Lecture 8

Parsers

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

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2012

UC Berkeley

Administrativia

You will earn PA extra credit for bugs in solutions, starter kits, handouts.

Today is back-to-basic Thursday.

We have some advanced material to cover.

Today: Parsing

Why parsing? Making sense out 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 to "my parents" or to "dedicated to".

Seven-foot doctors filed a law suit.

the dash associates "seven" to "foot" rather than to "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 need to be understood by parsing

The cs164 concise parsing story

Courses often spend two weeks on parsing. CS164 deals with parsing in 2 lectures, and teaches non-parsing lessons along the way.

1. Write a **random expression generator.**
2. **Invert** this recursive generator into a parser by replacing `print` with `scan` and `random` with `oracle`.
3. Now rewrite write this parser in Prolog, which is your oracle. This gives you the ubiquitous **recursive descent parser**.
4. An observation: this Prolog parser has no negation. It's in **Datalog**!
5. Datalog programs are evaluated bottom-up (dynamic programming). Rewriting the Prolog parser into Datalog thus yields **CYK parser**.
6. Datalog evaluation can be optimized with a Magic Set Transformation, which yields **Earley Parser**. (Covered in Lecture 9.)

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 character c is *regular expression*;
- inductive case: if e1, e2 are regular expressions then the following are also regular expressions:

  e1 | e2   e1 e2   e1*   (e1)

Example:

- a few strings in this language: a , b , a|b , (a* | a|b)*
- a few strings not in this language: || , a||b , |a , *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: will be rewritten to terminals

convention: capitalized

start non-terminal: starts the derivation of a string

convention: s.n.t. is always the first nonterminal mentioned

productions: rules that governs string derivation

ex 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?

Grammars vs. languages

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

grammar 1: S ::= Sa | ba

empty string

grammar 2: S ::= baA A ::= aA | ε

A language can be described with multiple grammars

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

Left recursive grammar: X ::= Xa | a

Right-recursive grammar: X ::= aX | a

neither: S ::= bA | baA

A ::= aAa | ε

Why do we care about left-/right-recursion?

Some parser can't handle left-recursive grammars.

  It may get them into infinite recursion.

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

Example 1:

  S ::= Sa | a   is rewritten into   S ::= aS | a

Example 2:

  E ::= a | E + E | E * E | (E)

  becomes

  E ::= T | T + E

  T = F | F * T

  F = a | (E)

T (a term) and F (a factor) introduce desirable precedence and associativity. More in L9.

Deriving a string from a grammar

How is a string derived in a grammar:

1. write down the start non-terminal *S*
2. rewrite *S* with the rhs of a production $S \to rhs$
3. pick a non-terminal *N*
4. rewrite *N* with the rhs of a production $N \to rhs$
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)*: $S \to T + E \to F + E \to a + E$

$\to a + T \to a + F \to a + a$

Generate a string from L(G)

Is there a recipe for printing all strings from L(G)?

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

Write function gen(G) that prints a string $s \in L(G)$.

If L(G) is finite, rerunning gen(G) should eventually print any string in L(G).

gen(G)

