Lecture 9

Bottom-up parsers

Datalog, CYK parser, Earley parser

Ras Bodik
Ali and Mangpo

Hack Your Language!
CS164: Introduction to Programming Languages and Compilers, Spring 2013
UC Berkeley

Hidden slides

This slide deck contains hidden slides that may help in studying the material.

- These slides show up in the exported pdf file but when you view the ppt file in Slide Show mode.

Today

Datalog

- a special subset of Prolog

CYK parser

- builds the parse bottom up

Earley parser

- solves CYK's inefficiency

Prolog Parser

top-down parser: builds the parse tree
by descending from the root

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]). --> true
```

What answers does this query return?
parse([a,+,a],Out)?

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

same f? no, we can rename one of them.

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

executed by hand. may be incorrect.

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

Construct also the AST

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

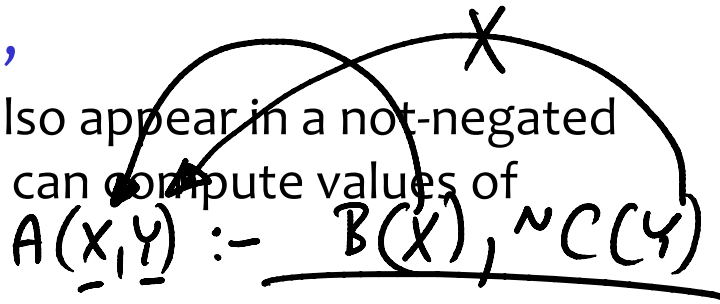
```
e(In,Out,T1)              :- t(In, Out, T1).
e(In,Out,plus(T1,T2)) :- t(In, [+|R], T1), e(R,Out,T2).
t(In,Out,T1)              :- f(In, Out, T1).
t(In,Out,times(T1,T2)):- f(In, [*|R], T1), t(R,Out,T2).
f([a|Out],Out, a).

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

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

Datalog
(a subset of Prolog, more or less)

Datalog: a well-behaved subset of Prolog

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,
  - 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
  - this can be satisfied by simply not using negation, is possible

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)

Why do we care about Datalog?

Predictable semantics:

- all Datalog programs terminate (unlike Prolog programs) – thanks to the restrictions above, which make the set of all possible proofs finite

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 can mechanically derive famous parsers

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

1) CYK parser == Datalog version of Prolog *rdp*
2) Earley == Magic Set transformation of CYK

There is a bigger cs164 lesson here:

- 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).

CYK parser

(can we run a parser in polynomial time?)

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)

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 substring input[i:j] can be derived from the non-terminal E.

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

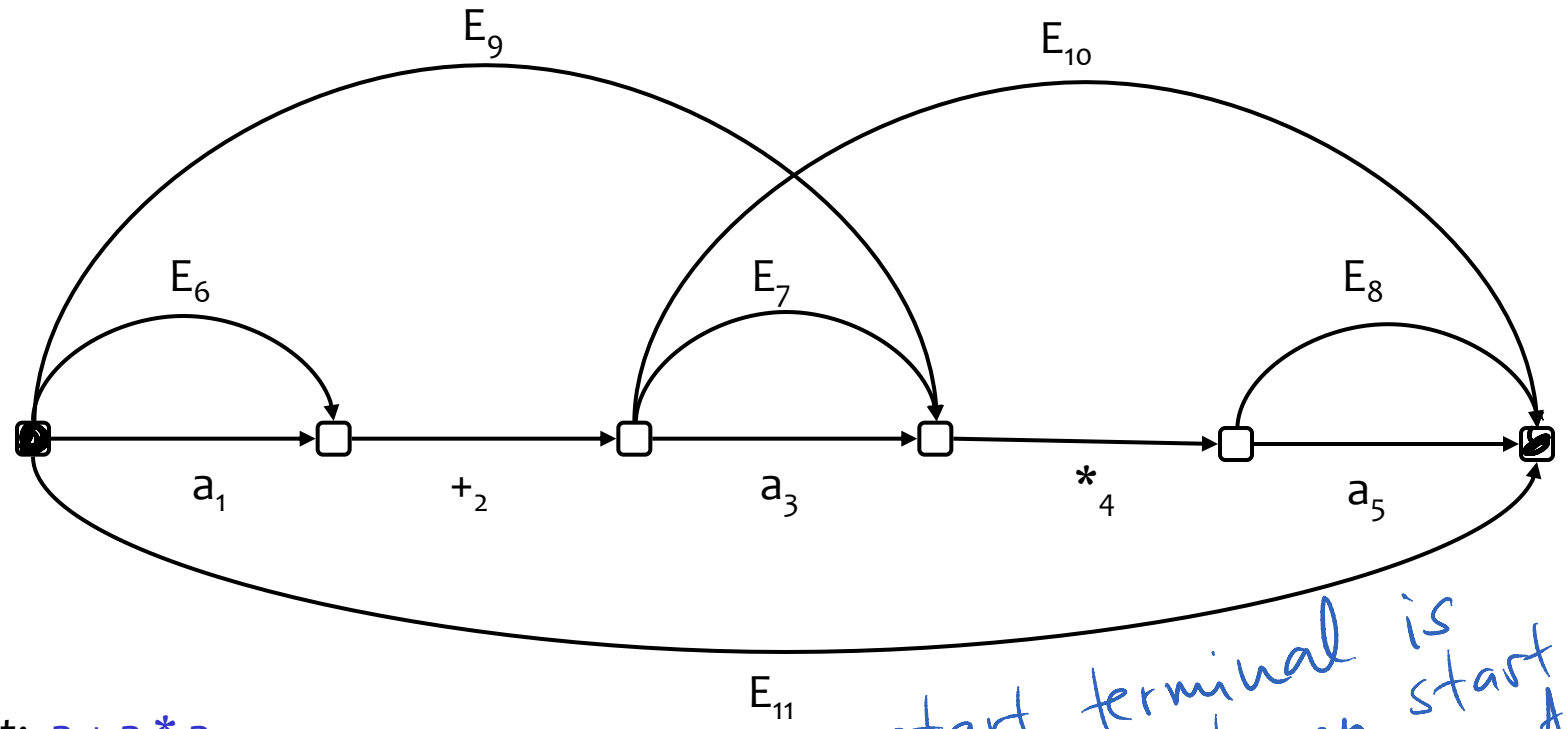
```
e(I,I+1) :- input[I]==‘a’.
e(I,J)   :- e(I,K), input[K]==‘+’, e(K+1,J).
e(I,J)   :- e(I,K), input[K]==‘*’, e(K+1,J).
```

A graphical way to visualize the 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)

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)

