Interpretation and Compilation of Languages

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

April 5, 2025

Lecture 5

based on lectures by Jean-Christophe Filliâtre and Léon Gondelman

previous editions by João Costa Seco, Luís Caires, and Bernardo Toninho

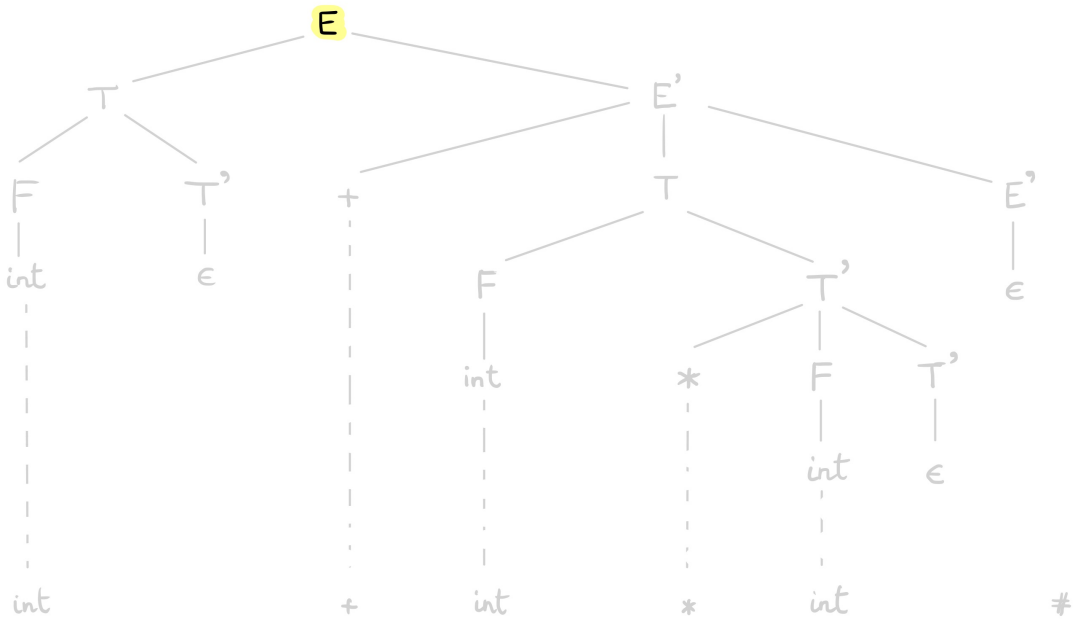
context-free grammars

- derivations (leftmost, rightmost)
- derivation trees
- ambiguities

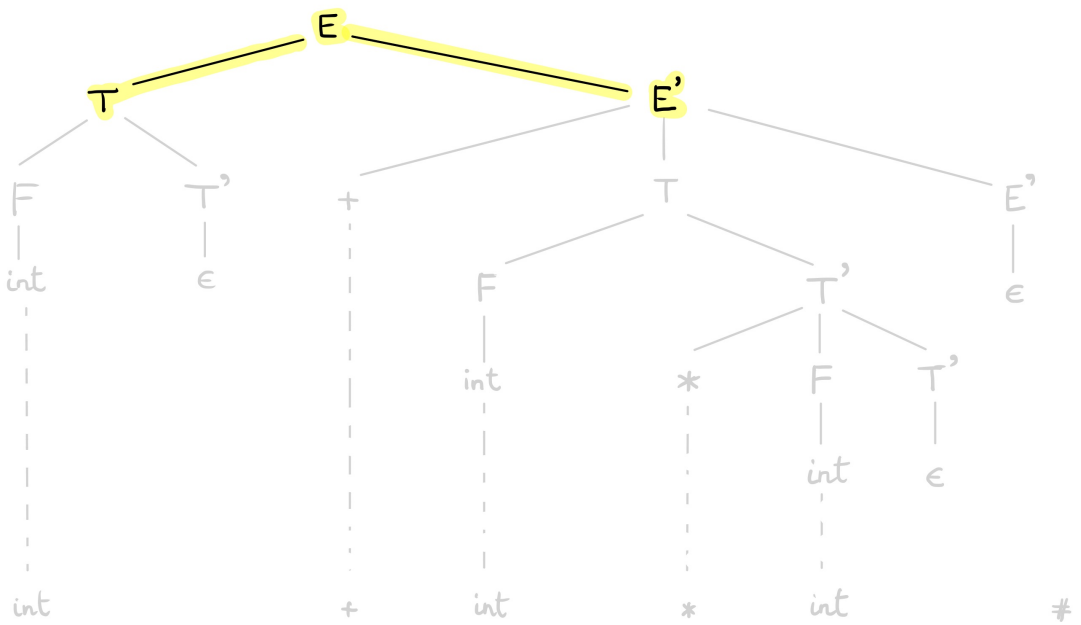
top-down parsing

- push-down automaton with a stack containing $(T \cup N)^\star$
- parsing input from left to right
- constructing derivation tree w.r.t. the leftmost derivation
- computing sets (null, first, follow) using fixpoint

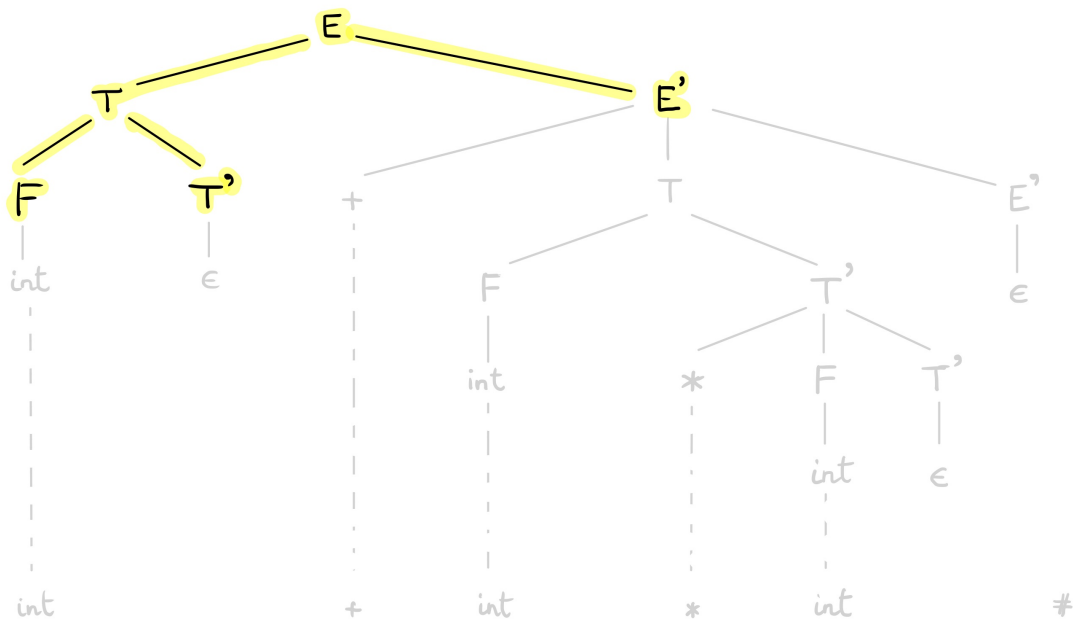
stack $\mapsto$	input
E	int+int*int#
$E'T$	int+int*int#
$E'T'F$	int+int*int#
$E'T'$int	int+int*int#
$E'T'$	+int*int#
E'	+int*int#
$E'T$+	+int*int#
$E'T$	int*int#
$E'T'F$	int*int#
$E'T'$int	int*int#
$E'T'$	*int#
$E'T'F$*	*int#
$E'T'F$	int#
$E'T'$int	int#
$E'T'$	#
E'	#
ϵ	#



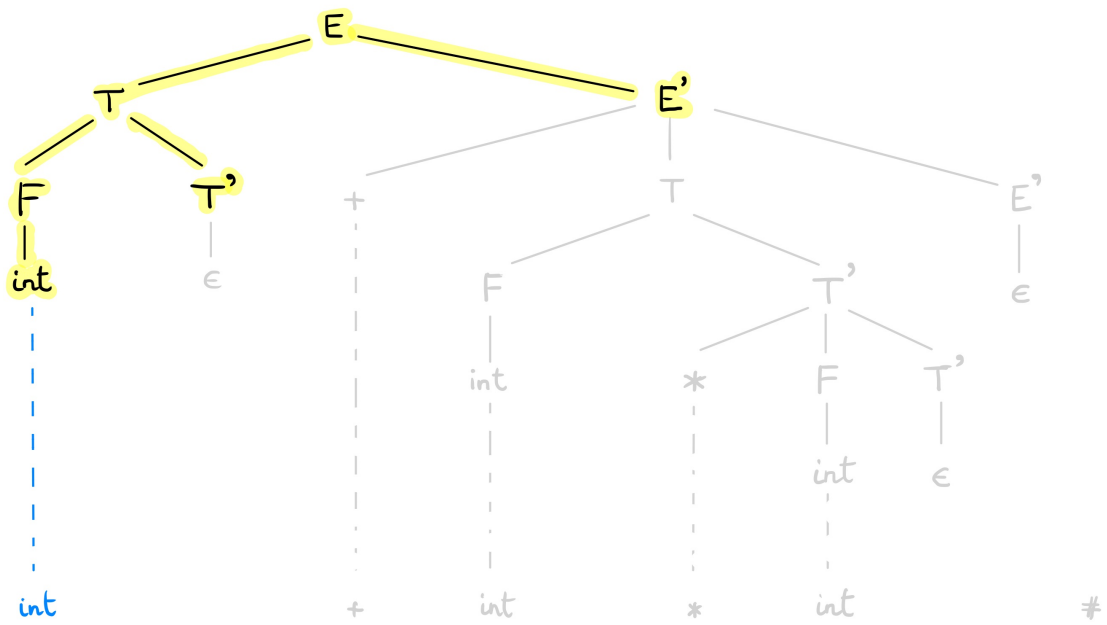
stack ↦	input
E	int+int*int#
$E'T$	int+int*int#
$E'T'F$	int+int*int#
$E'T'$int	int+int*int#
$E'T'$	+int*int#
E'	+int*int#
$E'T$+	+int*int#
$E'T$	int*int#
$E'T'F$	int*int#
$E'T'$int	int*int#
$E'T'$	*int#
$E'T'F$*	*int#
$E'T'F$	int#
$E'T'$int	int#
$E'T'$	#
E'	#
ϵ	#



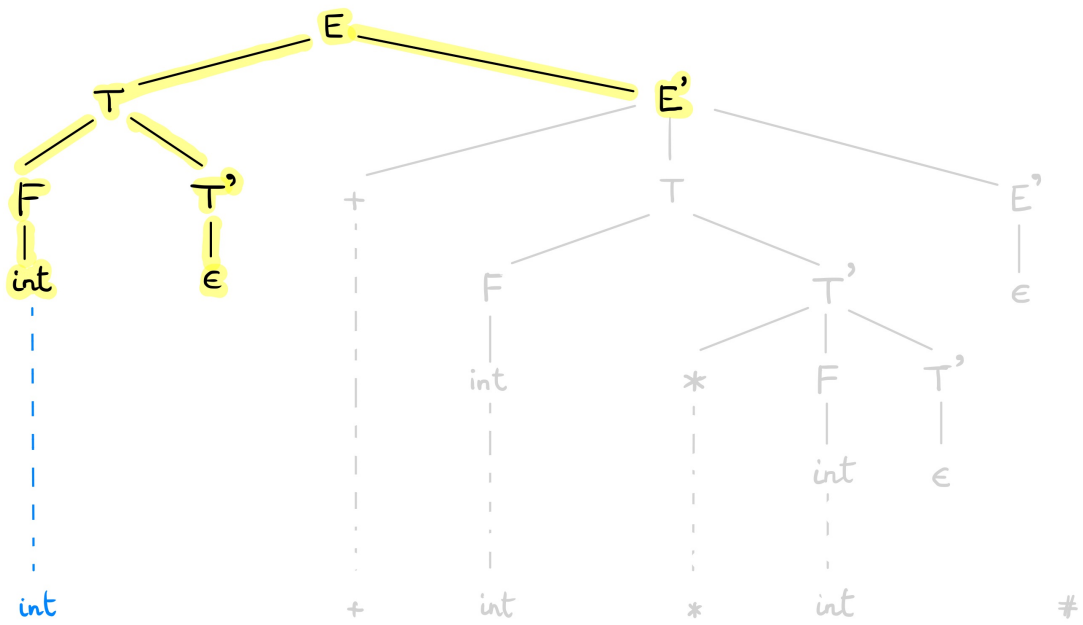
stack $\mapsto$	input
E	int+int*int#
$E'T$	int+int*int#
$E'T'F$	int+int*int#
$E'T'$int	int+int*int#
$E'T'$	+int*int#
E'	+int*int#
$E'T$+	+int*int#
$E'T$	int*int#
$E'T'F$	int*int#
$E'T'$int	int*int#
$E'T'$	*int#
$E'T'F$*	*int#
$E'T'F$	int#
$E'T'$int	int#
$E'T'$	#
E'	#
ϵ	#



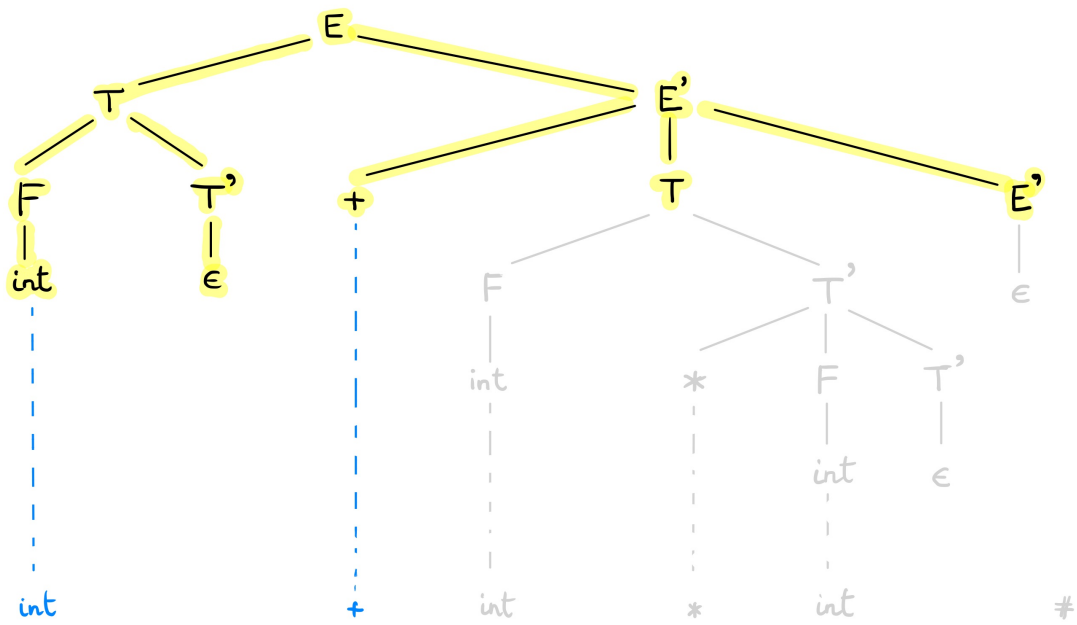
stack ↦	input
E	int+int*int#
$E'T$	int+int*int#
$E'T'F$	int+int*int#
$E'T'$int	int+int*int#
$E'T'$	+int*int#
E'	+int*int#
$E'T$+	+int*int#
$E'T$	int*int#
$E'T'F$	int*int#
$E'T'$int	int*int#
$E'T'$	*int#
$E'T'F$*	*int#
$E'T'F$	int#
$E'T'$int	int#
$E'T'$	#
E'	#
ϵ	#



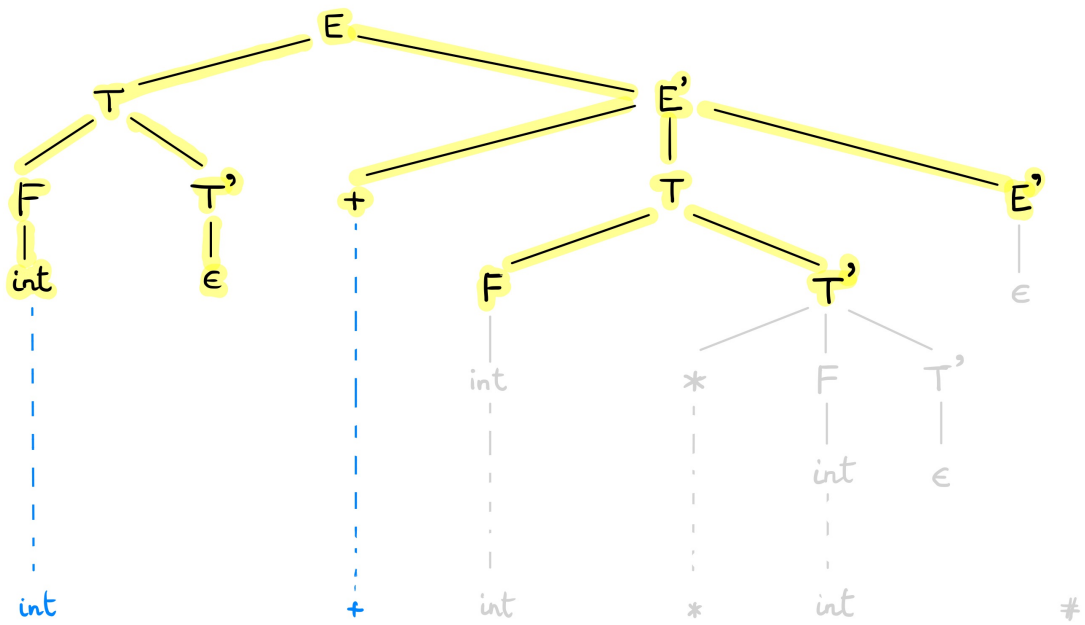
stack $\mapsto$	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T$`+`	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F$`*`	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	#
E'	#
ϵ	#



