COSC252: Programming Languages:

Syntactic Analysis

Jeremy Bolton, PhD
Asst Teaching Professor

GEORGETOWN UNIVERSITY

Outline

Notes: Read

I. Syntactic Analysis
   I. Context Free languages
      I. BNF Productions
      II. Parse Trees
      III. Derivations

II. Parsers …

GEORGETOWN UNIVERSITY

The Basics of Programming Languages

- Before we can learn about each of these steps, we will formalize the concepts and nomenclature

Recall: Generators to Characterize a Language

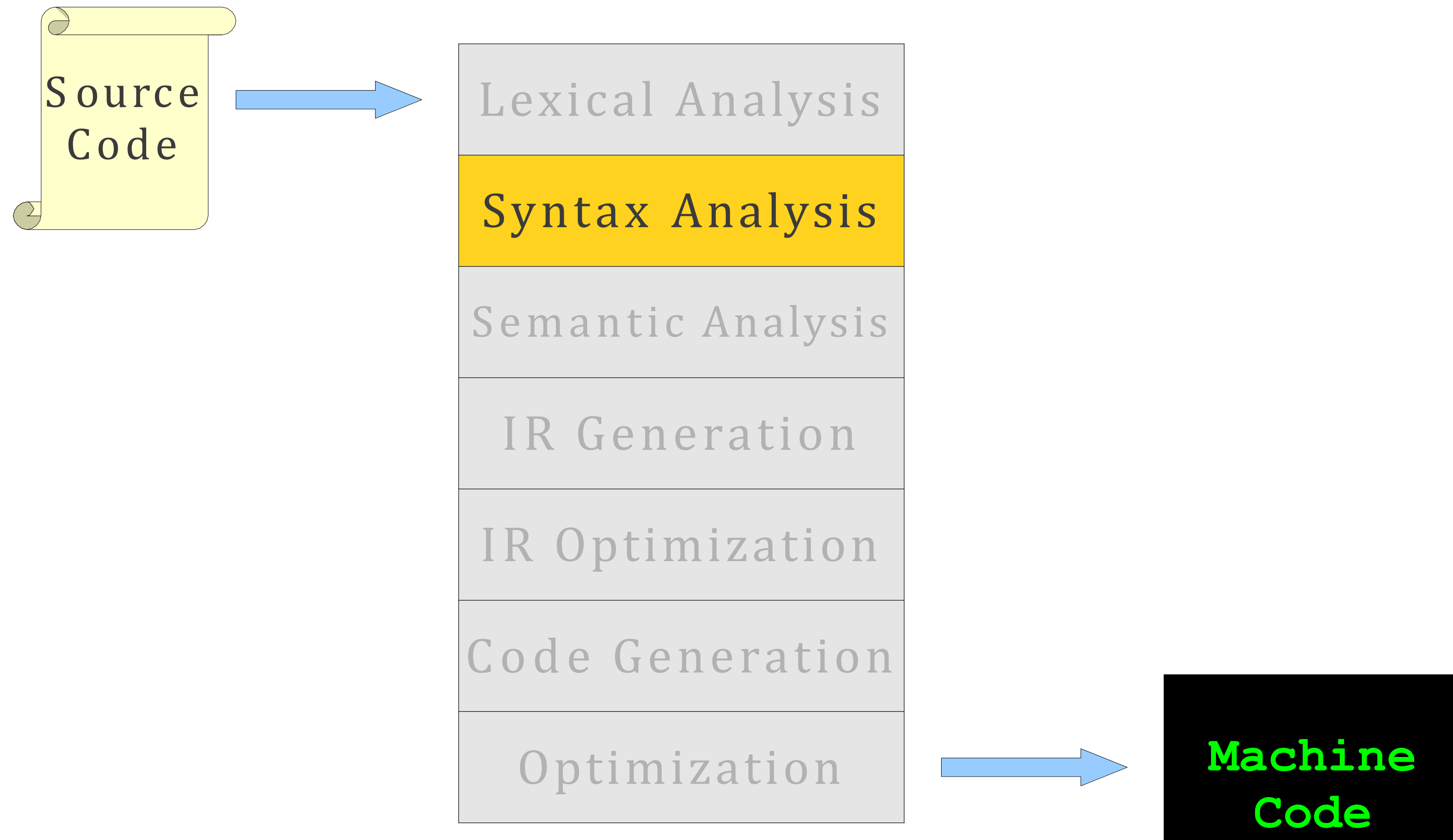
- Noam Chomsky
  - MIT Linguist
  - Published work on categories of Languages / Grammars, 1950s
    - Two categories are commonly used in Programming Languages
      - Context-Free Languages
        - » Often used to characterize programming language sentence structure
      - Regular Languages
        - » Often used to characterize the structure of lexemes / tokens

- John Backus and Peter Naur developed a formal notation for generating a Context Free Language, (similar to the notation used by Chomsky)
  - BNF (Bachus Naur Form)

Recall: Grammars

- Generators are often implemented as a set of rules, called grammars.
- BNF grammars are grammars consisting of
  - terminals: lexemes (integral syntactic units)
  - non-terminals: abstract compositions of terminals
    - Have at least two possible forms
  - productions: composition rules for non-terminals.

Where We Are



GEORGETOWN UNIVERSITY

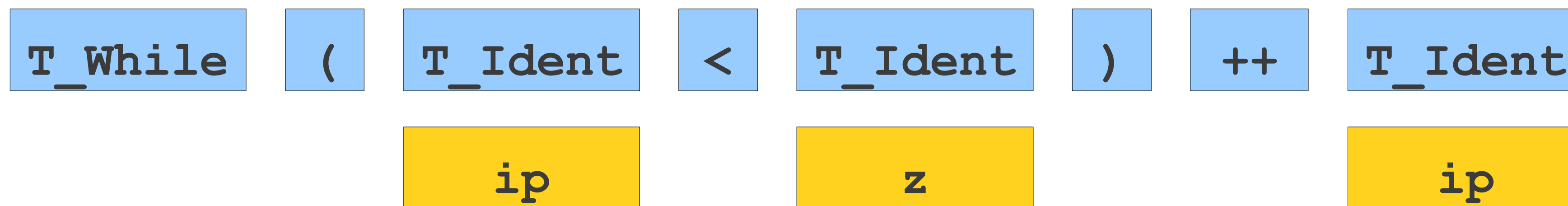
```
while (ip < z)
    ++ip;
```

GEORGETOWN UNIVERSITY

w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

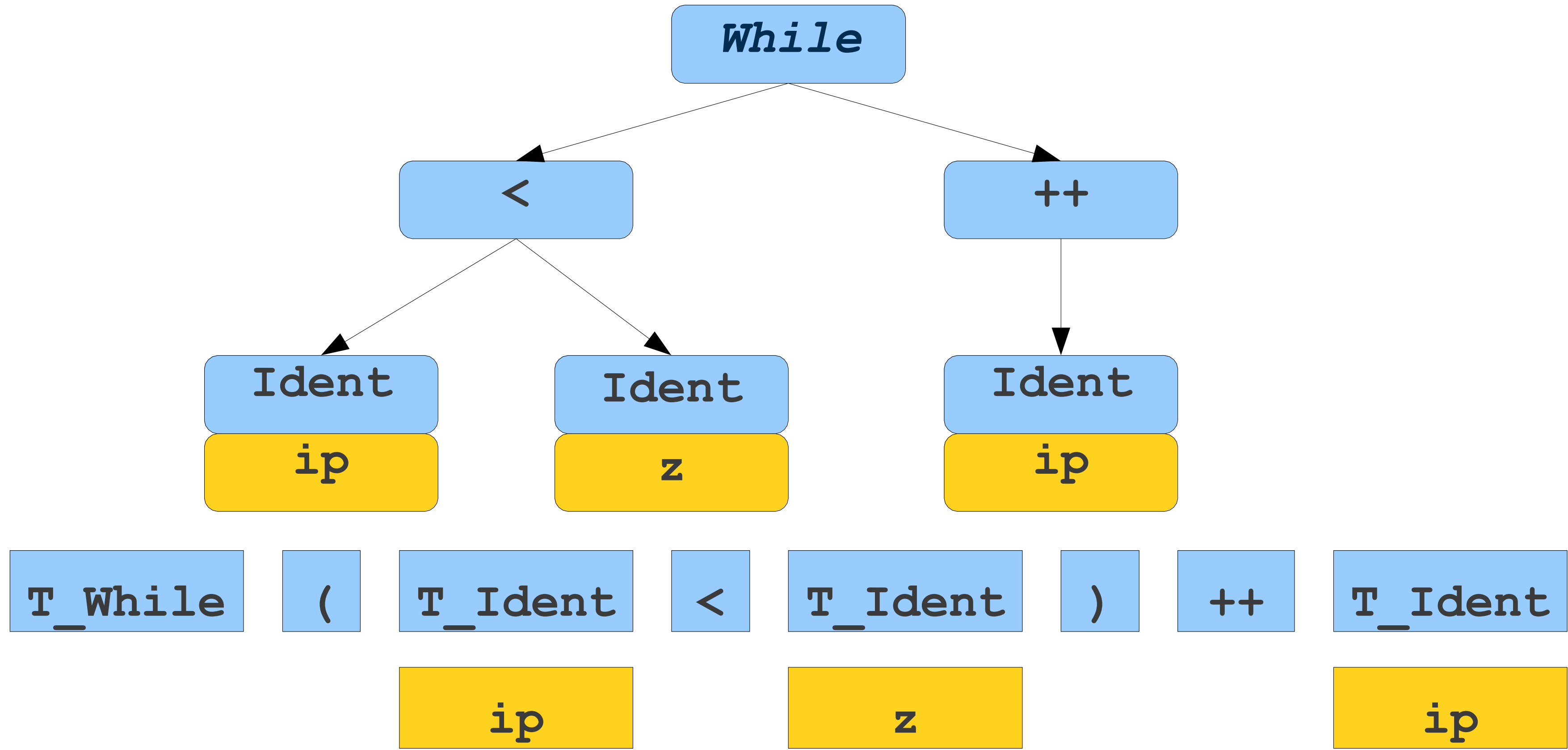
```
while (ip < z)
    ++ip;
```

GEORGETOWN UNIVERSITY



w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```



w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
++ip;
```

GEORGETOWN UNIVERSITY

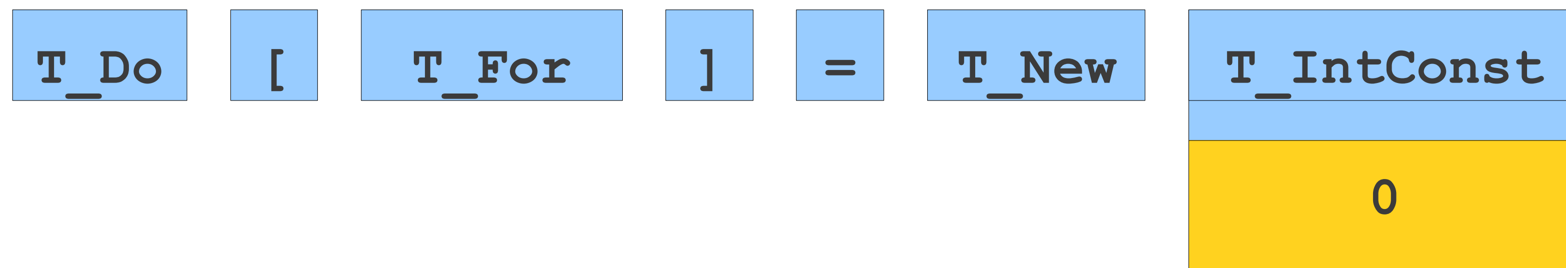
```
do[for] = new 0;
```

GEORGETOWN UNIVERSITY

d	o	[	f	o	r	]		=		n	e	w		0	;

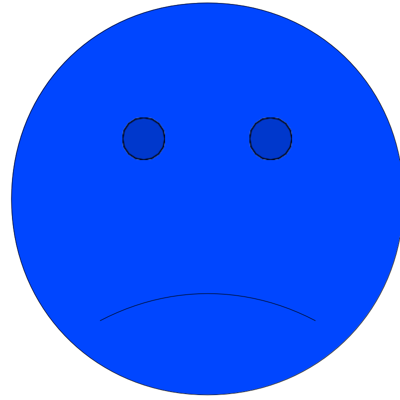
```
do[for] = new 0;
```

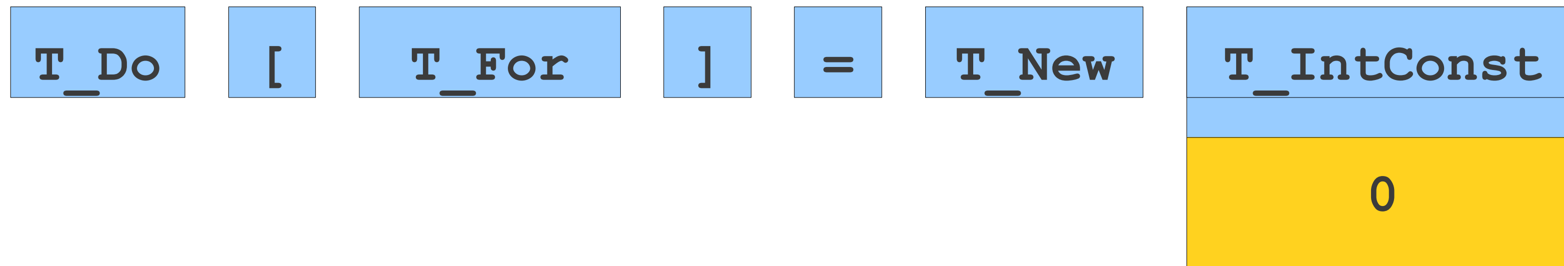
GEORGETOWN UNIVERSITY



do[for] = new 0;

GEORGETOWN UNIVERSITY





d o [f o r]   =   n e w   0 ;