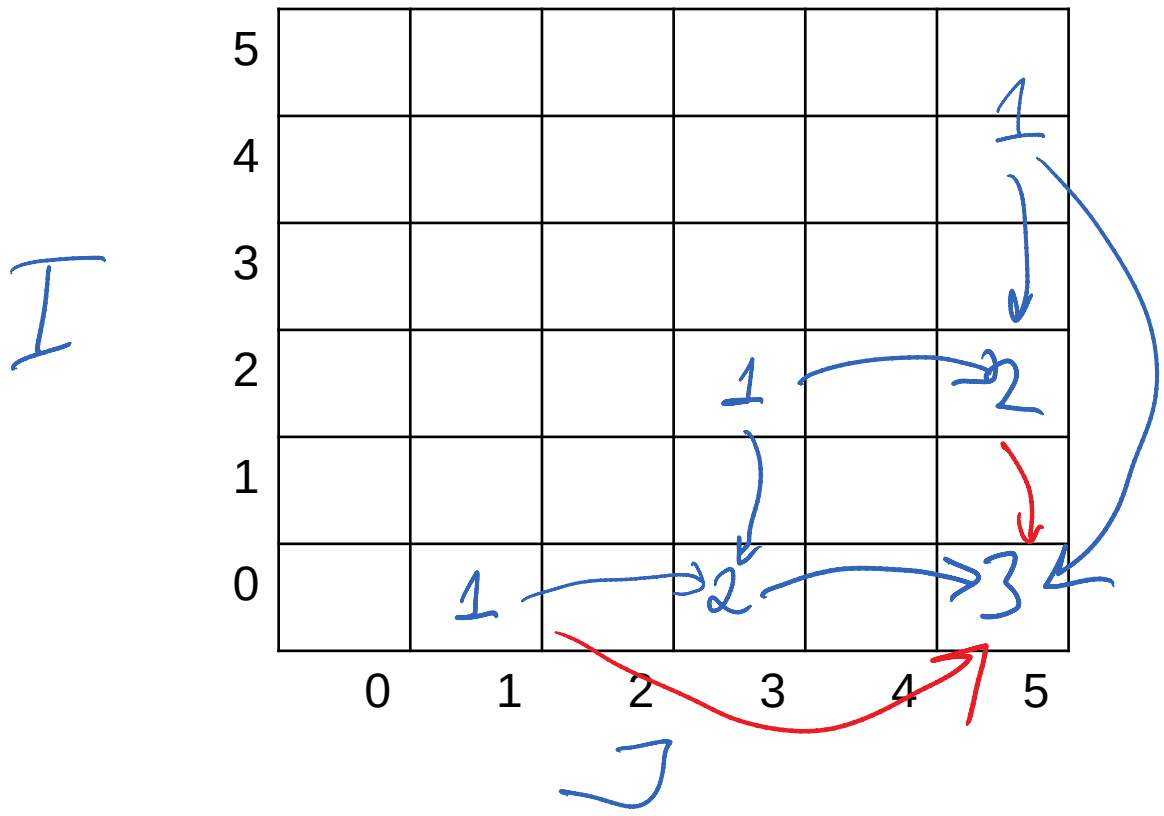
e(0,3) = true // using rule #2

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



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

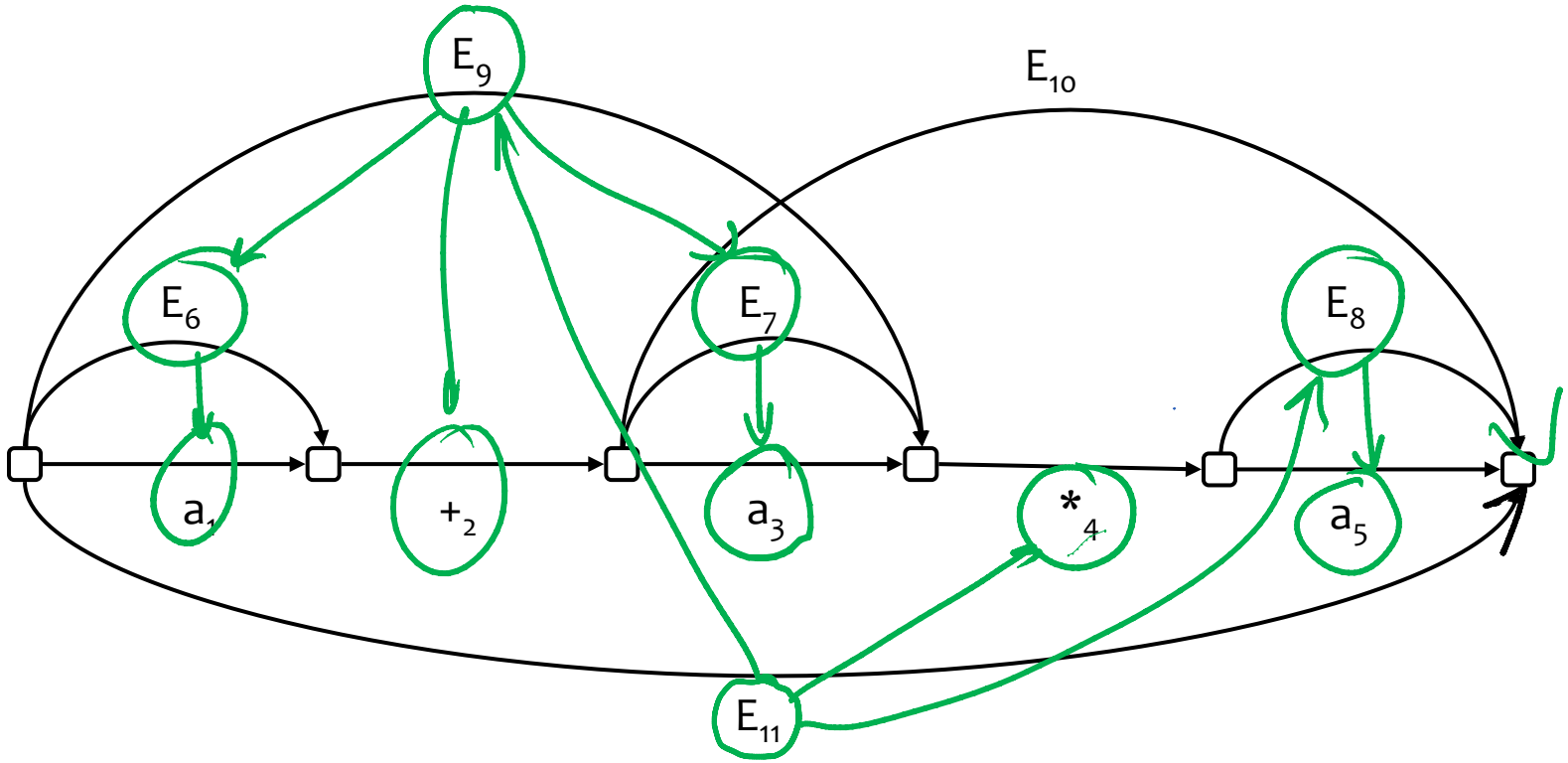
- edge *e*=(0,N,S) corresponds to the root of parse tree *r*
- edges that caused insertion of *e* are children of *r*

Helps to label edges with entire productions

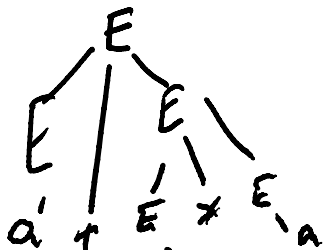
- not just the LHS symbol of the production
- make symbols unique with subscripts
- such labels make the parse tree explicit

A graphical way to visualize this evaluation

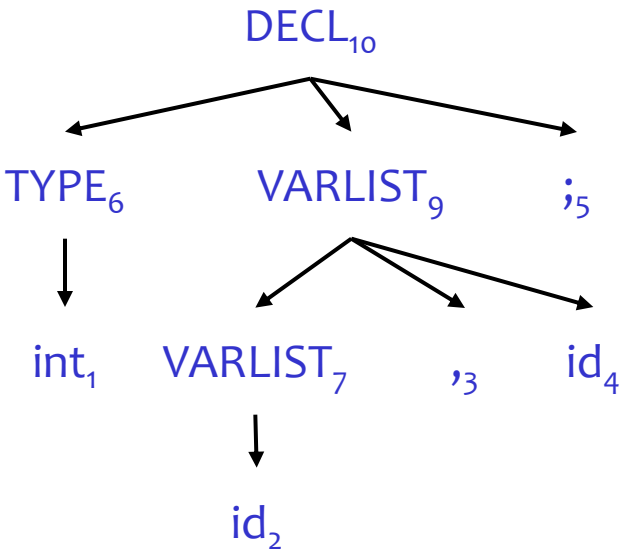
Parse tree:



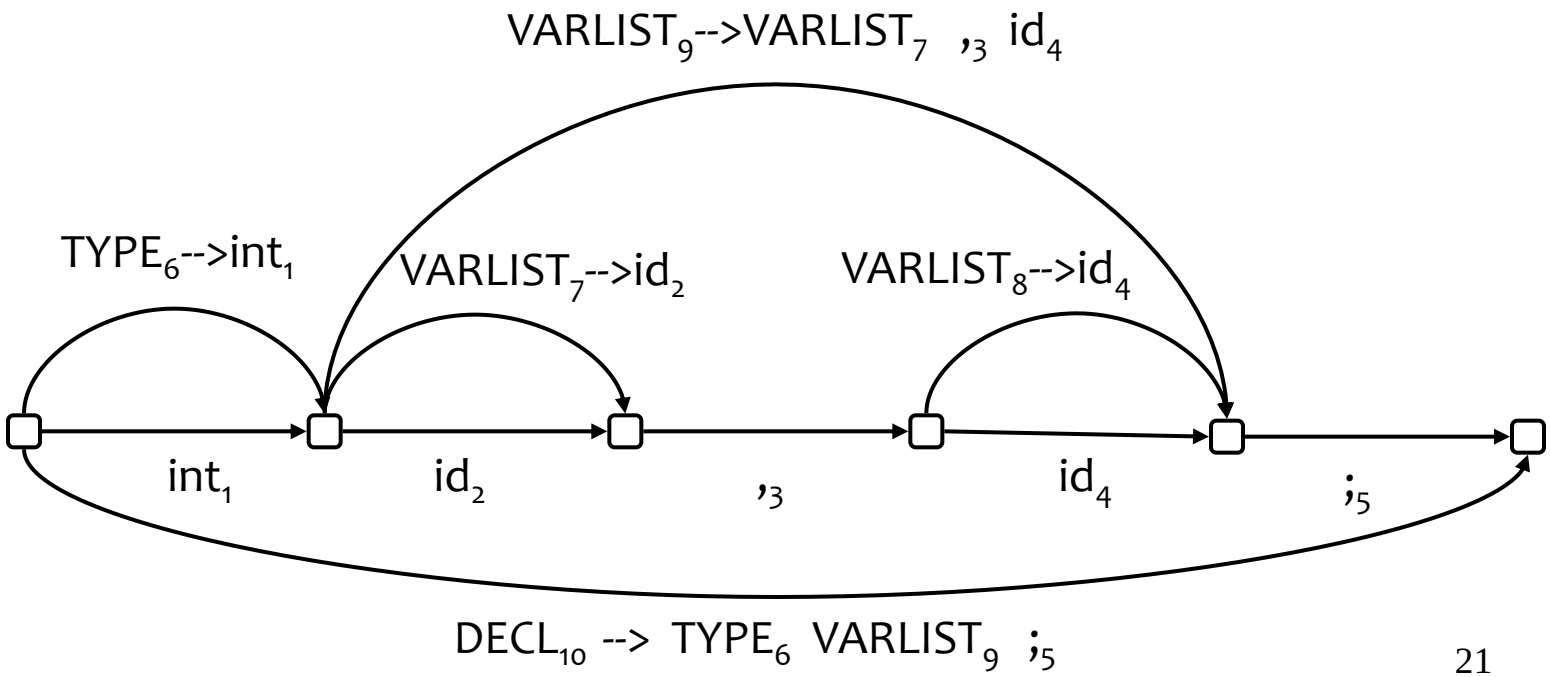
Input: a + a * a



Example of CYK execution



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



Grammars, derivations, parse trees

Example grammar

DECL --> TYPE VARLIST ;