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

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

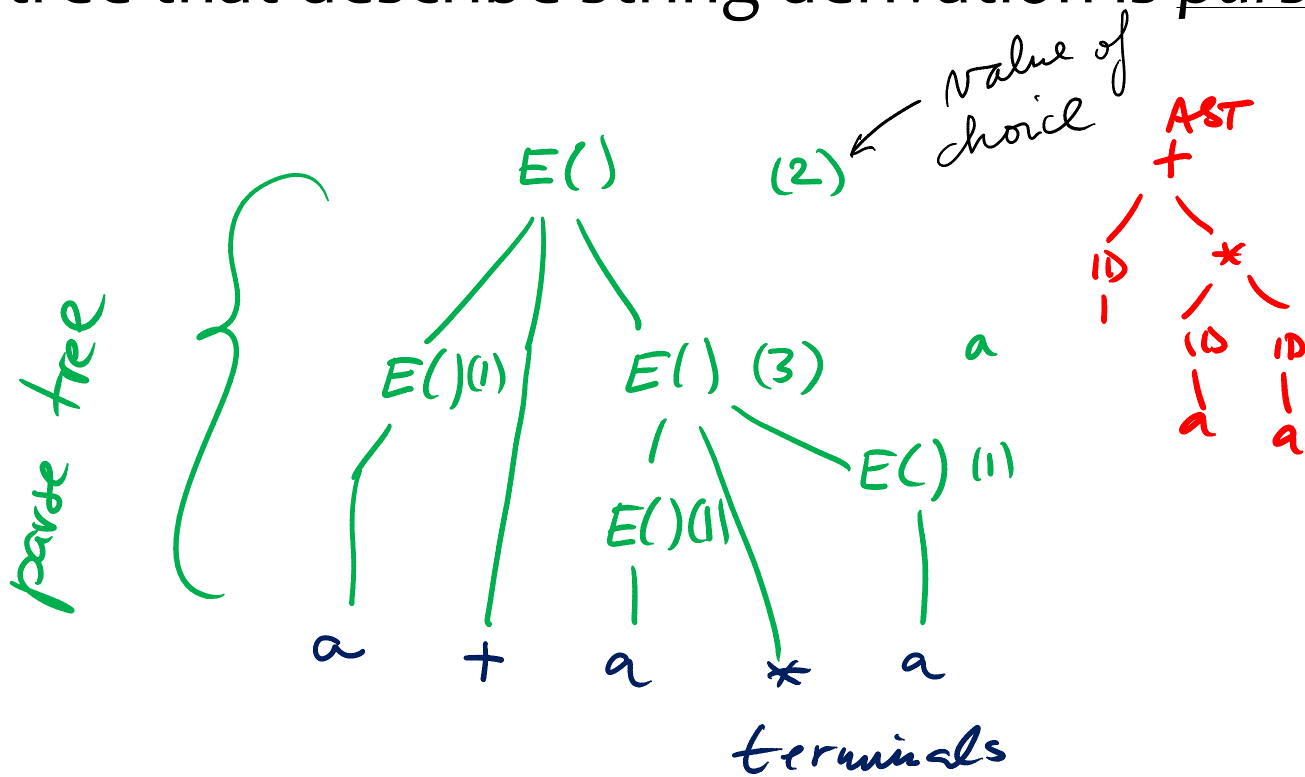
L(G) = { a, a+a, a*a, a*a+a, … }

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 <u>*parse tree*</u>.



Are we generating the string 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 <u>mechanically</u> from the generator!