```
do[for] = new 0;
```

GEORGETOWN UNIVERSITY

What is Syntax Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- In syntax analysis (or parsing), we want to interpret what those tokens mean.
- Goal: Recover the *structure* described by that series of tokens.
- Goal: Report *errors* if those tokens do not properly encode a structure.

GEORGETOWN UNIVERSITY

Formal Languages

- An alphabet is a set Σ of symbols that act as letters.
- A language over Σ is a set of strings made from symbols in Σ.
- When scanning, our alphabet was ASCII or Unicode characters. We produced tokens.
- When parsing, our alphabet is the set of tokens produced by the scanner.

GEORGETOWN UNIVERSITY

The Limits of Regular Languages

- When scanning, we used regular expressions to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.

  Cannot define a regular expression matching all expressions with properly balanced parentheses.

  Cannot define a regular expression matching all functions with properly nested block structure.

We need a more powerful formalism.

Context-Free Grammars

- A context-free grammar (or CFG) is a formalism for defining languages.
- Can define the context-free languages, a strict superset of the the regular languages.
- CFGs are best explained by example...

Try This: Define a BNF Production that assures that ALL parens are matched for an "expression".

GEORGETOWN UNIVERSITY

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

E → int
E → E Op E
E → (E)
Op → +
Op → -
Op → *
Op → /

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

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

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

```
  E
⇒ E Op E
⇒ E Op int
⇒ int Op int
⇒ int / int
```

GEORGETOWN UNIVERSITY

Context-Free Grammars

- Formally, a context-free grammar is a collection of four objects:
  - A set of nonterminal symbols (or variables),
  - A set of terminal symbols,
  - A set of production rules saying how each nonterminal can be converted by a string of terminals and nonterminals, and
  - A start symbol that begins the derivation.

E → int
E → E Op E
E → (E)
Op → +
Op → -
Op → *
Op → /

GEORGETOWN UNIVERSITY

A Notational Shorthand

E → int

E → E Op E

E → (E)

Op → +

Op → -

Op → *

Op → /

GEORGETOWN UNIVERSITY

A Notational Shorthand

E → **int** | E Op E | **(**E**)**

Op → **+** | **-** | ***** | **/**

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

S → **a*b**

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow A\mathbf{b}$$

GEORGETOWN UNIVERSITY

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow Ab$$
$$A \rightarrow Aa \mid \varepsilon$$

GEORGETOWN UNIVERSITY

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

S → a(b|c*)

GEORGETOWN UNIVERSITY

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

S → aX

X → (b|c*)

GEORGETOWN UNIVERSITY

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

S → aX
X → b | c*

GEORGETOWN UNIVERSITY

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow \mathbf{a}X$$
$$X \rightarrow \mathbf{b} \mid C$$

GEORGETOWN UNIVERSITY

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow \mathbf{a}X$$
$$X \rightarrow \mathbf{b} \mid C$$
$$C \rightarrow C\mathbf{c} \mid \varepsilon$$

CFGs for Programming Languages

```
BLOCK  →  STMT
       |  { STMTS }

STMTS  →  ε
       |  STMT STMTS

STMT   →  EXPR;
       |  if (EXPR) BLOCK
       |  while (EXPR) BLOCK
       |  do BLOCK while(EXPR);
       |  BLOCK
       |  …

EXPR   →  identifier
       |  constant
       |  EXPR + EXPR
       |  EXPR – EXPR
       |  EXPR * 
```

GEORGETOWN UNIVERSITY

Some CFG Notation

- We will be discussing generic transformations and operations on CFGs over the next two weeks.
- Let's standardize our notation.

GEORGETOWN UNIVERSITY

Some CFG Notation

- Capital letters at the beginning of the alphabet will represent nonterminals.
  - i.e. A, B, C, D
- Lowercase letters at the end of the alphabet will represent terminals.
  - i.e. `t`, `u`, `v`, `w`
- Lowercase Greek letters will represent arbitrary strings of terminals and nonterminals.
  - i.e. α, γ, ω

GEORGETOWN UNIVERSITY

Examples

- We might write an arbitrary production as

$$A \rightarrow \omega$$

- We might write a string of a nonterminal followed by a terminal as

$$A\mathtt{t}$$

- We might write an arbitrary production containing a nonterminal followed by a terminal as

$$B \rightarrow \alpha A \mathtt{t} \omega$$

GEORGETOWN UNIVERSITY

Derivations

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

- This sequence of steps is called a derivation.
- A string $\alpha A \omega$ yields string $\alpha \gamma \omega$ iff $A \to \gamma$ is a production.
- If α yields β, we write $\alpha \Rightarrow \beta$.
- We say that α derives β iff there is a sequence of strings where

$$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \beta$$

- If α derives β, we write $\alpha \Rightarrow^* \beta$.

GEORGETOWN UNIVERSITY

Leftmost Derivations

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT  → EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …

EXPR  → identifier
      | constant
      | EXPR + EXPR
      | EXPR – EXPR
      | EXPR * EXPR
      | EXPR = …
```

```
  STMTS
⇒ STMT STMTS
⇒ EXPR; STMTS
⇒ EXPR = EXPR; STMTS
⇒ id = EXPR; STMTS
⇒ id = EXPR + EXPR; STMTS
⇒ id = id + EXPR; STMTS
⇒ id = id + constant; STMTS
⇒ id = id + constant;
```

GEORGETOWN UNIVERSITY

Recall: Leftmost Derivations

- A leftmost derivation is a derivation in which each step expands the leftmost nonterminal.
- A rightmost derivation is a derivation in which each step expands the rightmost nonterminal.
- These will be of great importance when we talk about **parsing** !!!

GEORGETOWN UNIVERSITY

Related Derivations

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

- E
- ⇒ E Op E
- ⇒ int Op E
- ⇒ int * E
- ⇒ int * (E)
- ⇒ int * (E Op E)
- ⇒ int * (int Op E)
- ⇒ int * (int + E)
- ⇒ int * (int + int)

Derivations Revisited

- A derivation encodes two pieces of information:
  - What productions were applied produce the resulting string from the start symbol?
  - In what order were they applied?
- Multiple derivations might use the same productions, but apply them in a different order.

GEORGETOWN UNIVERSITY

Parse Trees: In this example we build using LMD

GEORGETOWN UNIVERSITY

Parse Trees

E

E

Parse Trees

E

E

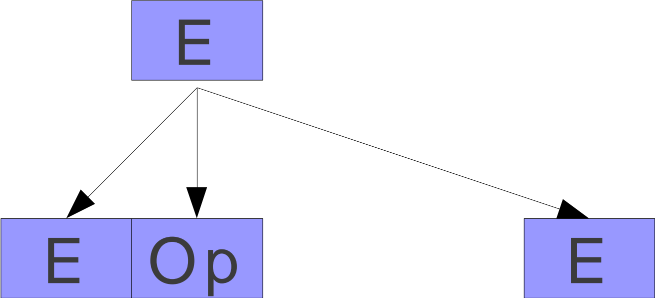
⇒ E Op E

GEORGETOWN UNIVERSITY

Parse Trees

E

$\Rightarrow$ E Op E

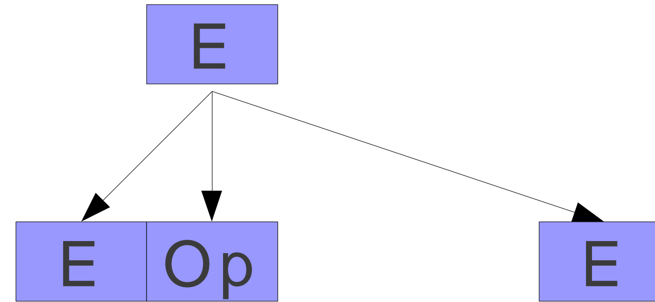


Parse Trees

E

⇒ E Op E

⇒ `int` Op E



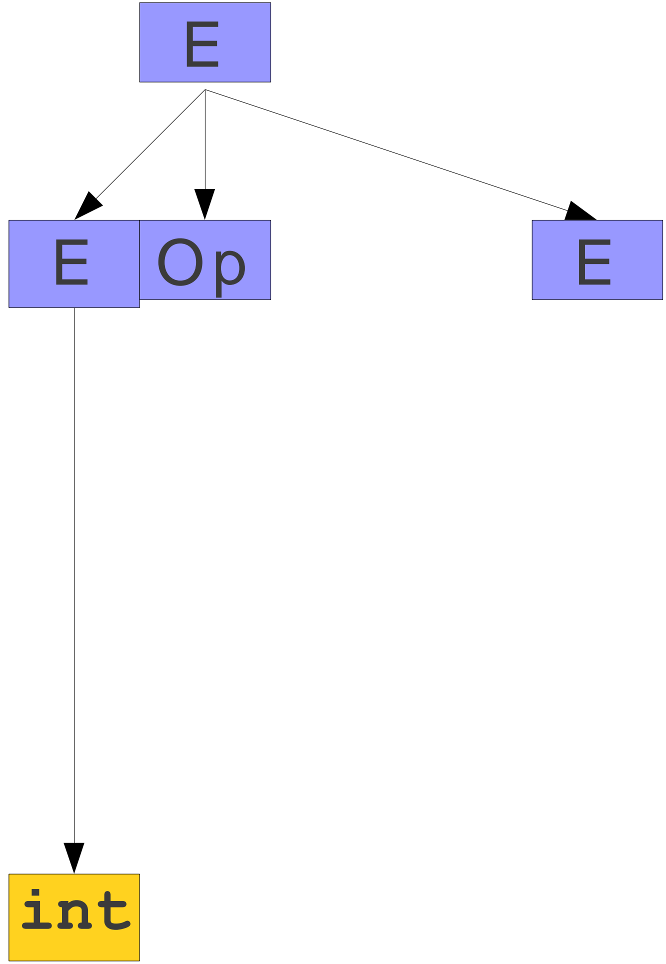
GEORGETOWN UNIVERSITY

Parse Trees

E

⇒ E Op E

⇒ `int` Op E



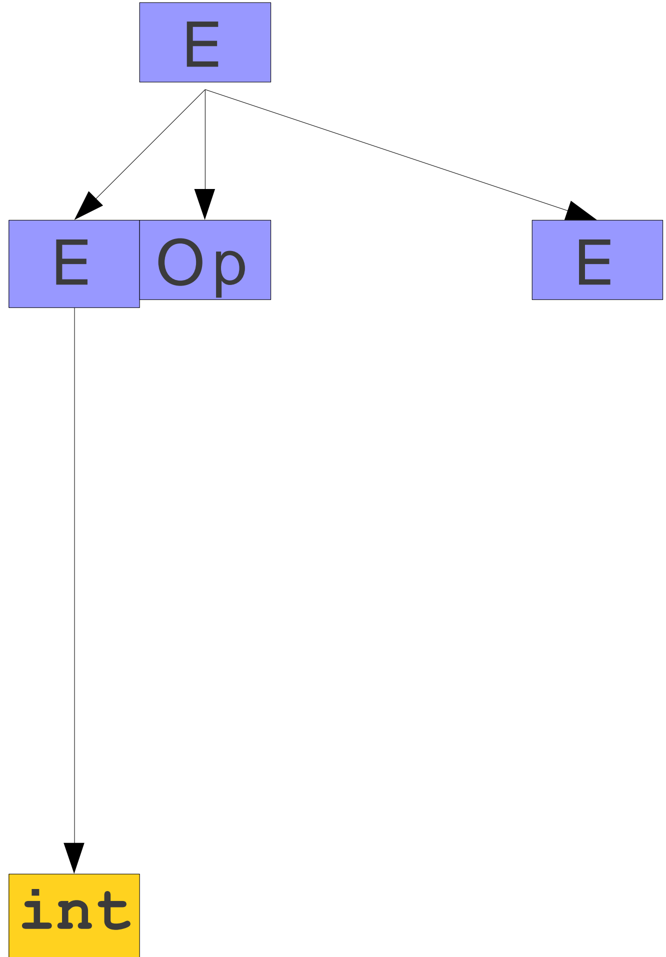
Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E



