50.051 Programming Language Concepts

W10-S3 & W11-S1 Bottom-Up Parsing

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Different types of Parsing

Two big families of parsing algorithms

Top-Down Parsing

- Seen before, best we could do was in the case of LL(1) grammars, but very few grammars are going to be LL(k).

Bottom-Up Parsing

- Start from the input string *x*, whose syntax neds to be verified.
- Work your way back to a start symbol.
- Basically, the Top-Down parsing task, but in reverse!
- Builds on ideas of Top-Down parsing, less intuitive but more efficient, and might work on non-LL(k) grammars.

Introduction example for Bottom-Up Parsing

Fact #1 (to be confirmed later): Bottom-Up Parsers can deal with non-LL(k) grammars and can handle left- and right-recursive grammars.

- Consider the following grammar:

$$E \rightarrow E + (E) \mid int$$

- This CFG is this not LL(1).
- If not convinced, consider the string int + (int) + (int).

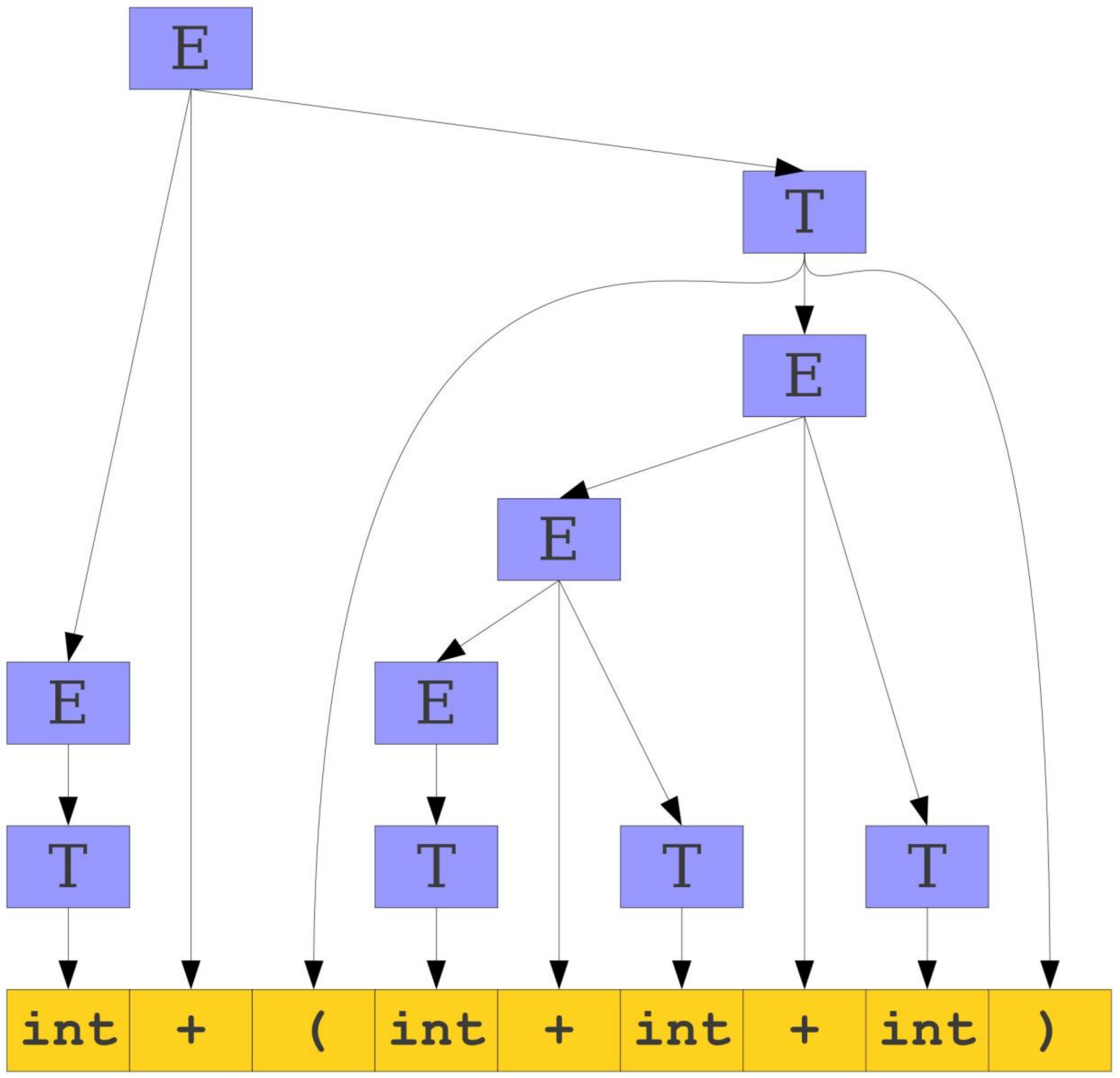
Introduction example for Bottom-Up Parsing

Procedure, in Layman terms: Bottom-Up Parsers attempts to reduce a string *x* to the start symbol by inverting productions.

- At the beginning, *x* should consist only of terminal symbols.
- Identify a substring β in *x*, such that $A \rightarrow \beta$ is a production rule of our CFG. In other words, it means $x = \alpha\beta\gamma$, with α and γ being strings of some sort (could be empty strings).
- Replace β with A inside of *x*, updating *x* as $\alpha A\gamma$.
- Keep on doing so, until *x* becomes the start symbol S of the CFG.

Introduction example for Bottom-Up Parsing

$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



Introduction example for Bottom-Up Parsing

$$\begin{aligned}
E &\to T\\
E &\to E + T\\
T &\to \texttt{int}\\
T &\to \texttt{(}E\texttt{)}
\end{aligned}$$

$$\begin{aligned}
& \texttt{int + (int + int + int)}\\
\Rightarrow\ & T\ \texttt{+ (int + int + int)}\\
\Rightarrow\ & E\ \texttt{+ (int + int + int)}\\
\Rightarrow\ & E\ \texttt{+ (}T\ \texttt{+ int + int)}\\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+ int + int)}\\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+}\ T\ \texttt{+ int)}\\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+ int)}\\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+}\ T\texttt{)}\\
\Rightarrow\ & E\ \texttt{+ (}E\texttt{)}\\
\Rightarrow\ & E\ \texttt{+}\ T\\
\Rightarrow\ & E
\end{aligned}$$

Introduction example for Bottom-Up Parsing

$$\begin{aligned}
E &\to T \\
E &\to E + T \\
T &\to \mathtt{int} \\
T &\to (E)
\end{aligned}$$

$$\begin{aligned}
& \mathtt{int} + (\mathtt{int} + \mathtt{int} + \mathtt{int}) \\
\Rightarrow\ & T + (\mathtt{int} + \mathtt{int} + \mathtt{int}) \\
\Rightarrow\ & E + (\mathtt{int} + \mathtt{int} + \mathtt{int}) \\
\Rightarrow\ & E + (T + \mathtt{int} + \mathtt{int}) \\
\Rightarrow\ & E + (E + \mathtt{int} + \mathtt{int}) \\
\Rightarrow\ & E + (E + T + \mathtt{int}) \\
\Rightarrow\ & E + (E + \mathtt{int}) \\
\Rightarrow\ & E + (E + T) \\
\Rightarrow\ & E + (E) \\
\Rightarrow\ & E + T \\
\Rightarrow\ & E
\end{aligned}$$

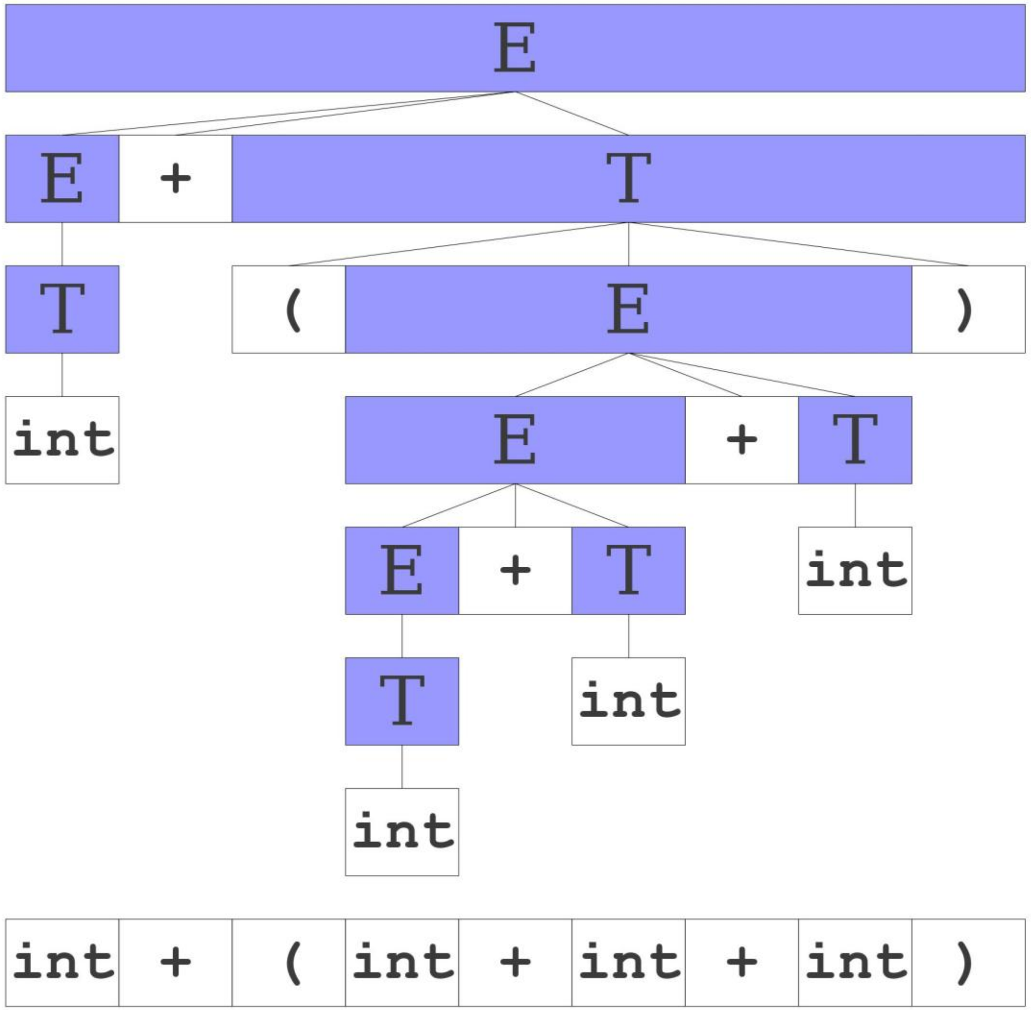
Introduction example for Bottom-Up Parsing

Fact #2: A Bottom-Up parser traces exactly what a top-down rightmost derivation would do, but in reverse.

- This has an interesting consequence…
- Let $\alpha\beta\gamma$ be a step of a bottom-up parse.
- Assume the next reduction is by $A \rightarrow \beta$
- **Then γ is necessarily a string of terminals!**
- Why? Because $\alpha A\gamma \rightarrow \alpha\beta\gamma$ is a step in a rightmost derivation!
  (If γ does not change, it means it does not contain any non-terminals)

Introduction example for Bottom-Up Parsing

int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



Introduction example for Bottom-Up Parsing

Follow-up idea from Fact #2: Split the string into two substrings.

- The right substring should consist only of terminal symbols, and has yet to be examined by the parser.
- The left substring could have terminals and non-terminals.
- Mark the dividing point using the | symbol.
  (Note that this symbol | is only for visualization purposes, it is not part of the string to be analysed!)
- At the beginning, the string x is therefore written as *x* = $|x_1 x_2 \dots x_n\$$, with $x_1, x_2, \dots x_n$ being terminal symbols.

Shift-Reduce Parsing

Fact #3: A Bottom-up Parser will attempt to revert the string by using only two possible actions.

- **Shifting:** Moves the separator one step to the right (one full symbol).

For instance, E | + (int) becomes E + | (int) after shifting.

- **Reducing:** Apply an inverse production rule at the right of the left end string.

For instance, the string E + (E + (E) |) can be reduced into E + (E |) by using the production E → E + (E).

Practice 1: Shift-Reduce tryout

Question: Consider the CFG on the right, what is the correct sequence of Reduce and Shift operations to use on the string *x* below?

int + int * int + int

$$
\begin{aligned}
E &\to F \\
E &\to E + F \\
F &\to F * T \\
F &\to T \\
T &\to \texttt{int} \\
T &\to (E)
\end{aligned}
$$

Practice 1: Shift-Reduce tryout

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

int	+	int	*	int	+	int

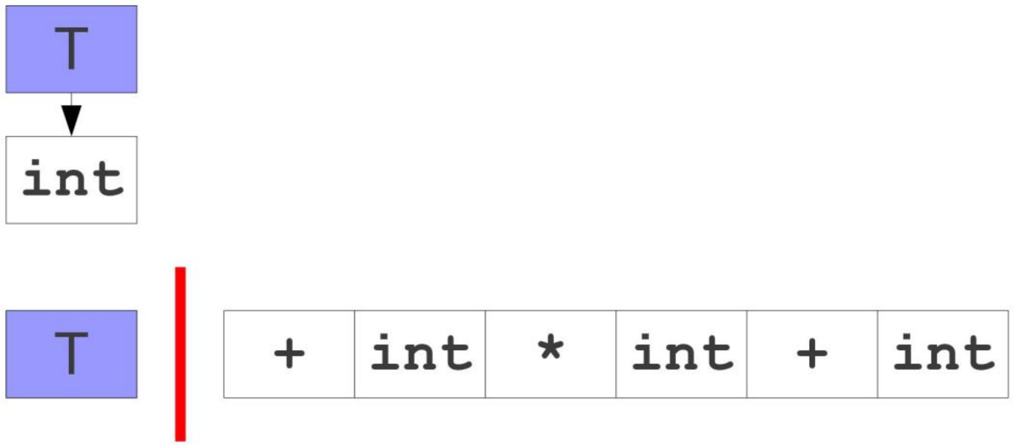
Practice 1: Shift-Reduce tryout

$$
\begin{aligned}
E &\rightarrow F \\
E &\rightarrow E + F \\
F &\rightarrow F * T \\
F &\rightarrow T \\
T &\rightarrow \texttt{int} \\
T &\rightarrow (E)
\end{aligned}
$$

