50.051 Programming Language Concepts

W10-S2 & W10-S3 Top-Down Parsing

Matthieu De Mari

The unspoken problem #2

Given a CFG and a string *x* whose syntax needs to be verified

- We understand how to use production rules and apply them manually to eventually find the string *x*.
- We understand how to derive a parse tree for the derivation.
- We understand how to add syntax-directed rules to our production rules and use them in the derivation.
- We understand how to use syntax-directed rules to transform our parse tree into an AST.
- We understand that the AST will give us the machine code translation, or at least something already close to what the CPU would need to execute to obtain the outcome of our source code.

The unspoken problem

For syntax analysis, we will now need

- **An algorithm to find if a given sequence of tokens *x* has a valid syntax and admits a valid derivation** for our given CFG,
- *(Also known as the **parsing** problem).*

Small tiny teeny issue: This is absolutely not an easy problem algorithmically speaking! (It is very much NP-Hard!)
Will take us the next three lectures…!
And we will not even dare to implement said algorithms…!

Different types of Parsing

Two big families of parsing algorithms

Top-Down Parsing

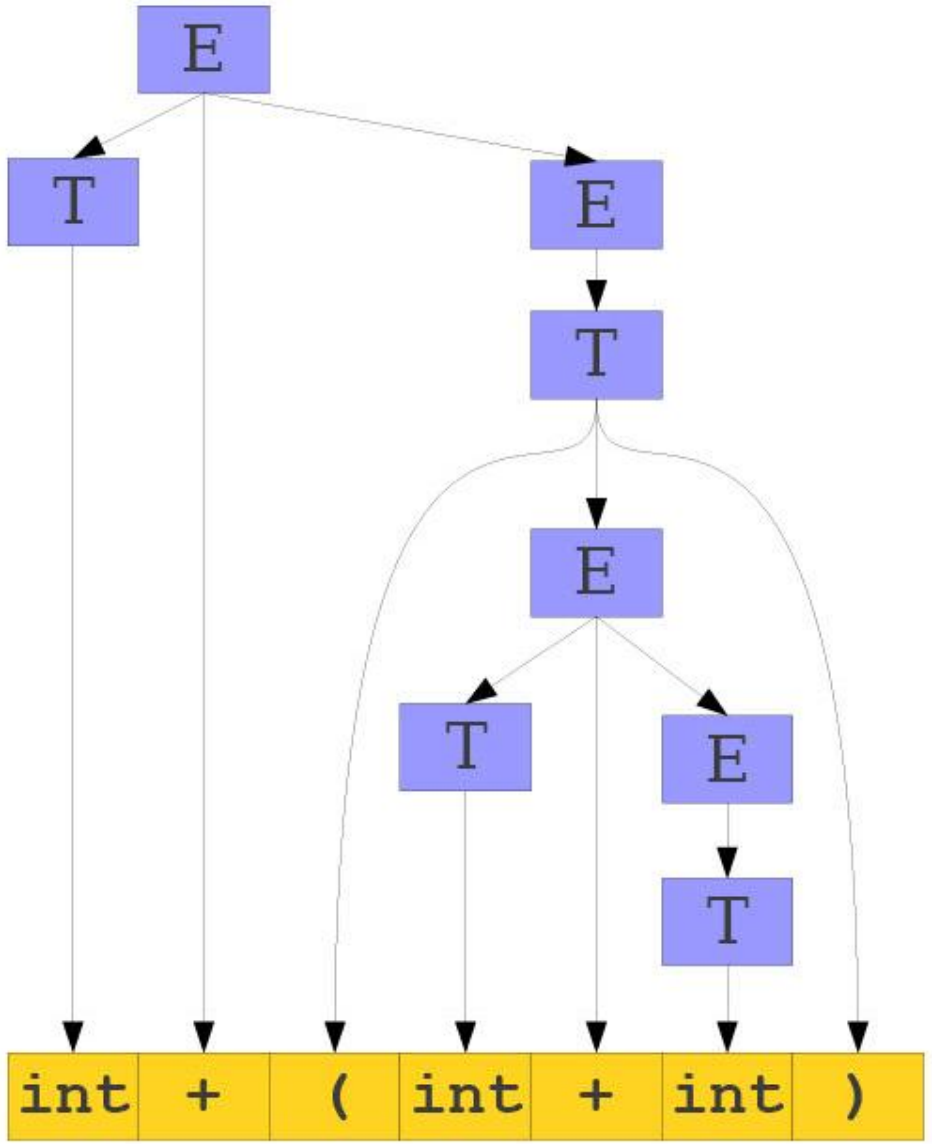
- To be discussed in W10S1.
- Most intuitive approach.
- Begin with the start symbol, try to guess the production rules to apply and end up with the stream of symbols you want.

Bottom-Up Parsing

- Something for later (W10S2-3).

Top-Down Parsing

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



Challenges in Top-Down Parsing

Top-down parsing begins with virtually no information.

- Begins with just the start symbol, which matches every program.

How can we know which productions to apply?

- In general, we cannot.

Conclusion

Good luck for the exam!

Challenges in Top-Down Parsing

Top-down parsing begins with virtually no information.

- Begins with just the start symbol, which matches every program.

How can we know which productions to apply?

- In general, we cannot.
- There are some grammars for which the best we can do is guess and backtrack if we are wrong.
- If we have to guess, how do we do it?

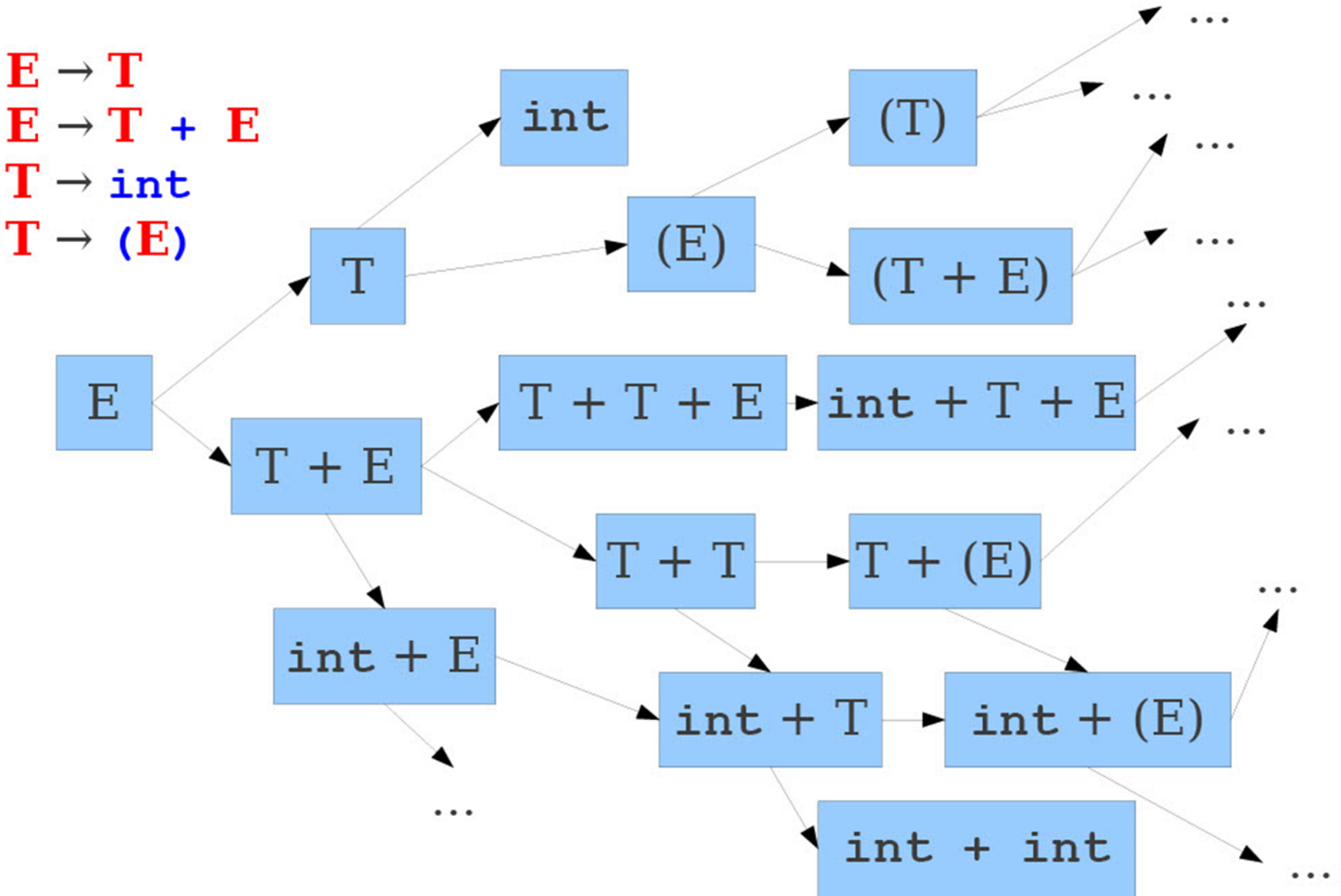
Top-Down as a Graph Search

First idea: treat parsing as a **graph search**.