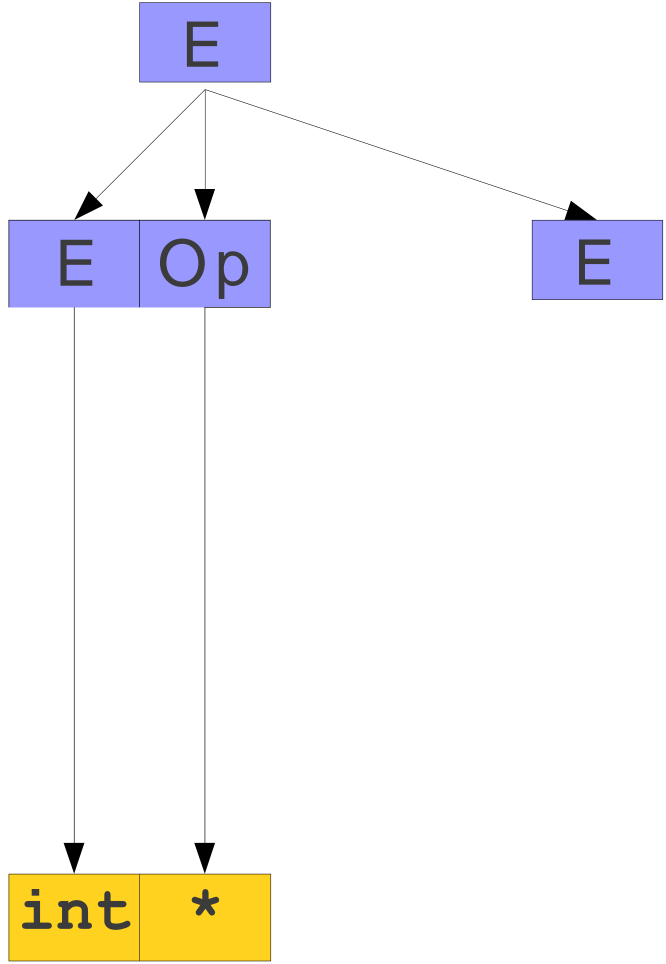
Parse Trees

E

⇒ E Op E

⇒ **int** Op E

⇒ **int *** E



GEORGETOWN UNIVERSITY

Parse Trees

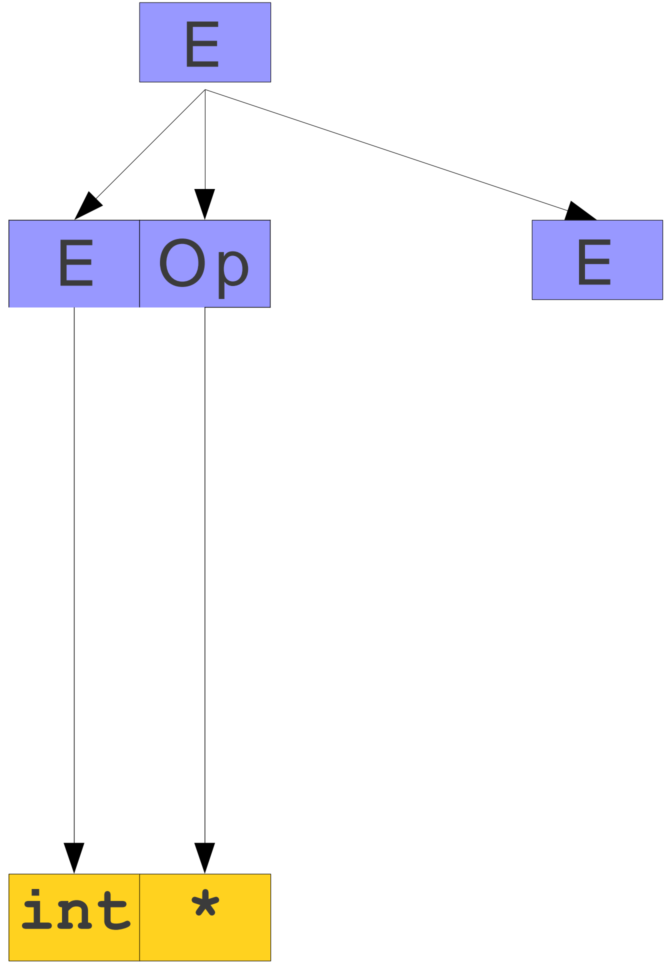
E

⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)



GEORGETOWN UNIVERSITY

Parse Trees

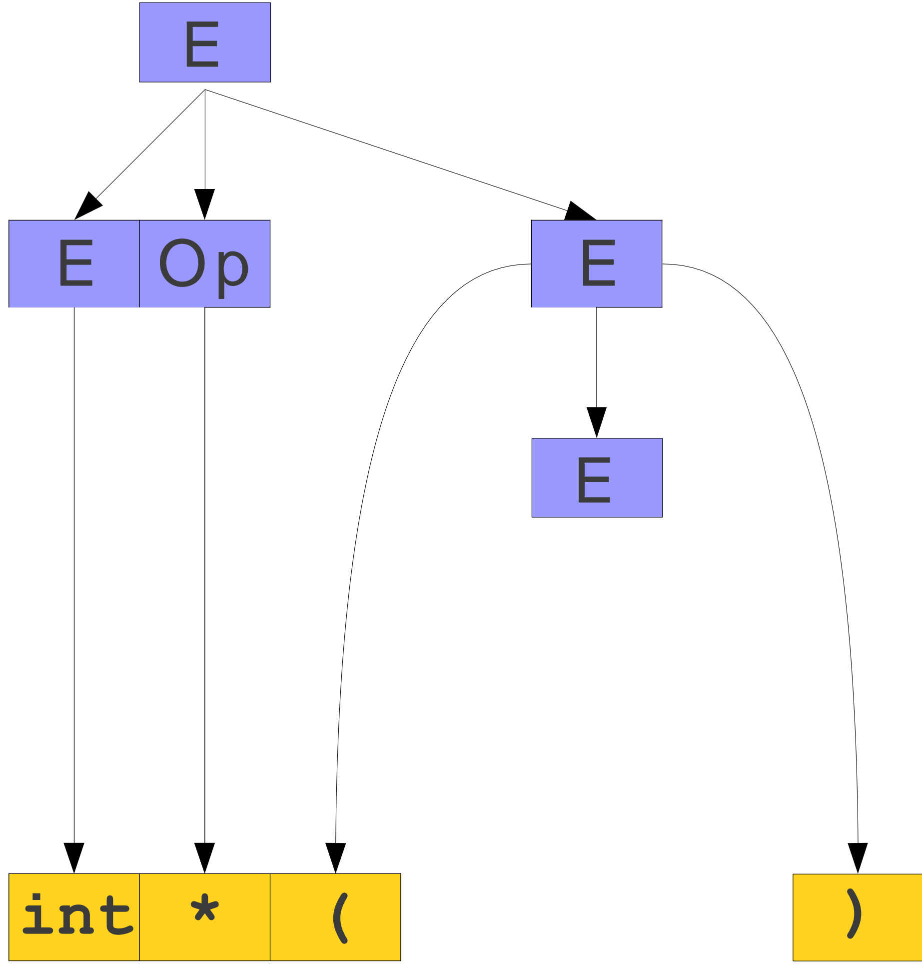
E

⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)



Parse Trees

E

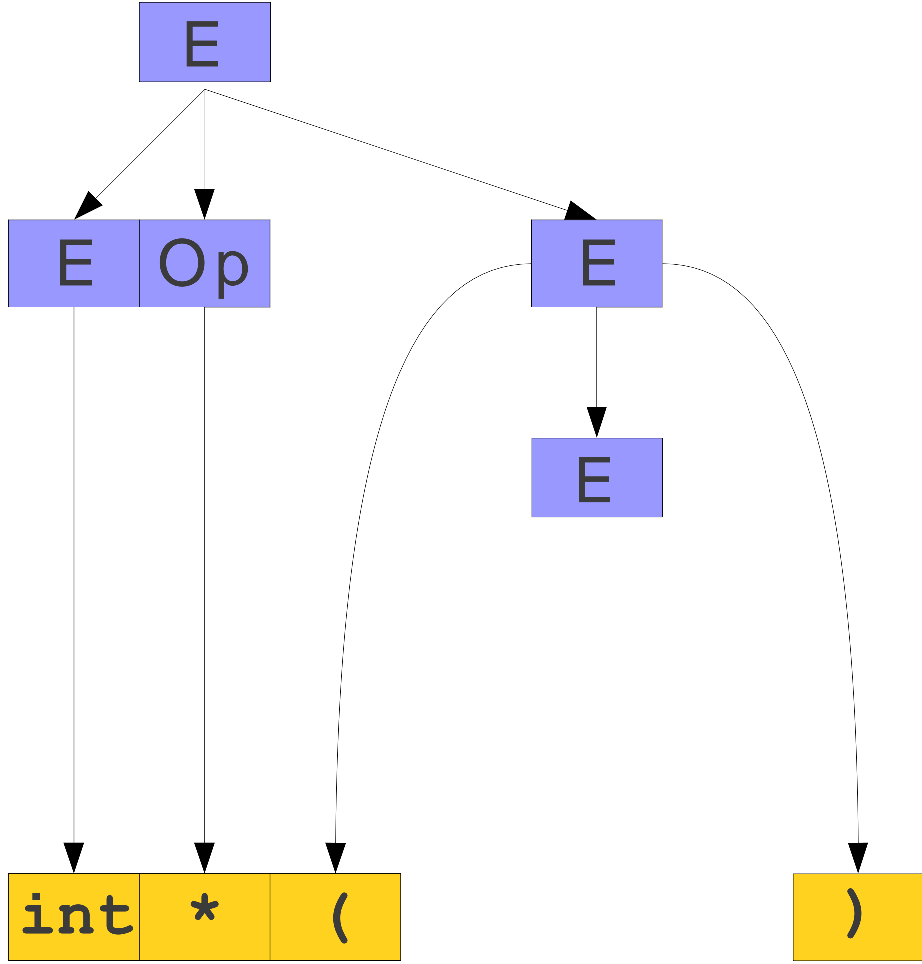
⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)



GEORGETOWN UNIVERSITY

Parse Trees

E

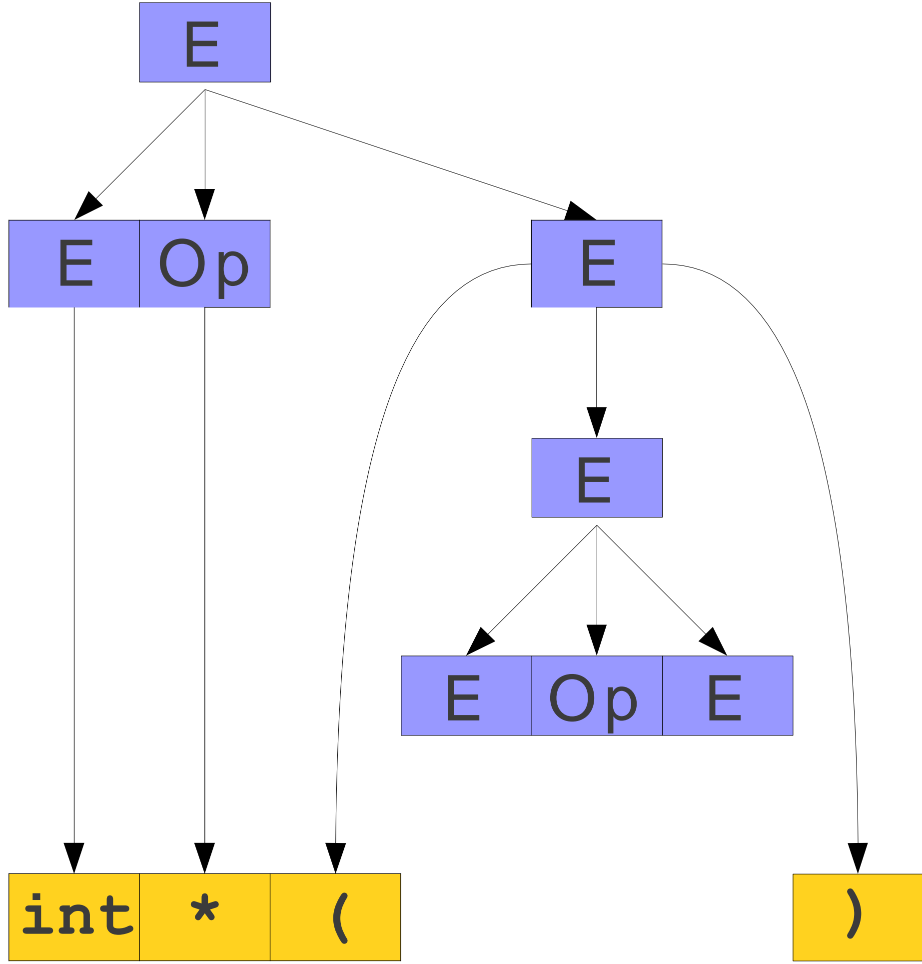
⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)



