Hack Your Language!

CSE401 Winter 2016

Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

Lecture 10: CYK and Earley Parsers

Building Parse Trees

CYK and Earley algorithms

More Disambiguation

Announcements

- HW3 due Sunday

- Project proposals due tonight
  - No late days

- Review session this Sunday 6-7pm EEB 115

Outline

- Last time we saw how to construct AST from parse tree

- We will now discuss algorithms for generating parse trees from input strings

Today

CYK parser

builds the parse tree bottom up

More Disambiguation

Forcing the parser to select the desired parse tree

Earley parser

solves CYK's inefficiency

CYK parser

Parser Motivation

- Given a grammar G and an input string s, we need an algorithm to:
  - Decide whether s is in L(G)
  - If so, generate a parse tree for s

- We will see two algorithms for doing this today
  - Many others are available
  - Each with different tradeoffs in time and space

CYK Algorithm

- Parsing algorithm for context-free grammars
- Invented by John **C**ocke, Daniel **Y**ounger, and Tadao **K**asami

- Basic idea given string **s** with **n** tokens:
  1. Find production rules that cover 1 token in s
  2. Use 1. to find rules that cover 2 tokens in s
  3. Use 2. to find rules that cover 3 tokens in s
  4. …

  N. Use N-1. to find rules that cover n tokens in s.
  If succeeds then s is in L(G), else it is not

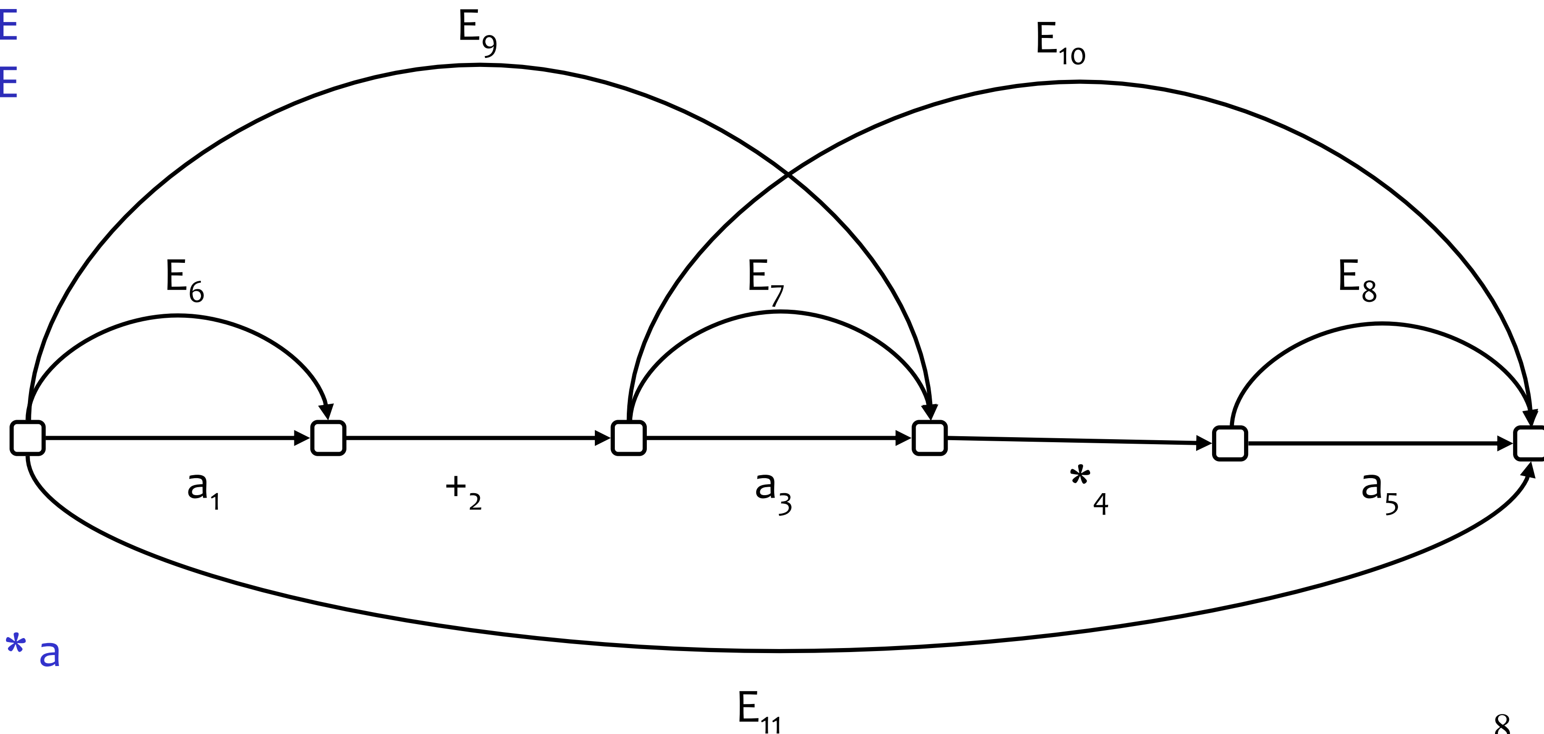
A graphical way to visualize CYK

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.

E → a
| E + E
| E * E



Input: a + a * a

CYK: the algorithm

CYK is easiest for grammars in Chomsky Normal Form

CYK is asymptotically more efficient in this form

$O(N^3)$ time, $O(N^2)$ space.

Chomsky Normal Form: only production rules of the following form are allowed:

$A \longrightarrow BC$ (A, B, C are non-terminals) or

$A \longrightarrow$ d (d is a terminal) or

$S \longrightarrow \varepsilon$ (only the start NT can derive the empty string)

All context-free grammars can be rewritten to this form

CYK Implementation

CYK uses a table e(N,N):

- set `e(i,j)` to true if the substring input[i:j] can be derived from the non-terminal E.
- `input[i:j]` is input from index i to index j-1

For the grammar

E → a | E + E | E * E

Rules:

```
e(I,I+1) if input[I]=='a'.
e(I,J)   if e(I,K), input[K]=='+', e(K+1,J).
e(I,J)   if e(I,K), input[K]=='*', e(K+1,J).
```

CYK Implementation

This is a form of **dynamic programming**

We use a table e(N,N) to store temporary results, and we use the temp results to compute new ones

Alternative: we take each rule **r** and check recursively whether **r** can be used to parse **s**

  But we will end up re-doing a lot of computation

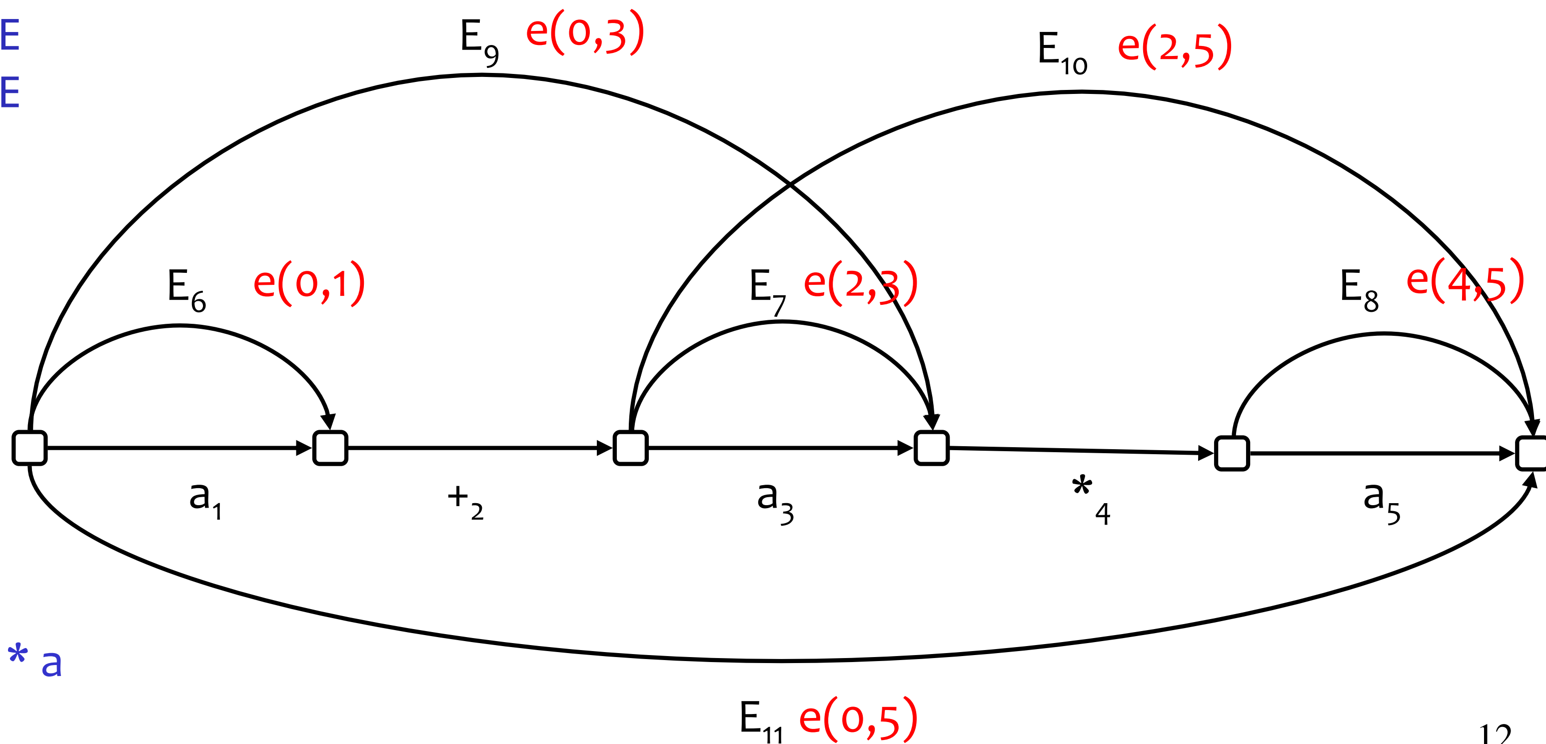
Illustration

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.

E → a
| E + E
| E * E



Input: a + a * a

CYK is dynamic programming

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 (segments of length 3)

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

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

Step 3: inductive case (segments of length 5)

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

Visualize this parser in tabular form

Step 1:

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

Step 2:

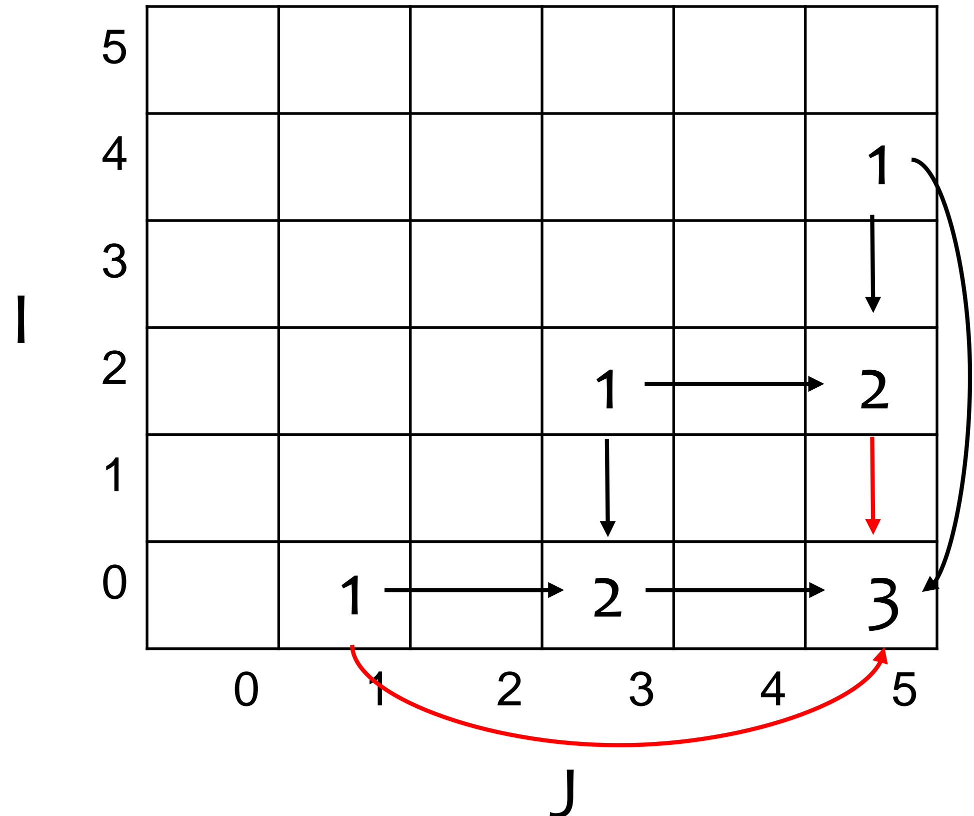
e(0,3) = true // using rule #2

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

Step 3:

e(0,5) = true

// using either rule #2 or #3



CYK Parser

Builds the parse tree bottom-up:

given a grammar with a rule A $\rightarrow$ B C,

whenever the parser finds adjacent B C edges in the "CYK graph",

it "reduces" B C to A,

adding the node A to the parse tree.