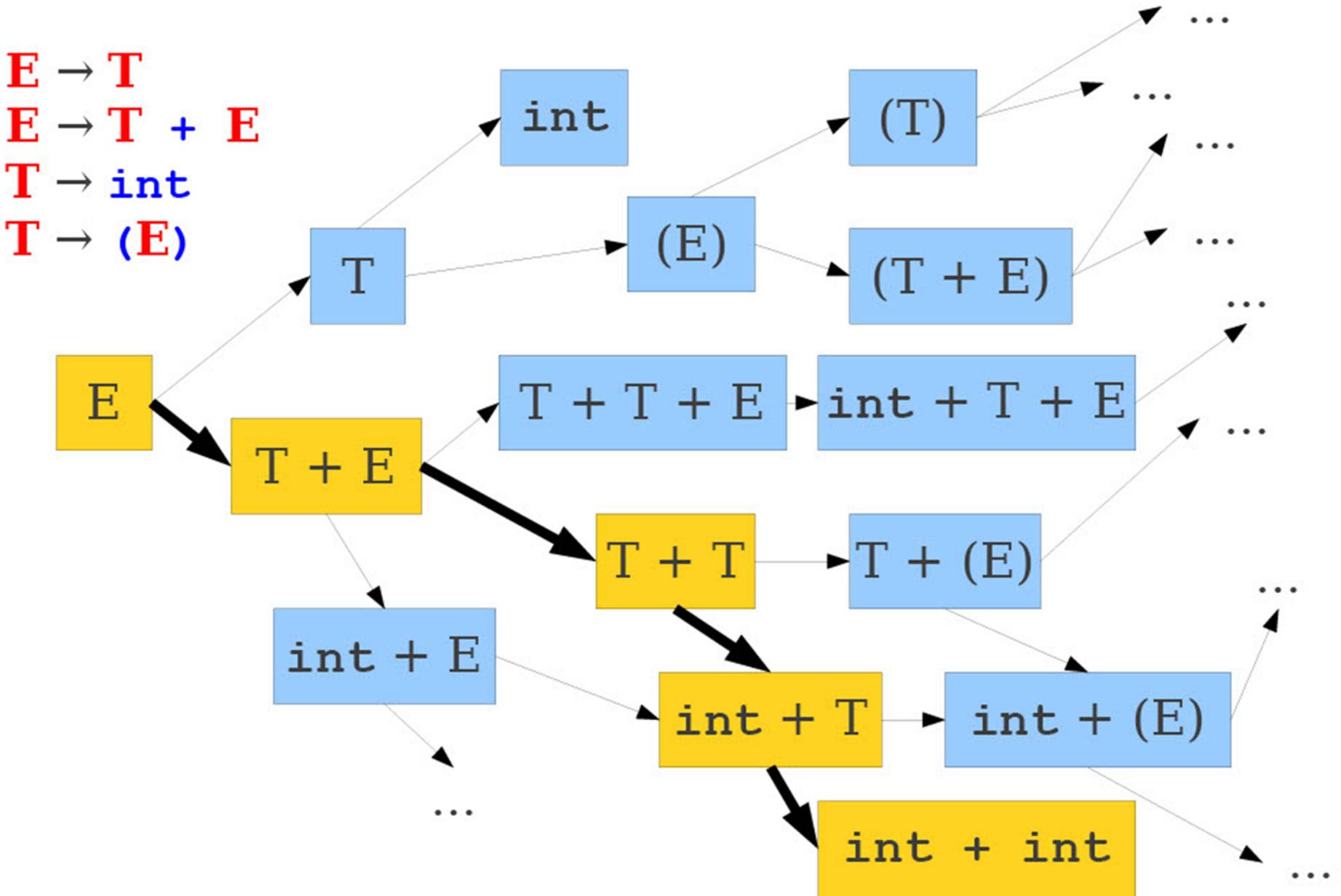
- Each node is a sentential form (a string of terminals and non-terminals symbols derivable from the start symbol).
- There is an edge from node α to node β iff α can be transformed into β using a production rule from the CFG.

Objective: Find if the graph of all possible applications of the CFG rules eventually admits a node whose sentential form corresponds to our string *x*.

Graph Search Problem



Graph Search Problem



BFS in action: demo on a simple CFG

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

```
int + int
```

BFS in action: demo on a simple CFG

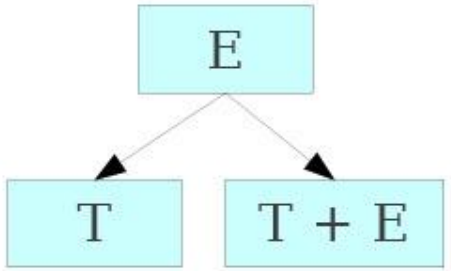
E

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

```
int + int
```

BFS in action: demo on a simple CFG

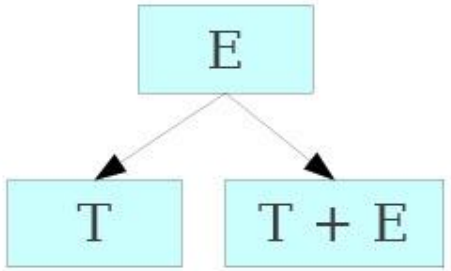
Worklist



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

`int + int`

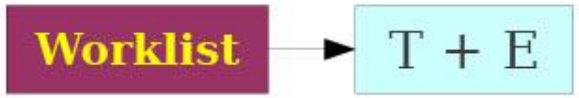
BFS in action: demo on a simple CFG



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

```
int + int
```

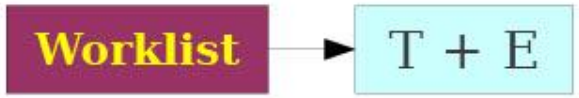
BFS in action: demo on a simple CFG

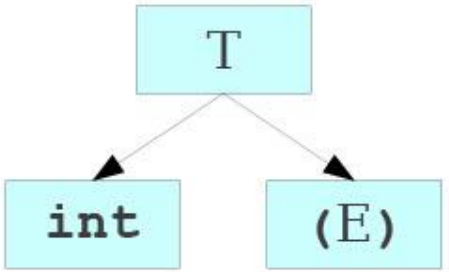


$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

```
int + int
```

BFS in action: demo on a simple CFG

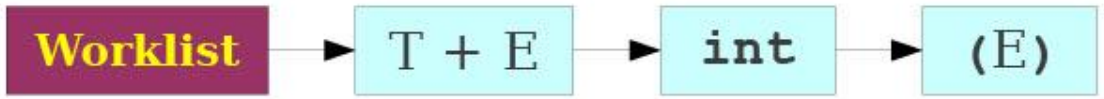


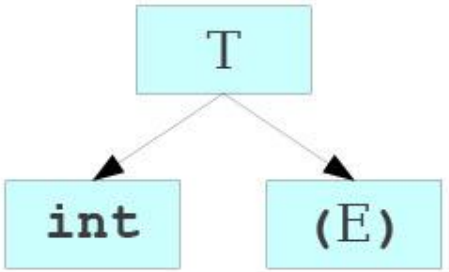


E → T
E → T + E
T → int
T → (E)

```
int + int
```

BFS in action: demo on a simple CFG

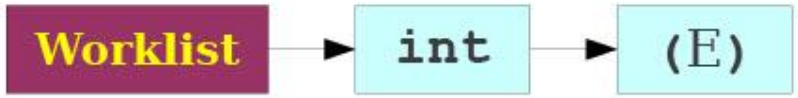


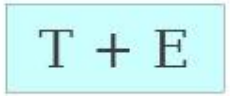


$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

`int + int`

BFS in action: demo on a simple CFG

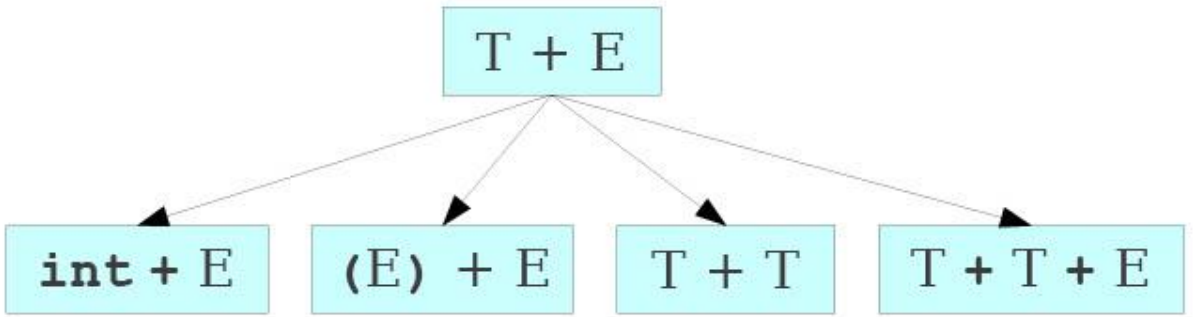




$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

`int + int`

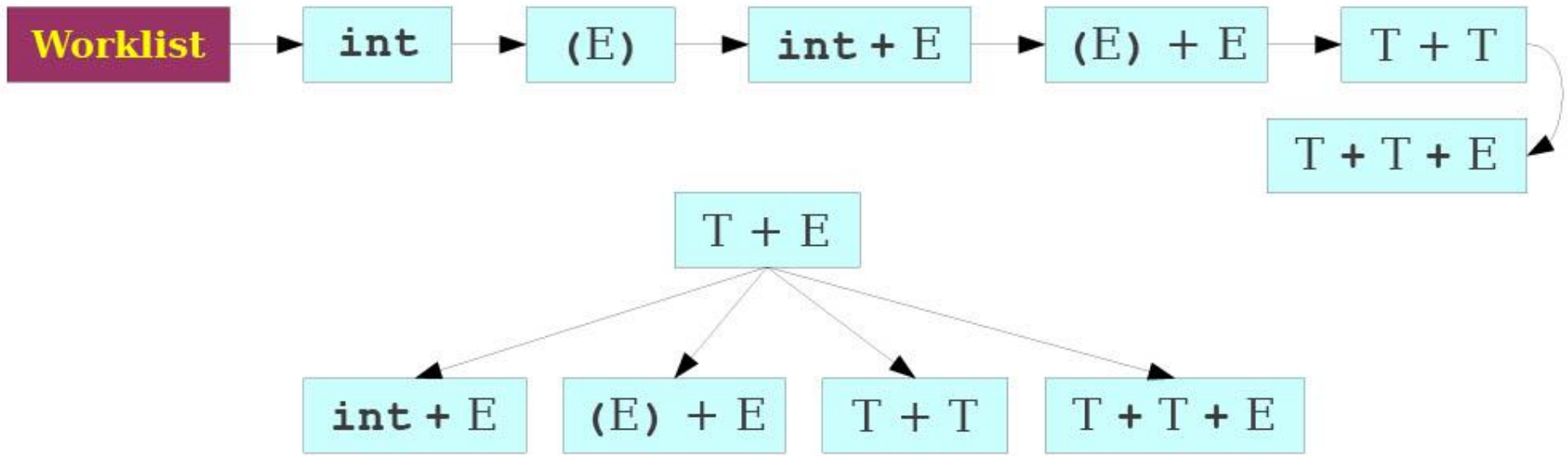
BFS in action: demo on a simple CFG



E → T
E → T + E
T → int
T → (E)

