State-Machine Lexing III
The Basics of Syntax Analysis
Recursive-Descent Parsing I

CS 331 Programming Languages

Lecture Slides

Monday, February 9, 2026

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

Unit Overview
Lexing & Parsing

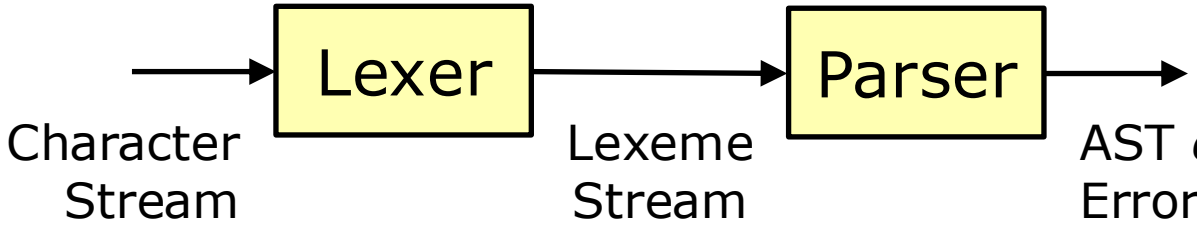
Topics

- ✓ Introduction to lexing & parsing
- ✓ The basics of lexical analysis
- ✓ State-machine lexing I
- ✓ State-machine lexing II
- State-machine lexing III
- The basics of syntax analysis
- Recursive-descent parsing I
- Recursive-descent parsing II
- Recursive-descent parsing III
- Shift-reduce parsing
- Parsing wrap-up

Lexical Analysis (Lexing)

Syntax Analysis (Parsing)

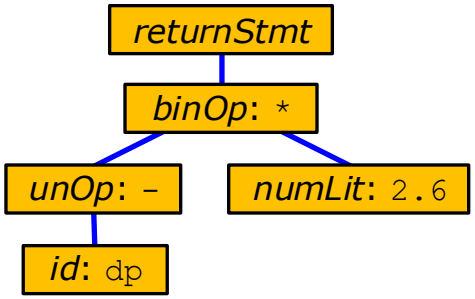
Review




```
return (-dp * 2.6) // x
```

```
return (-dp * 2.6) // x
```

key, op, id, op, num lit, punct, punct



Two steps:

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

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

Review
State-Machine Lexing I/II [1/2]

We have written a lexer as Lua module `lexer`.

See `lexer.lua`.

Internally, our lexer runs as a **state machine**.

- A state machine has a current **state**, which it must store.
- At each step, a state machine looks at the state and the input. It then decides what state to go to next.
- It may make other decisions as well.

As we write a state machine, an important question is *when do we add a new state?*

Two situations can be handled by the same state if they will react identically to all future input.

Review
State-Machine Lexing I/II [2/2]

DONE

- Handling of all lexeme categories, all necessary states.
- Written in class last time: handling of all **NumericLiteral** and **Operator** lexemes, states `DIGIT`, `DIGDOT`, `DOT`, `PLUS`, `MINUS`, `STAR`, handling of **Malformed** lexemes.
- Written after class: comments on all state-handler functions.

See `lexer.lua`.

`lexer.lua` is finished (hopefully).

State-Machine Lexing III

State-Machine Lexing III
More Issues — Multi-Symbol Look-Ahead [1/3]

With our lexical specification, it is tricky to handle "`+.`" and "`-.`". For example, "`+.6`" and "`+.a`" are analyzed very differently.

Input: `+.6`

Output:

`+.6` **NumericLiteral**

Input: `+.a`

Output:

`+` **Operator**

`.` **Operator**

`a` **Identifier**

When we see the dot, we do not know what to do with it—yet.

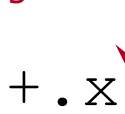
Our solution involved **multi-symbol look-ahead**: making a decision based on more than one upcoming symbol. Our lexer makes use of both the current and next characters instead of just the current character.

State-Machine Lexing III
More Issues — Multi-Symbol Look-Ahead [2/3]

There are other possible solutions to this problem.

For example, we could simply keep reading characters, spitting out a lexeme only when we know what it is. That would require more states, along with a way of handling a read position past the start of the next lexeme (more complexity!).

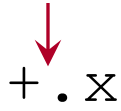
Produce "+" lexeme when we get **here**.



To do that, we could maintain another string holding the next lexeme after the one we are constructing (more complexity!).