TYPE --> int | float

VARLIST --> id | VARLIST , id

Example string

int id , id ;

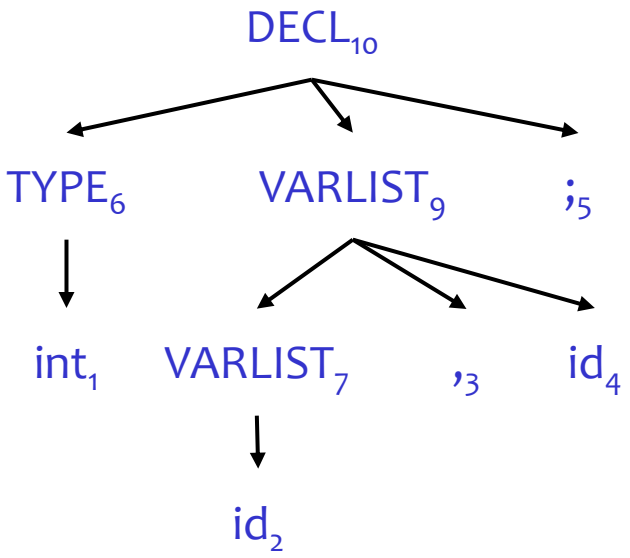
Derivation of the string

DECL --> TYPE VARLIST ;

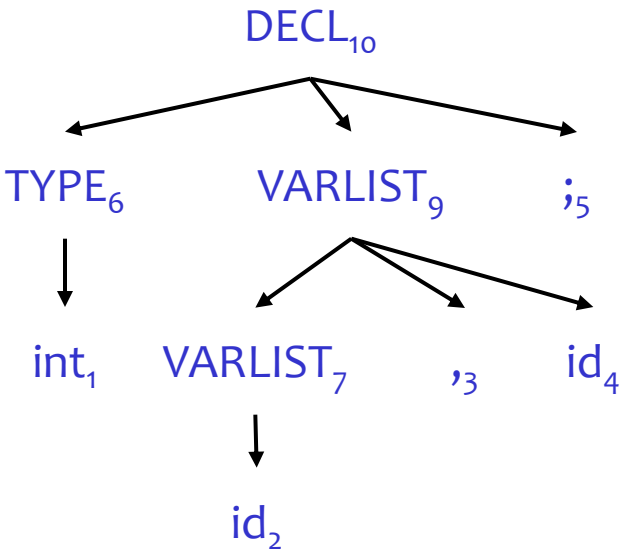
--> int VARLIST ;

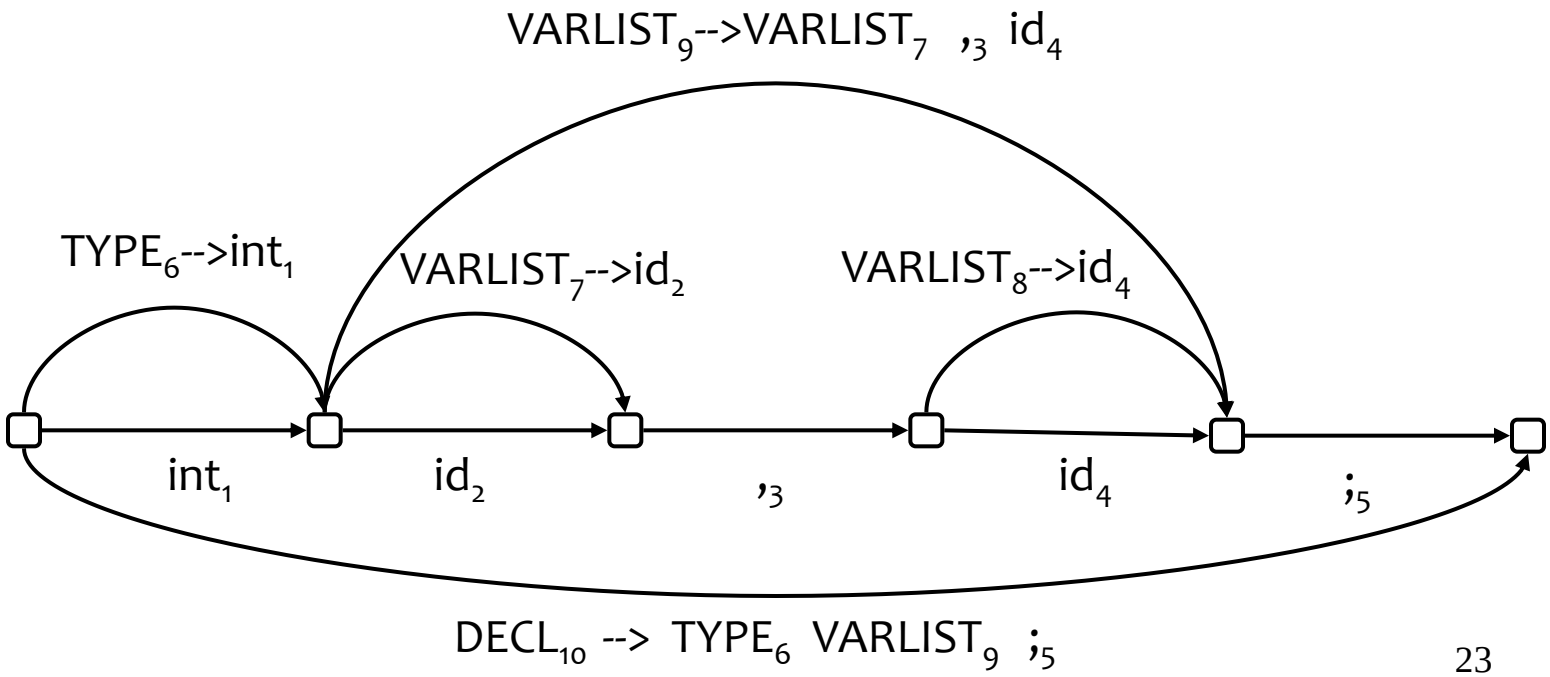
--> … -->

--> int id , id ;



CYK execution





Key invariant

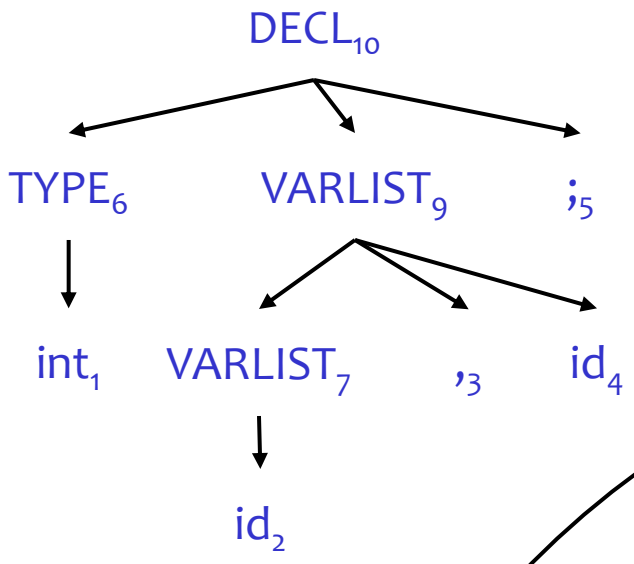
Edge (i,j,T) exists iff T -->* input[i:j]