int + int

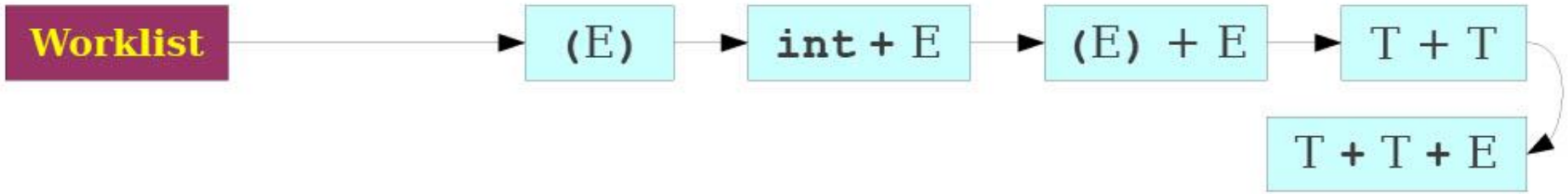
BFS in action: demo on a simple CFG



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

int + int

BFS in action: demo on a simple CFG

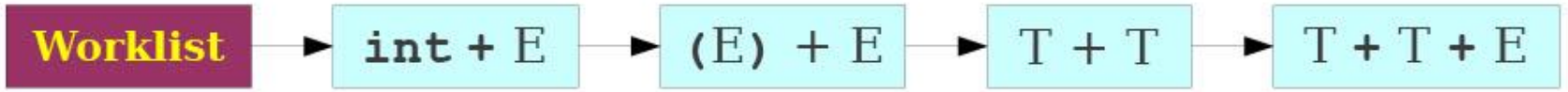


int

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

int + int

BFS in action: demo on a simple CFG

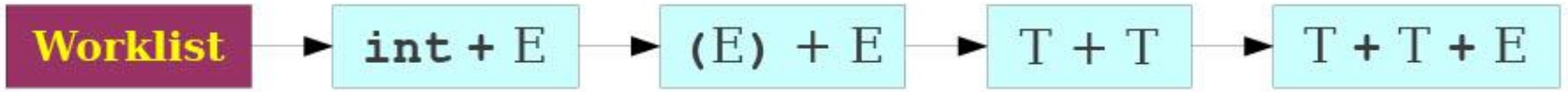


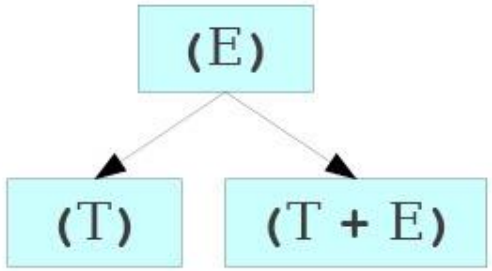
(E)

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

int + int

BFS in action: demo on a simple CFG

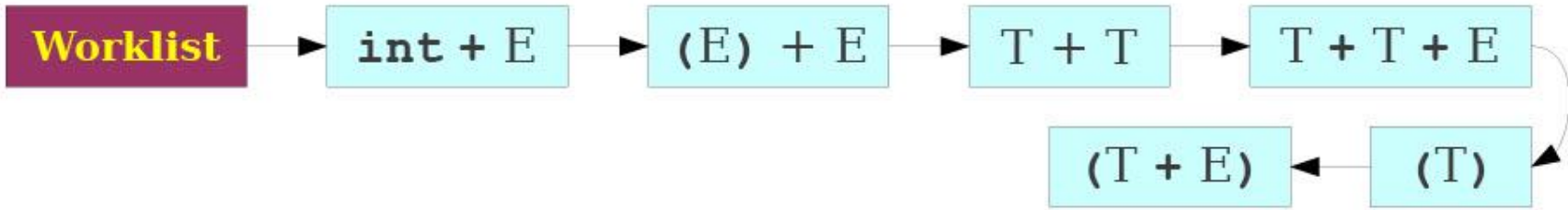


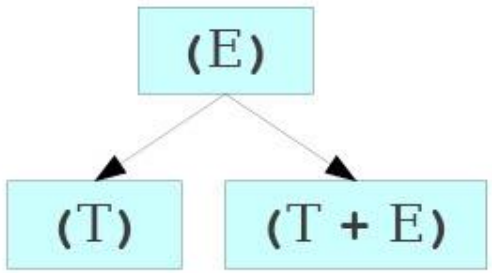


E → T
E → T + E
T → int
T → (E)

int + int

BFS in action: demo on a simple CFG





$E \rightarrow T$

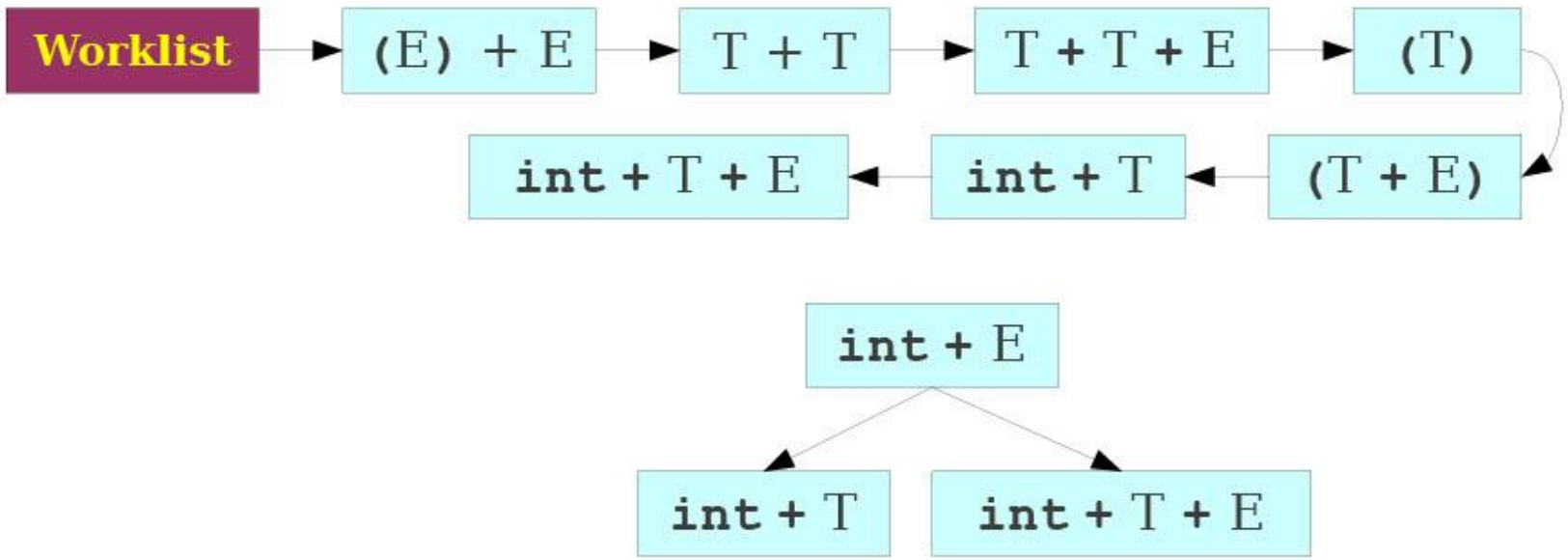
$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

```
int + int
```

