Recursive-Descent Parsing continued

CS 331 Programming Languages
Lecture Slides
Friday, February 14, 2025

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

Unit Overview
Lexing & Parsing

Topics

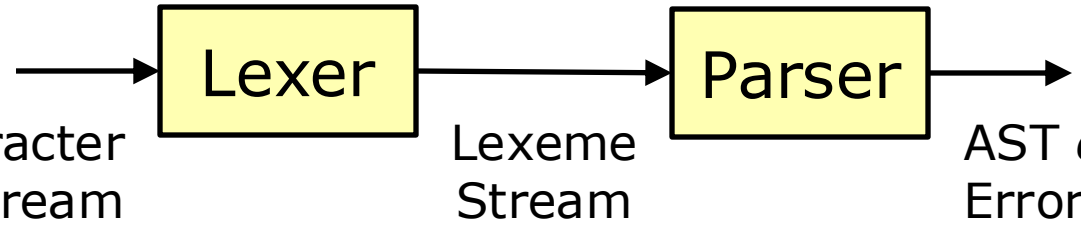
- ✓ Introduction to lexing & parsing
- ✓ The basics of lexical analysis
- ✓ State-machine lexing
- ✓ The basics of syntax analysis
- (part) Recursive-descent parsing
- Shift-reduce parsing
- Parsing wrap-up

Lexical Analysis (Lexing)

Syntax Analysis (Parsing)

Review

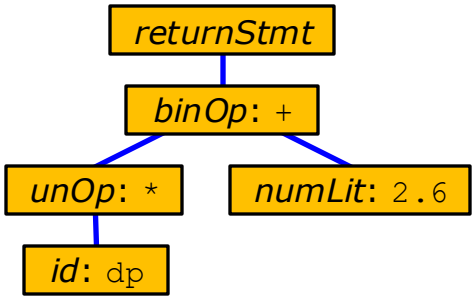
Review
Introduction to Lexing & Parsing




```
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.

Recursive-Descent Parsing [1/4]

Recursive Descent is a top-down parsing method.

Predictive Recursive Descent: no backtracking.

There is one parsing function for each nonterminal. This parses all strings that the nonterminal can be expanded into.

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

- A terminal in the right-hand side becomes a check that the input string contains the proper lexeme.
- A nonterminal becomes a call to its parsing function.
  - If that returns `false`, our function must return `false`—no backtracking.
- Brackets […] become a conditional.
- Braces { … } become a loop.

See `rdparser2.lua` *&* `use_rdparser2.lua`.

Recursive-Descent Parsing [2/4]

An **LL(*k*) grammar** (*k* is a number) is a CFG on which we can base a Predictive Recursive Descent parser that makes decisions using *k* upcoming lexemes.

Since we do not do multi-symbol look-ahead, the CFG that each of our parsers is based must be an LL(1) grammar.

With an LL(1) grammar, we must be able to make decisions (which production to use, whether to parse an optional portion, whether we have a syntax error) based only on the current lexeme.

Last time we looked at several non-LL(1) grammars.

Review
Recursive-Descent Parsing [3/4]

We wish to parse arithmetic expressions involving the usual binary operators, returning an *abstract syntax tree* (*AST*).

Operators are left-associative: "`a + b + c`" means "`(a + b) + c`".
Precedence is as usual: "`a + b * c`" means "`a + (b * c)`".
Override with parentheses: "`(a + b) * c`".

We use our in-class lexer, so "`k-4`" must be written as "`k - 4`".

Recursive-Descent Parsing [4/4]

Grammar 3 encodes associativity and precedence, and it allows use of parentheses to override them.

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
        …
```

The code on the right is a (correct!) translation of part of the grammar. But the code has problems, indicating that the grammar is not LL(1). We cannot use this grammar—directly.

Recursive-Descent Parsing

continued

Recursive-Descent Parsing
Transforming Grammars [1/3]

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

Here is Grammar A from last time. It is not LL(1). On the right is an LL(1) grammar that generates the same language.

Grammar A

xx → *xx* "`+`" "`b`" | "`a`"

Grammar Aa

xx → "`a`" *yy*

yy → "`+`" "`b`" *yy* | ""

Recursive-Descent Parsing
Transforming Grammars [2/3]

Here is Grammar B, which is not LL(1). Again, on the right is 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 [3/3]

For each of the other examples of non-LL(1) grammars discussed last time, we can find an LL(1) grammar that generates the same language.

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

In the specification of programming-language syntax, it is common to be faced with a grammar that is not LL(1), but that can be transformed to one that is LL(1).

Next, we look at this issue for our expression grammar.

