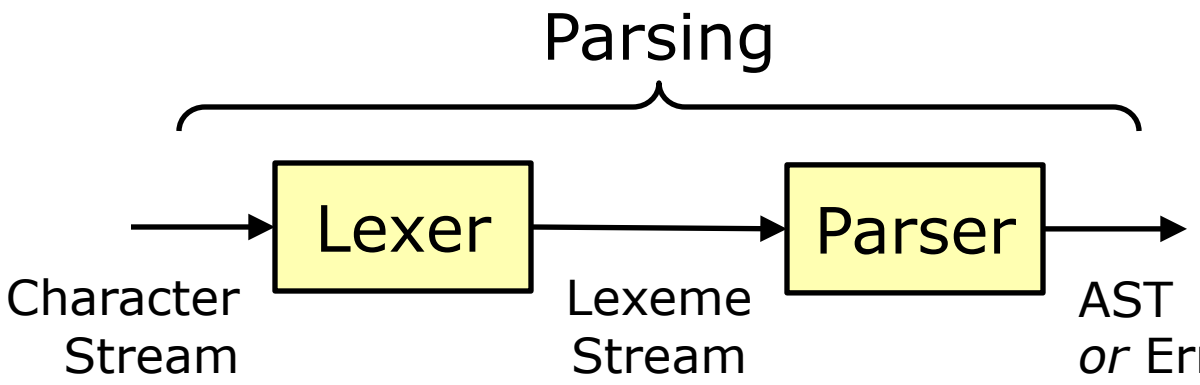
Introduction to Syntax Analysis
Recursive-Descent Parsing

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Friday, February 10, 2017

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

Review
Overview of Lexing & Parsing

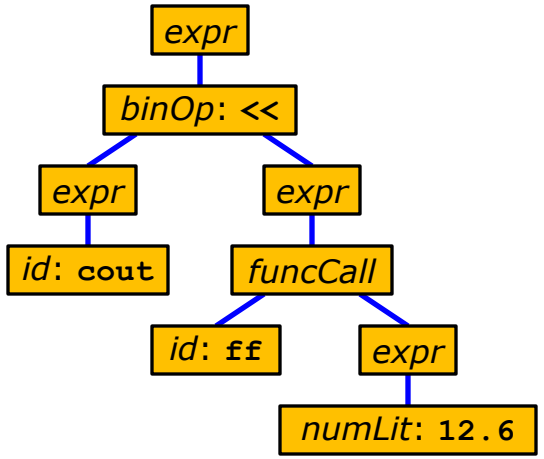


`cout << ff(12.6);`

`cout` `<<` `ff` `(` `12.6` `)` `;`

id op id lit punct

op op



Two phases:

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

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

Review
Introduction to Lexical Analysis

There are essentially three ways to write a lexer.

- Automatically generated, based on regular grammars or regular expressions for each lexeme category.
- Hand-coded state machine using a table.
- Entirely hand-coded state machine.

We wrote a lexer using the last method.

Writing a Lexer — Introduction, Design Decisions

Our lexer is implemented as Lua module **`lexer`**.

See **`lexer.lua`**.

Function **`lexer.lex`** provides an iterator. A for-in loop gives each lexeme as a pair: text (string) and category (number). The latter is an index for **`lexer.catnames`**. Code to use our lexer:

```
lexer = require "lexer"

for lexstr, cat in lexer.lex(program) do
    catstr = lexer.catnames(cat)
    io.write(string.format("%-10s  %s\n",
                           lexstr, catstr))
end
```

Writing a Lexer — Coding a State Machine

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

- A state machine has a current **state**, stored in variable `state`.
- It proceeds in a series of steps. At each step, it looks at the current character in the input and the current state. It then decides what state to go to next.

As we write a state machine, an important question is **when do we add a new state?** A good guiding principle:

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

Writing a Lexer — Notes [1/2]

With our lexeme specification, it is tricky to handle "`+.`" and "`-.`".

- For example, "`+.3`" is a single lexeme (*NumericLiteral*), while "`+.x`" is three lexemes: (*Operator*, *Operator*, *Identifier*).

We handle this using **lookahead**: peeking ahead in the input to get the information needed to make a decision. This is a common technique at all phases of parsing.

Writing a Lexer — Notes [2/2]

Our lexeme specification results in some curious behavior.

Input: `k - 4`

Output:

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

Input: `k-4`

Output:

```
k   Identifier
-4  NumericLiteral
```

One possible way of altering this (*not* done in lexer.lua): change the longest-lexeme rule. Allow the caller to set a flag during lexing, indicating that the next lexeme, if it begins with + or –, should always be interpreted as an *Operator*.

An important point: **sometimes the parser may need to guide the lexer**. This can affect the design of a lexer.

Introduction to Syntax Analysis
Basics [1/2]

We have covered lexical analysis. Now we look at the second stage of parsing, **syntax analysis**. In this stage:

- We determine whether the input is syntactically correct.
- If it is correct, then we usually output some representation of its structure: often an **abstract syntax tree** (**AST**).
- If it is not correct, we usually output information about the problem.