In the next lecture, we will also see top-down parsers.

these will be mostly based on backtracking

CYK Pseudocode

s = input string of *n* tokens: a_1 ... a_n.
r = # of non-terminal symbols in G: R_1 ... R_n

initialize all entries in P(N,N,r) = false
-- P(i,j,R_k) = R_k is used to parse input from i to j
for each *i* = 0 to *n-1*
  for each unit production $R_k \rightarrow a_i$
    P[*i*,*i+1*,*k*] = true

for each *i* = 2 to *n* -- *foreach rule that covers i tokens*
  for each *j* = 0 to *n-i* -- *start checking at pos j*
    for each *k* = 1 to *i*-1 -- *foreach partitioning of j to j+i*
      for each production rule $R_A \rightarrow R_B\ R_C$
        if *P*[*j*,*j+k*,*B*] and *P*[*j+k*,*j+i*,*C*] **then** *P*[*j*,*j+i*,*A*] = true

if any of *P*[0,*n-1*,*x*] is true **then** *s* is in L(G)
else *s* is not in L(G)

Illustration

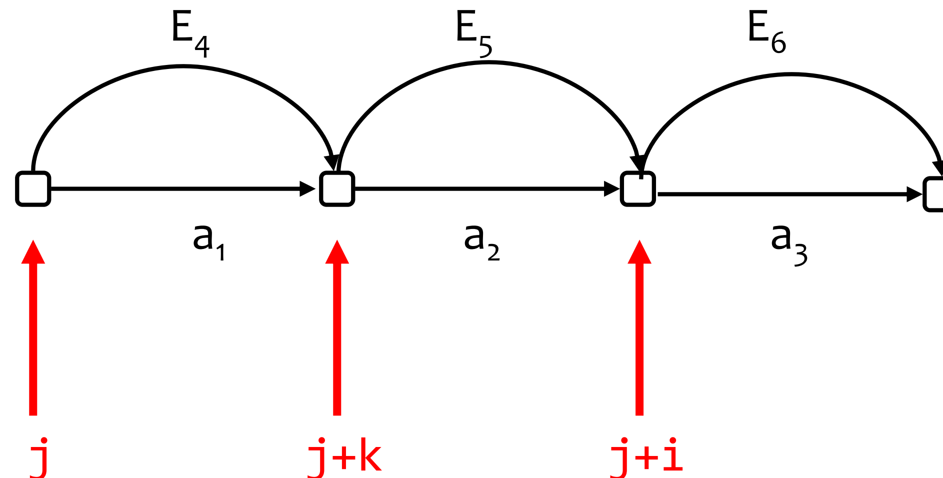
```
for each i = 2 to n
  for each j = 0 to n-i
    for each k = 1 to i-1
      for each production rule R_A → R_B R_C
        if P[j,j+k,B] and P[j+k,j+i,C]
        then P[j,j+i,A] = true
```

var	value
i	2
j	0
k	1

True P entries
P(0,1,E)
P(1,2,E)
P(2,3,E)

E → a
 | E E

Input: a a a



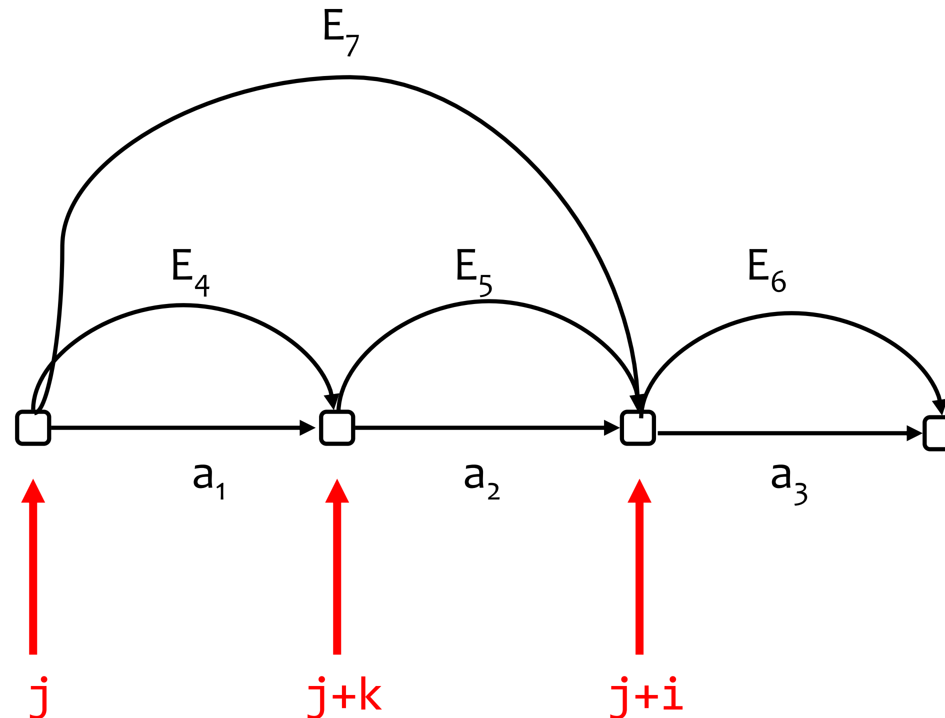
Illustration

```
for each i = 2 to n
  for each j = 0 to n-i
    for each k = 1 to i-1
      for each production rule R_A → R_B R_C
        if P[j,j+k,B] and P[j+k,j+i,C]
        then P[j,j+i,A] = true
```

var	value
i	2
j	0
k	1

True P entries
P(0,1,E)
P(1,2,E)
P(2,3,E)
P(0,2,E)

E → a
| E E



Input: a a a

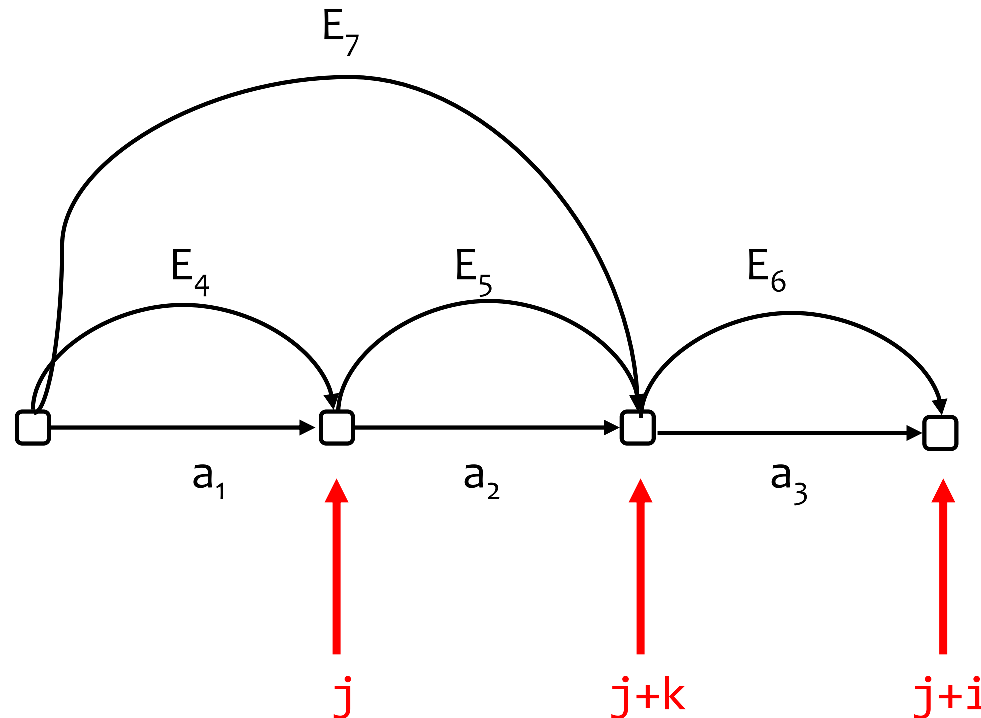
Illustration

```
for each i = 2 to n
  for each j = 0 to n-i
    for each k = 1 to i-1
      for each production rule R_A → R_B R_C
        if P[j,j+k,B] and P[j+k,j+i,C]
        then P[j,j+i,A] = true
```

var	value
i	2
j	1
k	1

True P entries
P(0,1,E)
P(1,2,E)
P(2,3,E)
P(0,2,E)

E → a
| E E



Input: a a a

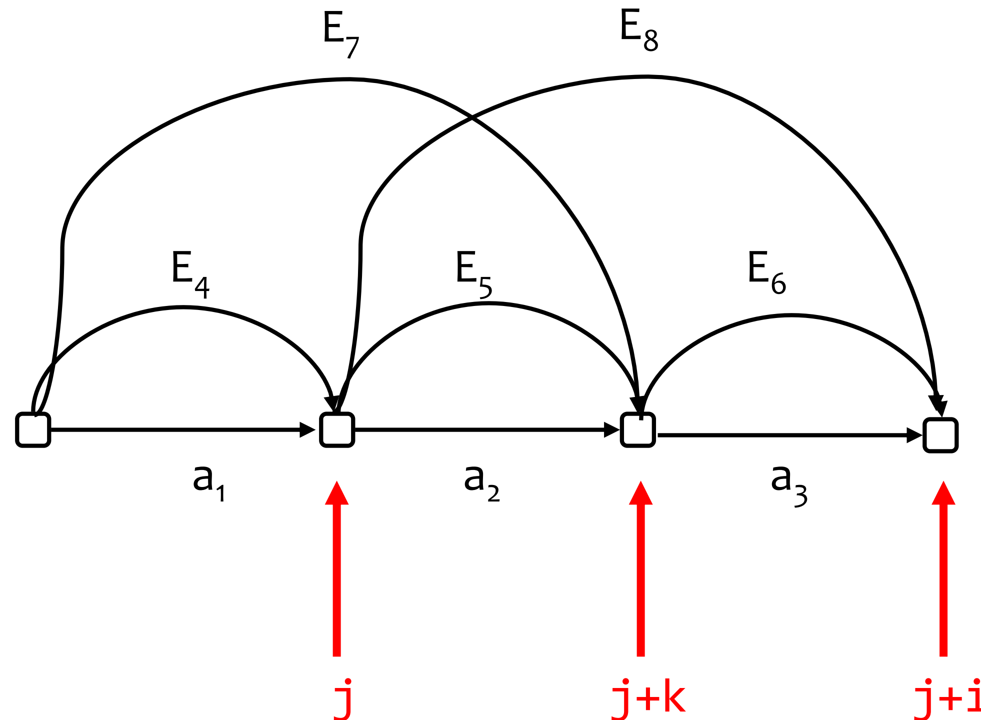
Illustration

```
for each i = 2 to n
  for each j = 0 to n-i
    for each k = 1 to i-1
      for each production rule R_A → R_B R_C
        if P[j,j+k,B] and P[j+k,j+i,C]
        then P[j,j+i,A] = true
```

var	value
i	2
j	1
k	1

True P entries
P(0,1,E)
P(1,2,E)
P(2,3,E)
P(0,2,E)
P(1,3,E)