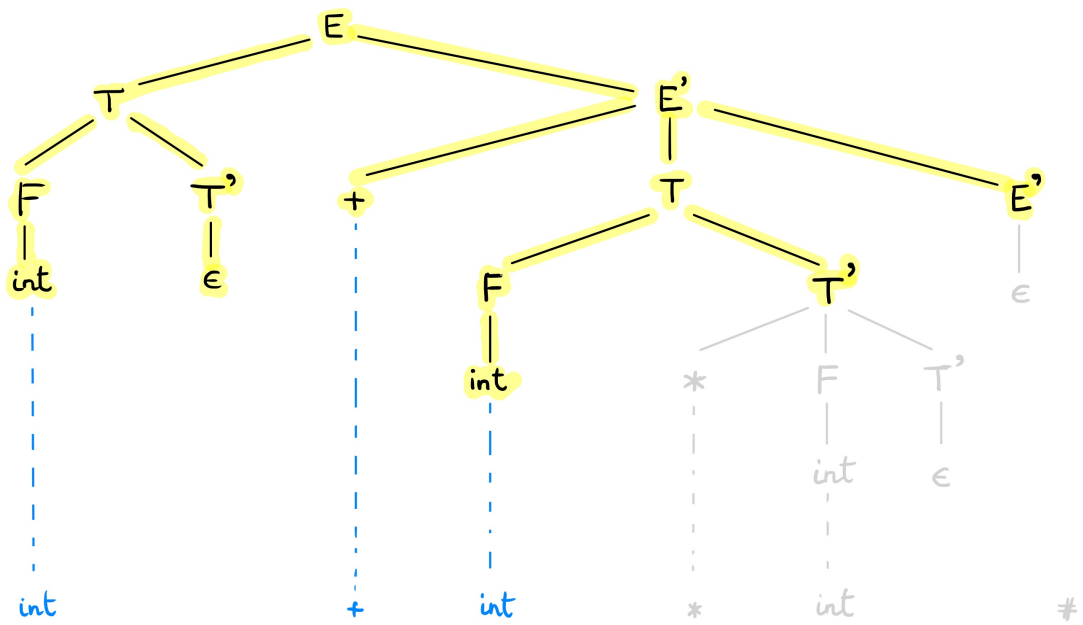
stack ↦	input
E	int+int*int#
$E'T$	int+int*int#
$E'T'F$	int+int*int#
$E'T'$int	int+int*int#
$E'T'$	+int*int#
E'	+int*int#
$E'T$+	+int*int#
$E'T$	int*int#
$E'T'F$	int*int#
$E'T'$int	int*int#
$E'T'$	*int#
$E'T'F$*	*int#
$E'T'F$	int#
$E'T'$int	int#
$E'T'$	#
E'	#
ϵ	#



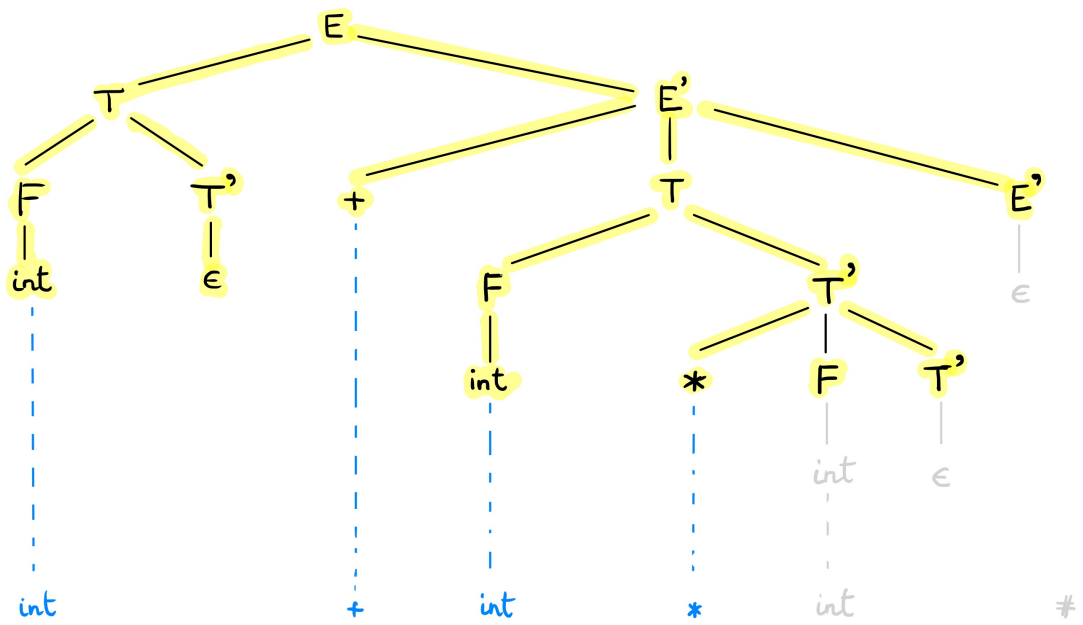
stack ↦	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T$`+`	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F$`*`	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	#
E'	#
ϵ	#



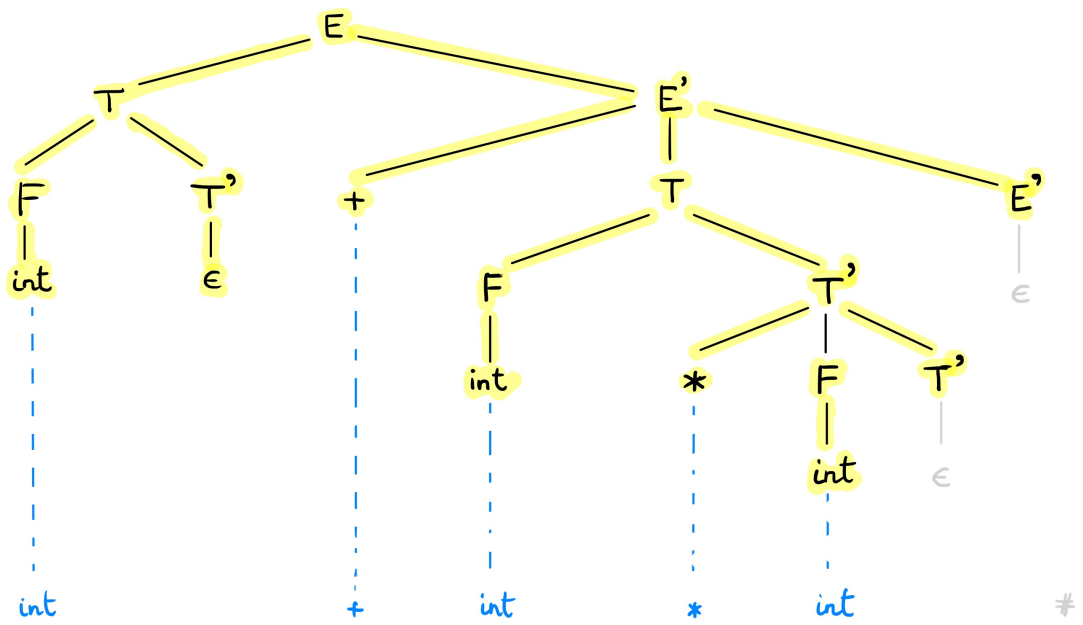
stack ↦	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T$`+`	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F$`*`	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	#
E'	#
ϵ	#



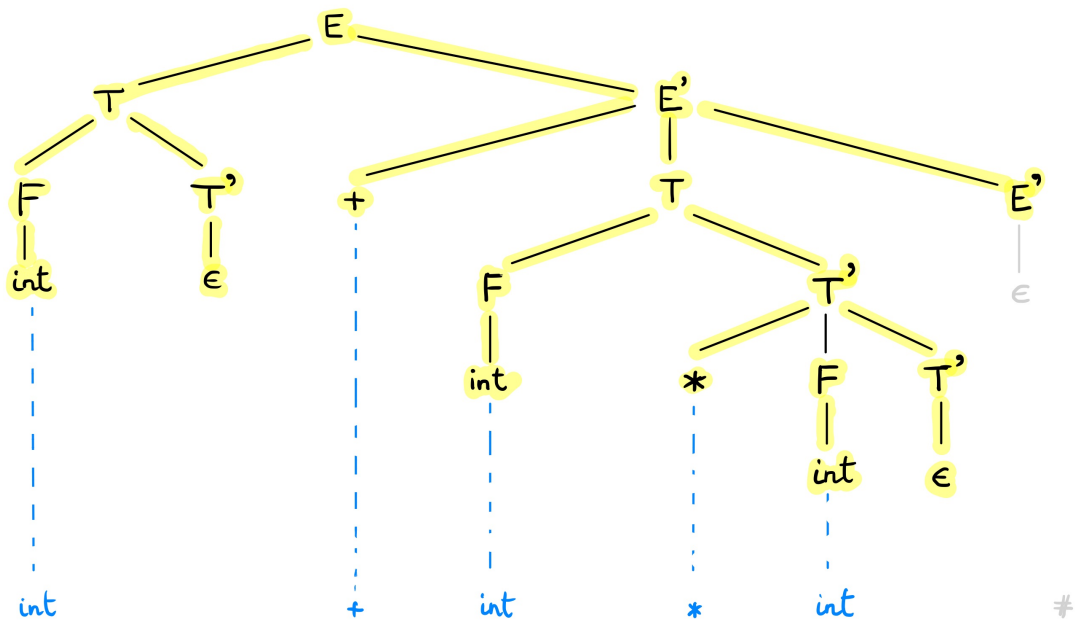
stack ↦	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T$`+`	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F$`*`	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	`#`
E'	`#`
ϵ	`#`



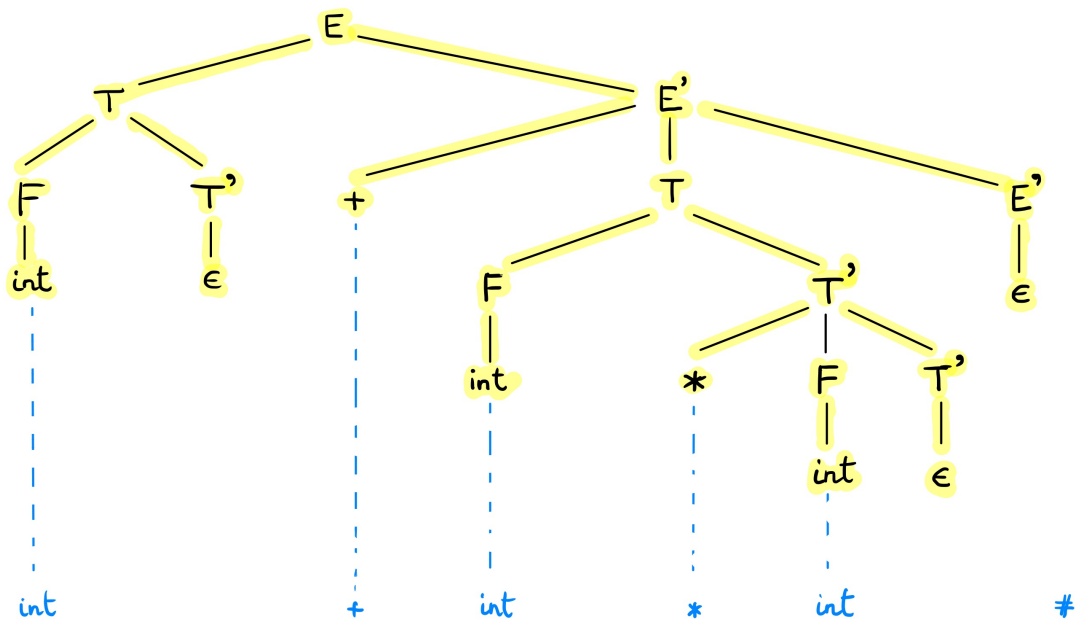
stack ↦	input
E	int+int*int#
$E'T$	int+int*int#
$E'T'F$	int+int*int#
$E'T'$int	int+int*int#
$E'T'$	+int*int#
E'	+int*int#
$E'T$+	+int*int#
$E'T$	int*int#
$E'T'F$	int*int#
$E'T'$int	int*int#
$E'T'$	*int#
$E'T'F$*	*int#
$E'T'F$	int#
$E'T'$int	int#
$E'T'$	#
E'	#
ϵ	#



stack ↦	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T$`+`	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F$`*`	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	`#`
E'	`#`
ϵ	`#`



stack ↦	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T$`+`	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F$`*`	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	`#`
E'	`#`
ϵ	`#`



Today: Syntactic Analysis (Parsing), part 2

1. bottom-up parsing
2. LR parsing algorithm
3. CUP tool
4. Menhir tool

Bottom-up Parsing

Main Idea

Scan the input from left to right.

Look for right-hand sides of production rules to build the derivation tree from bottom to top (*bottom-up parsing*).

Principle

The parser uses a stack that is a word of $(T \cup N)^\star$.

At each step, two actions can be performed

- a shift operation: we read a terminal from the input and we push it on the stack
- a reduce operation: the top of the stack is the right-hand side β of a production $X \to \beta$, and we replace β with X on the stack

Initially, the stack is empty.

When no more action can be performed, the input is recognized if it was read entirely and if the stack is limited to the axiom S.

Example 1

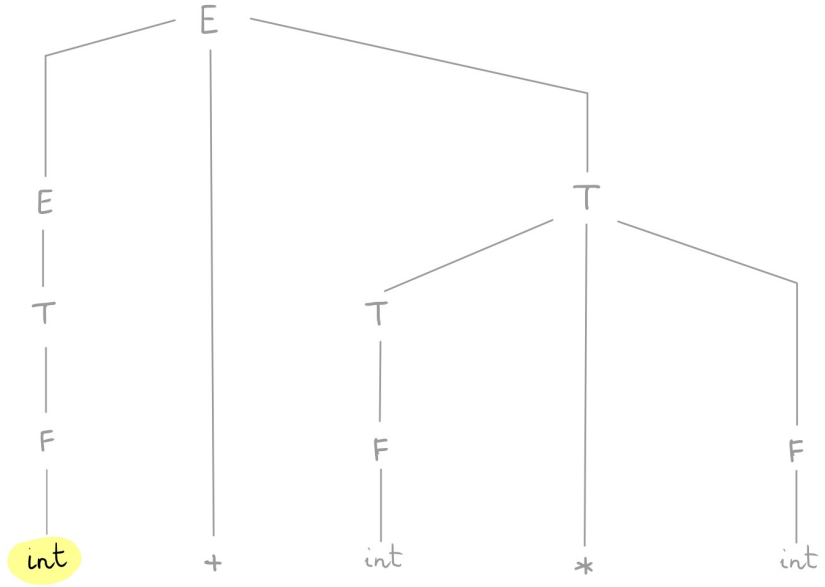
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

stack	input	action
ϵ	`int+int*int`	shift
`int`	`+int*int`	reduce $F \rightarrow$ `int`
F	`+int*int`	reduce $T \rightarrow F$
T	`+int*int`	reduce $E \rightarrow T$
E	`+int*int`	shift
E`+`	`int*int`	shift
E`+int`	`*int`	reduce $F \rightarrow$ `int`
$E+F$	`*int`	reduce $T \rightarrow F$
$E+T$	`*int`	shift
$E+T$`*`	`int`	shift
$E+T*$`int`		reduce $F \rightarrow$ `int`
$E+T*F$		reduce $T \rightarrow T*F$
$E+T$		reduce $E \rightarrow E+T$
E		success