As we have noted, syntax analysis is also called **parsing**; this is the narrow sense of the term.

Introduction to Syntax Analysis
Basics [2/2]

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. The actual string might be something like "`((abc_39))`".

Derivation

item

(*item*)

((*item*))

((*thing*))

((ID))

A general method for doing parsing is a **parsing algorithm**. A typical parsing algorithm is usable with any number of CFGs. When we implement a parsing algorithm to handle a *particular* CFG, we have a **parser**: a code module that does parsing.

Categories of Parsers [1/3]

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

Every parser goes through the steps to find a derivation based on the CFG being used. (It will generally not output this derivation; it probably will not even store it anywhere, but it will go though the steps.)

- A **top-down parsing algorithm** 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 parsing algorithm** goes through the derivation from bottom to top, beginning with the string to be derived (the program?), collapsing 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.

Some top-down parsers are in a category known as **LL parsers**, the name coming from the fact that they handle their input in a strictly **L**eft-to-right order, and they go through the steps to generate a **L**eftmost derivation.

If a top-down parsing algorithm is *not* an LL parser, this is typically because it does multiple-lexeme lookahead, and so does not handle its input in a strictly left-to-right order.

Top-down parsing code is often hand-coded—although top-down parser generators do exist.

We will look closely at a top-down parsing algorithm called **Recursive Descent**. This algorithm can be written to use multiple-lexeme lookahead, but if it does not, then it is in the LL category. An upcoming assignment will involve writing a Recursive-Descent parser.

Categories of Parsers [3/3]

Bottom-up parsers usually collapse 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.

Thus, some bottom-up parsing algorithms are in a category known as **LR parsers**, the name coming from the fact that they handle their input in a strictly **L**eft-to-right order, and they go through the steps to generate a **R**ightmost derivation.

If a bottom-up parsing algorithm is *not* an LR parser, this is, again, typically because it does multiple-lexeme lookahead.

Bottom-up parsing code is almost always automatically generated.

We will look closely at a bottom-up parsing algorithm called **Shift-Reduce**. However, you will *not* be required to implement a Shift-Reduce parser.

Categories of Grammars [1/2]

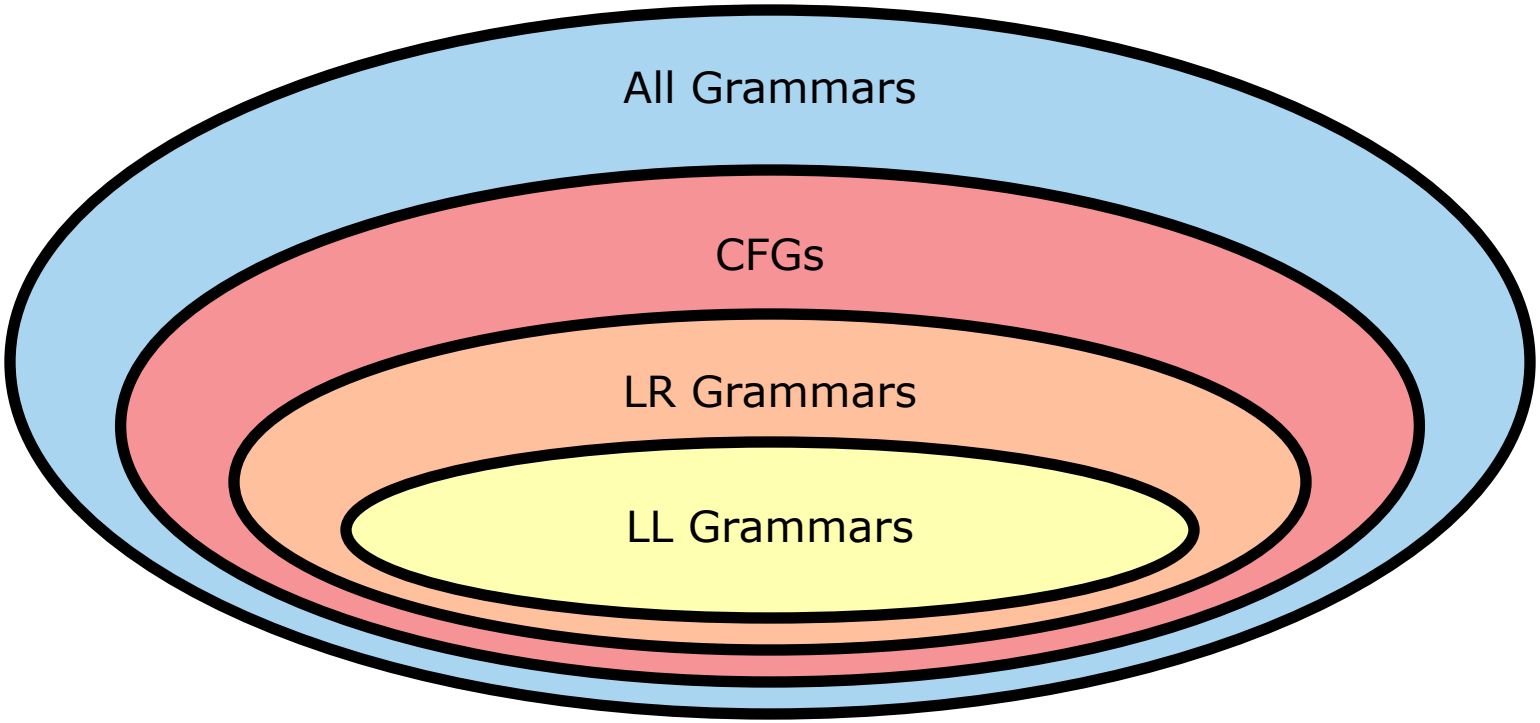
As a rule, a fast parsing algorithm is *not* capable of handling all CFGs. For each category of parsing algorithms (LL, LR, etc.), there is an associated category of grammars that such algorithms can handle.

