Thoughts on Assignment 3
Recursive-Descent Parsing continued

CS 331 Programming Languages
Lecture Slides
Wednesday, February 12, 2025

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

Thoughts on Assignment 3

Thoughts on Assignment 3
Introduction [1/3]

In Assignment 3 you will write a state-machine lexer, in the form of Lua module `lexit`, stored in file `lexit.lua`. It will be similar to module `lexer` (written in class), but it will be based on a different lexical specification.

Later, in Assignment 4, you will write a parser that uses your lexer. And in Assignment 6 you will write an interpreter that uses the output of your parser. The result will be an implementation of a programming language called **Fulmar**—which I have invented just for this class.

Thoughts on Assignment 3
Introduction [2/3]

Here is a sample Fulmar program.

```
# fibo (param in variable n)
# Return Fibonacci number F(n).
func fibo()
    currfib = 0
    nextfib = 1
    i = 0  # Loop counter
    while i < n
        tmp = currfib + nextfib
        currfib = nextfib
        nextfib = tmp
        i = i+1
    end
    return currfib
end
```

```
# Main program
# Print some Fibonacci numbers
how_many_to_print = 20

println("Fibonacci Numbers")

j = 0  # Loop counter
while j < how_many_to_print
    n = j  # Set param for fibo
    ff = fibo()
    println("F(", j, ") = ", ff)
    j = j+1
end
```

Thoughts on Assignment 3
Introduction [3/3]

The lexical structure of Fulmar is similar to that in `lexer.lua`:

- Whitespace is all *mostly* treated the same. Lexemes may be separated by whitespace, but they are generally not required to be.
- **Legal** characters are whitespace & printable ASCII.
- Comments may contain any character at all. They are skipped.
- The maximal-munch rule is followed.

Some of the differences from `lexer.lua`:

- A comment begins with "`#`" and continues to the next newline or the end of the input.
- There is another lexeme category: **StringLiteral**. These are surrounded by single quotes (`'` … `'`) or double quotes (`"` … `"`).
- There is another kind of **Malformed** lexeme: something that would be a **StringLiteral**, except the ending quote never appears.
- The `k - 4` / `k-4` issue is dealt with by requiring a **NumericLiteral** to begin with a digit. "`-4`" is an **Operator** and a **NumericLiteral**.

Thoughts on Assignment 3
Lexeme Categories [1/6]

Fulmar has 7 lexeme categories:

- **Keyword**
- **Identifier**
- **NumericLiteral**
- **StringLiteral**
- **Operator**
- **Punctuation**
- **Malformed**

We look briefly at each of these.

Thoughts on Assignment 3
Lexeme Categories [2/6]

A **Keyword** lexeme is one of the following 12:

```
chr    elif    else    end    func    if    print    println
readnum    return    rnd    while
```

An **Identifier** lexeme is like a C identifier. It cannot be one of the **Keyword** lexemes. Examples:

```
myvar    ___42_cr    HelloThere    RETURN
```

So the reserved words are the same as the **Keyword** lexemes.

Thoughts on Assignment 3
Lexeme Categories [3/6]

A **NumericLiteral** is *either* a sequence of one or more digits …

```
7   34567   9876   0001000   000
```

… *or* the above followed by an **exponent**, which is a letter “`e`” or “`E`”, an optional “`+`”, and then one or more digits:

```
7e34   34567E+1   9876e0   0001000E1234   000e+00
```

A **NumericLiteral** must begin with a digit. It cannot include a dot (“`.`”). An exponent cannot include “`-`”.

Thoughts on Assignment 3
Lexeme Categories [4/6]

A **StringLiteral** lexeme is a single or double ASCII quote, followed by zero or more characters, followed by a matching quote. There are no escapes. Here are some **StringLiteral** lexemes.

```
"Hello there!"    ''    '"'    "'--#!Ωé\"
```

Any character, legal or illegal, other than a newline or the starting quote, may appear inside a **StringLiteral**. The character “`#`” appearing in a **StringLiteral** does *not* begin a comment.

Thoughts on Assignment 3
Lexeme Categories [5/6]

An **Operator** lexeme is one of the following 17:

```
!    &&    ||    ==    !=    <    <=    >    >=    +    -    *
/    %     [     ]     =
```

A **Punctuation** lexeme is a single character that is not whitespace, not part of a comment, and not part of a lexeme in any of the other categories. Examples:

```
;    {    }    (    )    ,    &    $
```

The definition of the **Punctuation** category means that all input texts can be lexed. Every input character that is not skipped will be part of a lexeme in one of the specified categories.

Thoughts on Assignment 3
Lexeme Categories [6/6]

Malformed lexemes come in 2 kinds: *bad character* & *bad string*.