Or we could use **backtracking**: returning to a previous state. Backtracking is a useful idea, but it can lead to slow code.

Back up to **here**.



Due to speed issues, backtracking is mostly avoided in both lexing and parsing.

State-Machine Lexing III
More Issues — Multi-Symbol Look-Ahead [3/3]

In contrast, multi-symbol look-ahead tends to lead to simple, fast code. And it is easy to implement. It is a common technique in both lexing and parsing.

When the lexeme categories all form regular languages (as is common), multi-symbol look-ahead is never necessary in lexing. However, in some kinds of parsing, multi-symbol look-ahead may be required; there are PLs that cannot be parsed otherwise.

CFGs are actually classified according to the number of lexemes of look-ahead required by some parsing method. We talk about *LL(1) grammars*, *LL(2) grammars*, etc. More about this later.

State-Machine Lexing III
More Issues — Error Handling [1/5]

The lexical specification tells us to handle illegal characters by forming a single character **Malformed** lexeme.

But is that the best way? How else might we handle this error?

In general, there are three places where a possible error condition in a function might be dealt with.

1. **Before the function.** The caller can prevent the error, so that it never happens.
2. **In the function.** If the function encounters an error, then it can fix it, so the outside world never knows.
3. **After the function.** The function can signal the caller that an error has occurred, leaving it to the caller to deal with.

We look at these three in turn, in the context of our lexer finding an illegal character.

More Issues — Error Handling [2/5]

A possible error condition in a function can be dealt with **before the function**: the caller can prevent the error, so that it never happens.

Applying this idea in our lexer:

A lexer generally reads text straight from a source file. To *prevent* the occurrence of illegal characters would require a preprocessing step before calling the lexer.

But that would make our lexer inconvenient to use. ☹

State-Machine Lexing III
More Issues — Error Handling [3/5]

A possible error condition in a function can be dealt with **in the function.** If the function encounters an error, then it can fix it, so the outside world never knows.

Applying this idea in our lexer:

The only ways a lexer might "fix" illegal characters would be to skip them or change them to legal characters.

But that would change the definition of a syntactically correct program. ☹☹☹

A possible error condition in a function can be dealt with **after the function.** The function can signal the caller that an error has occurred, leaving it to the caller to deal with.

Applying this idea in our lexer:

The caller would usually be a parser. How could the lexer signal the parser that an illegal character has been encountered?

It could raise an exception—and Lua does have exceptions. This would require extra exception-handling code in the parser. ☹

Another option—the one chosen—is to extend the return values of the lexer with an extra category: **Malformed**. We signal the parser that an illegal character has occurred by returning a **Malformed** lexeme.

This method has a nice advantage; *see the next slide.*

State-Machine Lexing III
More Issues — Error Handling [5/5]

A parser must check whether each lexeme is what it wants. There must be code to deal with an unwanted lexeme.

```
if [lexeme is what we want] then
    [Yay!]
else
    [Uh oh, unwanted lexeme.]
end
```

A **Malformed** lexeme will result in this branch being executed.

A **Malformed** lexeme is always unwanted. Encountering one will result in the "else" branch being taken, above.

That branch must be written, regardless of whether **Malformed** lexemes are defined.

Result: our error signaling method *requires no additional code in the parser*. ☺

State-Machine Lexing III
More Issues — Numeric Literals [1/3]

Say our lexer is to be part of a parser for arithmetic expressions with syntax similar to that of Swift, Java, C, C++, and Lua.

Our lexer exhibits the following behavior.

Input: `k - 4`

Output:

`k`	**Identifier**
`-`	**Operator**
`4`	**NumericLiteral**

Input: `k-4`

Output:

`k`	**Identifier**
`-4`	**NumericLiteral**

This seems wrong. But it does follow the lexical specification.

Does this mean that our lexical specification is incorrect?

State-Machine Lexing III
More Issues — Numeric Literals [2/3]

Is our lexical specification incorrect?

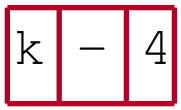
The output of a lexer is rarely needed for its own sake; lexing is typically just the first step toward the goal of constructing an AST, perhaps followed by generating executable code.

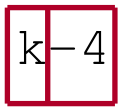
So we cannot really look at a lexical specification in isolation and call it *correct* or *incorrect*.

However, it is true that our lexical specification does not quite match the PL we probably envision it to be for. (This was intentional, but it is based on an actual mistake I made when writing a lexical specification some years ago.)