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

helps us consume entire string

scan

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 input starts with s,
               consume s; else abort }
```

if input is in L(G), oracle will avoid abort.

Parsing == reconstruction of the parse tree

Why do we need the parse tree?

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

Next slide shows use of parse tree for evaluation.

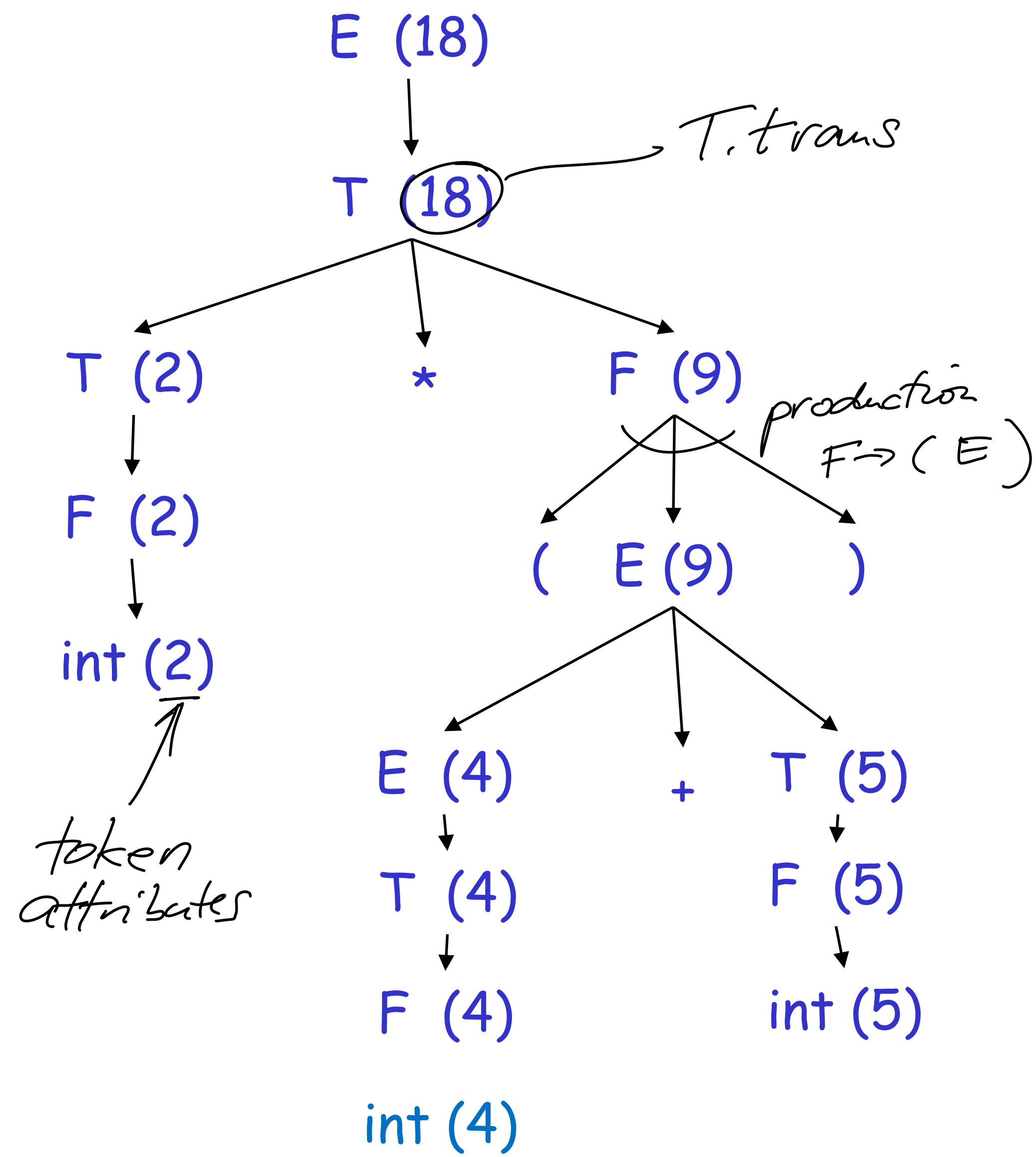
Exercise: construct AST from a parse tree.

Example 1: evaluate an expression (calculator)

Input: 2 * (4 + 5)

parse tree for this input w.r.t. grammar on next slide

Annotated Parse Tree:



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 the 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

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

Next we observe that the parser has special structure

- and permits polynomial time algorithm (via Datalog)

Backtracking parser in Prolog

Our grammar:

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

Backtracking parser for this grammar in Prolog

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

To parse, run query:

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

⇒ a+a is in L(G)

How does this parser work?

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))))
```

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.

3 choices

depth of recursion is n, the length of input

$\Rightarrow O(3^n)$

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.

CYK parser

(can we run our parser in polynomial time?)

Datalog: a well-behaved subset of Prolog

From wikipedia: Query evaluation in Datalog is based on first order logic, and is thus sound and complete.

- See *The Art of Prolog* for why Prolog is not logic (Sec 11.3)

Datalog is a restricted subset of Prolog

- disallows compound terms as arguments of predicates
  - p(1, 2) is admissible but not p(f1(1), 2). Hence can't use lists.
- only allows range-restricted variables, a(X) :- b(X) ok
  - each variable in the head of a rule must also appear in a not-negated clause in the body of this rule. Hence we can compute values of variables from ground facts.
- imposes stratification restrictions on the use of negation
  - it's sufficient that we don't use negation

Why do we care about Datalog?

