Lecture 9

Syntax-Directed Translation

grammar disambiguation, Earley parser, syntax-directed translation

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!
CS164: Introduction to Programming Languages and Compilers, Spring 2012
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

Refresh CYK parser

- builds the parse bottom up

Grammar disambiguation

- select desired parse trees without rewriting the grammar

Earley parser

- solves CYK's inefficiency

Syntax-directed translation

- it's a rewrite ("evaluation") of the parse tree

Grammars, derivations, parse trees

Example grammar

DECL --> TYPE VARLIST $\boxed{;}$

TYPE --> int | float

VARLIST --> id | VARLIST , id

Example string

int id , id ;

Derivation of the string

DECL --> TYPE VARLIST ;

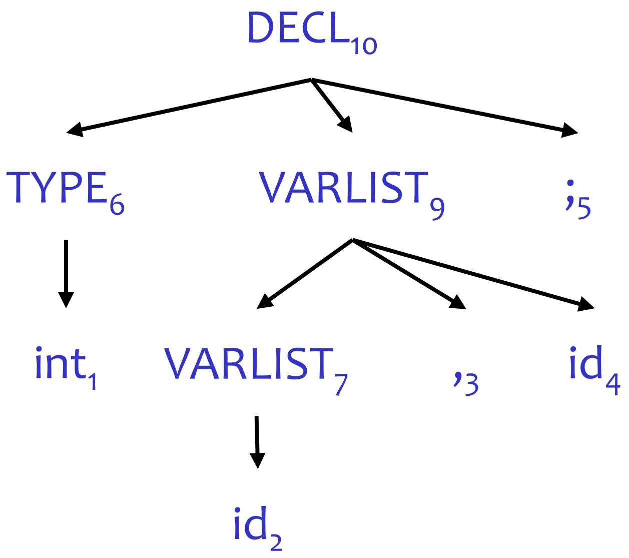
--> int VARLIST ;

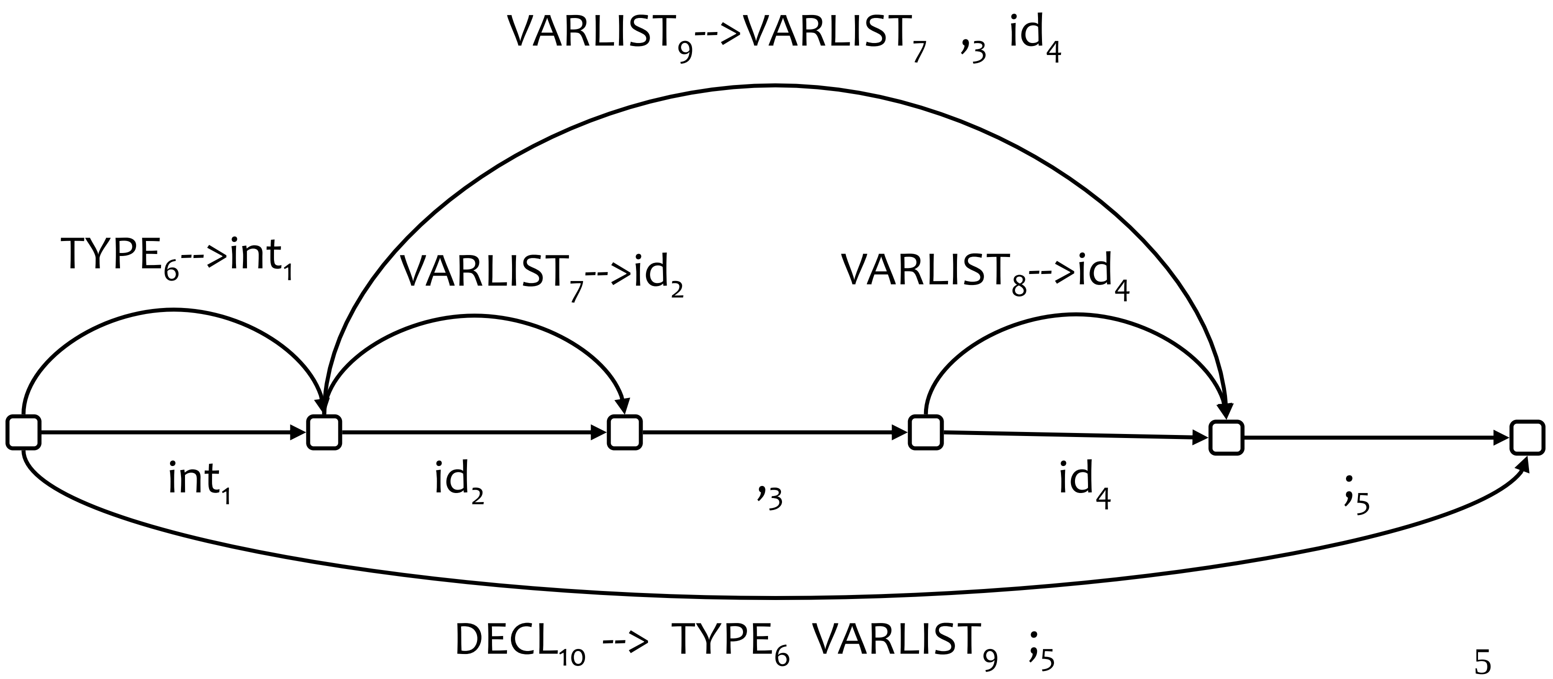
--> … -->

--> int id , id ;

$$\begin{tikzcd}
 & & \text{DECL}_{10} \arrow[dll] \arrow[d] \arrow[drr] & & \\
\text{TYPE}_6 \arrow[d] & & \text{VARLIST}_9 \arrow[dl] \arrow[dr] \arrow[drr] & & ;_5 \\
\text{int}_1 & \text{VARLIST}_7 \arrow[d] & & ,_3 & \text{id}_4 \\
 & \text{id}_2 & & &
\end{tikzcd}$$

CYK execution





Key invariant

Edge ($\underline{i}$,$\underline{j}$,$\underline{T}$) 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

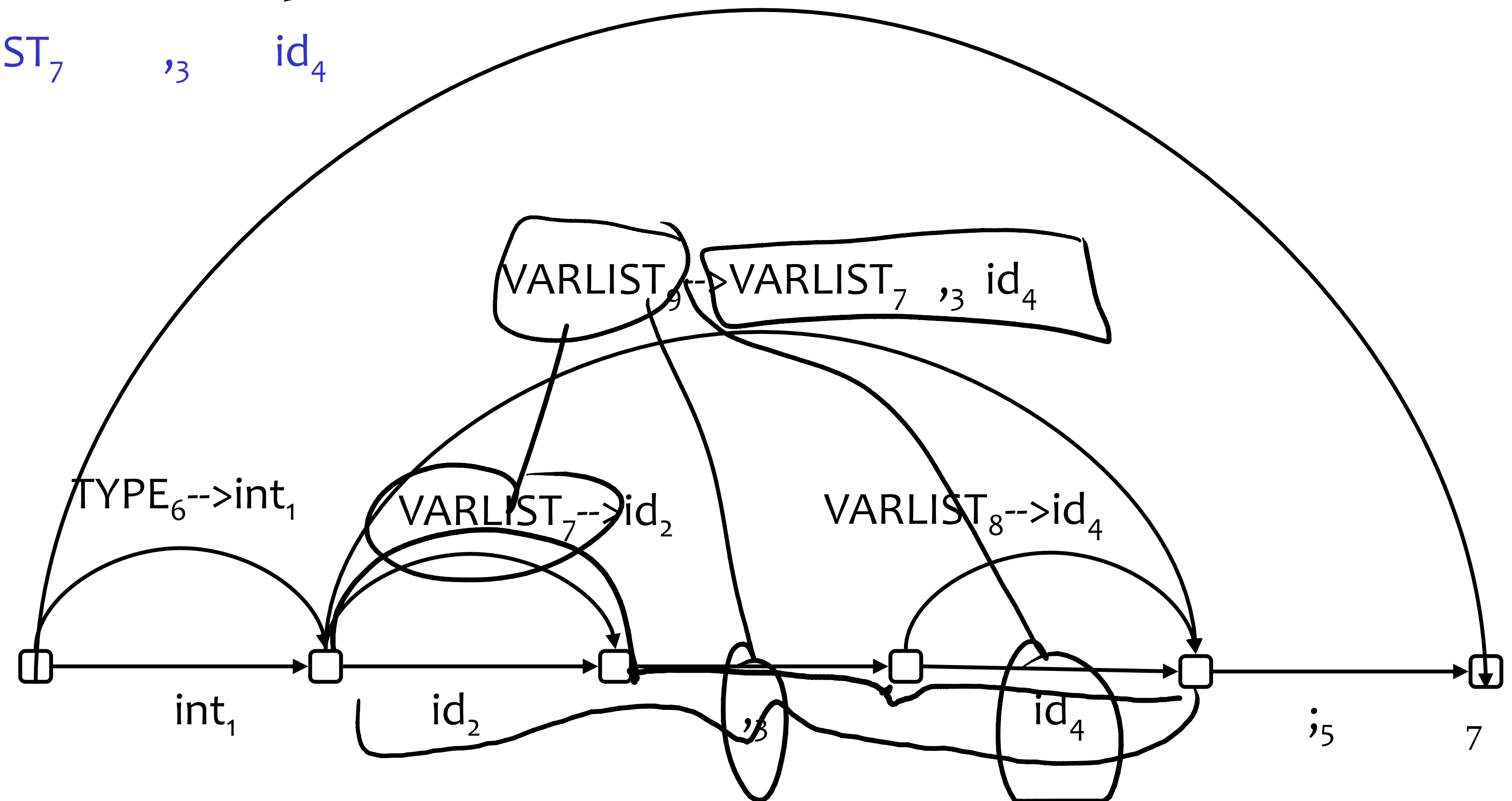
Corollary:

- 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

$$\begin{tikzcd}
 & DECL_{10} \arrow[dl] \arrow[dr] \arrow[drrr] & & & \\
TYPE_6 \arrow[d] & & VARLIST_9 \arrow[dl] \arrow[dr] \arrow[drr] & & ;_5 \\
int_1 & VARLIST_7 \arrow[d] & & ,_3 & id_4 \\
 & id_2 & & &
\end{tikzcd}$$

$DECL_{10} \text{ --> } TYPE_6 \; VARLIST_9 \; ;_5$



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 $\boxed{A \rightarrow B\ C,}$ when you find $\underline{\text{adjacent}}$ $\underline{B}$ $\underline{C}$ in the CYK graph, reduce $\underline{B\ C}$ to $\underline{A}$

See the algorithm in Lecture 8

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:

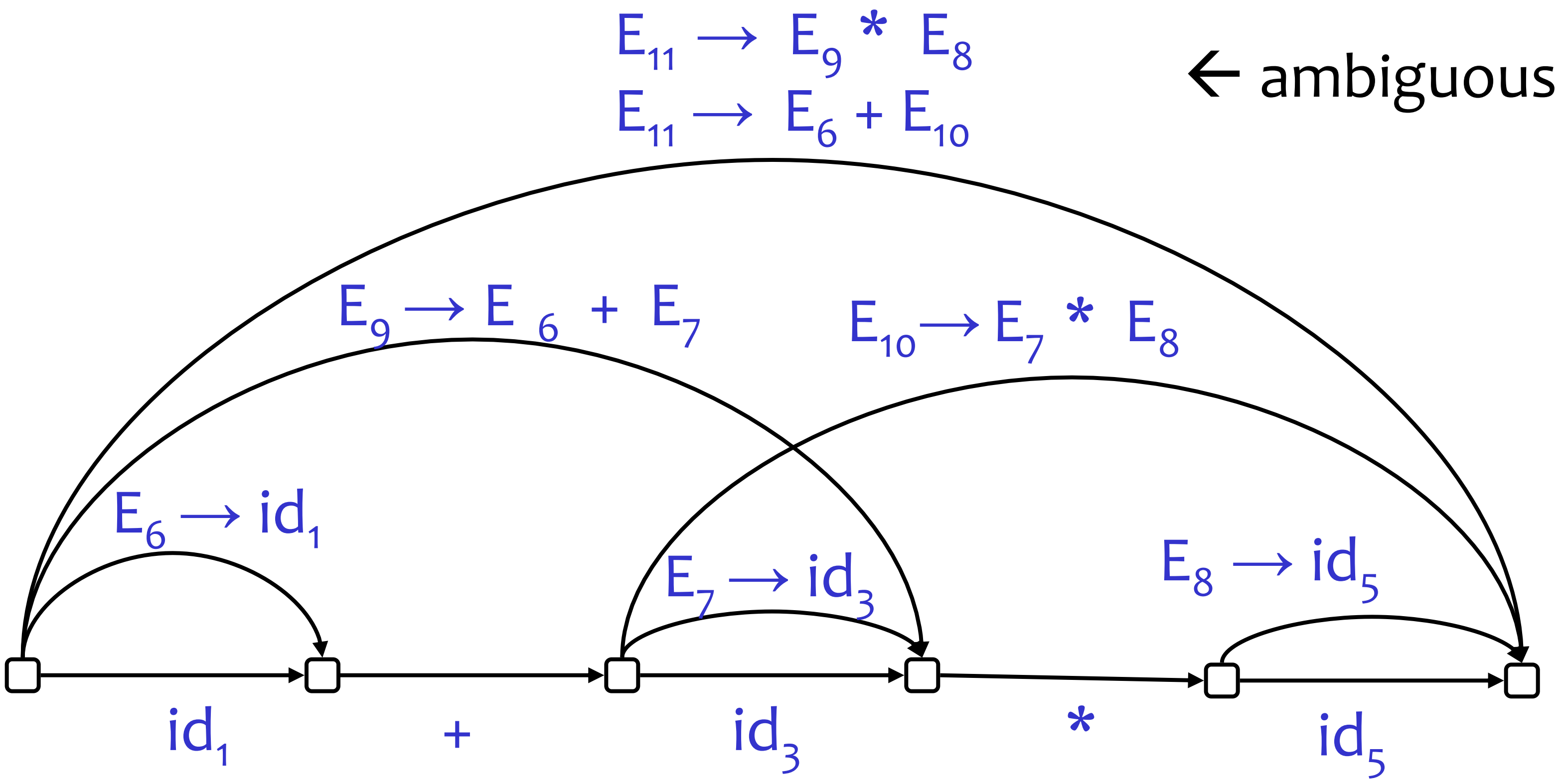
- the graph implementation may be too slow
- instead, store solutions to subproblems in a 2D array
    - solutions[i,j] stores a list of labels of all edges from i to j

Removing Ambiguity in the Grammar

How many parse trees are here?

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

input: id+id*id



$E_{11} \rightarrow E_9 * E_8$
$E_{11} \rightarrow E_6 + E_{10}$
← ambiguous
$E_9 \rightarrow E_6 + E_7$
$E_{10} \rightarrow E_7 * E_8$
$E_6 \rightarrow id_1$
$E_7 \rightarrow id_3$
$E_8 \rightarrow id_5$
id_1
+
id_3
*
id_5

PA5 warning: “Nested ambiguity”

Work out the CYK graph for this input: id+id*id+id.

Notice there are multiple “ambiguous” edges

- ie, edges inserted due to multiple productions
- hence there is exponential number of parse trees
- even though we have polynomial number of edges

The point:

don’t worry about exponential number of trees

We still need to select the desired one, of course

CYK on ambiguous grammar

same algorithm, but may yield multiple parse trees

- because an edge may be reduced (ie, inserted into the graph) using to multiple productions

we need to chose the desired parse tree

- we'll do so without rewriting the grammar

example grammar

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

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)

Ambiguity (Cont.)