GEORGETOWN UNIVERSITY

Parse Trees

E

⇒ E Op E

⇒ `int` Op E

⇒ `int * ` E

⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`

⇒ `int * (int` Op E`)`

GEORGETOWN UNIVERSITY

Parse Trees

E

⇒ E Op E

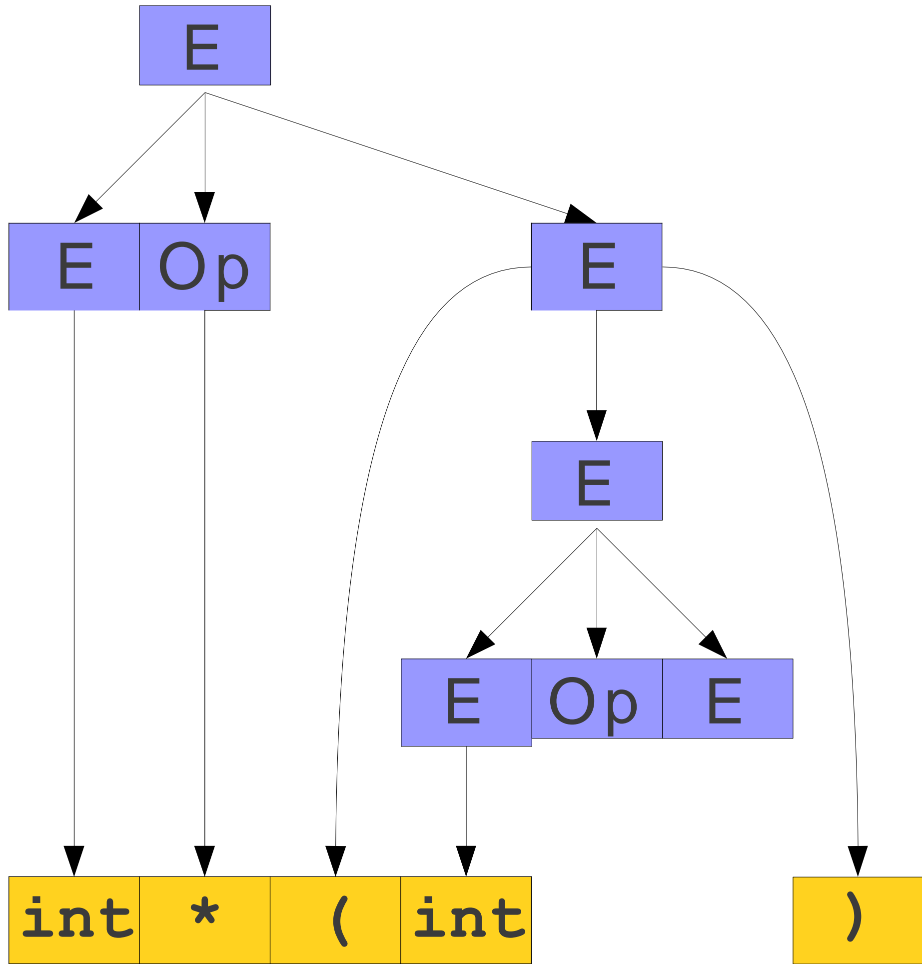
⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)



Parse Trees

E

⇒ E Op E

⇒ int Op E

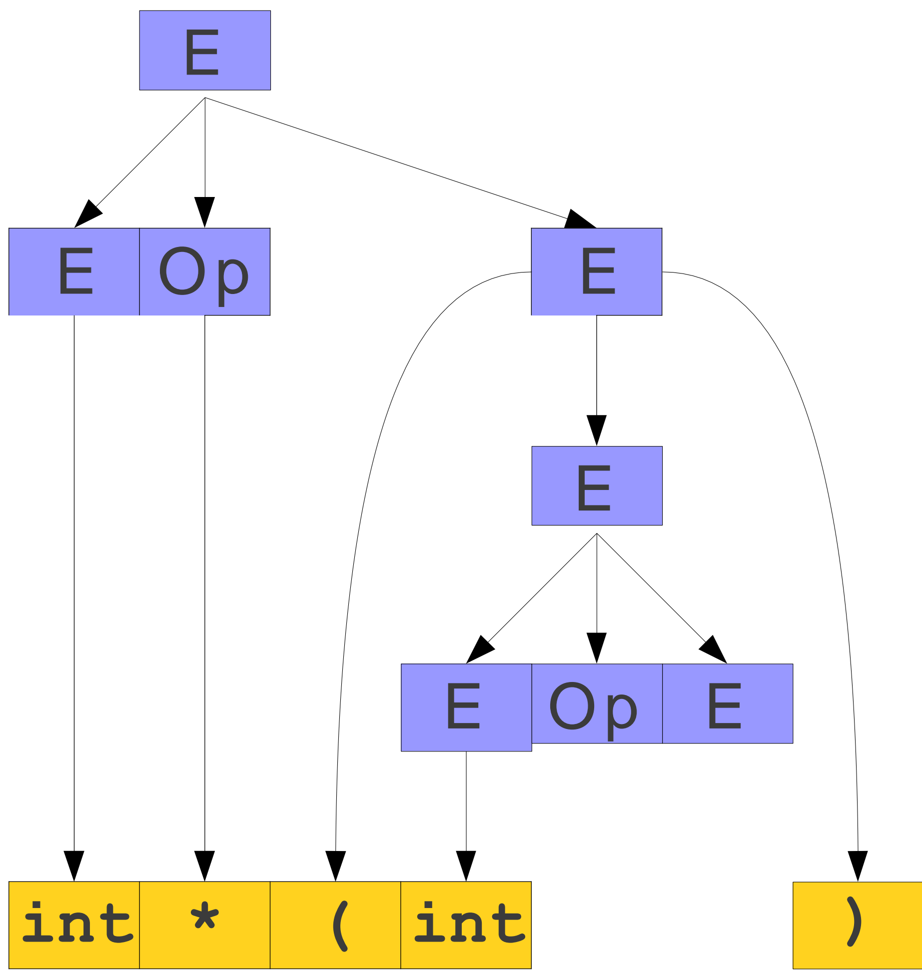
⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)



GEORGETOWN UNIVERSITY

Parse Trees

E

⇒ E Op E

⇒ int Op E

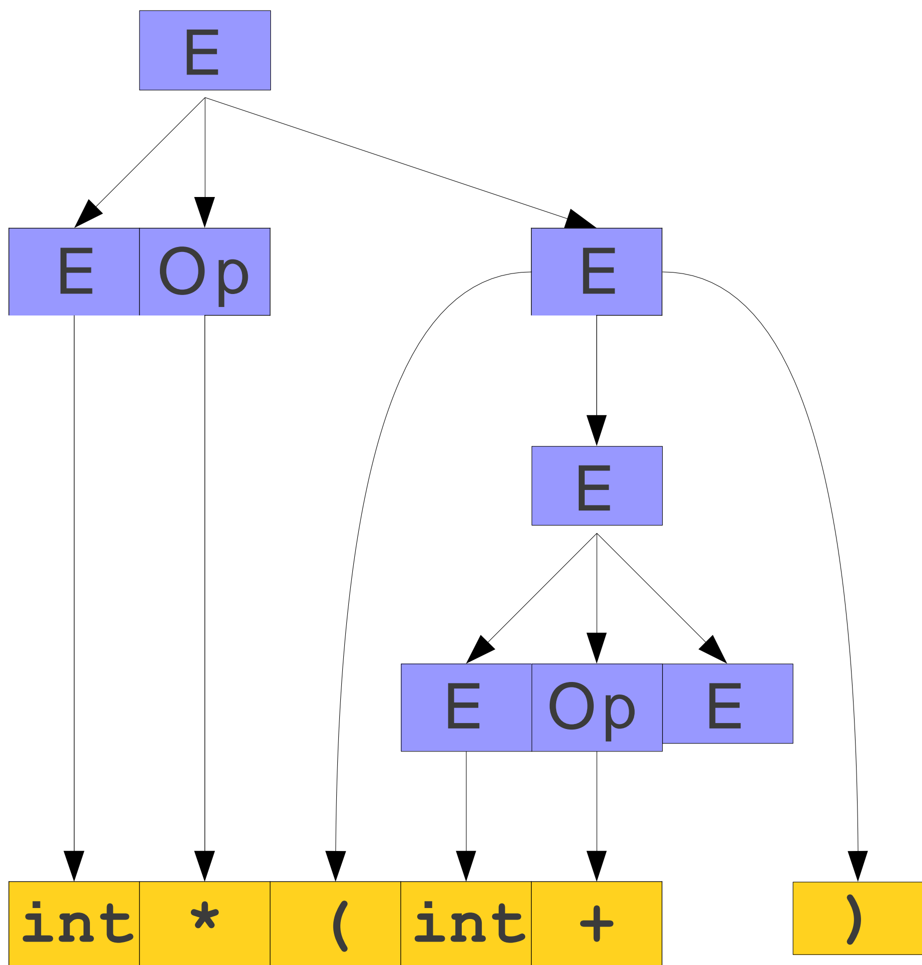
⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)



GEORGETOWN UNIVERSITY

Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E

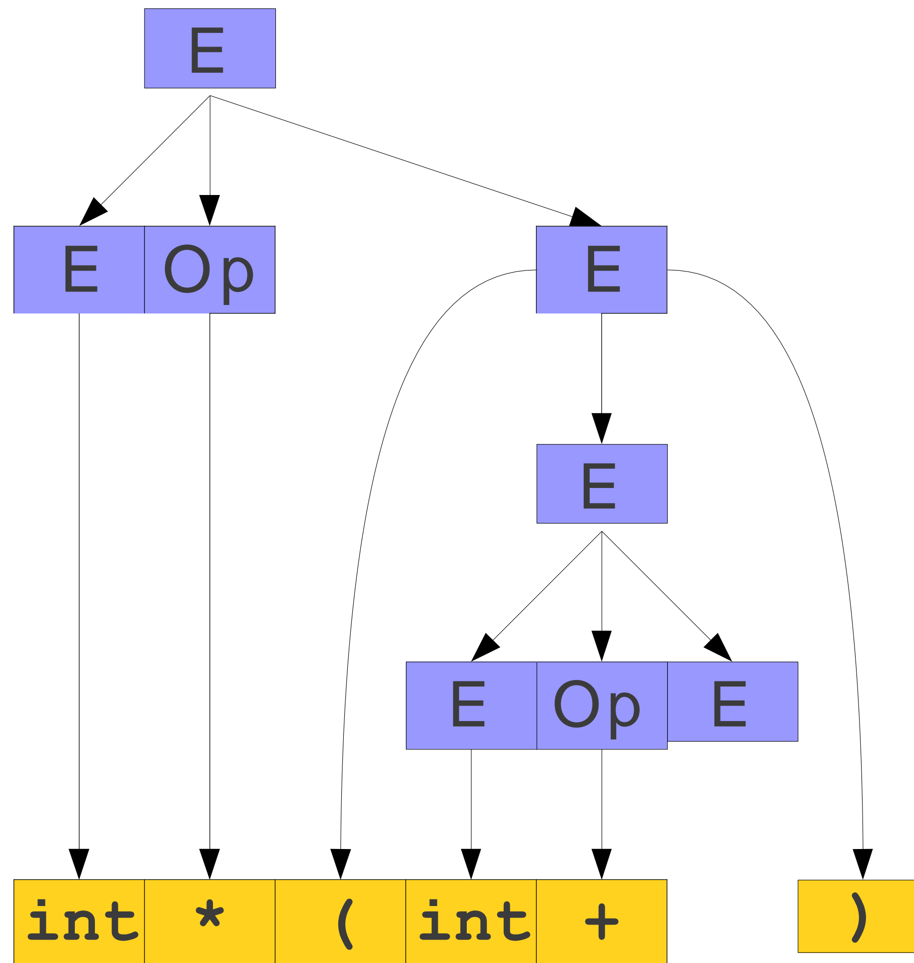
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