int	|	+	int	*	int	+	int

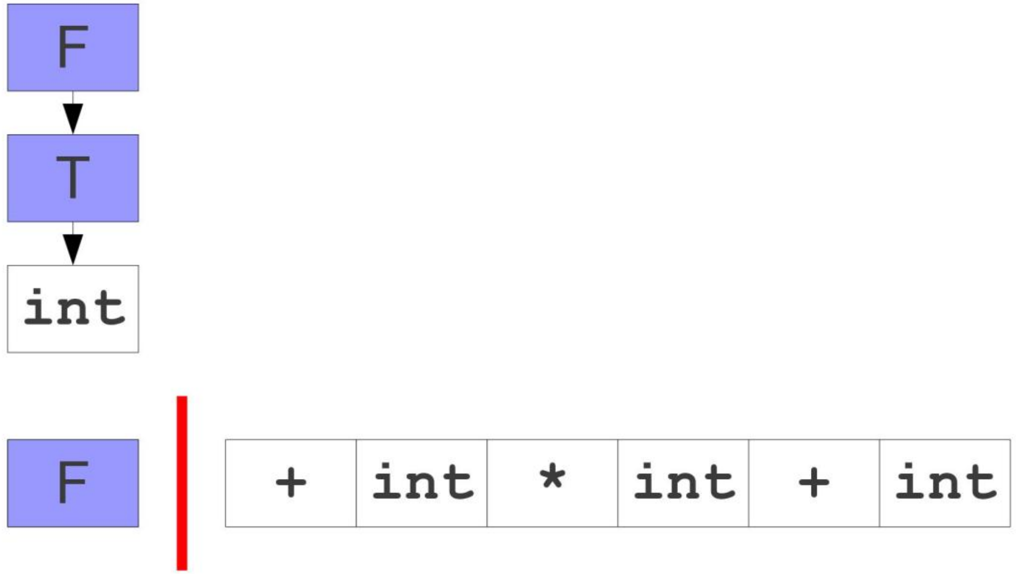
Practice 1: Shift-Reduce tryout

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



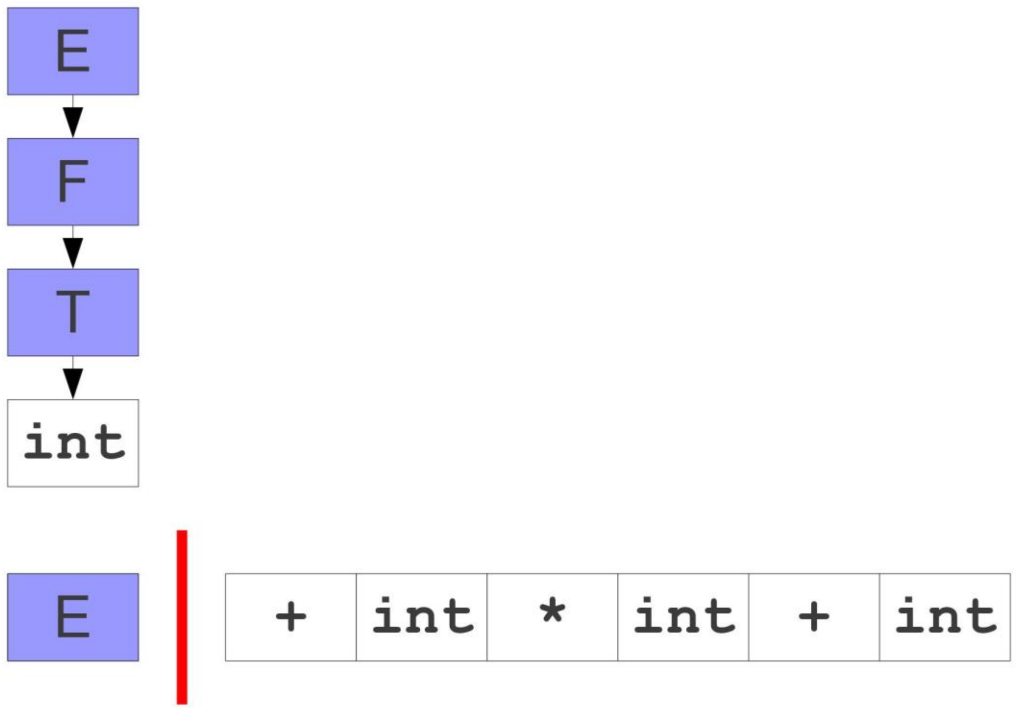
Practice 1: Shift-Reduce tryout

$$
\begin{aligned}
\mathbf{E} &\rightarrow \mathbf{F} \\
\mathbf{E} &\rightarrow \mathbf{E} + \mathbf{F} \\
\mathbf{F} &\rightarrow \mathbf{F} * \mathbf{T} \\
\mathbf{F} &\rightarrow \mathbf{T} \\
\mathbf{T} &\rightarrow \texttt{int} \\
\mathbf{T} &\rightarrow (\mathbf{E})
\end{aligned}
$$



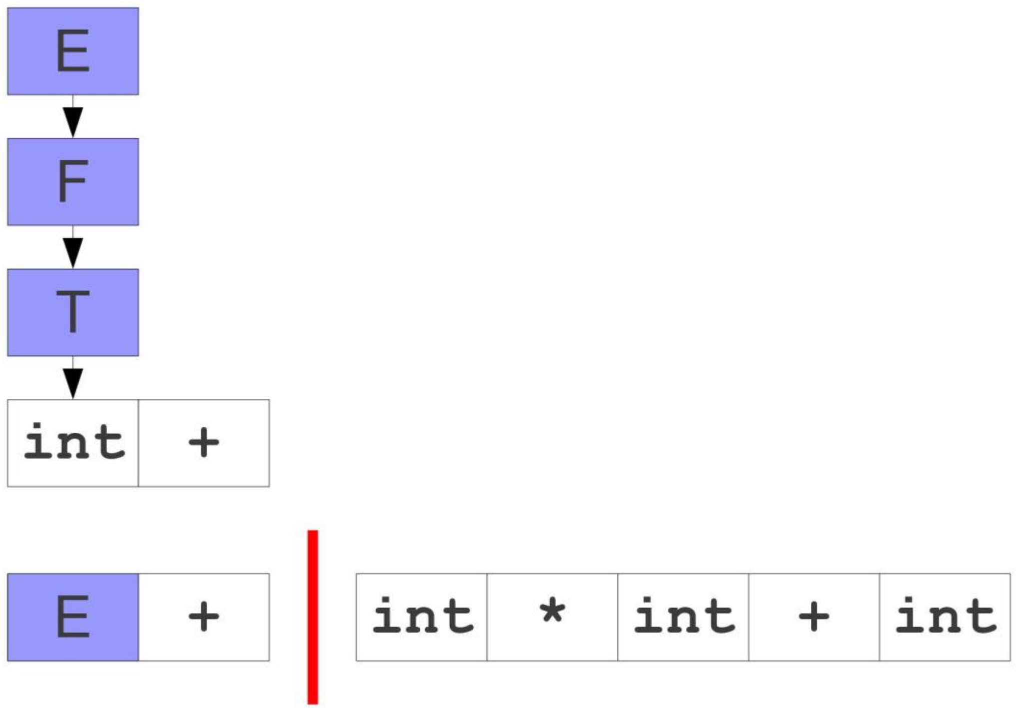
Practice 1: Shift-Reduce tryout

$E \to F$
$E \to E + F$
$F \to F * T$
$F \to T$
$T \to \texttt{int}$
$T \to (E)$



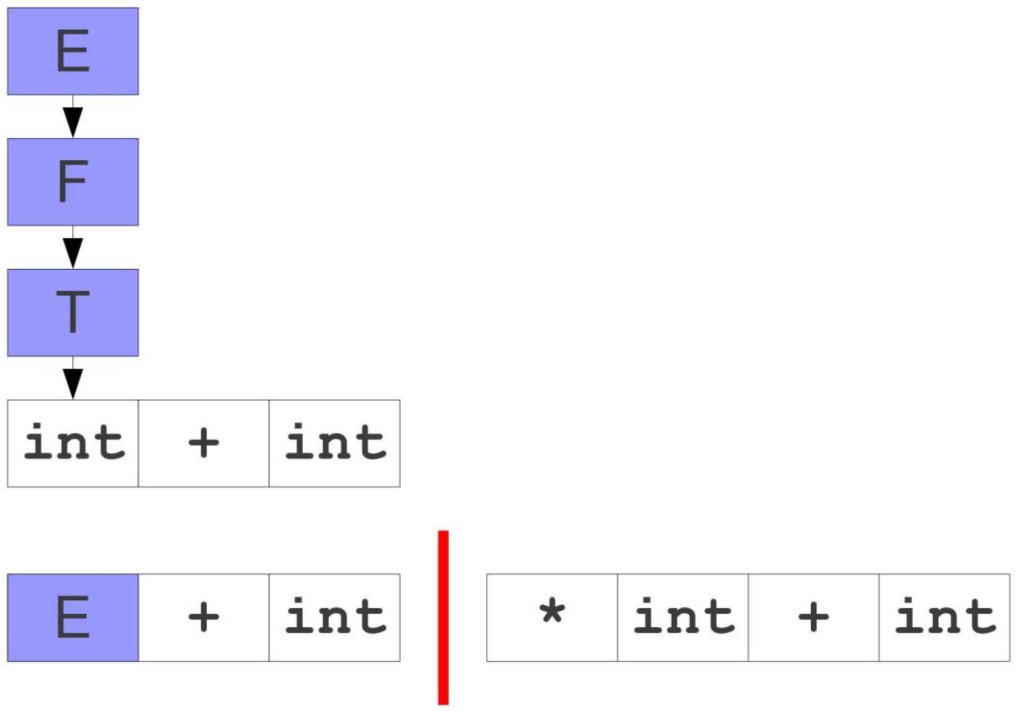
Practice 1: Shift-Reduce tryout

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \mathtt{int}$
$T \rightarrow (E)$



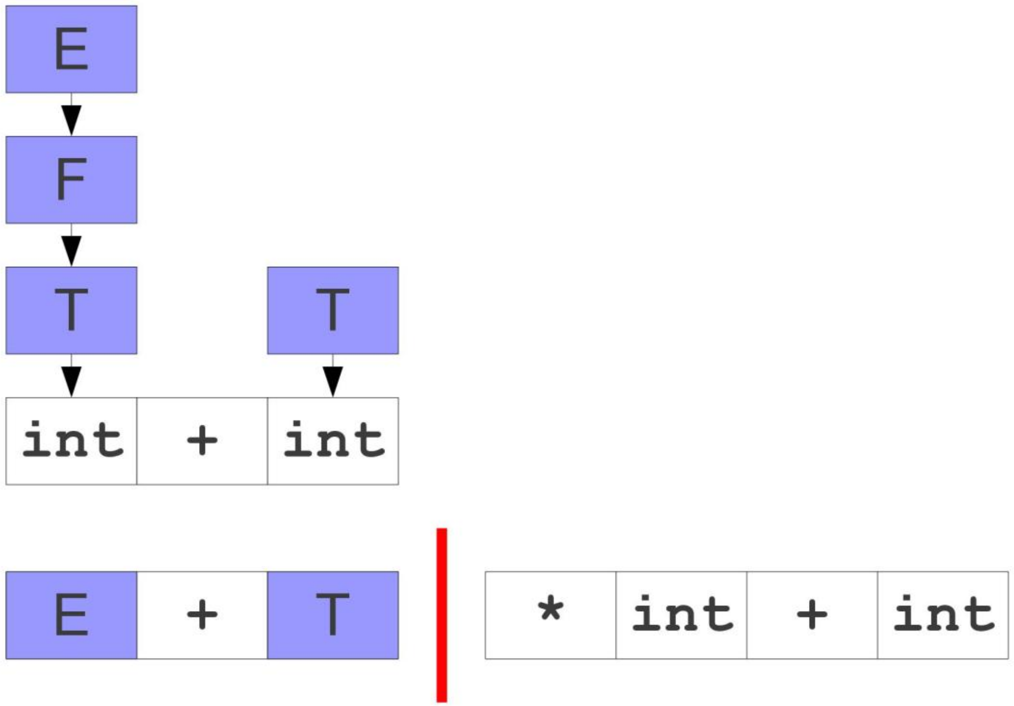
Practice 1: Shift-Reduce tryout

$$
\begin{aligned}
E &\rightarrow F \\
E &\rightarrow E + F \\
F &\rightarrow F * T \\
F &\rightarrow T \\
T &\rightarrow \texttt{int} \\
T &\rightarrow (E)
\end{aligned}
$$



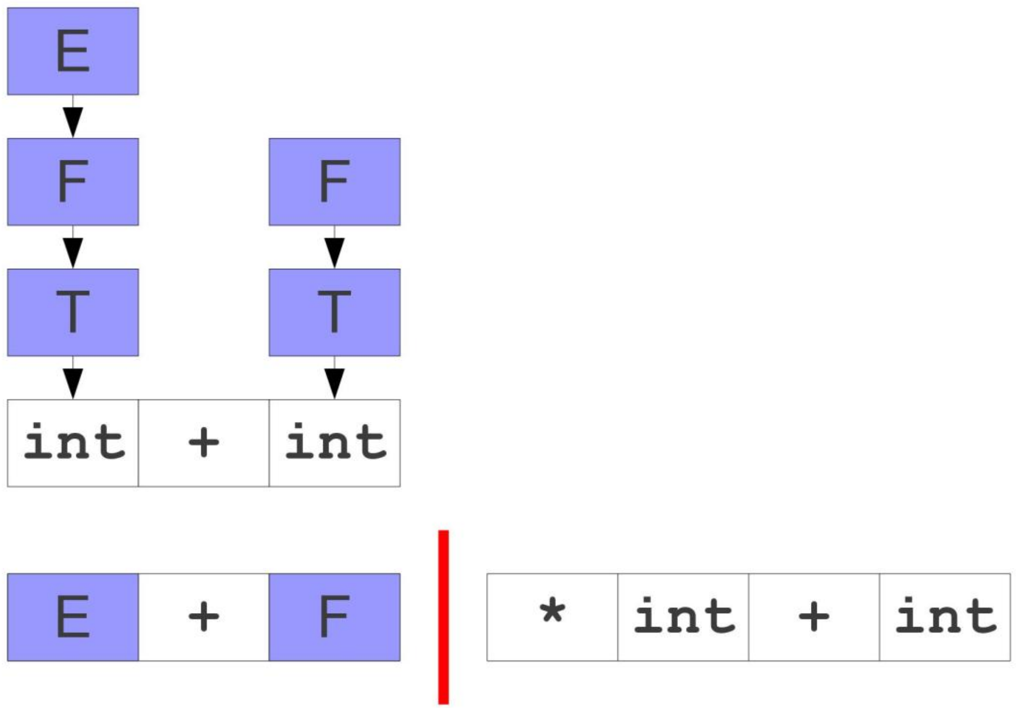
Practice 1: Shift-Reduce tryout

$$\mathbf{E} \to \mathbf{F}$$
$$\mathbf{E} \to \mathbf{E} + \mathbf{F}$$
$$\mathbf{F} \to \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \to \mathbf{T}$$
$$\mathbf{T} \to \texttt{int}$$
$$\mathbf{T} \to (\mathbf{E})$$