Ambiguity is **bad**

- Leaves meaning of some programs ill-defined

Ambiguity is **common** in programming languages

- Arithmetic expressions
- IF-THEN-ELSE

Ambiguity: Example

Grammar

E → E + E | E * E | (E) | int

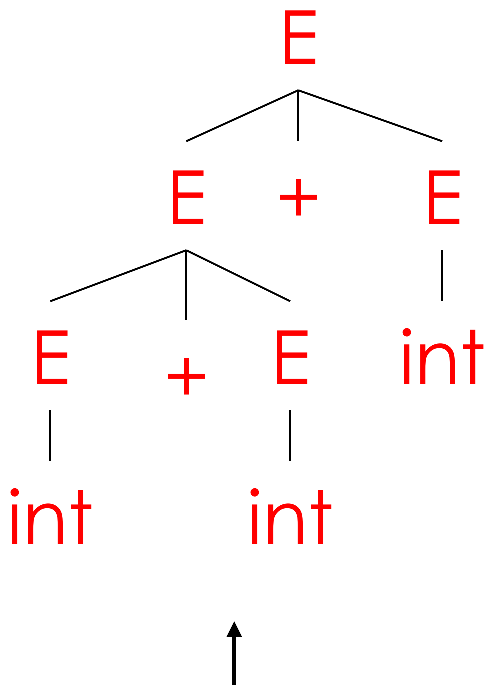
Strings

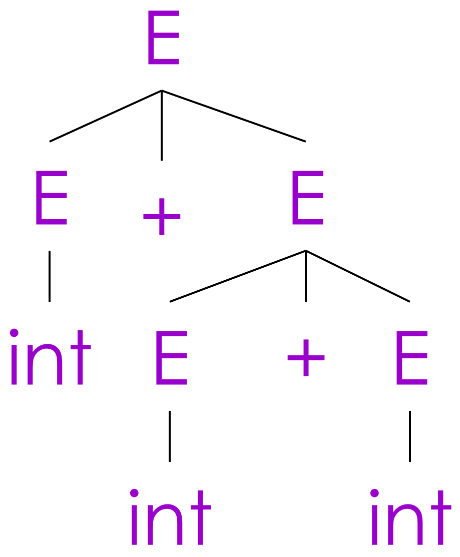
int + int + int

int * int + int

Ambiguity. Example

This string has two parse trees

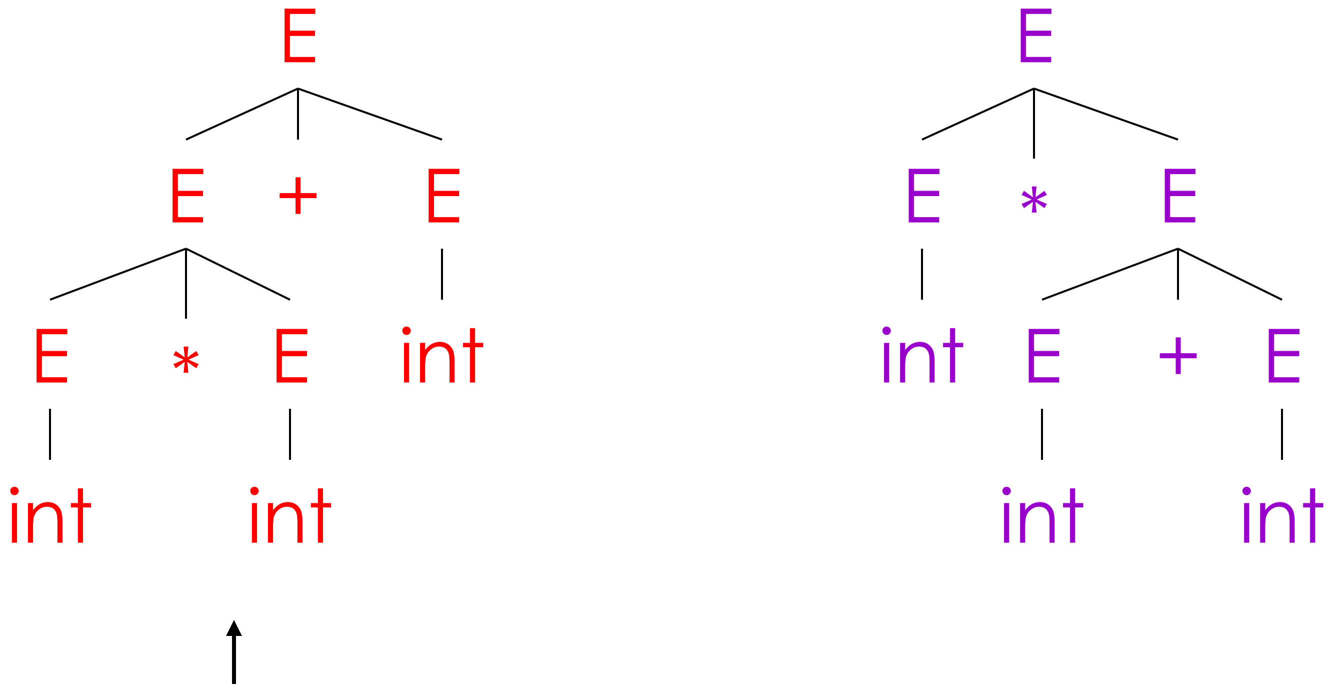




+ is left-associative

Ambiguity. Example

This string has two parse trees



* has higher precedence than +

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) Declare to the parser which productions to prefer

   works on most but not all ambiguities

2) Rewrite the grammar

   a general approach, but manual rewrite needed

   we saw an example in Lecture 8

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

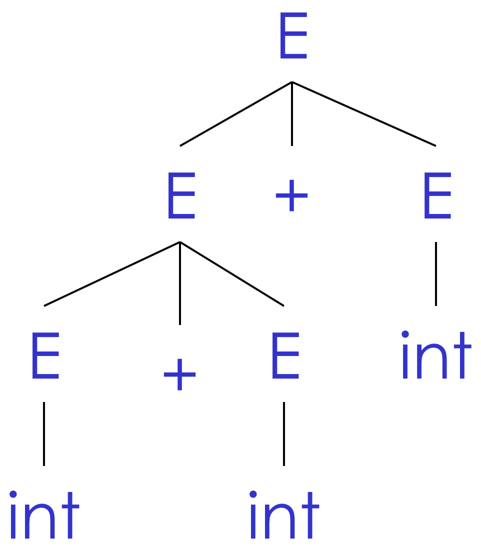
you will implement those in PA5

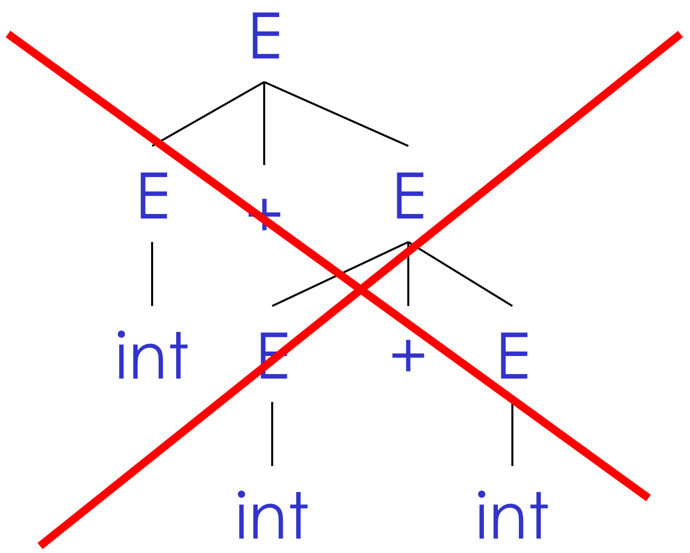
Examples …

Associativity Declarations

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

Ambiguous: two parse trees of int + int + int



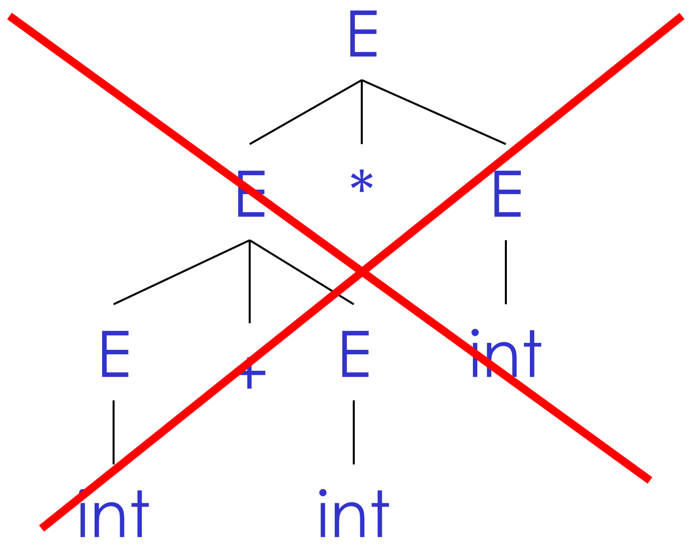


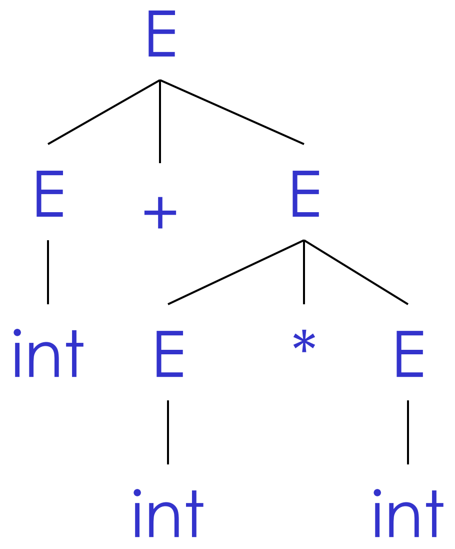
Left-associativity declaration: $\boxed{\text{%left} \; +}$

Precedence Declarations

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

- And the string int + int * int





Precedence declarations:

%left + -

%left * /

Ambiguity declarations

These are the two common forms of ambiguity

- precedence: * higher precedence than +
- associativity: + associates from to the left

Declarations for these two common cases

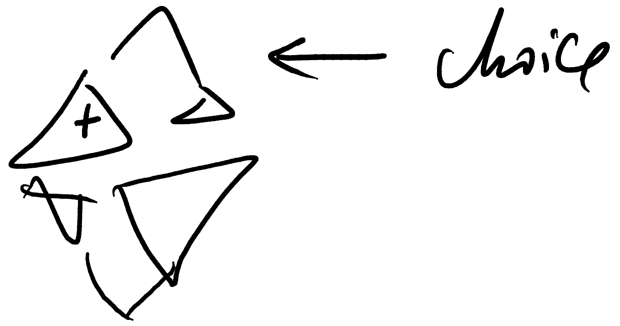
%left + -	+ and – have lower precedence than * and /
%left * /	these operators are left associative

Implementing disambiguity declarations

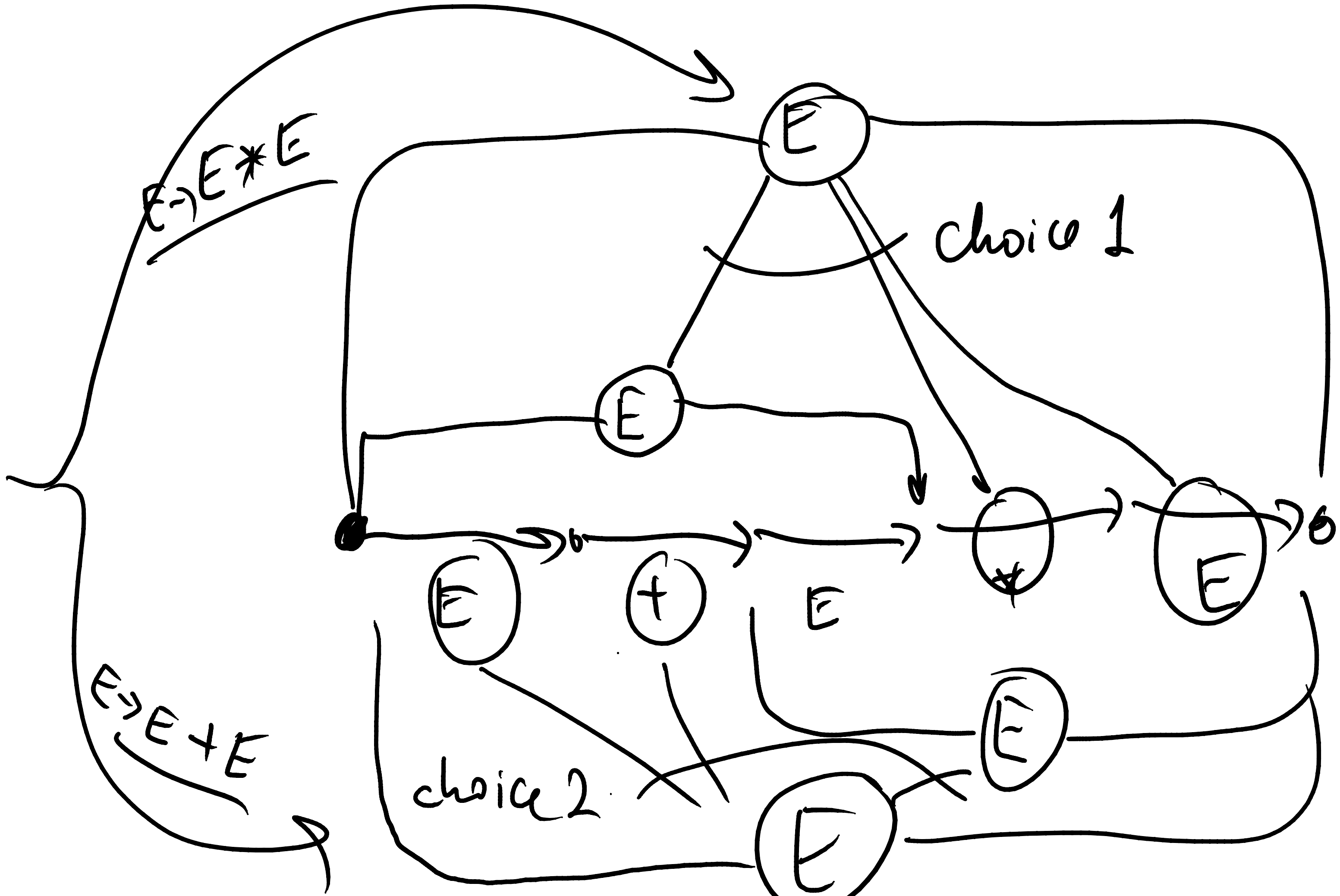
To disambiguate, we need to answer these questions:

Assume we reduced the input to E+E*E.
Now do we want parse tree (E+E)*E or E+(E*E)?

Similarly, given $\underline{E+E+E}$,
do we want parse tree $\underline{(E+E)+E}$ or $\underline{E+(E+E)}$?