GEORGETOWN UNIVERSITY

Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E

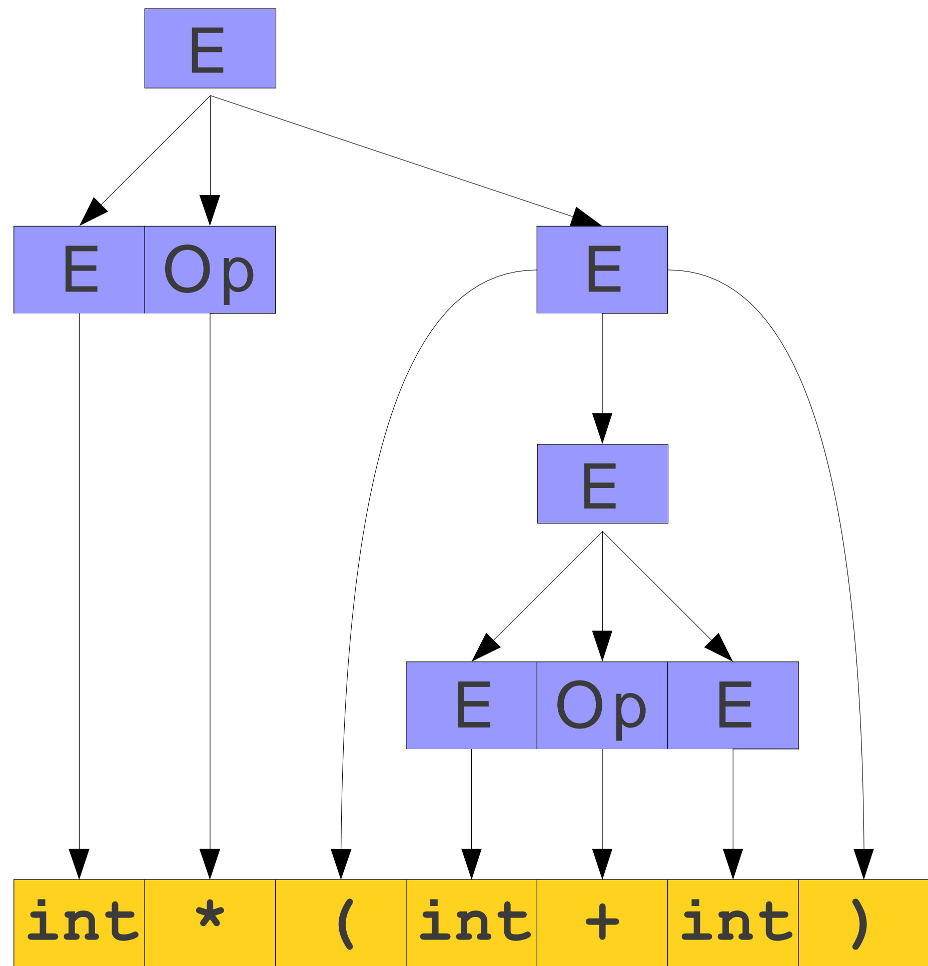
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



GEORGETOWN UNIVERSITY

Parse Trees: In this example we build using RMD

GEORGETOWN UNIVERSITY

Parse Trees

E

E

Parse Trees

E

E

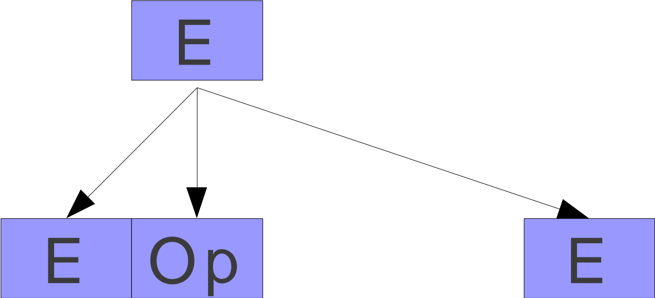
⇒ E Op E

GEORGETOWN UNIVERSITY

Parse Trees

E

$\Rightarrow$ E Op E

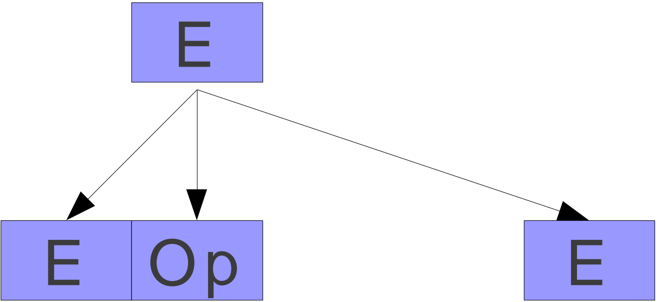


Parse Trees

E

⇒ E Op E

⇒ E Op (E)



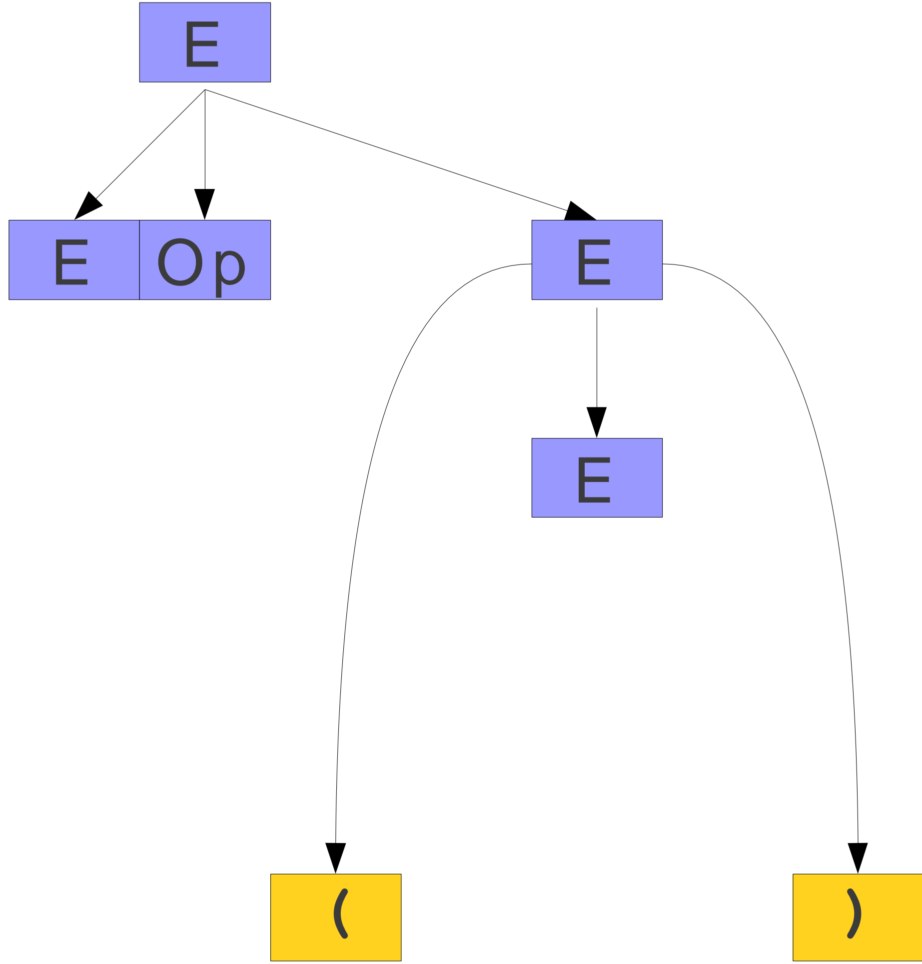
GEORGETOWN UNIVERSITY

Parse Trees

E

$\Rightarrow$ E Op E

$\Rightarrow$ E Op (E)



GEORGETOWN UNIVERSITY

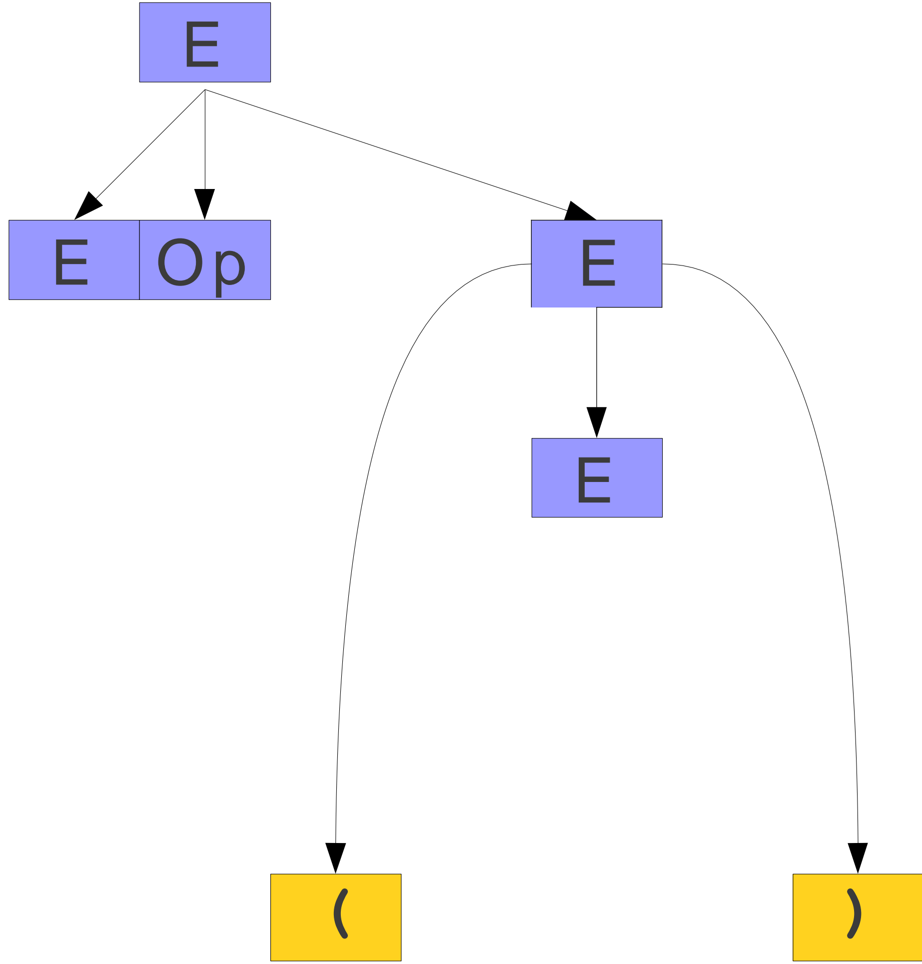
Parse Trees

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)



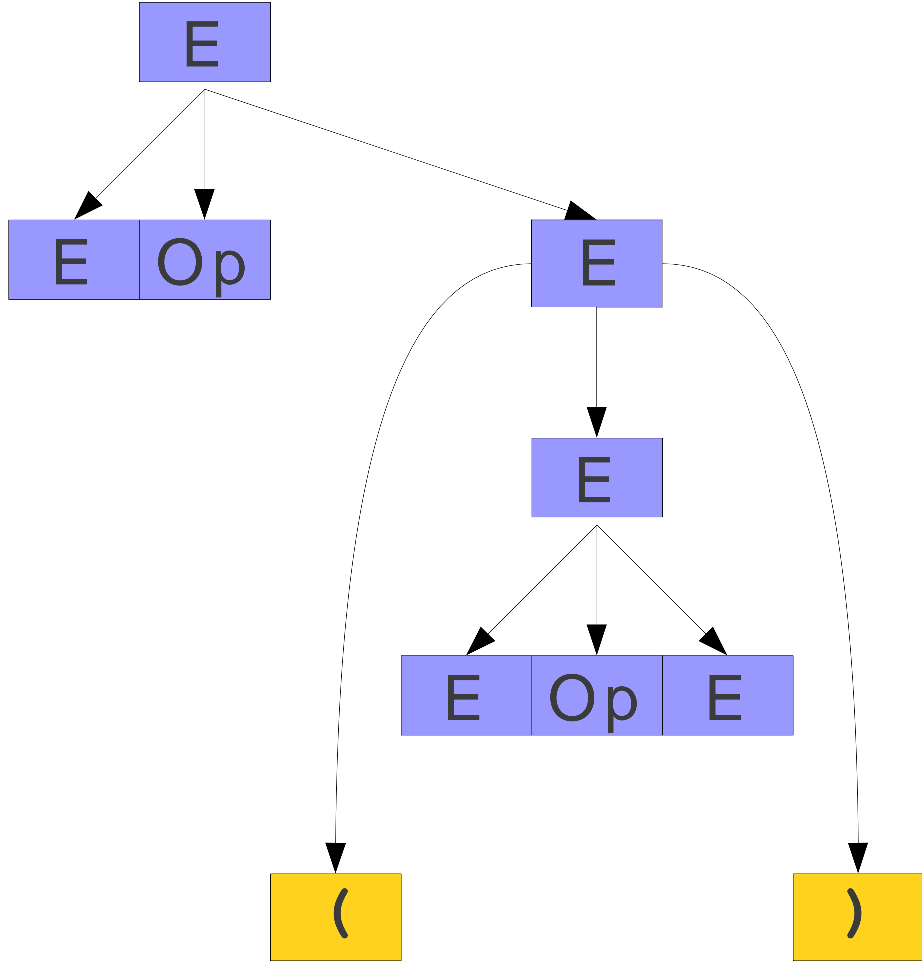
Parse Trees

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)



