Lecture 8

Grammars and Parsers

grammar and derivations, recursive descent parser vs. CYK parser, Prolog vs. Datalog

Ras Bodik
with Ali & Mangpo

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2013
UC Berkeley

Outline

Grammars: a concise way to define program syntax

Parsing: recognize syntactic structure of a program

Parser 1: *recursive descent* (backtracking)

Parser 2: *CYK* (dynamic programming algorithm)

Note: this file includes useful hidden slides which do not show in the PowerPoint Slide View.

Why parsing?

Parsers making sense of these sentences:

This lecture is dedicated to my parents, Mother Teresa and the pope.

The (missing) serial comma determines whether M.T.&p. associate with “my parents” or with “dedicated to”.

Seven-foot doctors filed a law suit.

does “seven” associate with “foot” or with “doctors”?

`if` E_1 **`then`** { **`if`** E_2 **`then`** E_3 **`else`** E_4 }

typical semantics associates “else E_4” with the closest if (ie, “if E_2”)

In general, programs and data exist in text form

which needs to be understood by parsing (and converted to tree form)

The cs164 parsing story

From string generation to Earley parser

1. Write a **random expression generator.**

2. **Invert** this generator into a parser by inverting `print` into `scan` and `random()` into `askOacle()`. The oracle constructs the parse tree.

3. Rewrite this parser in Prolog, which serves as your oracle.
   This gives you the ubiquitous **recursive descent parser**. Time = O(2^n)

4. Observe that this Prolog parser has no negation. It's in a **Datalog** subset of Prolog (more or less).

5. Datalog programs are evaluated bottom-up (dynamic programming).
   Rewriting the Prolog parser into Datalog gives us the **CYK parser**. O(n^3)

6. Datalog evaluation can be optimized with the Magic Set Transformation, which gives us **Earley Parser**. (Covered in Lecture 9.) O(n) with a suitable grammar. Earley is the basis for all efficient modern parsers.

Grammars

Grammar: a recursive definition of a language

Language: a set of (desired) strings

Example: the language of Regular Expressions (*RE*).

RE can be defined as a grammar:

base case: any input character c is *regular expression*;

inductive case: if e_1, e_2 are regular expressions, then the following four are also regular expressions:

$e_1 | e_2$ $e_1 e_2$ e_1^* (e_1)

red

Example:

a few strings in this language: a a|a a|b*

a few strings not in this language: *X empty string (ε) *|* (a))

Terminals, non-terminals, productions

The *grammar* notation:

R ::= c | R R | R|R | R* | (R)

terminals (red): input characters

also called the alphabet of the of the language

non-terminals: substrings in the language

these symbols will be rewritten to terminals

start non-terminal: starts the derivation of a string

convention: always the first nonterminal mentioned

productions: rules governing string derivation

RE has five: R ::= c, R ::= R R, R ::= R|R, R ::= R*, R ::=(R)

It’s *grammar*, not *grammer.*

“Not all writing is due to bad grammer.” (sic)

Saying “grammer” is a lexical error, not a syntactic (ie, grammatic) one.

In the compiler, this error is caught by the lexer.

lexer fails to recognize “grammer” as being in the lexicon.

In cs164, you learn which part of compiler finds errors.

lexer, parser, syntactic analysis, or runtime checks?

Deriving a string from a grammar

How is a string derived in a grammar:

S

1. write down the start non-terminal *S*
2. rewrite *S* with the rhs of a production *S* → *rhs*
3. pick a non-terminal *N*
4. rewrite *N* with the rhs of a production *N* → *rhs*
5. if no non-terminal remains, we have generated a string.
6. otherwise, go to 3.

Example:

grammar G: E ::= T | T + E   T ::= F | F * T   F ::= a | (E)

derivation of a string from *L*(*G*): ~~S~~ E $\rightarrow \mathrm{T} + \mathrm{E} \rightarrow \mathrm{F} + \mathrm{E} \rightarrow \mathrm{a} + \mathrm{E}$

$\rightarrow \mathrm{a} + \mathrm{T} \rightarrow \mathrm{a} + \mathrm{F} \rightarrow \mathrm{a} + \mathrm{a}$

Left- and right-recursive grammars

Grammars vs. languages

ba baa
baaa

Write a grammar for the language *all* strings ba^i, i>0.

grammar 1: S ::= Sa | ba

grammar 2: S ::= baA   A ::= aA | ε ← empty string

A language can be described with multiple grammars

L(G) = language (strings) described by grammar G

in our example, L(grammar 1) = L(grammar 2)

Left recursive grammar: X ::= Xa

X ::= Za
Z ::= Xb

Right-recursive grammar: X ::= aX

both l-rec and r-rec: S ::= bA | baA
A ::= aAa | ε

Why care about left-/right-recursion?