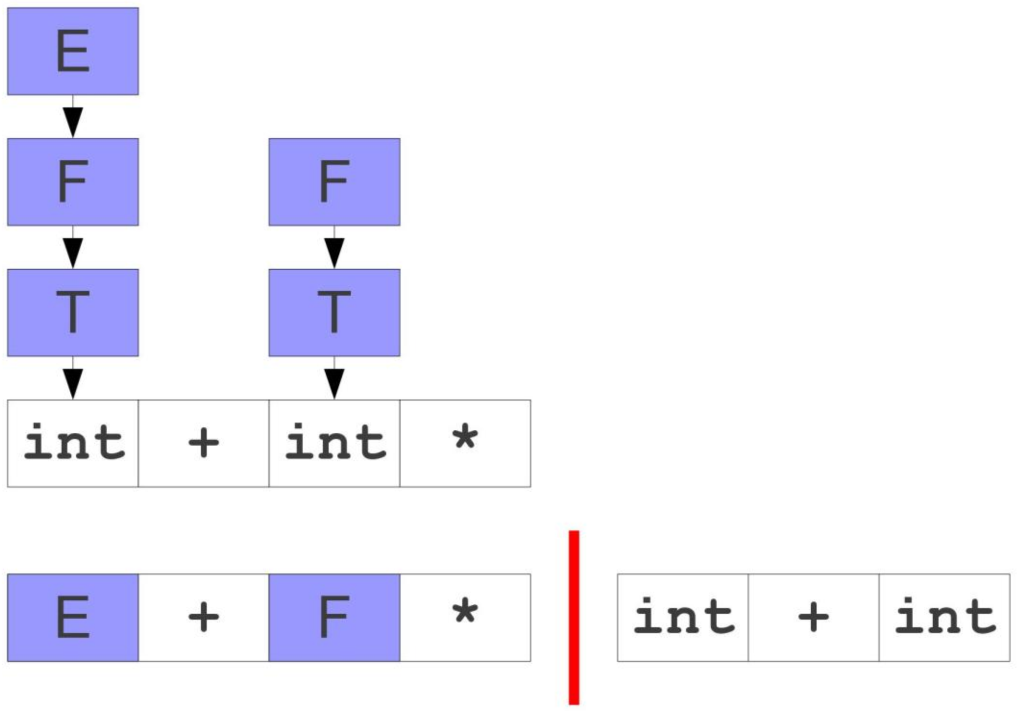
Practice 1: Shift-Reduce tryout

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow (\mathbf{E})$



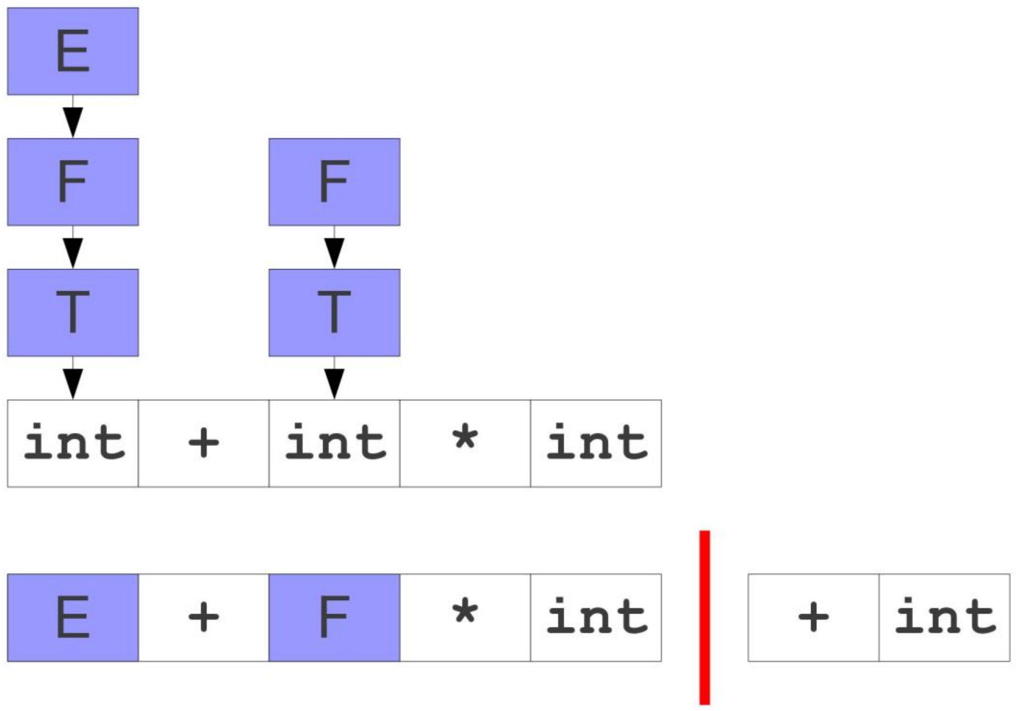
Practice 1: Shift-Reduce tryout

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



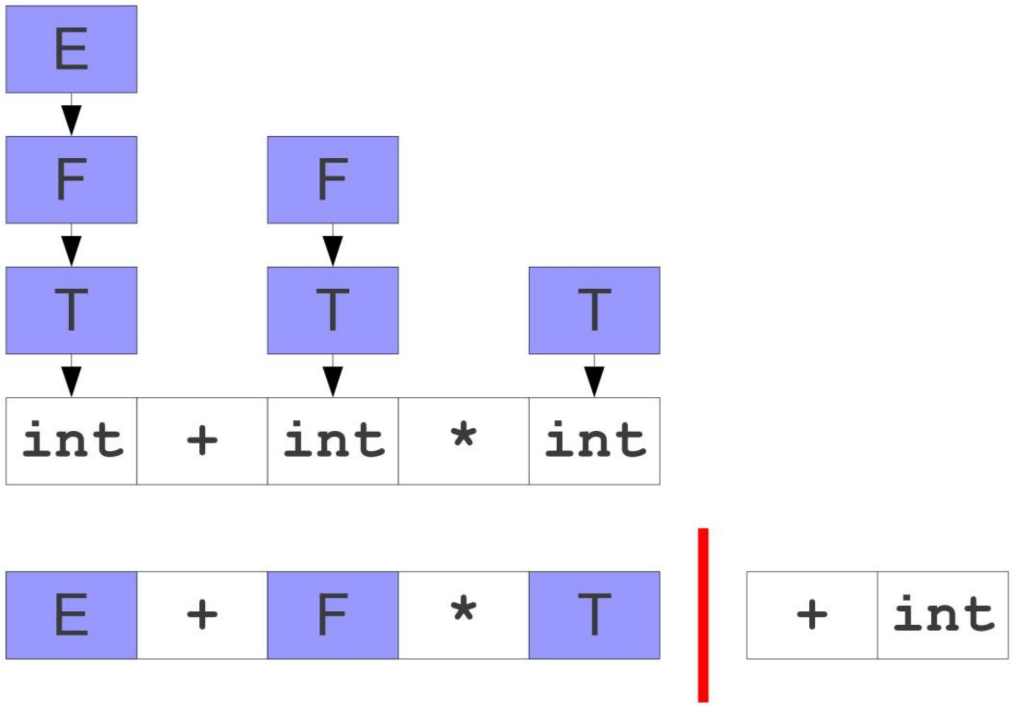
Practice 1: Shift-Reduce tryout

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow (\mathbf{E})$



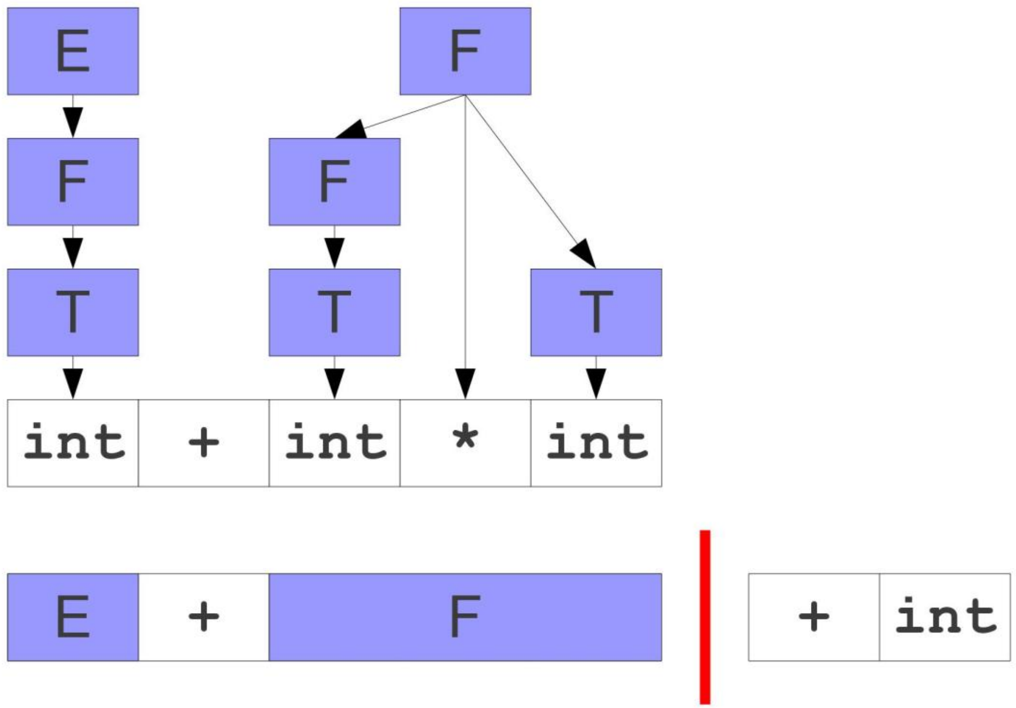
Practice 1: Shift-Reduce tryout

$$\mathbf{E} \rightarrow \mathbf{F}$$
$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$$
$$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow \mathbf{T}$$
$$\mathbf{T} \rightarrow \texttt{int}$$
$$\mathbf{T} \rightarrow (\mathbf{E})$$



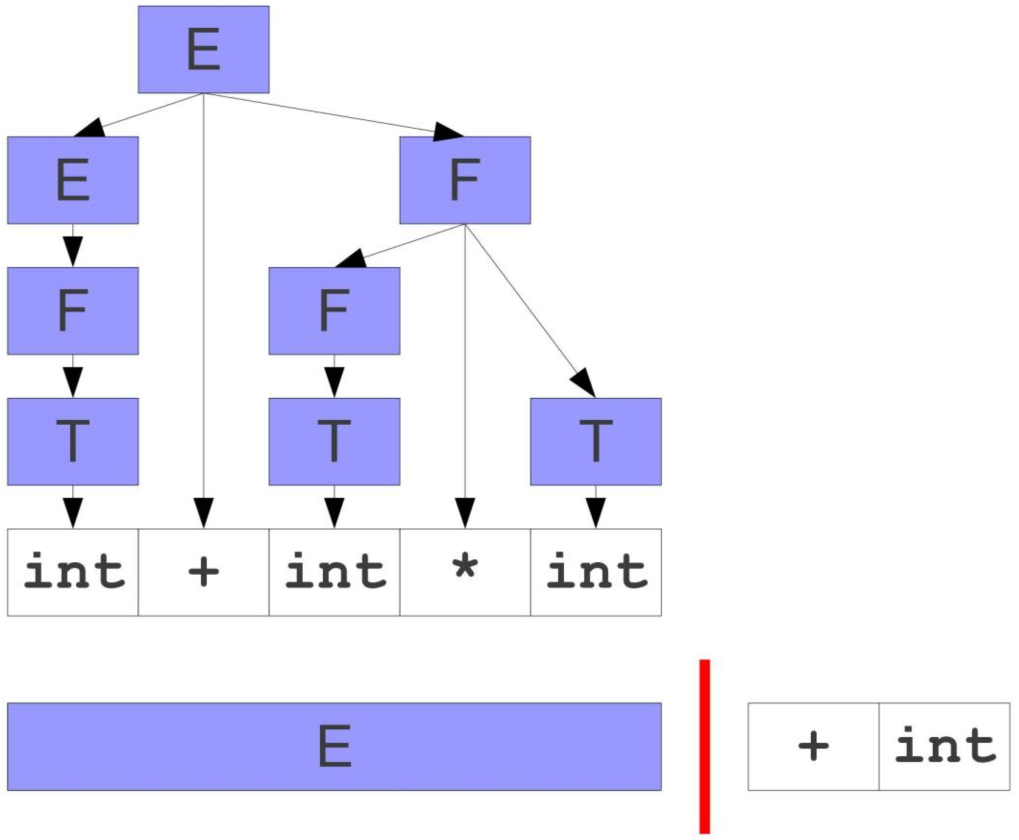
Practice 1: Shift-Reduce tryout

$$\mathbf{E} \rightarrow \mathbf{F}$$
$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$$
$$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathtt{int}$$
$$\mathbf{T} \rightarrow (\mathbf{E})$$