E → a
| E E



Input: a a a

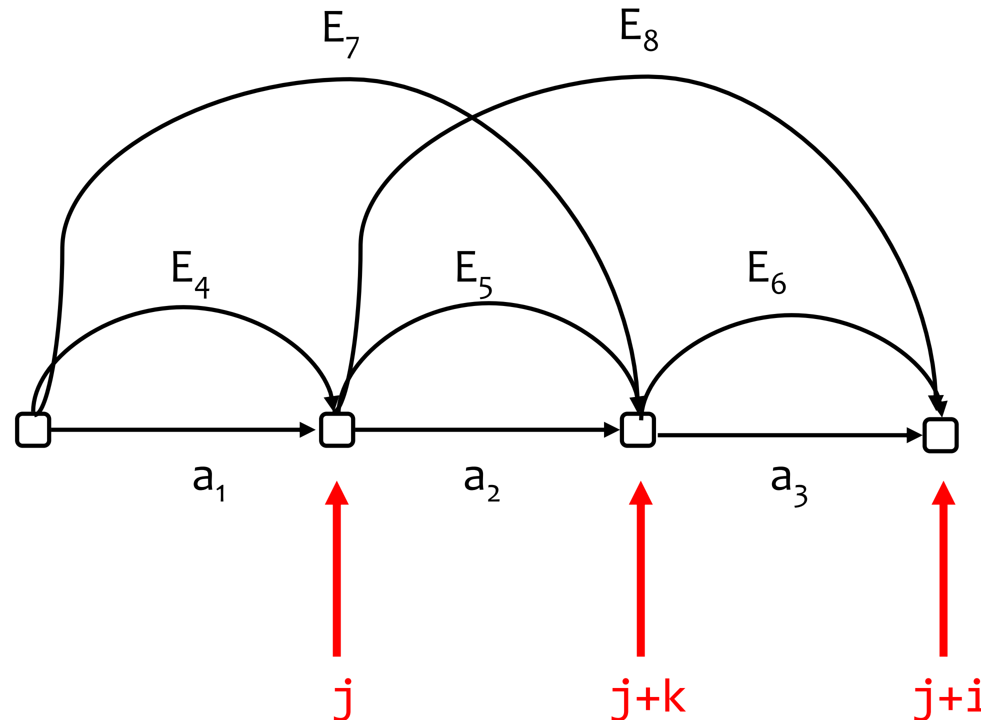
Illustration

```
for each i = 2 to n
  for each j = 0 to n-i
    for each k = 1 to i-1
      for each production rule R_A → R_B R_C
        if P[j,j+k,B] and P[j+k,j+i,C]
        then P[j,j+i,A] = true
```

var	value
i	3
j	
k	

True P entries
P(0,1,E)
P(1,2,E)
P(2,3,E)
P(0,2,E)
P(1,3,E)

E → a
  | E E



Input: a a a

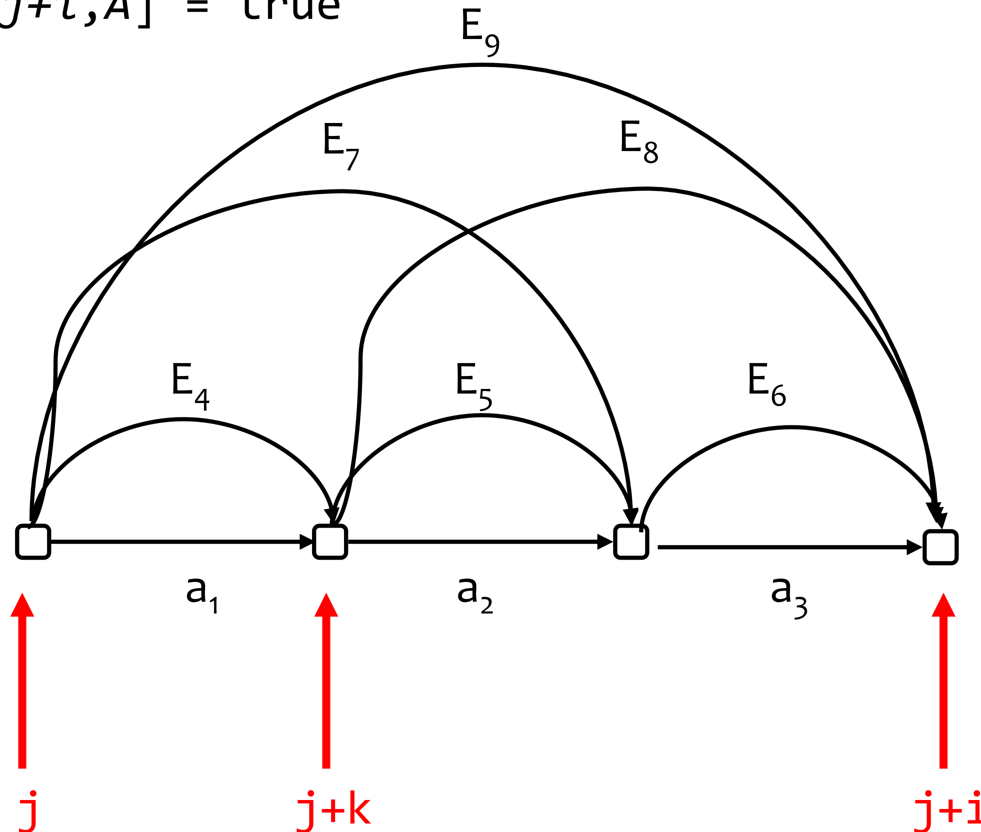
Illustration

```
for each i = 2 to n
  for each j = 0 to n-i
→   for each k = 1 to i-1
      for each production rule R_A → R_B R_C
        if P[j,j+k,B] and P[j+k,j+i,C]
        then P[j,j+i,A] = true
```

var	value
i	3
j	0
k	1

True P entries
P(0,1,E)
P(1,2,E)
P(2,3,E)
P(0,2,E)
P(1,3,E)
P(0,3,E)

E → a
| E E



Input: a a a

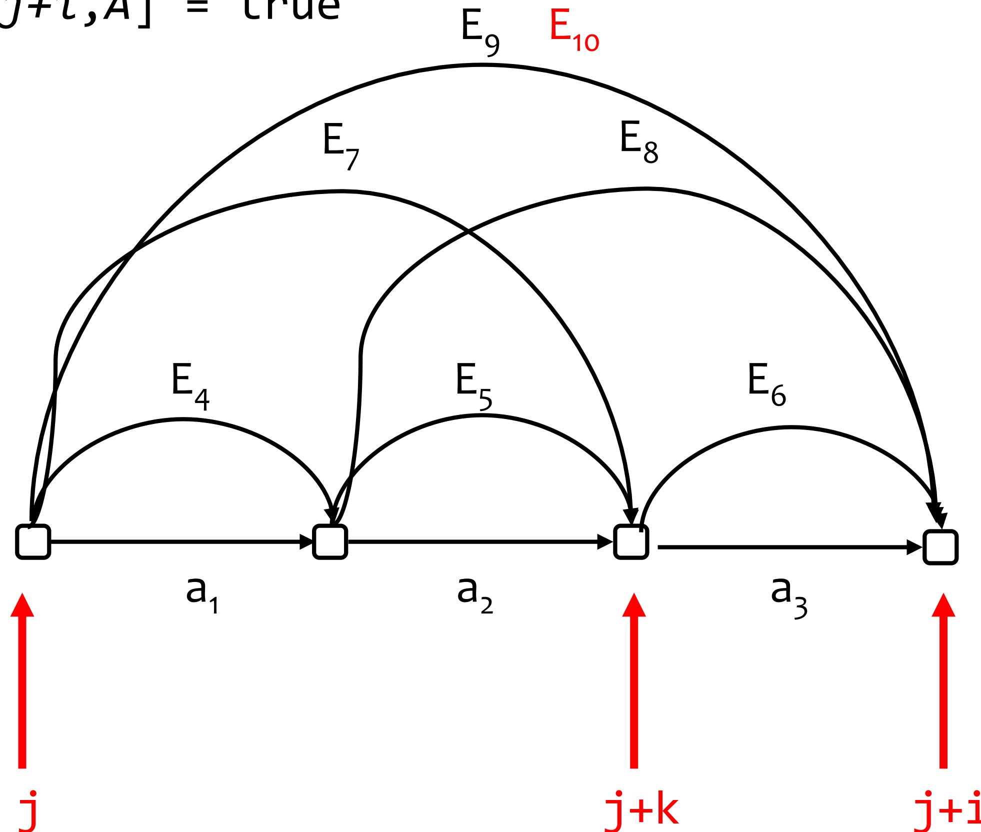
Illustration

```
for each i = 2 to n
  for each j = 0 to n-i
    for each k = 1 to i-1
      for each production rule R_A → R_B R_C
        if P[j,j+k,B] and P[j+k,j+i,C]
        then P[j,j+i,A] = true
```

var	value
i	3
j	0
k	2

True P entries
P(0,1,E)
P(1,2,E)
P(2,3,E)
P(0,2,E)
P(1,3,E)
P(0,3,E)
P(0,3,E)

E → a
| E E



Input: a a a

CYK Pseudocode

```
s = input string of n tokens: a_1 ... a_n.
r = # of non-terminal symbols in G: R_1 ... R_n

initialize all entries in P(N,N,r) = false
-- P(i,j,R_k) = R_k is used to parse input from i to j
for each i = 0 to n-1
  for each unit production R_k → a_i
    P[i,i+1,k] = true

for each i = 2 to n
  for each j = 0 to n-i
    for each k = 1 to i-1
      for each production rule R_A → R_B R_C
        if P[j,j+k,B] and P[j+k,j+i,C] then P[j,j+i,A] = true

if any of P[0,n-1,x] is true then s is in L(G)
else s is not in L(G)
```

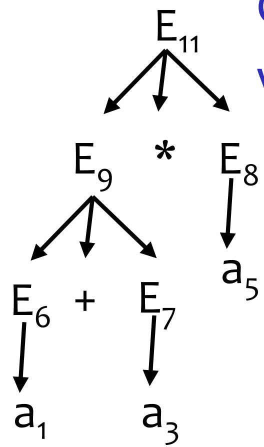
$O(N^2)$ space complexity

$O(N^3)$ time complexity

$\Theta(N^3 \cdot r)$ time complexity

Given a CYK graph, we can find a parse tree

Parse tree:



Q: Is this the tree we want?

Why is this not part of the tree?

E_9

E_{10}

E_6

E_7

E_8

a_1

$+_2$

a_3

$*_4$

a_5

E_{11}

Input: a + a * a

Ambiguous Grammars

From last lecture: One parse tree only!

The role of the grammar

- distinguish between syntactically legal and illegal programs

But that's not enough: it must also define a parse tree

- the parse tree conveys the meaning of the program
- associativity: left or right
- precedence: * before +

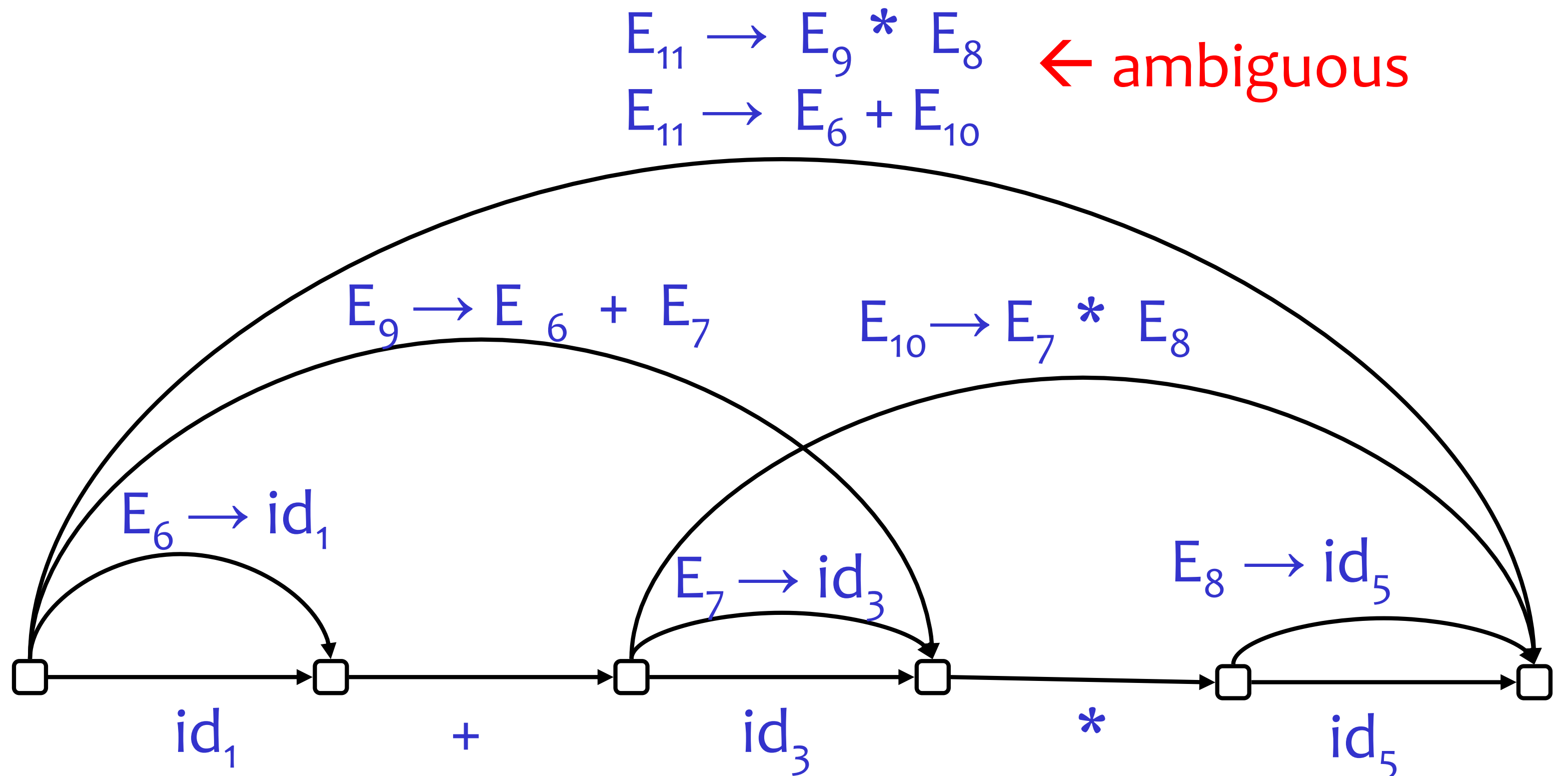
What if a string is parseable with multiple parse trees?