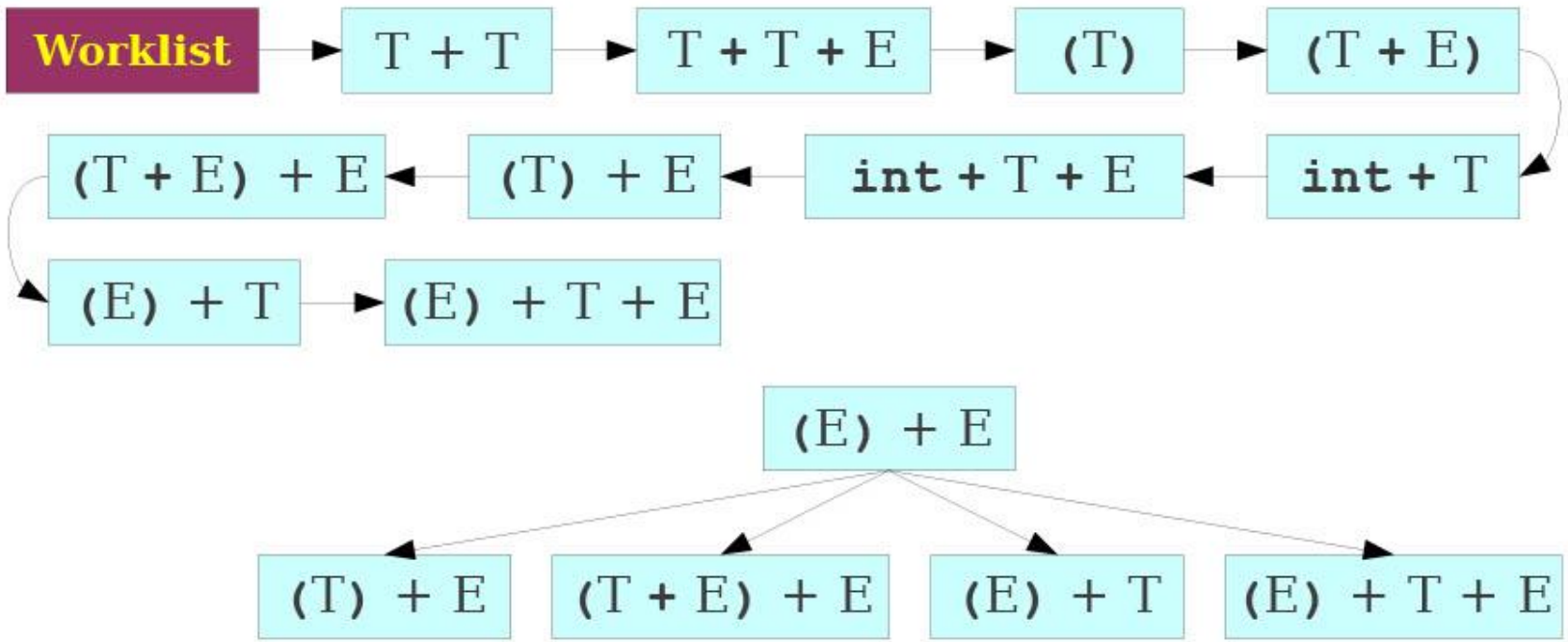
BFS in action: demo on a simple CFG



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

`int + int`

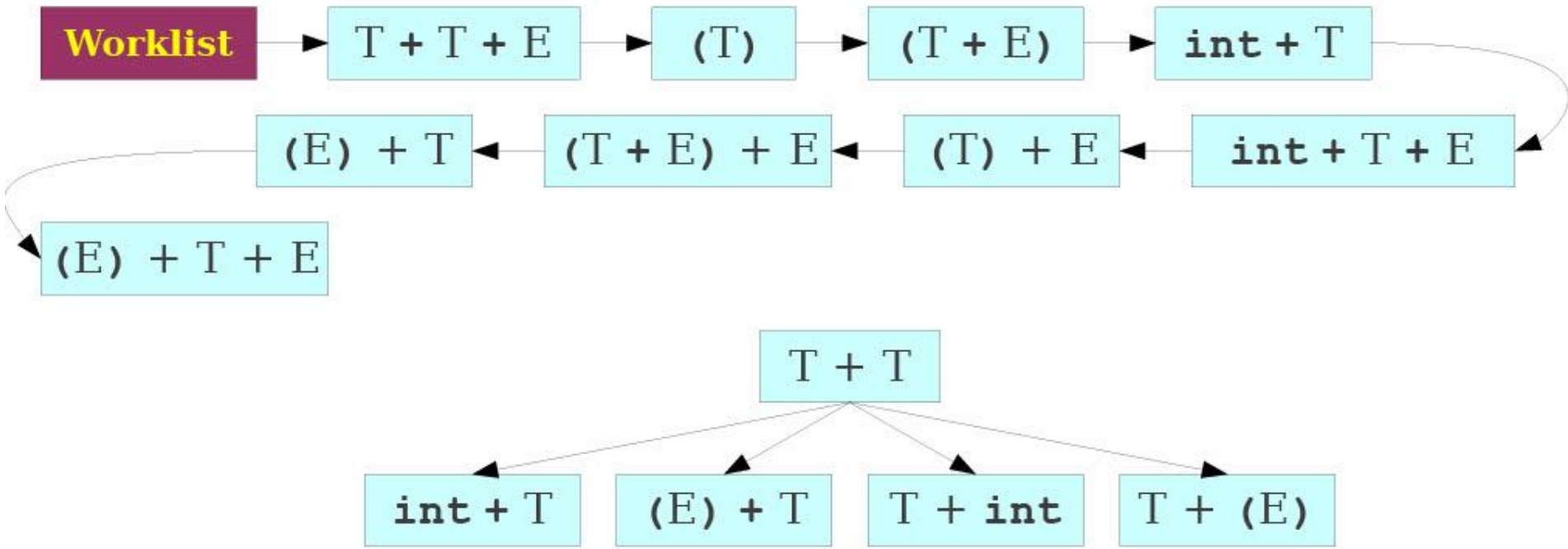
BFS in action: demo on a simple CFG



E → T
E → T + E
T → int
T → (E)

int + int

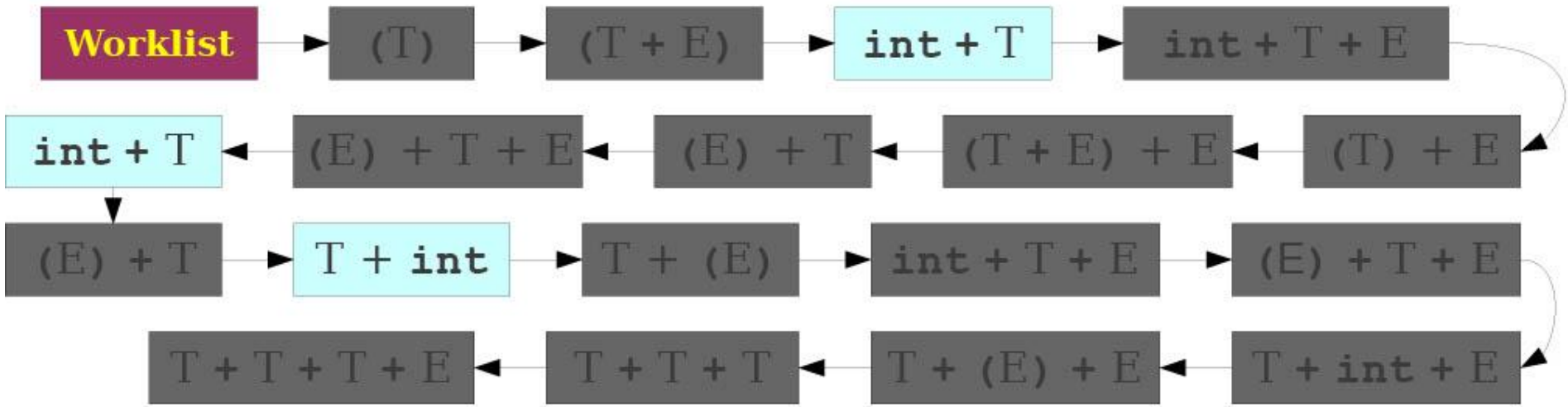
BFS in action: demo on a simple CFG



E → T
E → T + E
T → int
T → (E)

int + int

BFS in action: demo on a simple CFG



$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

BFS is slow, but guaranteed to work

Good: Guaranteed to work? But equivalent to “brute-force”?

Problem: time and memory usage due to lots of wasted effort…

- BFS generates a lot of sentential forms that could not possibly match.
- But in general, it is extremely hard to tell whether a sentential form can match…
- That is the job of parsing!
- **Complexity comes from a high branching factor:** Each sentential form can expand in (potentially) many ways for each non-terminal it contains.

Reducing the branching factor: Idea #1

- Suppose we are trying to match a sequence of tokens γ.
- Suppose we have a sentential form $\tau = \alpha\omega$, where α is a string of terminals and ω is a string of terminals and non-terminals.
- If α isn't a prefix of γ, then no string derived from τ can ever match γ.
- For instance, if $\alpha = int$, $\omega = + E$ and our string to match is $\gamma = (int + int) * int$, then $\tau = int + E$ cannot derive into γ.

Idea #1: If we can find a way to try to get a prefix of terminals at the front of our sentential forms, then we can start pruning out impossible options and reduce the branching factor in our graph.

Leftmost derivation

Definition (Leftmost Derivation):

In parsing, a **Leftmost Derivation** is a derivation where we always try to **transform the leftmost non-terminal first**.

For instance, consider that the current string is T + E, with T and E being non-terminals. The leftmost derivation will suggest to transform T first, and figure out how to transform E later.

Similarly, we can define a **Rightmost Derivation**, where we always try to transform the rightmost non-terminal first.

Updated algo: Leftmost BFS

Updated Algo #1: Do a BFS, only considering leftmost derivations.

1. Dramatically drops branching factor.
2. Increases likelihood that we get a prefix of non-terminals. When that happens, you can prune sentential forms that can't possibly match, to reduce the branching factor and avoid wasted effort.

Remember,

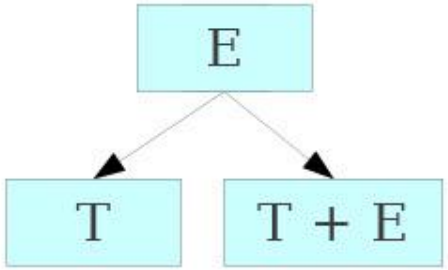
- If we have a sentential form $\tau = \alpha\omega$, where α is a string of terminals and ω is a string of terminals and non-terminals.
- And if α isn't a prefix of γ, then no string derived from τ can ever match γ. Close that branch in the graph!

Leftmost BFS in action

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

```
int + int
```

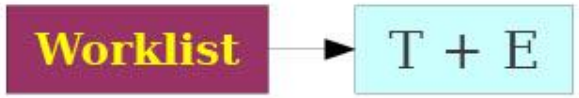
Leftmost BFS in action

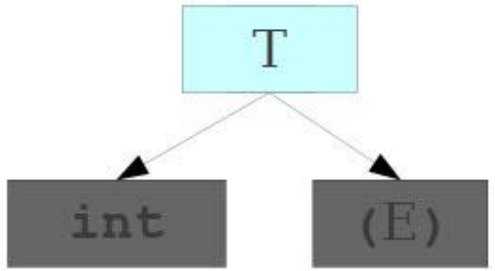


$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

`int + int`

Leftmost BFS in action



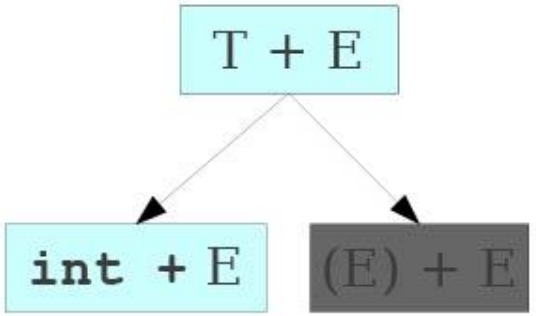


E → T
E → T + E
T → int
T → (E)

`int + int`

Leftmost BFS in action

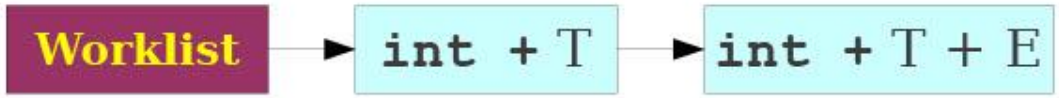
Worklist

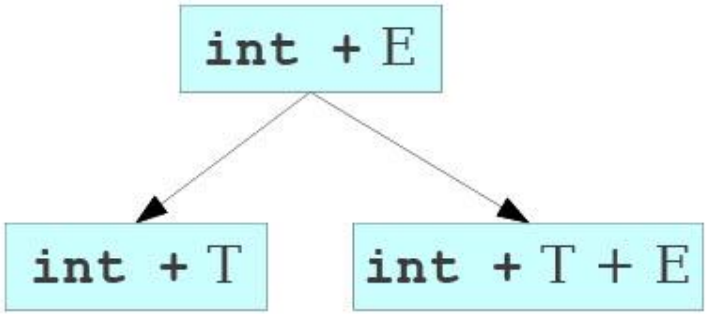


$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

`int + int`

Leftmost BFS in action

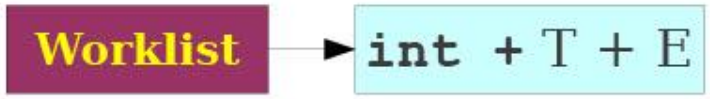


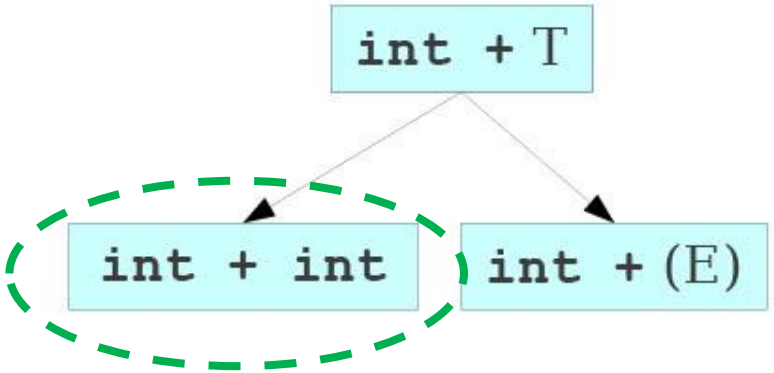


E → T
E → T + E
T → int
T → (E)

int + int

Leftmost BFS in action





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

int + int

Leftmost BFS: improvements and issues

Substantial speed, performance and memory improvement over the naïve BFS algorithm.