What can we do about this?

State-Machine Lexing III
More Issues — Numeric Literals [3/3]

Problem: 

vs. 

Ways of Dealing with This Issue

1. Leave the lexical specification alone. Programmers will have to insert space sometimes.
2. Do not always apply maximal munch: sometimes `+` or `–` is a one-character **Operator**, regardless of what follows. This could be a rule that the lexer applies in specified situations, or it could be done at the caller's request.
3. Do maximal munch, but write the lexical specification so that a **NumericLiteral** cannot begin with "`+`" or "`–`". (If we did this, then "`–4`" would be an **Operator** and a **NumericLiteral**.)

Option #3 is common. It is used in Java, C, C++, Lua, Python, and many other major PLs (but not all—Swift does not do this).

Note that `lexer.lua` still follows option #1.

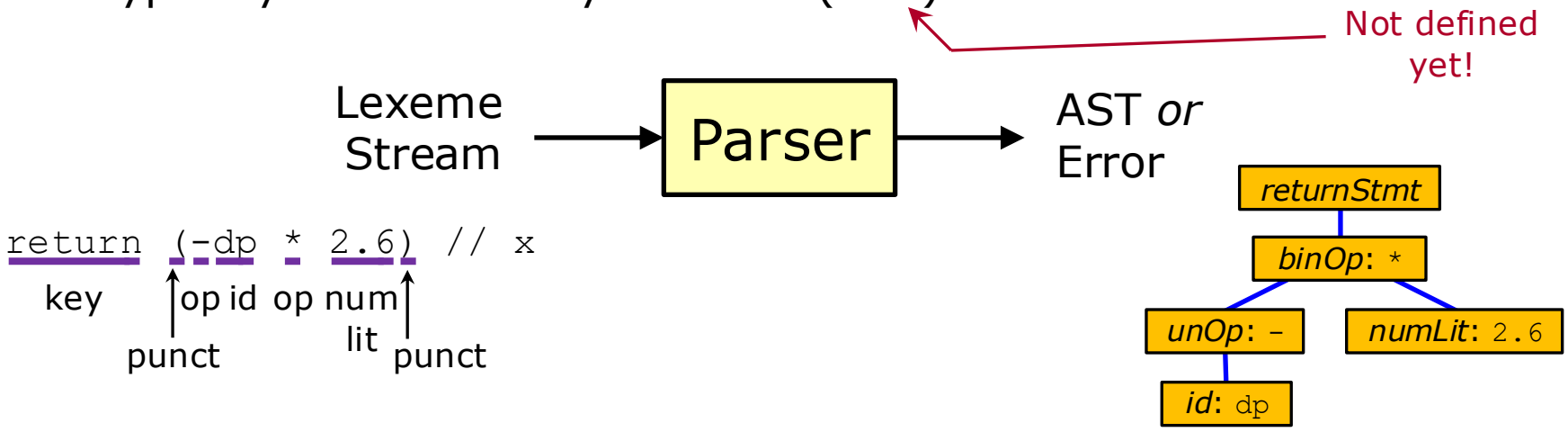
The Basics of Syntax Analysis

The Basics of Syntax Analysis
Introduction [1/2]

We have covered lexical analysis. Now we look at syntax analysis, or parsing.

If lexing is split off as a separate step, then a parser reads a lexeme stream. In addition, it will do the following:

- Determine whether the input is syntactically correct.
- If it is not correct, then output information about the problem.
- If it is correct, then output some representation of its structure, typically an abstract syntax tree (AST).



The Basics of Syntax Analysis
Introduction [2/2]

Syntax analysis is virtually always based on a context-free grammar (CFG) or some similar construction.

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

CFG (start symbol: *item*)

item → "(" *item* ")"
item → *args*
args → NUMLIT
args → "*" "*" "*"

Here, NUMLIT is a lexeme category. On the right, the actual string might be something like "`((+12.34))`".

Derivation

item
(*item*)
((*item*))
((*args*))
((NUMLIT))

There are many different parsing methods based on CFGs. Each is usable with a large number of CFGs—but generally not *all* CFGs.

The Basics of Syntax Analysis
Categories of Parsers [1/3]

Parsing methods can vary a great deal, but they come in two basic flavors: **top-down** and **bottom-up**.

Every grammar-based parser goes through the steps required to find a derivation. (It will usually not output this derivation, or even store it anywhere, but it must go though the steps.)