Predictable semantics:

- Restrictions make the set of all possible proofs finite, with the consequence that all Datalog programs terminate (unlike Prolog programs).

Efficient evaluation:

- Uses bottom-up evaluation (dynamic programming).
- Various methods have been proposed to efficiently perform queries, e.g. the Magic Sets algorithm,[3]

If interested, see more in wikipedia.

More why do we care about Datalog?

We mechanically derive a famous parsing algorithms.

- Mechanically, without thinking too hard.
- Indeed, the rest of the lecture is about

1) CYK == Datalog version of Prolog recursive descent
2) Earley == Magic Set transformation of CYK

A bigger lesson:

- restricting your language may give you desirable properties
  - Just think how much easier your PA1 interpreter would be to implement without having to support recursion. Although it would be much less useful without recursion. Luckily, with Datalog, we don't lose anything when it comes to parsing.

Turning our Prolog parser into Datalog

Recursive descent in Prolog, for E ::= a | a+E

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

Let's check the datalog rules:

- No negation: check
- Range restricted: check
- Compound predicates: nope (uses lists)

[a|Out] is cons(a,Out)

Turning our Prolog parser into Datalog, cont.

Let’s refactor the program a little, using the grammar

E --> a | E + E | E * E

Yes, with Datalog, we can use left-recursive grammars!

Datalog parser: `e(i,j)` is true iff the input[i:j] can be derived (ie generated) from the non-terminal E.

`input[i:j]` is input from index i to index j-1

```
e(I,I+1) :- input[I]==‘a’.
e(I,J)   :- e(I,K), input[K]==‘+’, e(K+1,J).
e(I,J)   :- e(I,K), input[K]==‘*’, e(K+1,J).
```

Bottom-up evaluation of the Datalog program

Input:

a + a * a

Let's compute which facts we know hold

we'll deduce facts gradually until no more can be deduced

Step 1: base case (process input segments of length 1)

e(0,1) = e(2,3) = e(4,5) = true

Step 2: inductive case (input segments of length 3)