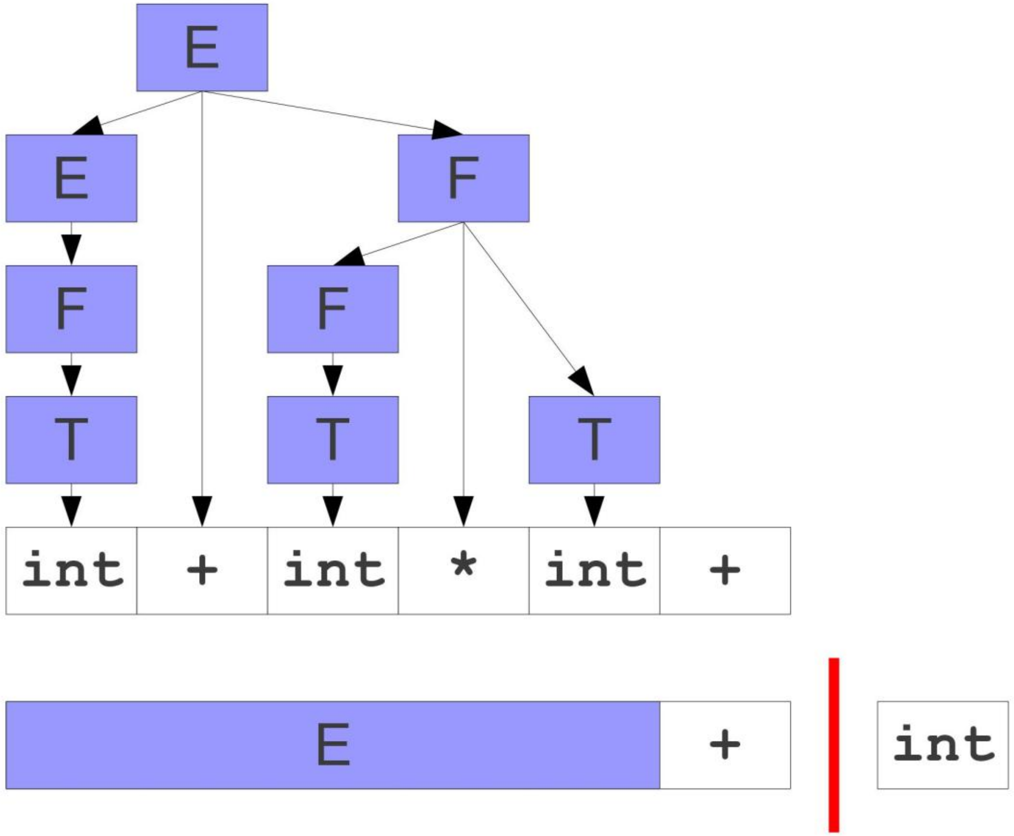
Practice 1: Shift-Reduce tryout

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow (\mathbf{E})$



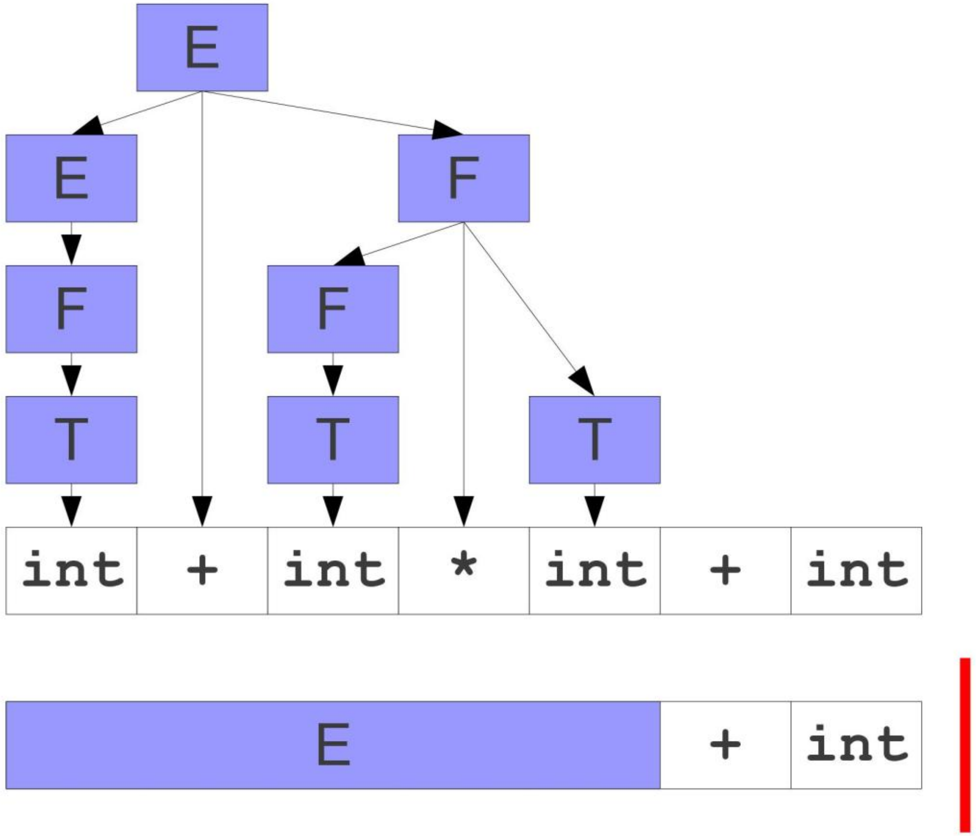
Practice 1: Shift-Reduce tryout

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`



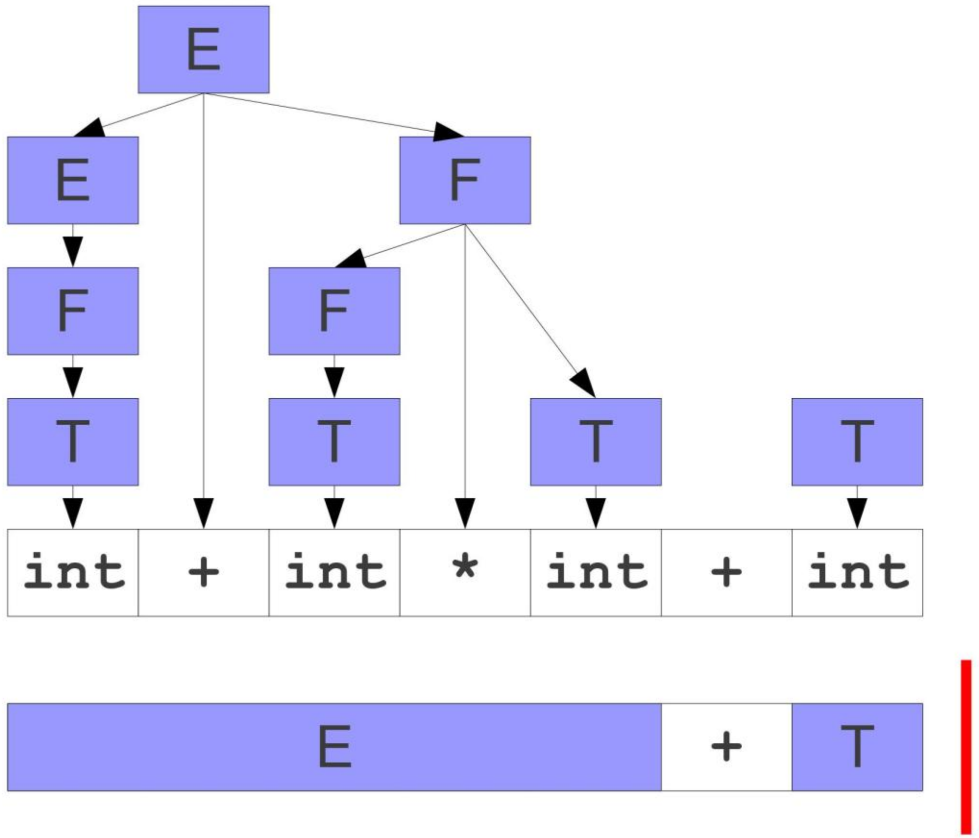
Practice 1: Shift-Reduce tryout

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow (\mathbf{E})$



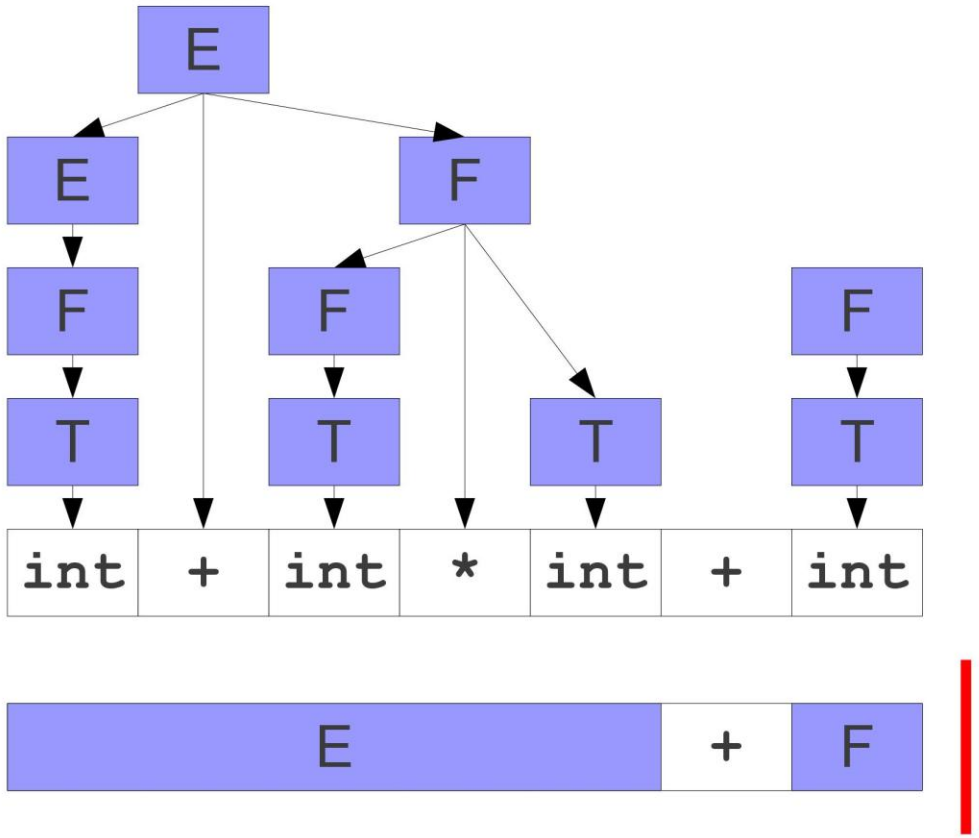
Practice 1: Shift-Reduce tryout

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`



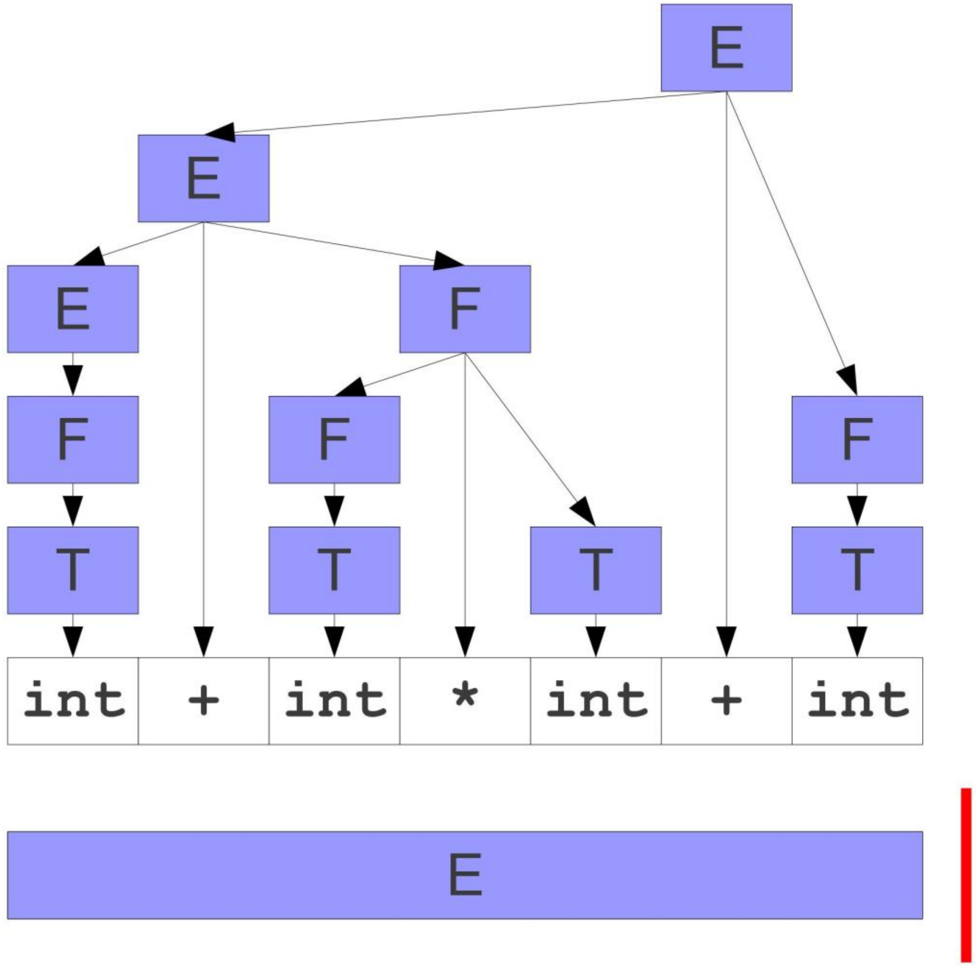
Practice 1: Shift-Reduce tryout

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow (\mathbf{E})$



Practice 1: Shift-Reduce tryout

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



Practical implementation of Shift-Reduce

Implementation is out of scope, but the practical implementation for the Shift-Reduce parser can be done as follows.

- **Left substring** can be implemented by a **stack**.
- Top of the stack is denoted by the separator symbol | .
- **Shifting pushes a terminal symbol on the stack.**
- **Reducing using a production rule A → B pops all the symbols in B off the top of the stack and then pushes a non-terminal symbol A on the stack to replace all the symbols in B.**
- (Also, two more actions technically being: **accept** the string if parsing has completed, or **reject** the string if derivation is not possible due to invalid syntax…)

Million dollar question

Million dollar question: How to decide when to Shift, when to Reduce, and with which production rule from the CFG?

A few observations we can make so far

1. **You can always shift at any time,** except when the separator | has reached the end of the string.
2. **When you cannot reduce the string using any production rule of the CFG, shift, by default.**
3. **The difficulty comes from deciding between shifting and reducing in scenarios where you could do both. To decide, we will reuse the idea of LL(1), about looking at the first terminal symbol to appear after the split symbol |.**

How to decide between shift/reduce?

First Idea: Build an FSM (again!), to decide when to shift or reduce.

- FSM will use, as input string, the current stack.
- The **actions** consist of the possible **terminals and non-terminals symbols** that the CFG possesses, as well as $.

How to decide between shift/reduce?

First Idea: Build an FSM (again!), to decide when to shift or reduce.

- The states will have **shift-reduce(rule)-accept-reject actions** assigned to them.
- Whatever final state we end up on after running the FSM will tell us which parsing action to use between shift and reduce.
- In the case of a reduce action, we will also mention the production rule to use to transform elements in our stack.
- Also, keep in mind that this is a predictive parser. This means one lookahead symbol should be used. In our case, this means that **the parsing action to use for a given final state might depend on what is currently the first terminal symbol immediatly after | .**

An example FSM for our CFG

Consider our simplified CFG from earlier, below.

$$1: E \rightarrow E + (E)$$

$$2: E \rightarrow int$$

Our FSM has the following **actions**: E, (,), +, int, $.

The parser has the following **parsing actions** to be assigned to **states**: shift, reduce using rule 1, reduce using rule 2, accept, reject.

For simplicity, we will not draw all **states transitions**.

If a state transition is used but does not appear in the FSM diagram (on the next slide), then we should stop the FSM and reject the string due to invalid syntax!

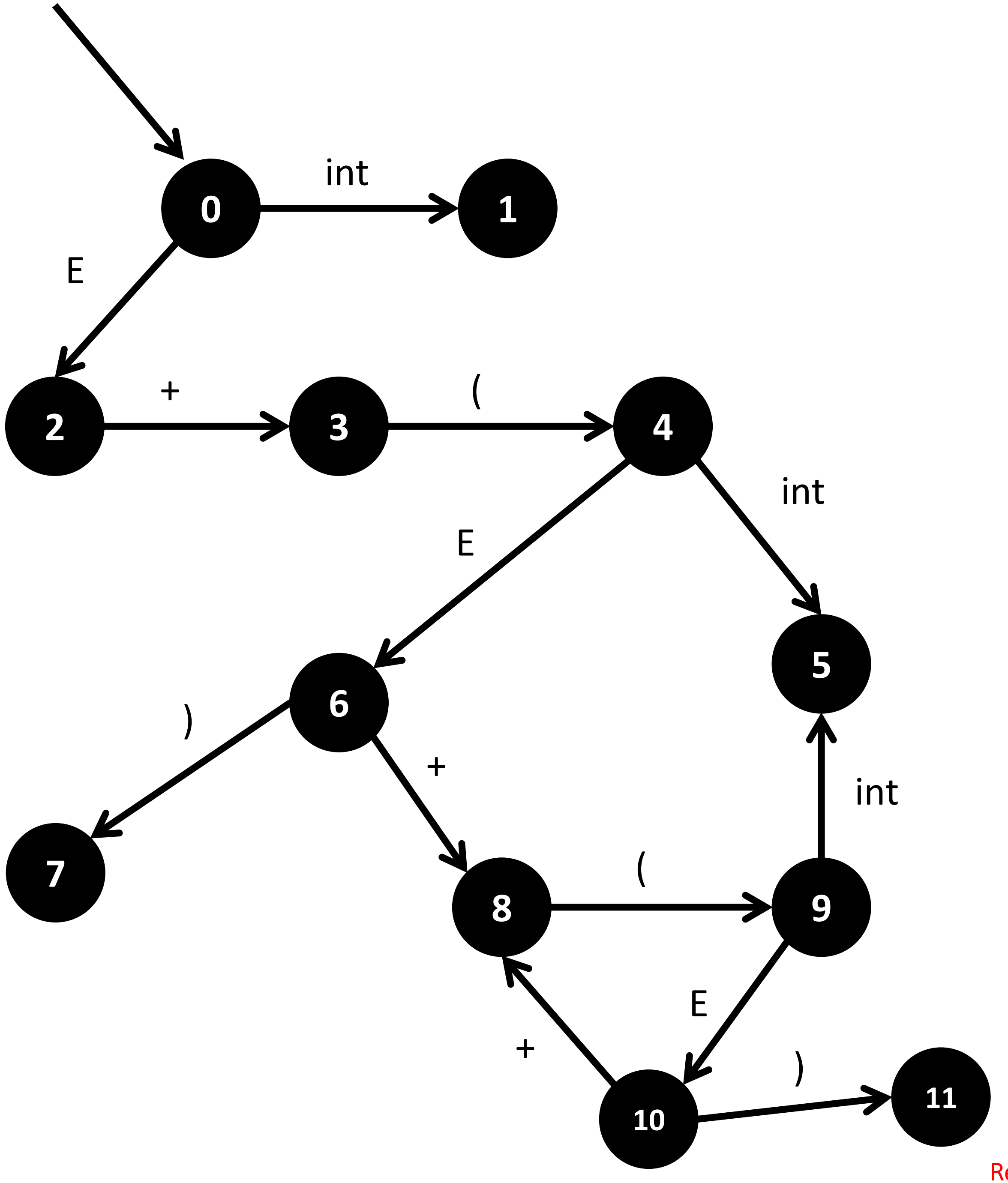
An example FSM for our CFG

Consider our simplified CFG from earlier, below.