- we say the grammar is ambiguous
- must fix the grammar (the problem is not in the parser)

How many parse trees are in this CYK graph?

grammar: $E \rightarrow id \mid E + E \mid E * E$

input: id+id*id



We can see the two trees better here

$E_{11} \rightarrow E_9 * E_8$

$E_9 \rightarrow E_6 + E_7$

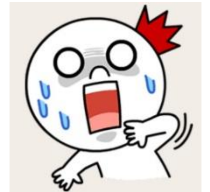
$E_6 \rightarrow id_1$

$E_7 \rightarrow id_3$

$E_8 \rightarrow id_5$

id_1 $+$ id_3 $*$ id_5

$E_{10} \rightarrow E_7 * E_8$



$E_{11} \rightarrow E_6 + E_{10}$

Dealing with Ambiguity

No general (automatic) way to handle ambiguity

Impossible to convert automatically an ambiguous grammar to an unambiguous one (we must state which tree desired)

Used with care, ambiguity can simplify the grammar

- Sometimes allows more natural definitions
- We need disambiguation mechanisms

There are two ways to remove ambiguity:

1) Rewrite the grammar (from last lecture)

   a general approach, but manual rewrite needed

   example: rewrite E::= E+E|E*E into E::=E+T|E, T::=T*F|T

2) Declare to the parser which productions to prefer

   works on most but not all ambiguities

Summary

G is <u>ambiguous</u>

iff there is <u>any</u> s in L(G) such that s has <u>multiple parse trees</u>

In expression grammars (such as E+E | E*E), ambiguity shows up as

- <u>precedence</u>: which of +,* binds stronger?
- <u>associativity</u>: is 1-2-3 parsed as (1-2)-3 or 1-(2-3)?

Test yourself

Work out the CYK graph for the input id+id*id+id

Notice there are multiple "ambiguous" edges

- ie, edges inserted by/due to distinct productions

How many edges are in the CYK graph?

- polynomial? N^2 or N^3? Exponential?

How many parse trees are represented by CYK graph?

- polynomial? N^2 or N^3? Exponential?

Disambiguation with precedence and associativity declarations

Precedence and Associativity Declarations

Instead of rewriting the grammar

- Use the more natural (ambiguous) grammar
- Along with disambiguating declarations

Bottom-up parsers like CYK and Earley allow declaration to disambiguate grammars

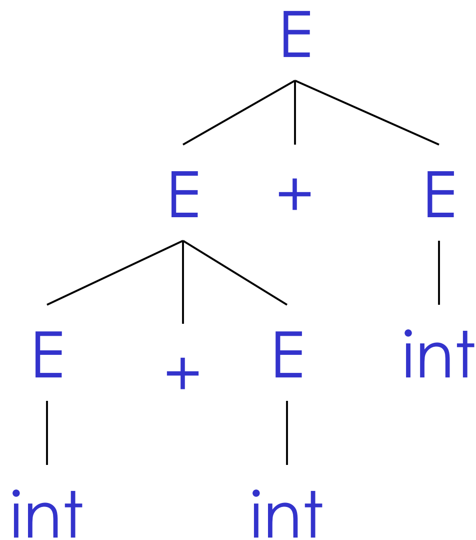
they are used in 401 parser

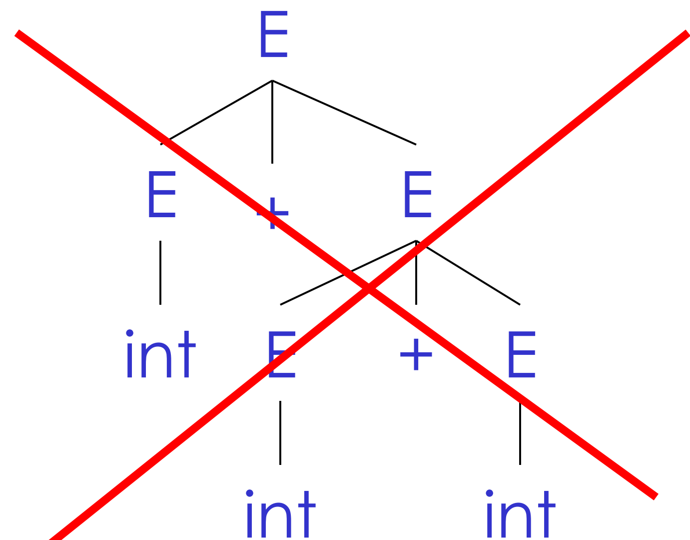
Examples …

Associativity Declarations

Consider the grammar $E \to E + E \mid int$

Ambiguous: two parse trees of int + int + int



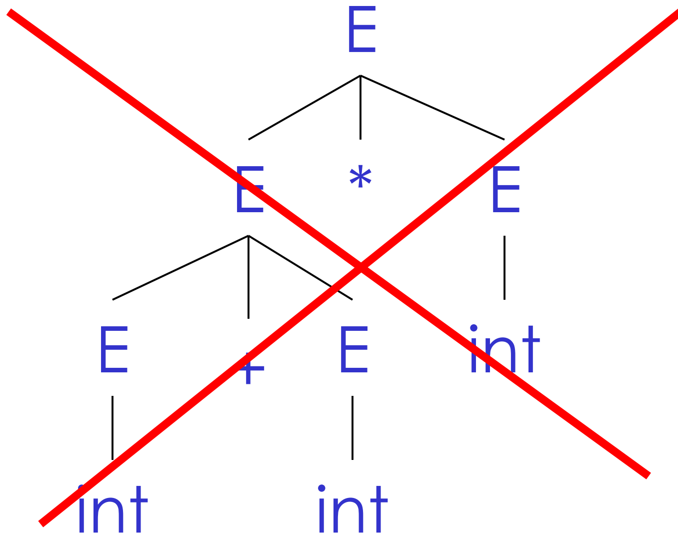


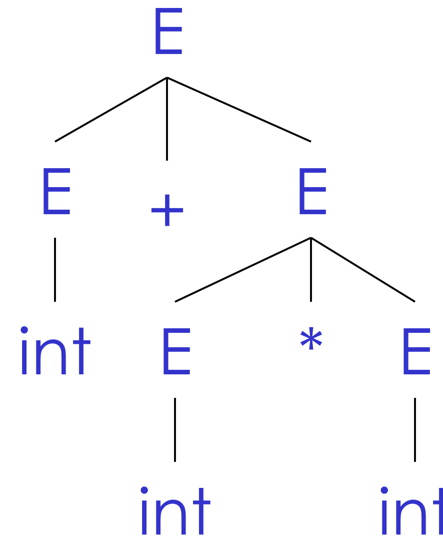
Left-associativity declaration: %left +

Precedence Declarations

Consider the grammar $E \rightarrow E + E \mid E * E \mid int$

- And the string int + int * int





Precedence declarations: %left +

%left *

Implementing disambiguation declarations

To disambiguate, we need to answer these questions:

Assume we reduced the input to E+E*E.

Now do we want parse tree (E+E)*E or E+(E*E)?

Similarly, given E+E+E,

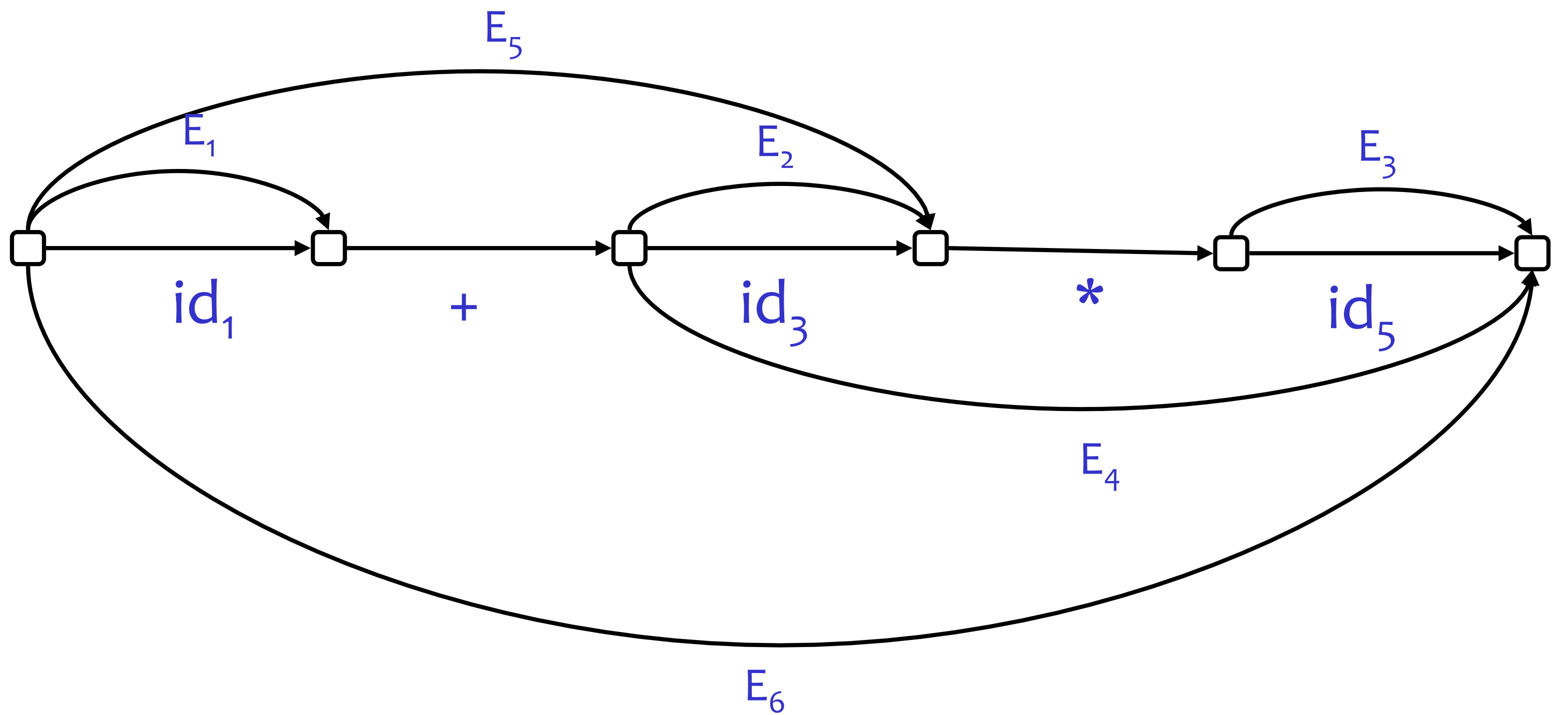
do we want parse tree (E+E)+E or E+(E+E)?