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

Derivation tree



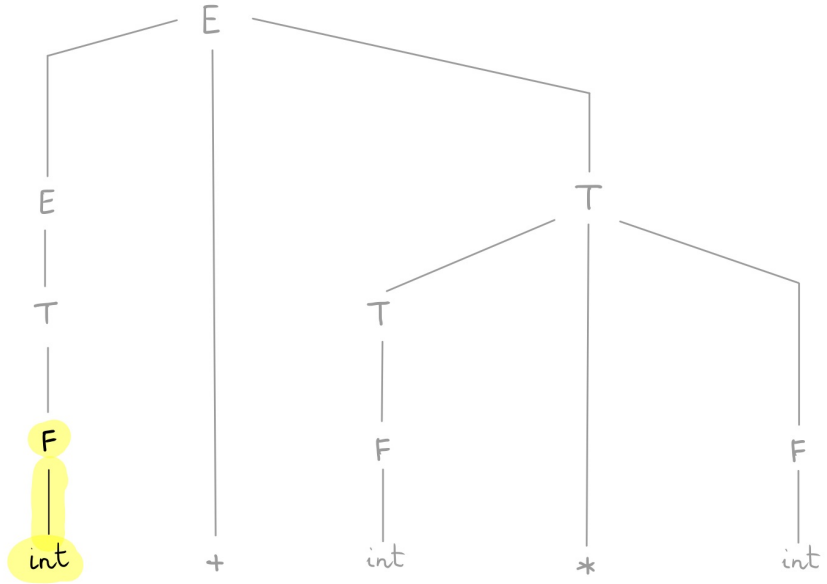
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T*F$
E+T		reduce $E \rightarrow E+T$
E		success

Derivation tree



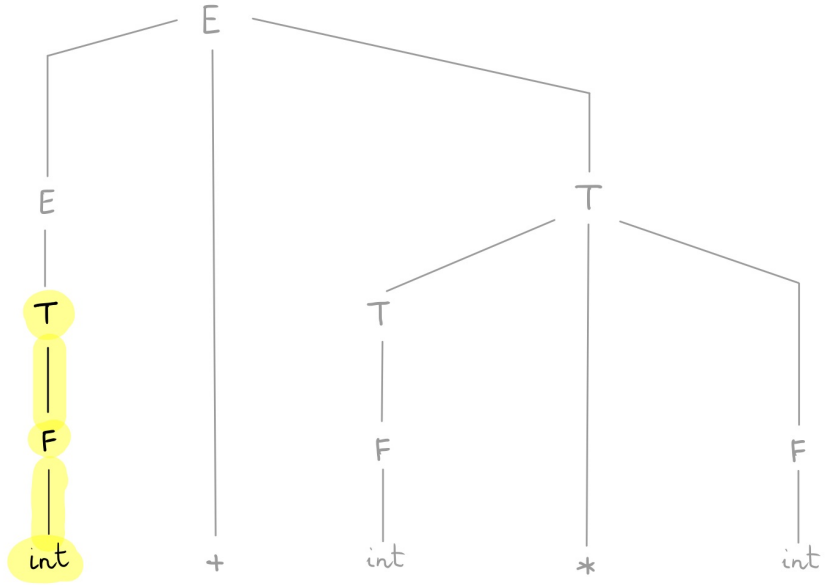
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

Derivation tree



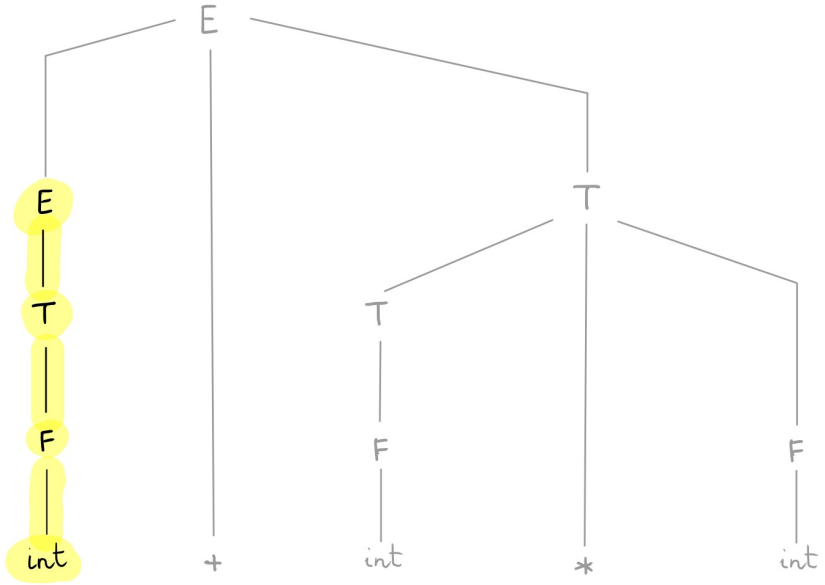
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

Derivation tree



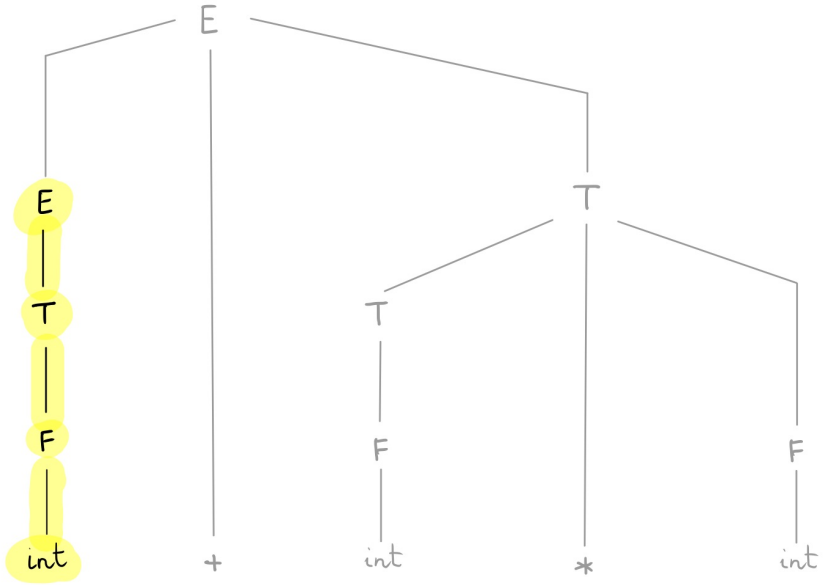
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

Derivation tree



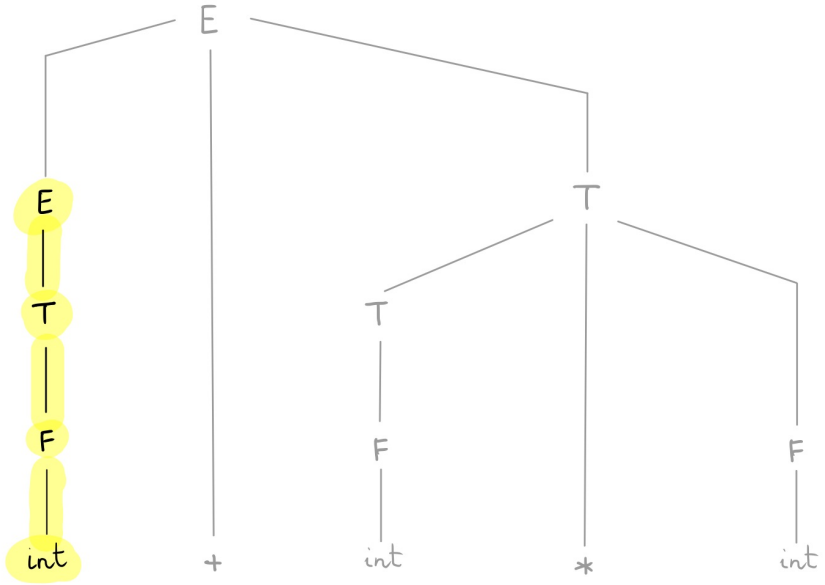
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

Derivation tree



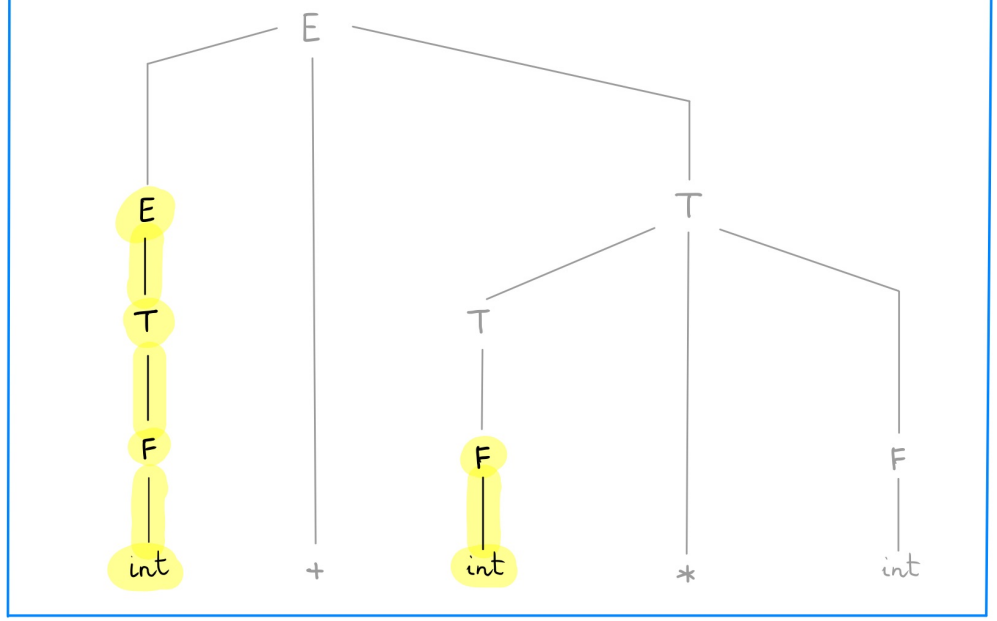
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

Derivation tree



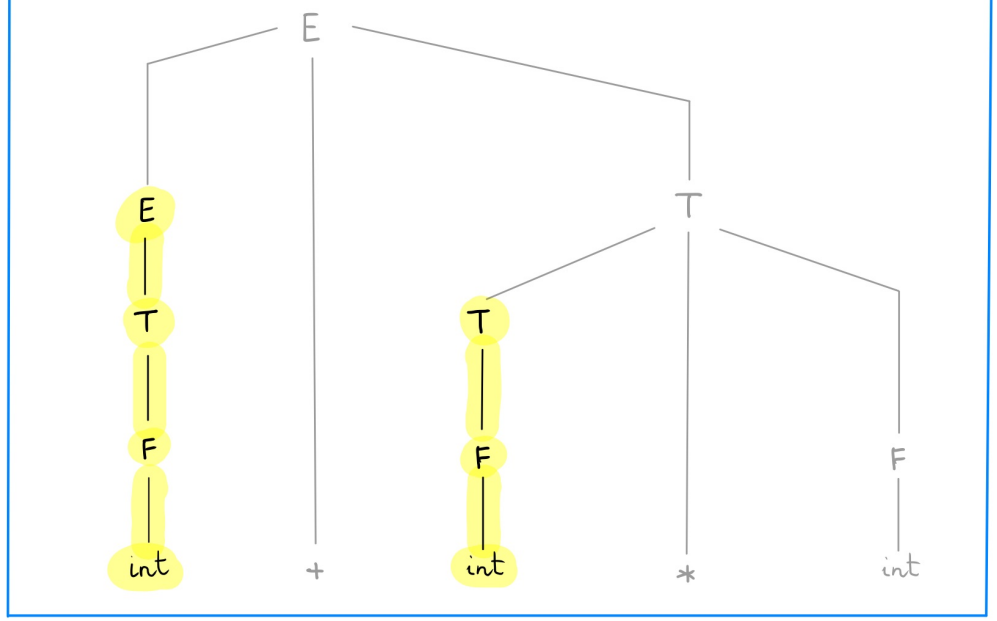
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

Derivation tree



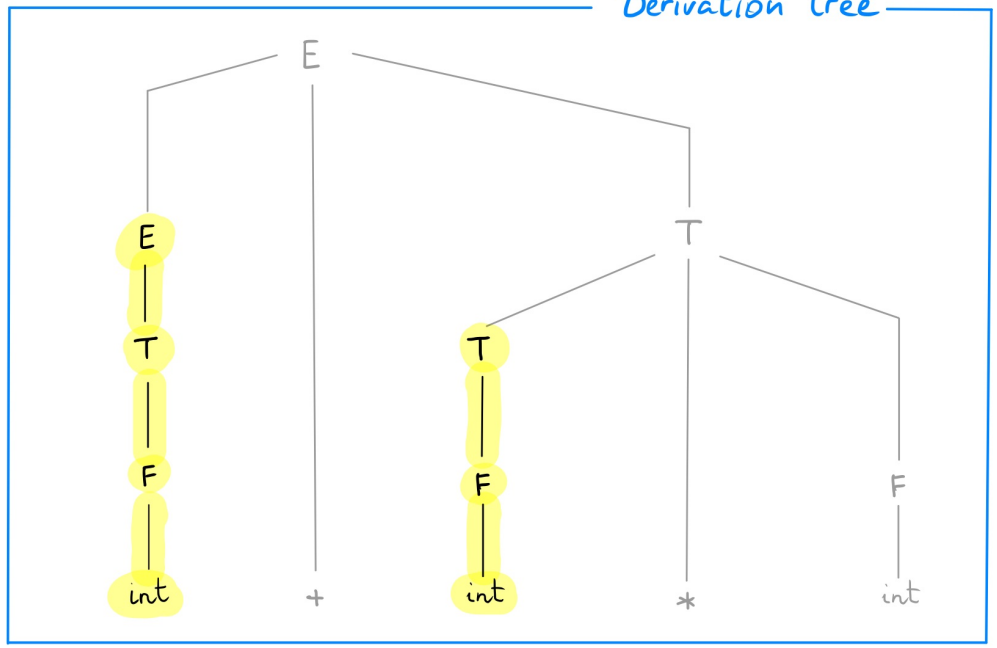
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

Derivation tree



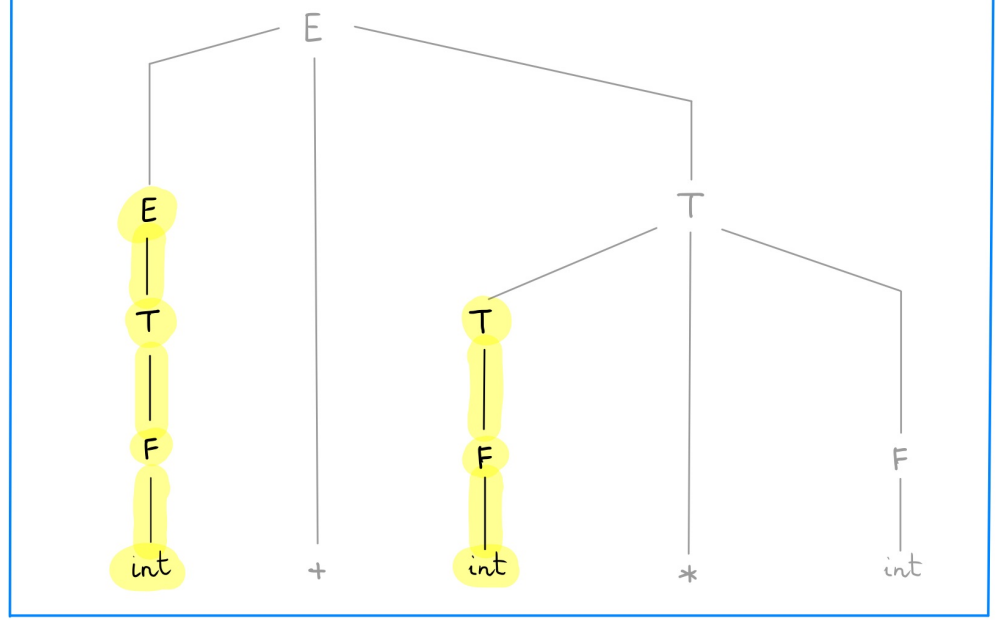
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

Derivation tree



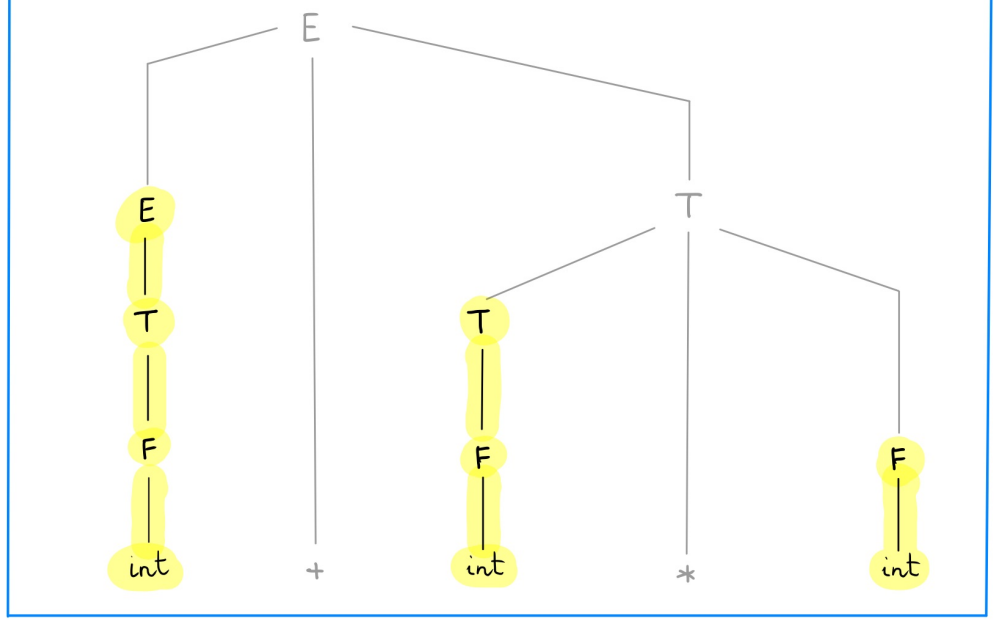
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T*F$
E+T		reduce $E \rightarrow E+T$
E		success