- A **top-down parser** goes through the derivation from top to bottom, beginning with the start symbol, **expanding** nonterminals as it goes, and ending with the string to be derived (the program?).
- A **bottom-up parser** goes through the derivation from bottom to top, beginning with the string to be derived (the program?), **reducing** substrings to nonterminals as it goes, and ending with the start symbol.

Categories of Parsers [2/3]

Top-down parsers usually expand the leftmost nonterminal first. Thus, they usually produce leftmost derivations.

Top-down parsing code is sometimes hand-coded and sometimes automatically generated.

We will look at a top-down parsing method called **Predictive Recursive Descent**. Assignment 4 will involve writing a Predictive Recursive-Descent parser.

Categories of Parsers [3/3]

Bottom-up parsers usually reduce to the leftmost nonterminal first.
But thinking of the derivation from top to bottom, this would mean that the leftmost nonterminal is expanded *last*; the rightmost nonterminal is expanded first, resulting in a rightmost derivation.

Bottom-up parsing code is almost always automatically generated.

We will look at a bottom-up parsing method called **Shift-Reduce**.
You will not be required to write a Shift-Reduce parser.

The Basics of Syntax Analysis
Categories of Grammars [1/4]

As a rule, fast parsing methods are *not* capable of handling all CFGs. For each kind of parser, there is a category of grammars that such parsers can handle.

A CFG that can be handled by a Predictive Recursive-Descent parser that bases its decisions on *k* input symbols is an **LL(*k*) grammar**. The name **LL** comes from the fact that these parsers read their input **L**eft-to-right and go through the steps necessary to construct **L**eftmost derivations.

k is a number. Here, *symbols* are lexemes.

So if a Predictive Recursive-Descent parser is based on a CFG, and it does *not* do multi-symbol look-ahead, then the grammar it uses must be an **LL(1) grammar**. Over the next few days we will discuss LL(1) grammars further.

Categories of Grammars [2/4]

Here is a simple example. Consider the following grammar.

$S \rightarrow aa$

$S \rightarrow ab$

The above grammar is not LL(1), since we cannot decide which production to use based only on one input symbol. However, this grammar is LL(2).

It is not hard to transform this grammar to an LL(1) grammar that generates the same language.

$S \rightarrow aX$

$X \rightarrow a$

$X \rightarrow b$

Q. The goal is to write a parser. Why are we discussing this?

A. As we will see, similar issues arise when we write a parser for a PL with left-associative binary operators [so `a*b*c` means `(a*b)*c`]—that is, nearly every PL.

Categories of Grammars [3/4]