Example



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

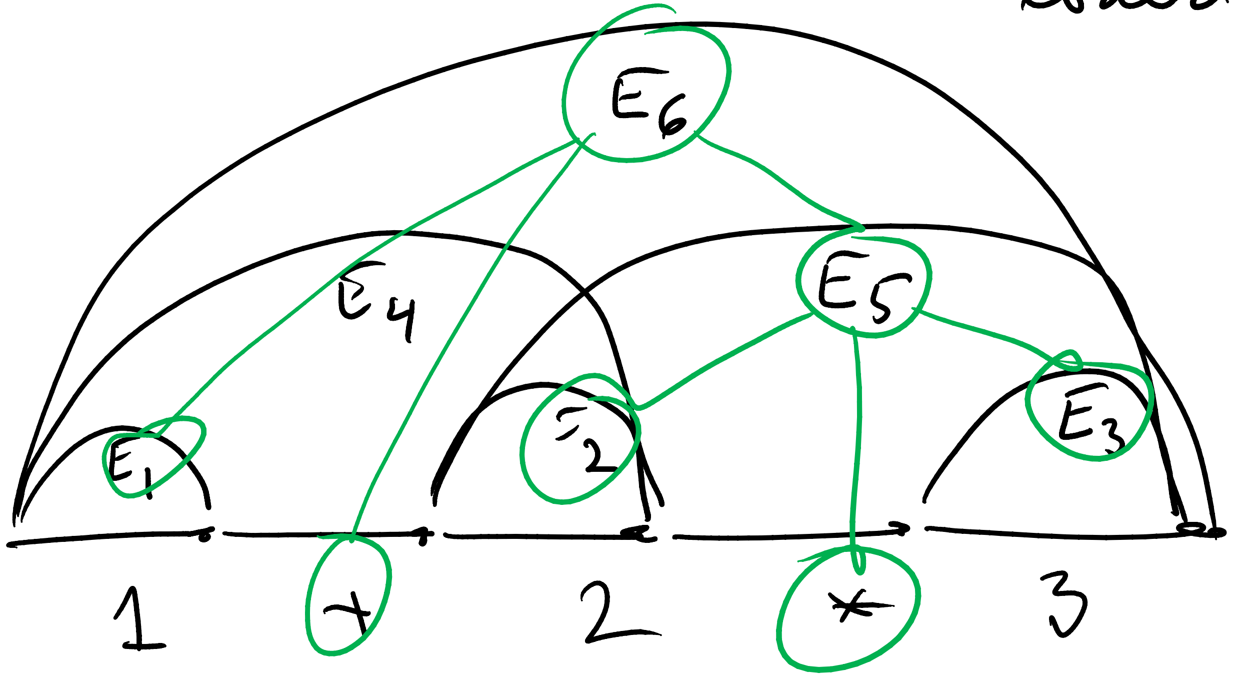
- when multiple productions compete for being a child in the parse tree, select the one with largest left subtree

Precedence

ambiguity what are children of E_6

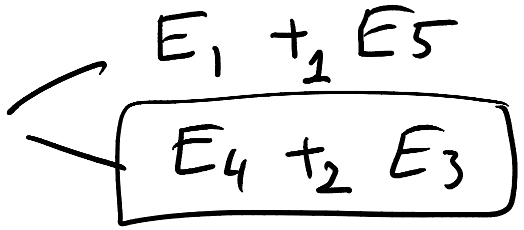
choices
- $E_4 * E_3$
- $\boxed{E_1 + E5}$

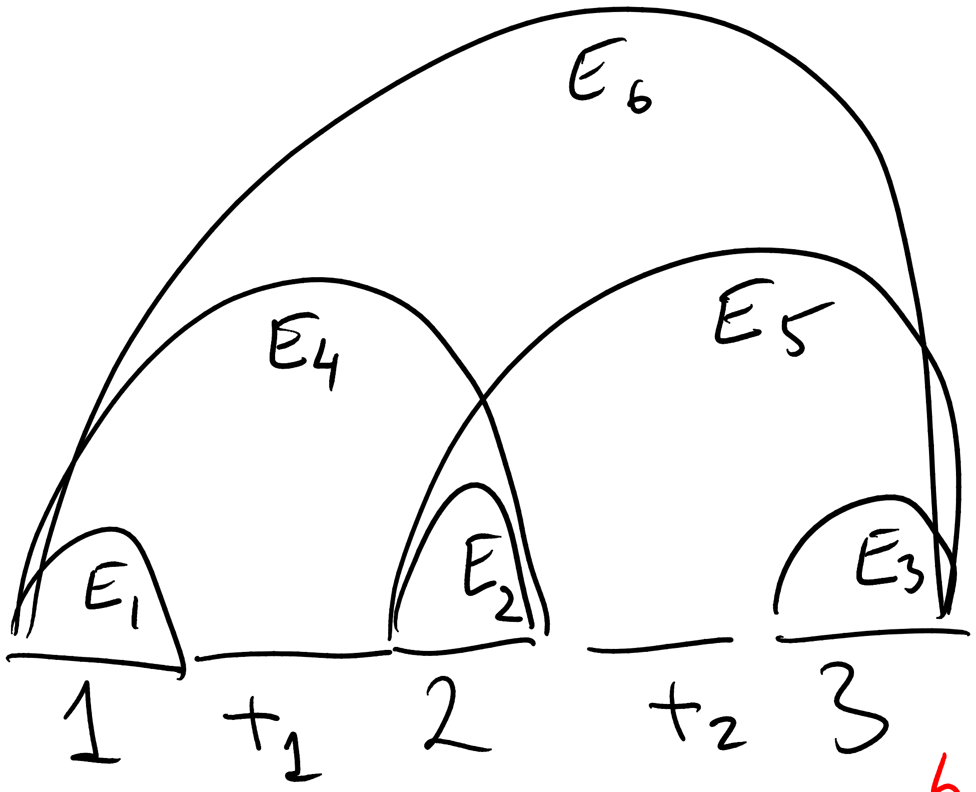
plus to be evaled after *



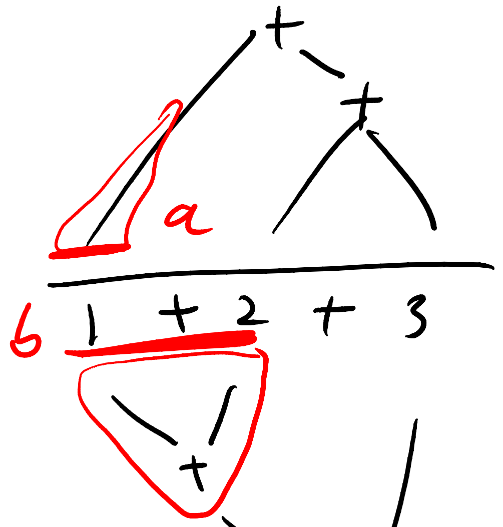
Associatiivity

Same ambiguity





%left '+'



b spans more than a

Where is ambiguity manifested in CYK?

```
for  i=0,N-1 do  enqueue( (i,i+1,input[i]) )     -- create terminal edges
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)
     for each rule T→AB do
       if edge (i,k,T) does not exists then
           add (i,k,T) to graph
           enqueue( (i,k,T) )
       else   -- Edge (i,k,T) already exists, hence potential ambiguity:
              -- Edges (i,j,A)(j,k,B) may be another way to reduce to (i,k,T).
              -- That is, they may be the desired child of (i,k,T) in the parse tree.
end while
```

(Find the corresponding points in the Earley parser)

More ambiguity declarations

%left, %right declare precedence and associativity

- these apply only for binary operators
- and hence they do not resolve all ambiguities

Consider the Dangling Else Problem

$E \to$ if E then E | if E then E else E

On this input, two parse trees arise

- input: if e1 then if e2 then e3 else e4
- parse tree 1: if e1 then {if e2 then e3 else e4}
- parse tree 2: if e1 then {if e2 then e3} else e4

Which tree do we want?

%dprec: another declaration

Another disambiguating declaration (see bison)

```
E →  if E then E                % dprec 1
   | if E then E  else  E       % dprec 2
   | 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
```

Need more information?

See handouts for projects PA4 and PA5
as well as the starter kit for these projects

Grammar Rewriting

Rewriting

Rewrite the grammar into a unambiguous grammar

- While describing the same language and eliminating undesirable prase trees

Example: Rewrite

E ♎ E + E | E * E | (E) | int

into

E ♎ E + T | T

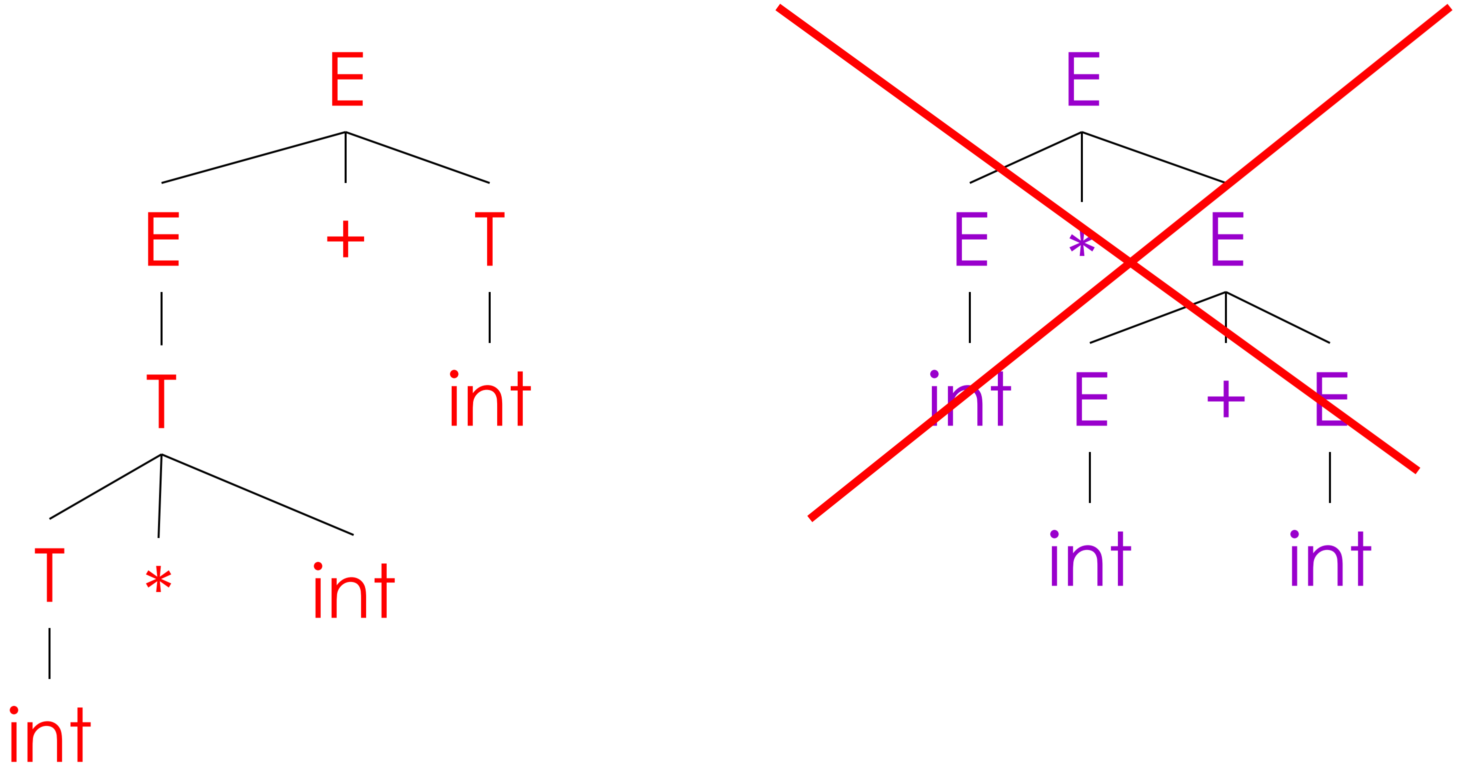
T ♎ T * int | int | (E)

Draw a few parse trees and you will see that new grammar

- enforces precedence of * over +
- enforces left-associativity of + and *

Parse tree with the new grammar

The int * int + int has ony one parse tree now



note that new nonterminals have been introduced

Rewriting the grammar: what's the trick?

Trick 1: Fixing precedence (* computed before +)

$E \longrightarrow E + E \mid E * E \mid id$

In the parse tree for id + id * id, we want id*id to be subtree of E+E.

How to accomplish that by rewriting?

Create a new nonterminal (T)

- make it derive id*id, …
- ensure T's trees are nested in E's of E+E

New grammar:

Rewriting the grammar: what's the trick? (part 2)

Trick 2: Fixing associativity (+, *, associate to the left)

$E \rightarrow E + E \mid T$

$T \rightarrow T * T \mid id$

In the parse tree for id + id + id, we want the left id+id to be subtree of the right E+id. Same for id*id*id.

Use left recursion

- it will ensure that +, * associate to the left

New grammar (a simple change):

$E \rightarrow E + E \quad \mid \quad T$

$T \rightarrow T * T \quad \mid \quad id$

Ambiguity: The Dangling Else

Consider the ambiguous grammar

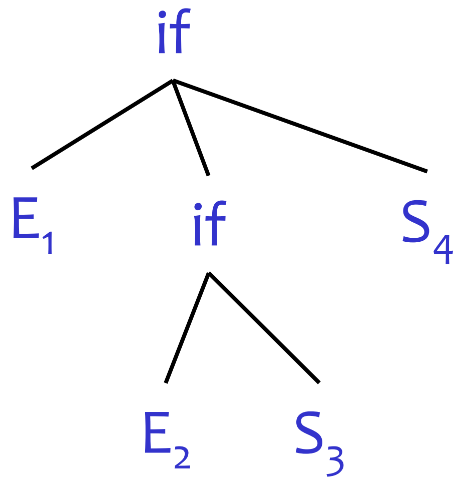
$$
\begin{aligned}
S \rightarrow\ & \text{if } E \text{ then } S \\
|\ & \text{if } E \text{ then } S \text{ else } S \\
|\ & \text{OTHER}
\end{aligned}
$$

