COMP 520 - Compilers

Lecture 03 – Compiler Theory and Formal Analysis

Annoucements

PA1 autograder is live on Gradescope

Entry code: PWBK78

PA1 submission instructions posted on Piazza

More Announcements

The autograder is built upon Gradescope’s recommendations.

The autograder does not accept filenames and folders that contain spaces (or tabs or other whitespace) in them.

(Your file can contain whitespace, but not the filename)

Please be cautious when uploading your source files to the autograder (name appropriately!)

Good Questions!

- Lexer is responsible for building the language’s Lexicon
- Some languages require a more robust Lexer
  - Recall C++ example from Lec02

- So what tokens are generated from the following?

`6234432whileclass`

When to use the Graph method?

- Primarily if you need to ***prove*** that your `TokenKind`/`TokenType` represents the CFG.

- Reducing the number of `TokenType`s makes it easier on the Parser even if the types must be manually differentiated later when analyzing context/generating code. (And when that happens, it is usually still easier to have a condensed set of `TokenType`s).

- But making things easier requires you to ***prove*** your method still adheres to the targeted language.

What to expect

- Lecture01- Intro to the course, grading structure, expectations.
- Lecture02- Massive content drop, prepares you for PA1 as early as possible to give you time to complete the assignment.

What to expect

- Lecture01- Intro to the course, grading structure, expectations.
- Lecture02- Massive content drop, prepares you for PA1 as early as possible to give you time to complete the assignment.

- From here on out, we can breathe a sigh of relief and slow down.
- We described what needs to be done in code but haven't formally described how parsing is done.
- So now we can shift back towards the science part of compilers!

Lecture03

Let’s get formal and describe Compilers in a way that cannot be misrepresented!

Context-Free Grammar

A review, and new Compiler-specific definitions

CFGs for Compilers

- The CFG, G, consists of:
  - N: Set of nonterminal symbols (elements start with an uppercase)
  - T: Set of terminal symbols (elements start lowercased)
  - A start symbol where $\mathrm{start} \in N$
  - A set of rewrite rules of the form A ::= α where
    - $\mathrm{A} \in N$
    - α is a sequence of $\mathrm{T} \cup \mathrm{N}$ or ε (empty sequence)

Formal Definition Review

- The CFG, G, consists of:
  - N: Set of nonterminal symbols (elements start with an uppercase)
  - T: Set of terminal symbols (elements start lowercased)
  - A start symbol where $\text{start} \in N$
  - A set of rewrite rules of the form A ::= α where
    - $\mathrm{A} \in N$
    - α is a sequence of $T \cup N$ or ε (empty sequence)
- (α sequence is a T∪N of fun to parse!)
- PAUSE!

Sentence Definition

- A sentence w consists exclusively of terminal symbols
- Consider a start symbol S where $S = \alpha_1$

- We will say that $\alpha_i \Rightarrow \alpha_{i+1}$ (α_i yields α_{i+1}) if…
  - When W ::= ω is a rule in G
  - When β, γ, ω (beta, gamma, omega) are sequences

$$(a_i \Rightarrow \alpha_{i+1}) \rightarrow (\alpha_i = \beta \mathrm{W} \gamma) \wedge (\alpha_{i+1} = \beta \omega \gamma)$$

Sentence Definition

- A sentence w consists exclusively of terminal symbols
- Consider a start symbol S where $S = \alpha_1$

- We will say that $\alpha_i \Rightarrow \alpha_{i+1}$ (α_i yields α_{i+1}) if…
  - When W ::= ω is a rule in G
  - When β, γ, ω (beta, gamma, omega) are sequences

$$(a_i \Rightarrow \alpha_{i+1}) \rightarrow (\alpha_i = \beta \mathrm{W} \gamma) \wedge (\alpha_{i+1} = \beta \omega \gamma)$$

- The sentence w is generated when
  - $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ where $\alpha_n = w$

Context-Free Language

- L(G) is the set of ALL sentences generated by G

$$L(G) = \{w | w \in T^* \text{ and } S \Rightarrow w\}$$

Context-Free Language

- L(G) is the set of ALL sentences generated by G

$$L(G) = \{w | w \in T^* \text{ and } S \Rightarrow w\}$$

- Example: What sentences are generated by this CFG?