Derivation tree



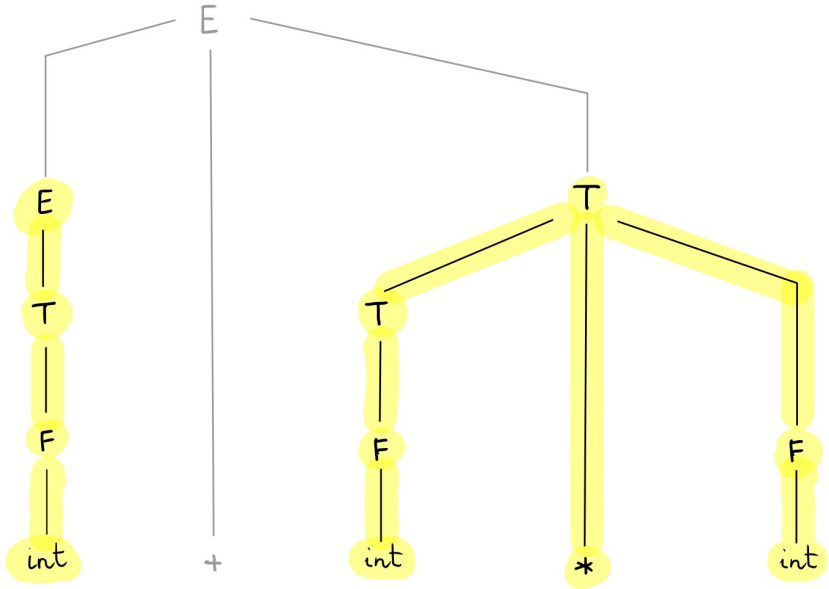
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T*F$
E+T		reduce $E \rightarrow E+T$
E		success

Derivation tree



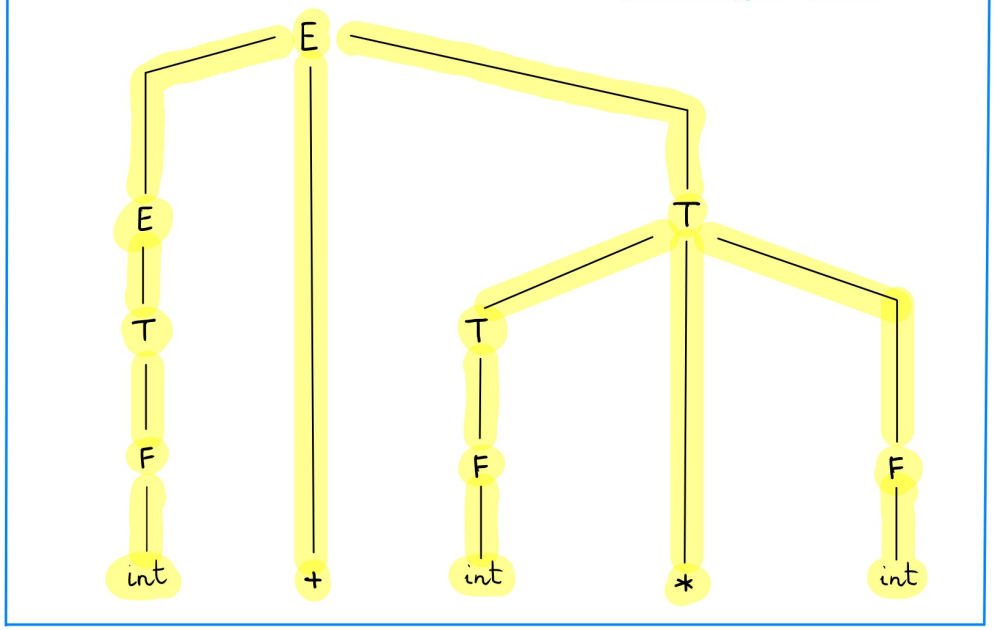
Grammar

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

Algorithm

stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

Derivation tree



Grammar

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

Example 2

$$
\begin{array}{rcl}
E & \rightarrow & E + E \\
 & | & (E) \\
 & | & \texttt{int}
\end{array}
$$

stack	input	action
ϵ	`int+int+int`	shift
`int`	`+int+int`	reduce $E \rightarrow$ `int`
E	`+int+int`	shift
E`+`	`int+int`	shift
E`+int`	`+int`	reduce $E \rightarrow$ `int`
E+E	`+int`	reduce $E \rightarrow E$+E
E	`+int`	shift
E`+`	`int`	shift
E`+int`		reduce $E \rightarrow$ `int`
E+E		reduce $E \rightarrow E$+E
E		success

Top-down and Bottom-up Parsing

In the top-down parsing, the crux was to be able to decide which action to perform on a stack reading the current token of the input (*this was last lecture*)

- that tool was the **expansion table** $T(X, a)$,
- the subtlety was to construct it using FIRST, FOLLOW & NULL sets.

In bottom-up parsing, the crux is to be able to decide how to choose between shift and reduce when reading a given token of the input

- that tool is a **deterministic finite automaton** whose transitions are labeled by $x_i \in T \cup N$ and states are stored on the stack alongside those symbols, *i.e.* the stack looks like $s_0\, x_1\, s_1\, x_2 \ldots x_n\, s_n$
- the subtlety is that the states are *items* of the form $[X \to \alpha \bullet \beta]$ where $X \to \alpha\beta$ is a grammar production; the intuition is "we want to recognize X, we've already seen α, but we still need to see β"

LR(k) Analysis

LR parser (Knuth, 1965)

How to choose between shift and reduce?

- using a finite deterministic automaton whose states are stored on the stack alongside grammar symbols $x \in T \cup N$
- and considering the first k tokens of the input.

This is called LR(k) analysis
(LR means "Left to right scanning, Rightmost derivation")

In practice $k = 1$
i.e., we only consider the first token to take the decision.

The stack looks like

$$s_0\, x_1\, s_1\, x_2 \ldots x_n\, s_n$$

where s_i is a state of the automaton and $x_i \in T \cup N$ as before.

Let a be the first token from the input; we look in the action table for state s_n and character a

- if **success** or **failure**, we stop
- if **shift**, we push on the stack a and the target state s of $s_n \xrightarrow{a} s$
- if **reduce** rule $X \to \alpha$, with α of length p, then we have α on top of the stack

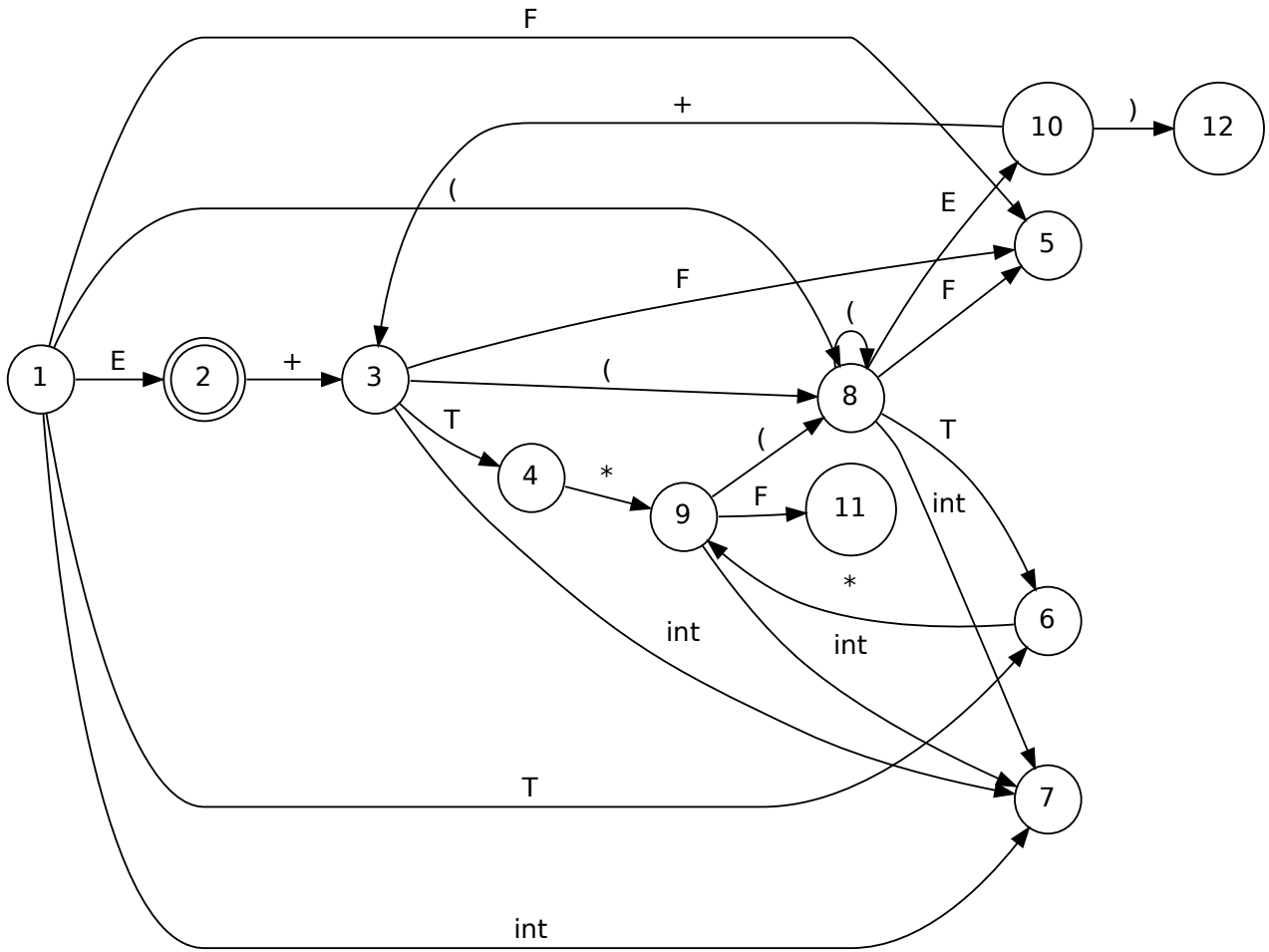
  $$s_0\, x_1\, s_1 \ldots x_{n-p}\, s_{n-p} \,|\, \alpha_1\, s_{n-p+1} \ldots \alpha_p\, s_n$$

  we pop it and we push $X\, s$, where s is the target state of the goto table for s_{n-p} and X, *i.e.* the stack becomes

  $$s_0\, x_1\, s_1 \ldots x_{n-p}\, s_{n-p}\, X\, s$$

In the example 1 before, we used the automaton

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



But in practice, we are working directly with the action and goto tables

- action table whose lines are indexed by the states and columns by terminals; the cell $\texttt{action}(s, a)$ indicates
  - $\texttt{shift}\ s'$ for reading a and pushing the new state s'
  - $\texttt{reduce}\ X \to \alpha$ for the corresponding rule reduction
  - success
  - failure

- goto table whose lines are indexed by the states and columns by the non-terminals; the cell $\texttt{goto}(s, X)$ indicates the state of the automaton after the reduction of X.

We also add the token # to indicate the end of the input.

We can do it as adding a new non terminal S (which is now the axiom of the grammar) and a new rule, e.g.

$$\begin{array}{lcl} S & \rightarrow & E\ \# \\ E & \rightarrow & \ldots \\ \vdots & & \end{array}$$

In our example 1, the tables are as follows:

1	E	$\rightarrow$	E + T	
2		$\mid$	T	
3	T	$\rightarrow$	T * F	
4		$\mid$	F	
5	F	$\rightarrow$	(E)	
6		$\mid$	`int`	

si = `shift` i

rj = `reduce` j

	action						*goto*		
state	(	)	+	*	`int`	#	E	T	F
1	s8				s7		2	6	5
2			s3			success			
3	s8				s7			4	5
4		r1	r1	s9		r1			
5		r4	r4	r4		r4			
6		r2	r2	s9		r2			
7		r6	r6	r6		r6			
8	s8				s7		10	6	5
9	s8				s7				11
10		s12	s3						
11		r3	r3	r3		r3			
12		r5	r5	r5		r5			

Example 1 Execution

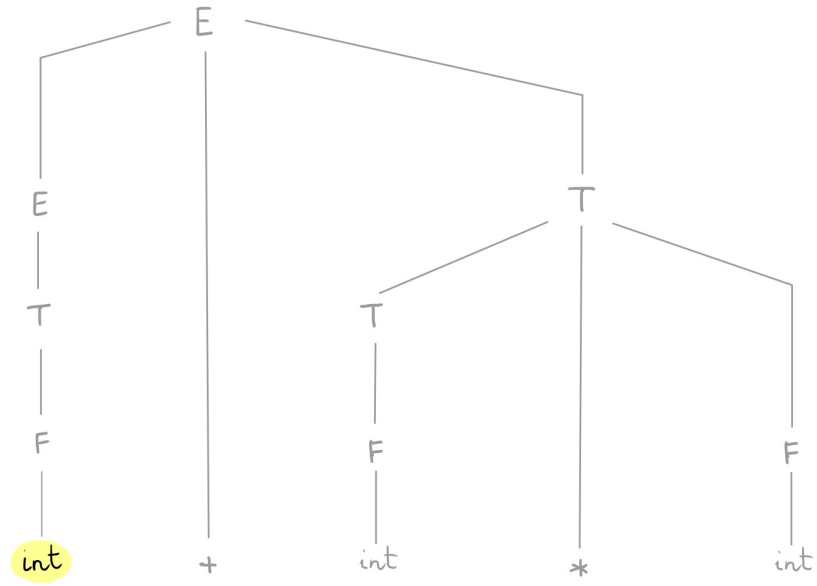
stack	input	action
1	`int+int*int#`	`s7`
1 `int` 7	`+int*int#`	$F \rightarrow$ `int`, g5
1 F 5	`+int*int#`	$T \rightarrow F$, g6
1 T 6	`+int*int#`	$E \rightarrow T$, g2
1 E 2	`+int*int#`	`s3`
1 E 2 + 3	`int*int#`	`s7`
1 E 2 + 3 `int` 7	`*int#`	$F \rightarrow$ `int`, g5
1 E 2 + 3 F 5	`*int#`	$T \rightarrow F$, g4
1 E 2 + 3 T 4	`*int#`	`s9`
1 E 2 + 3 T 4 * 9	`int#`	`s7`
1 E 2 + 3 T 4 * 9 `int` 7	#	$F \rightarrow$ `int`, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T*F$, g4
1 E 2 + 3 T 4	#	$E \rightarrow E+T$, g2
1 E 2	#	success

Algorithm

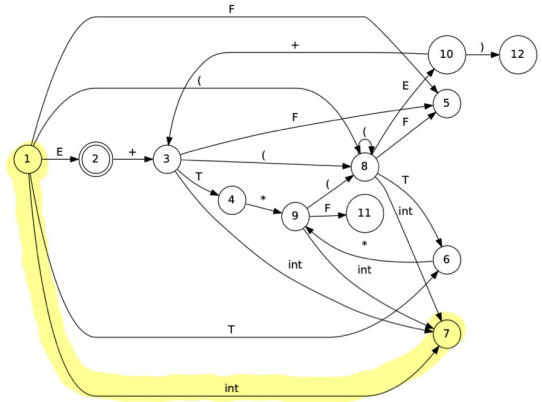
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

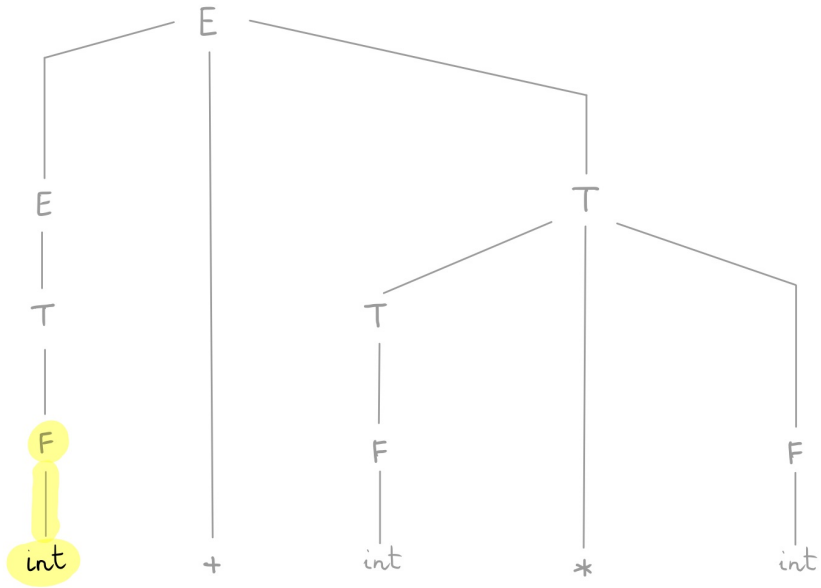
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