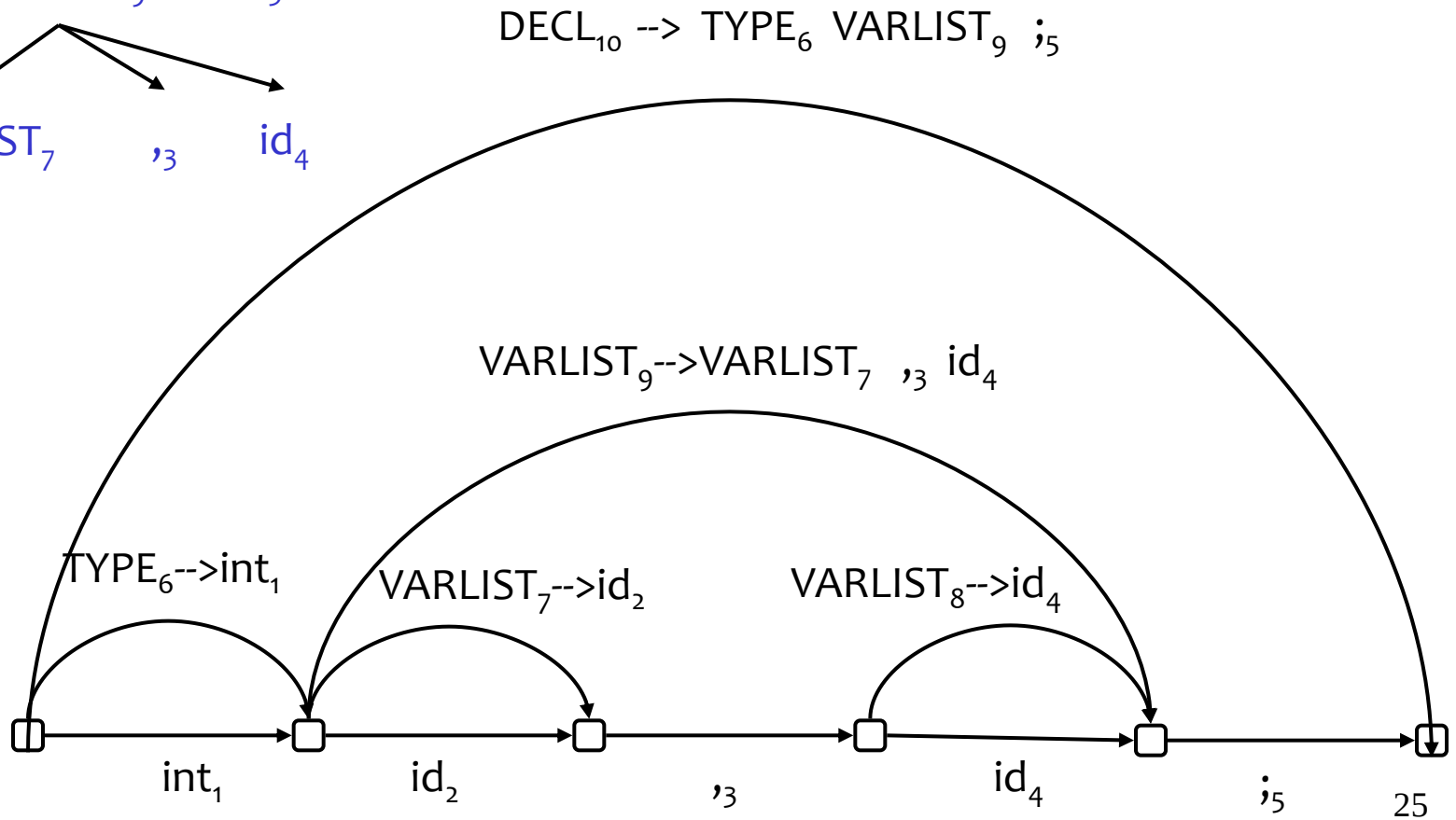
- **T -->* input[i:j]** means that the i:j slice of input can be derived from T in zero or more steps
- T can be either terminal or non-terminal

Corollary:

- input is from L(G) iff the algorithm creates the edge (0,N,S)
- N is input length

Constructing the parse tree from the CYK graph





CYK graph to parse tree

Parse tree nodes

obtained from CYK edges are grammar productions

Parse tree edges

obtained from reductions (ie which rhs produced the lhs)

CYK Parser

Builds the parse bottom-up

given grammar containing $A \rightarrow B\ C$, when you find adjacent B C in the CYK graph, reduce B C to 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: production forms allowed:

- $A \rightarrow BC$ or
- $A \rightarrow d$ or
- $S \rightarrow \varepsilon$ (only start non-terminal can derive ε)

Each grammar can be rewritten to this form

CYK: dynamic programming

Systematically fill in the graph with solutions to subproblems

- what are these subproblems?

When complete:

- the graph contains all possible solutions to all of the subproblems needed to solve the whole problem

Solves reparsing inefficiencies

- because subtrees are not reparsed but looked up

Complexity, implementation tricks

Time complexity: $O(N^3)$, Space complexity: $O(N^2)$

- convince yourself this is the case
- hint: consider the grammar to be constant size?

Implementation:

- the graph implementation may be too slow
- instead, store solutions to subproblems in a 2D array
  - solutions[i,j] stores a list of labels of all edges from i to j

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 ::= E+id | id

input: id+id+id

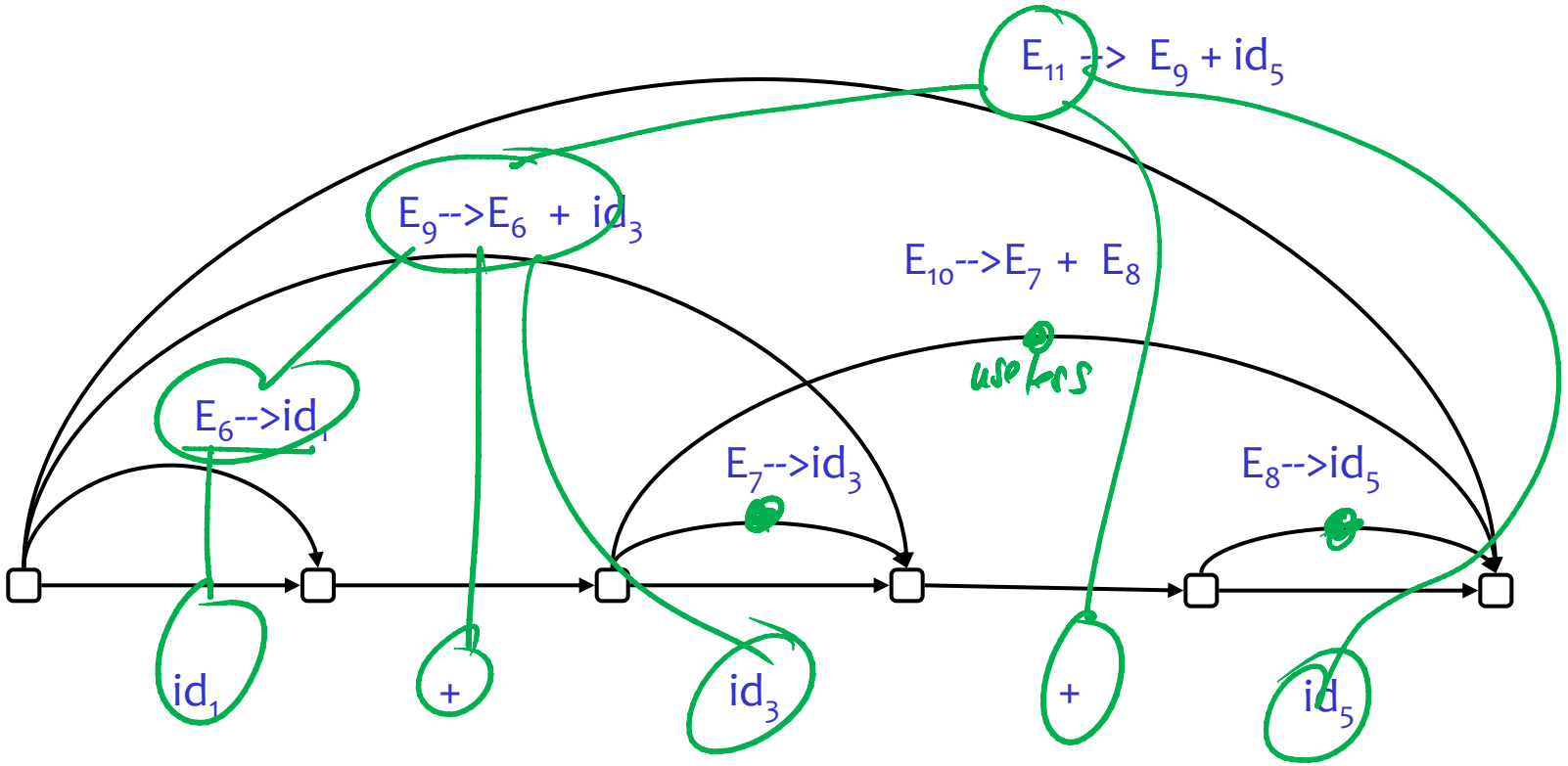
Can you spot the inefficiency?

This inefficiency is a difference between $O(n^3)$ and $O(n^2)$

It's parsing 100 vs 1000 characters in the same time!

Example

grammar: E⟶E+id | id



three useless reductions are done (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?