$$1: E \rightarrow E + (E)$$

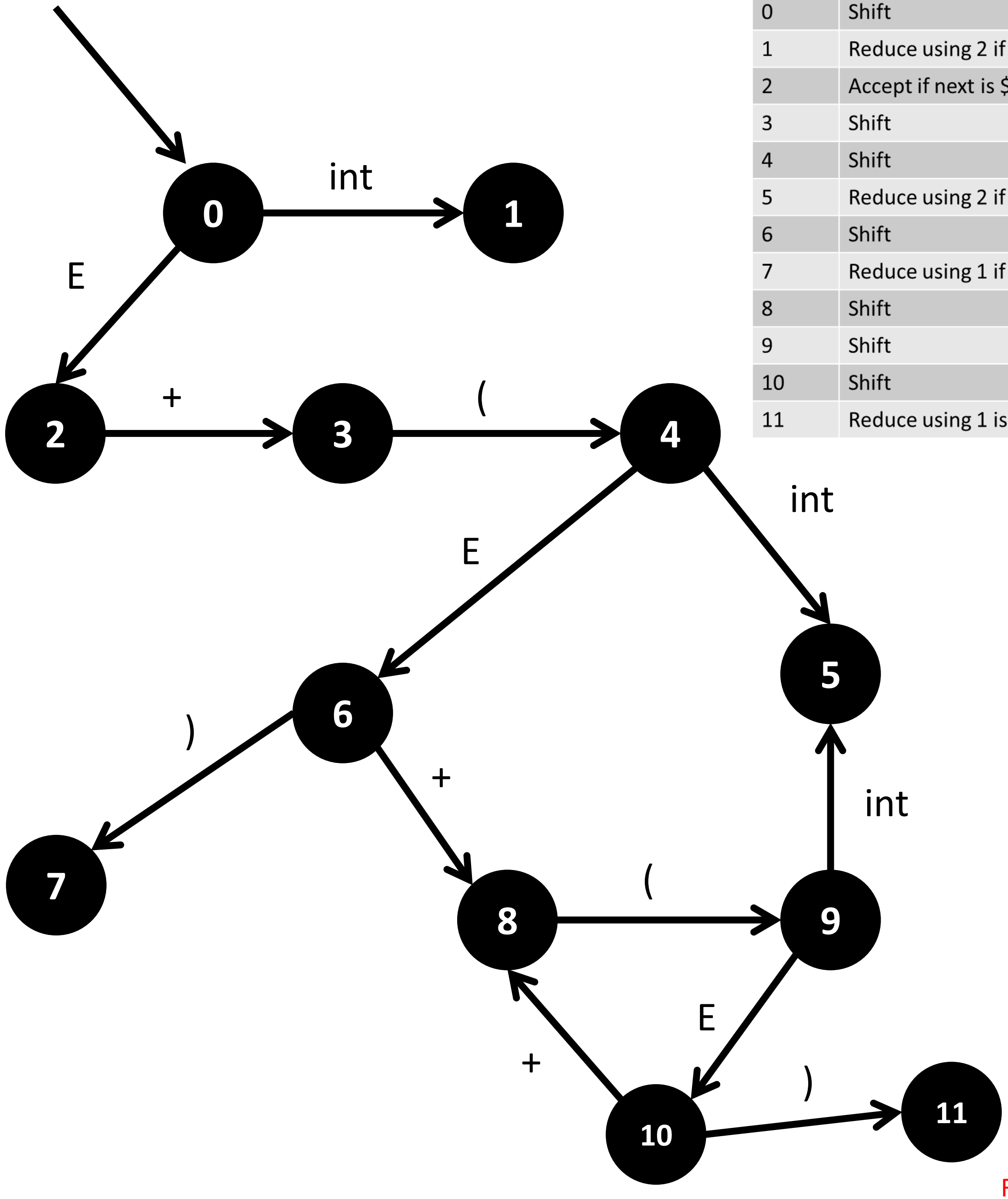
$$2: E \rightarrow int$$

The FSM hiding behind the predictive parser for our CFG above is…
(Given on the next slide)



State	Parsing action
0	Shift
1	Reduce using 2 if next is \$ or +
2	Accept if next is \$
3	Shift
4	Shift
5	Reduce using 2 if next is) or +
6	Shift
7	Reduce using 1 if next is \$ or +
8	Shift
9	Shift
10	Shift
11	Reduce using 1 is next is) or +

Note: shift if reduce condition is not met, if the next symbol after | is not listed. For instance, on state 1, we use reduce if the element after | is \$ or +; and shift otherwise.

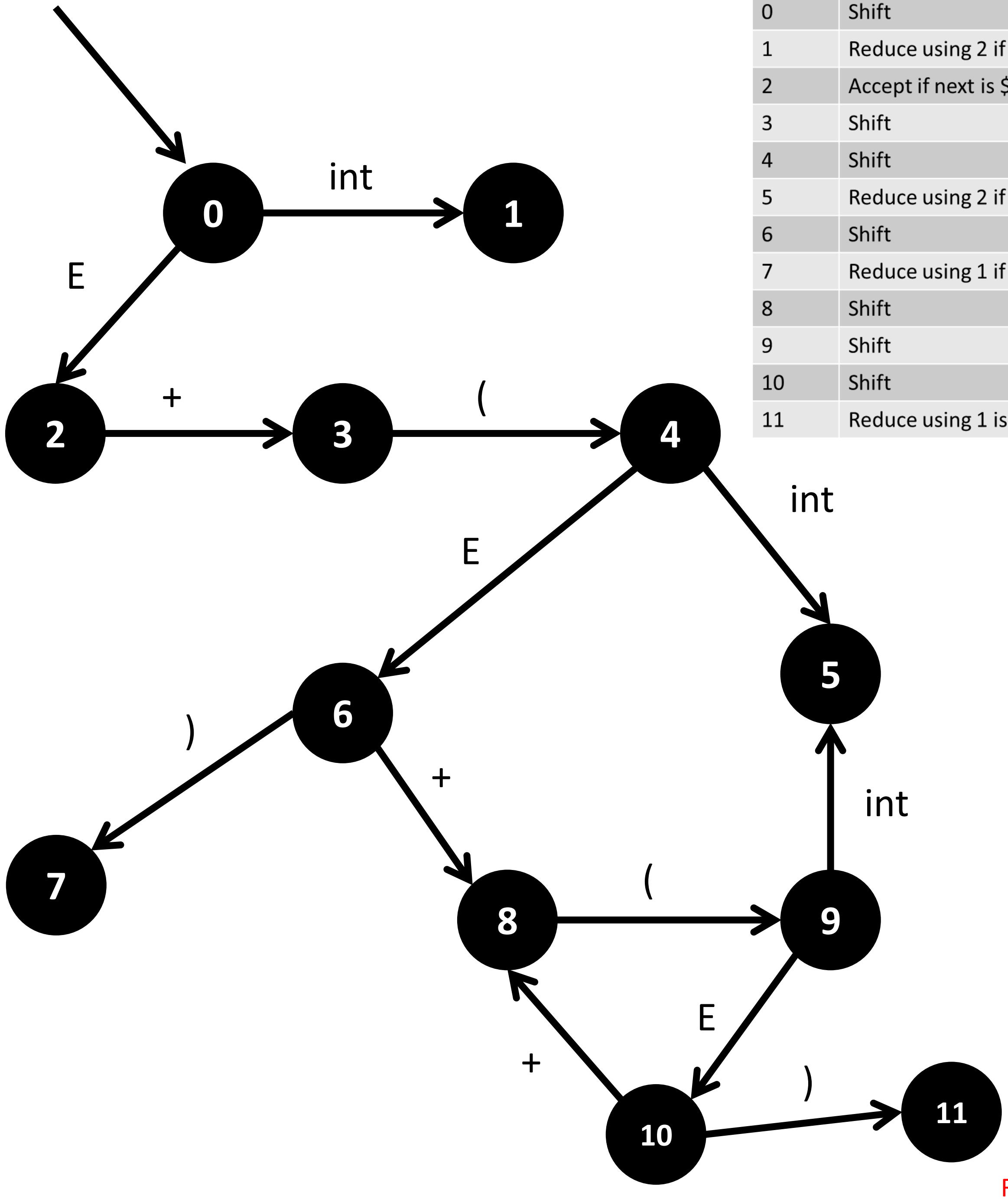


State	Parsing action
0	Shift
1	Reduce using 2 if next is $ or +
2	Accept if next is $
3	Shift
4	Shift
5	Reduce using 2 if next is) or +
6	Shift
7	Reduce using 1 if next is $ or +
8	Shift
9	Shift
10	Shift
11	Reduce using 1 is next is) or +

Let us try it on the string int + (int) + (int)!

|int + (int) + (int)$ [0, shift]
int | + (int) + (int)$ [01, reduce using 2]
E | + (int) + (int)$ [02, shift]
E +| (int) + (int)$ [023, shift]
E + (|int) + (int)$ [0234, shift]
E + (int|) + (int)$ [02345, reduce using 2]
E + (E|) + (int)$ [02346, shift]
E + (E)| + (int)$ [023467, reduce using 1]
E| + (int)$ [02, shift]
E+| (int)$ [023, shift]
E+(|int)$ [0234, shift]
E+(int|)$ [02345, reduce using 1]
E+(E|)$ [02346, shift]
E+(E)|$ [023467, reduce using 1]
E|$ [02, accept]

That works, first try, and did not have to branch and backtrack either!



State	Parsing action
0	Shift
1	Reduce using 2 if next is $ or +
2	Accept if next is $
3	Shift
4	Shift
5	Reduce using 2 if next is) or +
6	Shift
7	Reduce using 1 if next is $ or +
8	Shift
9	Shift
10	Shift
11	Reduce using 1 is next is) or +

Practice: What happens with int + (int + (int))?

How about int + int + (int)?

Answer: To be shown on board.

That is very powerful!

- This parser, whose name would be LR(1) according to our naming convention from earlier,…
- …Definitely works for that CFG!

Several follow-up questions, obviously:

1. How did we come up with the FSM and the FSM state to parsing actions mapping for our LR(1) parser?
2. Does the CFG need to have special properties, again, for this to algorithm to work without issues?

Addressing the questions

Does the CFG need to have special properties, again, for this to work?

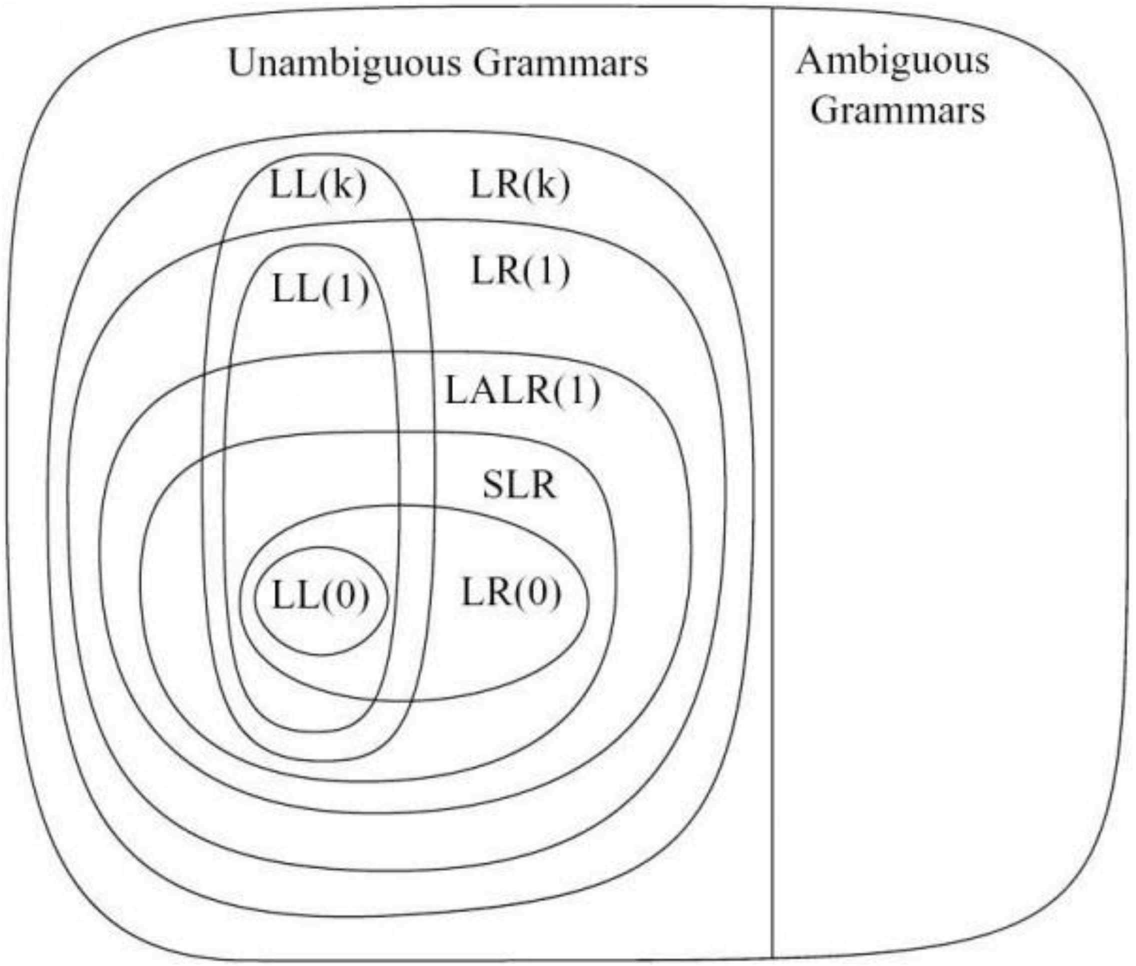
- Yes, the CFG needs to have certain properties, which will appear when we start building the FSM following from that CFG.
- We say that the **CFG is LR(1)**, if such an LR(1) parser can be used on it.
- **Good news #1:** LR(1) is a stronger type than LL(1). This means our LR(1) parser can cover more CFGs than LL(1)!
- **Good news #2:** LR(1) is a very large type for CFGs, and most programming languages CFGs are LR(1)!
  (For the other programming languages that are not LR(1), this means that more advanced parsers are required!)

Addressing the questions

LR(1) is a stronger type than LL(1). This means our LR(1) parser can cover more CFGs than LL(1)!

Also, did not mention, but LL(k) and LR(k) grammars are necessarily non-ambiguous!

Ambiguous grammars are problematic and should not be considered for programming languages anyway…
(Refer to W9S2 for reasons!).



Addressing the questions

But how did we come up with the FSM and the FSM states to parsing actions mapping for our LR(1) parser?