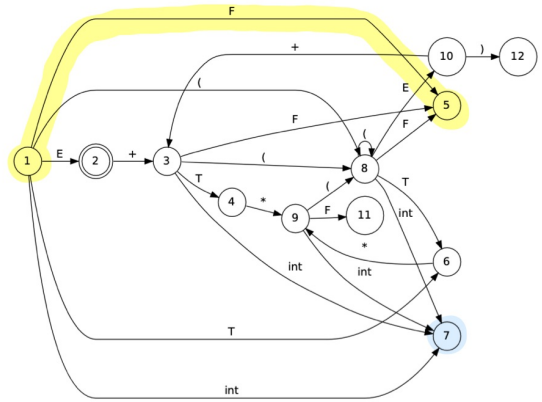
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

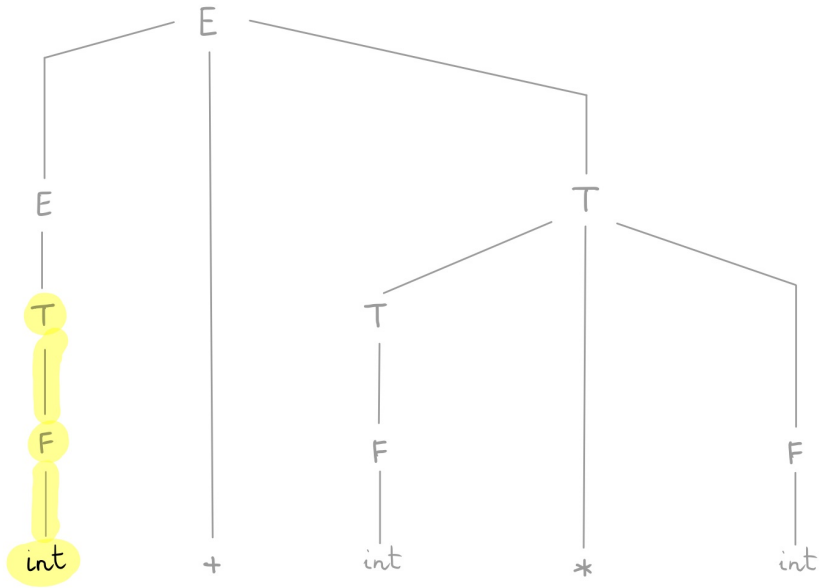
$$
\begin{aligned}
E &\rightarrow E + T \\
&\mid T \\
T &\rightarrow T * F \\
&\mid F \\
F &\rightarrow (E) \\
&\mid \texttt{int}
\end{aligned}
$$

Algorithm

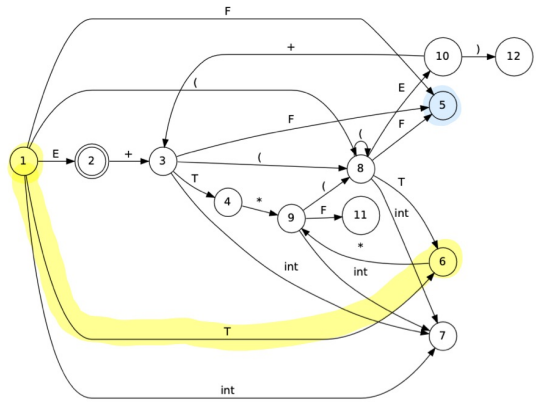
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

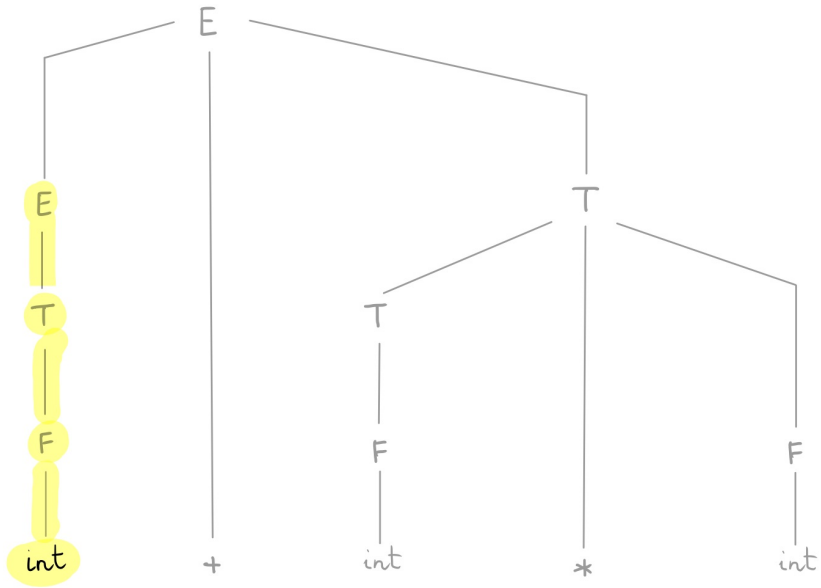
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

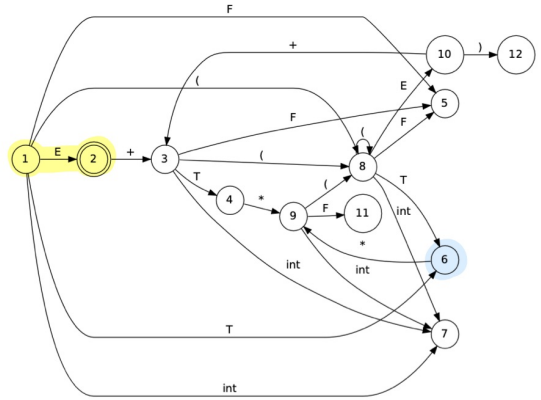
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

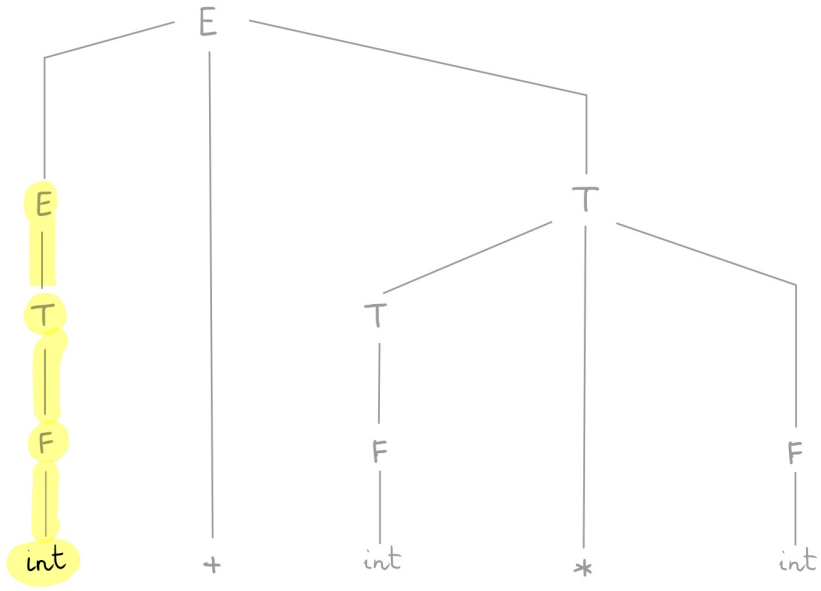
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

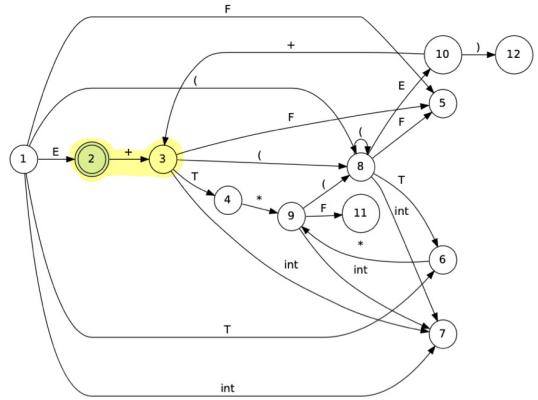
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

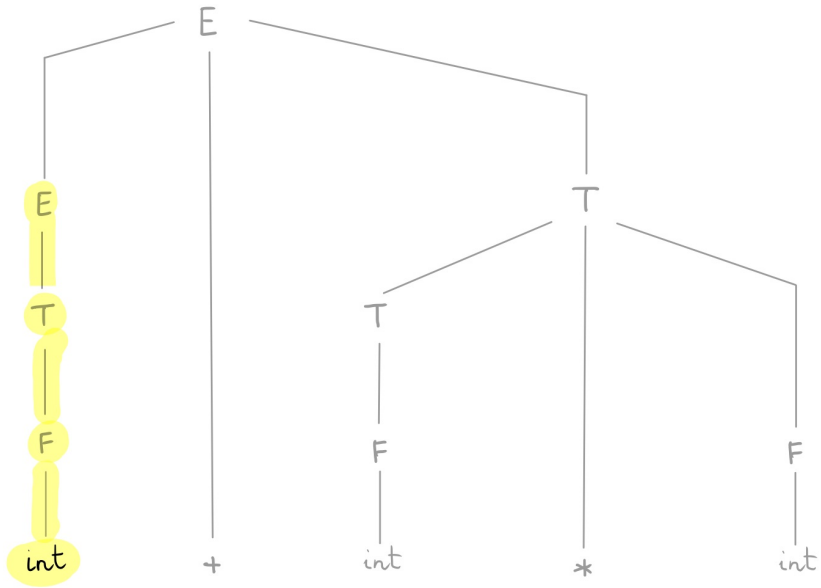
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

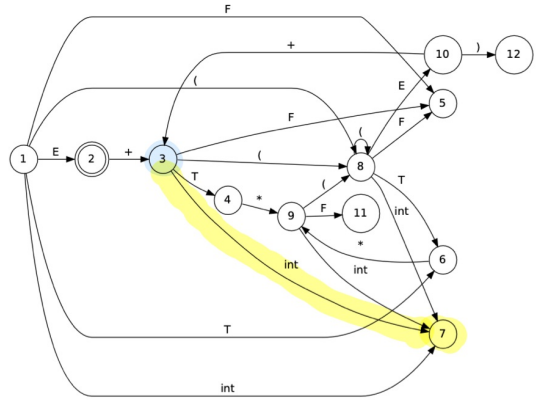
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

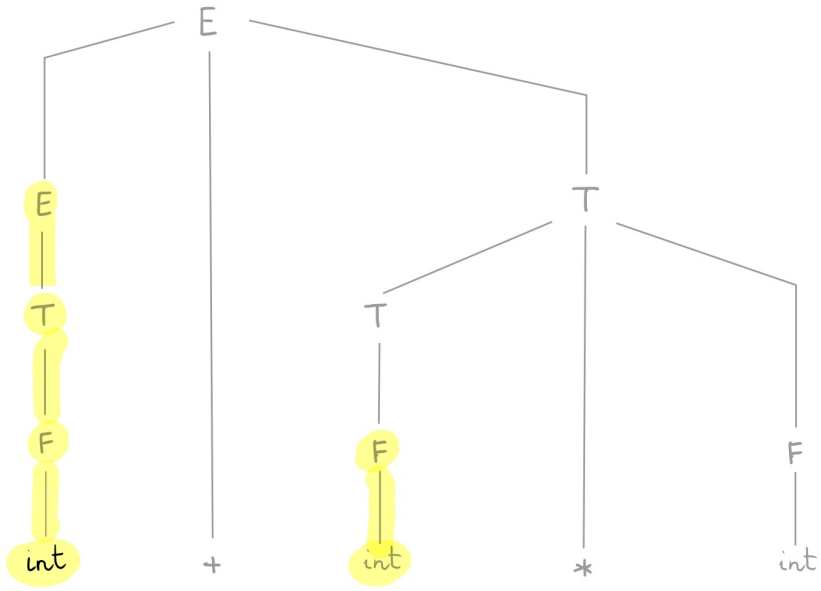
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

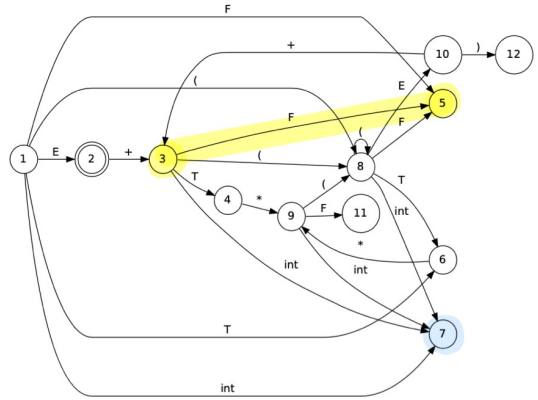
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

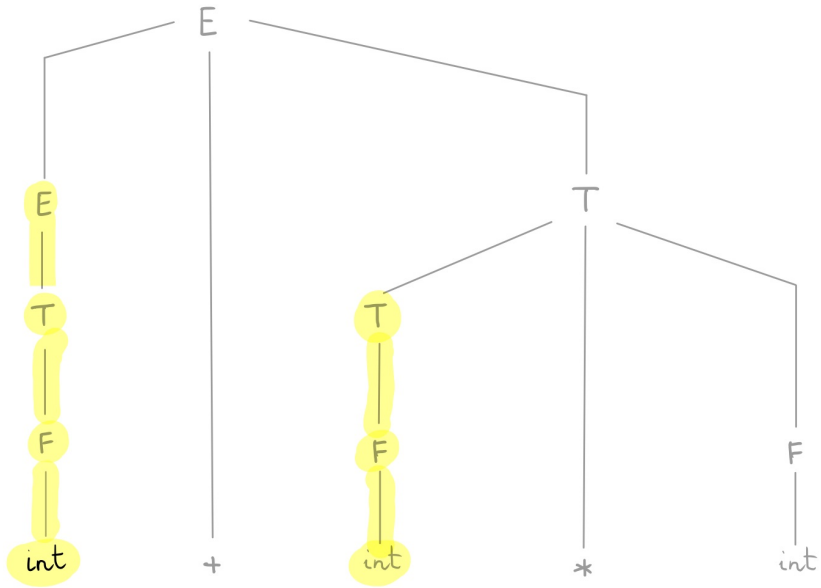
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

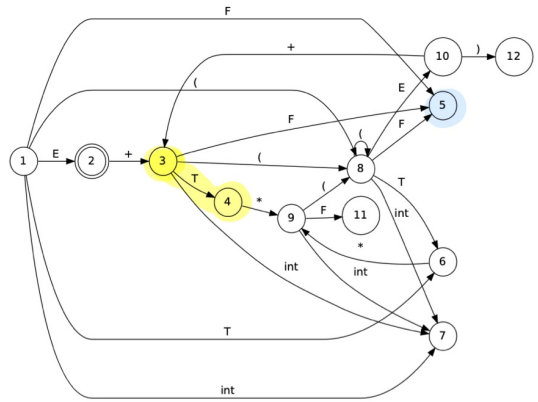
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

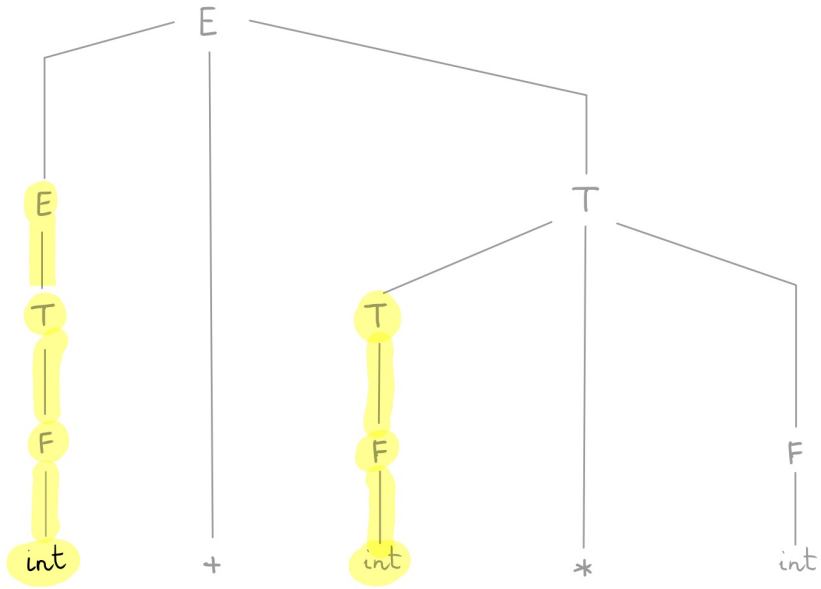
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

