha$ and $A \to \beta$ are productions in G, and the empty string can be derived from β, then there is no (terminal) symbol x that begins a string that can be derived from α, such that x can follow a string derived from A.

Then Grammar G is an LL(1) grammar.

(1) does not hold for Grammars A, A′ and B; (2) does not hold for Grammar C; and (3) does not hold for Grammar D.

*Adapted from A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, 1986 (“The Dragon Book”), p. 192.

Recursive-Descent Parsing
LL(1) Grammars [8/8]

In addition:

Fact. Suppose that G is an LL(1) grammar. Then G is not ambiguous.

Now suppose—as in our expression-parsing example—that we wish to write a Predictive Recursive-Descent parser, but our grammar is not LL(1). What can we do about this?

Recursive-Descent Parsing
Transforming Grammars [1/5]

If a grammar is not LL(1), this does not mean that the grammar is completely useless as a basis for a Recursive-Descent parser. We *might* be able to transform the grammar into an LL(1) grammar that generates the same language.

For example, here is Grammar A, which is not LL(1), along with an LL(1) grammar that generates the same language.

Grammar A

$xx \rightarrow xx$ "+" "b" | "a"

Grammar Aa

$xx \rightarrow$ "a" yy

$yy \rightarrow$ "+" "b" yy | ""

In practice, we might use braces to make this grammar more concise.

Transforming Grammars [2/5]

Grammar B, which is not LL(1), along with an LL(1) grammar that generates the same language.

Grammar B

xx → “a” *yy* | “a” *zz*

yy → “*”

zz → “/”

Grammar Ba

xx → “a” *yy*

yy → “*” | “/”

Recursive-Descent Parsing
Transforming Grammars [3/5]

Grammar C, which is not LL(1), along with an LL(1) grammar that generates the same language.

Grammar C

xx → *yy* | *zz*

yy → “” | “a”

zz → “” | “b”

Grammar Ca

xx → *yy* | *zz* | “”

yy → “a”

zz → “b”

Recursive-Descent Parsing
Transforming Grammars [4/5]

And Grammar D, which is not LL(1), along with an LL(1) grammar that generates the same language.

Grammar D

xx → *yy* "a"

yy → "a" | ""

Grammar Da

xx → "a" *yy*

yy → "a" | ""

Recursive-Descent Parsing
Transforming Grammars [5/5]

It is not at all uncommon to be faced with a grammar that is not LL(1), but that can be transformed easily to one that is LL(1). In particular, this is common in the specification of programming-language syntax.

Note, however, that there are context-free languages that cannot be generated by any LL(1) grammar—or LL(k) grammar—at all.

Back to Example #3: Expressions — Left-Associativity [1/5]

Now we return to our expression grammar. It is given below. Recall that this is not an LL(1) grammar.

Grammar 3

expr → *term*
| *expr* ("+" | "–") *term*
term → *factor*
| *term* ("*" | "/") *factor*
factor → ID
| NUMLIT
| "(" *expr* ")"

More generally, the natural grammar for expressions involving left-associative binary operators is not LL(1); it is, in fact, not LL(k) for any k.

An easy fix is to reorder the operands; for example,
expr ("+" | "–") *term* becomes *term* ("+" | "–") *expr*.

I will also use brackets ([…]) to make the grammar more concise.

Here is the result. This is an LL(1) grammar.

Grammar 3a

expr → *term* [("+" | "–") *expr*]
term → *factor* [("*" | "/") *term*]
factor → ID
| NUMLIT
| "(" *expr* ")"

But now we have a new problem. *See the next slide …*

Grammar 3a

expr → *term* [("+" | "-") *expr*]

term → *factor* [("*" | "/") *term*]

factor → ID

| NUMLIT

| "(" *expr* ")"

Grammar 3a is LL(1), but it encodes *right-associative* binary operators. We want our operators to be left-associative.

Left-associative: $\overbrace{\texttt{a + b}}^{expr} + \overbrace{\texttt{c}}^{term}$

Right-associative: $\underbrace{\texttt{a}}_{term} + \underbrace{\texttt{b + c}}_{expr}$

Recursive-Descent Parsing
Back to Example #3: Expressions — Left-Associativity [4/5]

Fortunately, all is not lost. Here is an idea that works.

Start with a problematic production from Grammar 3a.

expr → *term* [("+" | "–") *expr*]

Rewrite using braces:

expr → *term* { ("+" | "–") *term* }

Arguably, this still does not encode left-associative operators. However, our implementation would now involve a loop, not recursion. As we go through the loop, we can easily construct the correct AST for left-associative operators.

Recursive-Descent Parsing
Back to Example #3: Expressions — Left-Associativity [5/5]

Grammar 3b, below, is what we want. It works with a Predictive Recursive-Descent parser, and we can use it to parse left-associative binary operators.

Grammar 3b

$\boxed{\textit{expr} \;\to\; \textit{term}\ \{\ (\ \text{“+”}\ |\ \text{“–”}\ )\ \textit{term}\ \}}$

term → *factor* { (“*” | “/”) *factor* }

factor → ID

| NUMLIT

| “(” *expr* “)”

```
function parse_expr()
    if not parse_term() then
        return false
    end
    while matchString("+")
          or matchString("-") do
        if not parse_term() then
            return false
        end
    end
    return true
end
```

Now, what about generating an AST?

Recursive-Descent Parsing
TO BE CONTINUED …

Recursive-Descent Parsing will be continued next time.