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 grammars for expressions involving left-associative binary operators are not LL(1); they are, in fact, not LL(*k*) for any *k*.

Back to Example #3: Expressions — Left-Associativity [2/5]

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 …*

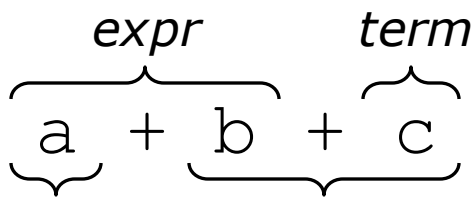
Back to Example #3: Expressions — Left-Associativity [3/5]

Grammar 3a

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

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

Left-associative:



Right-associative:

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 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

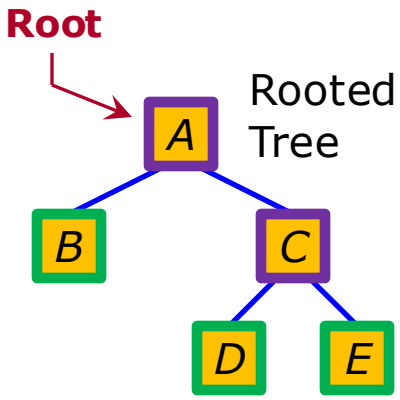
expr → *term* { ("+" | "-") *term* }
term → *factor* { ("*" | "/") *factor* }
factor → ID
| NUMLIT
| "(" *expr* ")"

```
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
    -- Construct AST here
    return true
end
```

Now, what about constructing an AST?

Back to Example #3: Expressions — ASTs [1/4]

Let's review the idea of a **rooted tree**.

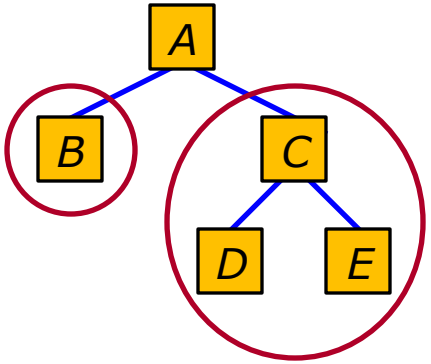


A node with no children is a **leaf**. Any other node is an **internal node**.

The tree to the right has 3 **leaves** (*B*, *D*, *E*) and 2 **internal nodes** (*A*, *C*).

The **subtrees** of an internal node are those rooted at each of its children.

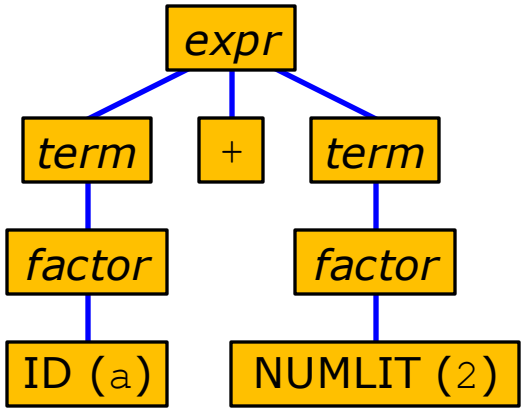
In the tree to the right, the two subtrees of the *A* node are circled.



Back to Example #3: Expressions — ASTs [2/4]

Recall that a **parse tree** includes one leaf node for each lexeme in the input, and also one node for each nonterminal in the derivation.

Here is the parse tree for `a + 2`, based on Grammar 3b.