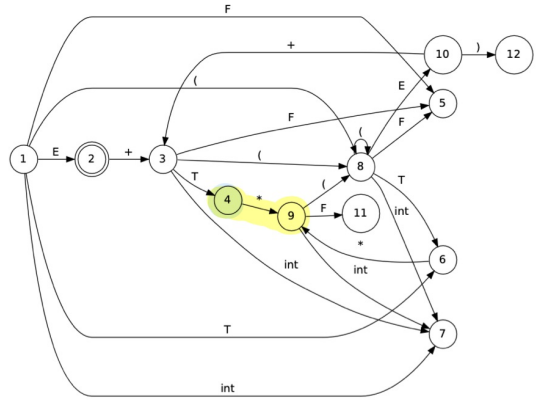
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

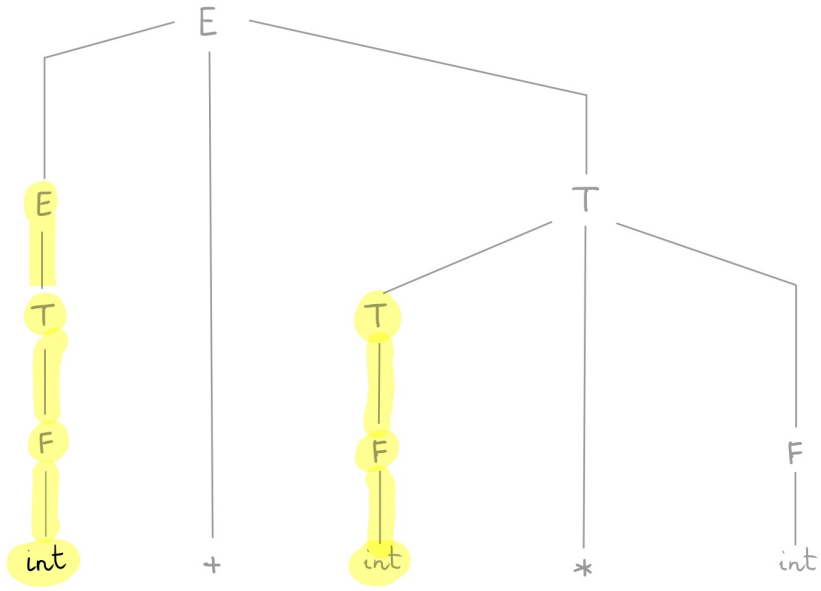
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

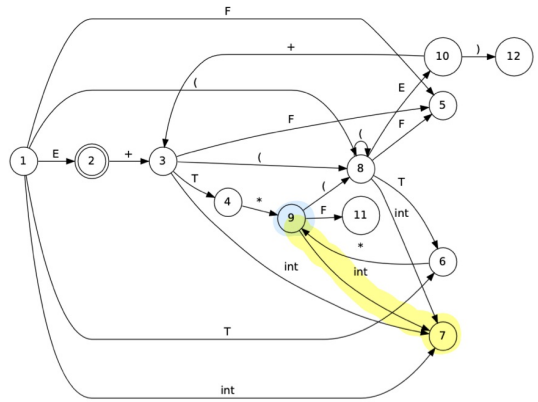
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

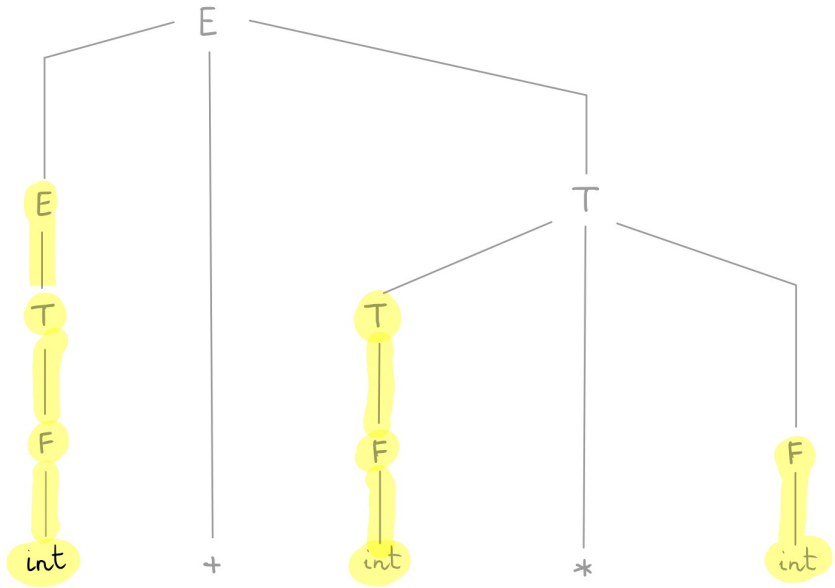
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

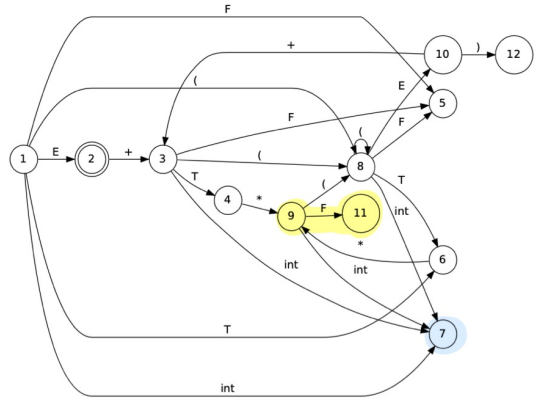
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

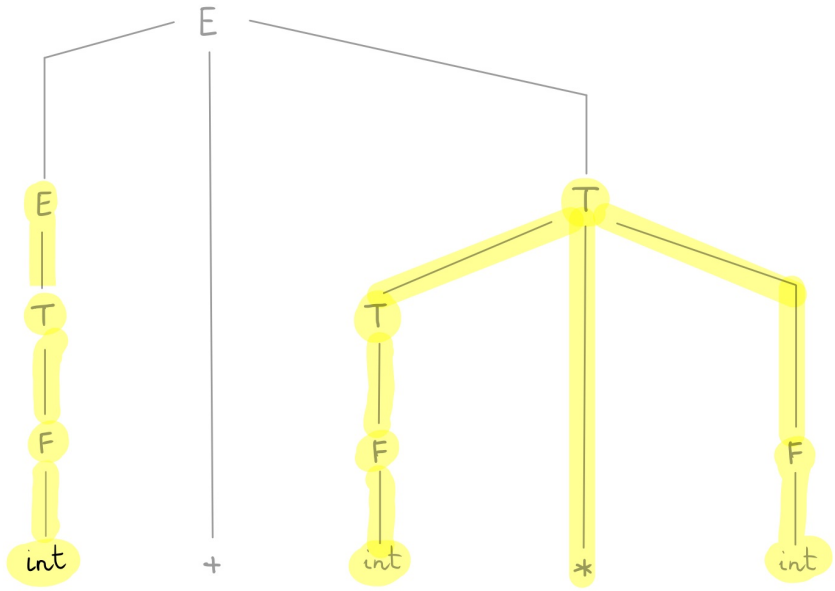
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

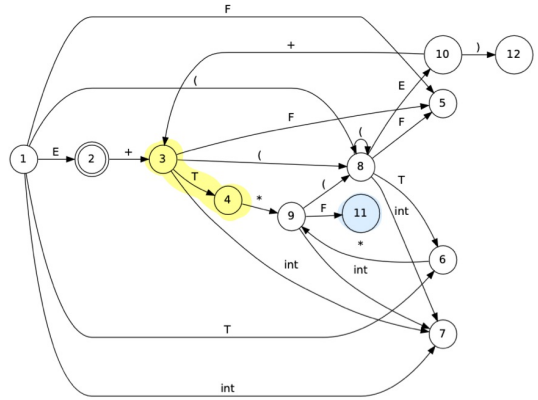
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T*F$
E+T		reduce $E \rightarrow E+T$
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T*F$, g4
1 E 2 + 3 T 4	#	$E \rightarrow E+T$, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

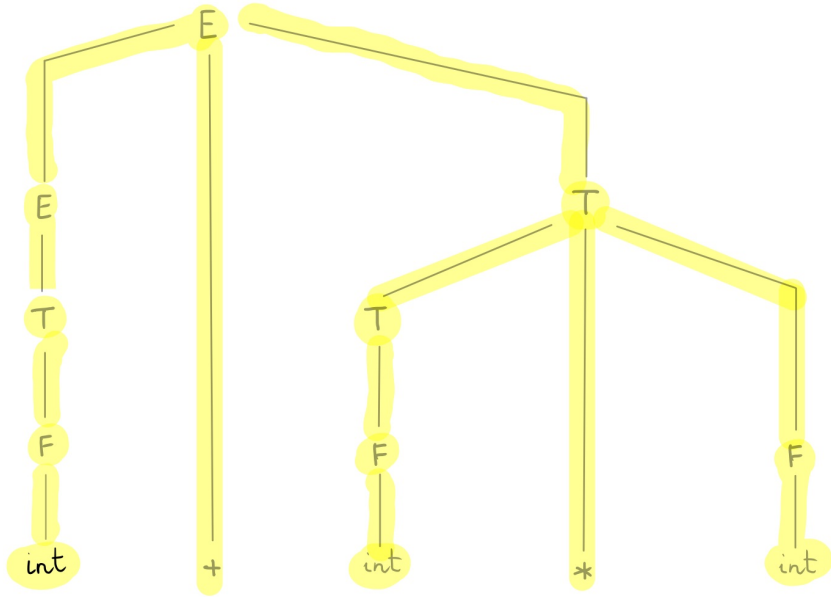
$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

Algorithm

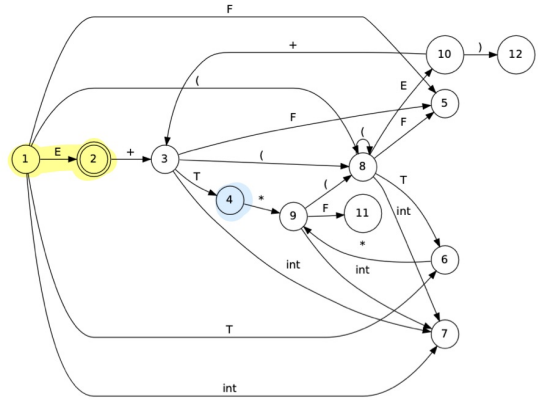
stack	input	action
ϵ	int+int*int	shift
int	+int*int	reduce $F \rightarrow$ int
F	+int*int	reduce $T \rightarrow F$
T	+int*int	reduce $E \rightarrow T$
E	+int*int	shift
E+	int*int	shift
E+int	*int	reduce $F \rightarrow$ int
E+F	*int	reduce $T \rightarrow F$
E+T	*int	shift
E+T*	int	shift
E+T*int		reduce $F \rightarrow$ int
E+T*F		reduce $T \rightarrow T$*F
E+T		reduce $E \rightarrow E$+T
E		success

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T$*F, g4
1 E 2 + 3 T 4	#	$E \rightarrow E$+T, g2
1 E 2	#	success

Derivation tree



Automaton



Grammar

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

The automaton is implemented as follows:

	action					*goto*
state	(	)	+	int	#	E
1	shift 4			shift 2		3
2	reduce $E \rightarrow$ int					
3			shift 6		success	
4	shift 4			shift 2		5
5		shift 7	shift 6			
6	shift 4			shift 2		8
7	reduce $E \rightarrow (E)$					
8	reduce $E \rightarrow E$+E					

(we show later how to built it)

Execution Example 2

	(	)	+	int	#	E
1	s4			s2		3
2	reduce $E \rightarrow$ int					
3			s6		ok	
4	s4			s2		5
5		s7	s6			
6	s4			s2		8
7	reduce $E \rightarrow (E)$					
8	reduce $E \rightarrow E$+E					

stack	input	action
1	int+int+int	s2
1 int 2	+int+int	$E \rightarrow$ int, g3
1 E 3	+int+int	s6
1 E 3 + 6	int+int	s2
1 E 3 + 6 int 2	+int	$E \rightarrow$ int, g8
1 E 3 + 6 E 8	+int	$E \rightarrow E$+E, g3
1 E 3	+int	s6
1 E 3 + 6	int	s2
1 E 3 + 6 int 2	#	$E \rightarrow$ int, g8
1 E 3 + 6 E 8	#	$E \rightarrow E$+E, g3
1 E 3	#	success

Building the Automaton

Let us use $k = 0$ for the moment.

We consider the following grammar:

$$
\begin{array}{lcl}
S & \rightarrow & E \\
 & & \\
E & \rightarrow & E\texttt{+}E \\
 & | & (E) \\
 & | & \texttt{int}
\end{array}
$$

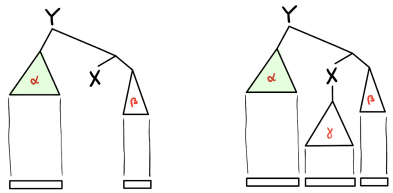
We start by constructing an automaton with ϵ-transitions denoted by $s_1 \xrightarrow{\epsilon} s_2$.

The states are *items* of the form

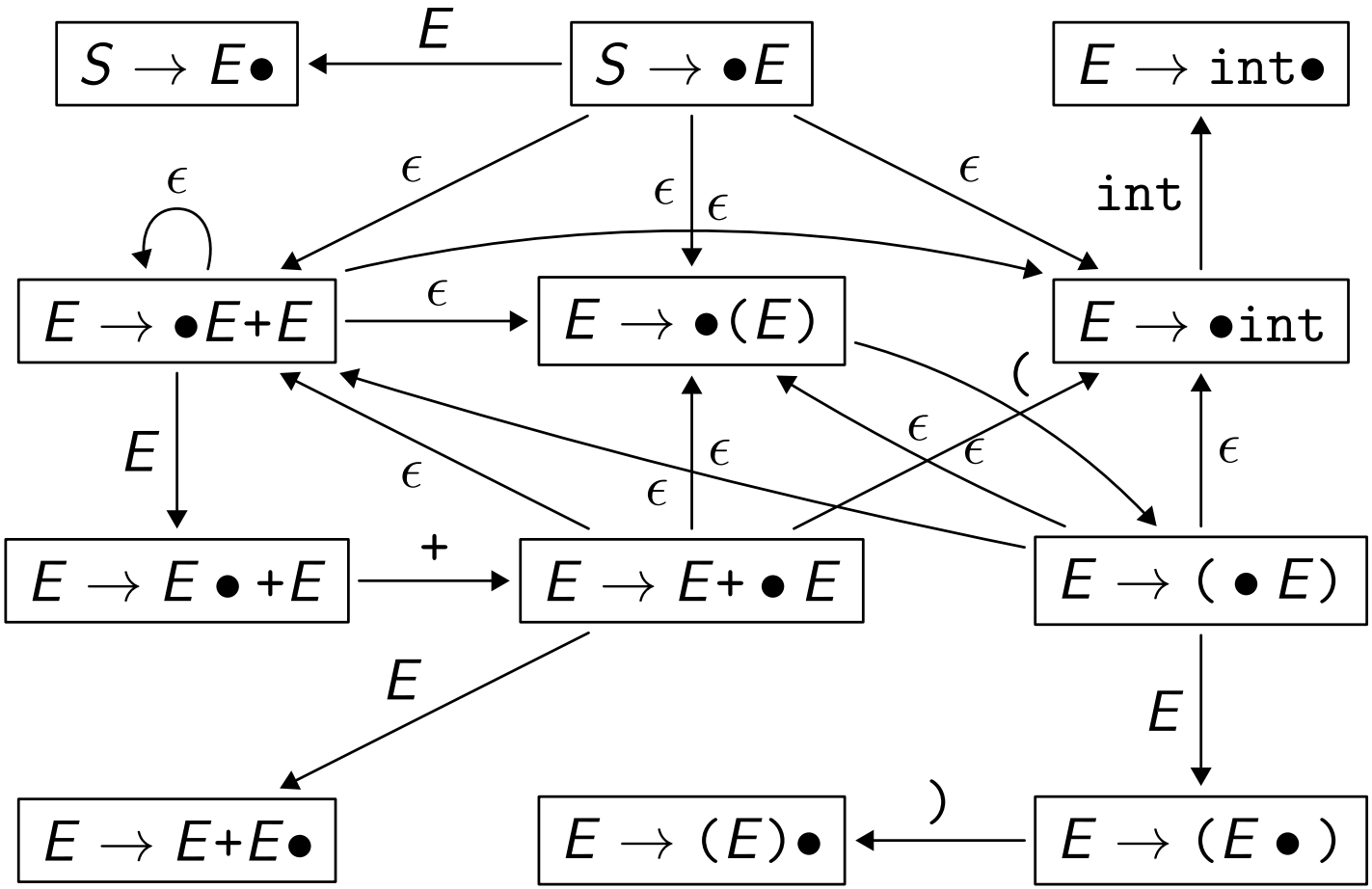
$$[X \to \alpha \bullet \beta]$$

where $X \to \alpha\beta$ is a grammar production; the intuition is
"we want to recognize X, we've already seen α and we still need to see β"
the transitions are labeled by $T \cup N$ are as follows:

$$\begin{array}{rcl} [Y \to \alpha \bullet a\beta] & \xrightarrow{a} & [Y \to \alpha a \bullet \beta] \\ [Y \to \alpha \bullet X\beta] & \xrightarrow{X} & [Y \to \alpha X \bullet \beta] \\ [Y \to \alpha \bullet X\beta] & \xrightarrow{\epsilon} & [X \to \bullet\gamma] \\ & & \text{for each production } X \to \gamma \end{array}$$



Example



$$
\begin{array}{lcl}
S & \to & E \\
E & \to & E{+}E \\
 & | & (E) \\
 & | & \texttt{int}
\end{array}
$$

Let's make LR automaton deterministic by regrouping the states connected by ϵ-transitions.