- A **bad character** is much like a **Malformed** lexeme in `lexer.lua`: a single illegal character that is not part of a comment or **StringLiteral**.
- A **bad string** is something that would be a **StringLiteral**, except that there is a newline or the input ends before the ending quote appears. Examples (each of the following must end at the end of the line or the end of the input):

```
"abc                         'a b c d " e f g
```

To check for a *bad string* lexeme, you will need to check for the end of the input. You can do this by checking whether the current character is the empty string.

```
if ch == "" then  -- This is Lua code
    …
```

Unit Overview
Lexing & Parsing

Topics

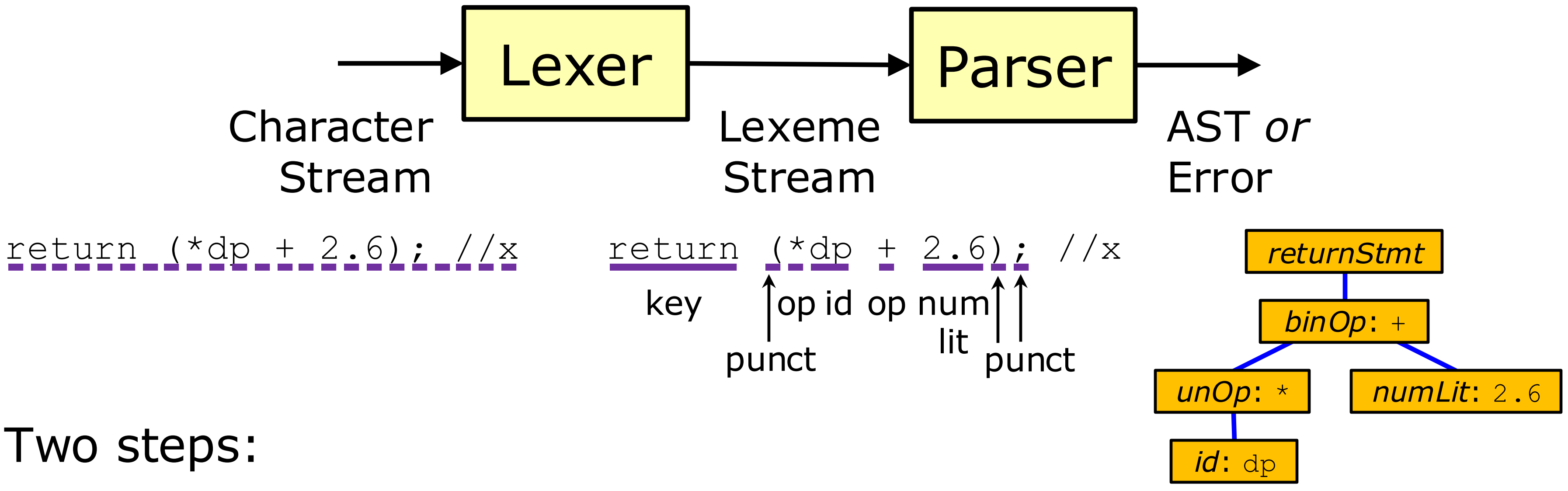
- ✓ Introduction to lexing & parsing
- ✓ The basics of lexical analysis
- ✓ State-machine lexing
- ✓ The basics of syntax analysis
- (part) Recursive-descent parsing
- Shift-reduce parsing
- Parsing wrap-up

Lexical Analysis (Lexing)

Syntax Analysis (Parsing)

Review

Review
Introduction to Lexing & Parsing



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
The Basics of Syntax Analysis — Categories of Parsers

A parser based on a CFG will do through the steps to construct a derivation. Based on how this is done, parsing methods can be divided into two broad categories: *top-down* and *bottom-up*.

Top-Down Parsers

- Go through the derivation top to bottom, **expanding** nonterminals.
- Sometimes hand-coded and sometimes automatically generated.
- Method we look at: **Predictive Recursive Descent**.

Bottom-Up Parsers

- Go through the derivation bottom to top, **reducing** substrings to nonterminals.
- Almost always automatically generated.
- Method we look at: **Shift-Reduce**.

Review
The Basics of Syntax Analysis — Categories of Grammars

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

The CFGs that a Predictive Recursive-Descent parser can be based on, if *k* input symbols (here, lexemes) are used to make each decision, are called **LL(*k*) grammars**. Similarly, Shift-Reduce parsers can be based on **LR(*k*) grammars**.

Every LL(1) grammar is an LR(1) grammar, while there are LR(1) grammars that are not LL(1) grammars.

So Shift-Reduce is a more general technique. Note, however, that when we write a compiler or interpreter for a programming language, we only need *one* grammar. If it is an LL(1) grammar, then a Predictive Recursive-Descent parser works fine.