- idea of Earley states is used by the faster parsers, like LALR
- so you learn the key idea from those modern parsers
- You will implement it in PA4
- In HW4 (required), you will optimize an inefficient version of Earley

Key idea

Process the input left-to-right

- as opposed to arbitrarily, as in CYK

Reduce only productions that appear non-useless

- consider only reductions with a chance to be in the parse tree

Key idea

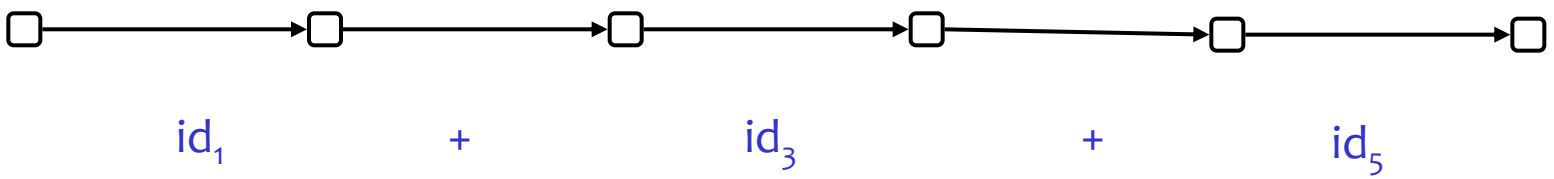
- decide whether to reduce based on **the input seen so far**
- after seeing more, we may still realize we built a useless tree

The algorithm

- Propagate a "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.

Key idea: suppress useless reductions

grammar: E⟶E+id | id

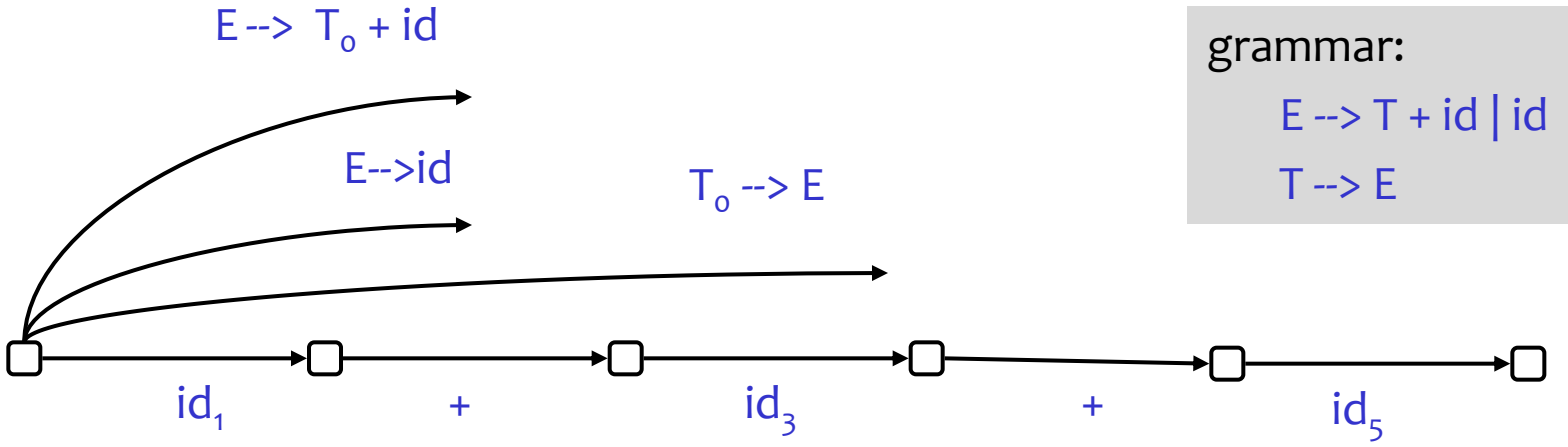


The intuition

Use CYK edges (aka reductions), plus more edges.

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



grammar:

E --> T + id | id

T --> E

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

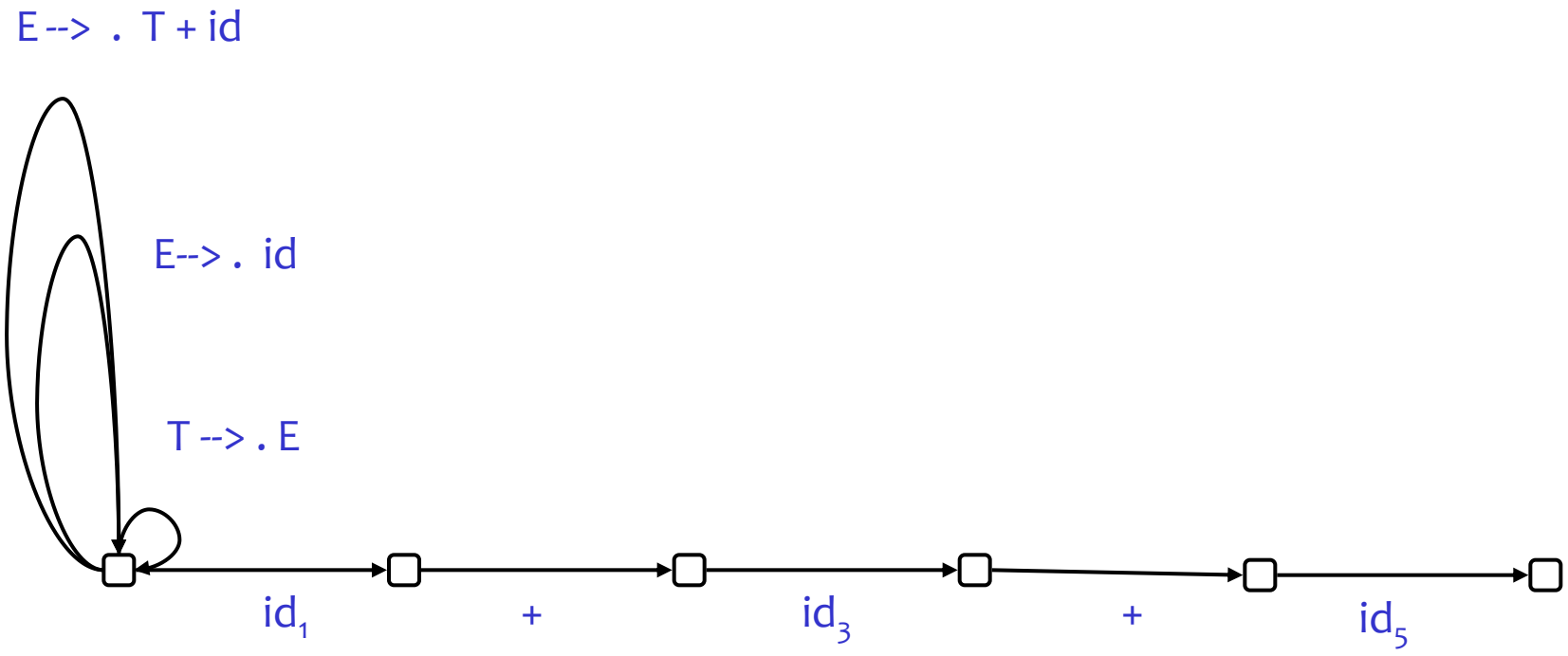
Example (1)

Initial predicted edges:

grammar:

E --> T + id | id

T --> E



Example (1.1)

Let's compress the visual representation:

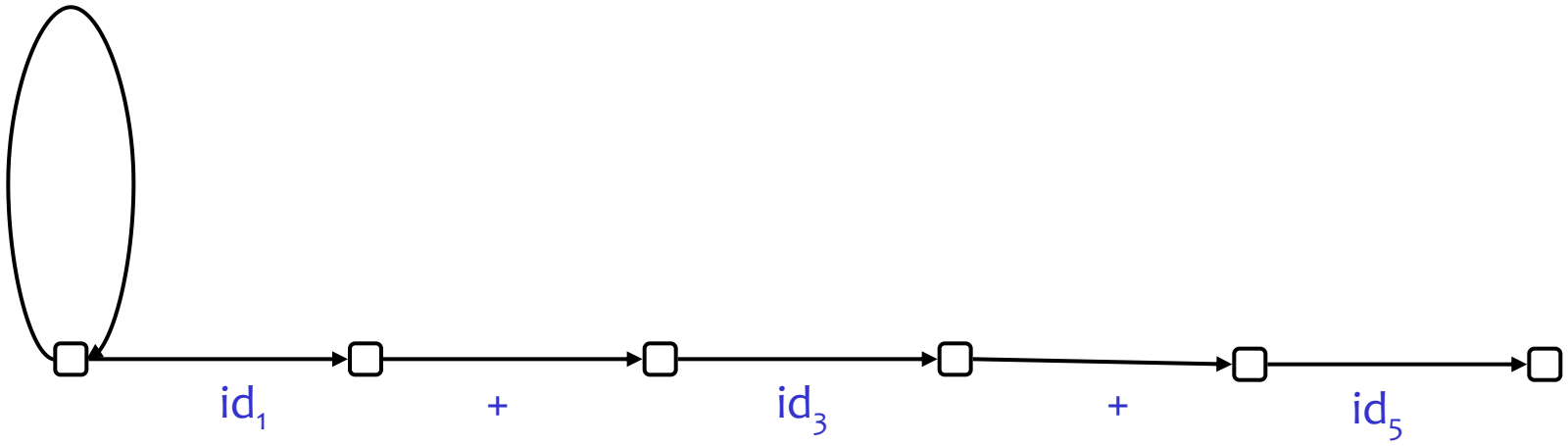
these three edges → single edge with three labels