S ::= A $

A ::= (A)

A ::= x

(Terminals are in orange)

Leftmost Derivation

Leftmost Derivation

- Leftmost derivation can generate any sentence in L(G)

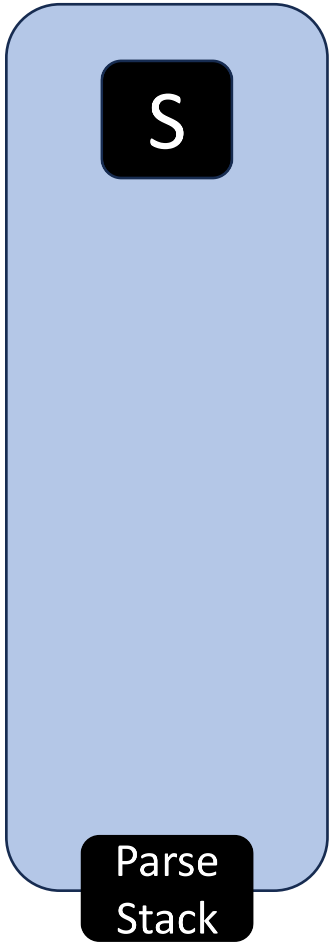
- Only modify our sentence generation rules slightly.
- We will say that $\alpha_i \Rightarrow \alpha_{i+1}$ if…
  - When W ::= ω is a rule in G
  - When β, γ, ω (beta, gamma, omega) are sequences
  - **Additional Rule**: β consists of zero or more **terminal** symbols

$$(a_i \Rightarrow \alpha_{i+1}) \rightarrow (\alpha_i = \beta \mathrm{W} \gamma) \land (\alpha_{i+1} = \beta \omega \gamma)$$

Let’s simulate top-down parsing using a pushdown automaton and leftmost derivation!

Top-down Parsing

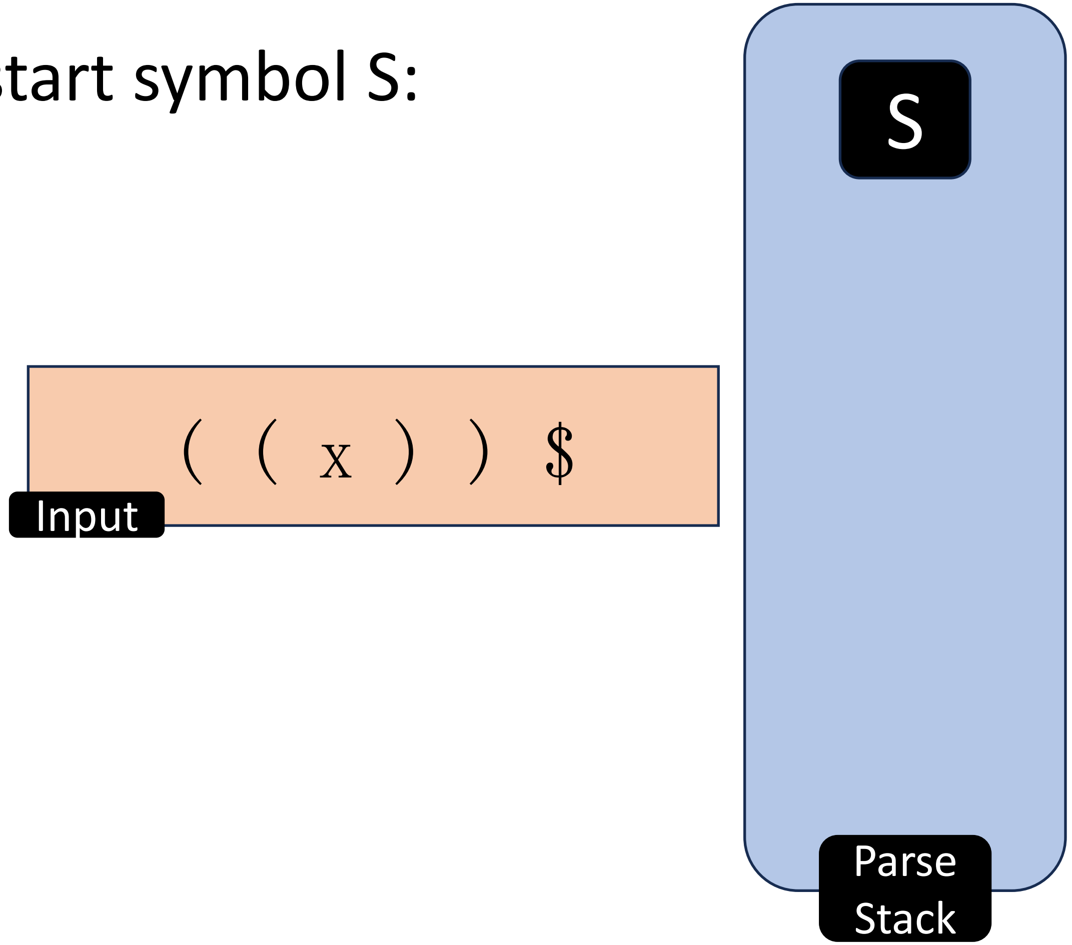
- A top-down parser simulates leftmost derivation!
- Create a parse stack that contains the start symbol S:



Top-down Parsing

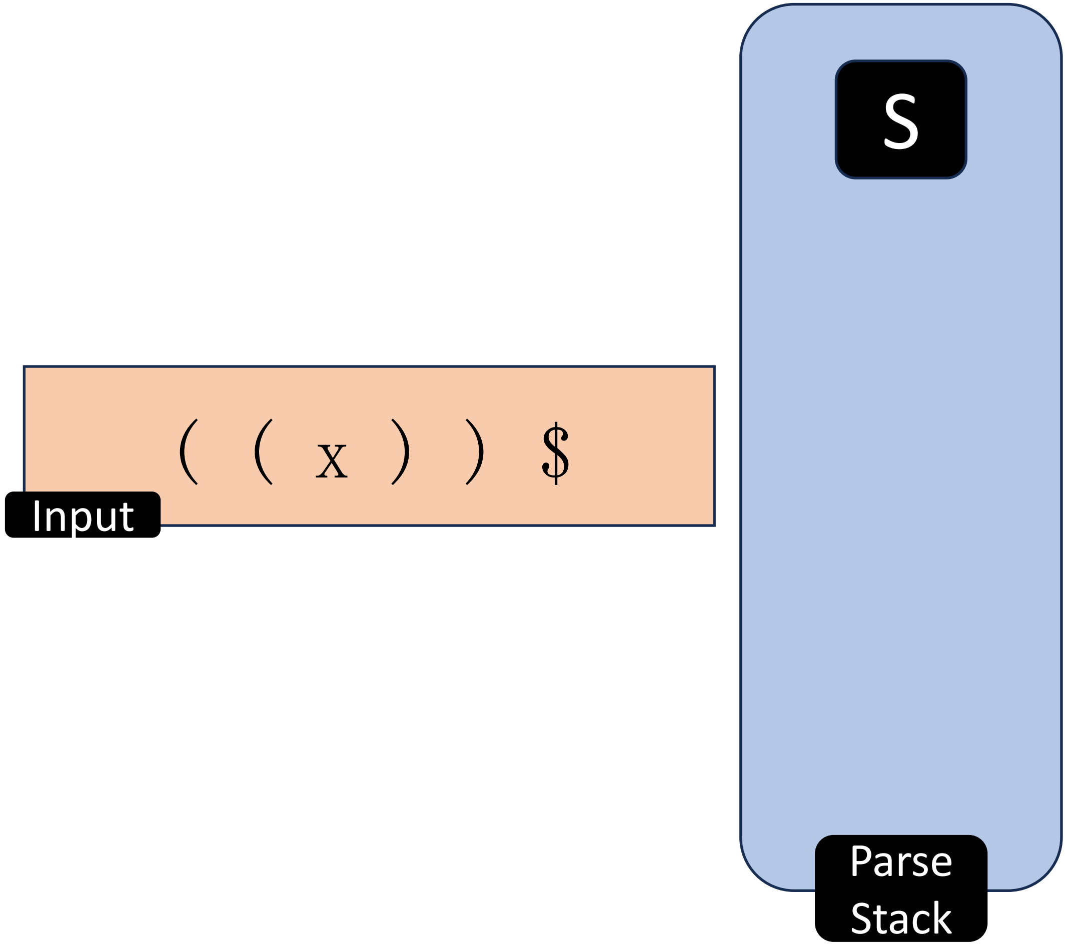
Create a parse stack that contains the start symbol S:

1) Read the input, w, left-to-right



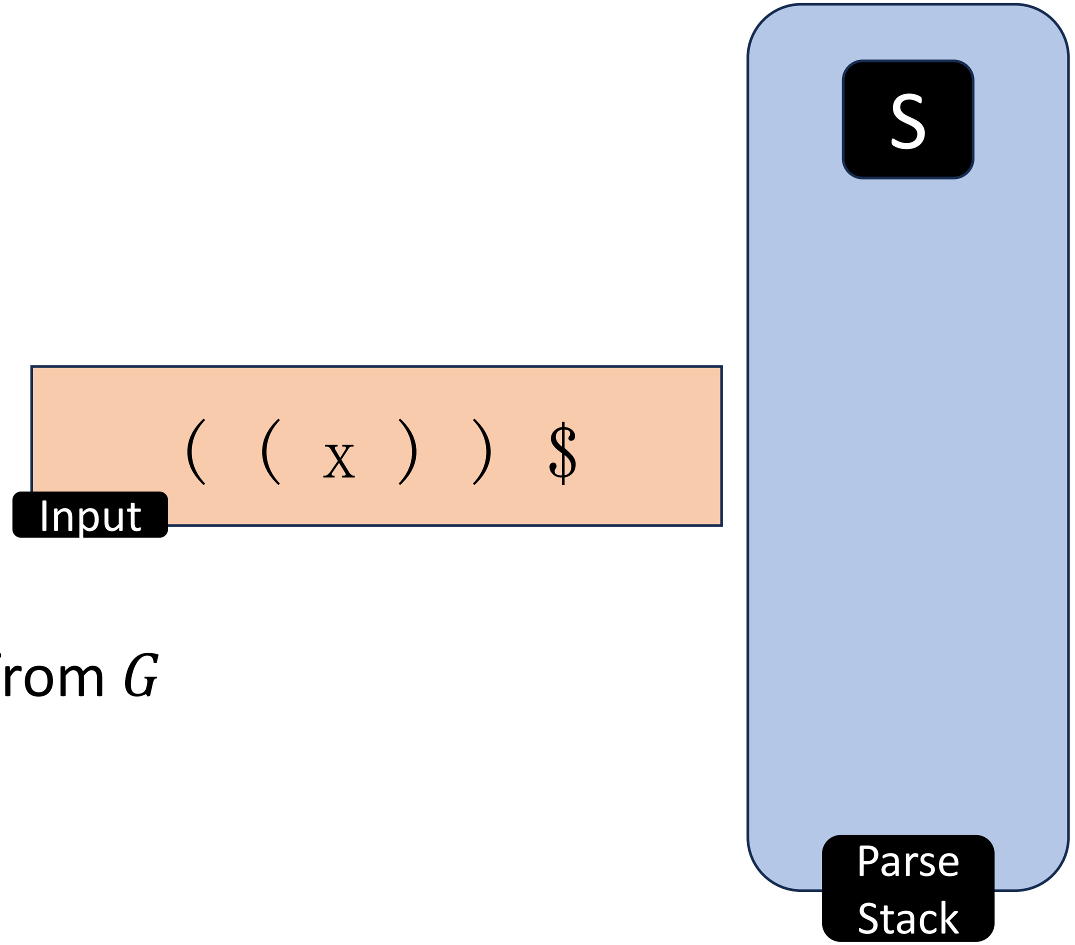
Top-down Parsing

1) Read the input, w, left-to-right
2) If the top of the parse stack is a terminal b, pop b from the stack



Top-down Parsing

1) Read the input, w, left-to-right
2) If the top of the parse stack is a terminal b, pop b from the stack

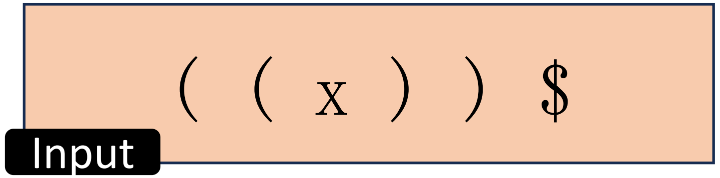


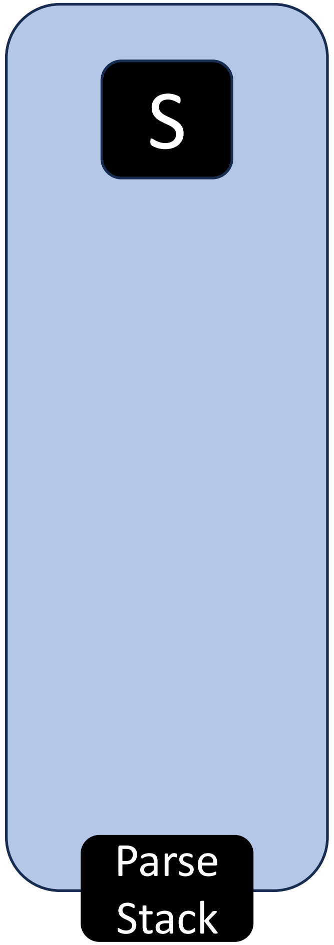
3) If the top is a non-terminal..
   - Need to predict the correct rule A ::= α from G
   - Pop A and push α