- Figuring out the FSM hiding behind a given CFG will be the not-so-easy part!
- Typically done by **parsing generators (e.g. bison, ANTLR, etc.)**.
- A typical parsing generator will look at a CFG and build a non-deterministic FSM first, eventually reducing it to a deterministic FSM later. This is tedious to explain, and very time-consuming… *(Can suggest additional reading if interested!)*
- Take it slow if you plan to try it on paper, it is a difficult/tedious task!

Production tracking in production rules

Definition (The point symbol in LR(1)):

The **point symbol (.)** in the context of LR(1) parsing is used to represent the **current position in the RHS of a production rule, while parsing a given input**.

It helps us **keep track of how much of the production rule has been seen and what has yet to be seen before we can use it to reduce.**

In simple terms, the dot (.) separates the part of the production rule that has been recognized (to the left of the dot) from the part that still needs to be parsed and seen via shifting (to the right of the dot).

Production tracking in production rules

Consider the production rule A $\rightarrow$ BCD as an example.

- If we place the dot at the beginning of the rule, which represented as the production rule A $\rightarrow$ .BCD with the dot in position 0,
- Then, this indicates that none of the right-hand side elements of the production rule have been recognized yet.

As the parser progresses/shifts and recognizes more symbols, the dot might move to the right:

- A $\rightarrow$ B.CD (after seeing B)
- A $\rightarrow$ BC.D (after seeing BC)
- A $\rightarrow$ BCD. (after seeing BCD)

This dot describes the current state of the parser: A $\rightarrow$ B.CD is then equivalent to our parser being in a state $\alpha B | \beta$ of some sort!

Production tracking in production rules

- When the dot is at the beginning or in the middle of a production rule (i.e. A → .BCD, A → B.CD, or A → BC.D), **we have not yet recognized enough symbols to be sure we can reduce some elements of our string into A using this production rule**.
- When the dot reaches the end of the production rule (i.e. A -> BCD.), it means the entire rule has been recognized, and the **parser <u>could</u> technically choose to perform a reduce action based on that production rule**.

Production tracking in production rules

An important theorem:

- Any production rule necessarily has a finite number of elements on the right-hand side. This means that **there is a limited number of possibilities for the dot to be at** for any given production rule.

For instance, the production rule A → BCD has four possibilities in terms of dot positions, being:

- *A → .BCD*
- *A → B.CD*
- *A → BC.D*
- *A → BCD.*

Production tracking in production rules

An important theorem:

- Any production rule necessarily has a finite number of elements on the right-hand side. This means that **there is a limited number of possibilities for the dot to be at** for any given production rule.
- But **our CFGs also have a finite number of production rules**.

For instance, the CFG below has four possible production rules.

$$E \rightarrow T,$$
$$E \rightarrow E + T,$$
$$T \rightarrow (E),$$
$$T \rightarrow int$$

Production tracking in production rules

An important theorem:

- Any production rule necessarily has a finite number of elements on the right-hand side. This means that **there is a limited number of possibilities for the dot to be at** for any given production rule.
- But **our CFGs also have a finite number of production rules**.

Important result: Combining these two pieces of information together means that **there is a finite number of possible dot representation for all the production rules of our CFG**.

→ And these could be used as **states for an FSM of some sort!**

A finite number of dot representations

For instance, the CFG below has four possible production rules.

$$E \rightarrow T,$$

$$E \rightarrow E + T,$$

$$T \rightarrow (E),$$

$$T \rightarrow int$$

According to our theorem, it means that there should be a finite number of dot representations for the production rules of that CFG.

That is shown below.

(**Note:** Convention to also add the dot representations for the start symbol of our CFG, which is *E*).

- *.E, E.,*
- *.T, T.,*
- *.E+T, E.+T, E+.T, E+T.,*
- *.(E), (.E), (E.), (E).,*
- *.int, int.*

From LR(0) to LR(1) items

Definition (LR(0) items for a given CFG):

For a given CFG, **all the possible right hand side formulas of all the production rules and their possible variations in terms of dot positions**, produce what is commonly referred to as the **LR(0) items** of the CFG. In the case of our previous CFG, the LR(0) items are simply:

- *.E, E.,*
- *.T, T.,*
- *.E+T, E.+T, E+.T, E+T.,*
- *.(E), (.E), (E.), (E).,*
- *.int, int.*

From LR(0) to LR(1) items

Definition (LR(1) items for a given CFG):

For a given CFG, the LR(1) items are the set of all possible tuples (A → α.Bβ, a), where

- A is a non-terminal symbol,
- α and β are strings of terminal and nonterminal symbols,
- B is a nonterminal symbol,
- and a is a lookahead symbol (all possible terminal symbols in the CFG or the end of string $ symbol).

For instance, (E→E+.T, '(') is a possible **LR(1) item**, that describes that the parser is in a configuration αE+|(β of some sort.

From LR(0) to LR(1) items

Some examples of LR(1) items for our CFG are then:

- **(E → .T, +):** This LR(1) item indicates that we are starting to parse the non-terminal E, and we have yet to see a T followed by a + symbol.
- **(E → .T,)):** This LR(1) item indicates that we are parsing the non-terminal E, and we have yet to see a T followed by a closing parenthesis.
- **(E → E +. T, +):** This LR(1) item indicates that we are parsing the non-terminal E, and we have already seen some prefix of the form "E + ..." that is now on the left of the |,and we have yet to see a T to be able to reduce, and that the element on the right-side of | is a + symbol.

Important: there is a finite number of LR(1) items in any given CFG.

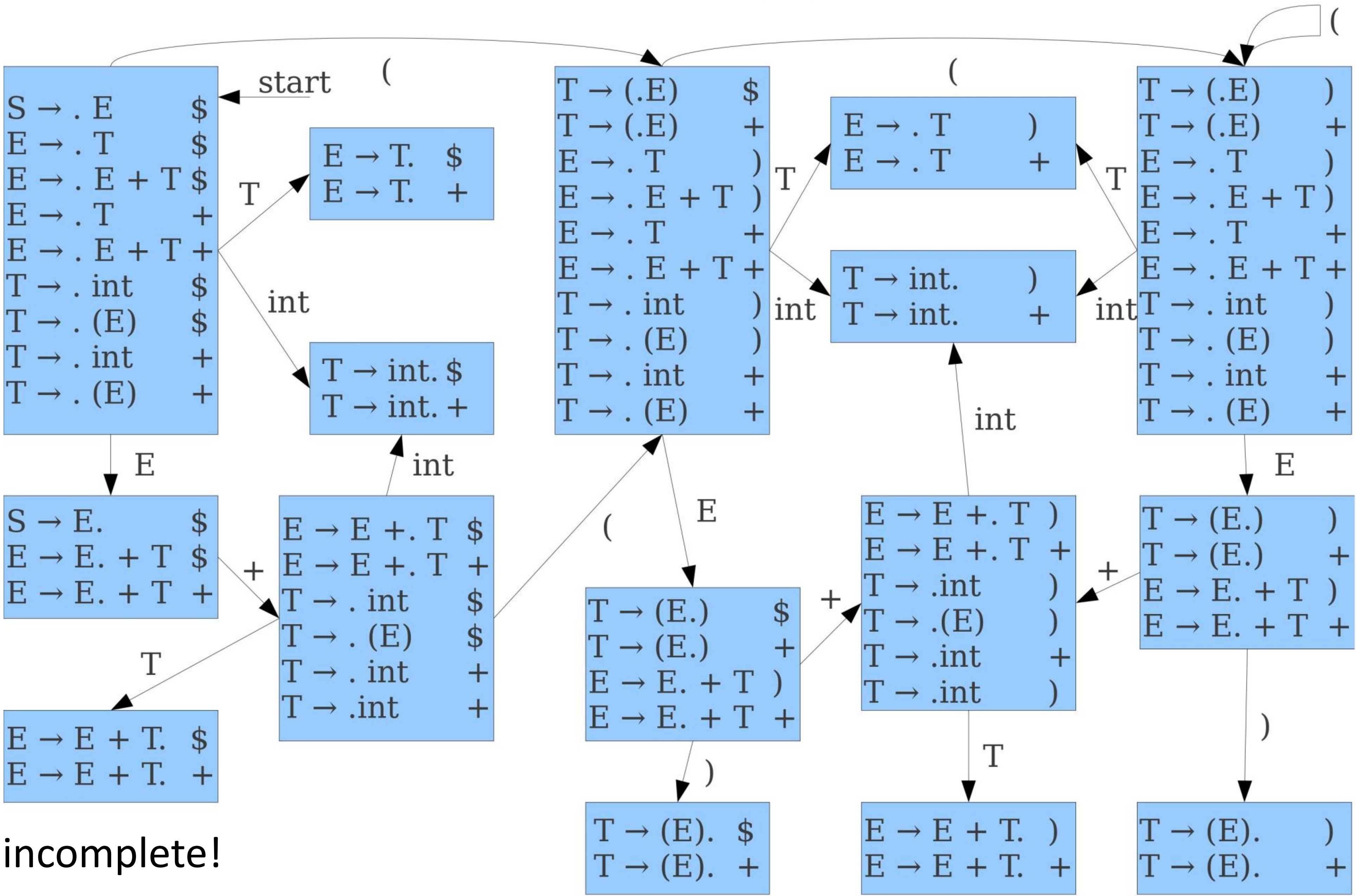
Building an FSM: states

- Let us use these **LR(1) items** as states **for an FSM of some sort!**
- How about the **input string** to be used for that FSM?
  As before, let us use the current left string in our parser stack.
- How about our FSM **actions**? As before, could consist of any terminal or non-terminal symbol in our CFG.

- How to establish a **transition logic** for our FSM then?

(Next slide shows an incomplete version of LR(1) FSM!
Be prepared, it is actually quite ugly!)

Tedious, not fun, but technically possible

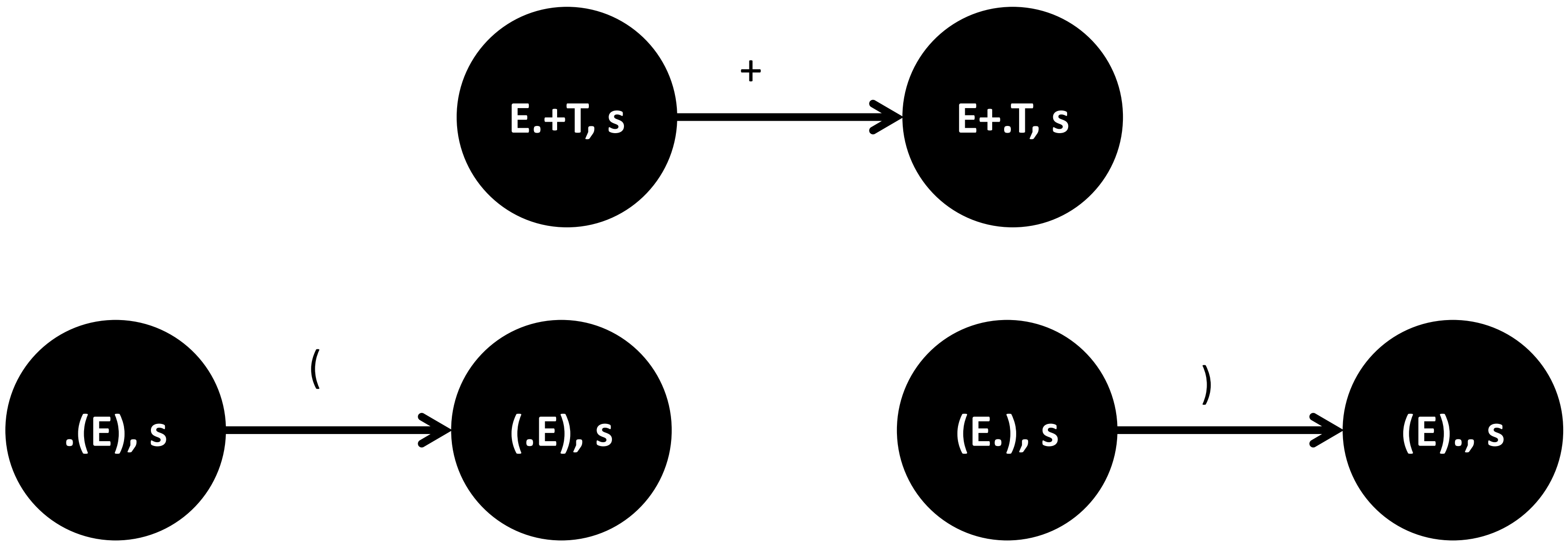


Note: it is incomplete!

Transition logic rules: Rule #1

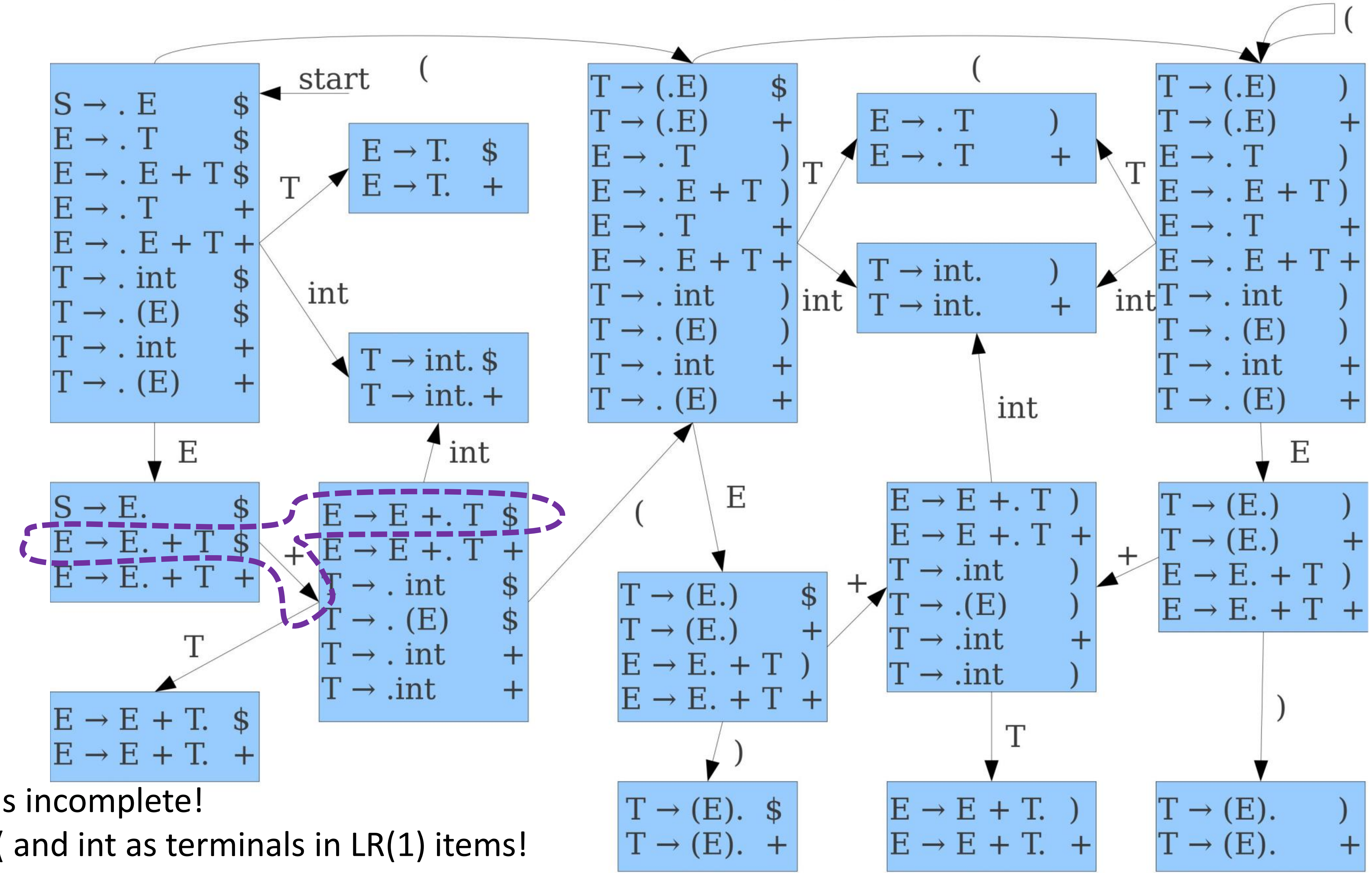
Rule #1, Shift: If there is an LR(1) item of the form (A → α.Xβ, s) used as the current state and ***X* is a terminal symbol**, add a transition to the state containing the item (A → αX.β, s) using the symbol X.