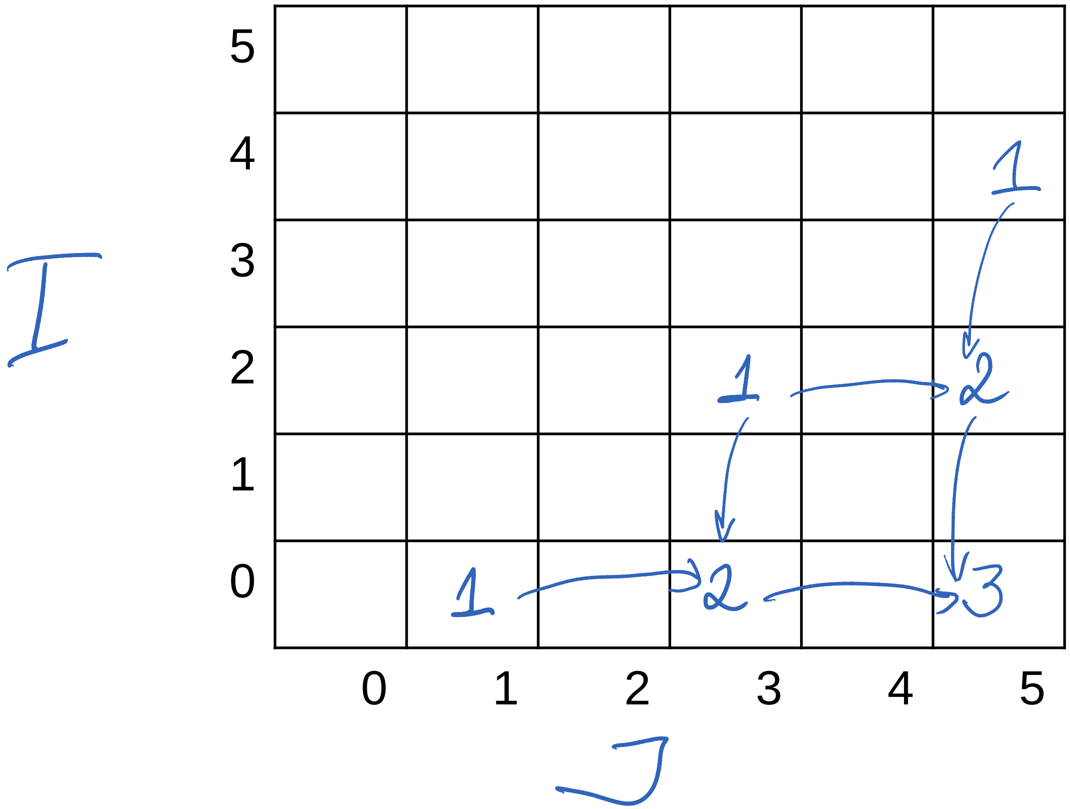
e(0,3) = true // using rule #2

e(2,5) = true // using rule #3

Step 2 again: inductive case (segments of length 5)

e(0,5) = true // using either rule #2 or #3

Visualize this parser in tabular form



Legend: # is step in which truth of fact e(I,J) was discovered.