Some parser can’t handle left-recursive grammars.

It may get them into infinite recursion.

Same principle as in Prolog programs that do not terminate.

Luckily, we can rewrite a l-rec grammar into a r/r one.

while describing the same language

Example 1:

S ::= Sa | a   can be rewritten to   S ::= aS | a

The typical expression grammar

A grammar of expressions:

G_1: E ::= n | E + E | E * E | (E)

G_1 is l-rec but can be rewritten to G_2 which is not

G_2: E ::= T | T + E

T ::= F | F * T

F ::= n | (E)

In addition to removing left recursion, nonterminals **T** (a term) and **F** (a factor) introduce desirable precedence and associativity. More in L9.

Is $L(G_1)=L(G_2)$?

That is, are these same sets of string? Yes.

The parsing problem

What the parser does

The *syntax-checking* parsing problem:

given an input string s and grammar G, check if $s \in L(G)$

The *parse-tree* parsing problem:

given an input string $s \in L(G)$, return the parse tree of s

Really

A Poor Man's Parser

Generate-and-test "parser"

We want to test if $s \in L(G)$. Our "algorithm":

- print a string $p \in L(G)$, check if $s = p$, repeat

The plan:

Write a function gen(G) that prints a string *p*∈ *L*(G).

If L(G) is finite, gen(G) will eventually print all strings in L(G).

Does this algorithm work?

Depends if you are willing to wait. ☺
Also, *L*(G) *may be infinite.*

This parser is useful only for instructional purposes

in case it's not clear already

gen(G)

Grammar G and its language *L(G)*:

G: E ::= a | E + E | E * E

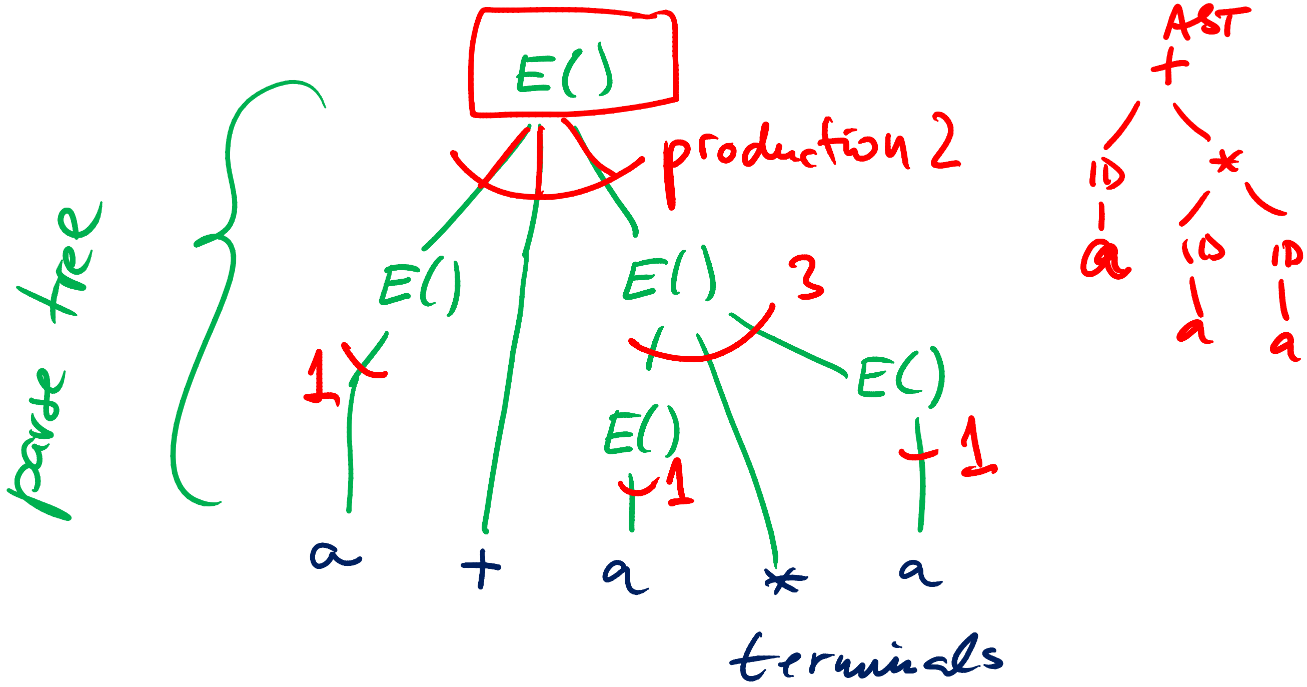
L(G) = { a, a+a, a*a, a*a+a, … }

For simplicity, we hardcode G into gen()

```
def gen() {  E(); print EOF }
def E() {
    switch (choice()):
    case 1: print "a"
    case 2: E(); print "+"; E()
    case 3: E(); print "*"; E()
}
```

Visualizing string generation with a **parse tree**

The tree that describe string derivation is *parse tree*.



Are we generating the ~~string~~ p.tree top-down or bottom-up?

Top-down. Can we do it other way around? Sure. See CYK.

Parsing

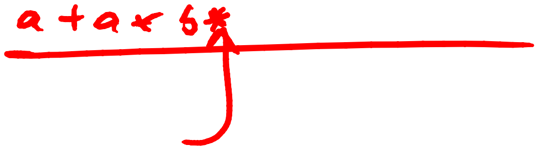
Parsing is the inverse of string generation:

given a string, we want to find the parse tree

If parsing is just the inverse of generation, let’s obtain the parser mechanically from the generator!

```
def gen() {  E(); ~~print~~ scan EOF }
def  E() {
   switch (~~choice()~~ oracle()):
   case 1: ~~print~~ scan “a"
   case 2: E(); ~~print~~ scan "+"; E()
   case 3: E(); ~~print~~ scan "*"; E()
}
```

Generator vs. parser



```
def gen() {  E(); print EOF }
def E() {  switch (choice()) {
              case 1: print “a"
              case 2: E(); print "+"; E()
              case 3: E(); print "*"; E() }}

def parse() {  E(); scan(EOF) }
def E() {  switch (oracle()) {
              case 1: scan("a")
              case 2: E(); scan("+"); E()
              case 3: E(); scan("*"); E() }}
def scan(s) { if rest of input starts with s,
              consume s; else abort }
```

Reconstruct the Parse Tree

Parse tree

Parse tree: shows how the string is derived from G

- *leaves:* input characters
- *internal nodes:* non-terminals
- *children of an internal node:* production used in derivation

Why do we need the parse tree?

We evaluate it to obtain the AST, or sometimes to directly compute the value of the program.

Test yourself: construct the AST from a parse tree.

Example: evaluate an expression on parse tree

Input: 2 * (4 + 5)

Grammar:

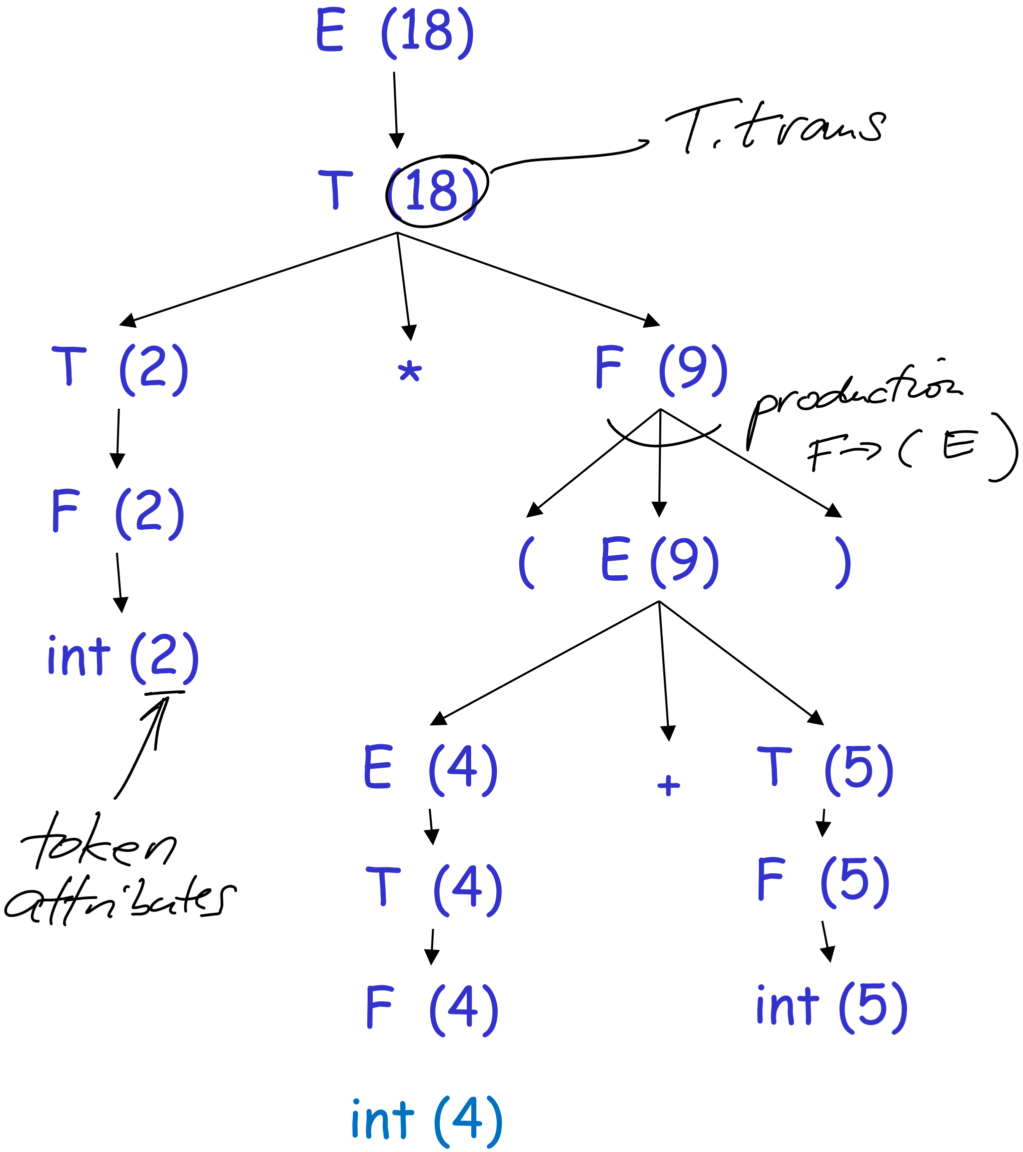
E ::= T | T + E

T ::= F | F * T

F ::= n | (E)

Parse Tree

(annotated with values):



E (18)

T (18)

T.trans

T (2)

*

F (9)

production F→(E)

F (2)

(E (9))

int (2)

token attributes

E (4)

+

T (5)

T (4)

F (5)

F (4)

int (5)

int (4)

Parse tree vs. abstract syntax tree

Parse tree = *concrete* syntax tree

- contains all syntactic symbols from the input
- including those that the parser needs “only” to discover
  - intended nesting: parentheses, curly braces
  - statement termination: semicolons

Abstract syntax tree (AST)

- abstracts away these artifacts of parsing,
- abstraction compresses the parse tree
  - flattens parse tree hierarchies
  - drops tokens

Add parse tree reconstruction to our parser

```
def parse() {  root = E(); scan(EOF);
               return root }
def E() {
   switch (oracle()) {
   case 1: scan("a")
           return (“a”,)
   case 2: left = E()
           scan("+")
           right = E()
           return (“+”, left, right)
   case 3: // analogous
}}
```

Python tuple

Recursive Descent Parser

(by implementing the oracle with Prolog)

How to implement our oracle? (hidden slide)

Recall *amb*: the nondeterministic evaluator from cs61A

(amb 1 2 3 4 5) evaluates to 1 or .. or 5

Which option does amb choose? One leading to success.

in our case, success means parsing successfully

How was amb implemented?

backtracking

Our parser with amb:

```
def E() { switch (amb(1,2,3)) {
                case 1: scan("a“)
                case 2: E(); scan("+“); E()
                case 3: E(); scan("*"); E() }}
```

Note: amb may not work with any left-recursive grammar

How do we implement the oracle

We could implement it with coroutines.

