Parsers

Top-down, bottom-up, disambiguation.

CS164: Introduction to Programming Languages and Compilers

Fall 2009

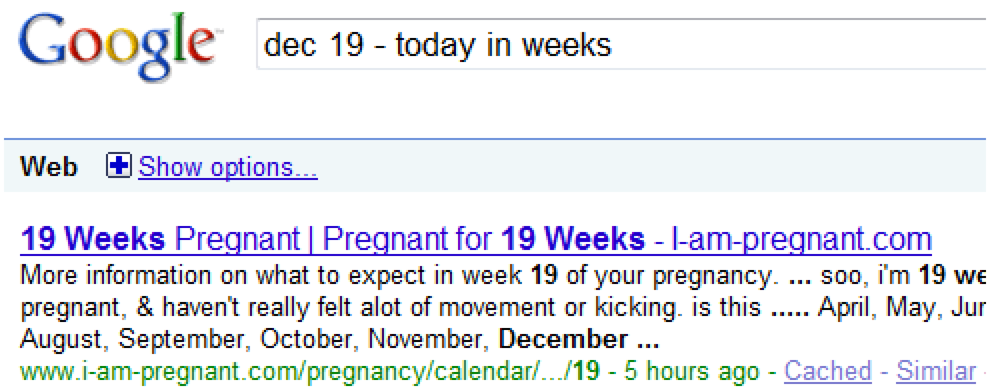
Instructor: **Ras Bodik**

GSI: **Joel Galenson**

UC Berkeley

Administrativia: Projects

How much time is left?



Dec 19 – today in weeks = 12 weeks

Administrativia: Projects

Project 1: you learnt about

- Metaprogramming, regexes, DOM, event-driven pming

Project 2: build your parser and write applications (3 weeks)

- google calc to AST; HTML to DOM
- simple DOM layout ==> gives us a scriptless browser

Project 3: design and implement a small language (3 weeks)

- Dynamically typed, a'la Python, Scheme, Ruby, Lua, …
- add some constructs for extending the language
- application: embed a simple DSL

Project 4: static analysis, static typing and compilation (2 wks)

- compile rather than interpret; needs static analysis for that

Project 5: your own language (4 weeks)

- or implement what we propose

Administrativia: Exams

Exams:

- 1st Midterm (80 minutes): **Oct 20**
- 2nd Midterm (80 minutes): **Dec 3** (last lecture)
- final projects (posters, demos and pizza): **Dec 19,** 12:30-3:30

Ras' office hours are moving to TT 5-6pm

Turn off your cell phones and close laptops.

Top-down parsing

Parse by trying all derivations.

Grammar: E ::= T + E | T   T ::= int | int * T | (E)

Two derivations:

T → <ins>int</ins>

bold

underlined

E → <ins>**T** + E</ins> → <ins>int</ins> + **E** → int + <ins>**T**</ins> → int + <ins>n</ins>

E → <ins>**T** + E</ins> → <ins>int * **T**</ins> + E → int * <ins>int</ins> + **E** → int * int + <ins>**T**</ins> → int * int + <ins>n</ins>

There are infinitely more of them.

How to explore them fast, not miss any?

$E ::= E * E \mid id \mid (E) \mid E+E \mid n$

Leftmost derivation

Always rewrite the leftmost symbol

Question: due to which step is this not a leftmost derivation?

$E \to (E) \to (E * E) \to (E + E * E) \to (n + E * E) \to (n + E * \underline{id}) \to (n + id * id)$

choices

$\left.\begin{array}{l} (E) \\ E+E \\ \vdots \end{array}\right\}$ 5 choices

Backtracking parser: derive all strings

- For each choice, keep a list of unexplored choices
- when all choices exhausted:
- backtrack to previous choice and try next choice

Efficiency:

do we need to derive till we get the final string?

Backtracking Parser

Grammar: E ::= T + E | T   T ::= int | int * T | (E)

Input: int * int

oops, mismatch => backtrack

E → T + E → int + E

oops

match

oops

E → T + E → int * T + E → int * int + E

oops

→ int * T + E → int * int * T + E

E → T + E → (E) + E   int * (E) + E

mismatch

E → T → int

E → T → $\boxed{\text{int *}}$ T → int * int

prefix of terminals only — what you match against input

$E ::= E \mid \underline{id}$

sometimes it works to fix the order

$E \to E \to E \to E \to E \to$

Parsing arbitrary grammars

left recursive

$E \to \dots \to E\underline{xdF}$

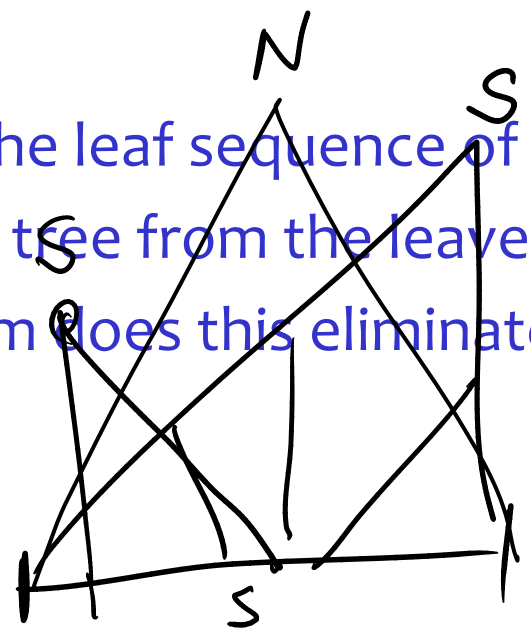
Why grow parse trees from the root?

Top-down parsers:

- proceed top down, from start non-terminal
- iterate over all derivation sequences, building trees
- sub-trees discarded during backtracking, then rebuilt

Bottom up parsers

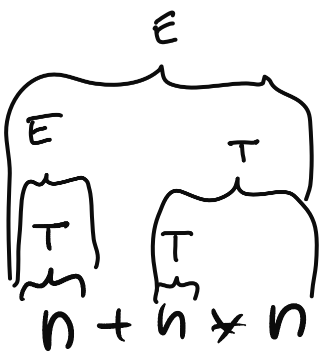
- The idea: the string is the leaf sequence of the parse tree
- can we grow the parse tree from the leaves to the root
- If we can, what problem does this eliminate?