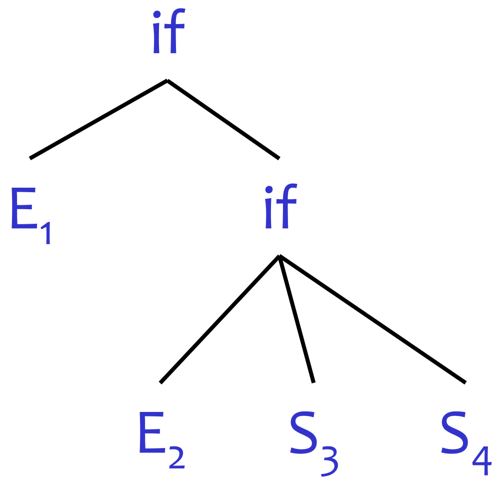
The Dangling Else: Example

- 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 */

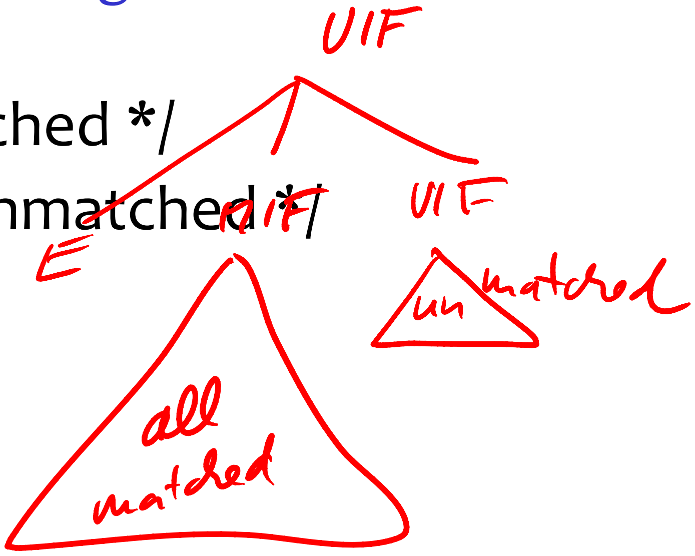
  | UIF /* some then are unmatched */

MIF → if E then MIF else MIF

  | OTHER

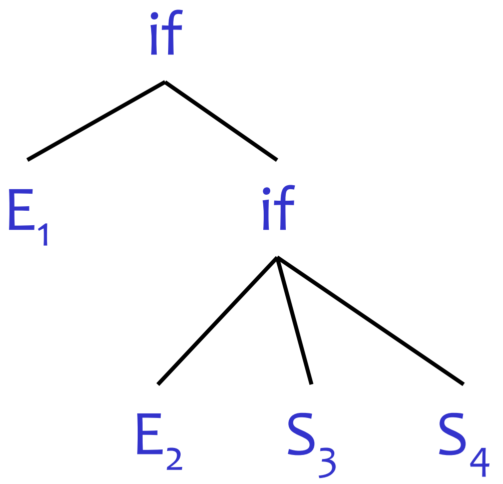
UIF → if E then S

  | if E then MIF else UIF

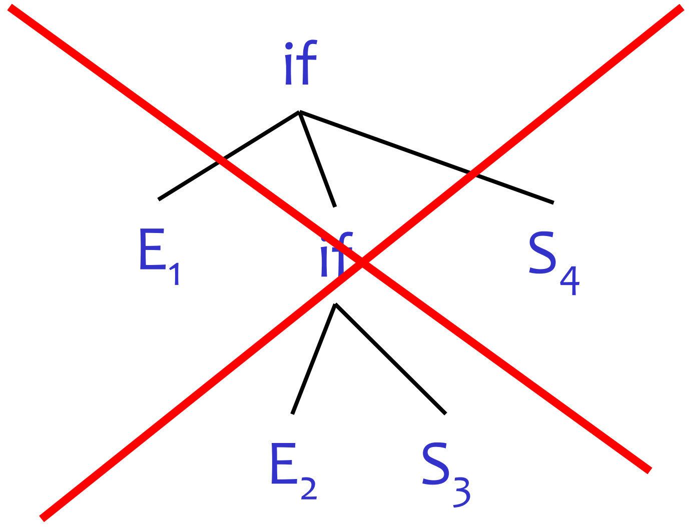


The Dangling Else: Example Revisited

- 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 ::= E+id | id

input: id+id+id

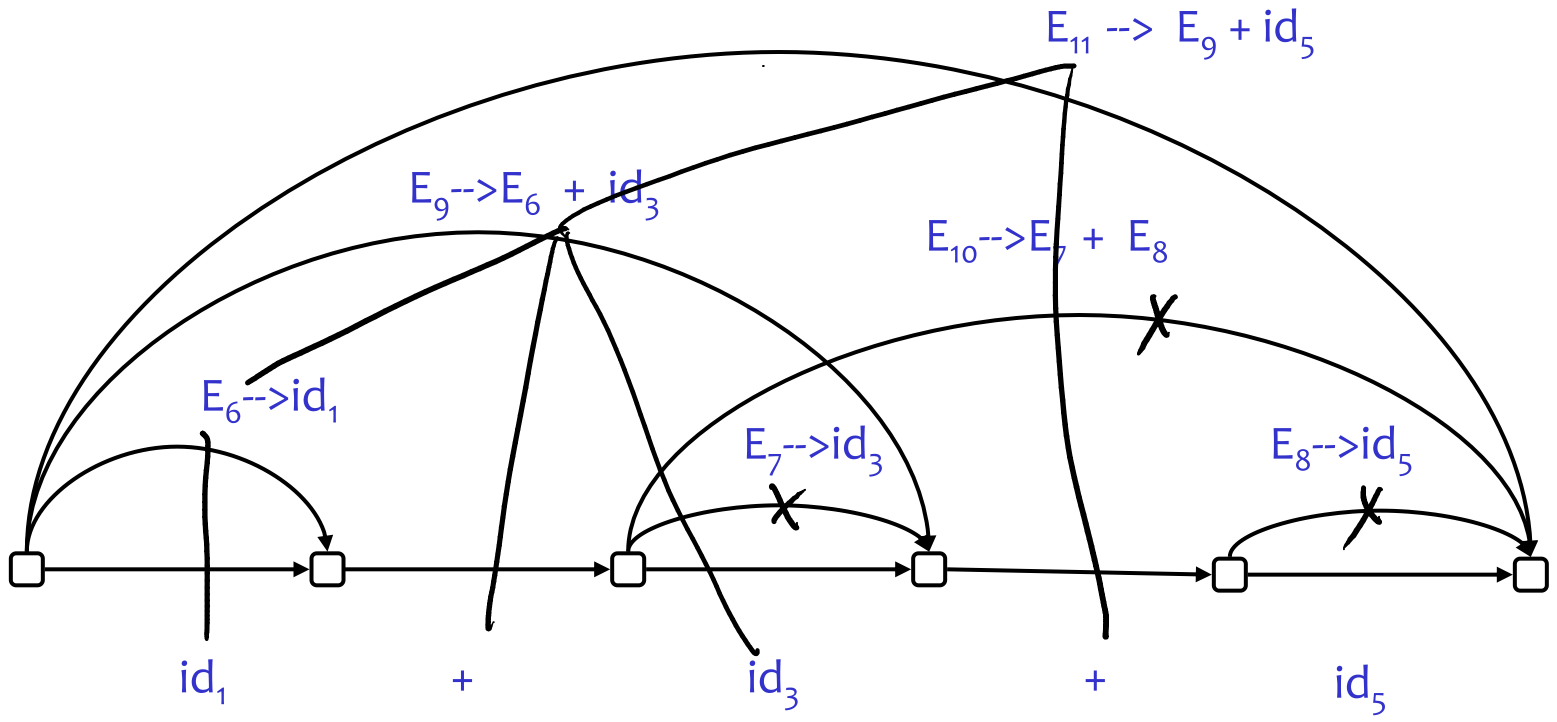
Can you spot the inefficiency?

This inefficiency is a difference between $\overset{\text{CYK}}{O(n^3)}$ and $\overset{\text{Earley}}{O(n^2)}$

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

Example

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



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

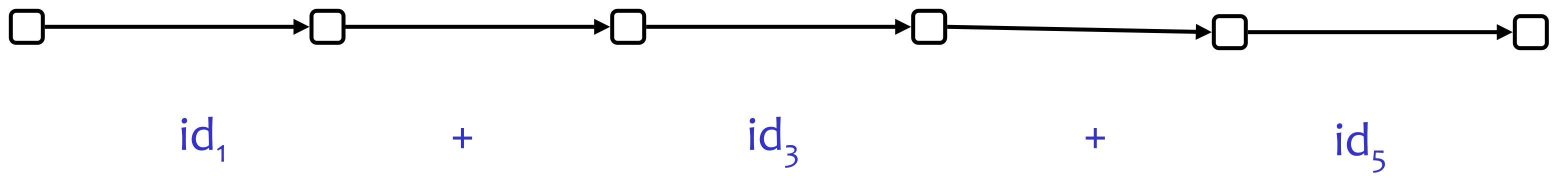
- decide whether to reduce based on **the input seen so far**
- after seeing more, we may still realize we built a useless tree

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

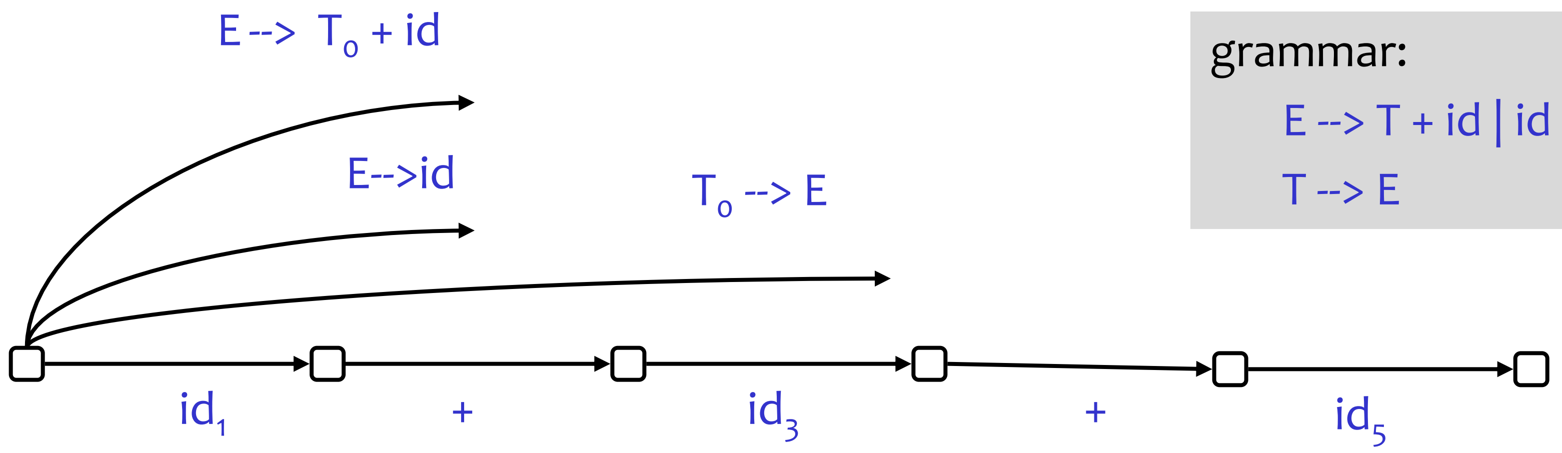


The intuition

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

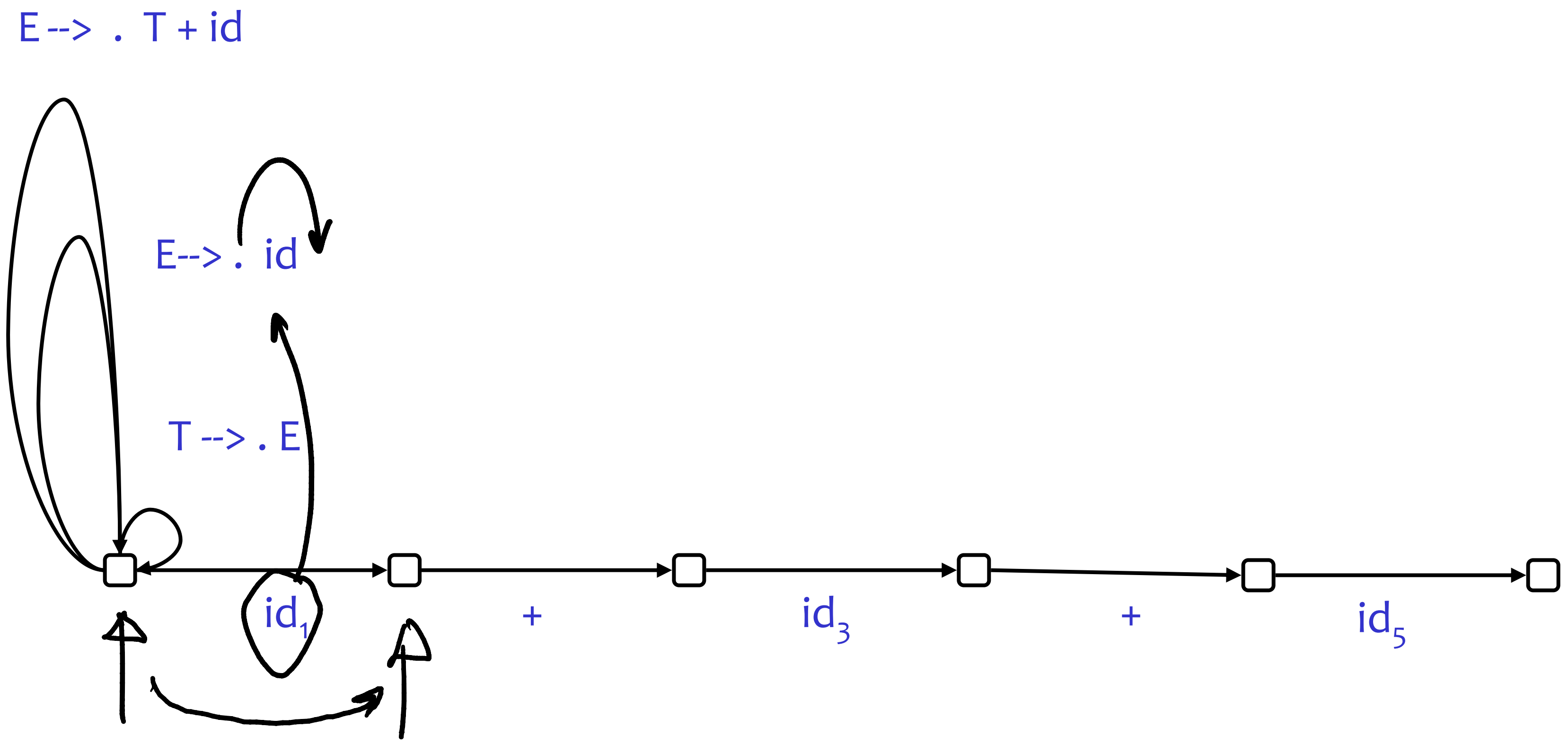
Example (1)

Initial predicted edges:

grammar:

E --> T + id | id

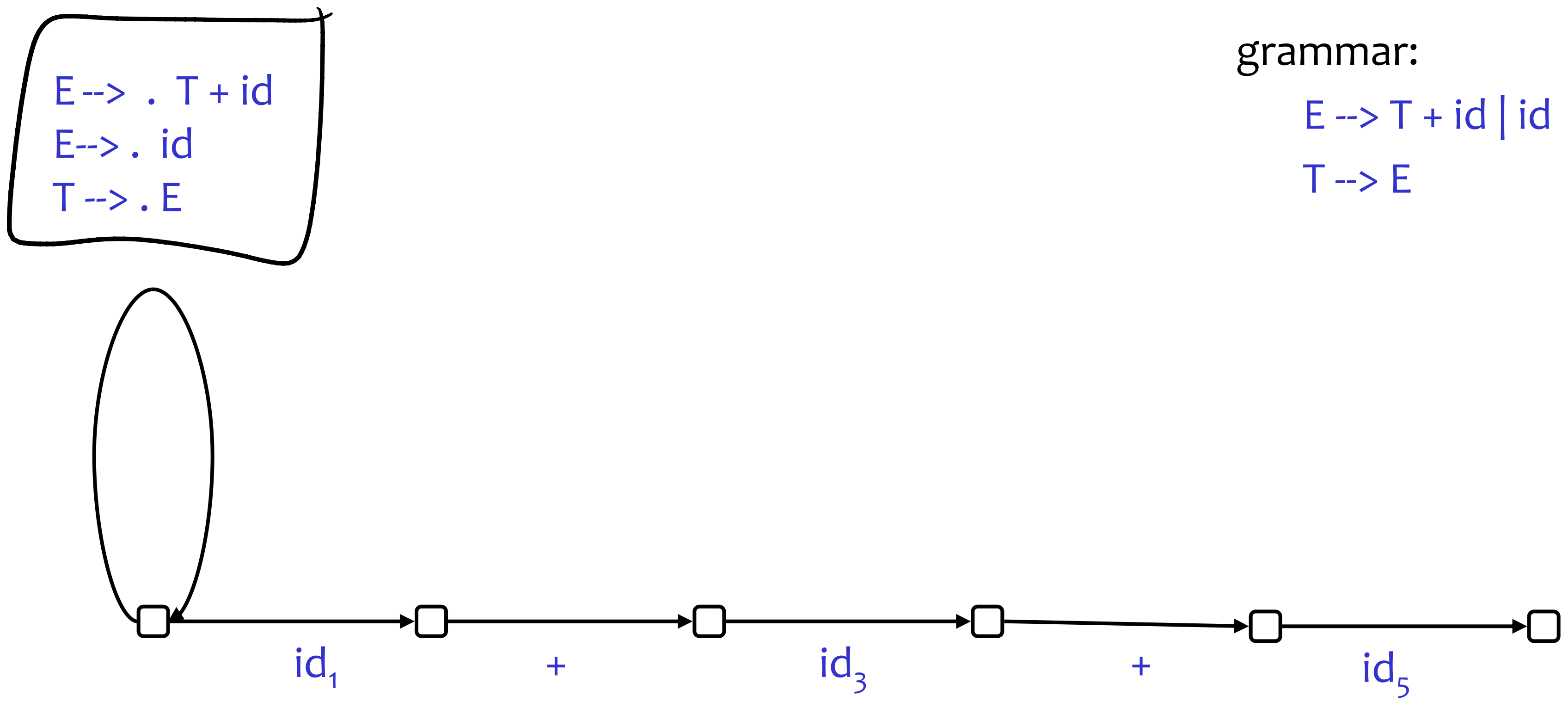
T --> E



Example (1.1)