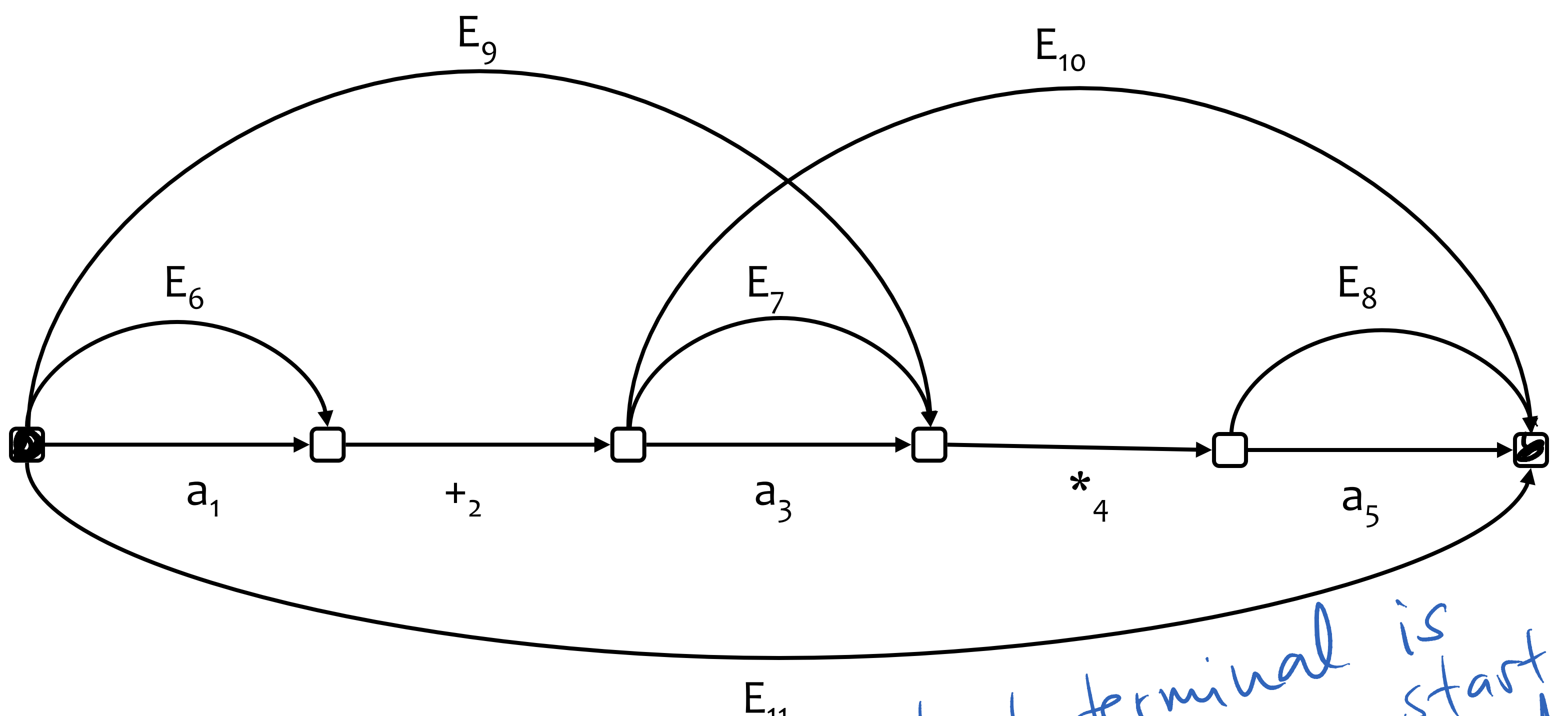
→ : deduction flow

A graphical way to visualize this evaluation

Initial graph: the input (terminals)

Repeat: add non-terminal edges until no more can be added.

An edge is added when adjacent edges form rhs of a grammar production.



Input: a + a * a

↑ start terminal is added between start and end ⇒ input is in L(G)

Home exercise: find the bug in this CYK algo

We assume that each rule is of the form A→BC, ie two symbols on rhs.

create a worklist of edges to process

```
for  i=0,N-1 do
    add (i,i+1,nonterm(input[i])) to graph  -- create nonterminal edges A→d
    enqueue( (i,i+1,nonterm(input[i])) )     -- nonterm() maps d to A !
while  queue not empty do
    (j,k,B)=dequeue()
    for each edge (i,j,A) do          -- for each edge "left-adjacent" to (j,k,B)
      if rule T→AB  exists then
        if edge e=(i,k,T) does not exists then  add e to graph; enqueue(e)
    for each edge (k,l,C) do          -- for each edge "right-adjacent" to (j,k,B)
      ... analogous ...
end while
if edge (0,N,S) does not exist  then "syntax error"
```

Constructing the parse tree

Nodes in parse tree correspond to edges in CYK reduction

- edge e=(0,N,S) corresponds to the root of parse tree r
- edges that caused insertion of e are children of r