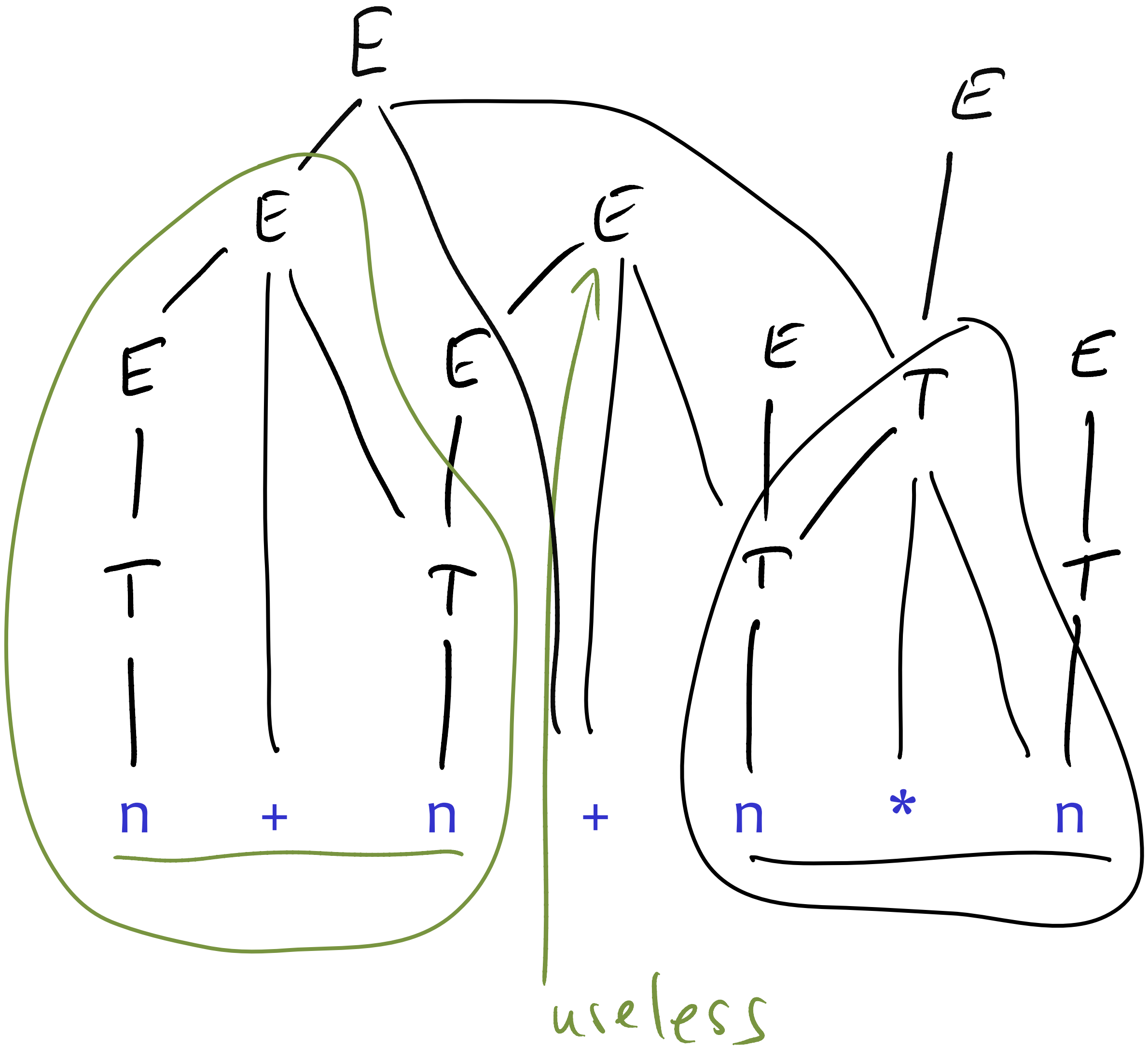


Grow the parse tree from the leaves

E ::= E + T | T

T ::= T * n | n





Input: n + n + n * n

CYK parser

Let's invent a way to grow the trees cleanly

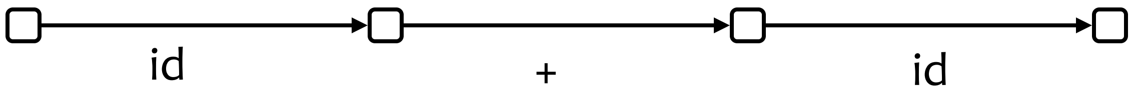
E ::= E + E | id

Input: id + id

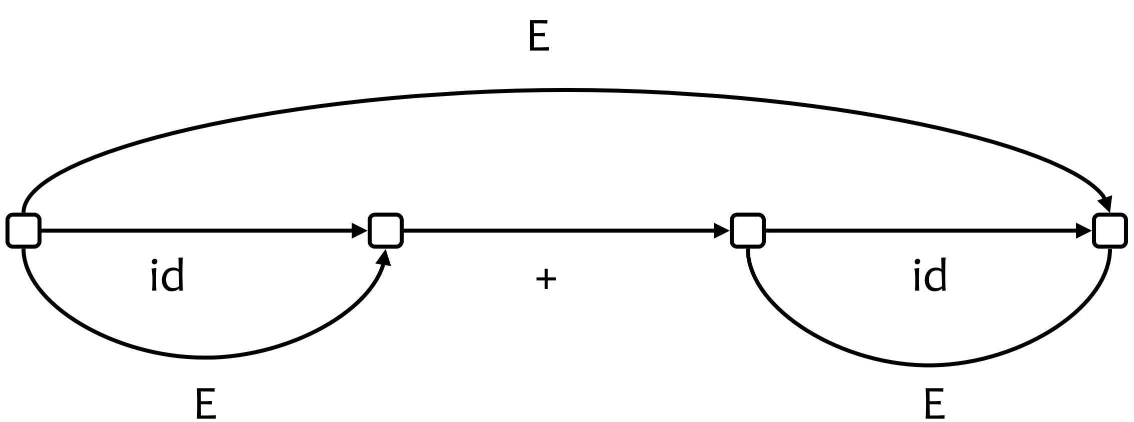
Bottom-up parsing: sequence of reductions

Parse trees represented as graphs

terminal Edges



non-terminal Edges



Key invariant

Edge (i,j,T) exists iff $T \rightarrow^* \text{input}[i:j]$

- **$T \rightarrow^*$ 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

CYK on a non-ambiguous grammar

assume arbitrary non-ε grammar

example

DECL → TYPE VARLIST ;

TYPE → int | float

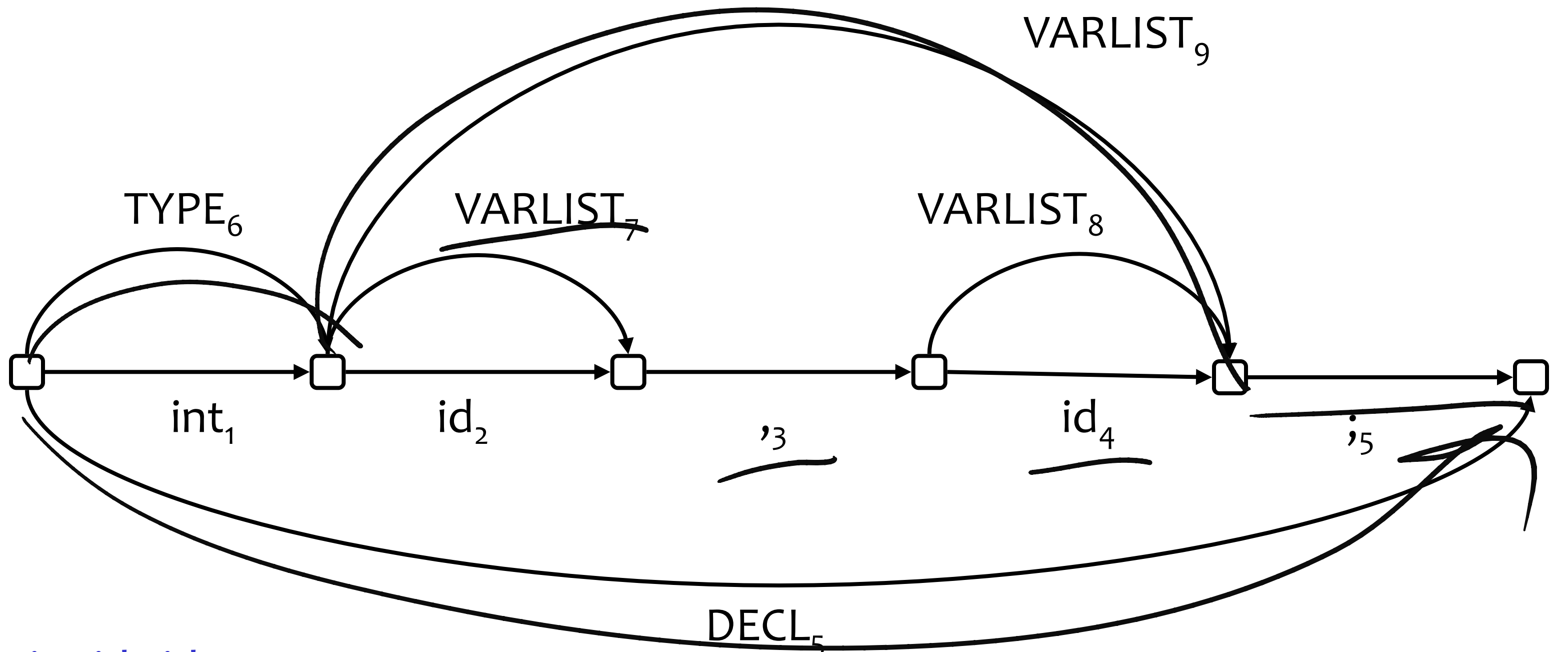
VARLIST → id | VARLIST , id

Example of CYK execution

Start with terminal edges, then:

keep adding non-terminal edges until no edge can be added.