The states of the resulting automaton are thus the *sets of items*

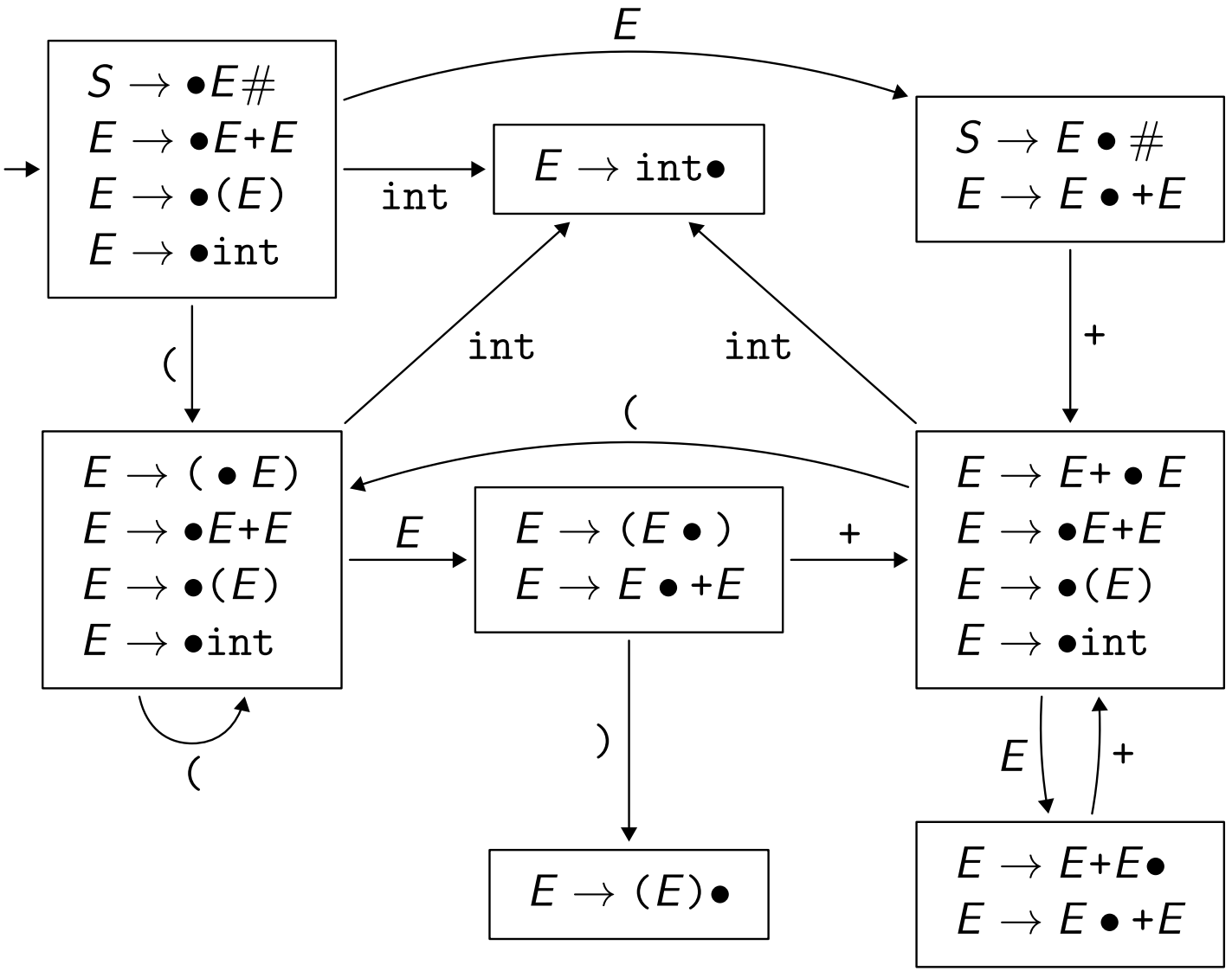
$$
\begin{array}{l}
E \rightarrow E \texttt{+} \bullet E \\
E \rightarrow \bullet E \texttt{+} E \\
E \rightarrow \bullet \texttt{(} E \texttt{)} \\
E \rightarrow \bullet \texttt{int}
\end{array}
$$

By construction, each state s is saturated by the following property

$$\begin{array}{rl} \text{if} & Y \to \alpha \bullet X \beta \in s \\ \text{and if} & X \to \gamma \text{ is a production} \\ \text{then} & X \to \bullet \gamma \in s \end{array}$$

The initial state is the one containing the item $S \to \bullet E\ \#$.

Example



$$
\begin{array}{lcl}
S & \to & E\ \# \\
E & \to & E{+}E \\
 & \mid & (E) \\
 & \mid & \texttt{int}
\end{array}
$$

The `action` table is built as follows:

- $\texttt{action}(s, \#) =$ success if $[S \to E \bullet \#] \in s$
- $\texttt{action}(s, a) = \texttt{shift}\ s'$ if we have $s \xrightarrow{a} s'$
- $\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$ if $[X \to \beta\bullet] \in s$, for all a
- failure otherwise

The `goto` table is built as follows:

- $\texttt{goto}(s, X) = s'$ if and only if we have $s \xrightarrow{X} s'$

On our example, we get

	action					*goto*
state	(	)	+	int	#	E
1	shift 4			shift 2		3
2	reduce $E \rightarrow$ int					
3			shift 6		success	
4	shift 4			shift 2		5
5		shift 7	shift 6			
6	shift 4			shift 2		8
7	reduce $E \rightarrow (E)$					
8			shift 6			
	reduce $E \rightarrow E$+E					

The LR(0) table may contain two kinds of conflicts

- a shift/reduce conflict, if we can do both a shift and a reduce action
- a reduce/reduce conflict, if we can do two different reduce actions

Definition (LR(0) grammar)

A grammar is said to be LR(0) if the table contains no conflict.

We have a shift/reduce conflict in state 8

$$\boxed{\begin{array}{l} E \to E{+}E\bullet \\ E \to E\bullet{+}E \end{array}}$$

It illustrates the ambiguity of the grammar on input `int+int+int`.

We can remove the conflict in two different ways:

- if we favor shift, we make + right associative
- if we favor reduce, we make + left associative
  (and we get the table we used earlier)

LR(0) tables quickly contain conflicts,
so let us try to remove some reduce actions.

A simple idea is to set $\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$ if and only if

$$[X \to \beta\bullet] \in s \quad \text{and} \quad a \in \textsc{follow}(X)$$

Definition (SLR(1) grammar)

A grammar is said to be SLR(1) if the resulting table contains no conflict.

(SLR means *Simple LR*)

The grammar

$$
\begin{array}{lcl}
S & \rightarrow & E\# \\
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

is SLR(1)

Exercise: check it (the automaton has 12 states)

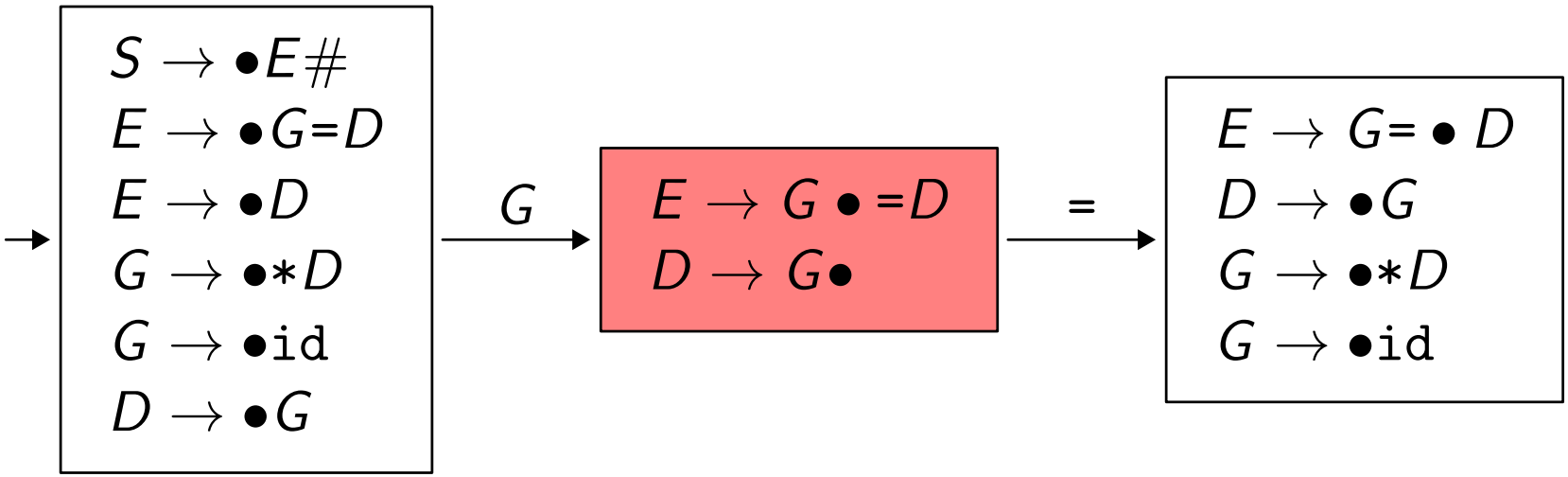
Limits of SLR(1) Analysis

In practice, SLR(1) grammars are not powerful enough.

Example:

$$
\begin{array}{lcl}
S & \rightarrow & E\# \\
E & \rightarrow & G = D \\
 & | & D \\
G & \rightarrow & *D \\
 & | & \texttt{id} \\
D & \rightarrow & G
\end{array}
$$

	=	
1	...	...
2	shift 3 reduce $D \rightarrow G$	...
3	⋮	⋱



We introduce a larger class of grammars, LR(1), with larger tables

Items now look like

$$[X \to \alpha \bullet \beta, a]$$

and the meaning is "we want to recognize X, we have already seen α, we still need to see β and then to check that the next token is a".

The LR(1) automaton has transitions

$$[Y \to \alpha \bullet a\beta, b] \xrightarrow{a} [Y \to \alpha a \bullet \beta, b]$$
$$[Y \to \alpha \bullet X\beta, b] \xrightarrow{X} [Y \to \alpha X \bullet \beta, b]$$

and in a state containing $[Y \to \alpha \bullet X\beta, b]$ we only include

$$[X \to \bullet\gamma, c] \quad \text{for all } c \in \textsc{first}(\beta b)$$

The initial state is that containing $[S \to \bullet\alpha, \#]$.

There is a reduce action for (s, a) only when s contains an item $[X \to \alpha\bullet, a]$.

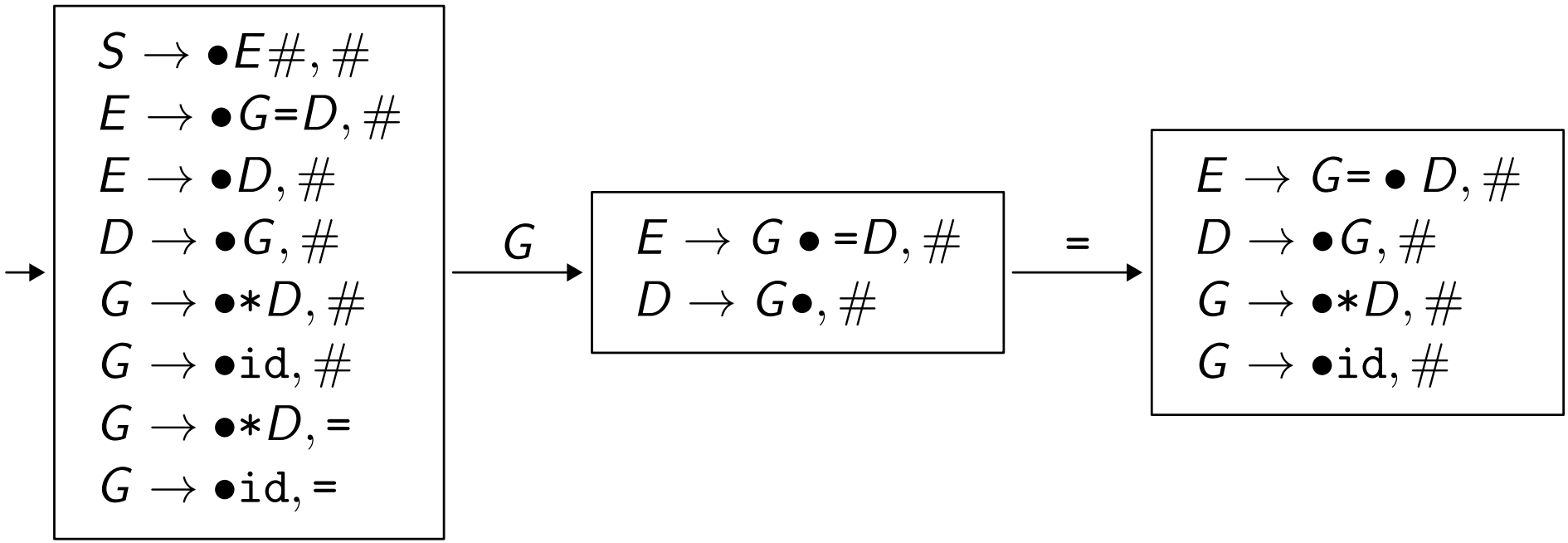
Definition (LR(1) grammar)

A grammar is said to be LR(1) if the resulting table contains no conflict.

Example

$$
\begin{array}{lcl}
S & \rightarrow & E\# \\
E & \rightarrow & G = D \\
 & | & D \\
G & \rightarrow & * D \\
 & | & \texttt{id} \\
D & \rightarrow & G
\end{array}
$$

	#	=	
1	...	...	...
2	reduce $D \rightarrow G$	shift 3	...
3	⋮	⋮	⋱



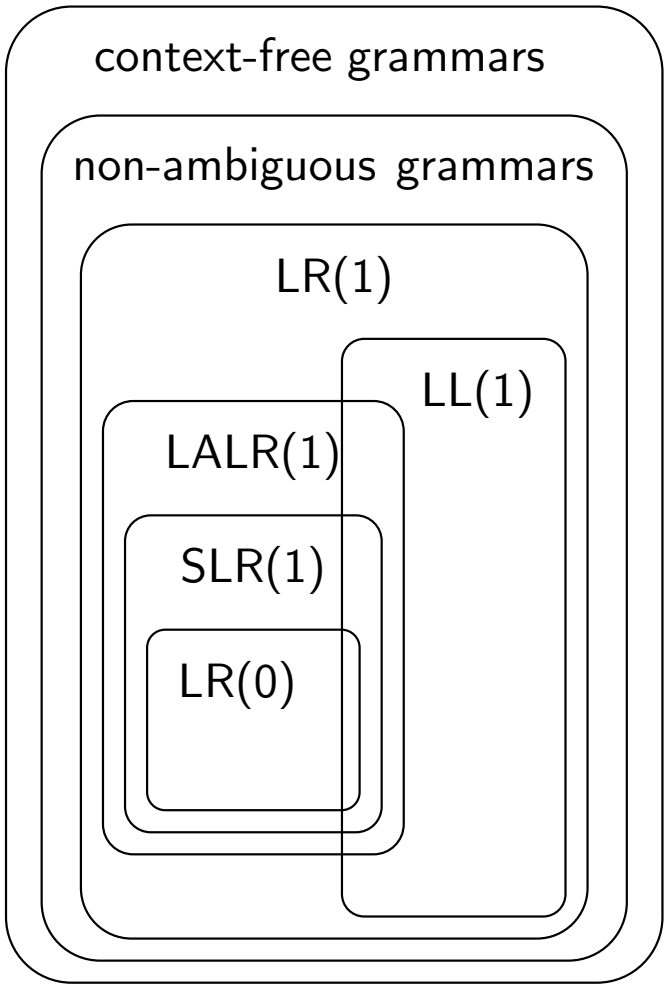
LALR(1)

The LR(1) tables can be large, so we introduced approximations.