Using Rule #1, connect the following state



Repeat for all possible pairs that satisfy the rule #1 condition!

Tedious, not fun, but technically possible



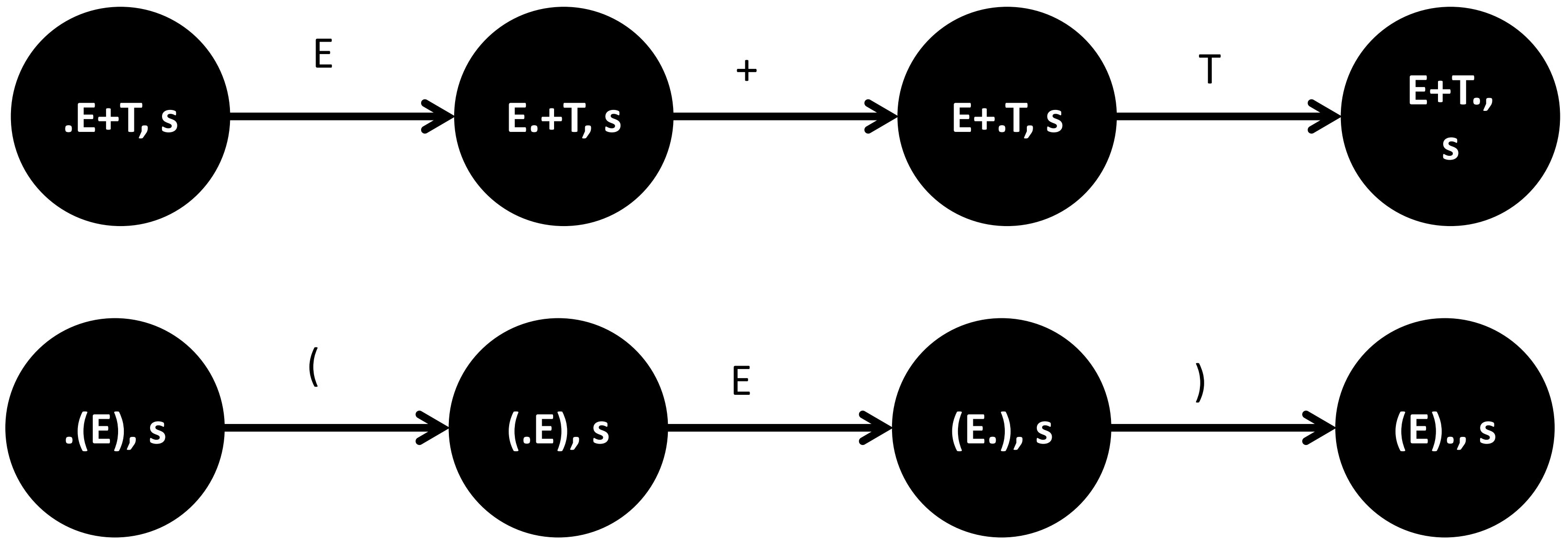
Note: it is incomplete!
Missing (and int as terminals in LR(1) items!

Transition logic rules: Rule #1&2

Rule #1, Shift transition: If there is an LR(1) item of the form (A → α.Xβ, s) used as the current state and X **is a terminal symbol**, add a transition to the state containing the item (A → αX.β, s) using the symbol X.

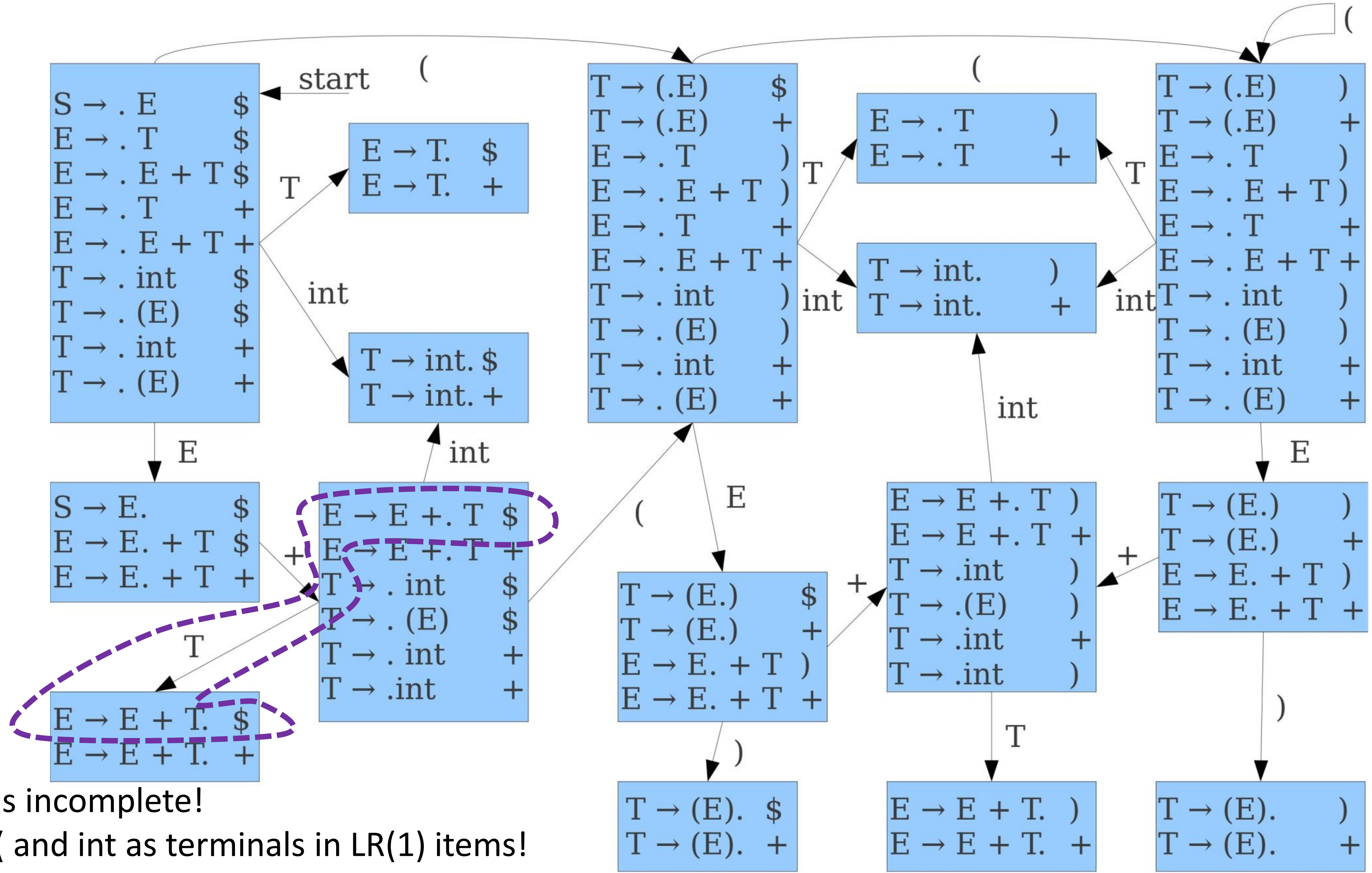
Rule #2, GoTo transition: If there is an LR(1) item of the form (A → α.Xβ, s) used as the current state and X **is a non-terminal symbol this time**, add a transition to the state containing the item (A → αX.β, s) on the symbol X. *(Roughly similar)*

Transition logic rules: Rule #2



Repeat for all possible pairs that satisfy the rule #1 and rule #2 condition!

Tedious, not fun, but technically possible



Note: it is incomplete!
Missing (and int as terminals in LR(1) items!

Transition logic rules: Rule #3

Rule #3, Closure rule: If there is an LR(1) item of the form (A → α.Xβ, s) used as the current state and $\boldsymbol{X}$ **is a non-terminal**, for each production X → γ and each terminal a in FIRST*(βs), add the item (X → .γ, a) to the current state if it is not already there.

This means that some states in our FSM will combine together!

Transition logic rules: Rule #3

Rule #3, Closure rule: If there is an LR(1) item of the form (A → α.Xβ, s) used as the current state and ***X* is a non-terminal**, for each production X → γ and each terminal a in FIRST*(βs), add the item (X → .γ, a) to the current state if it is not already there.

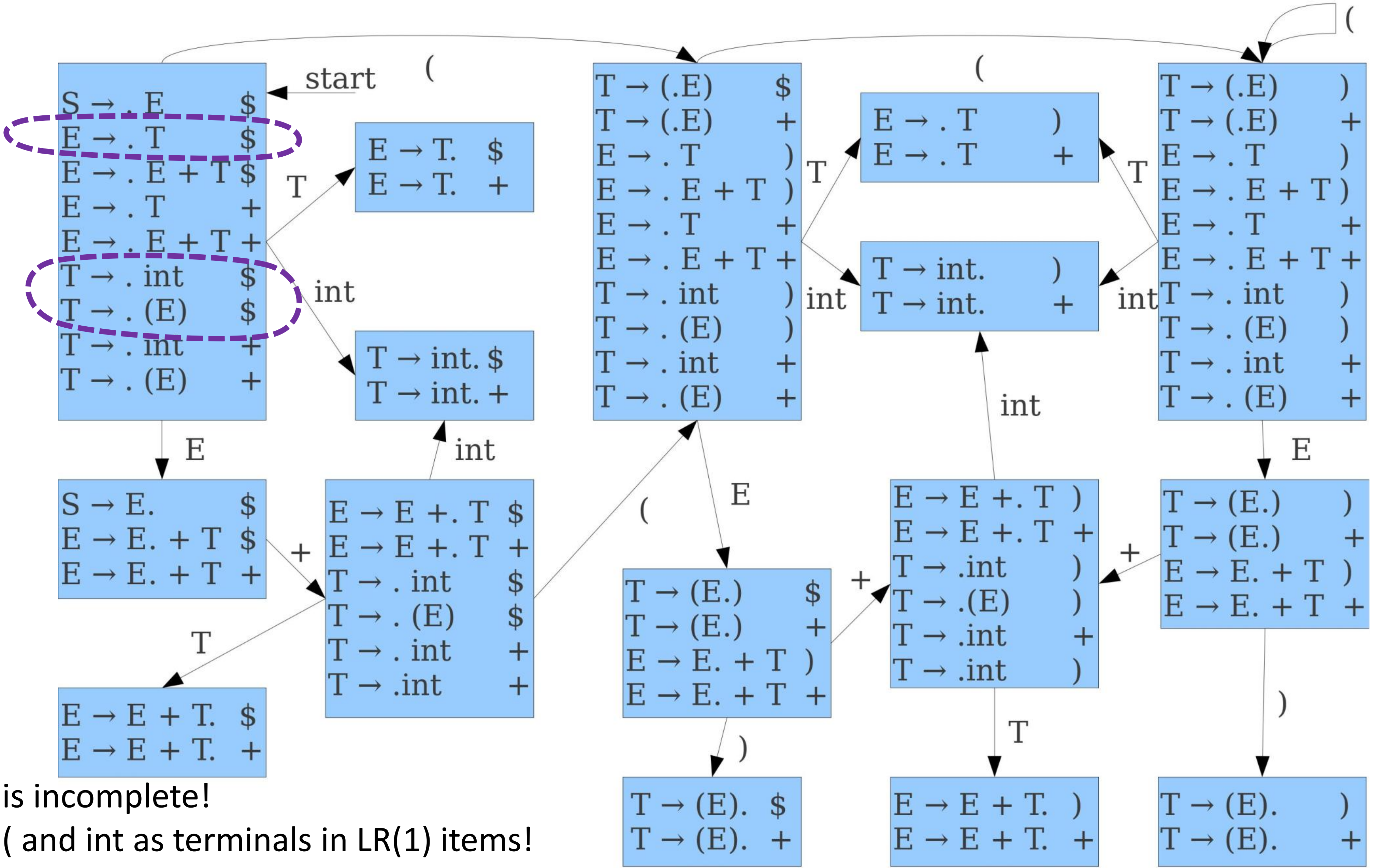
For instance: Consider the LR(1) item (E → .T, $).

Here, α = empty string, X = T, and β = empty string and s = $.
We have FIRST($) = {$}.

We will look for all LR(1) items having productions with T on the left hand side, and apply the dot on the leftmost position of the left-hand side. That is only two possibilities: (T → .int, $) and (T → .(E), $).

Rule #3 merges these LR(1) items into to the same state as (E → .T, $).

Tedious, not fun, but technically possible



Note: it is incomplete!
Missing (and int as terminals in LR(1) items!