edge added when any adjacent edges form rhs of a production



Input: int id , id ;

Constructing the parse tree

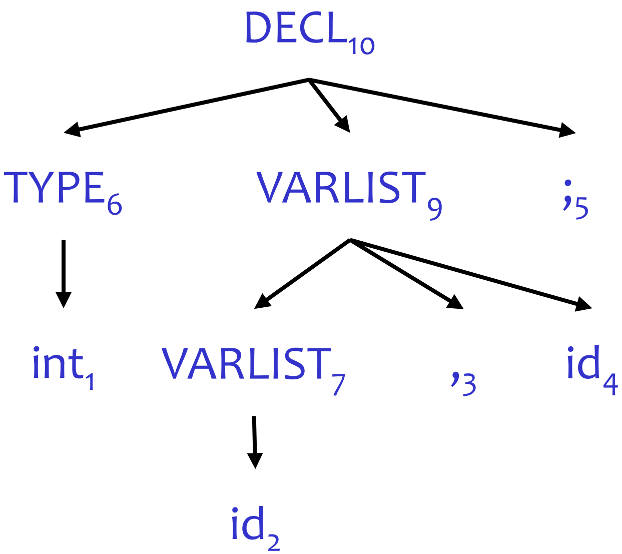
Nodes in parse tree correspond to edges in CYK reduction

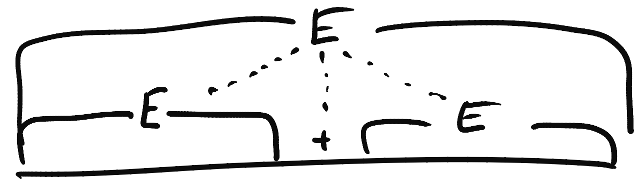
- edge *e*=(0,N,S) corresponds to the parse tree root *r*
- edges that caused insertion of *e* are children of *r*
- and so on

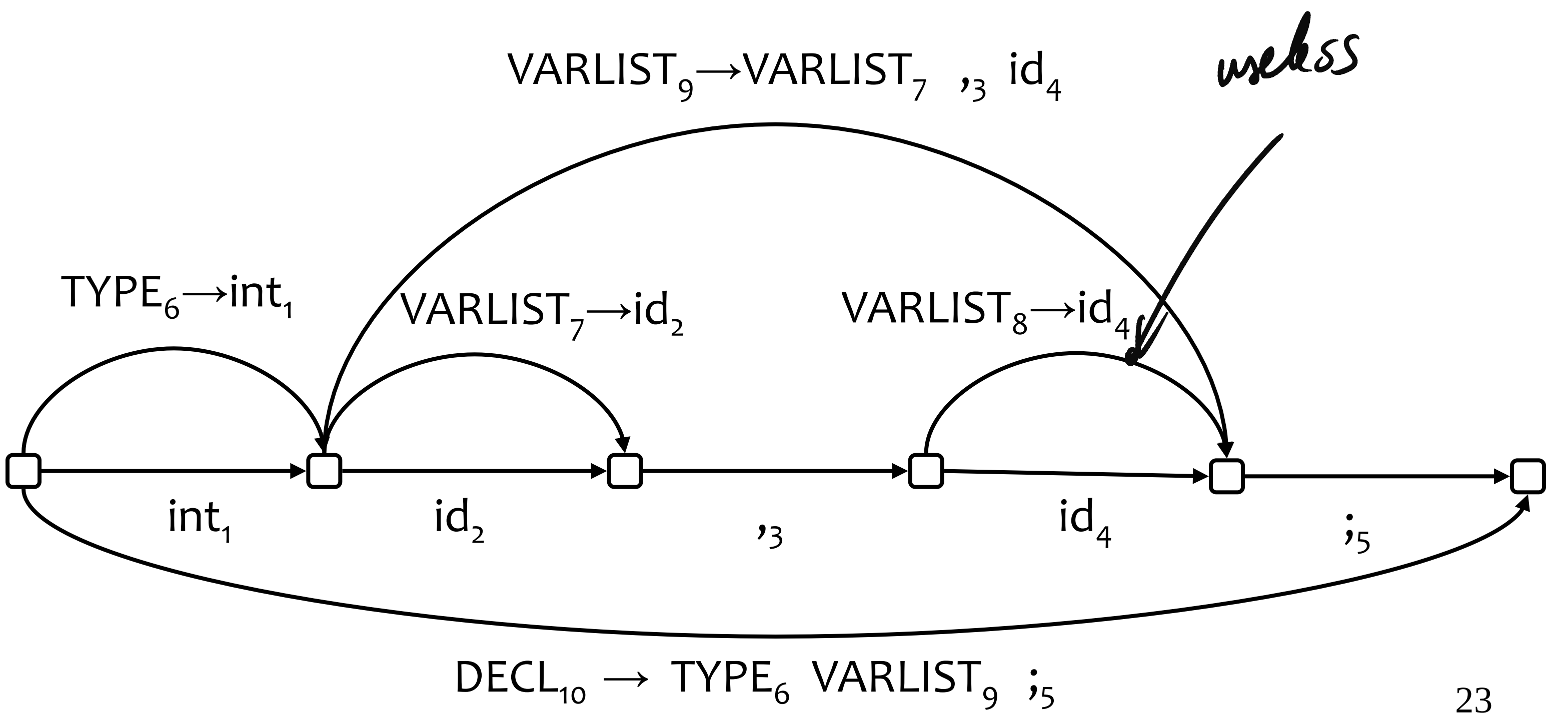
Helps to label edges with entire productions

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

Example of CYK execution







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:

- $A \longrightarrow BC$ or
- $A \longrightarrow \mathrm{d}$ or
- $S \longrightarrow \varepsilon$ (only start non-terminal can derive ε)

CYK: the algorithm (can you find the bug?)

```
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"
```

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

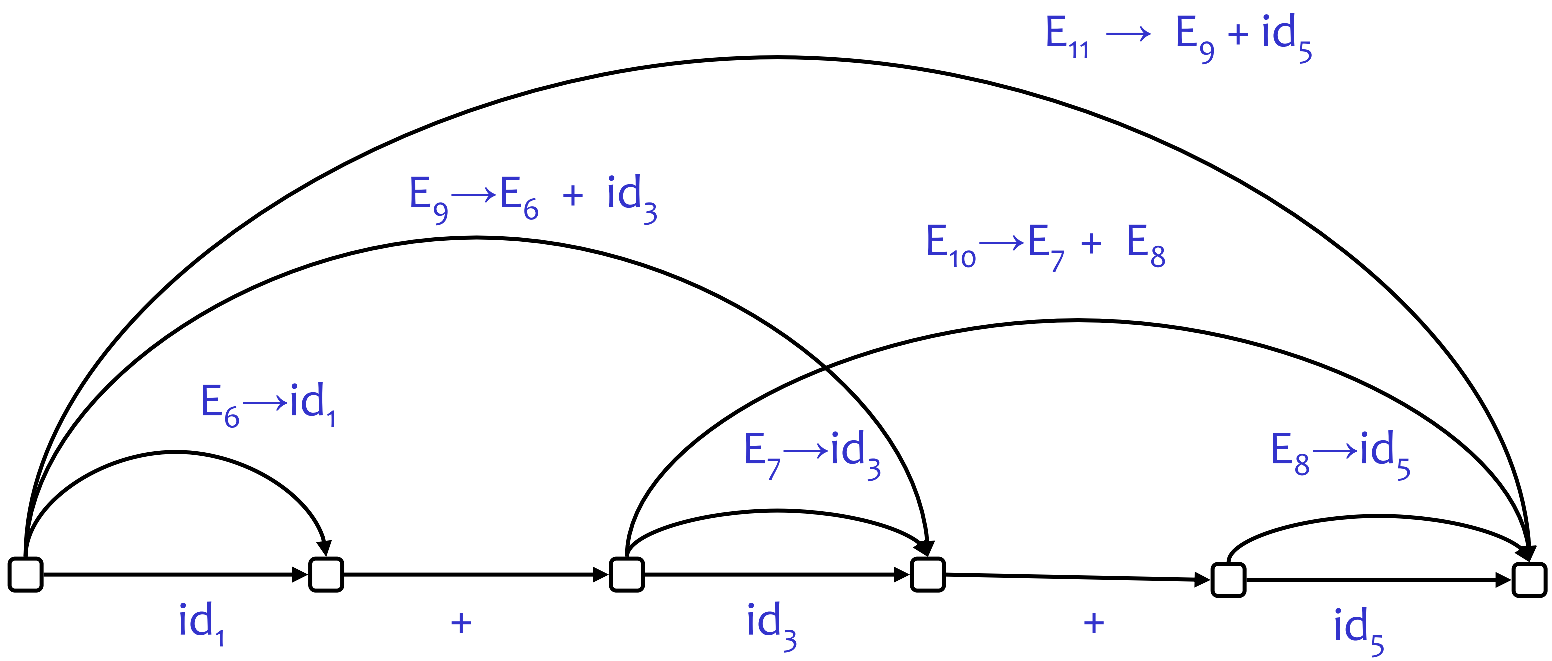
grammar: E ::= E+id | id

input: id+id+id

Can you spot the inefficiency?