Review
Recursive-Descent Parsing — Introduction, How It Works

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

- A top-down parsing method.
- Sometimes hand-coded and sometimes automatically generated.
- 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 can be slow. A **predictive** parser must always choose the right production the first time. This restricts the usable grammars: LL(*k*) grammars. So if we avoid multi-symbol look-ahead: 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 → *value*
value → NUMLIT
value → "*" "–" "*"

→

Grammar 1a

item → "(" *item* ")"
  | *value*
value → NUMLIT
  | "*" "–" "*"

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

See `rdparser1.lua`.

Review
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_value() 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.

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

Grammar 1a

item	→	"(" *item* ")"
	|	*value*
value	→	NUMLIT
	|	"*" "_" "*"

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

Review
Recursive-Descent Parsing — Handling Incorrect Input

A naively written parser may classify some inputs as correct when it cannot parse the entire input.

Example: `((42)))`

$$\underbrace{\texttt{((42))}}_{\text{Syntactically correct}}\ \Big|\ \underbrace{\texttt{)}}_{\text{Extra stuff}}$$

Reason. The extra stuff at the end is the caller's responsibility.

Solution. Call the *whole input* correct only if the end is reached.

Two Methods

- Introduce a new **end of input** lexeme (standard symbol: $). Revise the grammar to include it.
- After parsing, check whether the end of the input was reached.

We modified our parser to use the second method.

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

Recursive-Descent Parsing

continued

Recursive-Descent Parsing
Example #2: More Complex [1/3]

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

Grammar 2

item → "`(`" *item* "`)`"
  | *value*
value → NUMLIT { ("`,`" | "`;`") NUMLIT }
  | "`*`" "`-`" "`*`"
  | 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 from Grammar 1 are also in the new language. But Grammar 2 also allows strings like these:

- `((((1,2,3;4))))`
- `((abc[(def[*-*])]))`
- `xx[2,4,10]`

Recursive-Descent Parsing
Example #2: More Complex [2/3]

value → NUMLIT { ("," | ";") NUMLIT }
  | "*" "-" "*"
  | ID ["[" *item* "]"]

This is the production for the nonterminal *value* in Grammar 2. The production for *item* is unchanged from Grammar 1.

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` *&* `use_rdparser2.lua`*.*

Recursive-Descent Parsing
Example #2: More Complex [3/3]

Consider the following line from `rdparser2.lua`:

```
while matchString(",") or matchString(";") do
```

Q. What if the above sees "`,;`". Will it match *both* characters?

A. No, it will not, thanks to **short-circuiting**.

In a logical-OR, if the first operand is truthy, then there is no need to evaluate the second. So simply return the first. But if the first operand is falsy, then evaluate and return the second.

Similarly, in a logical-AND, if the first operand is falsy, then return it, without evaluating the second. Otherwise, evaluate and return the second operand.

Short-circuiting of logical-AND and logical-OR is common. It is found in Lua, C++, C, Java, Python, and many other PLs.

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 will 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 rewritten: "`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 allow backtracking, 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 directly on Grammar 3. This 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 [1/6]

An **LL(1) grammar** is a CFG that can be used by a Predictive Recursive Descent parser that bases each decision on a single lexeme (that is, without using multi-symbol look-ahead).

For a grammar to be LL(1), at each step in parsing we must be able to decide which production to use, or whether a syntax error has been found, based only on the current input lexeme.

Recall the origin of the name "LL": handling input in a **L**eft-to-right order, making some irrevocable decision at each step, and going through the steps required to generate a **L**eftmost derivation.

Now we look at examples of issues that prevent a CFG from being an LL(1) grammar.

Recursive-Descent Parsing
LL(1) Grammars [2/6]

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.

Recursive-Descent Parsing
LL(1) Grammars [3/6]

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

Grammar A′

xx → *yy* “`b`” | “`a`”

yy → *xx* “`+`”

Grammar A′ also contains left recursion—in indirect form. It is not an LL(1) grammar.

Recursive-Descent Parsing
LL(1) Grammars [4/6]

The grammar below illustrates another problem.

Grammar B

xx → "`a`" *yy* | "`a`" *zz*

yy → "`*`"

zz → "`/`"

If we do not use multi-symbol look-ahead, then we cannot even begin to write a Recursive-Descent parser for Grammar B. How would the code for function `parse_xx` start? Does it take the first or second option? There is no way to tell, without considering more than one lexeme when making the decision.

Recursive-Descent Parsing
LL(1) Grammars [5/6]

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*.

Recursive-Descent Parsing
LL(1) Grammars [6/6]

One last non-LL(1) grammar.

We will not look further at what an LL(1) grammar is. However, Grammars A–D illustrate essentially all the issues that prevent a grammar from being LL(1).

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, with one of these as input. What would happen?

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?

Answer coming up.

Recursive-Descent Parsing
TO BE CONTINUED …

Recursive-Descent Parsing will be continued next time.