We'll use use **logic programming** instead.

  After all, we already have oracle functionality in our Prolog

We will define a parser as a logic program

  backtracking will give it exponential time complexity

Backtracking parser in Prolog

Example grammar:

```
E ::= a
E ::= a + E
```

We want to parse a string a+a, using a query:

```
?- parse([a,+,a]).
true
```

⇒ a+a <u>is</u> in L(G)

Backtracking Prolog parser for this grammar

```
e([a|Out], Out).
e([a,+|R], Out) :- e(R,Out).
parse(S) :- e(S,[]).
```

How does this parser work? (1)

Let’s start with simple Prolog queries:

```
?- [H | T] = [a,+,a].
H = a,
T = [+, a].

       a
?- [a,+,b,+,c]=[a, + | Rest].
Rest = [b, +, c].
```

less concise

a(I,O) :- I=[a|T],
O=T.

How does this parser work? (2)

Let’s start with this (incomplete) grammar:

e([a|T], T).

E ::= a

Sample queries:

e([a,+,a],Rest). } corresponds to scan('a')

--> Rest = [+,a]

e([a],Rest).

-->Rest = []

e([a],[]).

like doing scan (EOF) at end parse

--> true // parsed successfully

Parser for the full expression grammar

E = T | T + E   T = F | F * T   F = a

```
e(In,Out) :- t(In, Out).
e(In,Out) :- t(In, [+|R]), e(R,Out).

t(In,Out) :- f(In, Out).
t(In,Out) :- f(In, [*|R]), t(R,Out).

f([a|Out],Out).

parse(S) :- e(S,[]).

?- parse([a,+,a,*,a],T). --> true
```

Construct also the parse tree

E = T | T + E   T = F | F * T   F = a

```
e(In,Out,e(T1))       :- t(In, Out, T1).
e(In,Out,e(T1,+,T2)) :- t(In, [+|R], T1), e(R,Out,T2).
t(In,Out,e(T1))       :- f(In, Out, T1).
t(In,Out,e(T1,*,T2)) :- f(In, [*|R], T1), t(R,Out,T2).
f([a|Out],Out,a).

parse(S,T) :- e(S,[],T).

?- parse([a,+,a,*,a],T).
T = e(e(a), +, e(e(a, *, e(a))))
```

bugs? see L9 for a corrected version.

Construct also the AST

E = T | T + E   T = F | F * T   F = a

```
e(In,Out,T1)          :- t(In, Out, T1).
e(In,Out,plus(T1,T2)) :- t(In, [+|R], T1), e(R,Out,T2).
t(In,Out,T1)          :- f(In, Out, T1).
t(In,Out,times(T1,T2)):- f(In, [*|R], T1), t(R,Out,T2).
f([a|Out],Out, a).

parse(S,T) :- e(S,[],T).

?- parse([a,+,a,*,a],T).
T = plus(a, times(a, a))
```

Running time of the backtracking parser

We can analyze either version. They are the same.

amb:

```
def E() { switch (oracle(1,2,3)) {
            case 1: scan("a“)
            case 2: E(); scan("+“); E()
            case 3: E(); scan("*"); E() }}
```

Prolog:

```
e(In,Out) :- In==[a|Out].
e(In,Out) :- e(In,T1), T1==[+|T2], e(T2,Out)
e(In,Out) :- e(In,T1), T1==[*|T2], e(T2,Out)
```

Recursive descent parser

This parser is known as recursive descent parser (rdp)

The parser for the calculator (Lec 2) is an *rdp*.

Study its code. rdp is *the* way to go when you need a small parser.

Crafting its grammar carefully removes exponential time complexity.

> Because you can avoid backtracking by facilitating making choice between rules based on immediate next input. See the calculator parser.

Summary

Summary

Languages vs grammars

a language can be described by many grammars

Grammars

string generation vs. recognizing if string is in grammar

random generator and its dual, oracular recognizer

Parse tree:

result of parsing is parse tree

Recursive descent parser

runs in exponential time.