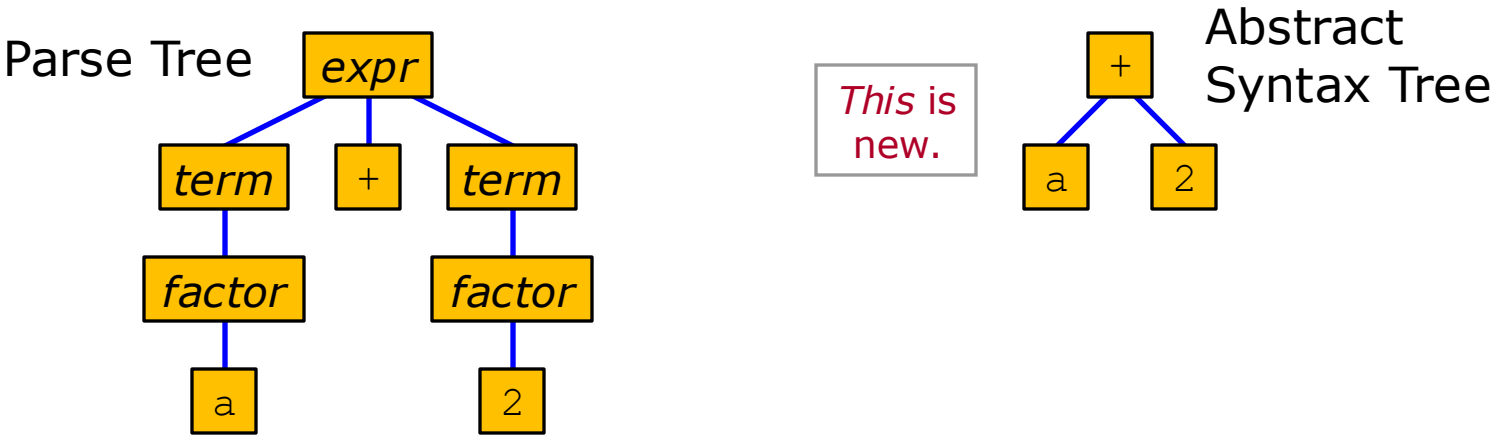


This is not new! It is the same kind of tree we drew before.

Back to Example #3: Expressions — ASTs [3/4]

An **abstract syntax tree** (**AST**) is a rooted tree representing the structure of parsed input. Each internal node represents an operation. Its subtrees are the ASTs for the entities the operation is applied to.

On the left is our parse tree for `a + 2`, slightly simplified. On the right is an AST for the same expression.

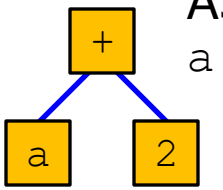


Operation may be broadly defined. For example, an AST node might represent the operation "execute these statements, in order". Its subtrees would be ASTs for the statements.

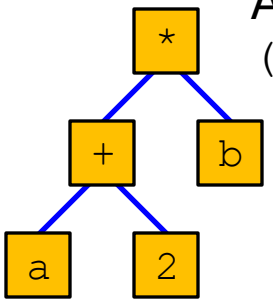
Back to Example #3: Expressions — ASTs [4/4]

The subtrees of an internal node in an AST are ASTs themselves. So we can build an AST from smaller ASTs in the same way we build an expression from smaller expressions.

AST for `a + 2`



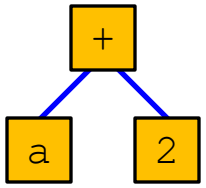
AST for `(a + 2) * b`

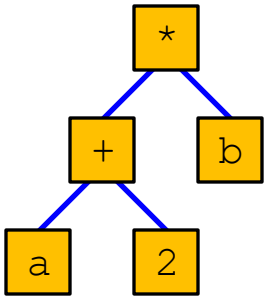


Observe that ASTs omit syntax that only serves to guide the parser—like semicolons and parentheses in a typical PL.

There is no universal specification for an AST. Instead, the ASTs used in a software project are specified in a way that meets the needs of that project.

Back to Example #3: Expressions — Representing ASTs [1/6]





We need to represent ASTs like these in Lua.

- Represent a single node by the string form of its lexeme.
- If there is more than one node in an AST, then represent the AST as an array whose first item represents the root, while each remaining item represents one of the subtrees of the root, in order.

The first AST, in Lua: `{ "+", "a", "2" }`

TRY IT. What is the Lua representation of the second AST?

Answer on next slide.

Recursive-Descent Parsing
Back to Example #3: Expressions — Representing ASTs [2/6]

We need to represent ASTs like these in Lua.

- Represent a single node by the string form of its lexeme.
- If there is more than one node in an AST, then represent the AST as an array whose first item represents the root, while each remaining item represents one of the subtrees of the root, in order.

The first AST, in Lua: `{ "+", "a", "2" }`

TRY IT. What is the Lua representation of the second AST?

Answer: `{ "*", { "+", "a", "2" }, "b" }`

Back to Example #3: Expressions — Representing ASTs [3/6]

It is better to describe our ASTs in a way that does not require tree drawings. So we specify the format of an AST for each line in our grammar. (Lines are numbered so we can refer to them.)

Grammar 3b

1. *expr* → *term* { ("+" | "–") *term* }
2. *term* → *factor* { ("*" | "/") *factor* }
3. *factor* → ID
4. | NUMLIT
5. | "(" *expr* ")"

1. If there is only a *term*, then the AST for the *expr* is the AST for the *term*. Otherwise, the AST is { *OO*, *AAA*, *BBB* }, where *OO* is the string form of the last operator, *AAA* is the AST for everything before it, and *BBB* is the AST for the last *term*.