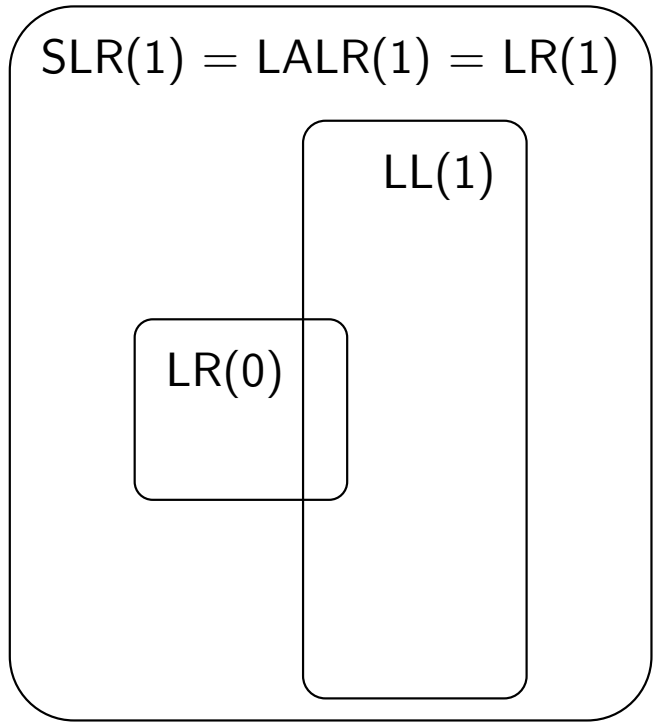
The class LALR(1) (*lookahead LR*) is such an approximation, used in tools of the `yacc` family.

Grammar Hierarchies

Grammars



Languages



Bottom-up parsing is powerful but computing the tables is complex.

We have tools to automate the process.

This is the big family of `yacc`, `bison`, `ocamlyacc`, `cup`, `menhir`, ...
(YACC means *Yet Another Compiler Compiler*).

Here we illustrate `cup` (for Java) and `menhir` (for OCaml).

We keep the example of the language of arithmetic expressions (with integer literals, parentheses, and addition) and manipulating the grammar

$$\begin{array}{rcl} E & \rightarrow & E + E \\ & | & (E) \\ & | & \texttt{int} \end{array}$$

We assume that abstract syntax and lexical analysis are already implemented.

CUP (Java)

In a file `Parser.cup`, we start with a prelude where we declare terminals and nonterminals

```
terminal Integer INT;
terminal LPAR, RPAR, PLUS;

non terminal Expr file;
non terminal Expr expr;
```

...

Then we declare the production rules and the corresponding actions

```
start with file;

file ::=
  expr:e
    {: RESULT = e; :}
;

expr ::=
  INT:n
    {: RESULT = new Ecst(n); :}
| expr:e1 PLUS expr:e2
    {: RESULT = new Esub(e1, e2); :}
| LPAR expr:e RPAR
    {: RESULT = e; :}
;
```

We compile file `Parser.cup` with

```
java -jar java-cup-11a.jar -parser Parser Parser.cup
```

Which signals an error:

```
Warning : *** Shift/Reduce conflict found in state #6
  between expr ::= expr PLUS expr (*)
  and     expr ::= expr (*) PLUS expr
  under symbol PLUS
  Resolved in favor of shifting.

Error : *** More conflicts encountered than expected
         - parser generation aborted
```

We can declare `MINUS` to be left associative.

```
precedence left MINUS;
```

(which favors reduction)

If there are more operators, we list them in increasing priorities

```
precedence left PLUS, MINUS;
precedence left TIMES, DIV, MOD;
```

Now CUP successfully terminates and produces two Java files:

- `sym.java` contains the declarations of tokens (`INT`, `LPAR`, `RPAR`, etc.)
- `Parser.java` contains the syntax analyzer, with a constructor

```
Parser(Scanner scanner)
```

and a method

```
Symbol parse()
```

Connecting jflex and CUP

We combine the code generated by jflex and CUP as follows:

```
Reader reader = new FileReader(file);
Lexer lexer = new Lexer(reader);
Parser parser = new Parser(lexer);
Expr e = (Expr)parser.parse().value;
try {
  System.out.println(e.eval());
} catch (Error err) {
  System.out.println("error: " + err.toString());
  System.exit(1);
}
```

The program must include the library java-cup-11a-runtime.jar.

Menhir (OCaml)

In a file `parser.mly`, we first declare terminals and nonterminals

```
%{
  ... arbitrary OCaml code ...
%}

%token PLUS LPAR RPAR EOF
%token <int> INT

%start <expr> file
```

...

Note: contrary to `CUP`, one has to declare `EOF`.

then we give the grammar production rules and the corresponding actions

```
%%

file:
  e = expr; EOF  { e }
;

expr:
| i = INT                        { Cte i }
| e1 = expr; PLUS; e2 = expr { Add (e1, e2) }
| LPAR; e = expr; RPAR           { e }
;

%%
```

Note: contrary to CUP, one has to add EOF

We compile file `arith.mly` as follows:

```
menhir -v arith.mly
```

which outputs two OCaml files `arith.ml(i)` that contain

- a data type `token`

```
type token = RPAR | PLUS | LPAR | INT of int | EOF
```

- a function

```
val file: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> expr
```

Connecting ocamllex and menhir

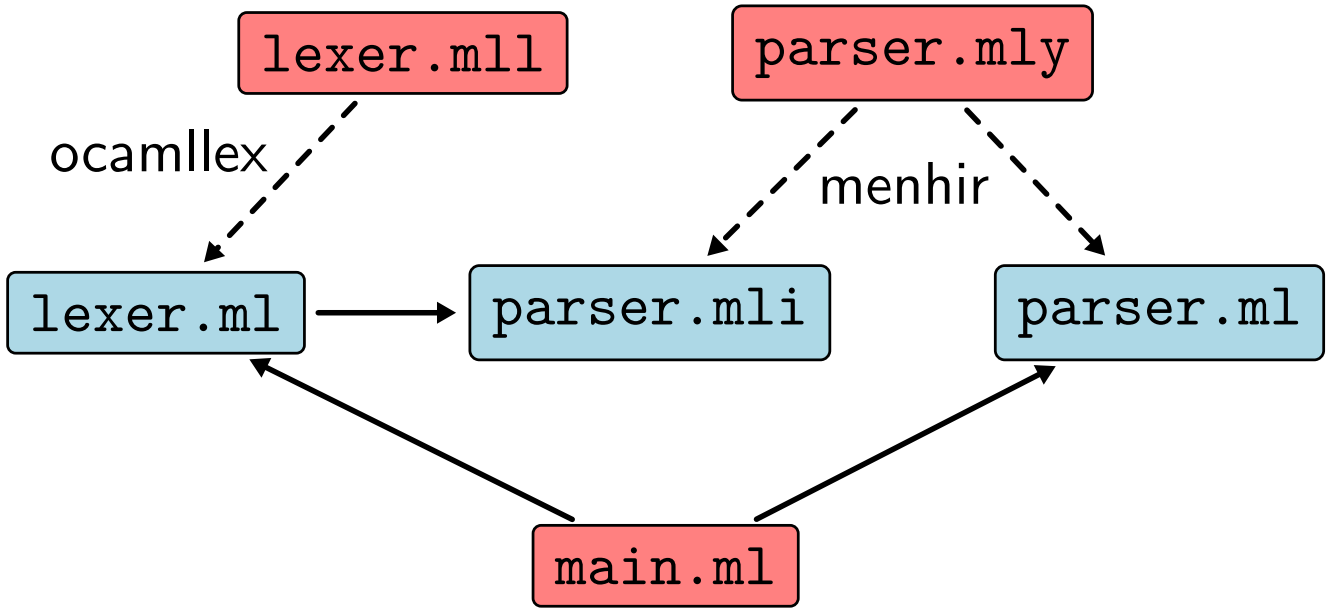
We combine ocamllex and menhir as follows:

- lexer.mll can mention tokens defined in the parser.mly

```
{
  open Parser
}
...
```

- lexer and parser are then combined as follows:

```
let c  = open_in file in
let lb = Lexing.from_file c in
let e  = Parser.file Lexer.next_token lb in
...
```



= implemented by the user

= constructed automatically

= dependencies

Conflicts

When the grammar is not LR(1), Menhir shows the conflicts to the user

- the file `.automaton` contains the LR(1) automaton (more later), with conflicts listed
- the file `.conflicts` contains an explanation for each conflict, as a sequence of tokens leading to two distinct derivation trees.

In our example, `menhir` signals a conflict:

```
% menhir -v arith.mly
Warning: one state has shift/reduce conflicts.
Warning: one shift/reduce conflict was arbitrarily resolved.
```

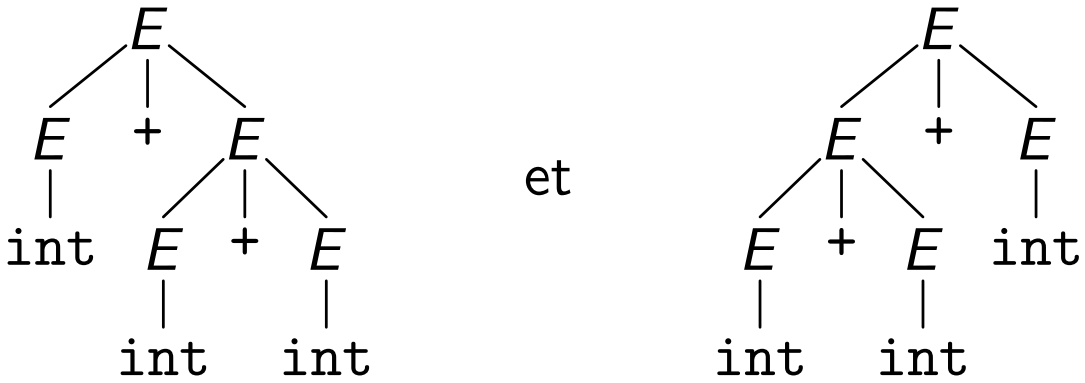
In the file `arith.automaton` we can find description:

```
State 6:
expression -> expression . PLUS expression [ RPAR PLUS EOF ]
expression -> expression PLUS expression . [ RPAR PLUS EOF ]
- On PLUS shift to state 5
- On RPAR reduce production expression -> expression PLUS expression
- On PLUS reduce production expression -> expression PLUS expression
- On EOF reduce production expression -> expression PLUS expression
** Conflict on PLUS
```

And the file `arith.conflicts` gives a more detailed explanation:

```
** Conflict (shift/reduce) in state 6.
** Token involved: PLUS
** This state is reached from file after reading:

expression PLUS expression

** In state 6, looking ahead at PLUS, shifting is permitted
** because of the following sub-derivation:

expression PLUS expression
                  expression . PLUS expression

** In state 6, looking ahead at PLUS, reducing production
** expression -> expression PLUS expression
** is permitted because of the following sub-derivation:

expression PLUS expression // lookahead token appears
```

Said otherwise, the conflict arises because we have a choice



One way to resolve the conflicts is to indicate to Menhir what to choose between shift and reduce.

This is done, by giving priorities to the tokens and productions,
and by giving the associativity rules.

(by default, the priority of a production is the one of its rightmost token,
but it can also by specified explicitly by the user)

If the priority of the production is higher than that of a token to read, then it's the reduction that is performed.

If it's the priority of the token that is higher,
then it's shift that is chosen.

(the associativity is used to resolve the conflict in case of equality: tokens with left associativity favor reduction and tokens with right associativity favor shift).

Back to Example

We solve the conflict by indicating that PLUS is left associative

```
%token PLUS LPAR RPAR EOF
%token <int> INT
%left PLUS
%start <expr> file
%%
file:
| e = expression; EOF  { e }
;
expr:
| i = INT                        { Cte i }
| e1 = expr; PLUS; e2 = expr { Add (e1, e2) }
| LPAR; e = expr; RPAR           { e }
;
```

We use the following convention:

- the order in which priorities are declared fixes the priorities: (first tokens have the weakest priorities)
- several tokens can be mentioned on the same line, having therefore the same priority

Example :

```
%left PLUS MINUS
%left TIMES DIV
```

A Classic

The following grammar has a conflict

```
expression:
| IF e1 = expression; THEN; e2 = expression
    { ... }
| IF e1 = expression; THEN; e2 = expression;
  ELSE; e3 = expression
    { ... }
| i = INT
    { ... }
| ...
```

(known as *dangling else*)

The conflict corresponds to the situation

```
IF a THEN IF b THEN c ELSE d
```

To associate the ELSE to the nearest THEN, we need to prioritize the shift action

```
%nonassoc THEN
%nonassoc ELSE
```

Localization

To enable subsequent analysis phases (e.g. typing) to locate error messages, we need to retain localization info in the abstract syntax tree.

`menhir` provides this info in `$startpos` and `$endpos`, two values of type `Lexing.position`; this information is passed to it by the lexical analyzer.

Note: `ocamllex` maintains automatically only the *absolute position* in the file; to have up-to-date line and column numbers, you need to call `Lexing.new_line` for each carriage return.

One way to preserve localization information in the abstract syntax tree is as follows:

```
type expression =
  { desc: desc;
    loc : Lexing.position * Lexing.position }

and desc =
  | Econst of int
  | Eplus  of expression * expression
  | Eneg   of expression
  | ...
```

Each node is thus decorated by a localization.

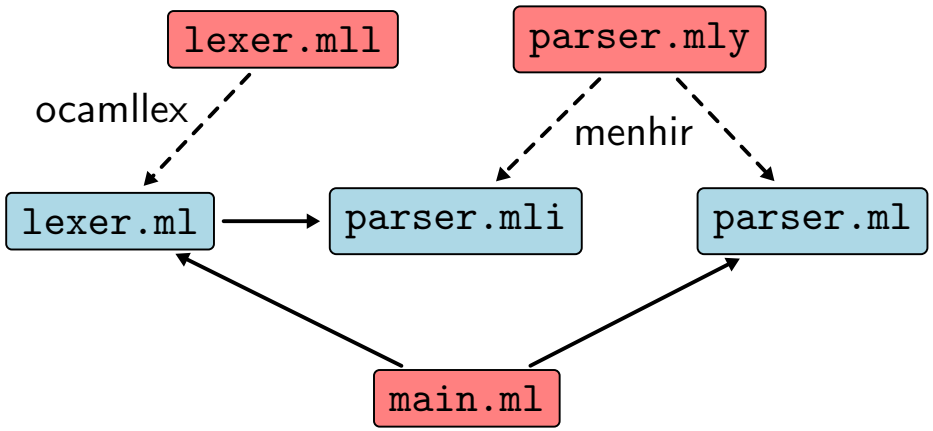
A grammar can then look like

```
expression:
| d = desc { { desc = d; loc = $startpos, $endpos } }
;

desc:
| i = INT                                    { Econst i       }
| e1 = expression; PLUS; e2 = expression { Eplus (e1, e2) }
| ...
```

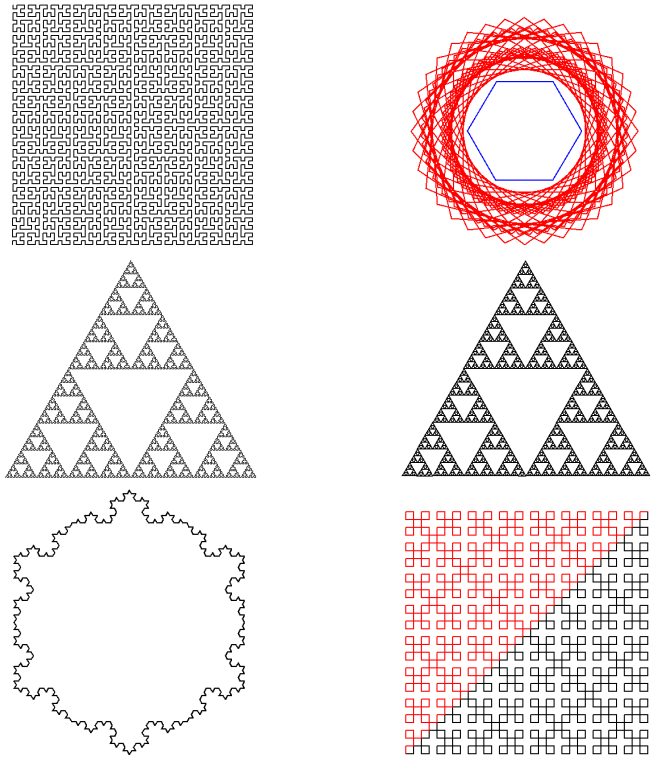
Just as with `ocamllex`, one need to apply `menhir` to the parser file, before using it in the compiler

With `dune` manager, we indicate the presence of the `menhir` file by:

```
(ocamllex
 (modules lexer))
(menhir
 (flags -explain -dump)
 (modules parser))
(executable
 (name minilang)
 ...
```



- lexical and syntax analysis of `mini-Turtle` drawing language
- (interpreter is given)



Bibliography

- Compilers, Principles, Techniques, and Tools, Alfred Aho and Monica S. Lam and Ravi Sethi and Jeffrey D. Ullman, Second Edition[Chapter 4.7], Addison-Wesley (2006) (*"the dragon book"*).