Top-down Parsing

- Create a parse stack that contains the start symbol S:

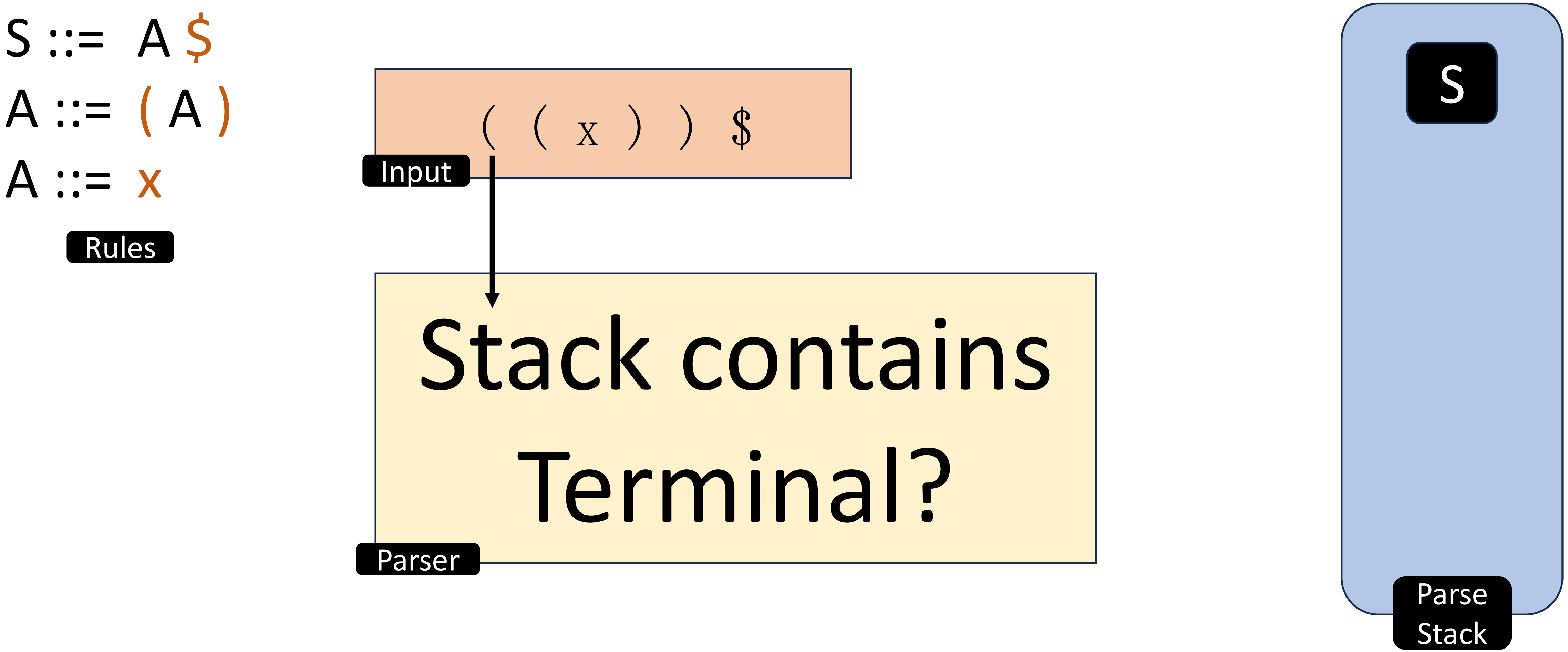
1) Read the input, w, left-to-right
2) If the top of the parse stack is a terminal b, pop b from the stack
3) If the top is a non-terminal..
   - Need to predict the correct rule A ::= α from G
   - Pop A and push α
4) Repeat until parse stack is empty or the input is exhausted





What does this look like in practice?