GEORGETOWN UNIVERSITY

Parse Trees

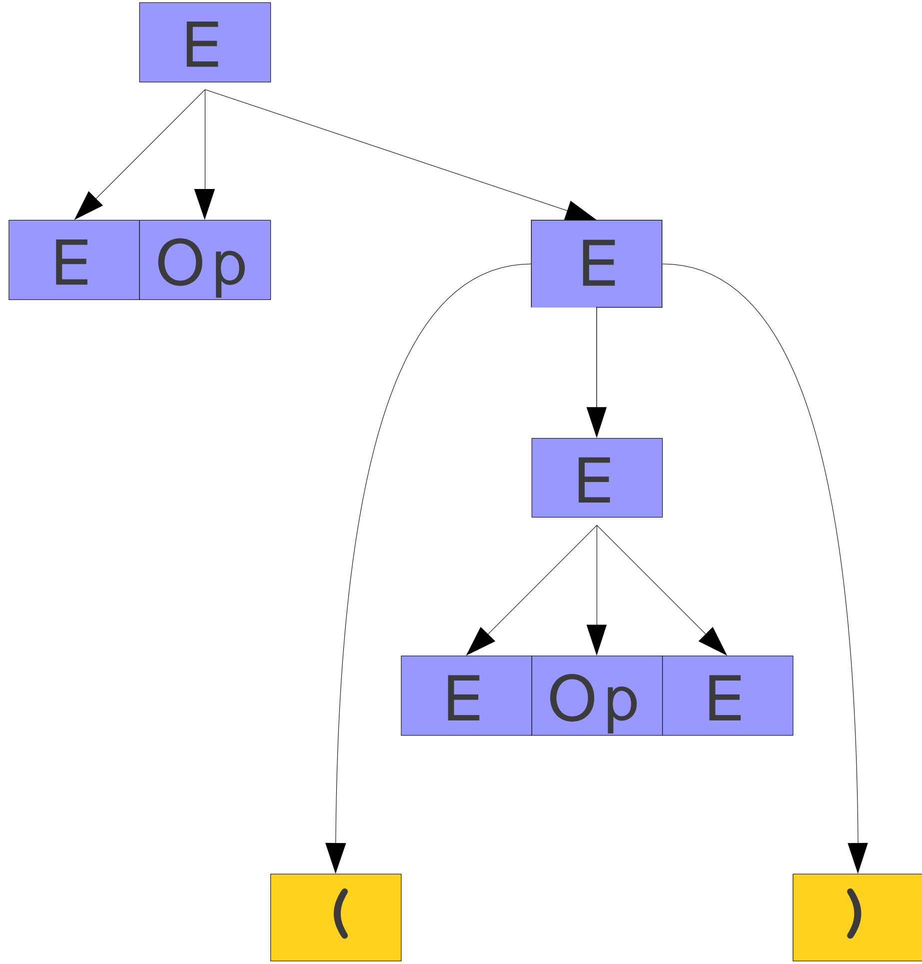
E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)



Parse Trees

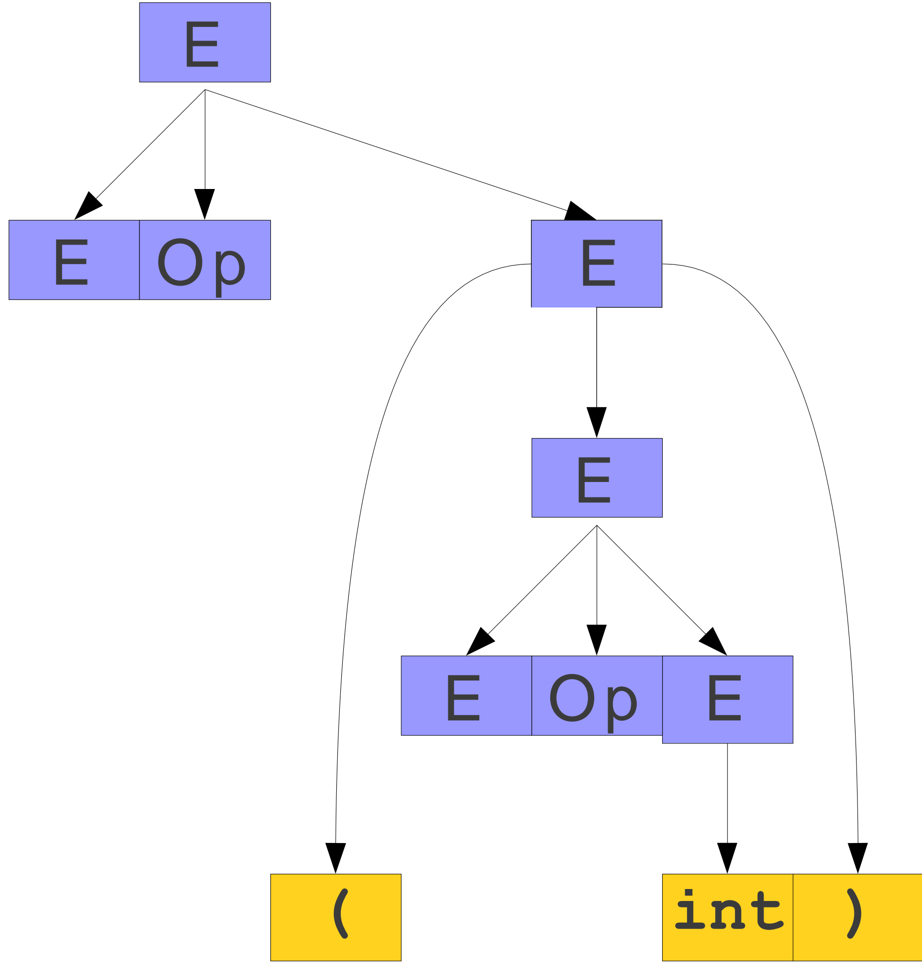
E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)



GEORGETOWN UNIVERSITY

Parse Trees

E

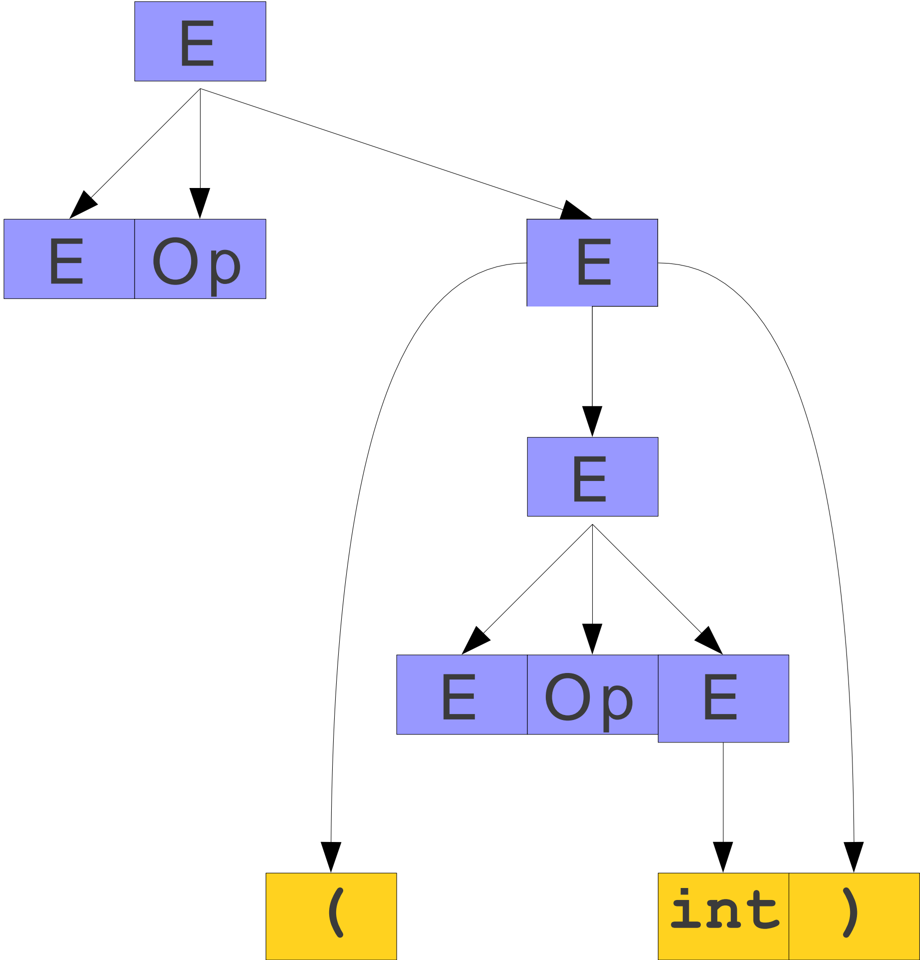
⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)



Parse Trees

E

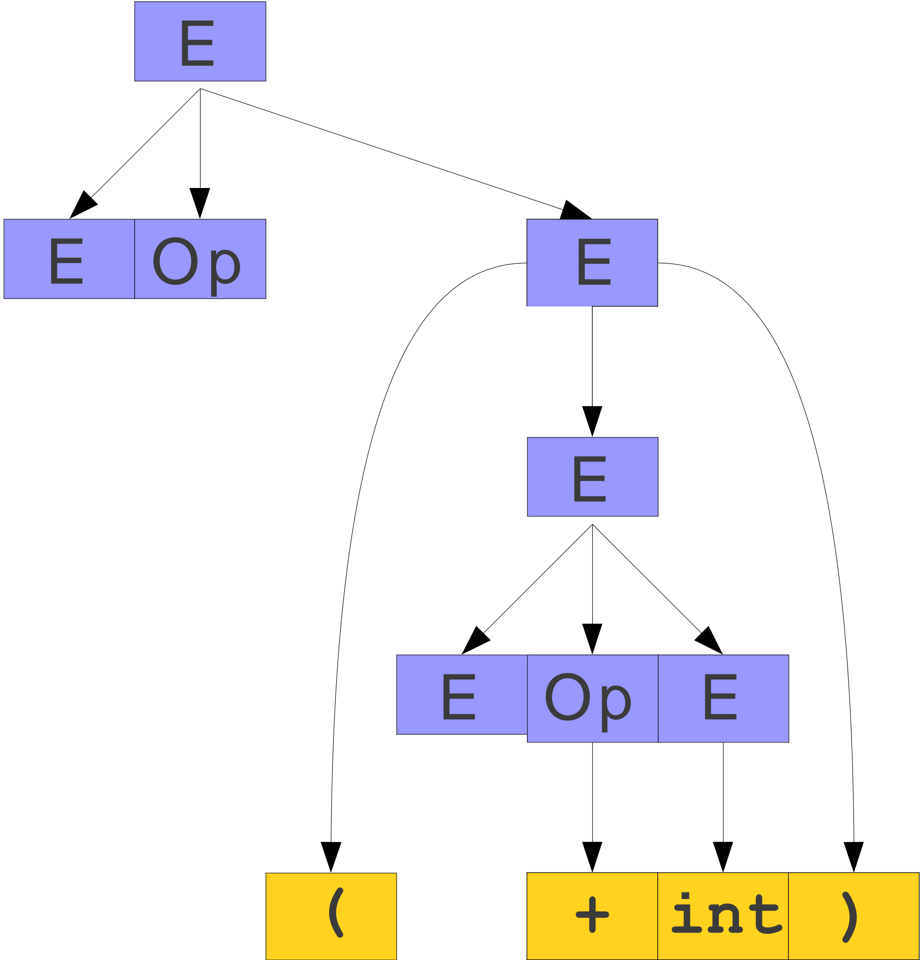
⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)