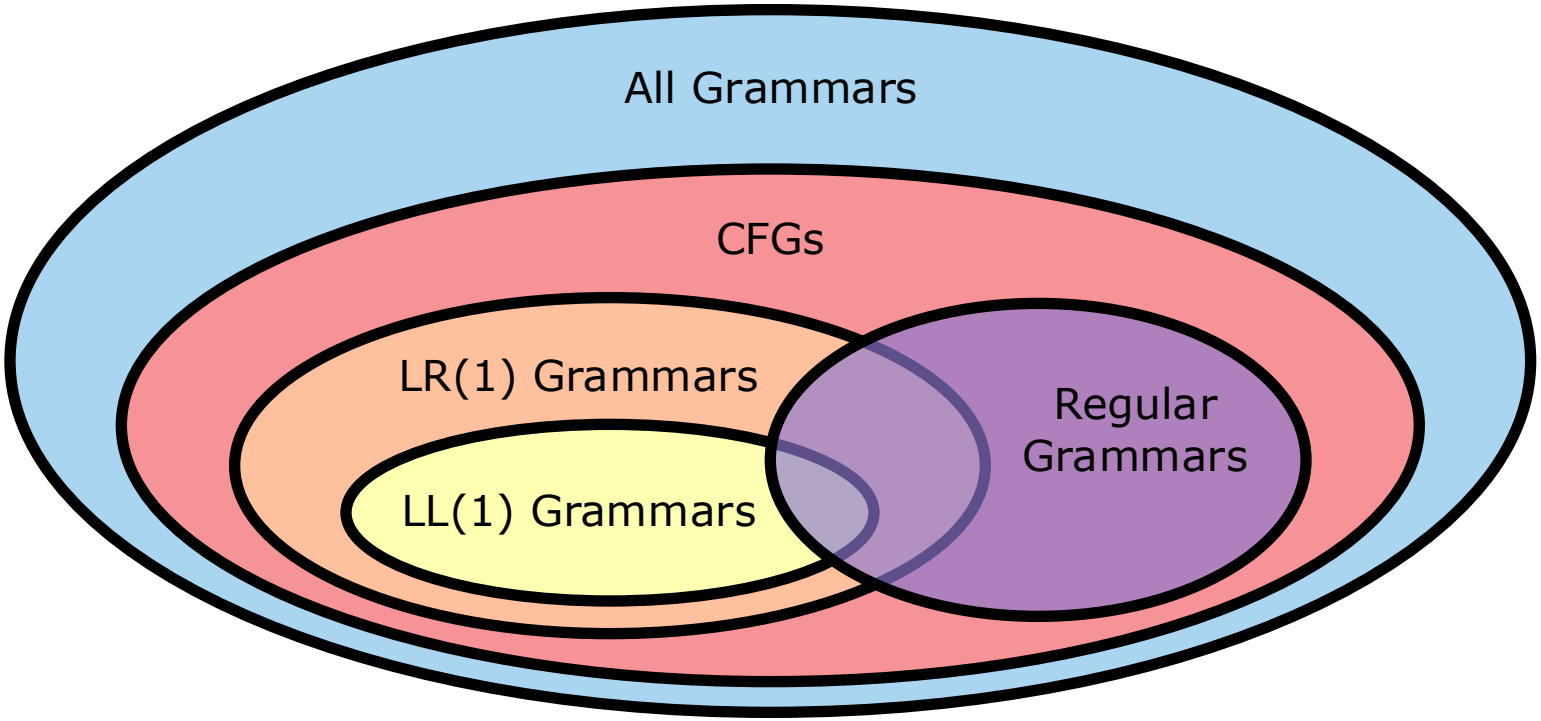
A grammar that can be handled by a Shift-Reduce parser that bases its decisions on *k* input symbols is an **LR(*k*) grammar**. The name **LR** comes from the fact that these parsers read their input **L**eft-to-right and go through the steps necessary to construct **R**ightmost derivations.

So if a Shift-Reduce parser is based on a CFG, and it does *not* do multi-symbol look-ahead, then the grammar it uses must be an **LR(1) grammar**.

The Basics of Syntax Analysis
Categories of Grammars [4/4]

It turns out that every LL(1) grammar is an LR(1) grammar, but there are LR(1) grammars that are not LL(1) grammars (for example, the non-LL(1) grammar from 2 slides back).

This diagram shows the relationship between grammar categories.



Recursive-Descent Parsing I

Recursive-Descent Parsing I
Introduction

Now we look at a parsing method called **Recursive Descent**.

- A top-down parsing method.
- Sometimes hand-coded and sometimes automatically generated.
- Has been known for decades. Still in common use.

When we write a Recursive-Descent parser, we choose what functions to write based on our grammar. Since our parser is tailored for a specific grammar, this is not code we can write once and use for many grammars. A different grammar requires writing a new parser.

Recursive-Descent Parsing I
How It Works [1/2]

A Recursive-Descent parser consists of a number of **parsing functions**. There is one parsing function for each nonterminal. Each parsing function is responsible for parsing all strings that its nonterminal can be expanded into.

So the parsing function corresponding to the start symbol is the one we call to parse the entire input (program?).

The code for a parsing function is essentially a translation into code of the right-hand side of the production for the nonterminal.

- A nonterminal in the right-hand side becomes a call to its parsing function—so the parsing functions are **mutually recursive**.
- A terminal in the right-hand side becomes a check that the input string contains the proper lexeme.

Recursive-Descent Parsing I
How It Works [2/2]

Suppose a Recursive-Descent parser is applying a production, but the input does not fit its right-hand side. There are two options:

1. Backtrack. Try another production.
2. Give up. Flag the input as syntactically incorrect.

Option #1 can result in a parser that is far too slow.

But option #2 is only correct if the chosen production was the right one. So we must be able to *predict* which production to use based on the next lexeme (more lexemes if we do look-ahead). This restricts which grammars we can use.

A parser that uses option #2 is said to be **predictive**. Again, the CFGs for which we can write a correct **Predictive Recursive-Descent** parser that bases its decisions on *k* lexemes are called **LL(*k*) grammars**.

Recursive-Descent Parsing I
Example #1: Simple [1/5]

Let's write a Predictive Recursive-Descent parser based on the following CFG.