Example

Precedence:

%left +

%left *



Implementing the declarations in CYK/Earley

precedence declarations

- when multiple productions compete for being a child in the parse tree, select the one with *least* precedence

left associativity

- when multiple productions compete for being a child in the parse tree, select the one with *largest* left subtree

Where is ambiguity manifested in CYK?

```
s = input string of n tokens: a1 ... an.
r = # of non-terminal symbols in G: R1 ... Rn

initialize all entries in P(N,N,r) = false
-- P(i,j,Rk) = Rk is used to parse input from i to j
for each i = 0 to n-1
  for each unit production Rk → ai
    P[i,i+1,k] = true

for each i = 2 to n -- foreach rule that covers i tokens
  for each j = 0 to n-i -- start checking at pos j
    for each k = 1 to i-1 -- foreach partitioning of j to j+i
      for each production rule RA → RB RC
        if P[j,j+k,B] and P[j+k,j+i,C] then P[j,j+i,A] = true

if any of P[0,n-1,x] is true then s is in L(G)
else s is not in L(G)
```

%dprec: precedence declaration

Another disambiguating declaration (see bison)

```
E →  if E then E                %dprec 2 // chosen over 1
   | if E then E  else  E       %dprec 1
   | OTHER
```

Without %dprec, we'd have to rewrite the grammar:

```
E  →    MIF             -- all then are matched
     |  UIF             -- some then are unmatched
MIF → if E then MIF else MIF
     |   OTHER
UIF → if E then E
     |   if E then MIF else UIF
```

Ambiguity: The Dangling Else

Consider the ambiguous grammar

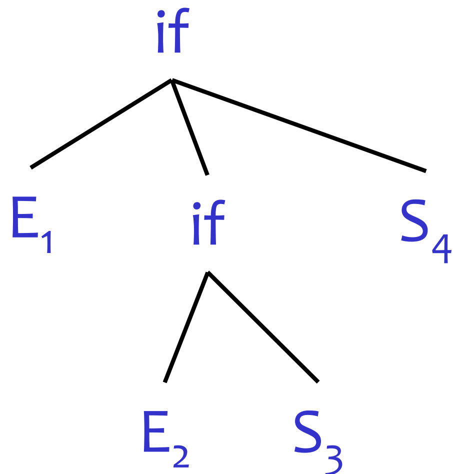
$$
\begin{aligned}
S \rightarrow\ & \text{if } E \text{ then } S \\
\mid\ & \text{if } E \text{ then } S \text{ else } S \\
\mid\ & \text{OTHER}
\end{aligned}
$$

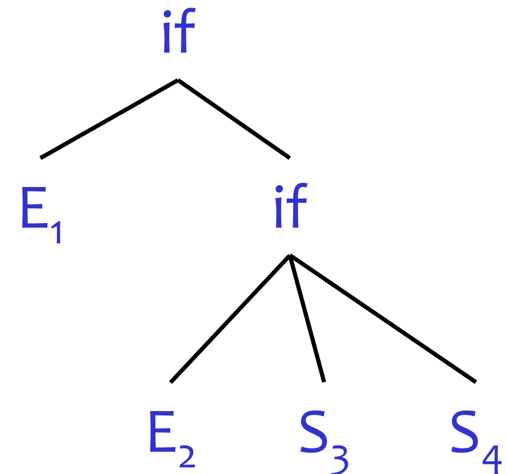
The Dangling Else: Example

- The expression

  if E_1 then if E_2 then S_3 else S_4

  has two parse trees





Typically we want the second form

The Dangling Else: A Fix

Usual rule: else matches the closest unmatched then

We *can* describe this in the grammar

Idea:

- distinguish matched and unmatched then's
- force matched then's into lower part of the tree

Rewritten if-then-else grammar

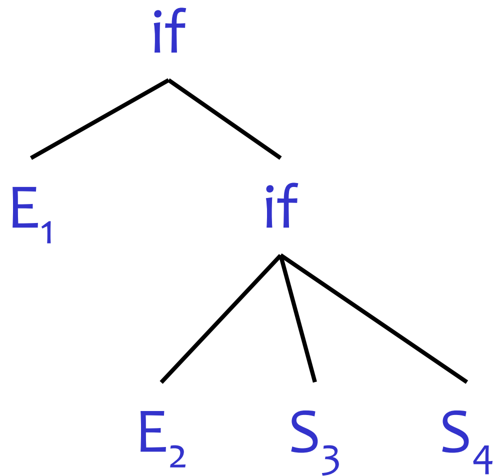
New grammar. Describes the same set of strings

- forces all matched ifs (if-then-else) as low in the tree as possible
- notice that MIF does not refer to UIF,
- so all unmatched ifs (if-then) will be high in the tree

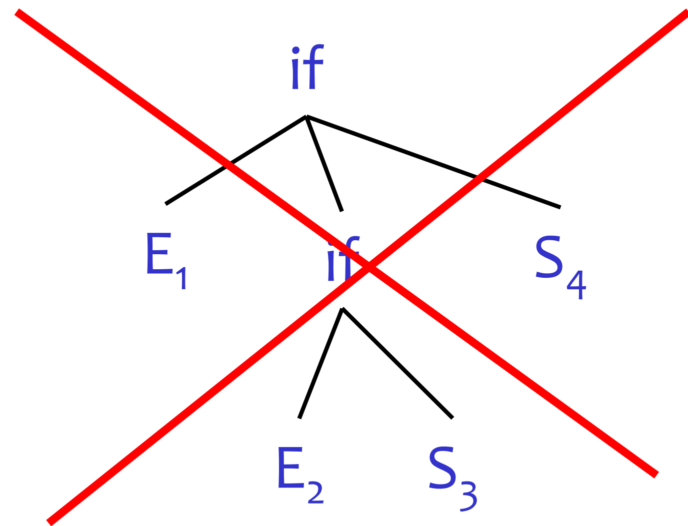
```
S → MIF                 /* all then are matched */
  | UIF                 /* some then are unmatched */

MIF → if E then MIF else MIF
    | OTHER
UIF → if E then S
    | if E then MIF else UIF
```

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then S_3 else S_4



- A valid parse tree (for a UIF)



- Not valid because the then expression is not a MIF

Earley Parser

Inefficiency in CYK

CYK may build useless parse subtrees

- useless = not part of the (final) parse tree
- true even for non-ambiguous grammars

Example

grammar: $E \rightarrow E{+}id \mid id$

input: `id+id+id`

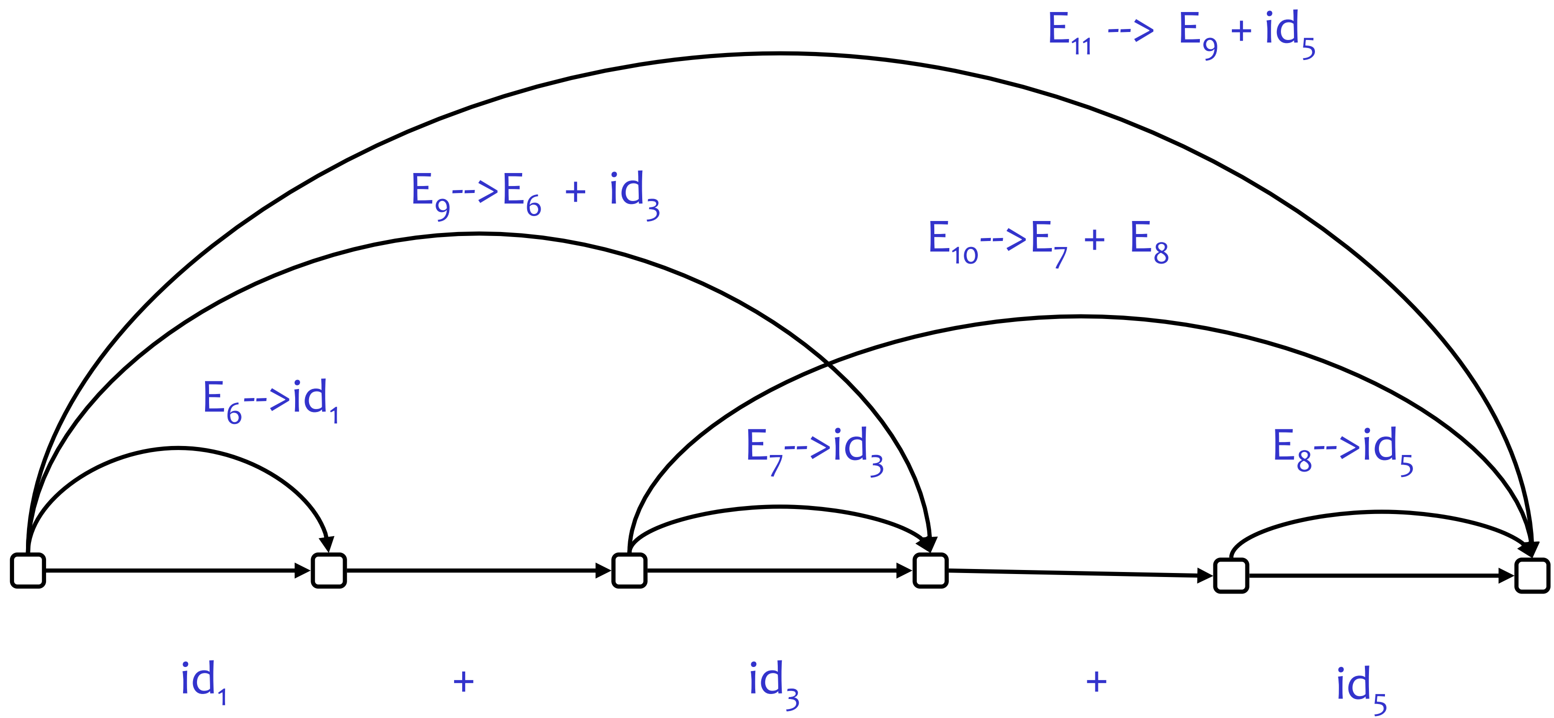
Can you spot the inefficiency? (see next slide)

It is the cause of the difference between $O(n^3)$ and $O(n^2)$, or parsing 100 vs 1000 chars in the same amount of time!

Earley is $O(n^2)$ for unambiguous grammars. For ambiguous grammars it is still $O(n^3)$.

Example

grammar: $E \longrightarrow E + id \mid id$



three useless reductions (E_7, E_8, and E_{10})

Earley parser fixes (part of) the inefficiency

space complexity:

- Earley and CYK are $O(N^2)$

time complexity:

- unambiguous grammars: Earley is $O(N^2)$, CYK is $O(N^3)$
- plus the constant factor improvement due to the inefficiency

why learn about Earley?

- the idea of Earley is used by modern fast parsers, e.g., LALR

Key idea

Process the input left-to-right

- as opposed to arbitrarily, as in CYK

Reduce only productions that appear non-useless

- non-useless == have a chance to be in the parse tree

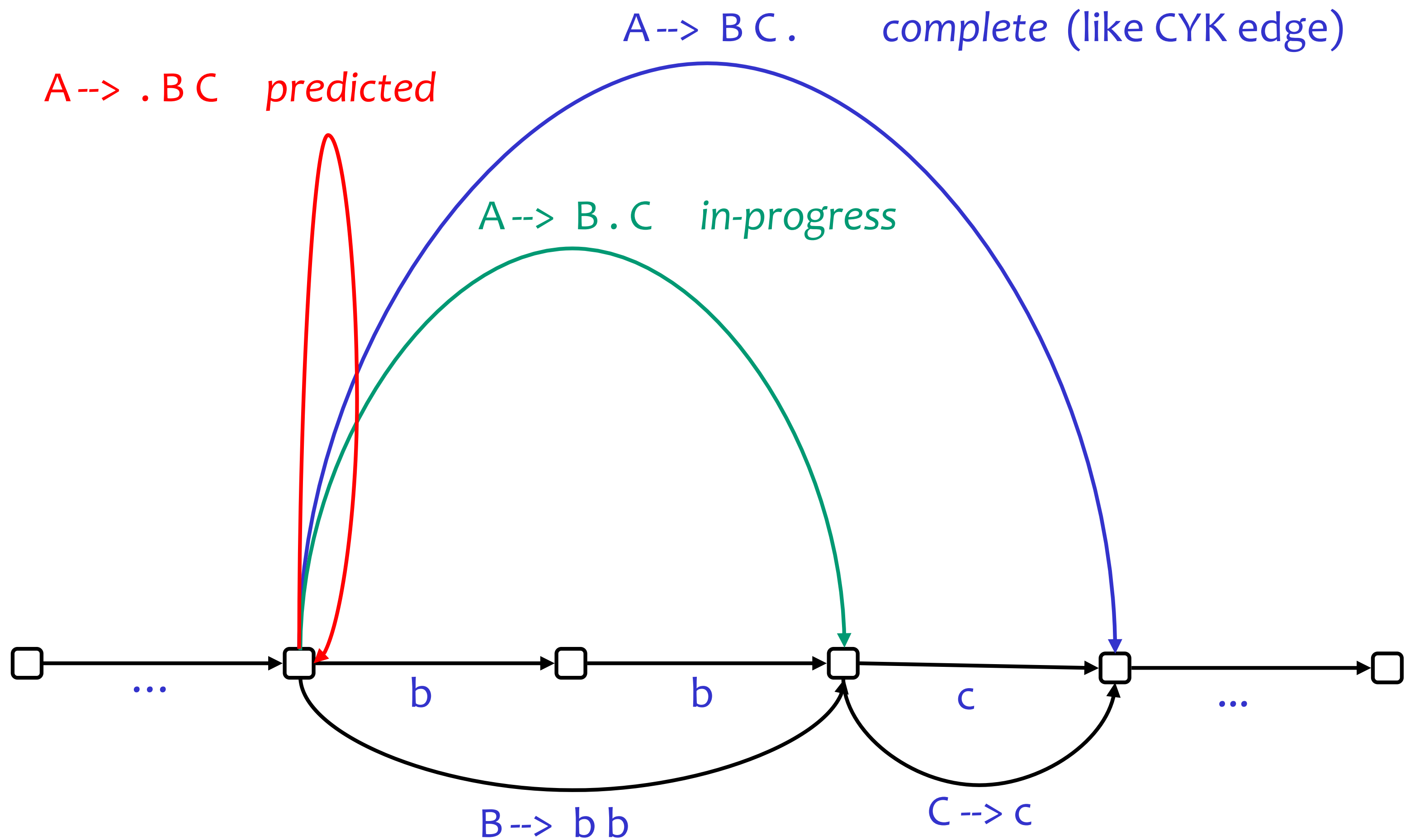
Key idea

- decide whether to reduce based on **the input seen so far**;
- after seeing more, we may still realize we built a useless tree

The algorithm

- Propagate the "context" of the parsing process.
- Context tells us what nonterminals can appear in the parse at the given point of input. Those that cannot won't be reduced.
- We are using the context as a filter!