The grammars that LL parsers can handle are called **LL grammars**. Similarly, LR parsers can handle **LR grammars**.

Curiously, every LL grammar is an LR grammar, while there are LR grammars that are not LL grammars. So LR parsing algorithms are more general. Note, however, that when we write a compiler for a programming language, we only need *one* grammar, and if it is an LL grammar, then an LL parser works fine.

Categories of Grammars [2/2]

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



Recursive-Descent Parsing
Introduction

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

- A top-down parsing algorithm.
- In the LL category—as long as multiple-lexeme lookahead is not used.
- Almost always hand-coded.
- Has been known for decades. Still in common use.

Algorithms you may be familiar with (Binary Search, Merge Sort, etc.) can be written once, and, if the implementation is suitably generic, never written again. But Recursive Descent is not like this. When we write a Recursive-Descent parser, we choose what functions to write based on our grammar. Thus a Recursive-Descent parser is tailored for a specific grammar; a different grammar requires writing a new parser.

Recursive-Descent Parsing
How It Works

A Recursive-Descent parser has one parsing function for each nonterminal in the grammar. Each parsing function is responsible for parsing all strings that the corresponding nonterminal can be expanded into. So the 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.

- Other nonterminals in the right-hand side become function calls—and 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
Example #1: Simple [1/5]

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

Grammar 1

item → "`(`" *item* "`)`"

item → *thing*

thing → ID

thing → "`%`"

The start symbol is *item*. ID represents the *Identifier* category from our lexer.

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

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

Grammar 1

item → "(" *item* ")"

item → *thing*

thing → ID

thing → "%"

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

Grammar 1a

item → "(" *item* ")"
| *thing*

thing → ID
| "%"

Example #1: Simple [3/5]

Grammar 1a

item → "`(`" *item* "`)`"
| *thing*

thing → ID
| "`%`"

Next we turn each production into code for a function.

It is convenient to name each parsing function after the nonterminal it handles. So the parsing function for *item* will be **`parse_item`**. And the parsing function for *thing* will be **`parse_thing`**.

A parser typically outputs an AST or an error message. For now, our parser will simply return **`true`** or **`false`**, depending on whether the input is syntactically correct. (We *will* eventually generate an AST.)

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

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

See **`rdparser1.lua`**.

In a parsing function (e.g., **`parse_item`** or **`parse_thing`**), 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`**.

There are two convenience functions: **`matchCat`** & **`matchString`**. Pass the former a lexeme category (e.g., **`lexer.ID`**). If the current lexeme is in this category, then it calls **`advance`** and returns **`true`**; otherwise, it returns **`false`**. Function **`matchString`** is similar, except that it takes a string to match (e.g., `"%"`).

I have also written a simple program that uses the parser.

See **`userdparser1.lua`**.

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

Grammar 1a

item → "(" *item* ")"
| *thing*

thing → ID
| "%"

TO DO

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

Done. See **`rdparser1.cpp`**.

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

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

- `xyz`
- `123`
- `%`
- `((abc_39))`
- `(((((%)))))`
- `(a,b,c)`
- `(((x))`
- `((x)))`
- `a,b,c`

Are these outputs what we want them to be?

For all but the last two, the output is what we expect. But the parser says the last two are correct. Obviously, they are not.

I claim, however, that *this is not a bug*. Read on …

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

Our parser says the following are both syntactically correct:

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

Why?

Remember: **Recursive** Descent. 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 "`((x))`" followed by extra characters: "`)`".

The second string is similar: correct "`a`" followed by extra "`,b,c`".

Recursive-Descent Parsing
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**. 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 is only an example. I will not be using this idea in our parser.

Recursive-Descent Parsing
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 we will eventually write. So I will be using it in all of our Recursive-Descent parsers.

TO DO

- Modify **`rdparser1.lua`** so that it implements the above idea.

Done. See **`rdparser1.cpp`**.

Recursive-Descent Parsing
TO BE CONTINUED …

Recursive-Descent Parsing will be continued next time.