- Uses the pruning property we established earlier.
- Can more rapidly find a valid parse of a program if one exists.
- Can easily be modified to find if a program cannot be parsed.

Unfortunately, there are still many problems.

Practice 1: Leftmost BFS problem.

Consider the CFG below.

$$A \rightarrow Aa$$
$$A \rightarrow Ab$$
$$A \rightarrow c$$

Draw the result of the graph search for the Leftmost BFS algorithm trying to match the string *x = "caaaaaaaa".*

What seems to be the issue here?

Practice 1: Leftmost BFS problem.

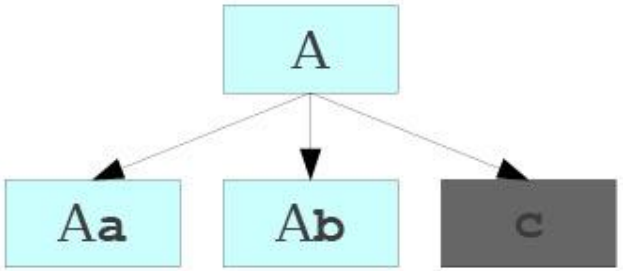
Worklist

A

$$\mathbf{A} \rightarrow \mathbf{Aa} \mid \mathbf{Ab} \mid \mathbf{c}$$

caaaaaaaaaa

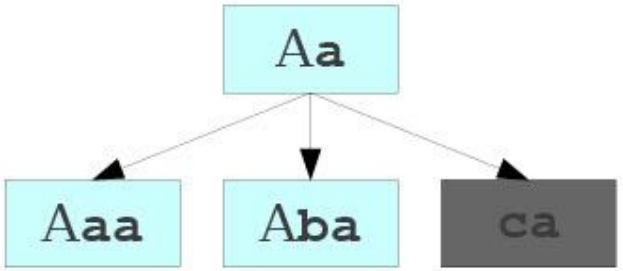
Practice 1: Leftmost BFS problem.



$$A \rightarrow Aa \mid Ab \mid c$$

caaaaaaaaa

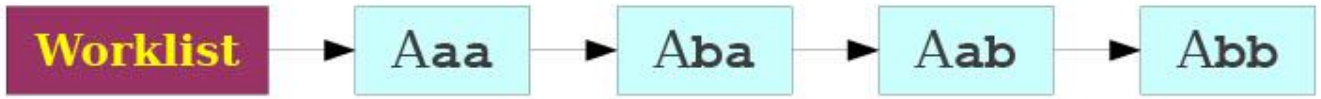
Practice 1: Leftmost BFS problem.

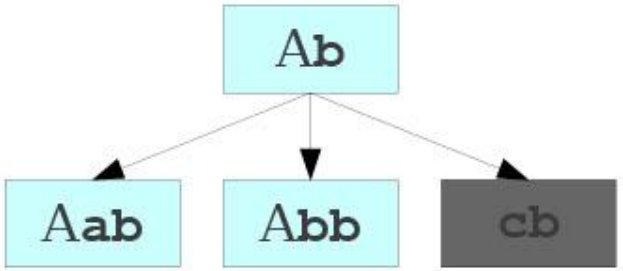


$$A \rightarrow Aa \mid Ab \mid c$$

caaaaaaaaaa

Practice 1: Leftmost BFS problem.





$$A \rightarrow Aa \mid Ab \mid c$$

caaaaaaaaaa

Practice 1: Leftmost BFS problem.

Consider the CFG below.

$$A \rightarrow Aa$$
$$A \rightarrow Ab$$
$$A \rightarrow c$$

What seems to be the issue here?

Grammar like these will lead to exponential computation times for Leftmost BFS!

But many CFGs have to follow this format to establish precedence or associativity (Refer to W9S2-3 lectures!)

Using DFS instead of BFS?

Crazy idea: How about using DFS instead of BFS then?

- Has a lower memory usage as it only considers one branch at a time.
- On many grammars it will run very quickly
- And it is fairly easy to implement as a set of mutually recursive functions.

Worth a try?

Leftmost DFS in action

E

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

int + int

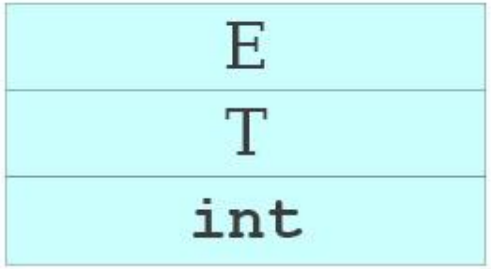
Leftmost DFS in action

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


```
int + int
```

Leftmost DFS in action

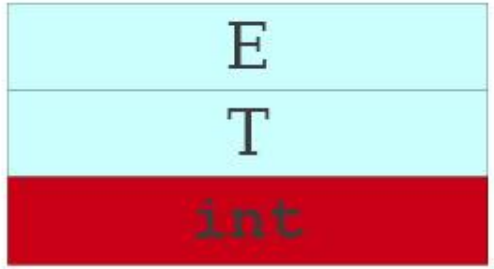
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



`int + int`

Leftmost DFS in action

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



`int + int`

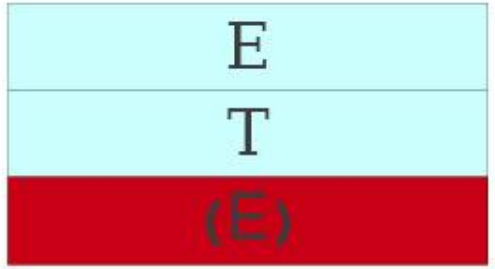
Leftmost DFS in action

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


```
int + int
```

Leftmost DFS in action

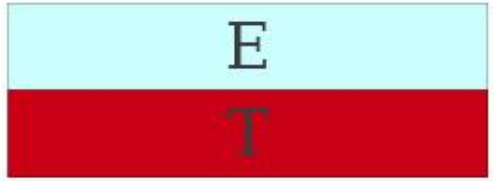
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$




```
int + int
```

Leftmost DFS in action

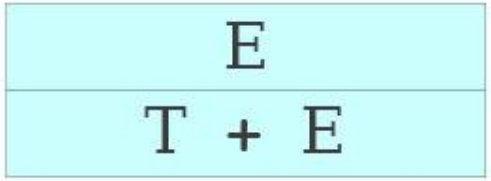
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



`int + int`

Leftmost DFS in action

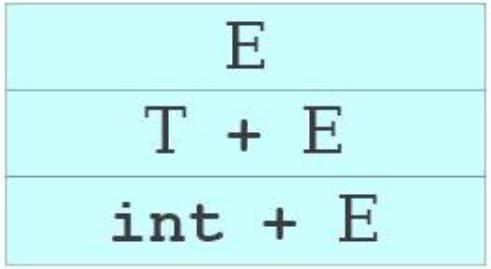
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$




```
int + int
```

Leftmost DFS in action

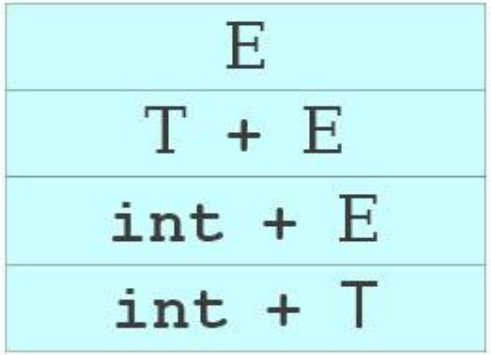
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



`int + int`

Leftmost DFS in action

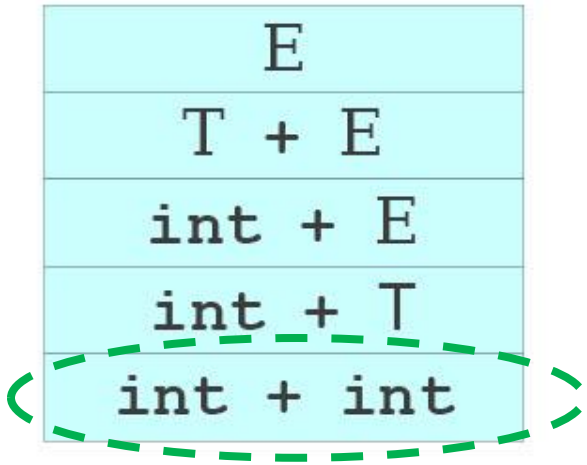
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



`int + int`

Leftmost DFS in action

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



int + int

Practice 2: Leftmost DFS problem.

Consider the very simple CFG below.

$$A \rightarrow Aa$$
$$A \rightarrow c$$

Q1. What are the acceptable strings according to this CFG? Would that correspond to a RegEx of some sort?

Q2. Draw the result of the graph search for the Leftmost DFS algorithm trying to match the string *x = "c".*

What seems to be the issue here?

Practice 2: Leftmost DFS problem.