Context given by three kinds of edges



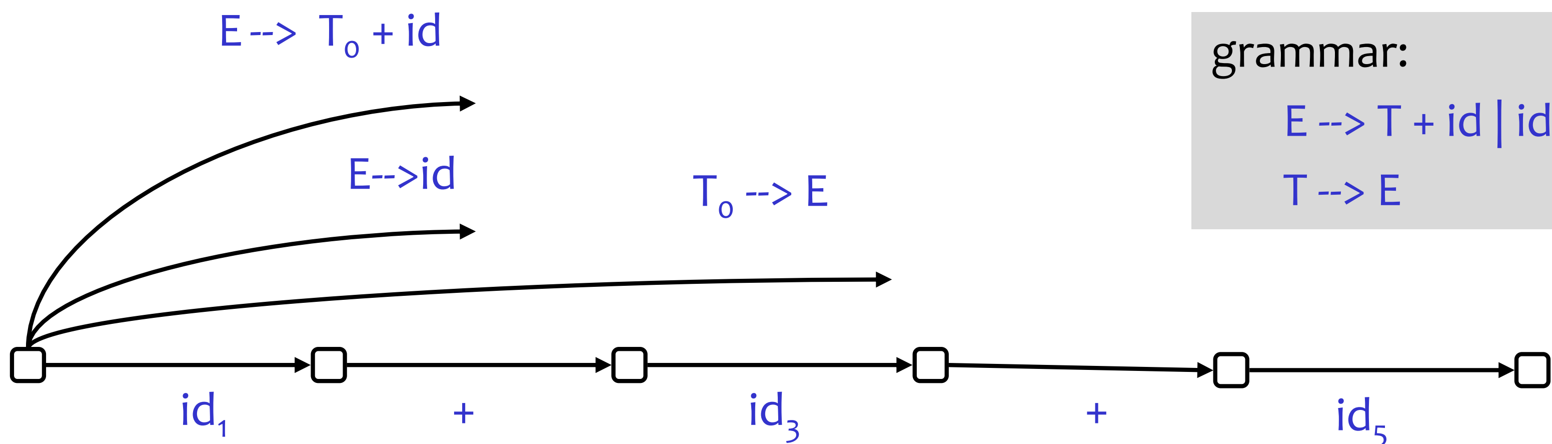
Edges track left-to-right progress: you can add A→BC. only if you previously added A→B.C

The intuition

Idea: We ask "What CYK edges can possibly start in node 0?"

1) those reducing to the start non-terminal
2) those that may produce non-terminals needed by (1)
3) those that may produce non-terminals needed by (2), etc

Convince yourself that these three edges can be in some parse tree!
Note that we are leaving the target nodes free.



61

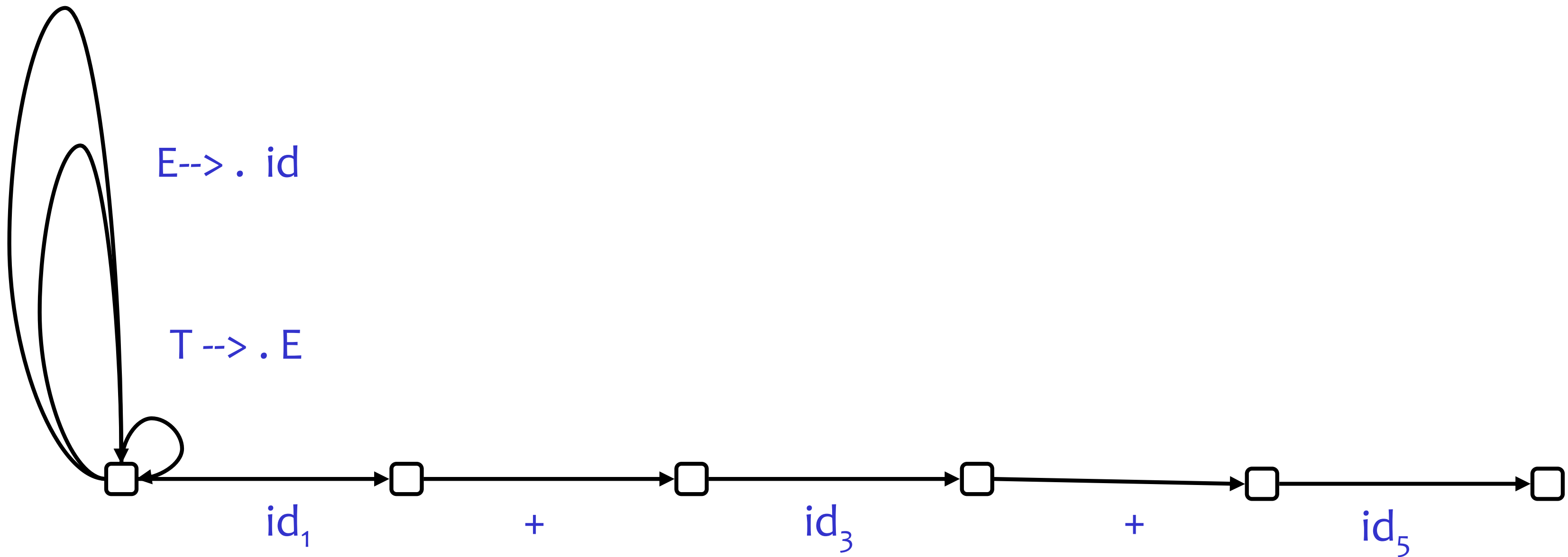
Example (1)

Initial predicted edges:

grammar:

E --> T + id | id

T --> E



Example (1.1)

Let's compress the visual representation:

these three edges → single edge with three labels

grammar:

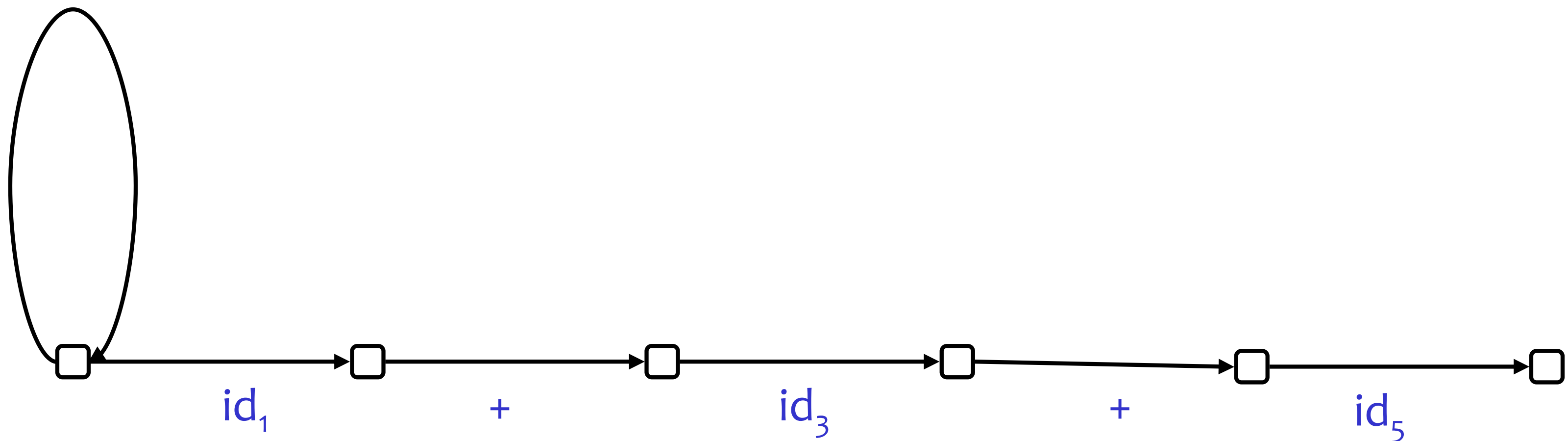
E --> T + id | id

T --> E

E --> . T + id

E--> . id

T --> . E



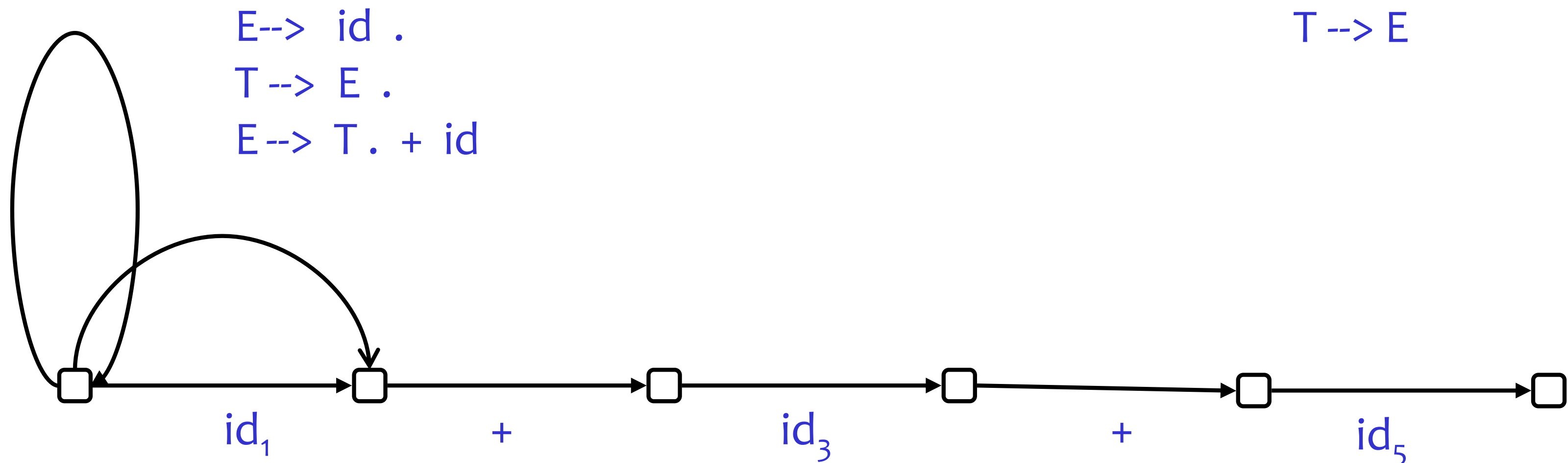
Example (2)

We advance the dot across id, which produces a complete edge, which leads to another complete edge, which in turn leads to an in-progress edge.