Let's compress the visual representation:

these three edges → single edge with three labels



grammar:

E --> T + id | id

T --> E

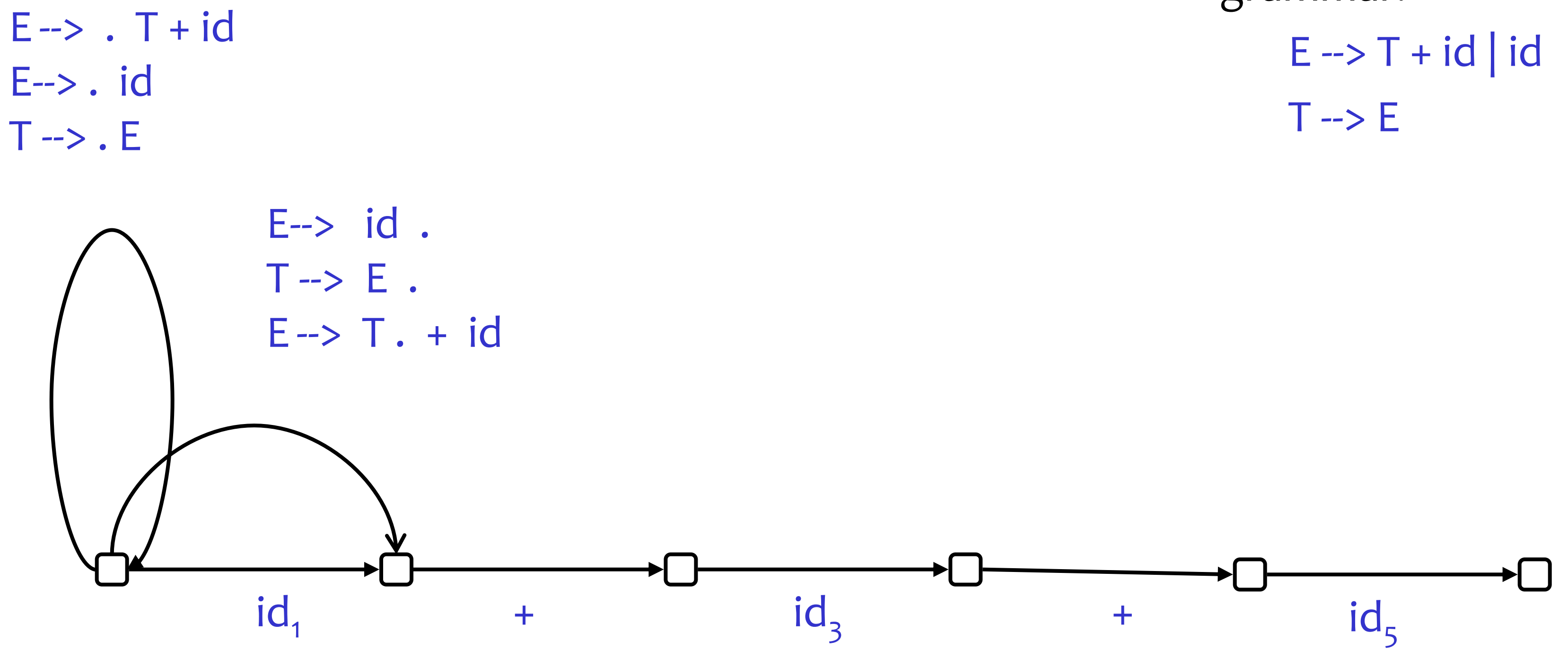
Example (2)

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

grammar:

E --> T + id | id

T --> E



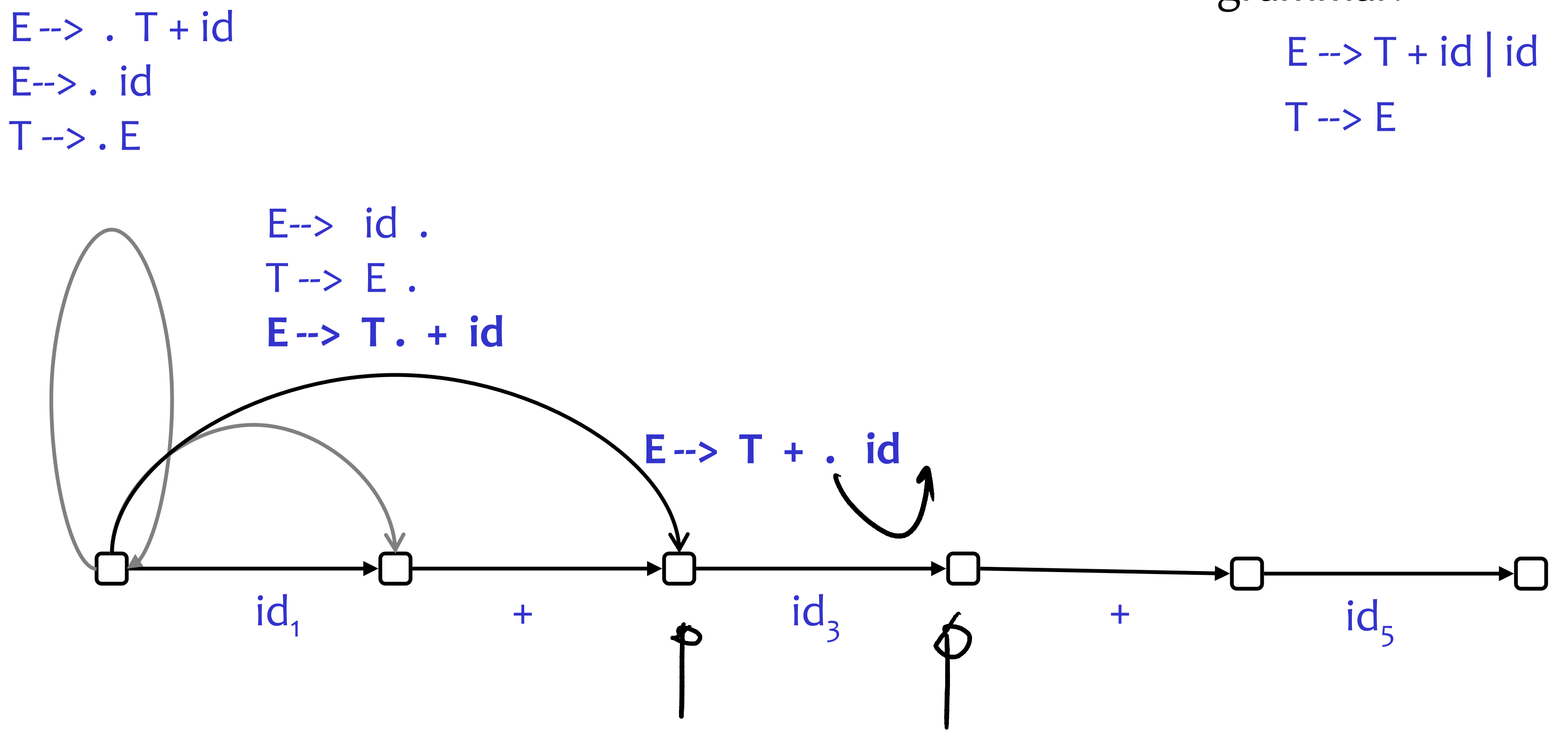
Example (3)

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

grammar:

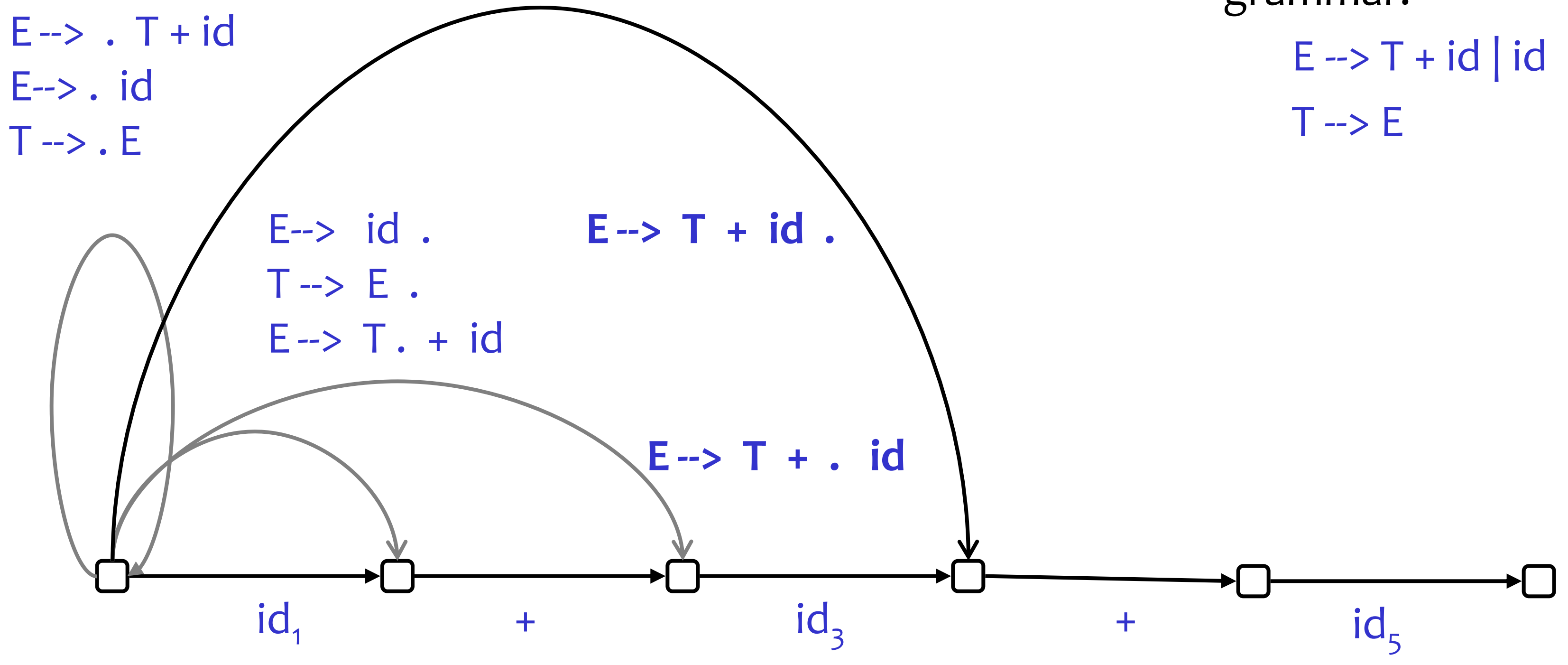
E --> T + id | id

T --> E



Example (4)

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



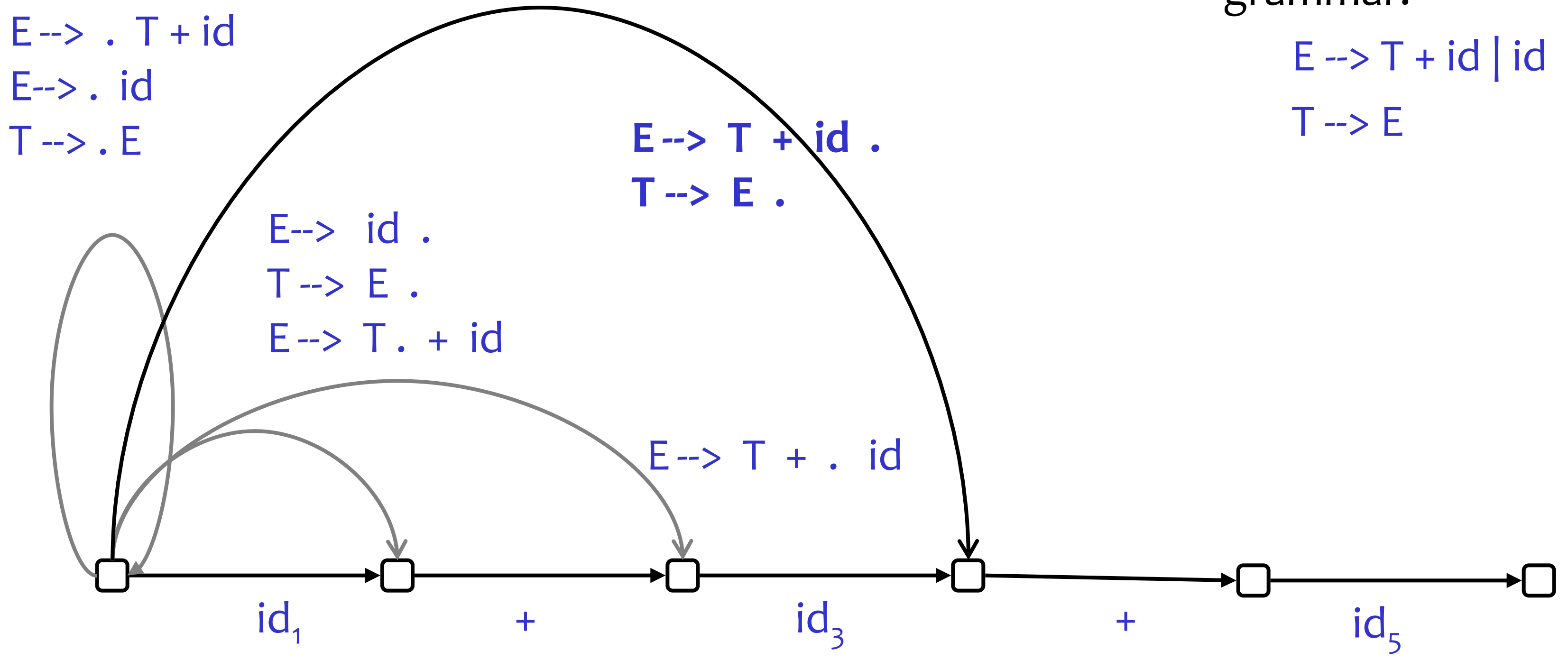
grammar:

E --> T + id | id

T --> E

Example (5)

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



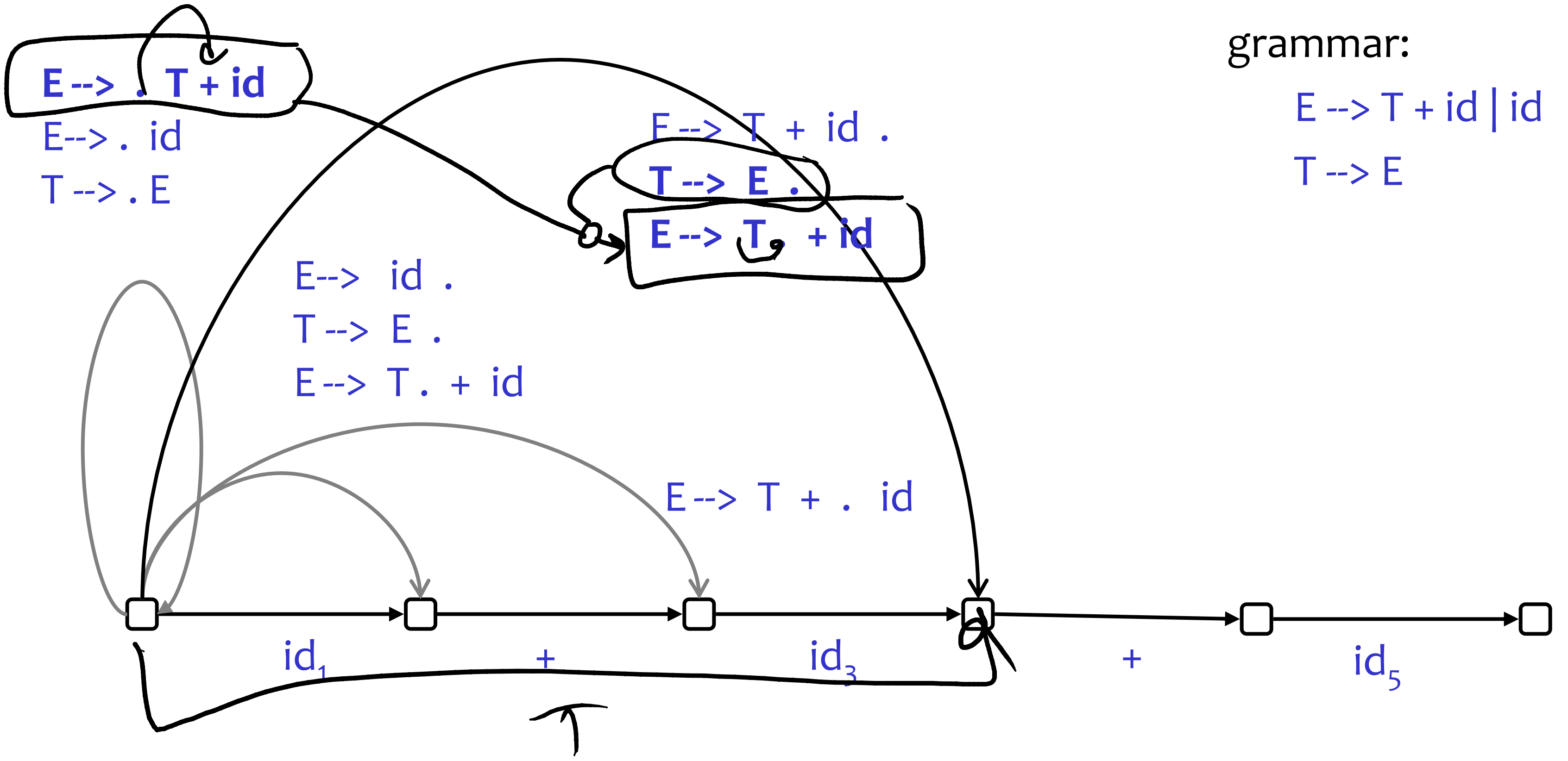
grammar:

E --> T + id | id

T --> E

Example (6)

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



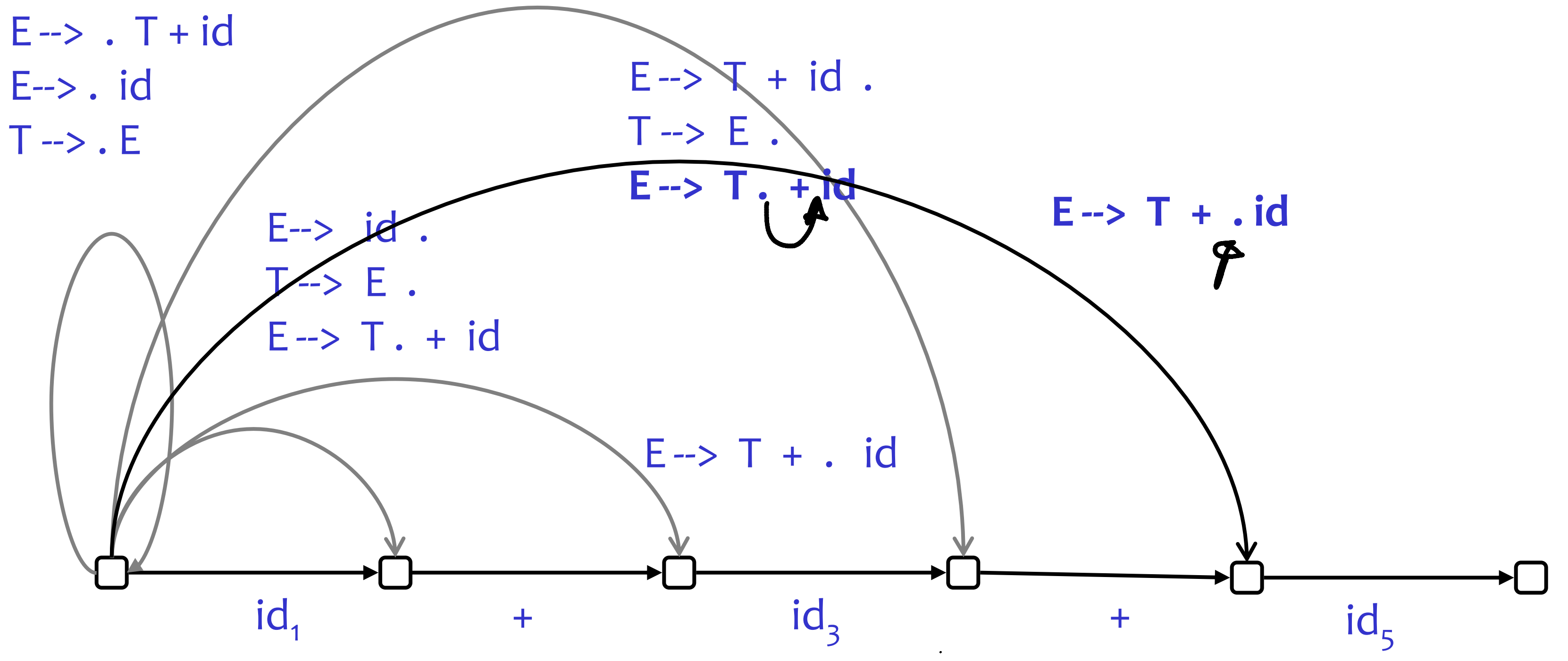
grammar:

E --> T + id | id

T --> E

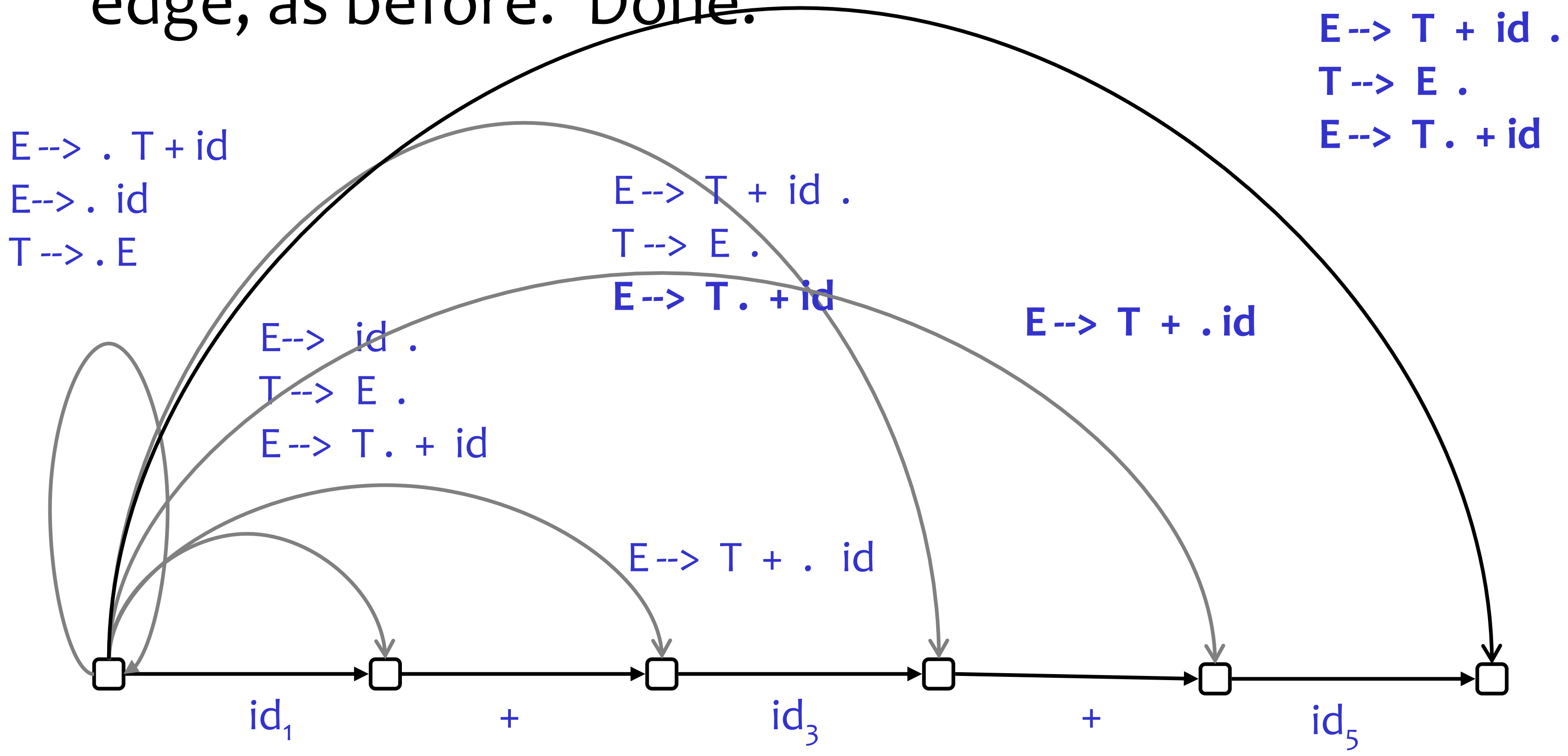
Example (7)

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



Example (8)

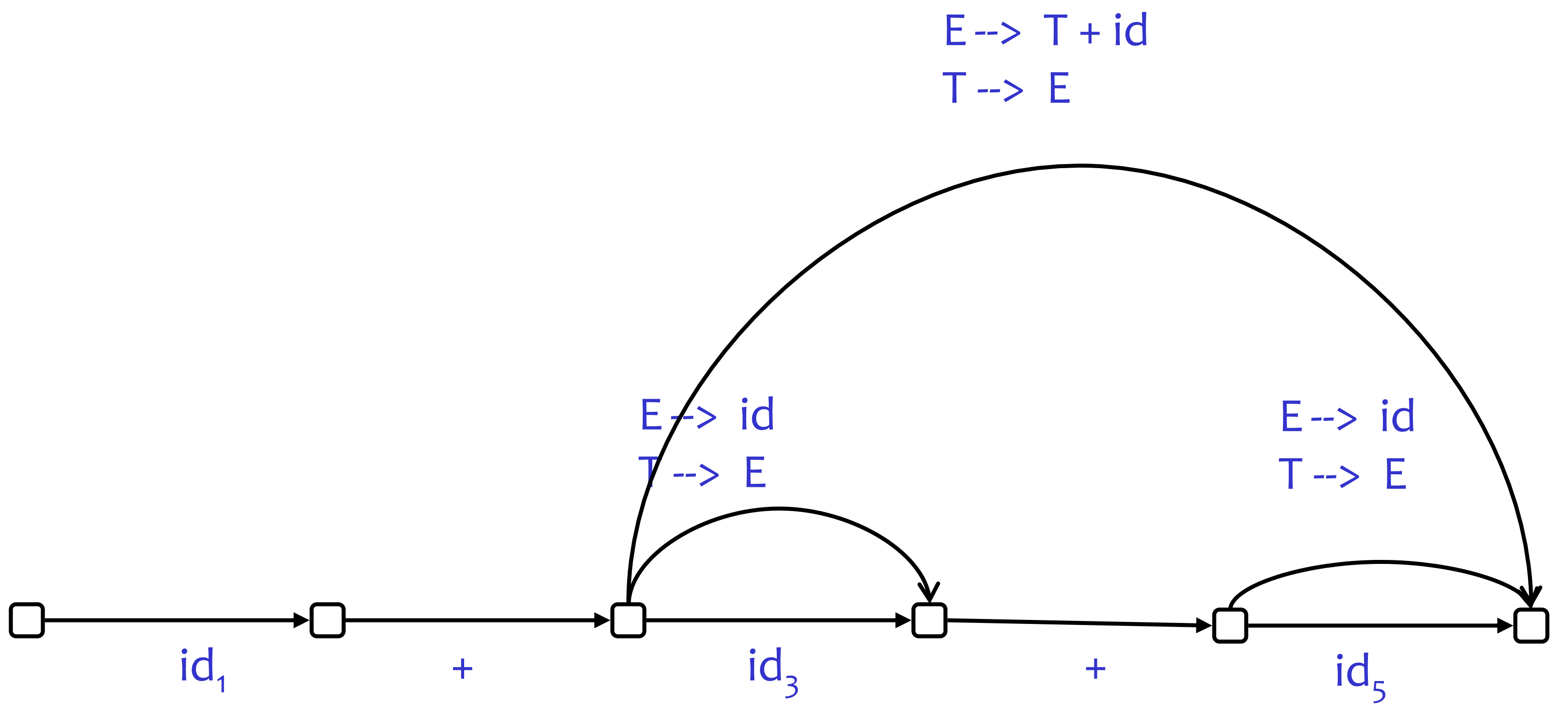
Advance again, creating a complete edge, which leads to a another complete edges and an in-progress edge, as before. Done.



Example (a note)

Compare with CYK:

We avoided creating these six CYK edges.



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.

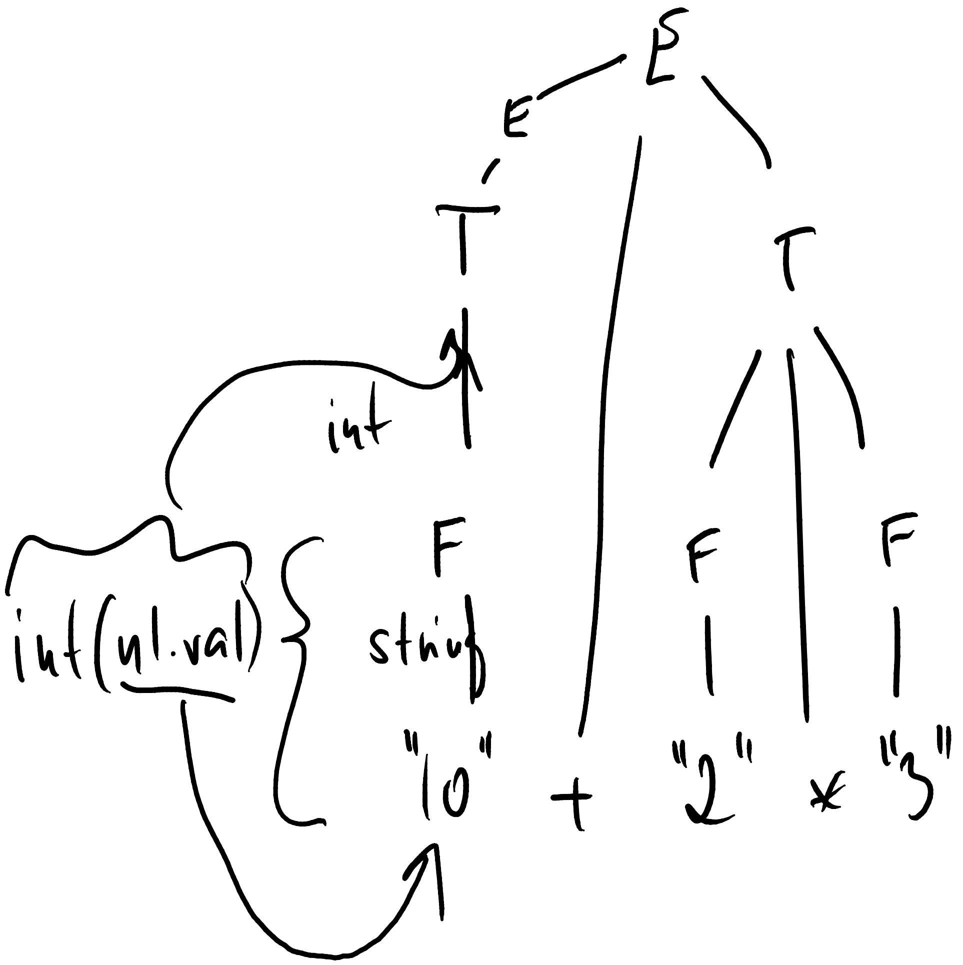
Syntax-directed translation

evaluate the parse (to produce a value, AST, …)