Transition logic rules: Rule #4

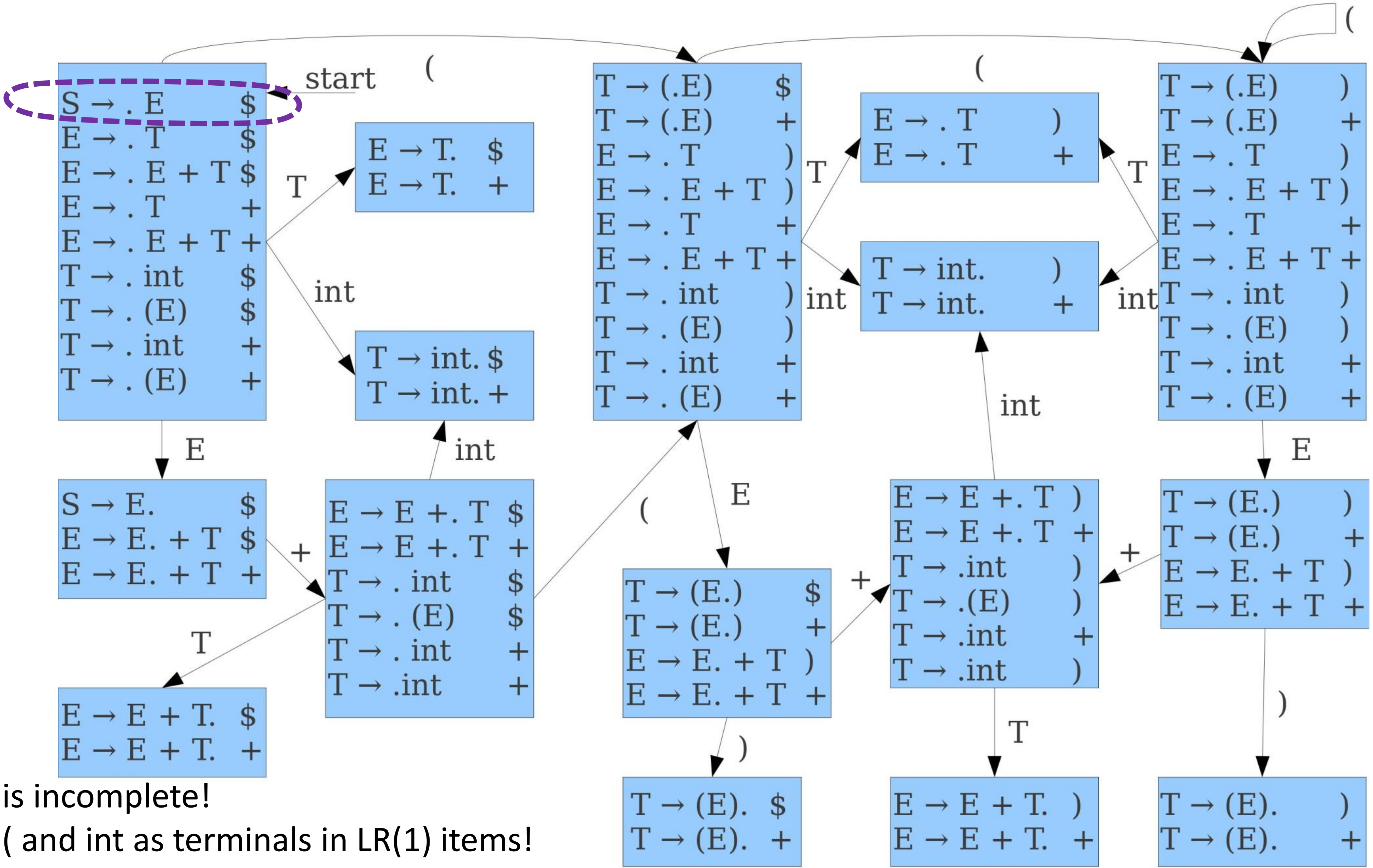
Repeat rules #1, #2 and #3 on all LR(1) items, resolving all possible connections and merging and you will obtain the "simplest" version of the FSM to use for your LR(1) parser!

(Obviously, not implementing that!)

Rule #4, Start State rule: The start state of the LR(1) parsing FSM will be the one that contains the item (S → . E, $), where S is the start symbol and $ is the end-of-input marker.

(Note: it might require that you rewrite your CFG so that it only has one production rule using the start symbol S).

Tedious, not fun, but technically possible



Note: it is incomplete!
Missing (and int as terminals in LR(1) items!

Getting there!

Ok, so now, it means that we have all it takes to produce a (massive) **deterministic FSM.**

(Implementing and automating that would be a nightmare though, so I am somewhat glad we are not going to do it!)

- This FSM is called the **LR(1) finite state automata** for the CFG and will be **used by the LR(1) parser**, in an almost identical way as shown earlier.
- The LR(1) will now run as follows, using one of four possible actions, being: shift, reduce using a production rule of some sort, accept, reject.
- **Our last step is then to assign a parser action to every FSM state!**

Lifting the final shift-reduce conflict

Our last step is then to assign a parser action to every FSM state!

1. **Shift**: If the FSM stops in a state containing an LR(1) item of the form (A → α.Xβ, s) and the next input symbol is X (a terminal symbol), the parser will shift.
   This will move the input symbol X onto the stack.

Lifting the final shift-reduce conflict

Our last step is then to assign a parser action to every FSM state!

2. **Reduce**: If the parser is in a state containing an LR(1) item of the form (A → α. , s) and the next input symbol on the right side of the separator | matches the lookahead symbol s…
   Then the parser will perform a reduction by using the production rule A → α in reverse and will update the stack on the left side of |.

Lifting the final shift-reduce conflict

Our last step is then to assign a parser action to every FSM state!

3. **Accept**: If the FSM stops in the state containing an LR(1) item of the form (S → E. , $) and the next input symbol after | is $ (the end-of-input marker), the LR(1) parser will accept the input string as valid.

The produced derivation is then our answer!

Lifting the final shift-reduce conflict

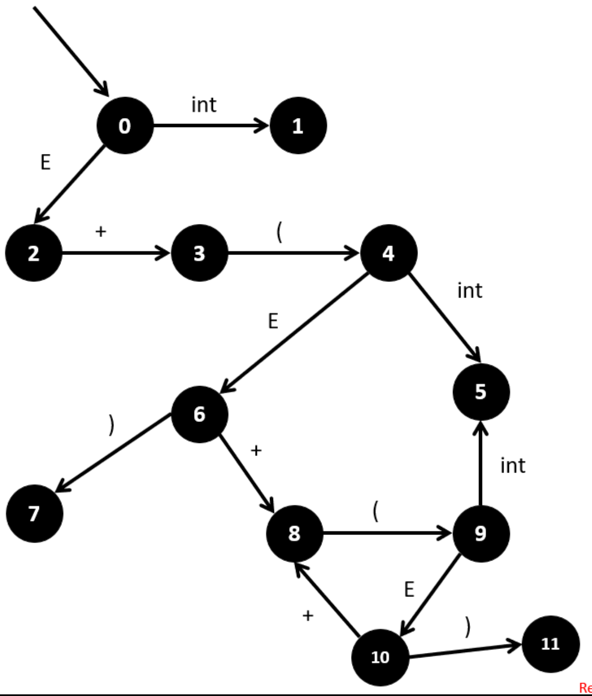
Our last step is then to assign a parser action to every FSM state!

4. **Error**: If the parser encounters an input symbol that does not correspond to any valid transition in the FSM, it will report a parsing/syntax error.

Lifting the final shift-reduce conflict

After that, rename all states as simple indexes {0, 1, 2, … N}, and put the parsing actions into a table, and tada!

Can this be automated? Yes, but damn tedious.



State	Parsing action
0	Shift
1	Reduce using 2 if next is $ or +
2	Accept if next is $
3	Shift
4	Shift
5	Reduce using 2 if next is) or +
6	Shift
7	Reduce using 1 if next is $ or +
8	Shift
9	Shift
10	Shift
11	Reduce using 1 is next is) or +

Note: shift if reduce condition is not met, if the next symbol after | is not listed. For instance, on state 1, we use reduce if the element after | is $ or +; and shift otherwise.

Restricted

But wait…

According to our previous definitions, LR(1) will always attempt to reduce whenever possible?

But, technically, it feels like I could decide to reduce, or I could decide to shift some more symbols and reduce later, maybe even using a different production rule…!

I feel that this could still be a problem for my LR(1) FSM in certain CFGs!

Question: In this LR(1), are there scenarios where we should postpone an opportunity to reduce, shift some more, and then reduce using a different production rule?

A quick word about conflicts

Definition (Shift-Reduce conflicts):

A **shift-reduce conflict** occurs in a parsing situation **when the LR(1) FSM has a state for which we cannot decide whether to shift the next input symbol or reduce by applying a production rule**.

This **ambiguity** can lead to different parse trees for the same (syntactically valid) input string.

When such a conflict exists, we say that the CFG is not LR(1), meaning that it cannot be unambiguously parsed using an LR(1) parser.

In those cases, more advanced parsers, like LR(k), must be used (out-of-scope though!)

A quick word about conflicts

Definition (Reduce-Reduce conflicts):

A **reduce-reduce conflict** occurs if **the LR(1) FSM has a state, where we could use two or more reductions with different production rules**.

It means that the parser cannot unambiguously decide which production rule to apply for reducing the current input.

When such a conflict exists, we say that the CFG is not LR(1), meaning that it cannot be unambiguously parsed using an LR(1) parser.

In those cases, more advanced parsers, like LR(k), must be used (<u>out-of-scope</u> though!)

But wait…

Question: Are there scenarios where we should postpone an opportunity to reduce, shift some more, and then reduce using a different production rule?

Or in other words, are all CFGs immune to Shift-reduce and Reduce-reduce conflicts?

Answer: If the CFG has Shift-reduce and Reduce-reduce conflicts, then we simply say that it is not LR(1). If the CFG is LR(1), no need to worry about that, reduce whenever you can and no conflicts will arise.

An important question

Erm, will sound like a stupid question, but is our CFG actually LR(1)?

$$E \rightarrow T,$$

$$E \rightarrow E + T,$$

$$T \rightarrow (E),$$

$$T \rightarrow int$$

An important question

Erm, will sound like a stupid question, but is our CFG actually LR(1)?

$$E \rightarrow T,$$
$$E \rightarrow E + T,$$
$$T \rightarrow (E),$$
$$T \rightarrow int$$

Yes, it is!
Phew…!

(Try it, for extra practice?)

An important question

Ok, how about the YACC CFG that you have shown us for the C programming language in a previous lecture? Is it LR(1)?

(http://www.lysator.liu.se/c/ANSI-C-grammar-y.html)

…

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;
```

An important question

Ok, how about the YACC CFG that you have shown us for the C programming language in a previous lecture? Is it LR(1)?

(http://www.lysator.liu.se/c/ANSI-C-grammar-y.html)

Yes, it is!


```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
	: IDENTIFIER
	| CONSTANT
	| STRING_LITERAL
	| '(' expression ')'
	;

postfix_expression
	: primary_expression
	| postfix_expression '[' expression ']'
	| postfix_expression '(' ')'
	| postfix_expression '(' argument_expression_list ')'
	| postfix_expression '.' IDENTIFIER
	| postfix_expression PTR_OP IDENTIFIER
	| postfix_expression INC_OP
	| postfix_expression DEC_OP
	;

argument_expression_list
	: assignment_expression
	| argument_expression_list ',' assignment_expression
	;

unary_expression
	: postfix_expression
	| INC_OP unary_expression
	| DEC_OP unary_expression
	| unary_operator cast_expression
	| SIZEOF unary_expression
	| SIZEOF '(' type_name ')'
	;

unary_operator
	: '&'
	| '*'
	| '+'
	| '-'
	| '~'
	| '!'
	;

cast_expression
	: unary_expression
	| '(' type_name ')' cast_expression
	;
```

An important question

Ok, how about the YACC CFG that you have shown us for the C programming language in a previous lecture? Is it LR(1)?

(http://www.lysator.liu.se/c/ANSI-C-grammar-y.html)

Yes, it is!

This means an LR(1) parser will work nicely for a C compiler!

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;
```

An important question

How about C++?

Well… The CFG of C++ is <u>not</u> LR(1).
This means we need a more advanced type of parser.

As a matter of fact, the C++ grammar is context-sensitive in certain places (e.g., due to **the "most vexing parse" problem*).**

This makes pure CFG-based parsing insufficient (Ouch!).

(But it is <u>out-of-scope</u>).

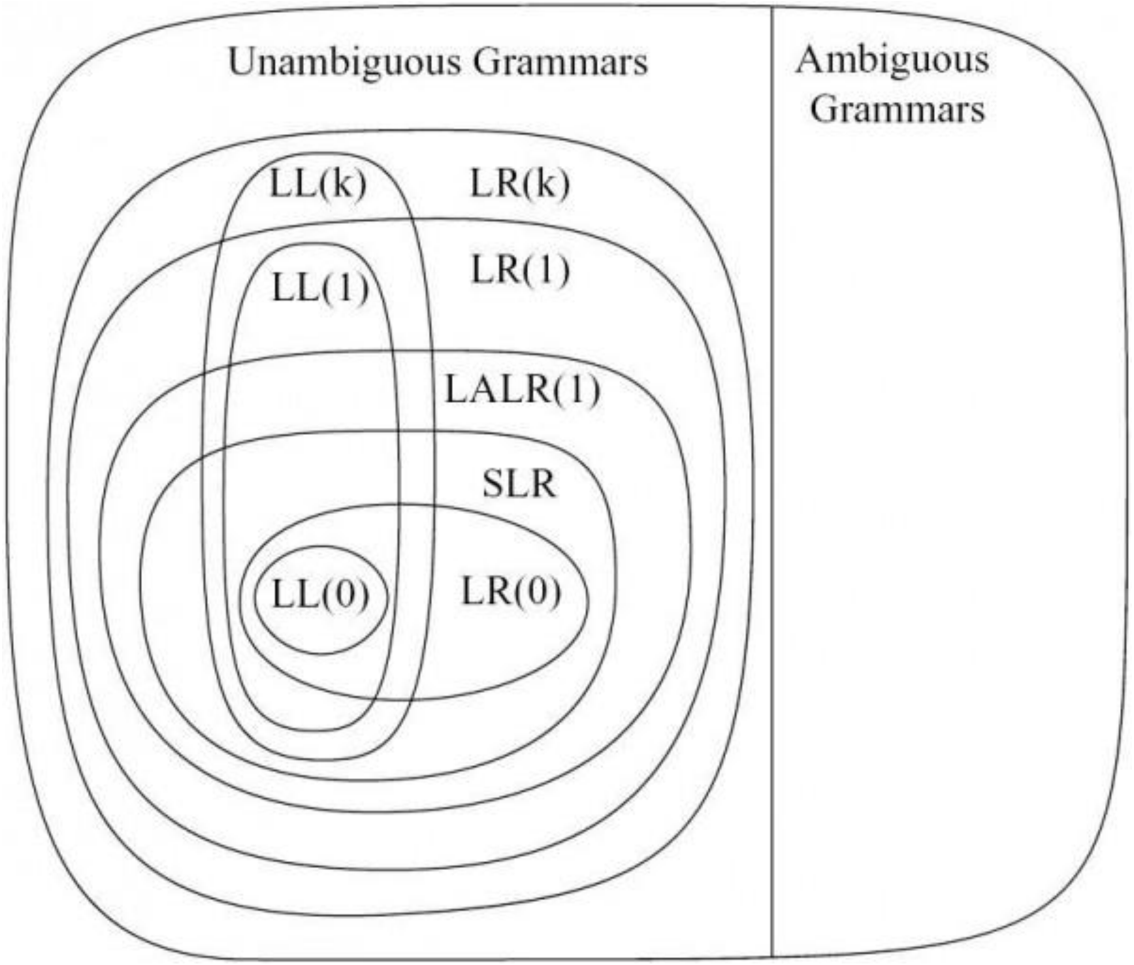
*See https://www.youtube.com/watch?v=ByKf_foSlXY

Addressing the questions

As we have mentioned, there are additional algorithms to investigate if LR(1) does not work for your CFG!

And there is more stuff beyond CFGs (especially if the grammar is NOT context-free).

(This is definitely <u>out-of-scope</u> for this class, though!)



An important question

How about Python then?

Since v3.9, Python is relying on a different and simpler parser, called PEG, which belong to the class of LL(1).

This means you can use an even simpler parser than LR(1), to do the CFG parsing job!

(But it is also out-of-scope).

Conclusion

- Top-Down parsing was inconclusive but gave us good insights.
- The C programming language is LR(1), we can use a bottom-up parser for the compiler of that language.
- Many other languages might not be LR(1) and might require more advanced classes of parsers, which are out-of-scope.
- The implementation of said parser, while interesting, is typically out-of-scope.

Conclusion

So, here is where we are.

- If our source code passes the lexical analysis and tokenizes with no trouble,
- And if there exist a derivation for the stream of tokens, that fits the YACC CFG,
- Then the code is also valid in terms of lexica and syntax (big result!).

This is not over yet, as there are still some errors that could technically make the source code malfunction, but 90% of the errors have been covered so far!

Next, semantic analysis!

Conclusion

- If interested to learn more about parsers, the reference course is the one from Stanford

https://web.stanford.edu/class/cs143/

- Available for free online and comes with video recordings

https://www.edx.org/course/compilers