E --> . T + id
E--> . id
T --> . E

grammar:

E --> T + id | id
T --> E



Example (3)

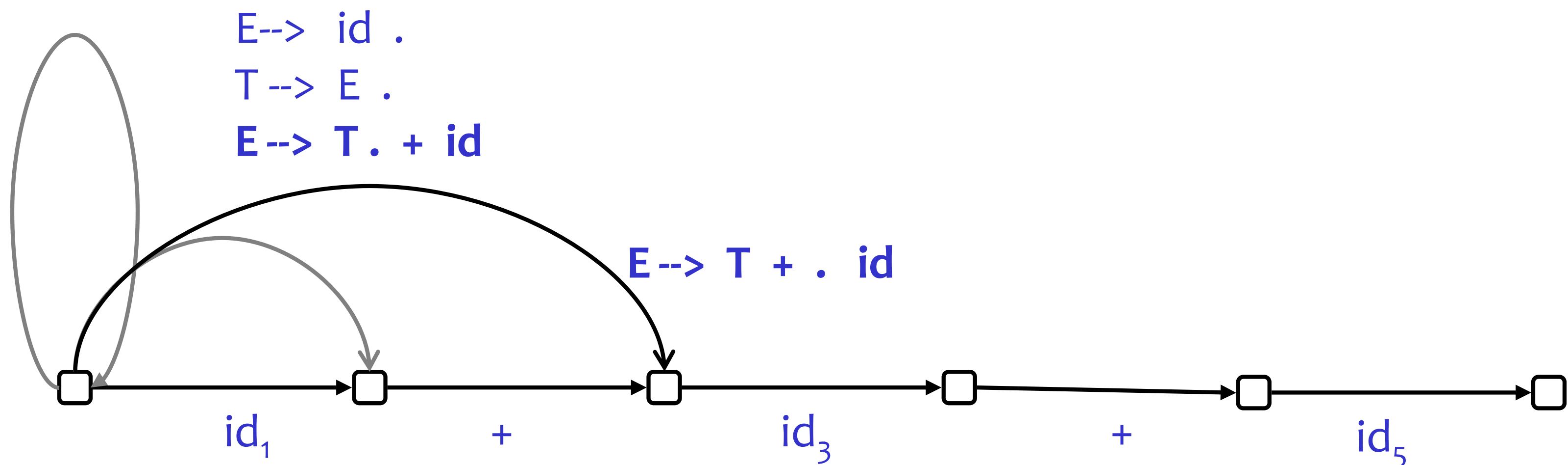
We advance the in-progress edge, the only edge we can add at this point.

E --> . T + id
E--> . id
T --> . E

grammar:

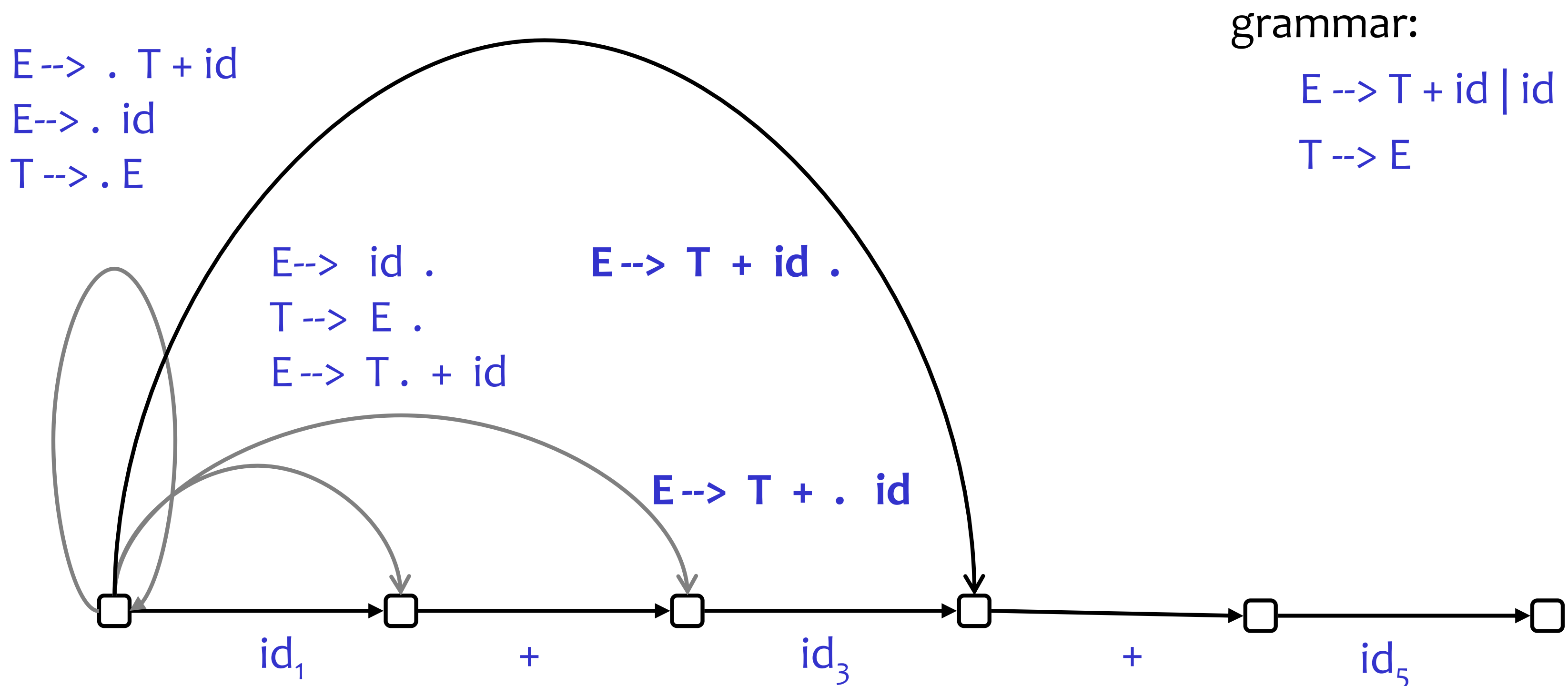
E --> T + id | id

T --> E



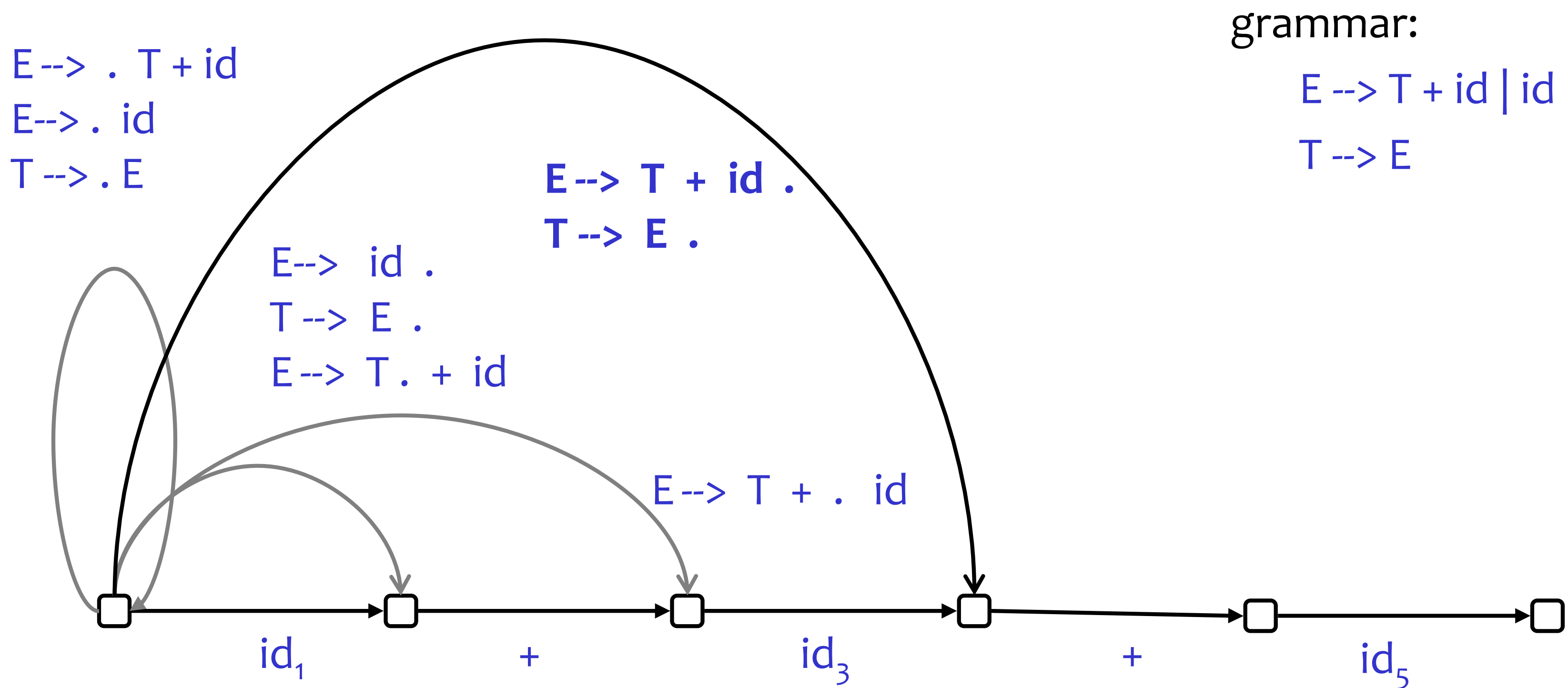
Example (4)

Again, we advance the in-progress edge. But now we created a complete edge.



Example (5)

The complete edge leads to reductions to another complete edge, exactly as in CYK.



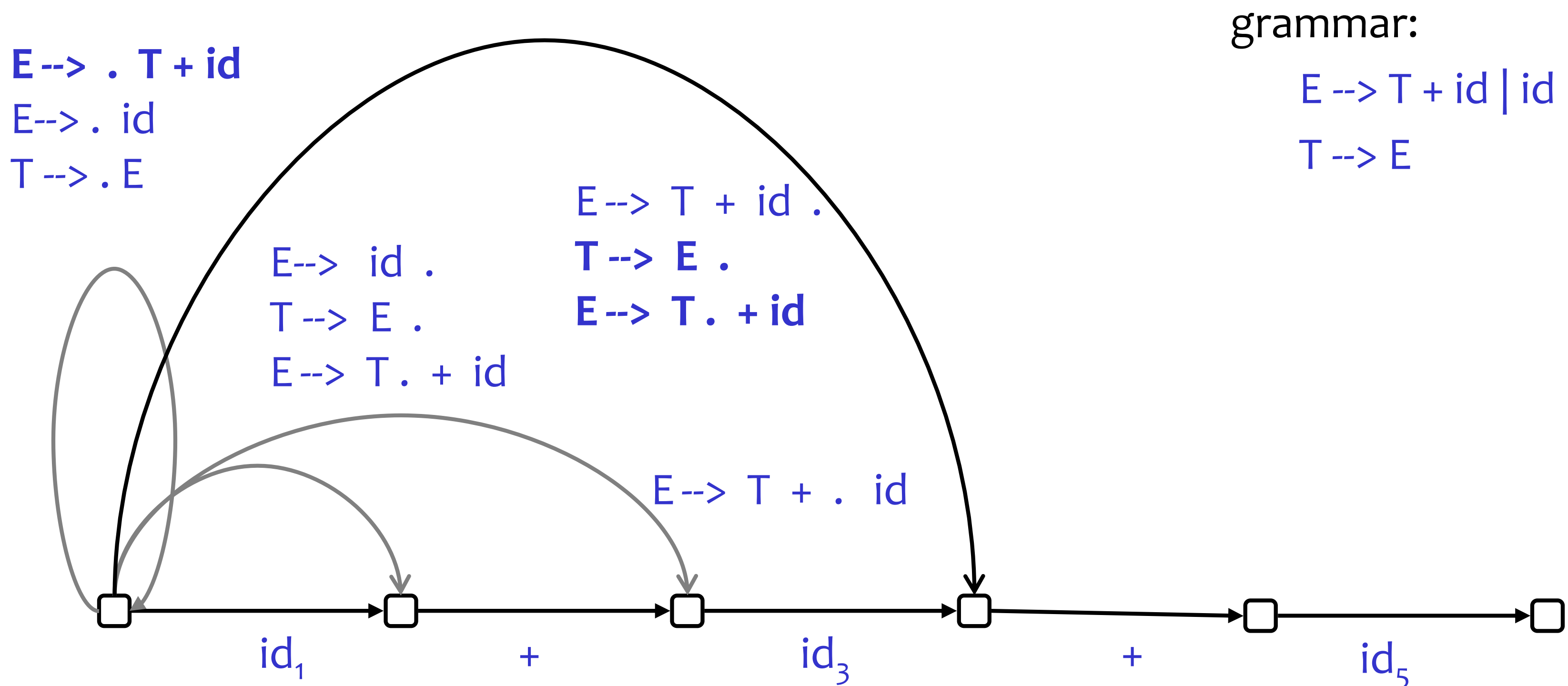
grammar:

E --> T + id | id

T --> E

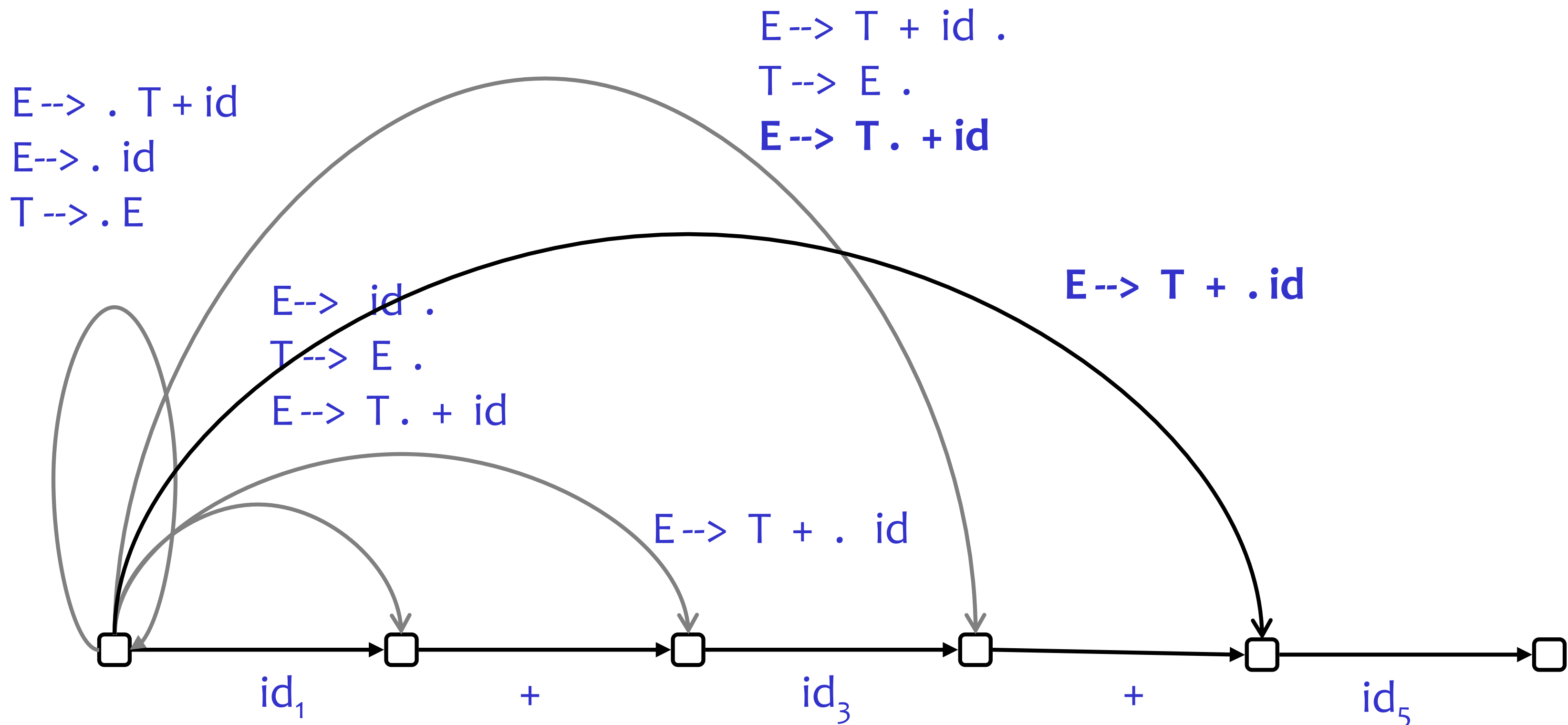
Example (6)

We also advance the predicted edge, creating a new in-progress edge.



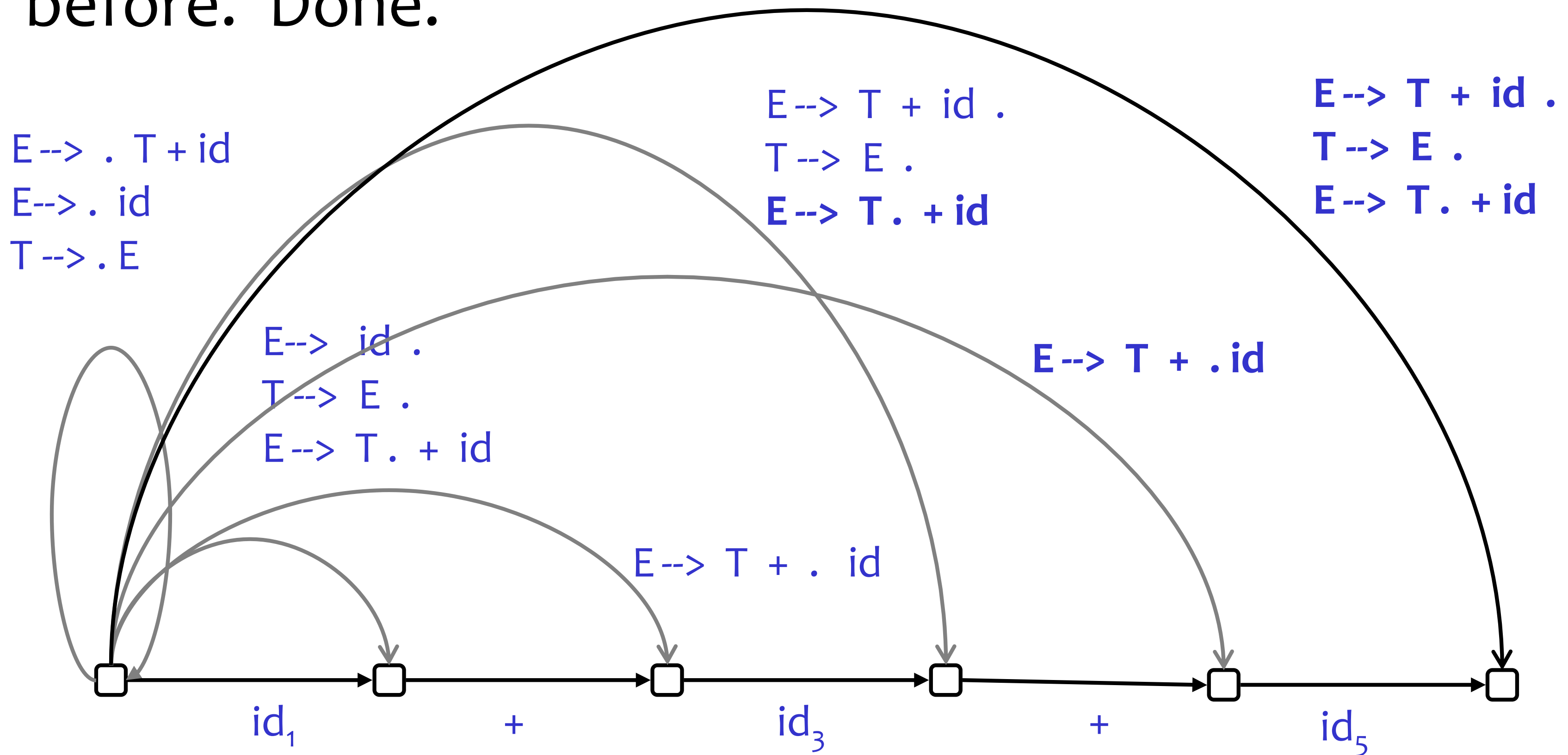
Example (7)

We advance the predicted edge across +, creating a new in-progress edge.



Example (8)

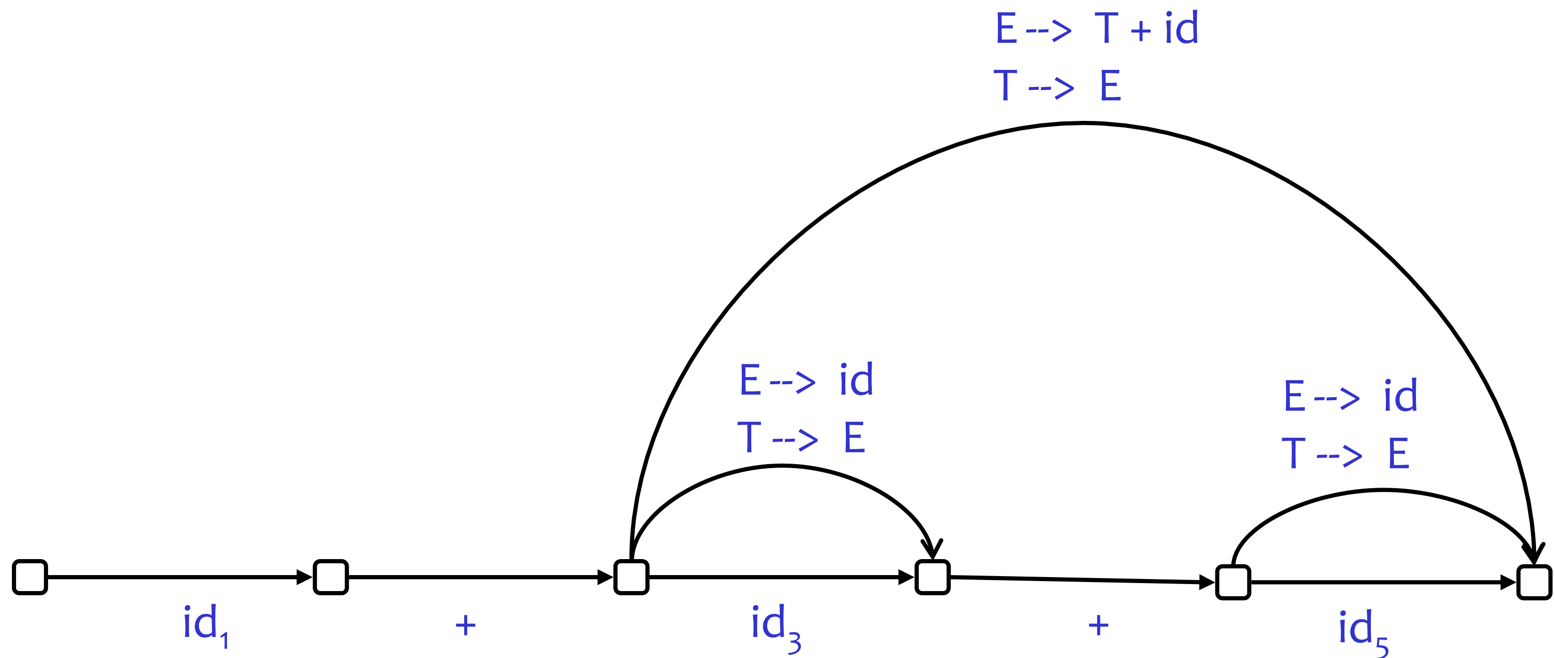
Advance again, creating a complete edge, which leads to more complete edges and an in-progress edge, as before. Done.



Example (a note)

Compare with CYK:

We avoided creating these six CYK edges.



In more detail: Three kinds of edges

Productions extended with a dot '.'

. indicates position of input (how much of the rule we saw).

Complete: A --> B C .

We found an input substring that reduces to A.

These are the reduction CYK edges.

Predicted: A --> . B C

We are looking for a substring in the input that reduces to A

But we have seen nothing of B C yet (the dot is before B C).

In-progress: A --> B . C

Like predicted but we've seen a substring that reduces to B.

Earley Algorithm

Three main functions that do all the work:

For all terminals in the input, left to right:

Scanner: advance the dot across the next input symbol

Repeat until no more edges can be added:

Predict: add predictions to the graph

Complete: move the dot to the right across a non-terminal when that non-terminal is found

Earley pseudocode in more detail

```
Add edge (0,0,(S -> . alpha)) to graph, for all S -> alpha

for all tokens inp[j] on the input, left to right

    ADVANCE across the next token:
    for each edge (i,j-1,N -> alpha . inp[j] beta)
       add edge (i,j,N -> alpha inp[j] . beta)

   Repeat COMPLETE and PREDICT until no more dges can be added

    COMPLETE productions

    for each edge (i,j,N -> alpha .)
      for each edge (k,i,M -> beta . N gamma)
         add edge (k,j,M -> beta N . gamma)

    PREDICT what the parser is to see on input (move dots in edges that are in progress)

    for each edge (i,j,N -> alpha . M beta)
       for each production M -> gamma
          add edge (j,j,M -> . gamma)
```

Prediction

Prediction (def):

determining which productions apply at current point of input

performed top-down through the grammar

by examining all possible derivation sequences

this will tell us

which non-terminals we can use in the tree starting at the current point of the string

we will do prediction not only at the beginning of parsing

but at each parsing step

Example: Given E --> . T + id we predict T --> . E

HW3

You are given a clean Earley parser in Python

It visualizes the parse.

Your goal is to rewrite the grammar

to make the parser run in linear time.