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

grammar:

E --> T + id | id

T --> E



Example (2)

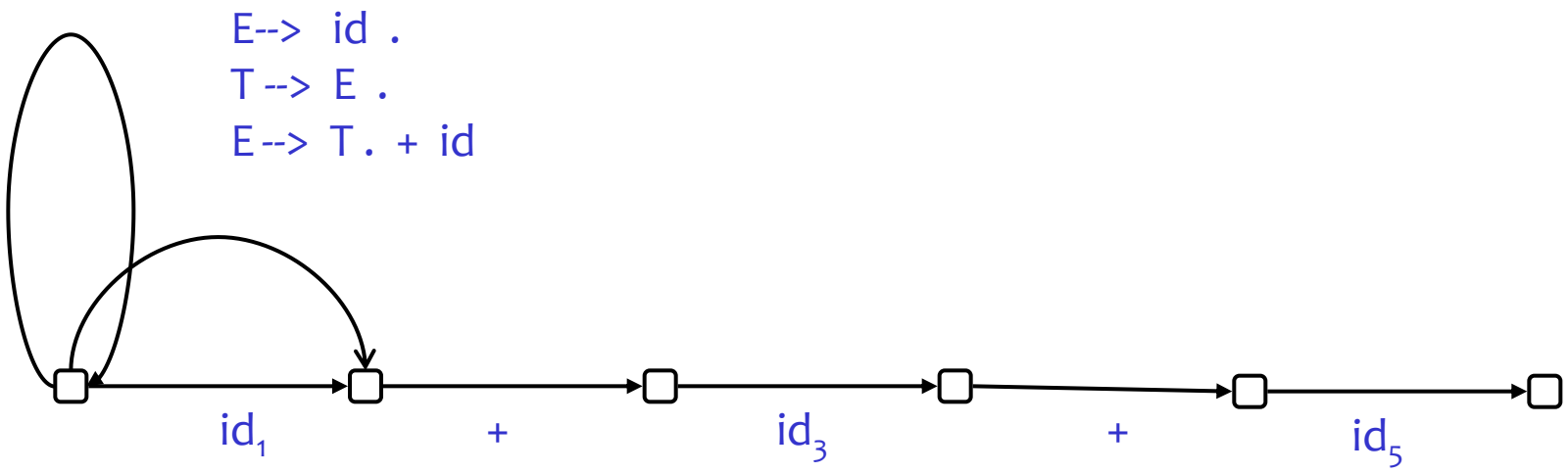
We add a complete edge, which leads to another complete edge, and that in turn leads to a in-progress edge

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

grammar:

E --> T + id | id

T --> E



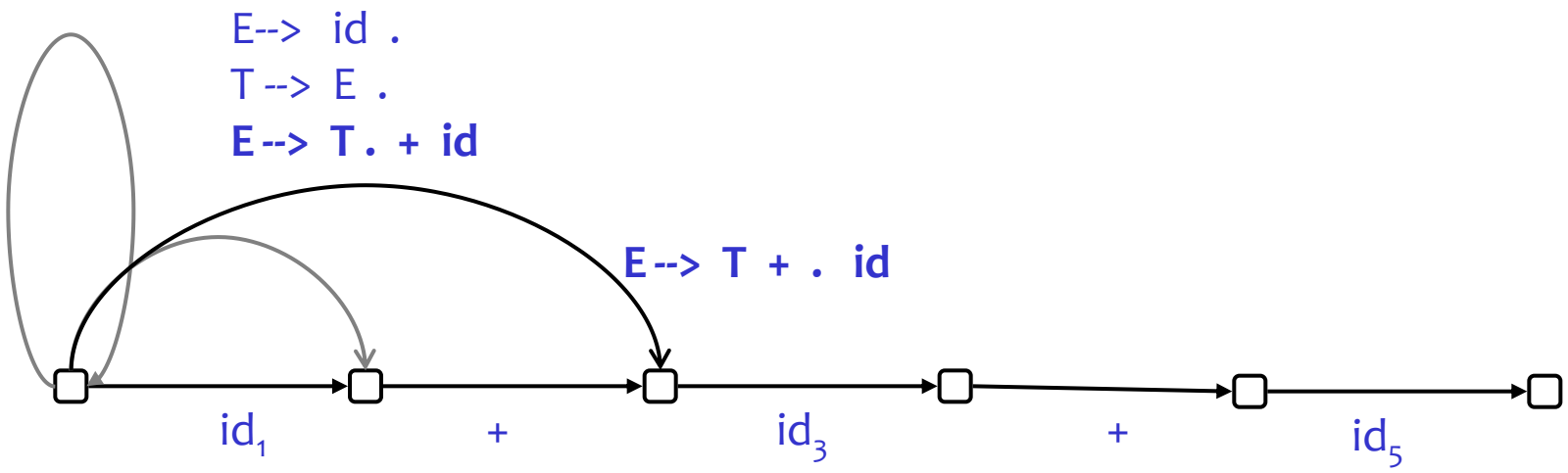
Example (3)

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

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

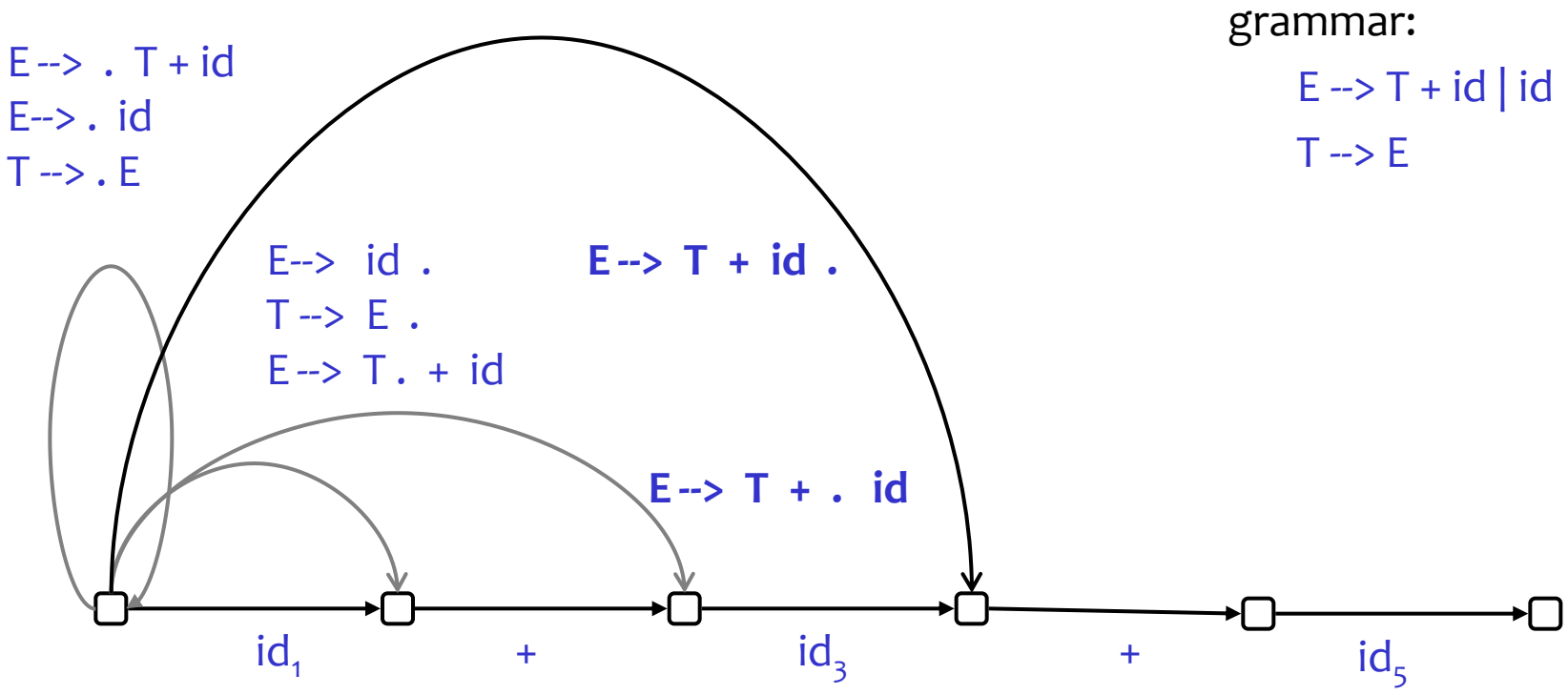
grammar:

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



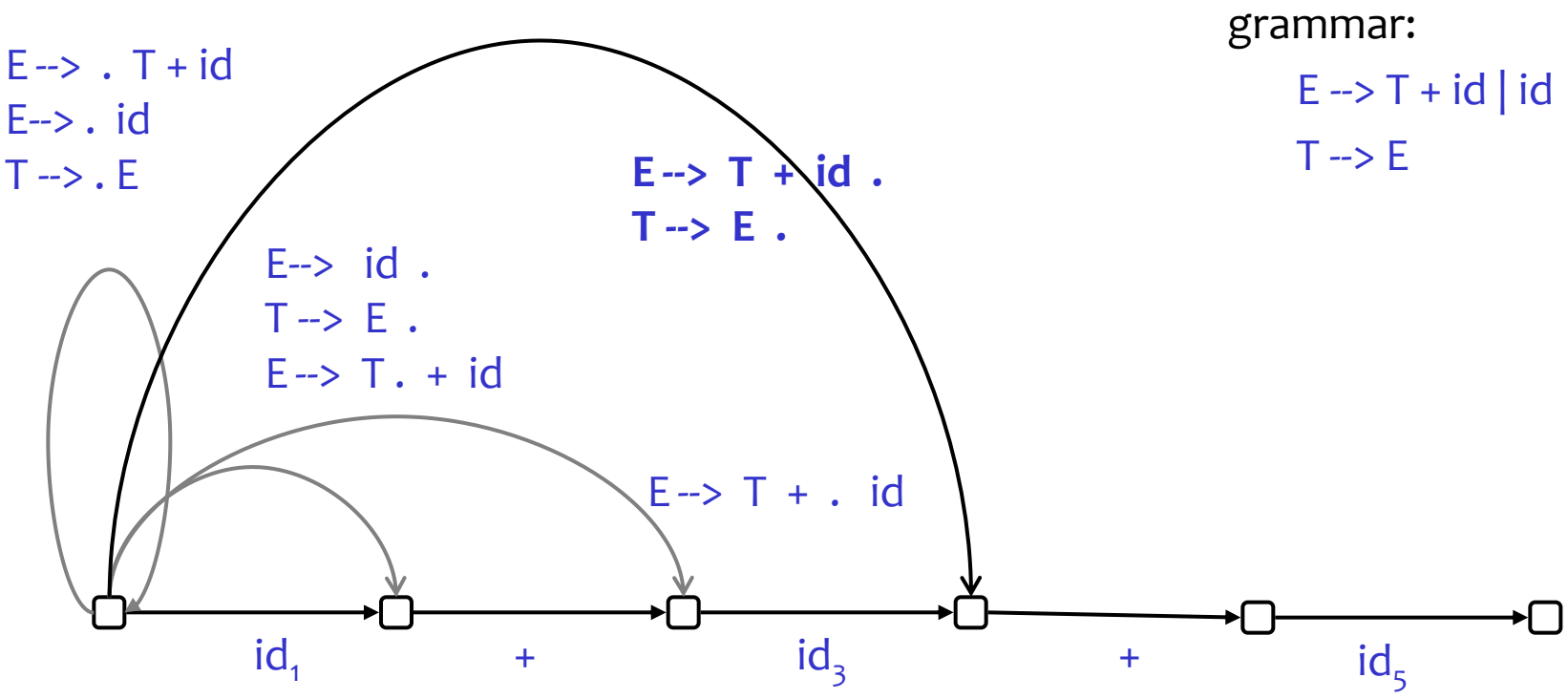
Example (4)

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



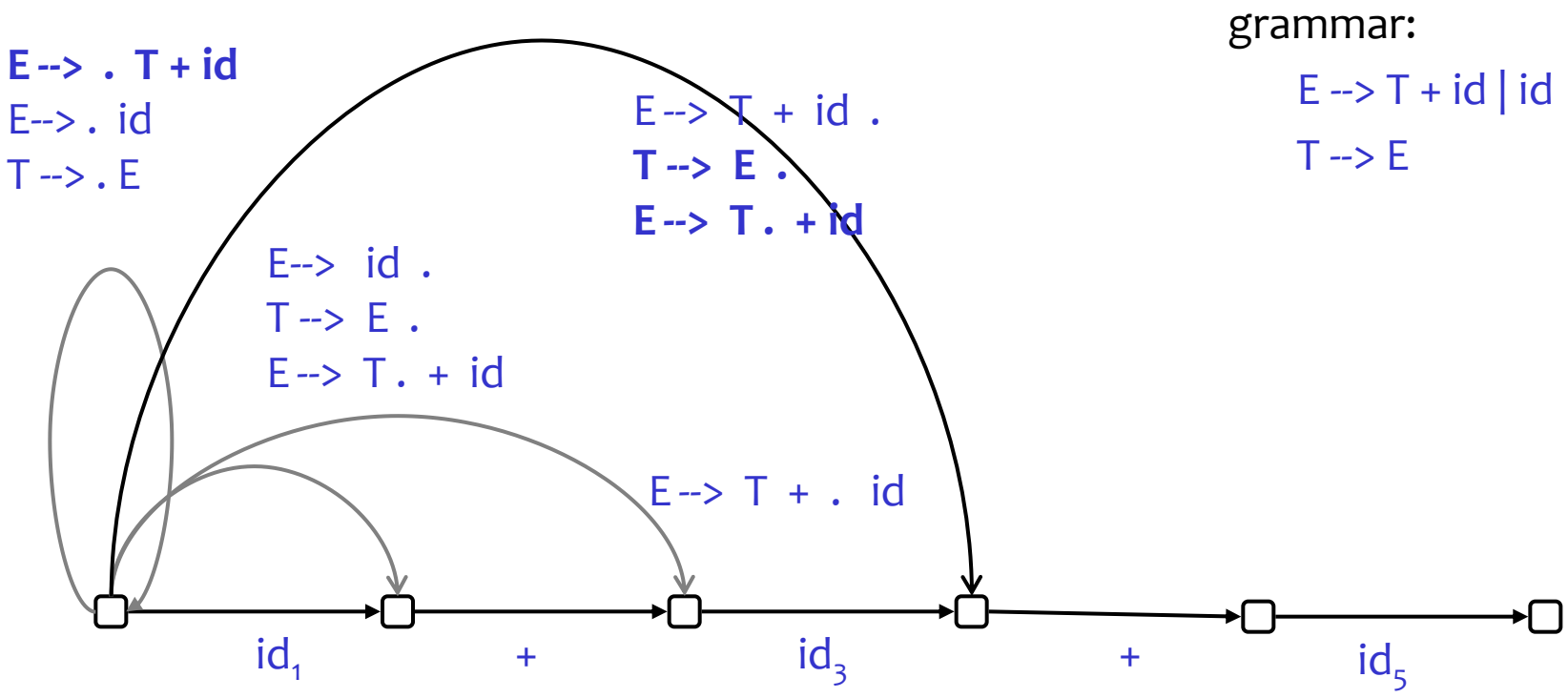
Example (5)

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



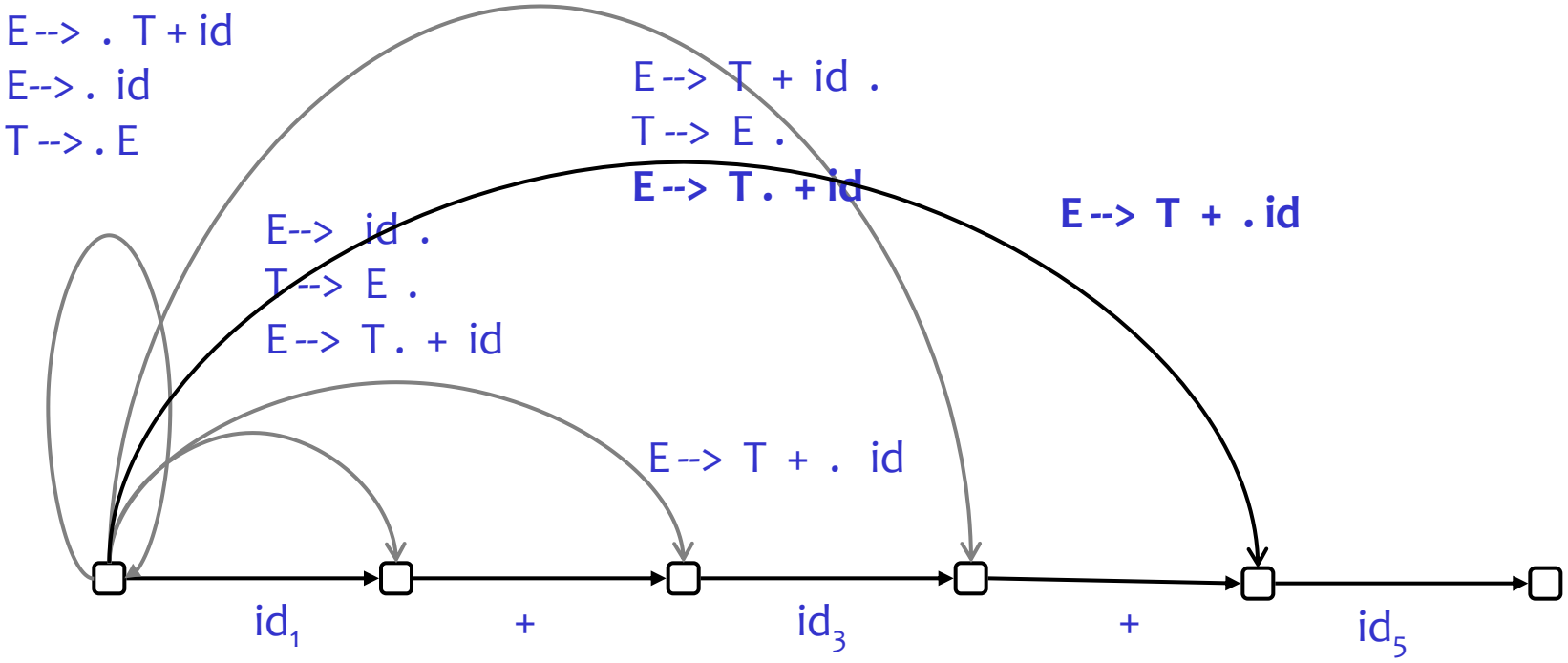
Example (6)