Parse Trees

E

⇒ E Op E

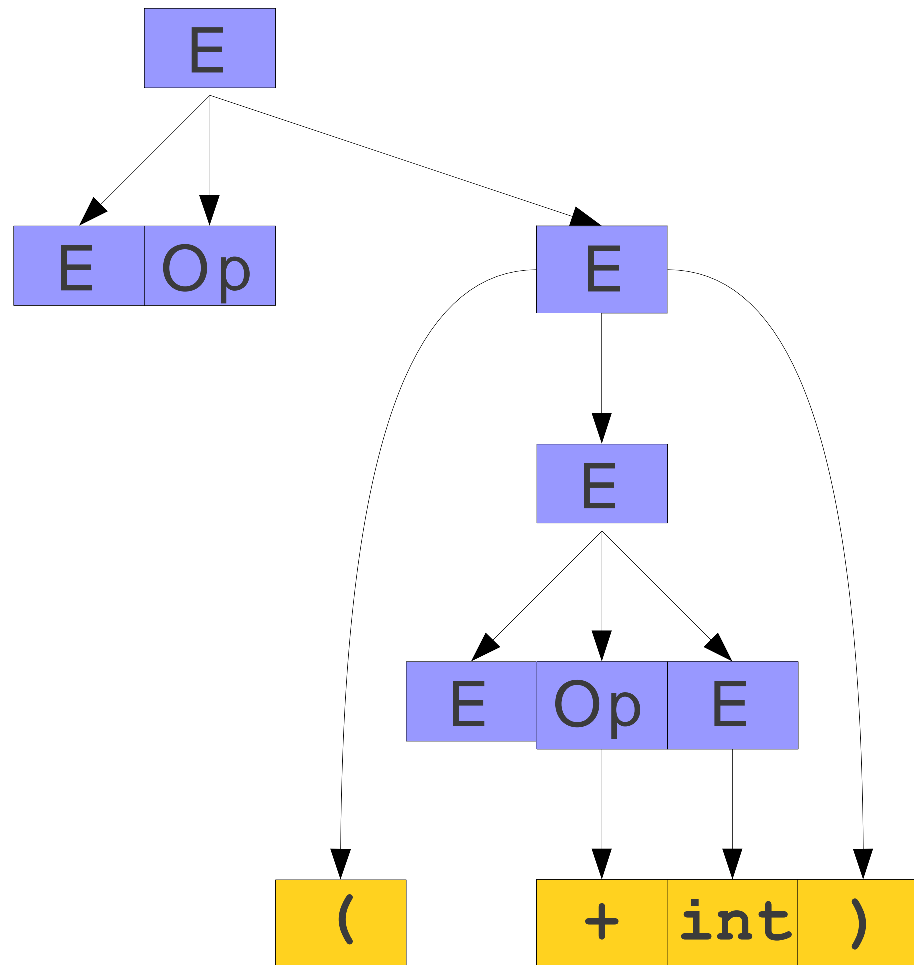
⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op **int**)

⇒ E Op (E + **int**)

⇒ E Op (**int** + **int**)



Parse Trees

E

⇒ E Op E

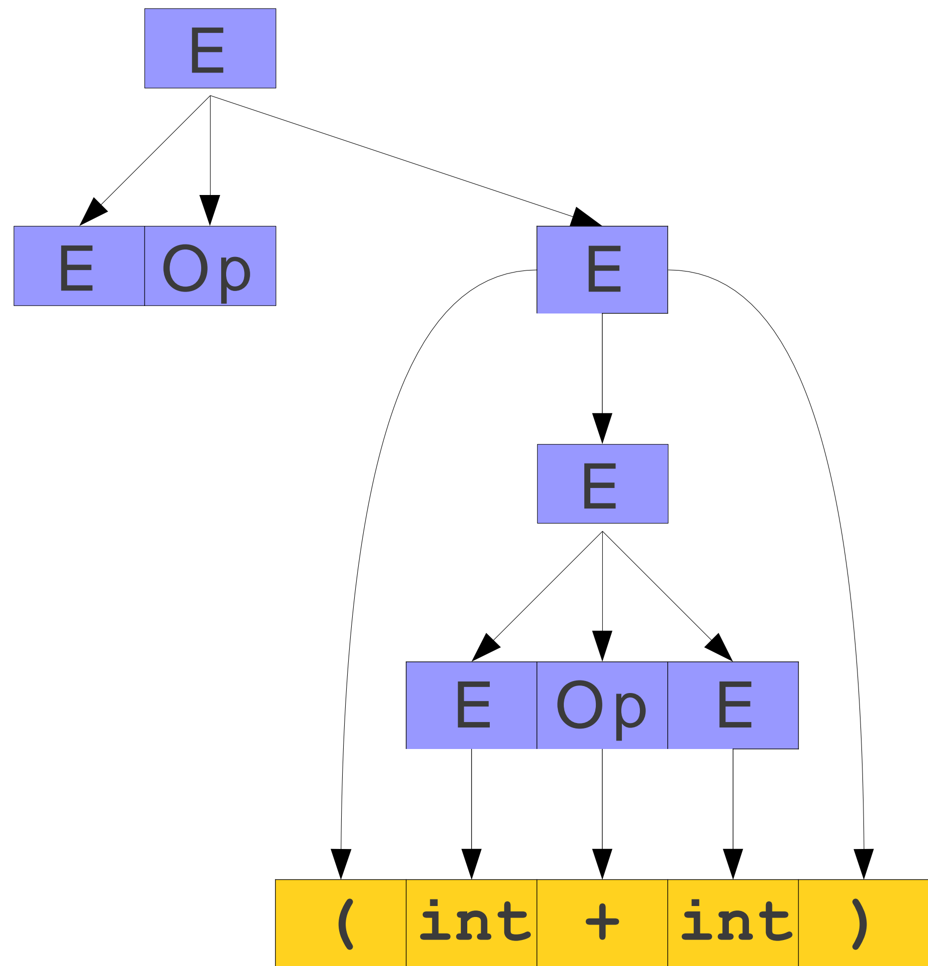
⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

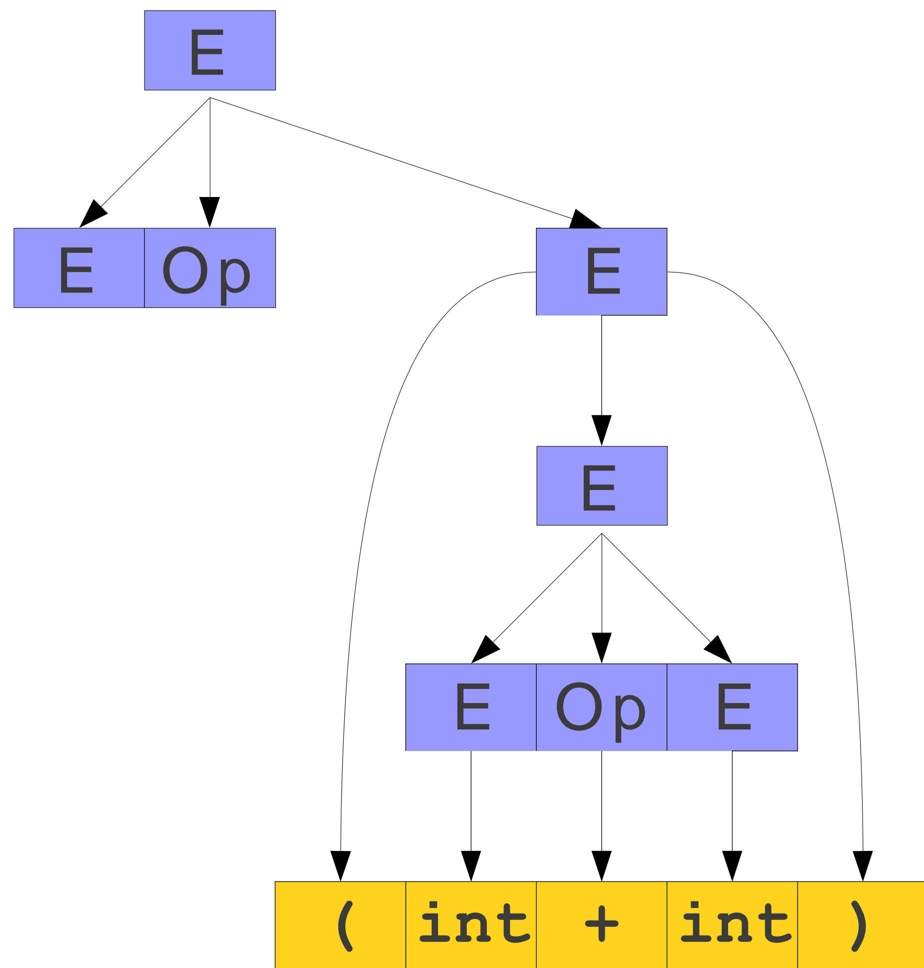
⇒ E Op (int + int)



GEORGETOWN UNIVERSITY

Parse Trees

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



Parse Trees

E

⇒ E Op E

⇒ E Op (E)

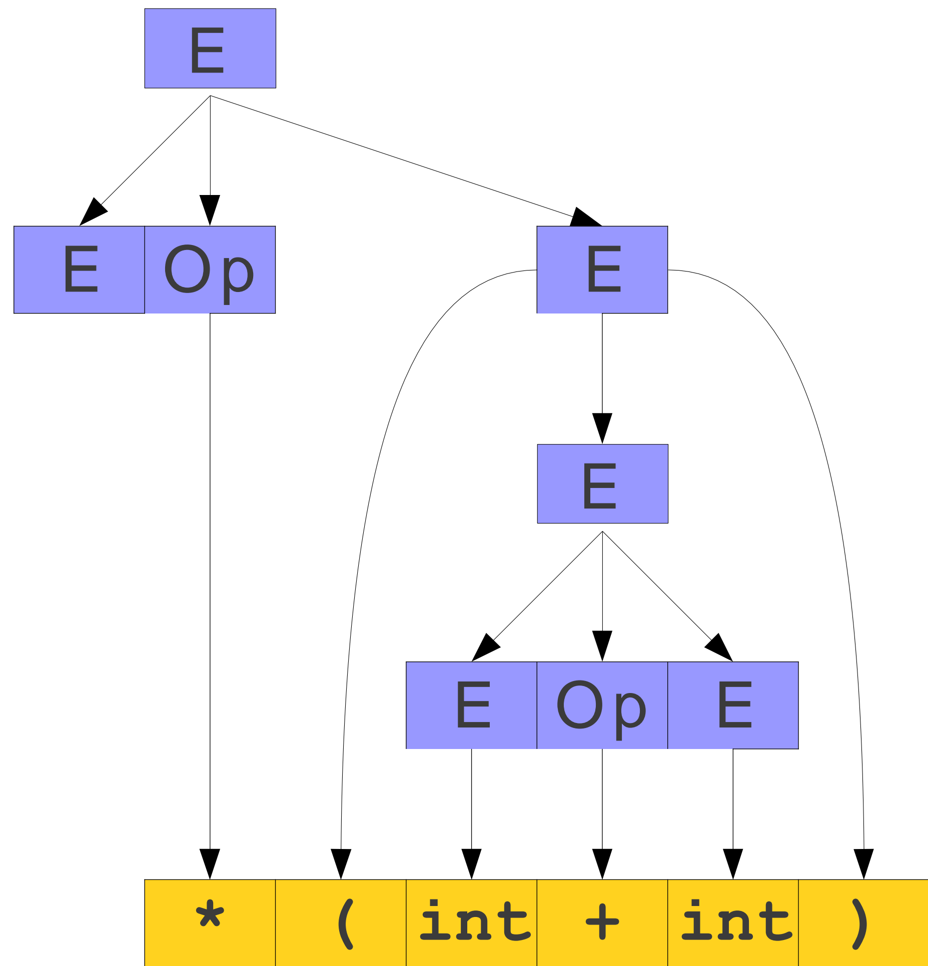
⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)

⇒ E * (int + int)