Back to Example #3: Expressions — Representing ASTs [4/6]

A *term* is handled similarly.

Grammar 3b

1. *expr* → *term* { ("+" | "-") *term* }
2. *term* → *factor* { ("*" | "/") *factor* }
3. *factor* → ID
4. | NUMLIT
5. | "(" *expr* ")"

2. If there is only a *factor*, then the AST for the *term* is the AST for the *factor*. Otherwise, the AST is { *OO*, *AAA*, *BBB* }, where *OO* is the string form of the last operator, *AAA* is the AST for everything before it, and *BBB* is the AST for the last *factor*.

Back to Example #3: Expressions — Representing ASTs [5/6]

A *factor* has multiple options.

Grammar 3b

1. *expr* → *term* { ("+" | "-") *term* }
2. *term* → *factor* { ("*" | "/") *factor* }
3. *factor* → ID
4. | NUMLIT
5. | "(" *expr* ")"

3. AST for the *factor*: string form of the ID.
4. AST for the *factor*: string form of the NUMLIT.
5. AST for the *factor*: AST for the *expr*.

Back to Example #3: Expressions — Representing ASTs [6/6]

Grammar 3b

1. *expr* → *term* { ("+" | "-") *term* }
2. *term* → *factor* { ("*" | "/") *factor* }
3. *factor* → ID
4. | NUMLIT
5. | "(" *expr* ")"

Applying the various rules, the AST for `(a + 2) * b` is

```
{ "*", { "+", "a", "2" }, "b" }
```

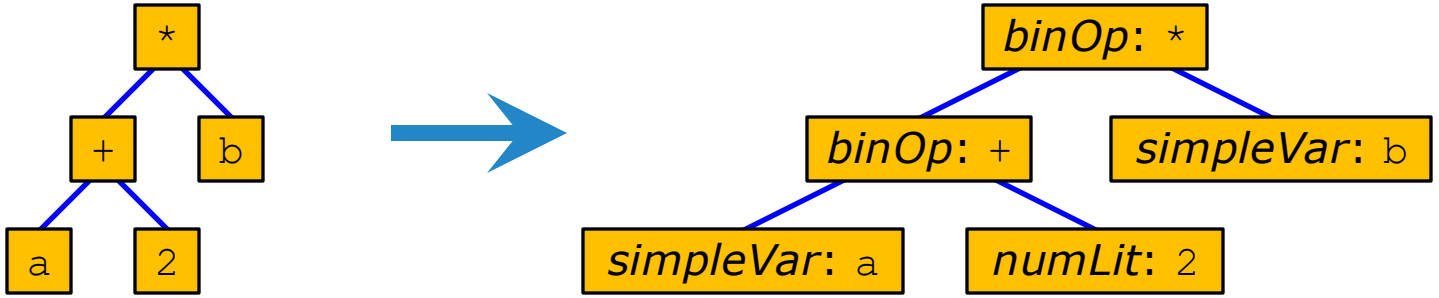
Same as before, but this time we did not need to draw a tree.

Each parsing function can now return a pair: a Boolean and an AST. The Boolean indicates a correct parse, as before. The AST is only valid if the Boolean is `true`, in which case it will be in the specified format.

Back to Example #3: Expressions — Better ASTs [1/3]

Our ASTs are not quite good enough. When a compiler or interpreter uses an AST, it needs to know what kind of entity each node represents. The parser knows this; we can include the information in each node.

We have three kinds of nodes: binary operators, simple variables, and numeric literals.



Back to Example #3: Expressions — Better ASTs [2/3]

In the Lua form of our AST, replace each string with a two-item array. The first item in the array will be one of three constants: `BIN_OP`, `SIMPLE_VAR`, or `NUMLIT_VAL`. The second item will be the string form of the lexeme.

```
"/"                    { BIN_OP, "/" }
"abc"          →       { SIMPLE_VAR, "abc" }
"123"                  { NUMLIT_VAL, "123" }
```

So the AST for `a + 2` changes as shown below.

```
{ "+", "a", "2" }      {{ BIN_OP, "+" },
                        { SIMPLE_VAR, "a" },
                        { NUMLIT_VAL, "2" }}
```

Recursive-Descent Parsing
Back to Example #3: Expressions — Better ASTs [3/3]

```
"/"                { BIN_OP, "/" }
"abc"              { SIMPLE_VAR, "abc" }
"123"              { NUMLIT_VAL, "123" }
```

TO DO

- Based on Grammar 3b, write a Predictive Recursive-Descent parser that constructs and returns these improved ASTs.

Done. See `rdparser3.lua` *&* `use_rdparser3.lua`*.*