Example

- grammar: $E \longrightarrow E+id \mid id$
- three useless reductions are done (E_7, E_8 and E_{10})



Earley parser fixes (part of) the inefficiency

Earley does not eliminate all such redundant parse trees

- (find a simple grammar + input s/t a useless subtree is built)

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 some the key idea from those parsers

Key idea

Process the input left-to-right

as opposed to arbitrarily, as in CYK

Reduce only productions that appear non-useless

based on what we saw in the input so far

after seeing more, we may still realize we built a useless tree

In other words:

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

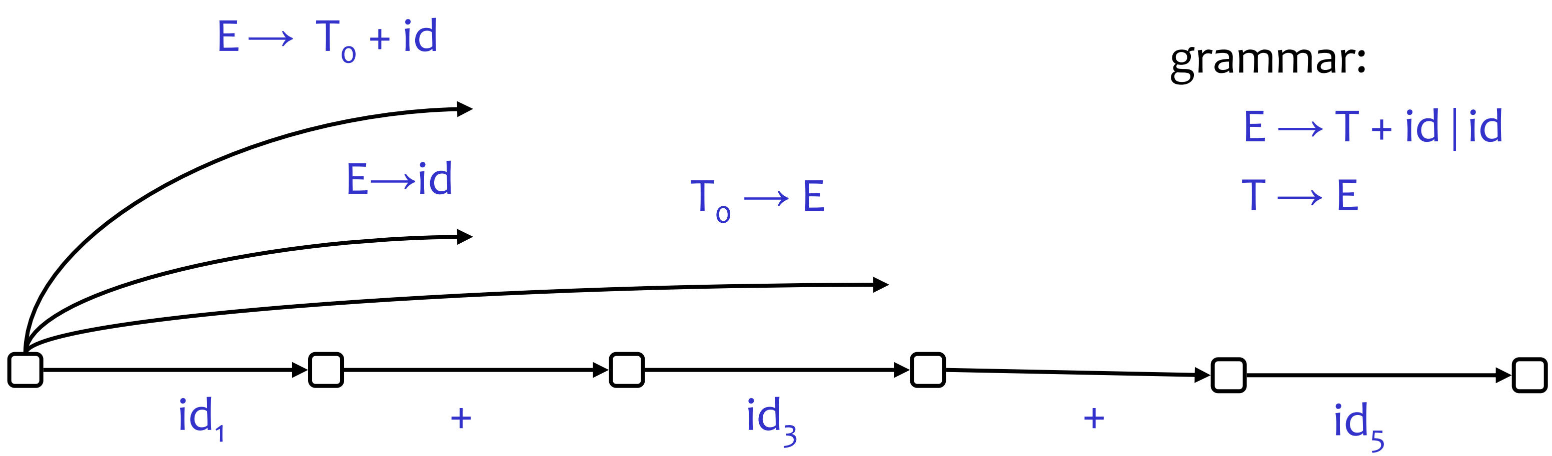
How do we accomplish this?

- with some top-down parsing logic!
- Earley combines bottom-up and top-down parsing

The intuition

What reductions can possibly emanate from 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:

$E \rightarrow T + id \mid id$

$T \rightarrow E$

Prediction

Prediction (def):

determining which productions apply at current point of input

performed top-down

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

Generalize CYK edges: Three kinds of edges

Productions extended with a dot '.'

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

1. **Completed:** $A \to B\ C\ .$

   We found an input substring that reduces to A

   These are the original CYK edges.

2. **Predicted:** $A \to .\ B\ C$

   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

3. **In-progress:** $A \to B\ .\ C$

   like (2) but have already seen substring that reduces to B

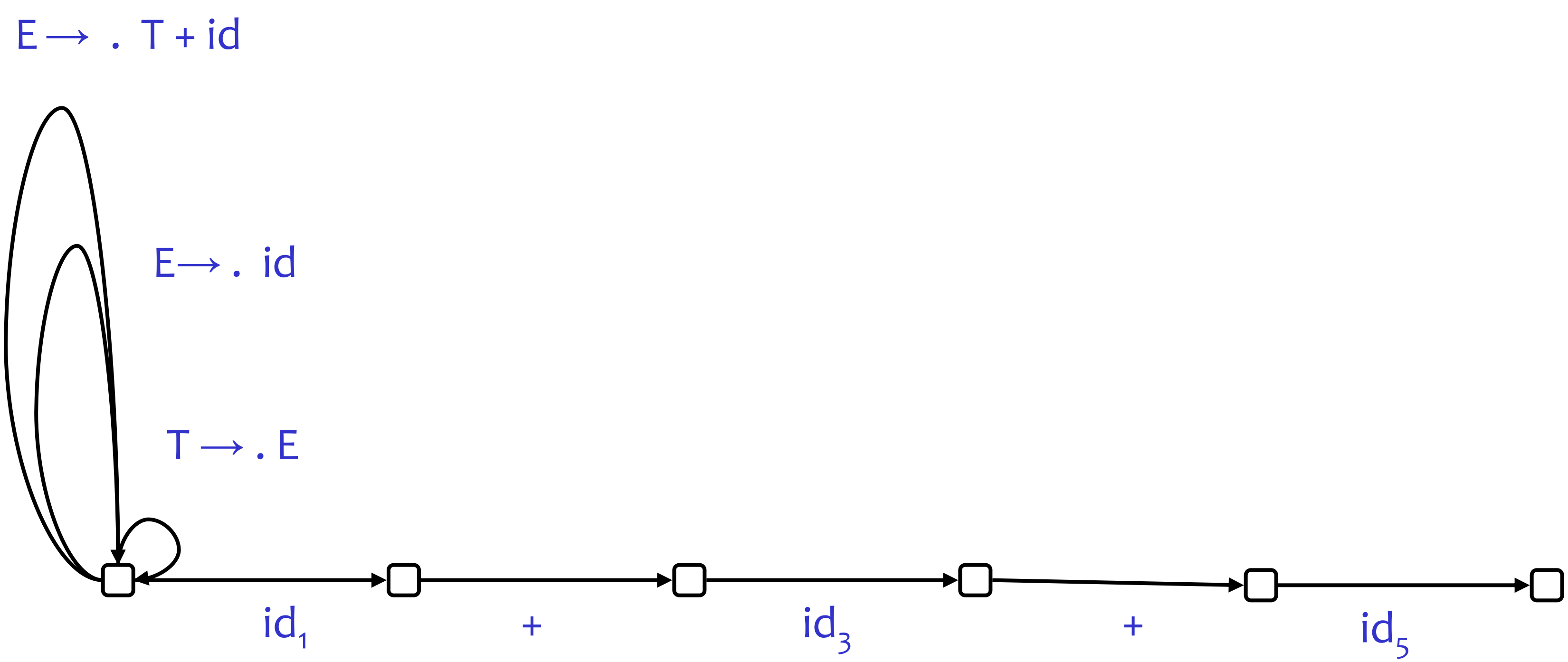
Example (1)

Initial predicted edges:

grammar:

$E \longrightarrow T + id \mid id$

$T \longrightarrow E$



Example (1.1)