Grammar 1

item → "(" *item* ")"

item → *args*

args → NUMLIT

args → "*" "*" "*"

The start symbol is *item*.

NUMLIT represents the **NumericLiteral** category from our lexer.

Our parser will be written in Lua. It will take input from our in-class lexer (module `lexer`).

Recursive-Descent Parsing I
Example #1: Simple [2/5]

A Recursive-Descent parser has one parsing function for each nonterminal. It is appropriate to begin by combining productions with a common left-hand side.

Grammar 1

item → "(" *item* ")"
item → *args*
args → NUMLIT
args → "*" "*" "*"

Grammar 1a

item → "(" *item* ")"
| *args*
args → NUMLIT
| "*" "*" "*"

Example #1: Simple [3/5]

Grammar 1a

item → "(" *item* ")"
| *args*

args → NUMLIT
| "*" "*" "*"

Next we turn each production into code for a function.

Let's name each parsing function after its nonterminal. So the parsing function for *item* will be `parse_item`. And the parsing function for *args* will be `parse_args`.

A parser typically outputs either an AST or an error message. But *for now*, our parser will simply return `true` or `false`, depending on whether the input is syntactically correct. Eventually, we will write code to construct an AST.

Recursive-Descent Parsing I
Example #1: Simple [4/5]

I have written a framework for a Recursive-Descent parser that uses our lexer. This is a Lua module exporting a function `parse`.

In a parsing function (e.g., `parse_item` or `parse_args`), the current lexeme & category are in variables `lexstr` & `lexcat`, respectively. When the function starts, a lexeme is already in these variables. To move to the next lexeme, call `advance`.

Pass function `matchCat` a lexeme category (e.g., `lexer.NUMLIT`). If the current lexeme is in this category, then it sets variable `matched` to the string form of the lexeme, calls `advance`, and returns `true`; otherwise, it returns `false`—with no `advance` call.

`matchString` is similar, but it takes a string to match (e.g., `">="`).

I have written a simple program that uses this parser.

See `rdparser1.lua` *&* `use_rdparser1.lua`*.*

Recursive-Descent Parsing I
Example #1: Simple [5/5]

Grammar 1a

item → "(" *item* ")"
| *args*

args → NUMLIT
| "*" "*" "*"

TO DO

- Write a Predictive Recursive-Descent parser based on Grammar 1a.

Done. See `rdparser1.lua`.

Recursive-Descent Parsing I
Handling Incorrect Input [1/4]

What output does our parser give for each of the following inputs?

1. `""`
2. `"123"`
3. `"xyz"`
4. `"***"`
5. `"((+12.34))"`
6. `"( (((( * * * )) )))"`
7. `"(1,2,3)"`
8. `"(((42)))"`
9. `"((42)))"`
10. `"1,2,3"`

Q. Are these outputs what we want them to be?

A. For all but #9 and #10, the output is what we expect. But the parser says those two are correct. Obviously, they are not.

I claim, however, that *this is not actually the parsing-function bug that we might think it is*. Read on …

Recursive-Descent Parsing I
Handling Incorrect Input [2/4]

Our parser says the following are both syntactically correct:

- `"((42)))"`
- `"1,2,3"`

Why?

Function `parse_item` is called to parse the entire input. It is *also* called, recursively, to parse an *item* between parentheses. When the latter invocation of the function sees "`)`" following a correct parse, it must simply return, assuming that the "`)`" is handled by its caller.

So `parse_item` sees the first string above as a syntactically correct string "`((42))`" followed by extra stuff: "`)`".

The second string is similar: a correct "`1`" followed by extra "`,2,3`".

Recursive-Descent Parsing I
Handling Incorrect Input [3/4]

Our parsing functions are acting correctly. But the parser is not giving us the information we need. What can we do about this?

One common solution is to add another lexeme category: **end of input**. There is 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* ")"
| *args*
args → NUMLIT
| "*" "*" "*"

This idea will *not* be used in our current parser.

Recursive-Descent Parsing I
Handling Incorrect Input [4/4]

Another solution is to add an extra check at the end of parsing, to see whether all lexemes have been read. If we do this, then 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.

TO DO

- Modify `rdparser1.lua` to implement the above idea.
- Modify `use_rdparser1.lua` so that it uses the new information.

Done. See `rdparser1.lua` *&* `use_rdparser1.lua`*.*