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



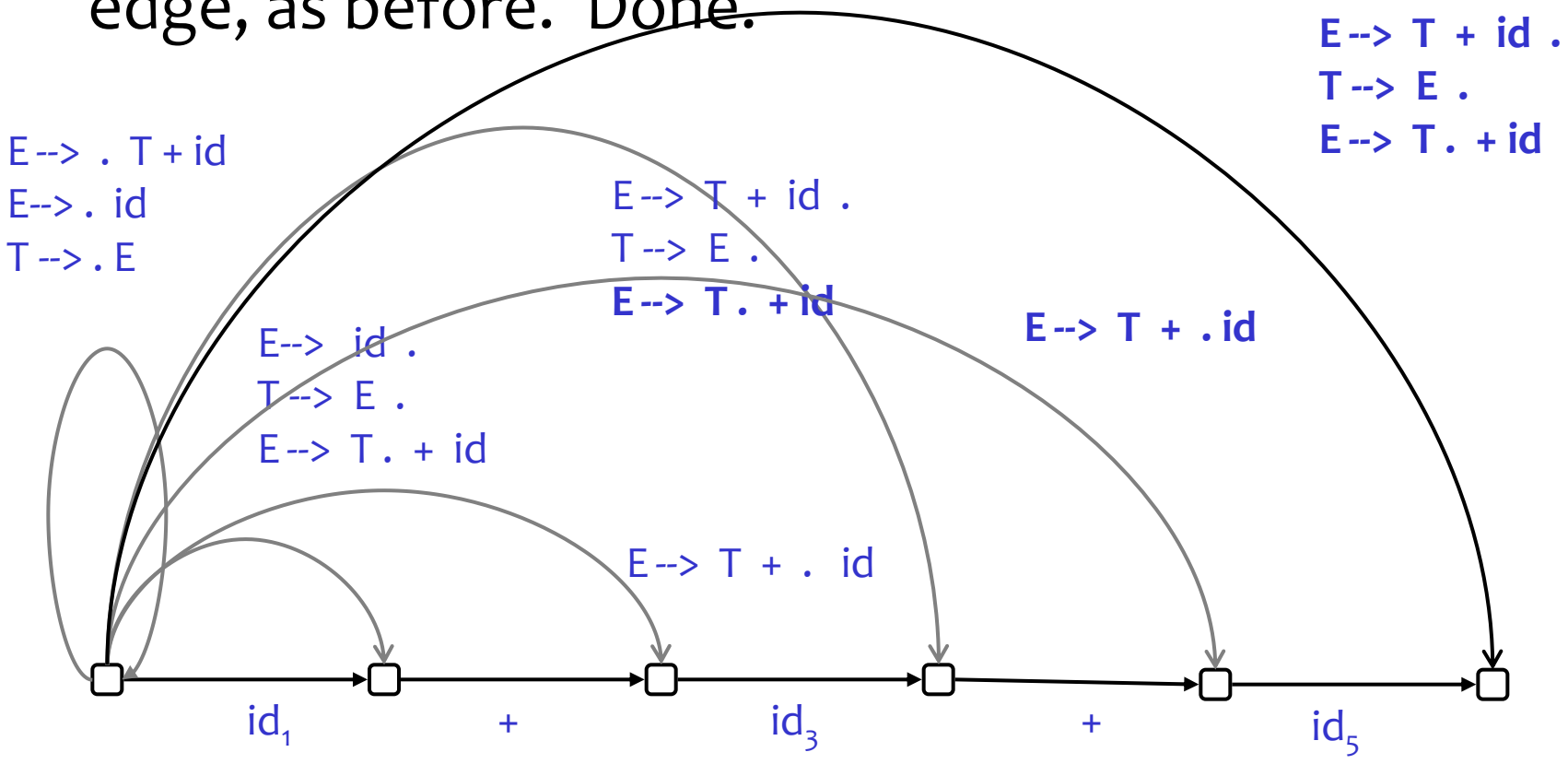
Example (7)

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



Example (8)

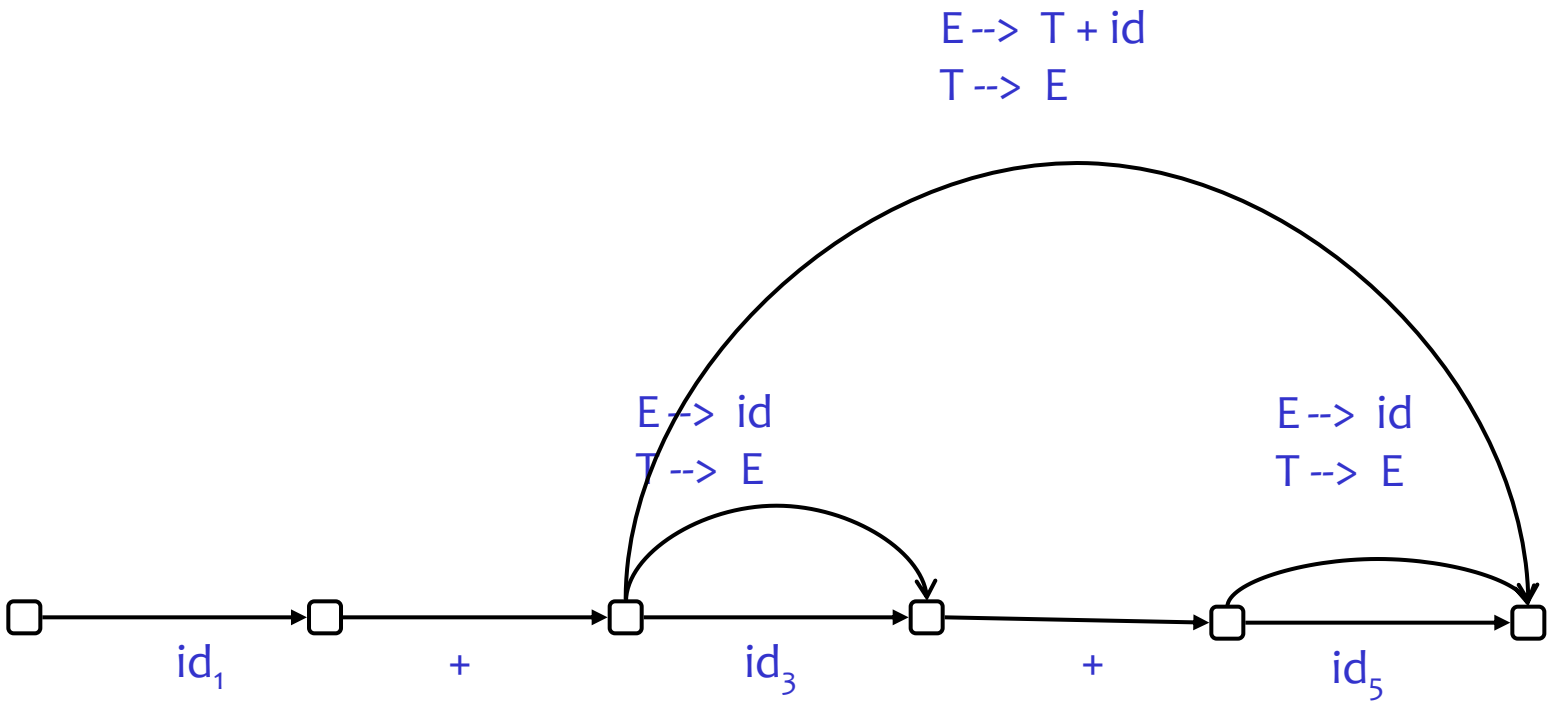
Advance again, creating a complete edge, which leads to a another complete edges and an in-progress edge, as before. Done.



Example (a note)

Compare with CYK:

We avoided creating these six CYK edges.



Generalize CYK edges: Three kinds of edges

Productions extended with a dot '.'

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

Completed: A --> B C .

We found an input substring that reduces to A

These are the original CYK edges.

Predicted: A --> . B C

we are looking for a substring that reduces to A …

(ie, if we allowed to reduce to A)

… but we have seen nothing of B C yet

In-progress: A --> B . C

like (2) but have already seen 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: moves the dot across a terminal found next on the input

Repeat until no more edges can be added:

Predict: adds predictions into the graph

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

HW4

You'll get a clean implementation of Earley in Python

- It will visualize the parse.
- But it will be very slow.

Your goal will be to optimize its data structures

- And change the grammar a little.
- To make the parser run in linear time.