Let's compress the visual representation:

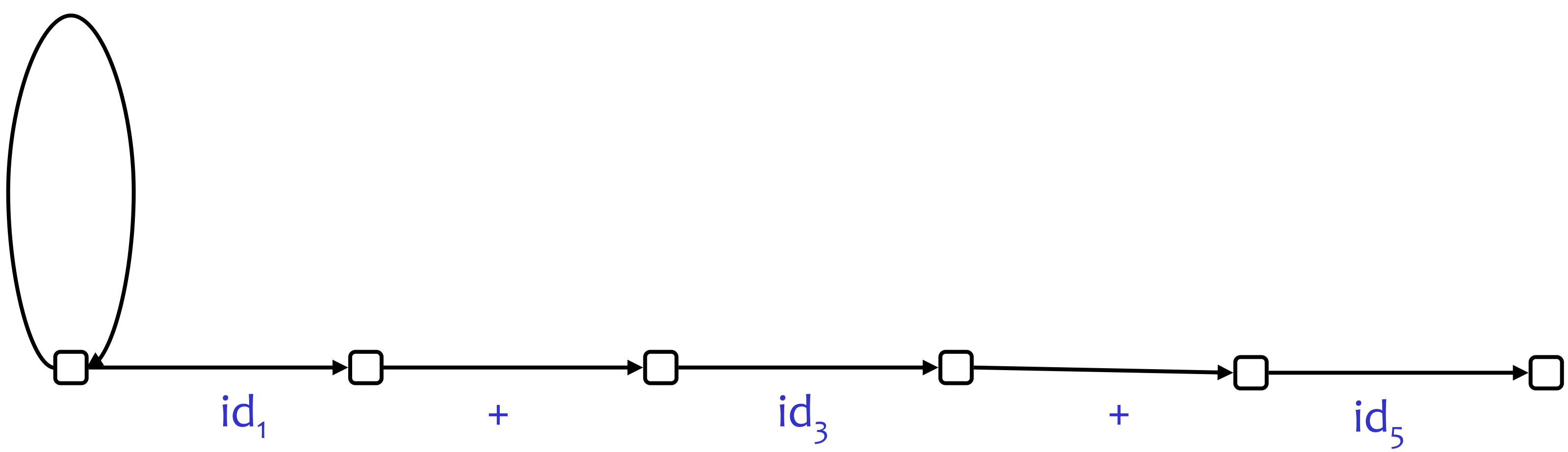
these three edges → single edge with three labels

$E \longrightarrow\ .\ T + id$
$E \longrightarrow .\ id$
$T \longrightarrow . E$

grammar:

$E \longrightarrow T + id \mid id$

$T \longrightarrow E$



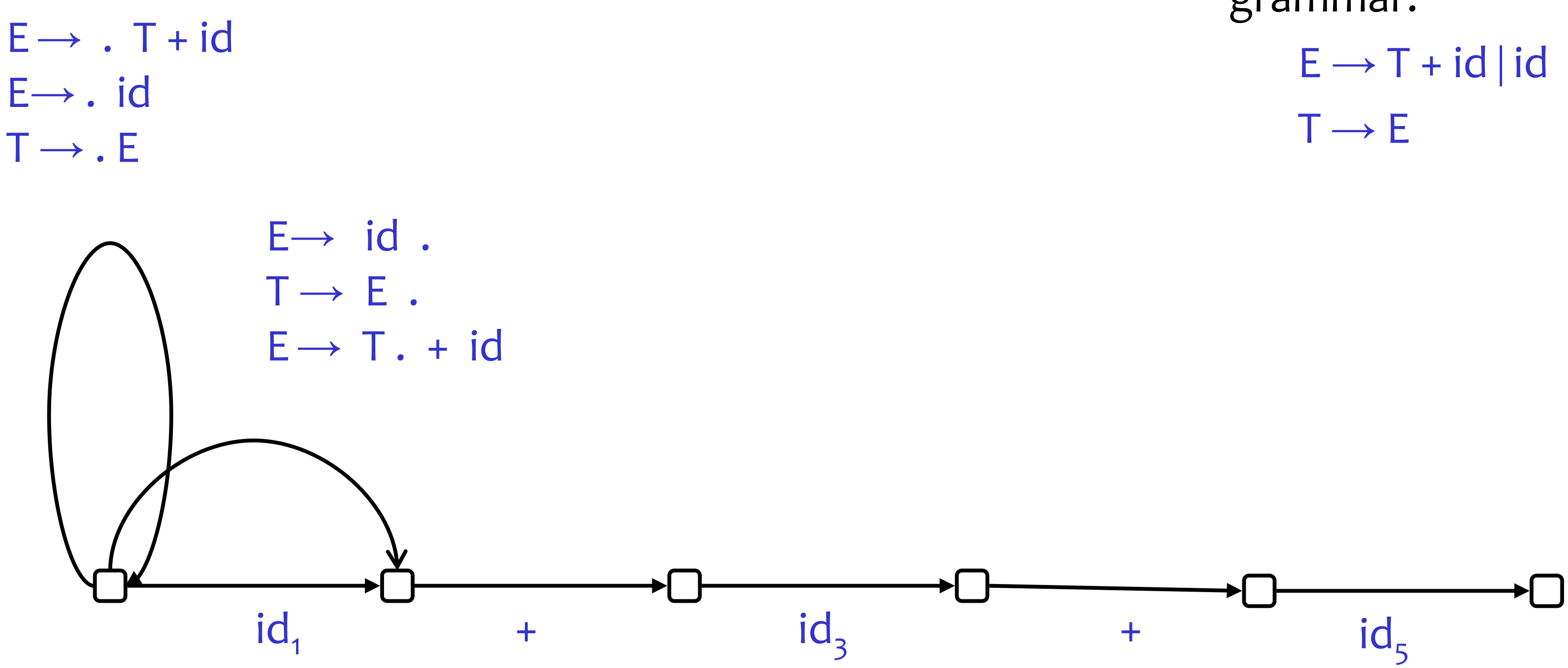
Example (2)

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

grammar:

$E \longrightarrow T + id \mid id$

$T \longrightarrow E$



Example (3)

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

$E \longrightarrow\ .\ T + id$

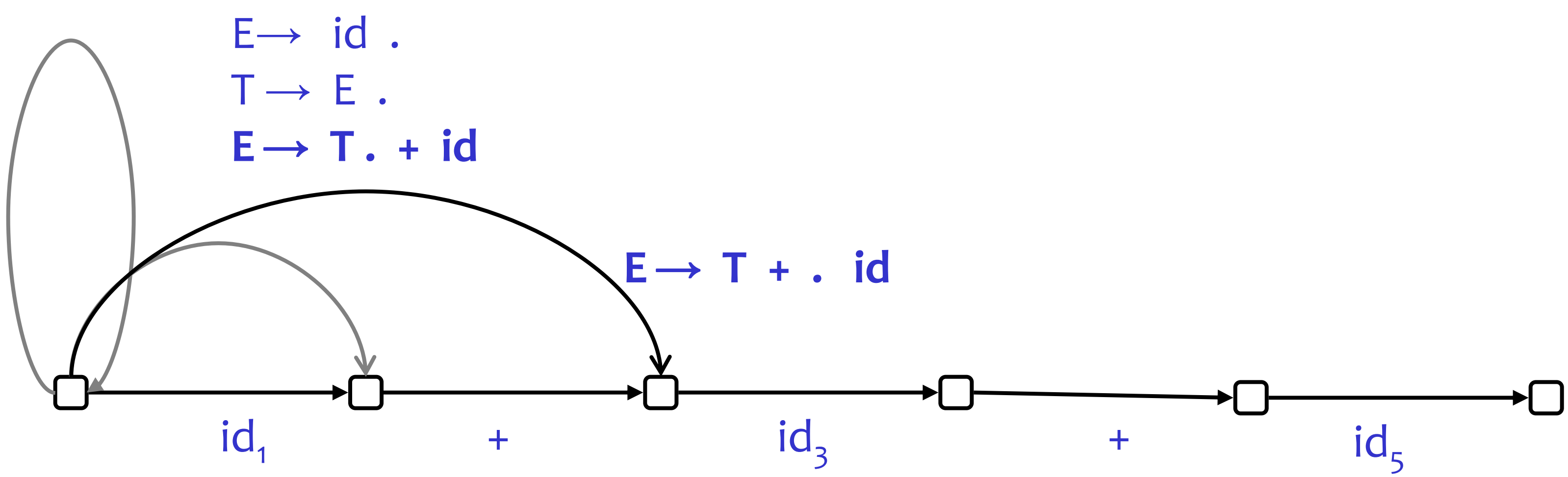
$E \longrightarrow .\ id$

$T \longrightarrow .\ E$

grammar:

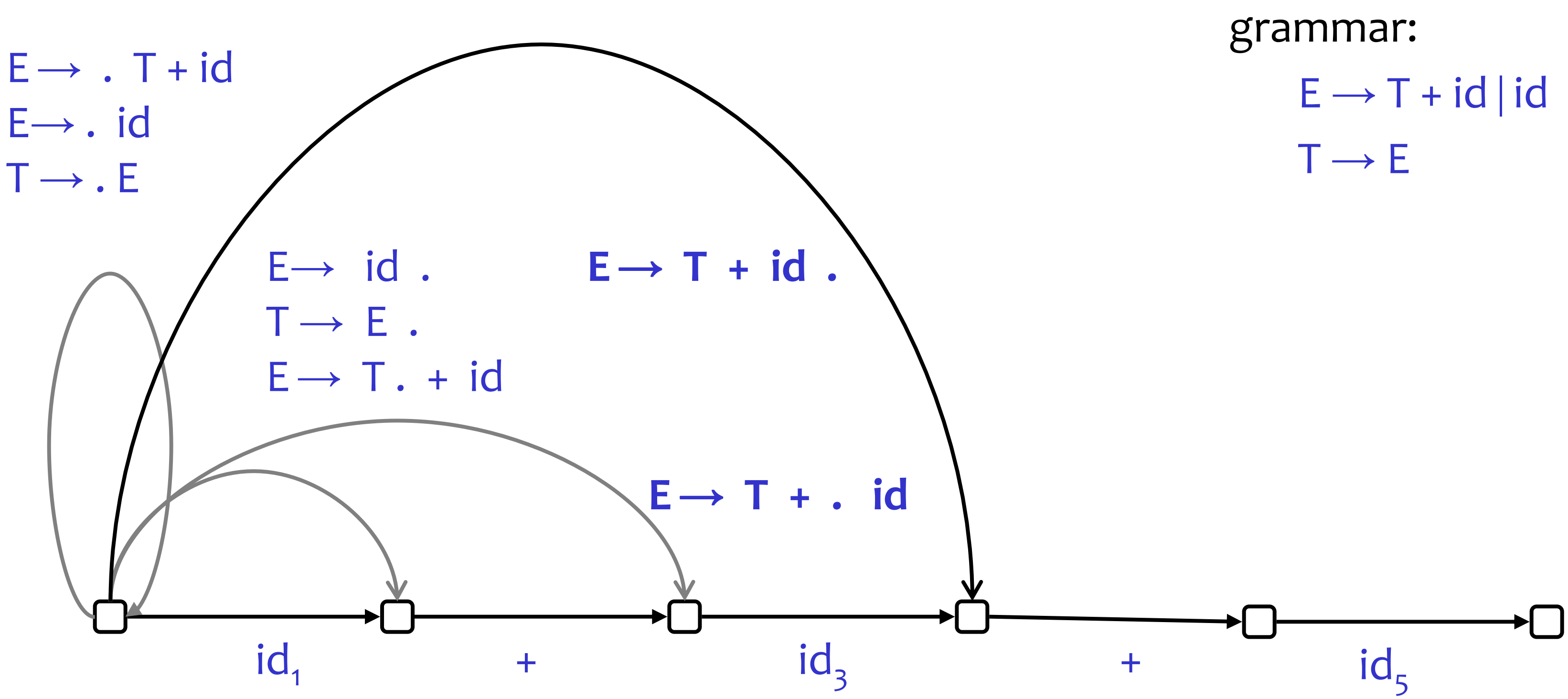
$E \longrightarrow T + id \mid id$

$T \longrightarrow E$



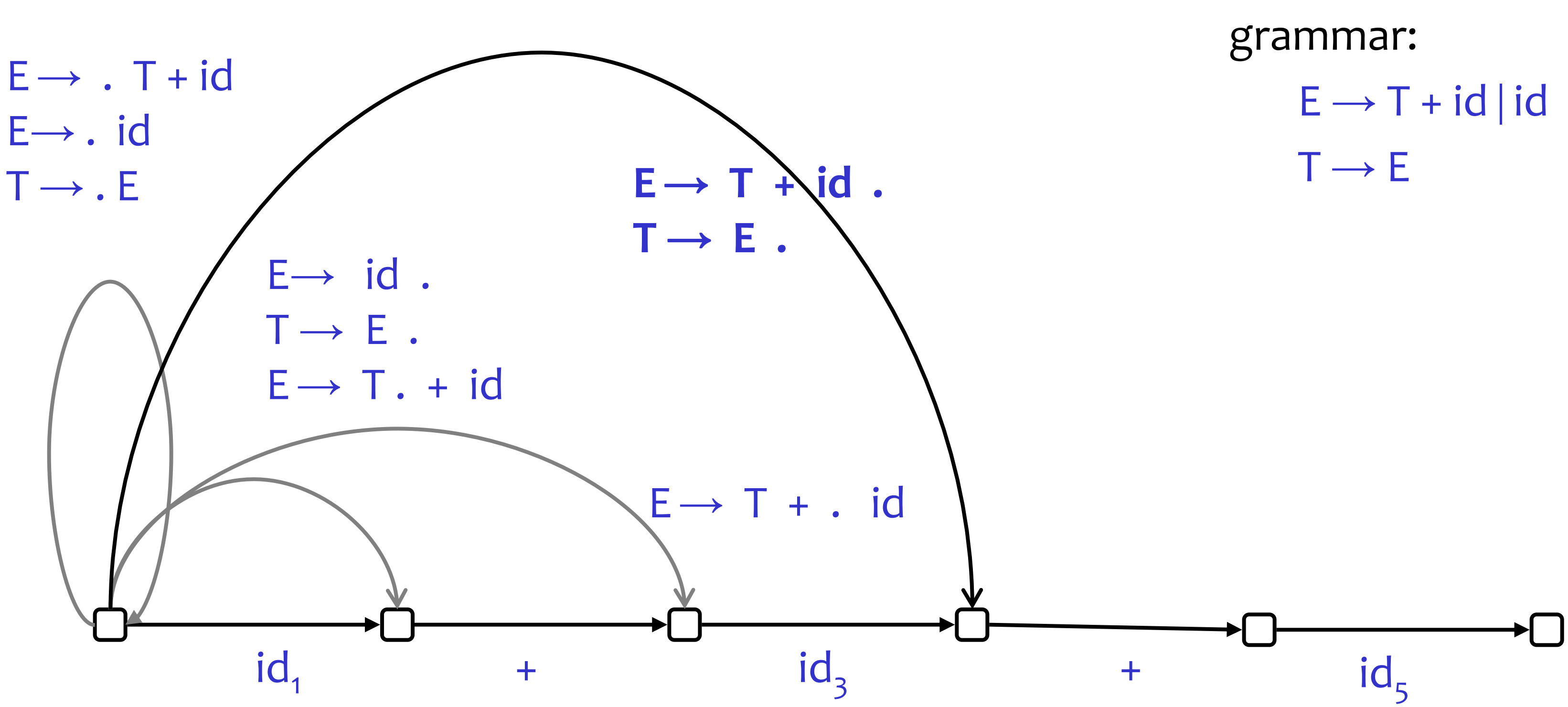
Example (4)