Top-down Parsing



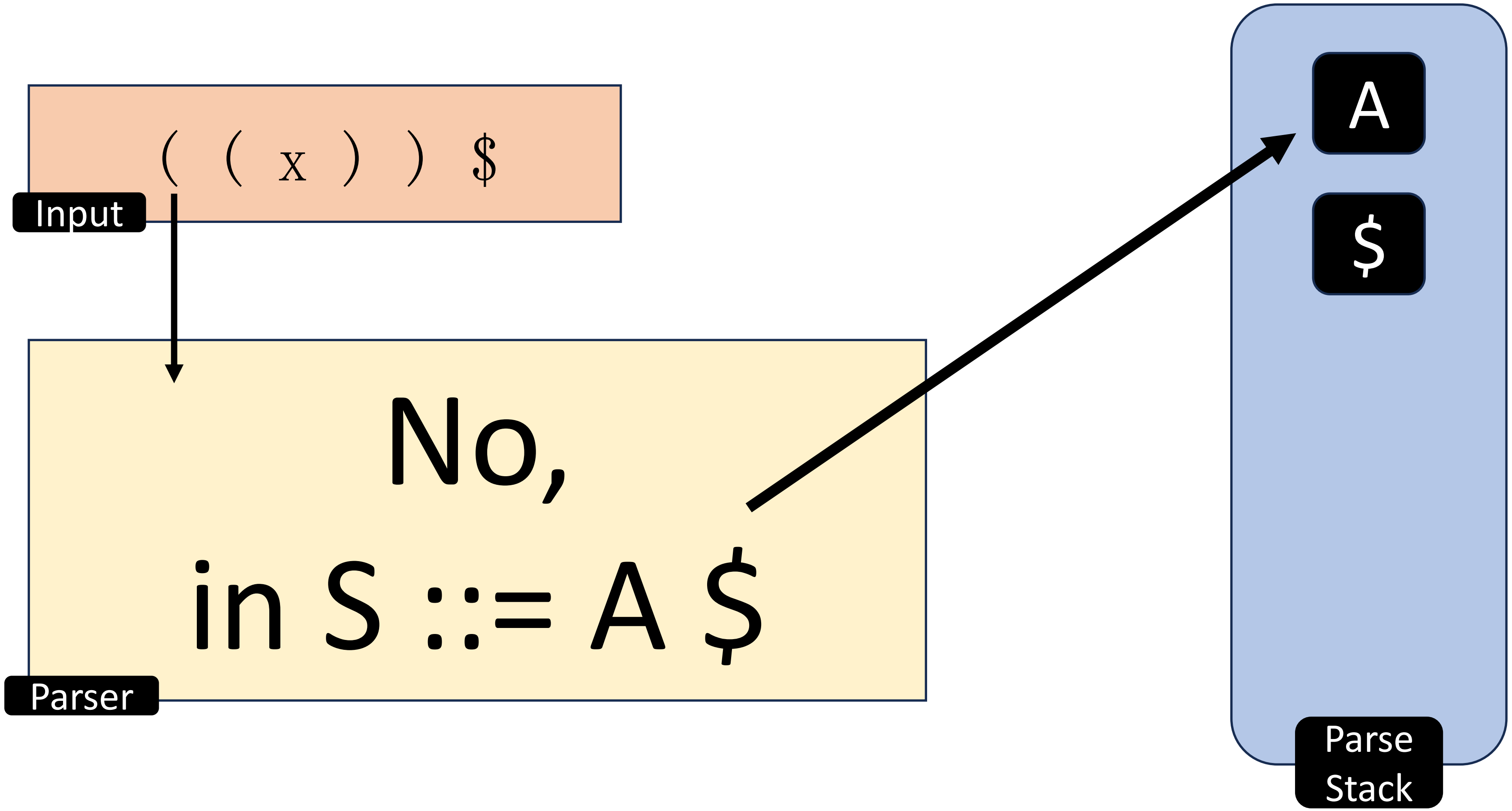
Top-down Parsing

S ::= A $

A ::= (A)

A ::= x

Rules



Top-down Parsing

S ::= A $

A ::= (A)

A ::= x

Rules

((x)) $

Input

No,

in A ::= (A)

Parser

(

A

)

$

Parse Stack

Top-down Parsing

S ::= A $

A ::= (A)