Example grammar in CS164

```
E -> E '+' T
   |  T
;
T -> T '*' F
   |  F
   ;
F -> /[0-9]+/
   | '(' E ')'
   ;
```

Build a parse tree for 10+2*3, and evaluate



Same SDT in the notation of the cs164 parser

Syntax-directed translation for evaluating an expression

```
%%
E -> E '+' T    %{ return n1.val + n3.val %}
  |  T          %{ return n1.val %}
  ;
T -> T '*' F    %{ return n1.val * n3.val }%
  |  F
  ;
F -> /[0-9]+/   %{ return int(n1.val) }%
  | '(' E ')'   %{ return n2.val }%
  ;
```

Build AST for a regular expression

```
%ignore /\n+/

%%

// A regular expression grammar in the 164 parser

R ->  'a'                   %{ return n1.val %}
  | R '*'                   %{ return ('*', n1.val) %}
  | R R                     %{ return ('.', n1.val, n2.val) %}
  | R '|' R                 %{ return ('|', n1.val, n3.val) %}
  | '(' R ')'               %{ return n2.val %}
  ;
```

Extra slides

Predictor

- procedure Predictor((u, v, A --> α . B β))

```
    for each B --> χ  do  enqueue( (???,v, B --> . χ ) )
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 --> $\chi$ . ) )
    for each (u', u, A --> $\alpha$ . B $\beta$) do
            enqueue( (u', v, A --> $\alpha$ B . $\beta$) )
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 --> $\alpha$ . d $\beta$)  )
    enqueue( (u, v+1, A --> $\alpha$ d . $\beta$) )
end
```

- Applied when:
  - advance dot over a terminal

The parse tree

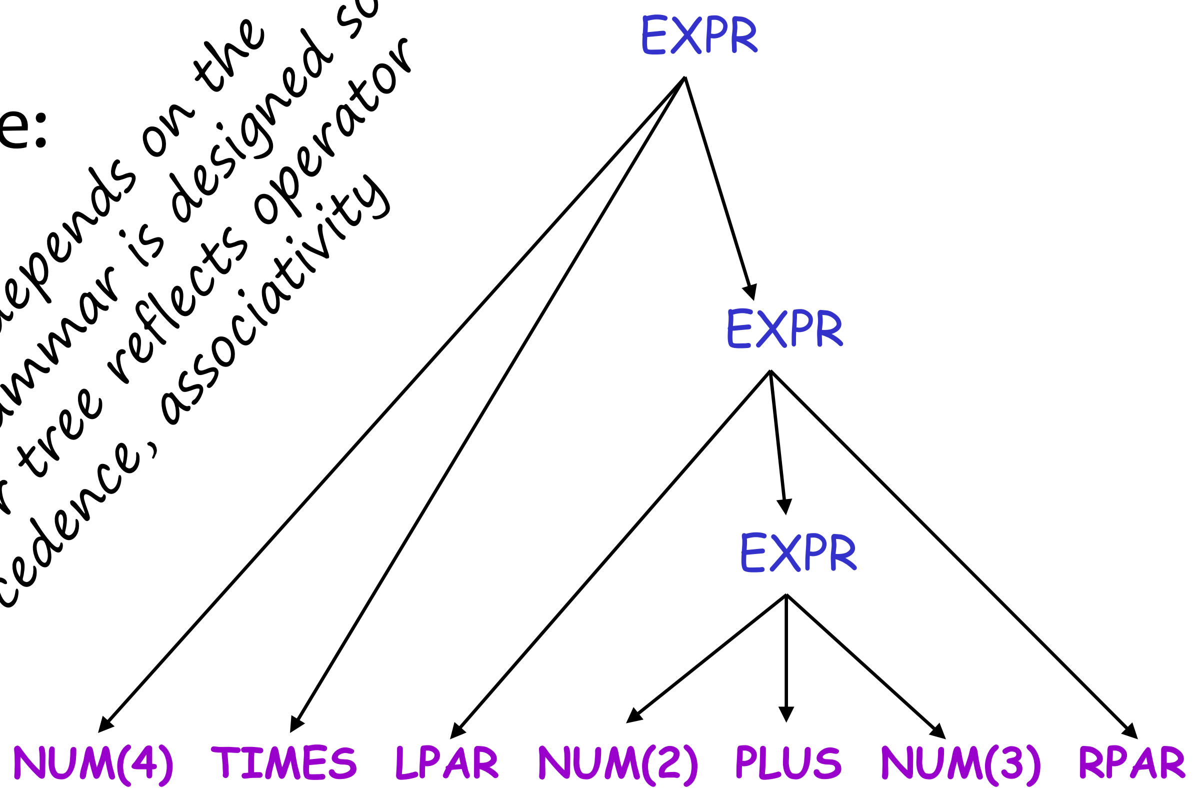
represents the tree structure in flat sequences

Parse tree example

Source: 4*(2+3)

Parser input: **NUM(4), TIMES, LPAR, NUM(2), PLUS, NUM(3), RPAR**

Parse tree:

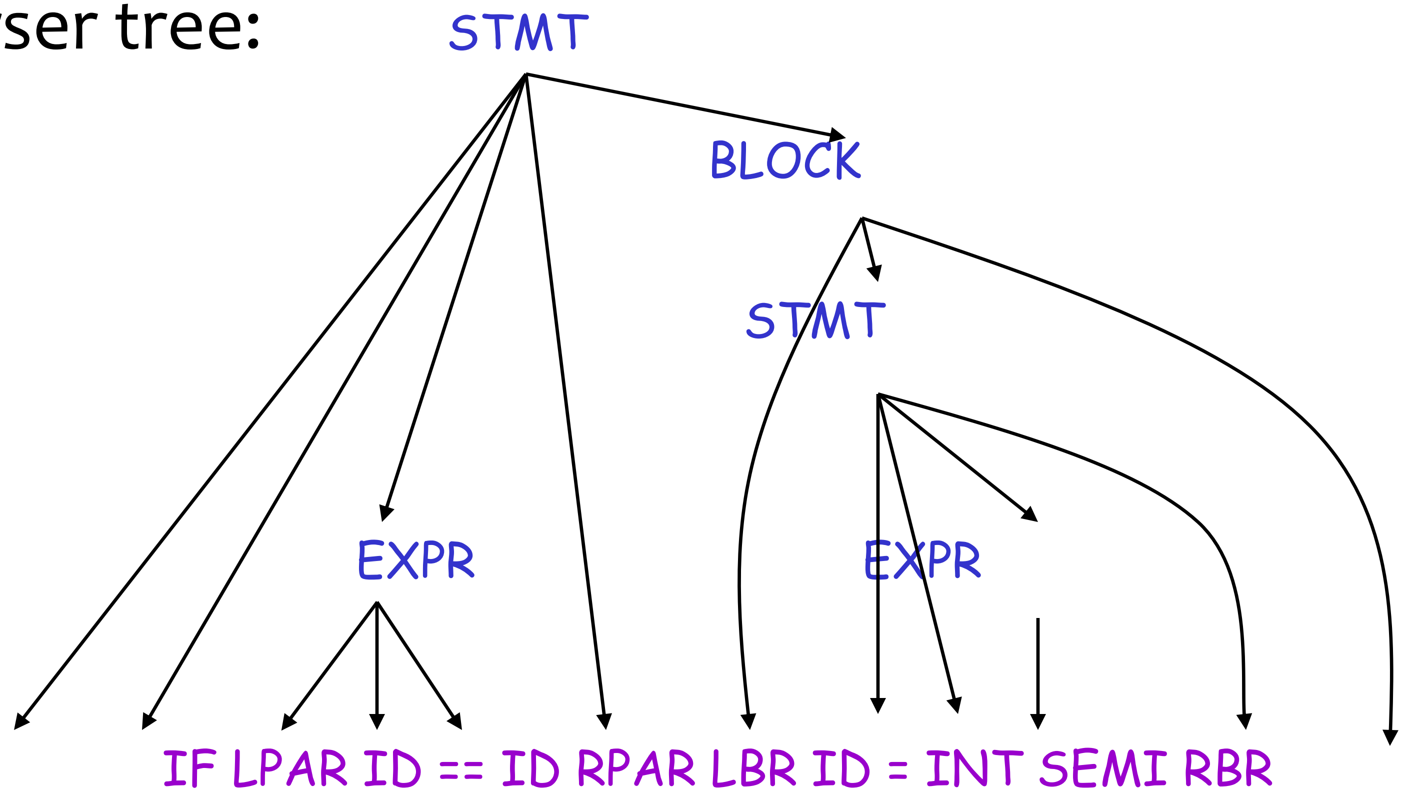


parser tree depends on the grammar; grammar is designed so that parser tree reflects operator precedence, associativity

☞ leaves are tokens (terminals), internal nodes are non-terminals

Another example

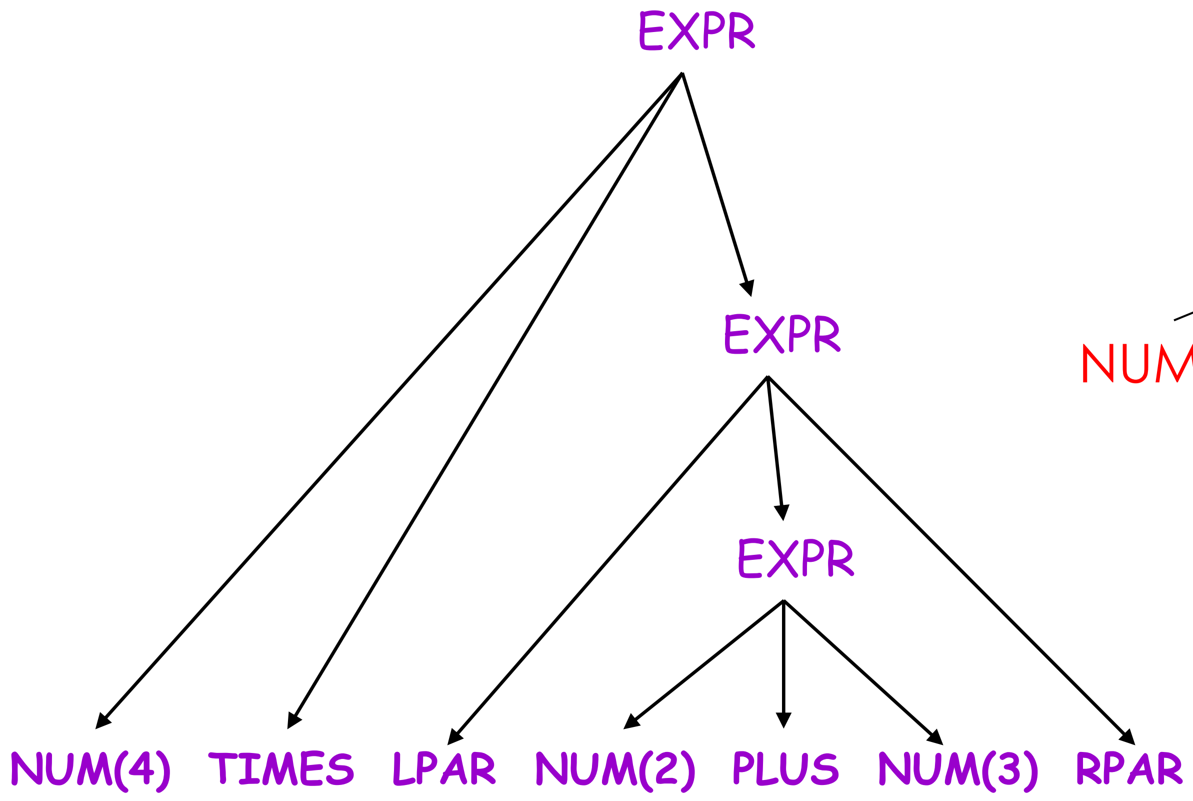
- Source: if (x == y) { a=1; }
- Parser input: **IF, LPAR, ID, EQ, ID, RPAR, LBR, ID, AS, INT, SEMI, RBR**
- Parser tree:

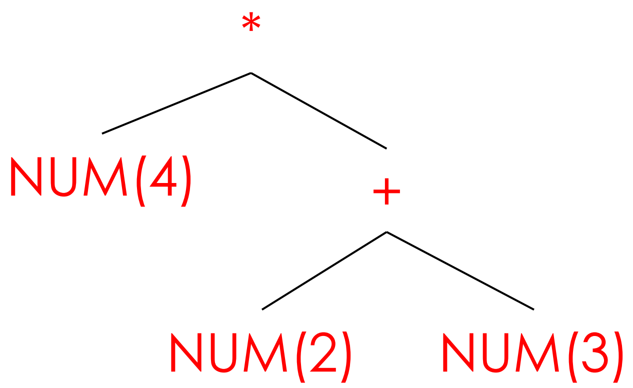


The Abstract Syntax Tree

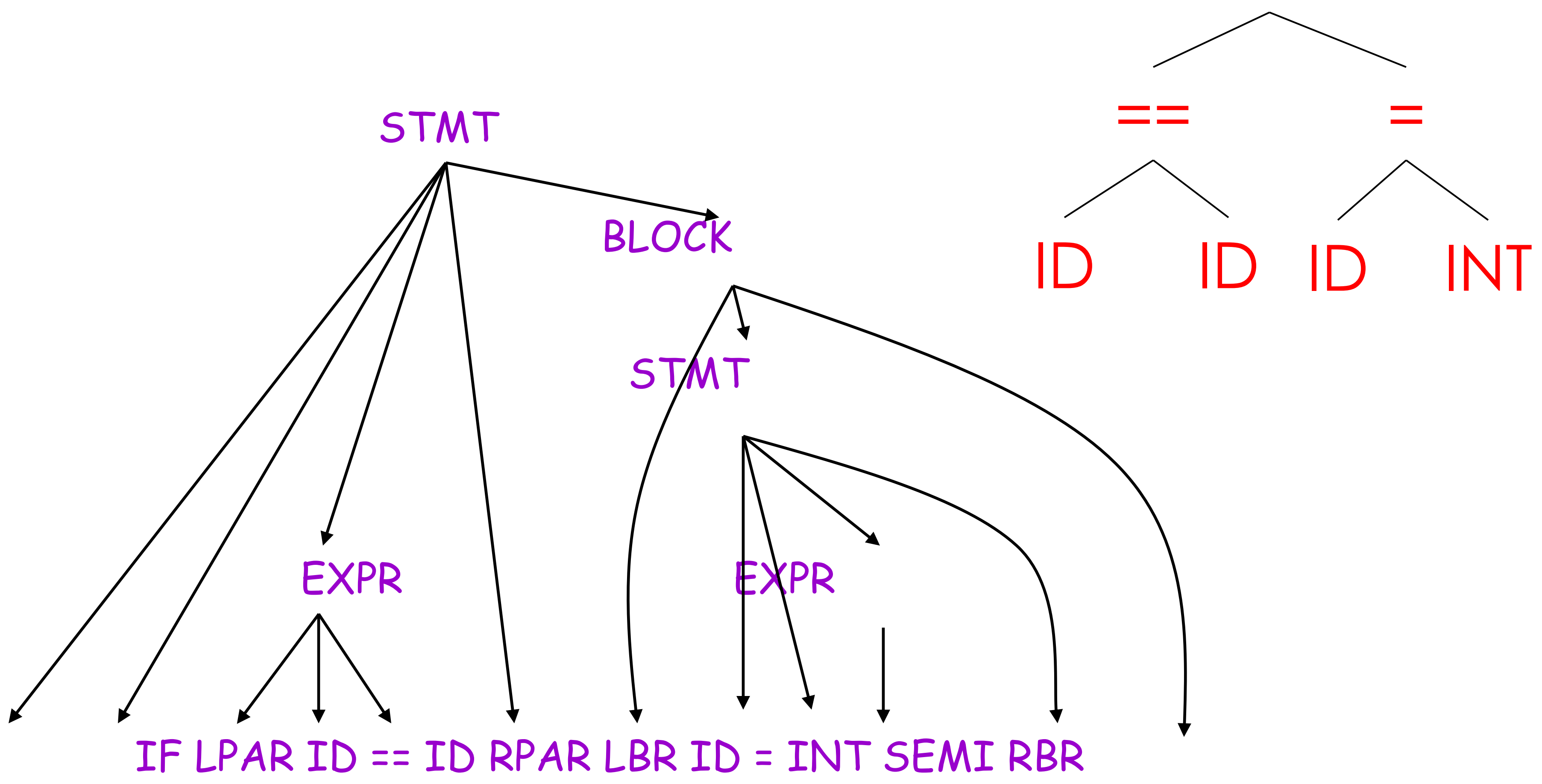
a compact representation of the tree structure