Consider the very simple CFG below.

$$A \rightarrow Aa$$
$$A \rightarrow c$$

Q1. What are the acceptable strings according to this CFG? Would that correspond to a RegEx of some sort?

Any string that start with exactly one “c” and is followed by zero or more “a”. That would be equivalent to the RegEx “^ca*$”.

Practice 2: Leftmost DFS problem.

Consider the very simple CFG below.

$$A \rightarrow Aa$$
$$A \rightarrow c$$

Q2. Draw the result of the graph search for the Leftmost DFS algorithm trying to match the string *x = "c".*

What seems to be the issue here?

Aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

$$A \rightarrow Aa \mid c$$

A
Aa
Aaa
Aaaa
Aaaaa

c ?

The problem with Leftmost BFS and DFS

Both Leftmost BFS and Leftmost DFS have a problem that follows from having a CFG that contains a **left-recursive non-terminal symbol.**

Definition (Left-recursive non-terminal symbol in a CFG):

In a CFG, a non-terminal symbol A is said to be **left-recursive**, if and only if **there exists a left-recursive production rule, typically written as**

$$A \rightarrow A\alpha$$

With α being a string of terminal and non-terminal symbols.

Similarly, we can define **right-recursive**.

The problem with Leftmost BFS and DFS

Both Leftmost BFS and Leftmost DFS have a problem that follows from having a CFG that contains a **left-recursive non-terminal symbol.**

- **Leftmost BFS:** Takes exponential time and memory usage in presence of a left-recursive production rule.
- **Leftmost DFS:** Fails, plain and simple.
- **Rightmost BFS and DFS will have the same problems in presence of a right-recursive production rule.**

Serious issue: It might not always be easy or even possible to remove left- or right-recursive production rules from a given CFG, as it often serves for defining left- and right-associativity in certain operators!

The problem with Leftmost BFS and DFS

Both Leftmost BFS and Leftmost DFS have a problem that follows from having a CFG that contains a **left-recursive non-terminal symbol.**

- **Leftmost BFS:** Takes exponential time and memory usage in presence of a left-recursive production rule.
- **Leftmost DFS:** Fails, plain and simple.
- **Rightmost BFS and DFS will have the same problems in presence of a right-recursive production rule.**

Conclusion: Using Leftmost/Rightmost BFS/DFS is not going to help with our parsing problem. Need more sophisticated algorithms.

Quiz time!

What is the difference between leftmost and rightmost derivations?

A. Leftmost derivations start with the rightmost non-terminal and work towards the leftmost non-terminal, while rightmost derivations do the opposite.

B. Leftmost derivations start with the leftmost non-terminal and work towards the rightmost non-terminal, while rightmost derivations do the opposite.

C. Leftmost derivations and rightmost derivations are the same thing, what matters is whether BFS or DFS is used only.

D. Leftmost derivations are used to describe languages that read from left to right (e.g. English), rightmost do the opposite (e.g. Arabic).

Quiz time!

What is the difference between leftmost and rightmost derivations?

A. Leftmost derivations start with the rightmost non-terminal and work towards the leftmost non-terminal, while rightmost derivations do the opposite.

B. Leftmost derivations start with the leftmost non-terminal and work towards the rightmost non-terminal, while rightmost derivations do the opposite.

C. Leftmost derivations and rightmost derivations are the same thing, what matters is whether BFS or DFS is used only.

D. Leftmost derivations are used to describe languages that read from left to right (e.g. English), rightmost do the opposite (e.g. Arabic).

Quiz time!

When parsing, which algorithm is the best?

A. BFS, always because it is guaranteed to converge.

B. DFS, because it is guaranteed to converge, it is also faster and more efficient than BFS.

C. When doing the graph search, we should in fact use DFS on some branches and BFS on some other branches of the graph.

D. Honestly, all these algorithms have problems.

Quiz time!

When parsing, which algorithm is the best?

A. **BFS, always because it is guaranteed to converge. (Yes, but it has another problem, its heavy memory and computational cost)**
B. **DFS, because it is guaranteed to converge, it is also faster and more efficient than BFS (Incorrect, it will fail miserably with recursive grammars)**
C. **When doing the graph search, we should in fact use DFS on some branches and BFS on some other branches of the graph. (And how exactly do you plan to do that?)**
D. **Honestly, all these algorithms have problems.**

Quiz time!

Which of the following is a left-recursive grammar?

A. S → aS | b

B. S → AB, A → aAa | a, B → bBb | b

C. S → Sb | a | b

D. S → Aa | bB, A → SA | a, B → SB | b, S → a | b

Quiz time!

Which of the following is a left-recursive grammar?

A. S → aS | b

B. S → AB, A → aAa | a, B → bBb | b

C. S → Sb | a | b

D. S → Aa | bB, A → SA | a, B → SB | b, S → a | b

(S could be seen as a recursive symbol but we need to use two production rules in a row, being S → Aa and A → SA?)

Predictive Parsing to the rescue?

The Leftmost/Rightmost BFS/DFS algorithms are backtracking algorithms.

- Try productions and backtrack if they do not work.
- Try to match prefix by sheer dumb luck, really.

Can we do better?

Yes! With another class of parsing algorithms called **predictive parsers**.

Core idea for predictive parsers: At any given time, use the remaining input to predict, without backtracking, which production rule should be used.

Predictive Parsing to the rescue?

Predictive parsers are fast.

- Many predictive parsing algorithms can be made to run in linear time.
- As will be seen later, can be table-driven for extra performance.

Predictive parsers are weak.

- Not all grammars can be accepted by predictive parsers, and these grammars will have to be readjusted manually.
  (Remember, designing a good CFG is not algorithmically possible!)
- Will often trade expressiveness for speed.

Predictive Parsing to the rescue?

Central question for predictive parsers: Given just the start symbol, how do you know which productions to use to get to the input string?

Idea: Use a lookahead predictive procedure.

- When trying to decide which production rule to use to replace the leftmost non-terminal symbol, **look at the next *k* symbols of the input to help make the decision of which production rules to use or eliminate.**
- **Corollary: Predictive parsing is only possible if the CFG allows to predict which production to use given some lookahead symbols. *(This might not be possible with all CFGs!)***

An illustrated Predictive Parser at play

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

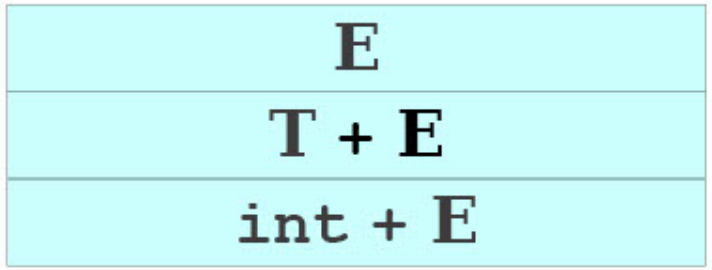
An illustrated Predictive Parser at play

E
T + E

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

int	+	(	int	+	int	)

An illustrated Predictive Parser at play



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

An illustrated Predictive Parser at play

E
T + E
int + E
int + T

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

int	+	(	int	+	int	)

An illustrated Predictive Parser at play

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

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

int	+	(	int	+	int	)

An illustrated Predictive Parser at play

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

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

int	+	(	int	+	int	)

An illustrated Predictive Parser at play

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

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

int	+	(	int	+	int	)

An illustrated Predictive Parser at play

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

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

int	+	(	int	+	int	)

An illustrated Predictive Parser at play

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

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

int	+	(	int	+	int	)

An illustrated Predictive Parser at play

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

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

int	+	(	int	+	int	)

Naming convention for predictive parsers

Top-down predictive parsing has a **naming convention**:

- First letter, L: Left-to-right scan of the symbols (R, if right-to-left).
- Second letter, L: Leftmost derivation (R, if rightmost instead).
- Number (1): One symbol of lookahead (could be another number *k*).

LL(1): Construct a leftmost derivation (.L), scanning symbols from left to right (L.), for the sequence of symbols.

- When replacing a non-terminal, we predict the production rule to use by looking at the next symbol (1) of the input.
- **The decision on which production rule to use is then forced.**
- Our previous demo was technically LL(2), two yellow blocks needed.

LL(1) will often define some parse tables

How to read the LL(1) table.

- **Line** = Which non-terminal symbol is about to be replaced.
- **Column** = What is the next terminal symbol in x that must match this non-terminal.
- **In the cell** = What to replace the non-terminal symbol (line) with.

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

	int	(	)	+	*
E	int	(E Op E)			
Op				+	*

LL(1) at work!

E$	(int + (int * int))$

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

	int	(	)	+	*
E	1	2			
Op				3	4

Notes

- The Table on the left now contains the index of the production rule to use to make the replacement (equivalent to previous table)
- Additional symbol: using the $ symbol to indicate the end of the two strings (current string being parsed and target string to match).

LL(1) at work!

E$	(int + (int * int))$

(1) **E** → **int**
(2) **E** → **(E Op E)**
(3) **Op** → **+**
(4) **Op** → *****

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) at work!

(1) **E** → **int**
(2) **E** → **(E Op E)**
(3) **Op** → **+**
(4) **Op** → *****

	int	(	)	+	*
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$

LL(1) at work!

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

	int	(	)	+	*
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$

LL(1) at work!

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

	int	(	)	+	*
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$

LL(1) at work!

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

	int	(	)	+	*
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$

LL(1) at work!

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

	int	(	)	+	*
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$

LL(1) at work!

(1) **E** → `int`
(2) **E** → `(` **E Op E** `)`
(3) **Op** → `+`
(4) **Op** → `*`

	int	(	)	+	*
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$

LL(1) at work!

(1) **E** → **int**
(2) **E** → **(E Op E)**
(3) **Op** → **+**
(4) **Op** → *****

	int	(	)	+	*
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$
))$	))$
)$	)$
$	$

LL(1) will often define some parse tables

How to read the LL(1) table.

- **Line** = Which non-terminal symbol is about to be replaced
- **Column** = What is the next terminal symbol in x that must match this non-terminal
- **In the cell** = What to replace the non-terminal symbol (line) with.

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

	int	(	)	+	*
E	int	(E Op E)			
Op				+	*

Empty cell? = Invalid syntax

Detecting syntax errors with LL(1): mismatch

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

	int	(	)	+	*
E	1	2			
Op				3	4

E$	int + int$
int $	int + int$
$	+ int$

Here “int + int” is invalid according to our CFG (it says we should use parentheses with addition, always). The parsed string produced by LL(1) eventually runs out of non-terminal symbols before the input string *x* does. That is a **mismatch (syntax) error**.