A ::= x

Rules

((x)) $

Input

Yes,

Read in a (

Parser

(

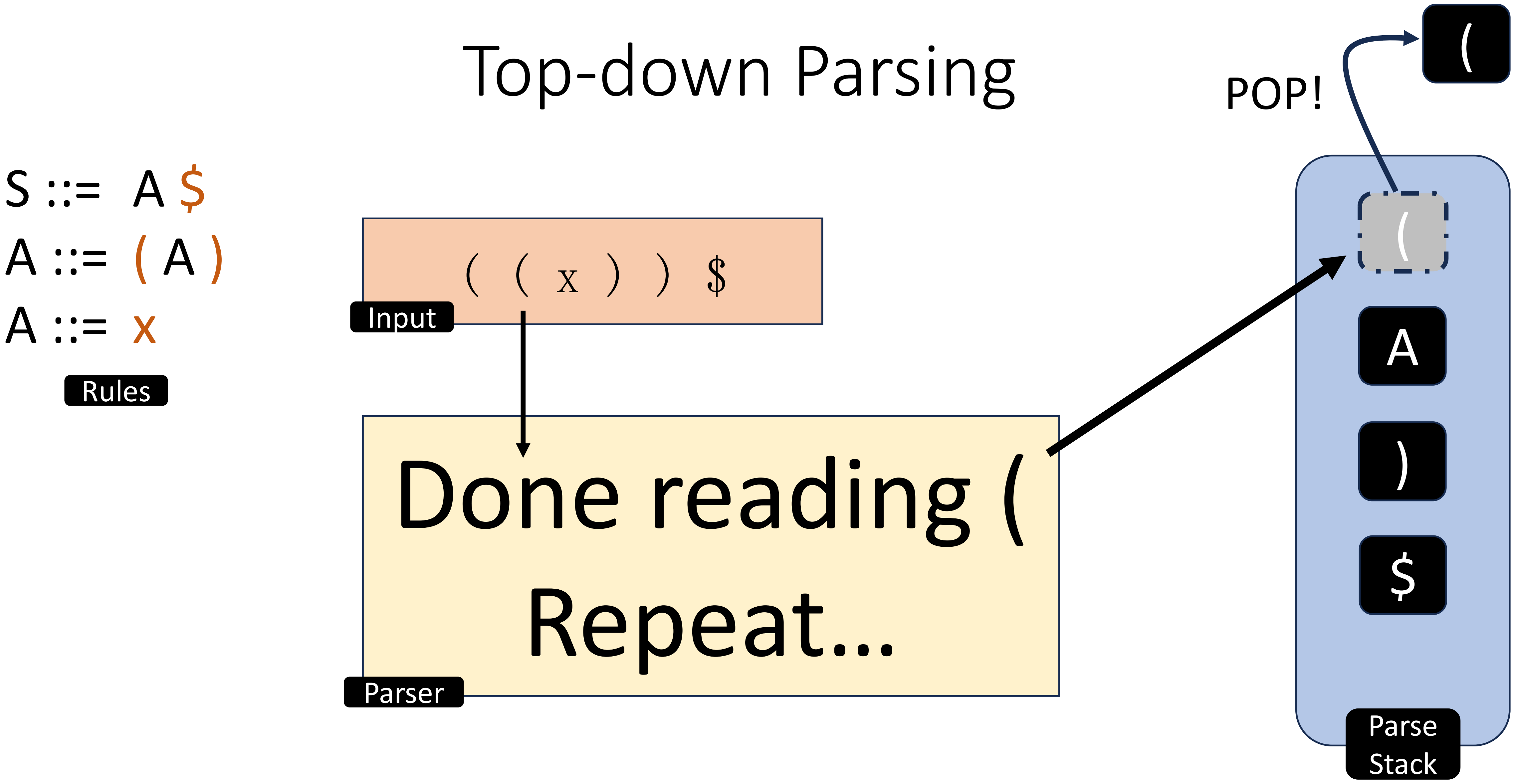
A

)

$

Parse Stack

Top-down Parsing



Final Rule

Input $w \in L(G)$ if

the stack is ______

(and/or)

the input is _______

Final Rule

Input $w \in L(G)$ if

the stack is EMPTY

AND

the input is EXHAUSTED

Full Example when w=(x)$ and $\mathrm{w} \in L(G)$

Input seen	Stack	Input left	Action
	S	(x) $	predict S ::= A$
	A$	(x) $	see "(", predict A ::= (A)
	(A) $	(x) $	match terminal
(	A) $	x) $	see "x", predict A ::= x
(	x) $	x) $	match terminal
(x	) $	) $	match terminal
(x)	$	$	match terminal
(x) $			stack empty, no input left – sentence recognized

Leftmost derivation

Key ideas and Starter Sets

- Resolve choices in grammar rules by looking at the next symbol(s)
  - A ::= (A)
  - A ::= x
- Two choices for A. Which terminals appear at the **start** of each choice?
  - **Starters** of (A) = { (}
  - **Starters** of x = { x }

Key ideas and Starter Sets

- Resolve choices in grammar rules by looking at the next symbol(s)
  - A ::= (A)
  - A ::= x
- Two choices for A. Which terminals appear at the start of each choice?
  - **Starters** of (A) = { (}
  - **Starters** of x = { x }
- **Formally**: when the **starters** are **disjoint**, we can always resolve the choice by looking at the next input symbol

The LL(1) condition

Your new best friend!