AST is a compression of the parse tree





Another example



- Parse tree determined by the grammar

AST determined by the syntax-directed translation (many designs possible)

Parse Tree Example

Given a parse tree, reconstruct the input:

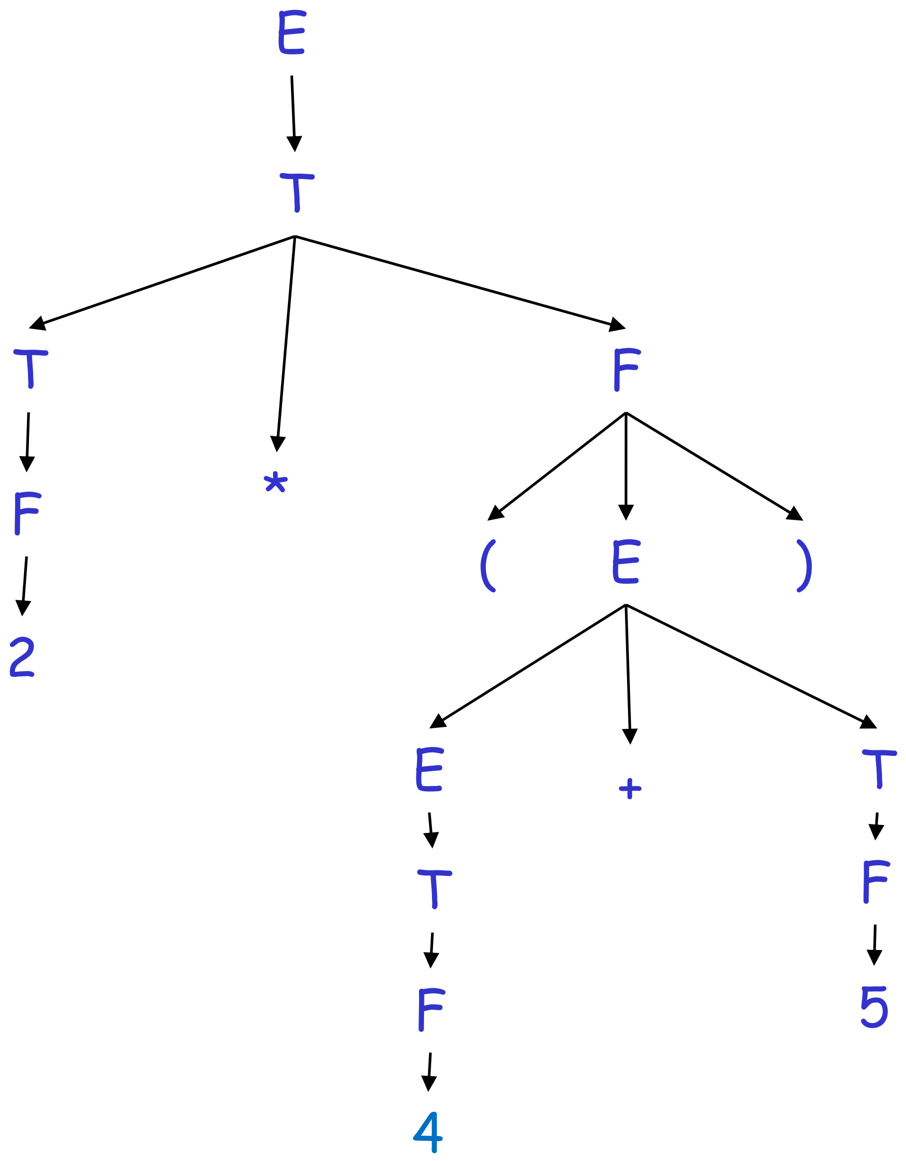
Input is given by leaves, left to right. In our case: 2*(4+5)

Can we reconstruct the grammar from the parse tree?:

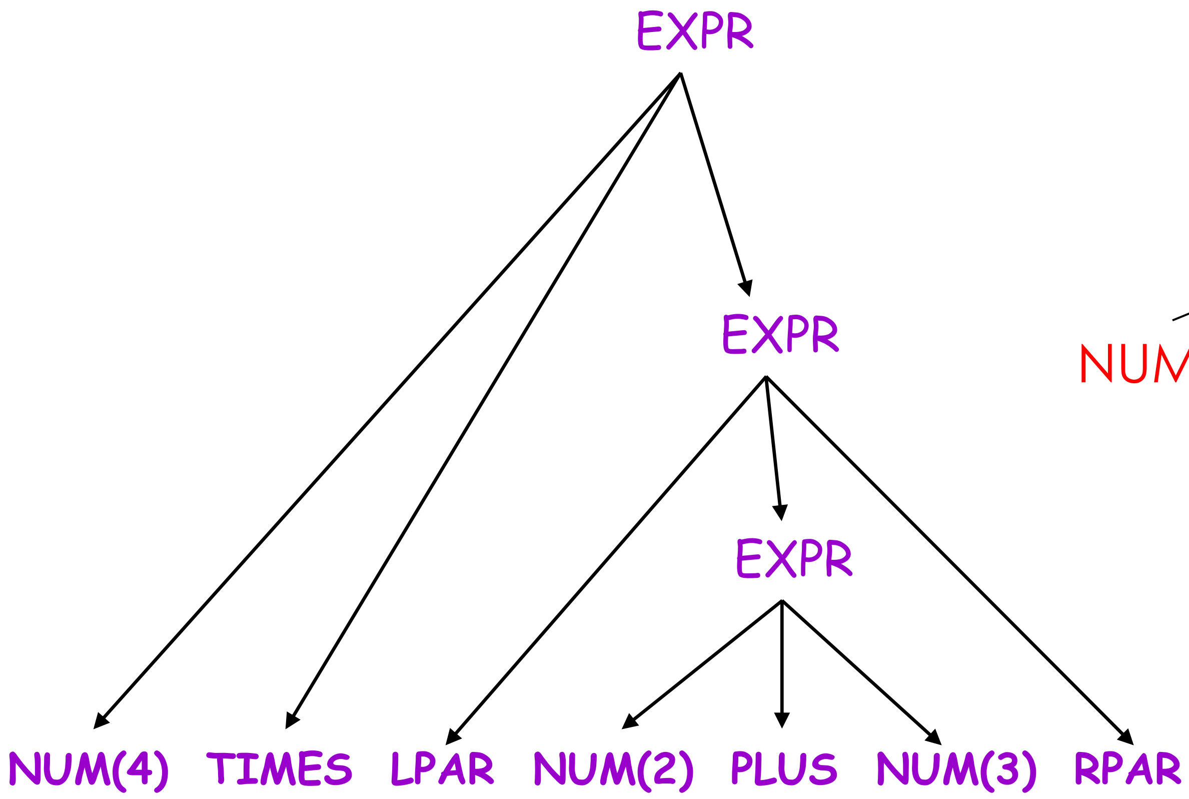
Yes, but only those rules that the input exercised. Our tree tells us the grammar contains at least these rules:

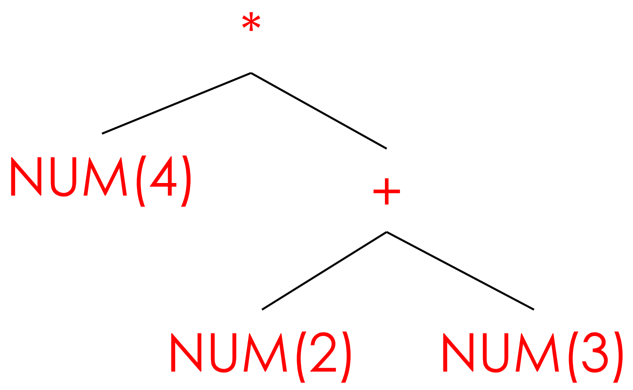
E ::= E + T | T
T ::= T * F | F
F ::= (E) | n

Evaluate the program using the tree:



Another application of parse tree: build AST





AST is a compression (abstraction) of the parse tree

What to do with the parse tree?

Applications:

- evaluate the input program P (interpret P)
- type check the program (look for errors before eval)
- construct AST of P (abstract the parse tree)

- generate code (which when executed, will evaluate P)
- compile (regular expressions to automata)

- layout the document (compute positions, sizes of letters)
- programming tools (syntax highlighting)

When is syntax directed translation performed?

Option 1: parse tree built explicitly during parsing

- after parsing, parse tree is traversed, rules are evaluated
- less common, less efficient, but simpler
- we'll follow this strategy in PA6

Option 2: parse tree never built

- rules evaluated during parsing on a conceptual parse tree
- more common in practice
- we'll see this strategy in a HW (on recursive descent parser)

Syntax-directed translation (SDT)

SDT is done by extending the grammar

- a translation rule is defined for each production:

given a production

$X \rightarrow d\ A\ B\ c$

the translation of X is defined in terms of

- translation of non-terminals A, B
- values of attributes of terminals d, c
- constants

translation of a (non-)terminal is called an attribute

- more precisely, a **synthesized** attribute
- (synthesized from values of children in the parse tree)

Specification of syntax-tree evaluation

Syntax-directed translation (*SDT*) for evaluating an expression

E_1 ::= E_2 + T	E_1.trans = E_2.trans + T.trans
E ::= T	E.trans = T.trans
T_1 ::= T_2 * F	T_1.trans = T_2.trans * F.trans
T ::= F	T.trans = F.trans
F ::= int	F.trans = int.value
F ::= (E)	F.trans = E.trans

SDT = grammar + “translation” rules

rules show how to evaluate parse tree

Same SDT in the notation of the cs164 parser

Syntax-directed translation for evaluating an expression

value of node E

val of T

```
%%
E -> E '+' T     %{ return n1.val + n3.val %}
   |  T          %{ return n1.val %}
   ;
T -> T '*' F     %{ return n1.val * n3.val }%
   |  F          missing rule is same as
                        %{ return n1.val }%
   ;
F -> /[0-9]+/    %{ return int(n1.val) }%
   | '(' E ')'   %{ return n2.val }%
   ;
```

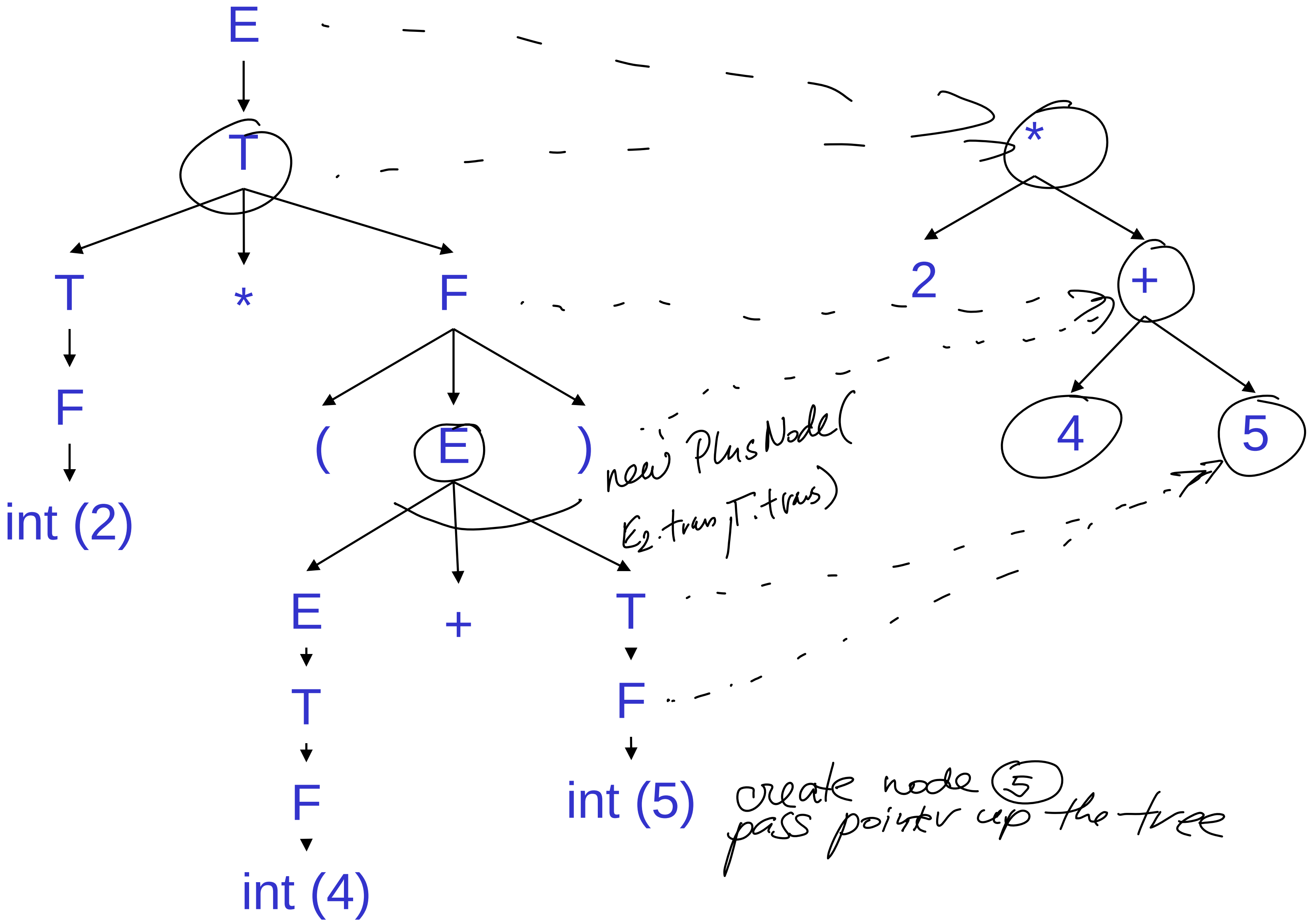
Example SDT: Compute type of expression + typecheck

```
E -> E + E     if (($E_2$.trans == INT) and ($E_3$.trans == INT))
                    then $E_1$.trans = INT
                    else $E_1$.trans = ERROR
E -> E and E   if (($E_2$.trans == BOOL) and ($E_3$.trans == BOOL))
                    then $E_1$.trans = BOOL
                    else $E_1$.trans = ERROR
E -> E == E    if (($E_2$.trans == $E_3$.trans) and
                  ($E_2$.trans != ERROR))
                    then $E_1$.trans = BOOL
                    else $E_1$.trans = ERROR
E -> true           E.trans = BOOL
E -> false          E.trans = BOOL
E -> int            E.trans = INT
E -> ( E )          $E_1$.trans = $E_2$.trans
```

AST-building translation rules

$E_1 \rightarrow E_2 + T$	E_1.trans = new PlusNode(E_2.trans, T.trans)
$E \rightarrow T$	E.trans = T.trans
$T_1 \rightarrow T_2 * F$	T_1.trans = new TimesNode(T_2.trans, F.trans)
$T \rightarrow F$	T.trans = F.trans
$F \rightarrow$ int	F.trans = new IntLitNode(int.value)
$F \rightarrow (E)$	F.trans = E.trans

Example: build AST for 2 * (4 + 5)


E
T
T
*
F
F
int (2)
(
E
)
E
+
T
T
F
F
int (4)
int (5)
*
2
+
4
5
new PlusNode(
E2.trans, T.trans)
create node 5
pass pointer up the tree