Detecting syntax errors with LL(1): no prediction

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

	int	(	)	+	*
E	1	2			
Op				3	4

E$	(int (int))$
(E Op E)$	(int (int))$
E Op E)$	int (int))$
int Op E)$	int (int))$
Op E)$	(int))$

Here “(int (int))” is invalid according to our CFG (can only use parentheses with plus symbols, not without).
The LL(1) eventually refers to an empty cell in the table, which indicates that there is no way to parse the string further. **Syntax error!**

The LL(1) algorithm, detailed

Suppose our CFG has start symbol S and we have manually defined a LL(1) parsing table T.

We want to parse string ω.

The LL(1) algorithm steps are as follows

1. Initialize a stack #1 with “S$”
2. Initialize a stack #2 containing the characters in the string ω, terminate with $.
3. Repeat 4, 5, 6 (next slide) until one of the stacks becomes empty (only $ left). **If both are empty at the same time, success!**

The LL(1) algorithm, detailed

4. Draw character *t* from the top of stack #2.
5. If the top of the stack #1 is a terminal symbol *r*:
   - If *r* and *t* do not match, **report a syntax error, end the derivation.**
   - Otherwise remove the character *t* and pop *r* from the stack.
6. Otherwise, the top of the stack #1 is a non-terminal symbol A:
   - If T[A, t] is empty, no prediction can be made, **report a syntax error, end the derivation**.
   - Otherwise, simply replace the symbol A at top of the stack #1 with the production that is in T[A, t].

Building a LL(1) parsing table

Question: How can we build a LL(1) parsing table for a given CFG?

What we want: The next character should uniquely identify a production rule to use, so we should pick a production that ultimately starts with that character.

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

	int	(	)	+	*
E	int	(E Op E)			
Op				+	*

Building a LL(1) parsing table

Property: Deciding for the appropriate production to use in $T[A, t]$.

$T[A, t]$ should suggest to use a production rule $A \to \omega$ iff. ω can later be used to derive something starting with t.

Mathematically, this writes as:

$T[A, t] = \omega$ iff. $A \to \omega$ exists and $\omega \Rightarrow^* t\omega'$

Could be figured out manually (tedious).

E → `int`
E → `(` **E Op E** `)`
Op → `+`
Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	`int`	`(`E Op E`)`			
Op				+	*

Definition: FIRST Sets F

Definition (FIRST Set F for a given non-terminal symbol):

The **FIRST Set of a non-terminal symbol A**, denoted $F(A)$ in a context-free grammar is **the set of terminal symbols that can appear as the first symbol in any string derived from that non-terminal**.

In other words, the FIRST set of a non-terminal represents all possible prefix terminal symbols that can be generated by applying the production rules of the grammar, starting from that non-terminal.

Mathematically: $F(A) = \{t \mid A \Rightarrow* \ t\omega \quad \text{for some } \omega\}$

Property #1: if A is a terminal symbol, then, $F(A) = \{A\}$.

Property #2: if $A \rightarrow B\omega$ and $B \rightarrow c$, then $c \in F(A)$.

Computing a FIRST Sets F

Consider the CFG below.

S → AB

A → a | c

B → b

In our CFG above, S, A, and B are non-terminal symbols, and a, b and c are terminal symbols.

What is F(A) here?

$$F(A) = \{a, c\}$$

What is F(B) here?

$$F(B) = \{b\}$$

What is F(S) here?

According to Property #2, that would be the same as $F(A)$.

$$F(S) = \{a, c\}$$

Practice 3: computing FIRST sets for a CFG

Consider the second CFG below.

S → ACB

A → a | B | ϵ

B → b

C → c | d

Note: here ϵ denotes the empty string, meaning that A could be replaced with nothing.

In general, using ϵ in a CFG makes things much more difficult.

Question: Looking at the CFG on the left, what are $F(A)$ and $F(S)$?

Practice 3: computing FIRST sets for a CFG

Consider the second CFG below.

S → ACB

A → a | B | ϵ

B → b

C → c | d

Note: here ϵ denotes the empty string, meaning that A could be replaced with nothing.

In general, using ϵ in a CFG makes things much more difficult.

Question: Looking at the CFG on the left, what are $F(A)$ and $F(S)$?

Answer: The ϵ makes it trickier.

We have

$$F(A) = \{a, \epsilon\} \cup F(B) = \{a, b, \epsilon\}$$

The FIRST set for S is a bit awkward to build, but could we do…

$$F(S) = F(a) \cup F(B) \cup F(C) - \{\epsilon\}$$
$$= \{a, b, c, d\}?$$

But is this $=$ correct mathematically speaking? (Unproven at the moment)

Building a LL(1) table using FIRST sets.

The following algorithm constructs an LL(1) parse table for a CFG with no ϵ-productions.

- Compute the FIRST sets for all non-terminal symbols in the grammar.
- For each production $A \rightarrow t\omega$, set $T[A, t] = t\omega$.
- For each production $A \rightarrow B\omega$, set $T[A, t] = B\omega$, for each $t \in F(B)$.

When ϵ-productions are present, it might get trickier...
(As hinted and discussed in Practice 3)

Building a LL(1) table using FIRST sets.

To demonstrate how it is done, let us consider the CFG below.

S → AB

A → aX

B → bY

X → x

Y → y

Step 1: Initialize an empty table with rows for each non-terminal (S, A, B, X, and Y) and columns for each terminal (a, b, x, y) and the end-of-input marker ($).

(Shown later…)

Building a LL(1) table using FIRST sets.

To demonstrate how it is done, let us consider the CFG below.

S → AB

A → aX

B → bY

X → x

Y → y

Step 2: For each non-terminal, compute the first sets:

- $F(Y) = \{y\}$
- $F(X) = \{x\}$
- $F(A) = \{a\}$
- $F(B) = \{b\}$
- $F(S) = F(A) = \{a\}$

Building a LL(1) table using FIRST sets.

To demonstrate how it is done, let us consider the CFG below.

S → AB

A → aX

B → bY

X → x

Y → y

Step 3: Use each first set to populate the table.

	a	b	x	y
S	AB			
A	aX			
B		bY		
X			x	
Y				y

Ready to use LL(1) now!

Practice 4: LL(1) table with weird CFGs

Consider the second CFG below.

S → Aa | Ab

A → x | y

Question: Can you build the LL(1) parsing table for this CFG?

What appears in the T[S,x] cell?

Would that be acceptable for the LL(1) parser?

Practice 4: LL(1) table with weird CFGs

Consider the second CFG below.

S → Aa | Ab

A → x | y

Question: Can you build the LL(1) parsing table for this CFG?

What appears in the T[S,x] cell?

Would that be acceptable for the LL(1) parser?

Answer: In this CFG, we have the following LL(1) parsing table.

	x	y
S	Aa Ab	Aa Ab
A	x	y

The table does not tell us much!

FIRST-FIRST conflicts

In general, we do not like having a CFG that produces multiple entries in a given LL(1) parsing table, as it would mean branches in the graph search…

This is commonly referred to as a FIRST-FIRST conflict, as many possible rules could be used for a given non-terminal and terminal pair.

This unfortunately leads to the LL(1) parser having to explore multiple branches and not being able to prune as much.

	x	y
S	Aa Ab	Aa Ab
A	x y	x y

FIRST-FIRST conflicts

This might become especially true in the cases of CFGs with ϵ-productions.

When a production appears, a non-terminal symbol might disappear and let another non-terminal symbol appear in its place, as was the case in Practice 3. This is called a FIRST-FIRST conflict.

Practice 3: computing FIRST sets for a CFG

Consider the second CFG below.

S → ACB

A → a | B | ϵ

B → b

C → c | d

Note: here ϵ denotes the empty string, meaning that A could be replaced with nothing.

In general, using ϵ in a CFG makes things much more difficult.

Question: Looking at the CFG on the left, what are $F(A)$ and $F(S)$?

Answer: The ϵ makes it trickier.

We have

$$F(A) = \{a, \epsilon\} \cup F(B) = \{a, b, \epsilon\}$$

The FIRST set for S is a bit awkward to build, but could we do…

$$F(S) = F(a) \cup F(B) \cup F(C)$$
$$= \{a, b, c, d\}?$$

But is this $=$ correct mathematically speaking? (Unproven at the moment)

Main takeaway from Practice 3

The FIRST set for S is a bit awkward to build, but could we do…

$$F(S) = F(a) \cup F(B) \cup F(C)$$
$$= \{a, b, c, d\}?$$

But is this $=$ correct mathematically speaking?
(Unproven at the moment)

Need a formalism to cover for ϵ-productions!

Definition: FIRST Sets F

Definition (FIRST Set F for a given non-terminal symbol):

The **FIRST Set of a non-terminal symbol A**, denoted $F(A)$ in a context-free grammar is **the set of terminal symbols that can appear as the first symbol in any string derived from that non-terminal**.

In other words, the FIRST set of a non-terminal represents all possible first terminal symbols that can be generated by applying the production rules of the grammar, starting from that non-terminal.

Mathematically: $F(A) = \{t \mid A \Rightarrow^* t\omega \quad \text{for some } \omega\}$

Property #1: if A is a terminal symbol, then, $F(A) = \{A\}$.

Property #2: if $A \rightarrow B\omega$ and $B \rightarrow c$, then $c \in F(A)$.

Definition: FIRST* Sets F

Definition (FIRST* Set F^* for a given non-terminal symbol):

We can generalize FIRST sets to any string with FIRST* sets.

We simply have $F^*(\omega)$ being the set of all terminals (or ϵ) that can appear at the start of a string derived from ω.

Property #1: If $S \rightarrow AB\omega$, and A has a production rule that contains ε, but not B, then $F^*(S) = F(A) \cup F(B) - \{\varepsilon\}$.

Property #2: We can build $F(A)$ incrementally, by repeating the following operation, for each production $A \rightarrow \alpha$.

$$F(A) = F(A) \cup F^*(\alpha)$$

Definition: FIRST* Sets F

Going back to our previous CFG

S → ACB

A → a | B | ϵ

B → b

C → c | d

We can now correctly establish what $F(A)$ is using .