LL(1) Condition

- Guarantees that the parser can **ALWAYS** predict the correct rule based on…
  - The next **(1)** symbol
  - When reading **L**eft to right
  - Using **L**eftmost derviation

LL(1) Condition

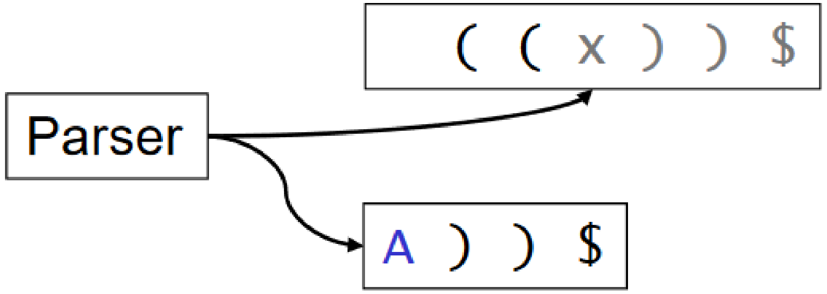
- Guarantees that the parser can **ALWAYS** predict the correct rule based on…
  - The next **(1)** symbol
  - When reading **L**eft to right
  - Using **L**eftmost derviation

- If your CFG is LL(1), you will make your compiler developer quite happy.
- *Question*: Why?

Recursive Descent Parsing

Recursive Descent Parsing

- Implementation uses a lot of recursion!
- Each non-terminal gets a parseN() method where N is a non-terminal.



CFG G_0

S ::= A \$

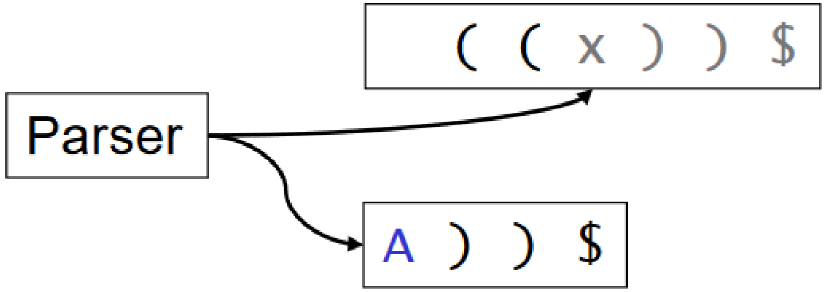
A ::= (A)

A ::= x

```
parseS() {
    parseA();
    accept( ‘$’ );
}

parseA() {
    if ( currChar == ‘(‘ ) {
         accept( ‘(’ );
         parseA();
         accept( ‘)’ );
    }
    else
         accept( ‘x’ );
}
```

Recursive Descent Parsing



CFG G_0

S ::= A $

A ::= (A)

A ::= x

```
parseS() {
   parseA();
   accept( '$' );
}

parseA() {
   if ( currChar == '(' ) {
       accept( '(' );
       parseA();
       accept( ')' );
   }
   else
       accept( 'x' );
}
```

- *Question*: Where is the parse **stack** *automatically* maintained in recursive descent parsing?

Recursive Descent Parsing

- Once again, $w \in L(G)$ if
  - Parse stack is empty AND w is exhausted

- Also explains why exception handling is useful here, it helps unwind the stack to ensure your program can recover from syntax errors.

Recursive Descent Parsing

- PA1 starter code lightly enforces this style (easy to change)
- Note: You do not have to do recursive descent (i.e., you can do the PDA example from earlier)

- We recommend recursive descent parsing
  - (A little easier to work with on your first compiler)

- *True or False*: any recursive algorithm can be rewritten as a non-recursive algorithm? If so, what would that be for recursive descent?

Enjoy your weekend!

- Make sure to start on PA1
- If you weren’t doing anything fun, something **super exciting** you can work on:

  Think about whether Java, miniJava, or other languages are LL(1)

- Apologies if I took over your weekend plans with the exciting prospect above, parse responsibly!
- Next week: Grammar transformations, ENBF, and cool properties we can exploit.

End