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



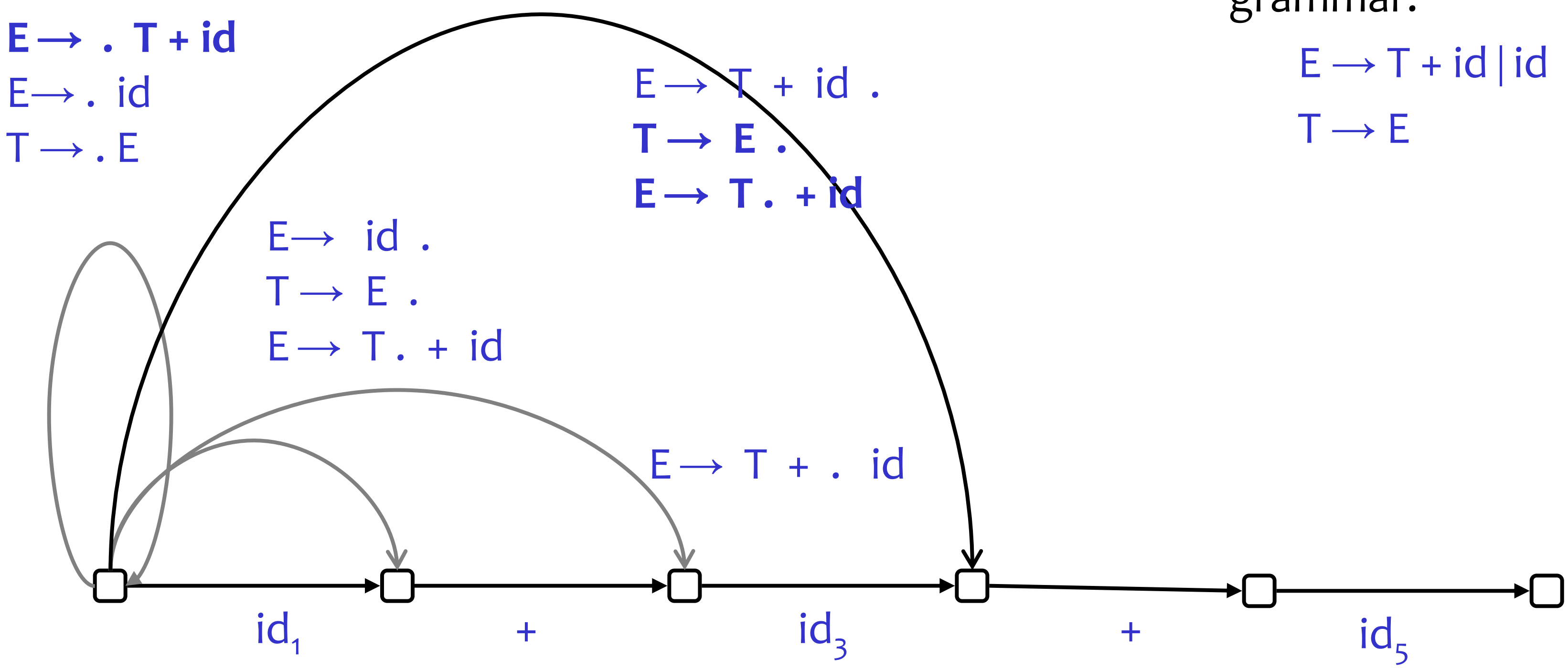
Example (5)

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.



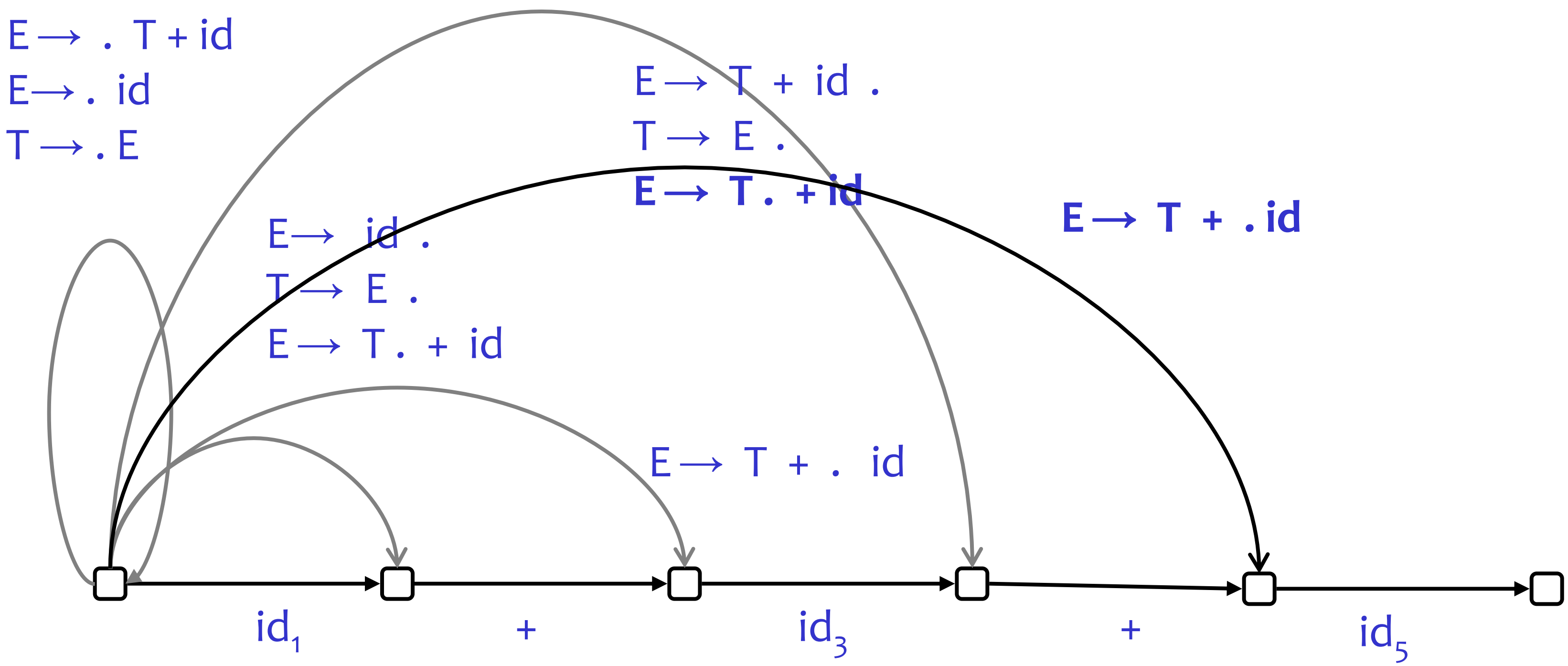
grammar:

E ⟶ T + id | id

T ⟶ E

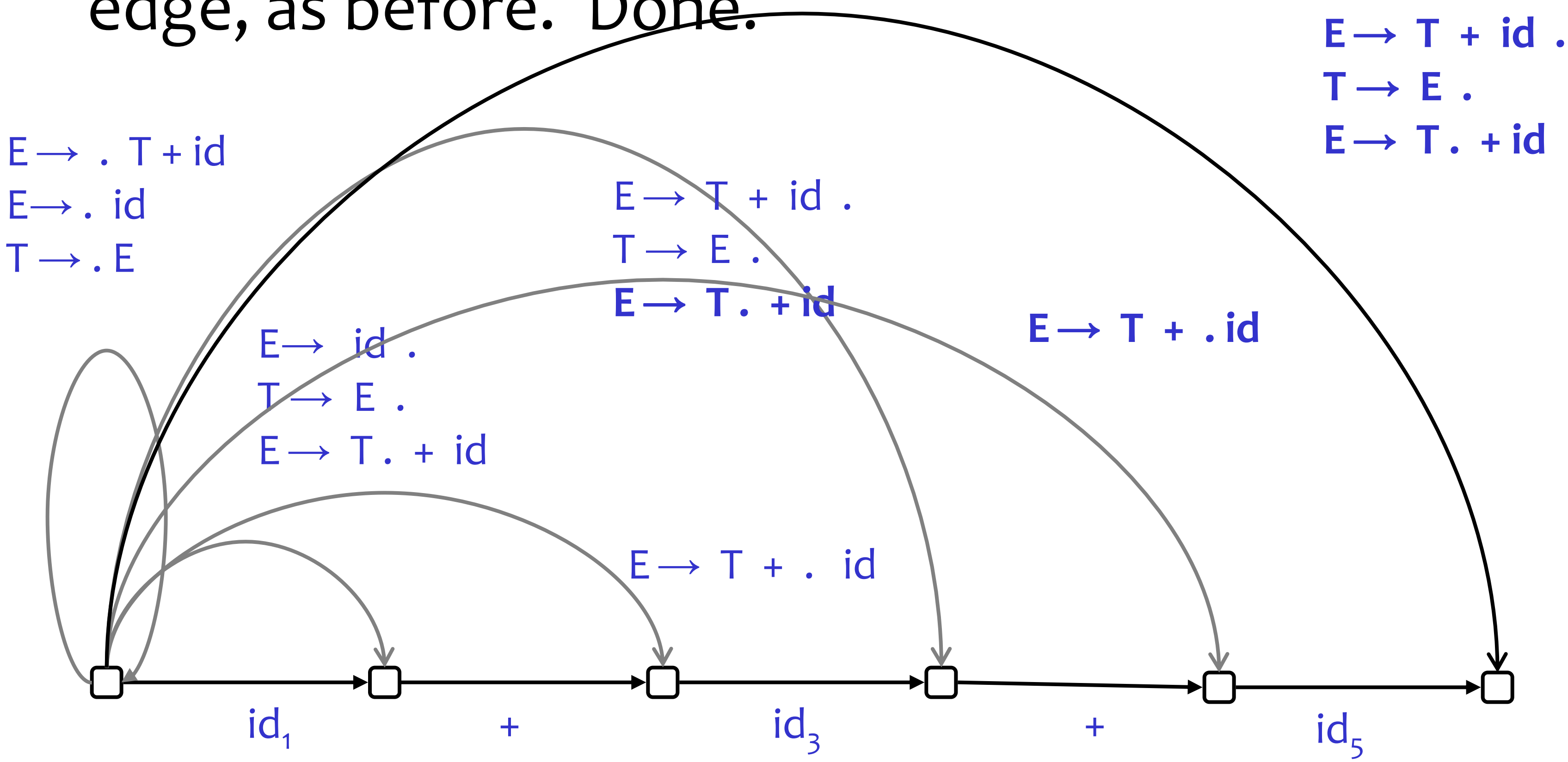
Example (7)

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



Example (8)

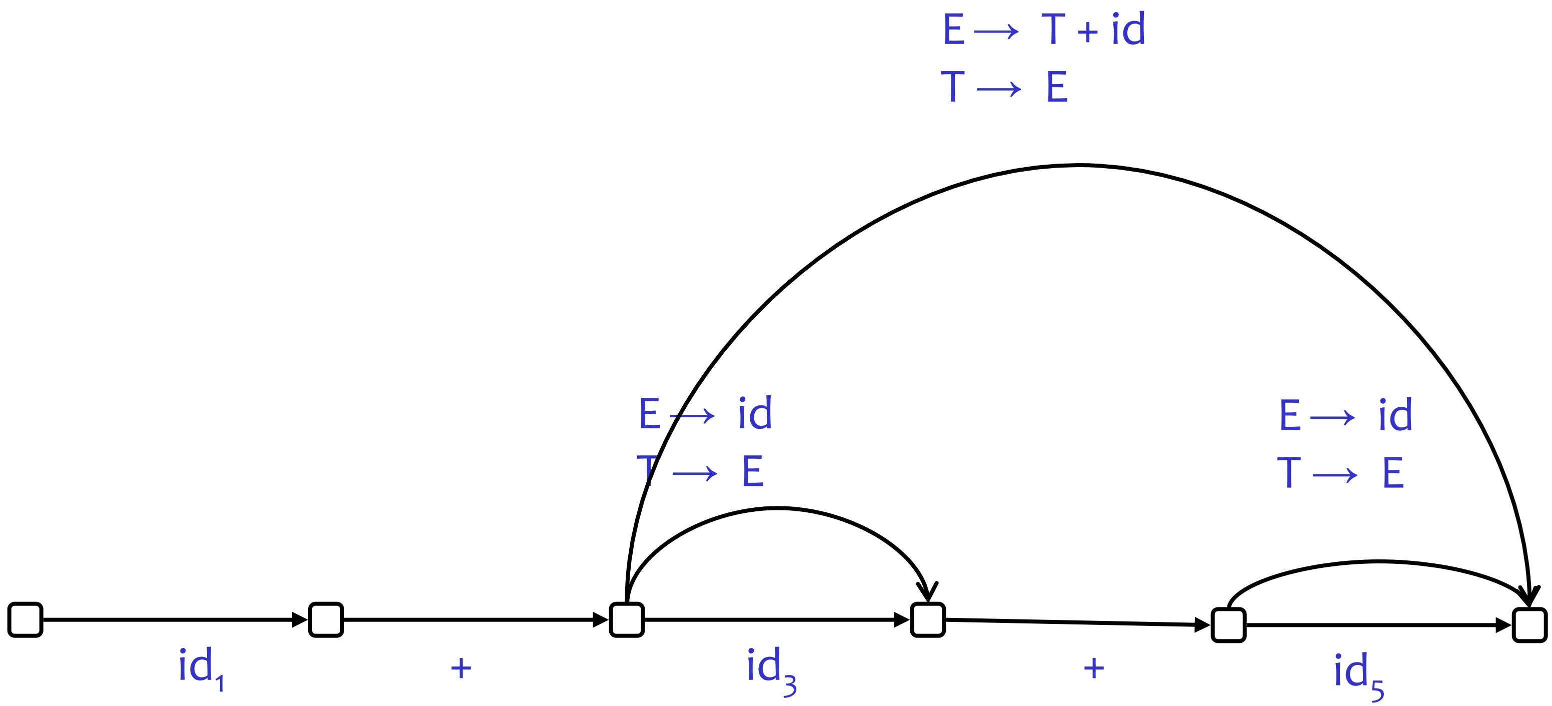
Advance again, creating a complete edge, which leads to a another complete edges and an in-progress edge, as before. Done.



Example (a note)

Compare with CYK:

We avoided creating these six CYK edges.



Earley at a glance

1. Insert edge (0,0, S$\rightarrow$. α) for all productions of S
2. Processing edges in turn. Split by edge type:
   - when dot is before a terminal d:
     - advance dot across d if d is next on input
     - next on input = to the right of the edge
   - when dot is before a non-terminal
     - predict new productions
   - when dot at end of production:
     - (we have reduced to a non-terminal T)
     - advance dot across the T in all edges expecting that non-terminal
     - ie, where dot is before T

- At the end, see edge (0,N, S $\rightarrow$ α .) exists.

Earley Algorithm: details

- Three main functions that do all the work:
  - **Predictor:** adds predictions into the chart
  - **Completer:** moves the dot to the right across a non-terminal when that non-terminal is found
  - **Scanner:** moves the dot across terminals found on the input

Predictor

- procedure Predictor((u, v, A $\rightarrow \alpha$. B β))
  for each B $\rightarrow \gamma$ do enqueue((???,v, B $\rightarrow$. γ))
  end

- Intuition:
  - new edges represent top-down expectations
- Applied when?
  - an edge **e** has a non-terminal **T** to the right of a dot
  - generates one new state for each production of **T**
- Edge placed where?
  - between same nodes as **e**

Completer

procedure Completer((u,v, $B \rightarrow \gamma$.))
for each (u', u, $A \rightarrow \alpha$. B β) do
enqueue((u', v, $A \rightarrow \alpha$ B . β))
end

- Intuition:
  - parser has reduced a substring to a non-terminal **B**
  - so must advance edges that were looking for **B** at this position in input. CYK reduction is a special case of this rule.
- Applied when:
  - dot has reached right end of rule.
  - new edge advances the dot over **B.**
- New edge spans the two edges (ie, connects u' and v)

Scanner

procedure Scanner($(u,v, A \rightarrow \alpha \; . \; d \; \beta)$)
    enqueue($(u, v+1, A \rightarrow \alpha \; d \; . \; \beta)$)
end

- Applied when:
  - advance dot over a terminal

Earley algorithm

- remove an edge from the queue
- apply one of the three actions, depending on edge type
- repeat while queue nonempty