Using our F^* notation and the previous properties, we can now rigorously prove that:

F(A) = F(a) ∪ F(B) ∪ F(ε) = {a, b, ε}

F(C) = F(c) ∪ F(d) = {c, d}

F(S) = F^*(ACB)

= F(A) ∪ F(C) - {ε}

= {a, b, c, d}

Definition: FOLLOW Sets N

Definition (FOLLOW Set N for a given non-terminal symbol):

The **FOLLOW Set of a non-terminal symbol A**, denoted $N(A)$ in a context-free grammar, is the **set of terminal symbols that can appear immediately after that non-terminal symbol in any sentential form derived from the grammar's start symbol**.

In other words, the FOLLOW set of a non-terminal represents all possible terminal symbols that can be generated by applying the production rules of the grammar and appear right after that non-terminal in any derivation.

Mathematically: $N(A) = \{t \mid S \Rightarrow* \omega A t \omega' \text{ for some } \omega, \omega'\}$

Properties of the FOLLOW sets

You can compute the FOLLOW sets for a given CFGs, iteratively, as follows.

- If A is the start symbol, then the end-of-input marker ($) is in $N(A)$.
- If $B \rightarrow \alpha A \omega$ is a production rule, then you can set
$$N(A) = N(A) \cup F^*(\omega) - \{\epsilon\}$$
- If $B \rightarrow \alpha A \omega$ is a production rule and $\epsilon \in F^*(\omega)$, then you can set
$$N(A) = N(A) \cup N(B)$$

Final LL(1) parsing table algorithm

- **Step 1:** Given a CFG, compute the FIRST and FOLLOW sets, F(A) and N(A), for every non-terminal symbol A.
- **Step 2:** For each rule $A \to \omega$, and for each terminal symbol in F(ω), set $T[A, t] = \omega$. Keep in mind that, by convention, ϵ is technically not a terminal symbol!
- **Step 3:** For each rule $A \to \omega$, if $\epsilon \in F^*(\omega)$, set $T[A, t] = \omega$ for each $t \in N(A)$.

Note: Technically, and even though the notation is somewhat bad, we can describe the set procedure in 2-3 as equivalent to

For each rule $A \to \omega$, and for each terminal symbol in $F^*(\omega N(A))$, set $T[A, t] = \omega$. (Keep in mind that ϵ is not a terminal symbol).

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Notes

- When the CFG contains ϵ-production rules, you should add the string termination symbol $ in the LL(1) table and treat it as a terminal symbol.

	+	-	#	$
Num				
Sign				
Digits				
More				
Digit				

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Step 1: Compute the FIRST and FOLLOW sets for all non-terminal symbols in our CFG.
(In the interest of time, showing results only, up to you to try it again at home for practice! Solution in backup slides!)

	+	-	#	$
Num				
Sign				
Digits				
More				
Digit				

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ - 0 5 1 6 2 7 3 8 4 9	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num				
Sign				
Digits				
More				
Digit				

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ -	+ -	0 5	0 5	0 5
0 5	ε	1 6	1 6	1 6
1 6		2 7	2 7	2 7
2 7		3 8	3 8	3 8
3 8		4 9	4 9	4 9
9				ε

Next, repeat Steps 2 and 3, as mentioned earlier to fill the LL(1) parsing table!

			#	$
Num				
Sign				
Digits				
More				
Digit				

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ - 0 5 1 6 2 7 3 8 4 9	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign				
Digits				
More				
Digit				

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ -	+ -	0 5	0 5	0 5
0 5	ε	1 6	1 6	1 6
1 6		2 7	2 7	2 7
2 7		3 8	3 8	3 8
3 8		4 9	4 9	4 9
4 9				ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits				
More				
Digit				

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num		Sign		Digit		Digits		More	
+	-	+	-	0	5	0	5	0	5
0	5	ε		1	6	1	6	1	6
1	6			2	7	2	7	2	7
2	7			3	8	3	8	3	8
3	8			4	9	4	9	4	9
4	9							ε	

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits			**Digits More**	
More				
Digit				

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num		Sign		Digit		Digits		More	
+	-	+	-	0	5	0	5	0	5
0	5	ε		1	6	1	6	1	6
1	6			2	7	2	7	2	7
2	7			3	8	3	8	3	8
3	8			4	9	4	9	4	9
4	9							ε	

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits			**Digits More**	
More			**Digits**	
Digit				

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | ... | 9

Num		Sign		Digit		Digits		More	
+	-	+	-	0	5	0	5	0	5
0	5	ε		1	6	1	6	1	6
1	6			2	7	2	7	2	7
2	7			3	8	3	8	3	8
3	8			4	9	4	9	4	9
4	9							ε	

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits			**Digits More**	
More			**Digits**	
Digit			#	

Building an LL(1) table with epsilon prod.

Num	→	**Sign Digits**
Sign	→	+ | - | ε
Digits	→	**Digit More**
More	→	**Digits** | ε
Digit	→	0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ - 0 5 1 6 2 7 3 8 4 9	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**	**Sign Digits**	
Sign	+	-		
Digits			**Digits More**	
More			**Digits**	
Digit			#	

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ -	+ -	0 5	0 5	0 5
0 5	ε	1 6	1 6	1 6
1 6		2 7	2 7	2 7
2 7		3 8	3 8	3 8
3 8		4 9	4 9	4 9
4 9				ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**	**Sign Digits**	
Sign	+	-	ε	
Digits			**Digits More**	
More			**Digits**	
Digit			#	

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num		Sign		Digit		Digits		More	
+	-	+	-	0	5	0	5	0	5
0	5	ε		1	6	1	6	1	6
1	6			2	7	2	7	2	7
2	7			3	8	3	8	3	8
3	8			4	9	4	9	4	9
4	9							ε	

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**	**Sign Digits**	
Sign	+	-	ε	
Digits			**Digits More**	
More			**Digits**	
Digit			#	

Building an LL(1) table with epsilon prod.

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num		Sign		Digit		Digits		More	
+	-	+	-	0	5	0	5	0	5
0	5	ε		1	6	1	6	1	6
1	6			2	7	2	7	2	7
2	7			3	8	3	8	3	8
3	8			4	9	4	9	4	9
4	9							ε	

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**	**Sign Digits**	
Sign	+	-	ε	
Digits			**Digits More**	
More			**Digits**	ε
Digit			#	

LL(1) is cumbersome but seems to work!

Does LL(1) work well on all CFGs?

- No, we have seen in Practice 4, that this might not be the case.
- In general, any CFG with a left recursion will lead to having multiple productions rules in a given entry of the LL(1) table.
- This could lead to FIRST-FIRST conflicts. If it does, we simply say, in that case that **the CFG is not LL(1).**
- **While it might be possible sometimes to rewrite a non-LL(1) CFG into an equivalent LL(1) CFG, there is technically no clear method on how to do so 100% of the time.**
- **And most grammars cannot be represented as LL(1) anyway…!** ***(The CFG for C is not LL(1) unfortunately!)***

Predictive Parsing to the rescue?

Quick Question: How many symbols *k* should we look ahead then?

- There is a **trade-off** on the number of symbols *k* we should use.
- Increasing the number of lookahead tokens increases the number of grammars we can parse, but complicates the parser.
  A grammar might not be LL(1), but it could be LL(k), with k>1!
- A smaller number of lookahead tokens simplifies the parser, but forces grammars and syntax being expressed to be simpler as well.

LL(k) algorithms are more robust but not immune to problems either.

The method and implementation of LL(k) parsing algorithms is definitely out-of-scope for this class.

Summary and conclusion

- **Top-Down parsing:** Try to derive the user program from the start symbol of the programming language CFG.
- **Leftmost BFS and DFS:** Uncommon in practice and fail for very simple recursive CFGs.
- **Predictive LL(1) parsers:** scans from left to right, lookahead of one symbol to try and find the next leftmost derivation to use. Faster than basic BFS and DFS. Not immune to problems when CFG is not LL(1).
- **Predictive LL(k) parsers:** same as LL(1) but k symbols of lookahead. Covers more possible grammars, but computationally more expensive and still not immune to problems.

→ Difficult and somewhat impossible task so far! More importantly the CFG for the C language is not LL(k)…
Will need other approaches (to be discussed next lecture).

FIRST sets for our Epsilon CFG in LL(1) demo

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More

FIRST sets for our Epsilon CFG in LL(1) demo

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | 2 | ... | 9

Terminal symbols have themselves as FIRST sets

Num	Sign	Digit	Digits	More
	+ -	0 5 1 6 2 7 3 8 4 9		

FIRST sets for our Epsilon CFG in LL(1) demo

FIRST(Num) receives the values of FIRST(Sign), but not epsilon.

Num → **Sign Digits**
Sign → + | – | ε
Digits → Digit More
More → Digits | ε
Digit → 0 | 1 | 2 | … | 9

Num	Sign	Digit	Digits	More
+ –	+ –	0 5 1 6 2 7 3 8 4 9		

FIRST sets for our Epsilon CFG in LL(1) demo

Same with FIRST(Digits) and FIRST(Digit)

Num → Sign Digits
Sign → + | - | ε
Digits → Digit More
More → Digits | ε
Digit → 0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
+ -	+ -	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	

FIRST sets for our Epsilon CFG in LL(1) demo

Same with FIRST(More) and FIRST(Digits)

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
+ -	+ -	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9

FIRST sets for our Epsilon CFG in LL(1) demo

Epsilon appears in FIRST(Sign) and FIRST(More)

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | 2 | … | 9

Num	Sign	Digit	Digits	More
+ -	+ -	0 5	0 5	0 5
	ε	1 6	1 6	1 6
		2 7	2 7	2 7
		3 8	3 8	3 8
		4 9	4 9	4 9
				ε

FIRST sets for our Epsilon CFG in LL(1) demo

Since Epsilon appears in FIRST(Sign), FIRST(Num) should receive the terminals in FIRST(Digits)

Num → **Sign Digits**
Sign → + | – | ε
Digits → Digit More
More → Digits | ε
Digit → 0 | 1 | 2 | … | 9

Num	Sign	Digit	Digits	More
+ –	+ –	0 5	0 5	0 5
0 5	ε	1 6	1 6	1 6
1 6		2 7	2 7	2 7
2 7		3 8	3 8	3 8
3 8		4 9	4 9	4 9
4 9				ε