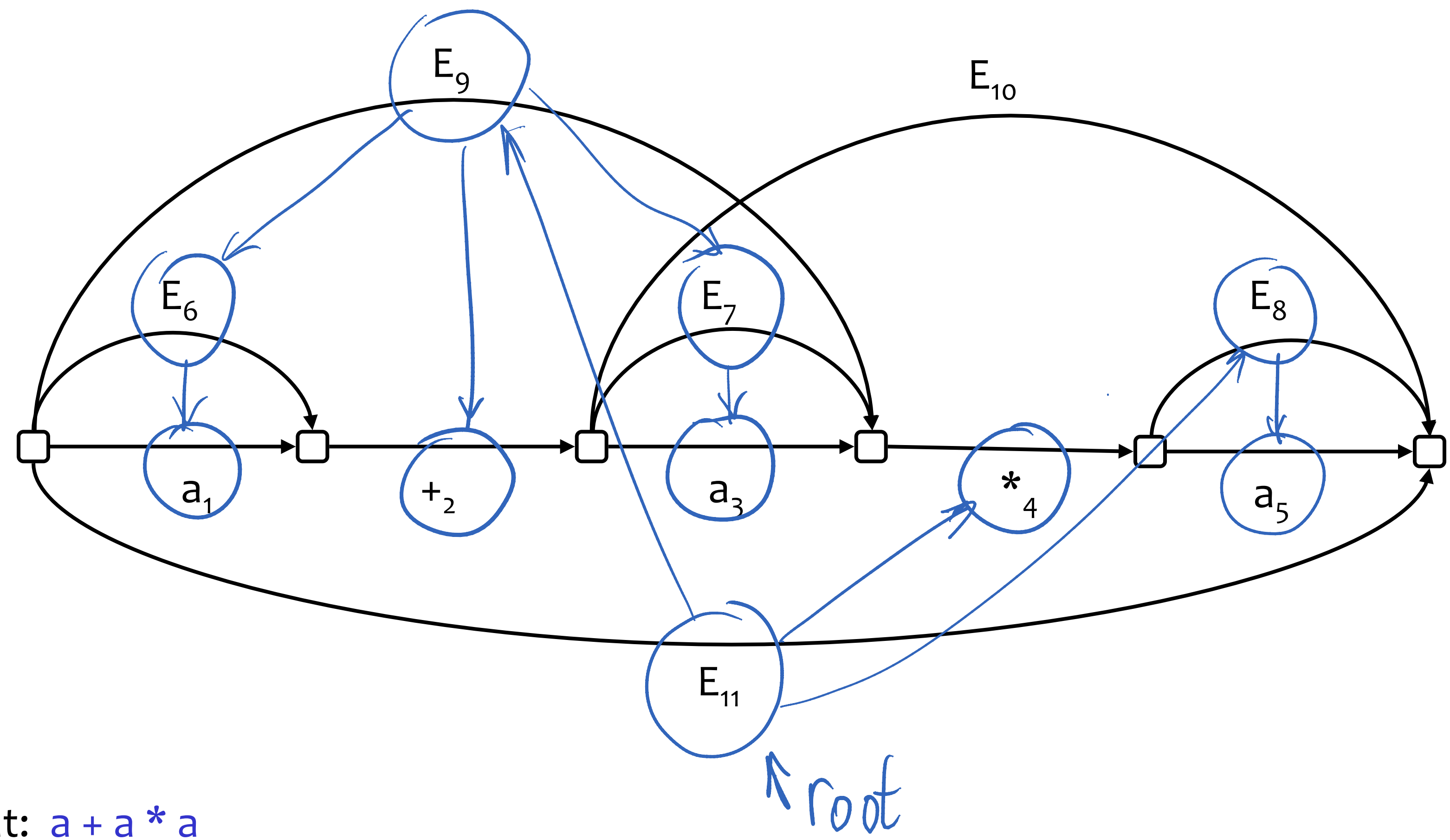
Helps to label edges with entire productions

- not just the LHS symbol of the production
- make symbols unique with subscripts
- such labels make the parse tree explicit

A graphical way to visualize this evaluation

Parse tree: o of the two parse trees is shown in blue.

another is not shown



Input: a + a * a

Summary

Languages vs grammars

a language can be described by many grammars

Prolog vs Datalog

top-down evaluation with backtracking vs bottom-up evaluation (table-filling dynamic programming)

Grammars

string generation vs. recognizing if string is in grammar

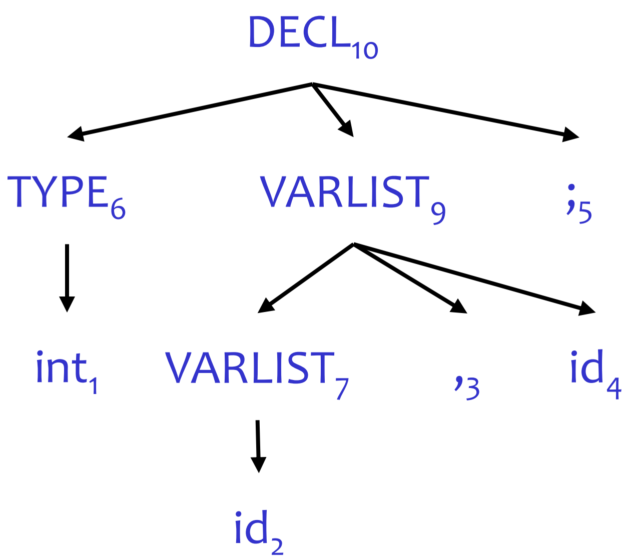
random generator and its dual, oracular recognizer

Parse tree:

result of parsing is parse tree

CYK is $O(N^3)$ time. Recursive descent is exp time.

Example of CYK execution



you should be able to reconstruct the grammar from this parse tree (find the productions in the parse tree)