Parse Trees

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

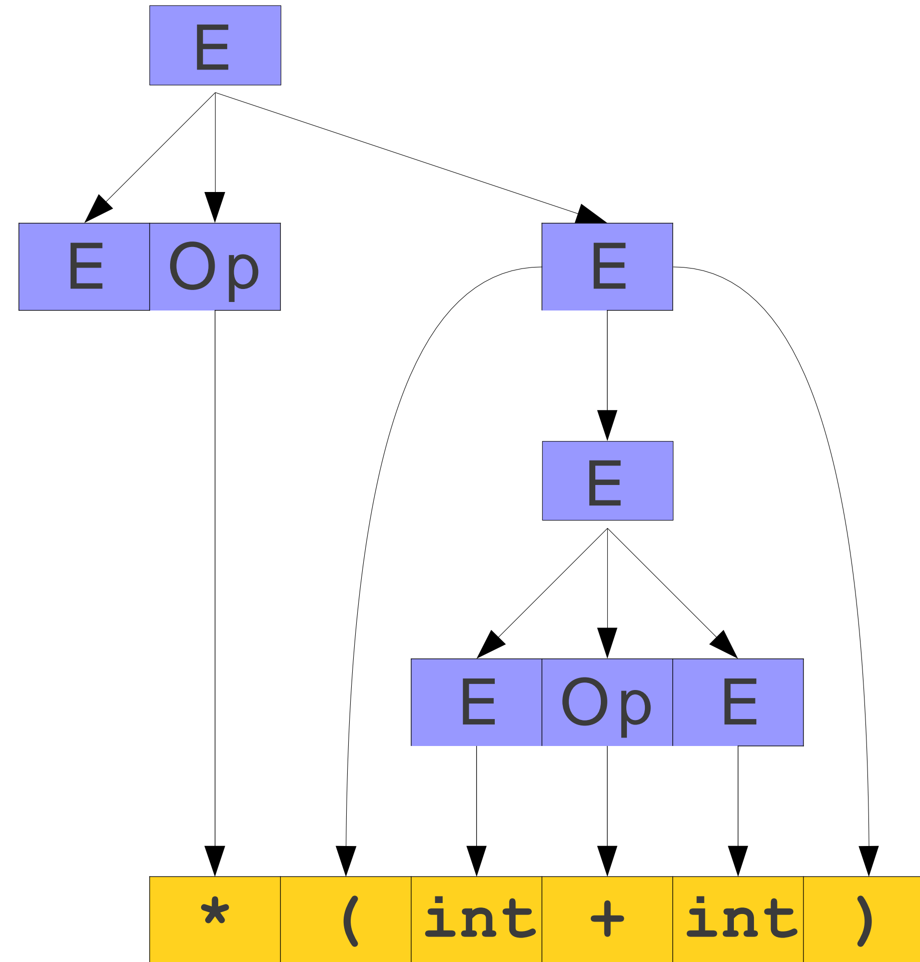
⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)

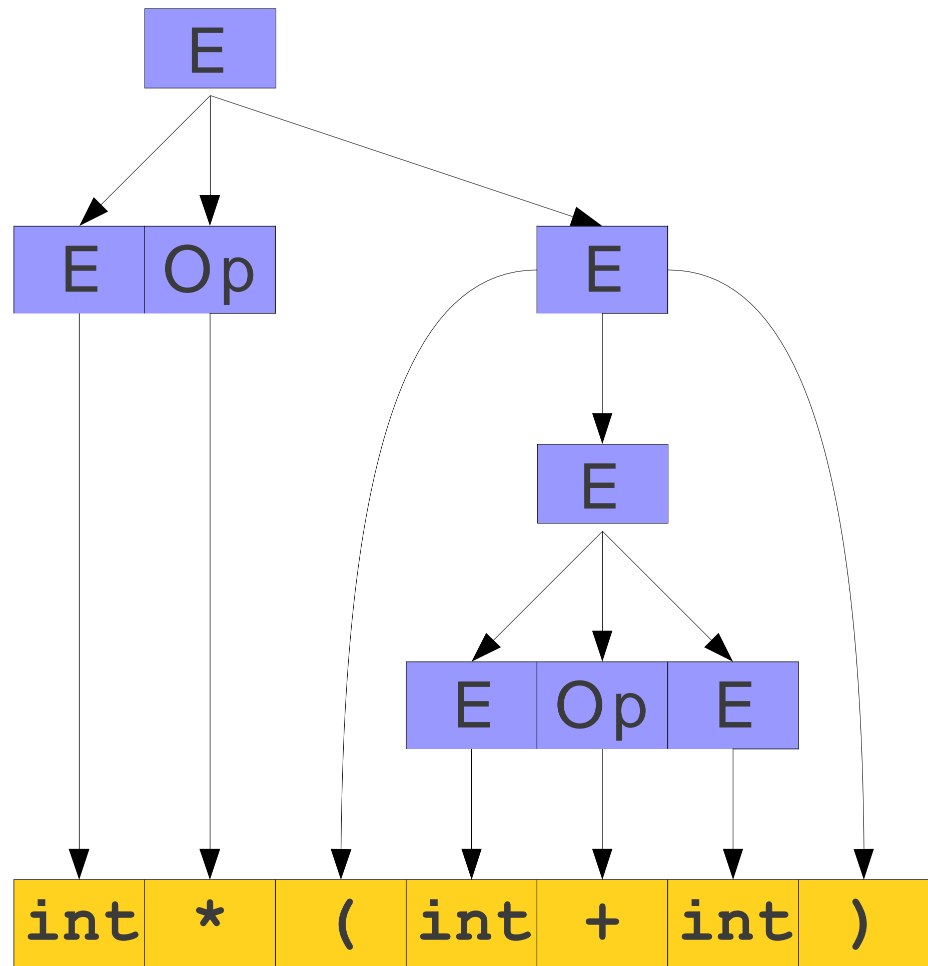
⇒ E * (int + int)

⇒ int * (int + int)

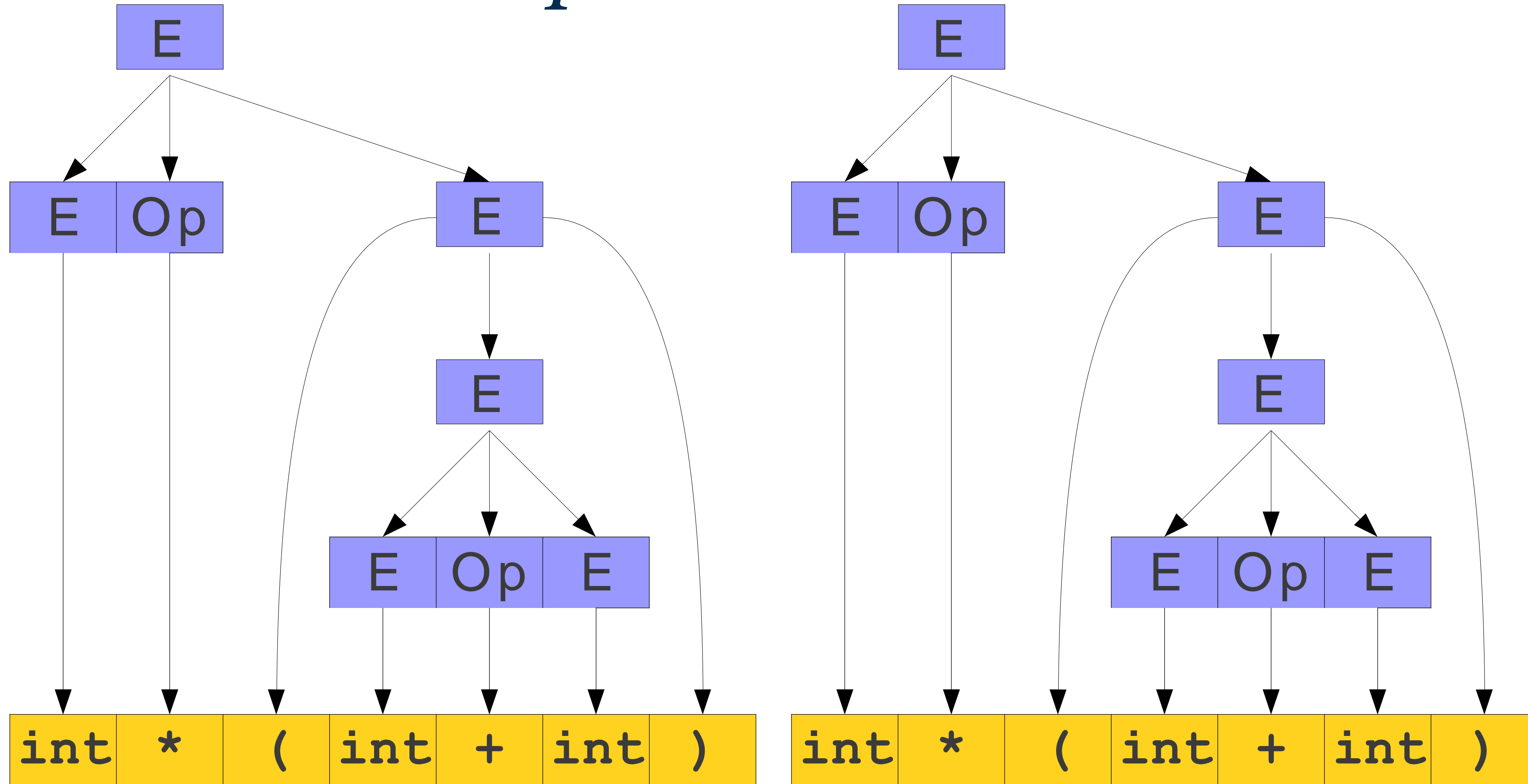


Parse Trees

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



For Comparison



Parse Trees

- A parse tree is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- In-order walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

GEORGETOWN UNIVERSITY

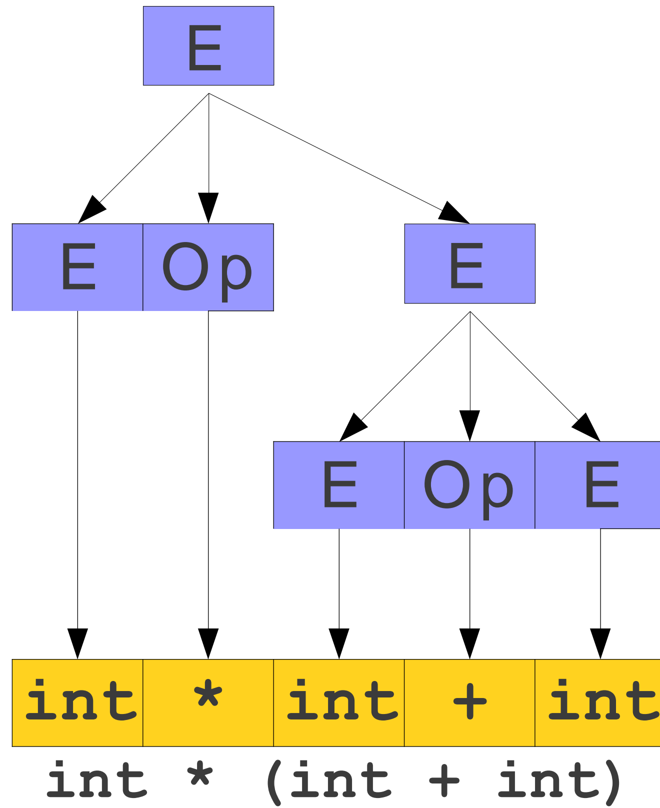
The Goal of Parsing

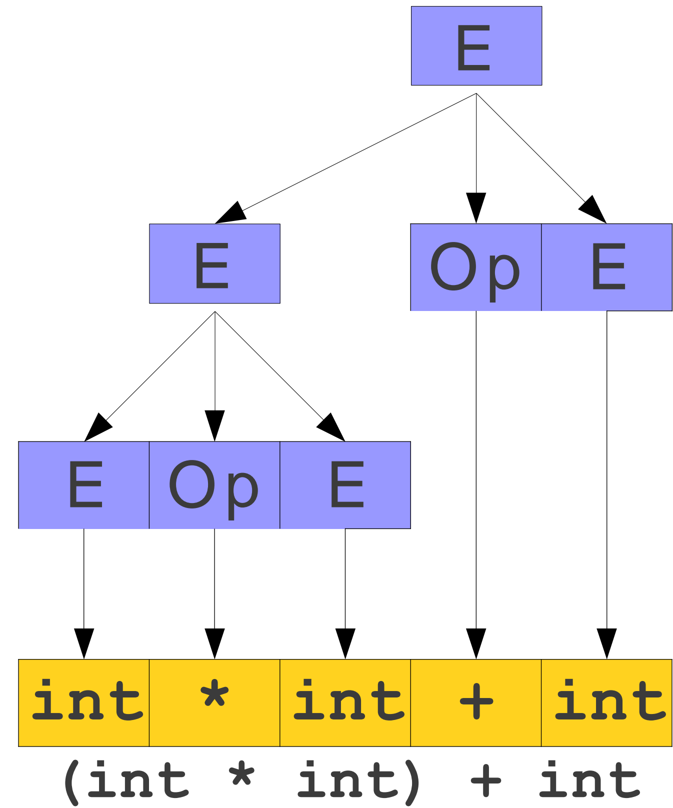
- Goal of syntax analysis: Recover the structure described by a series of tokens.
- If language is described as a CFG, goal is to recover a parse tree for the the input string.
  - Usually we do some simplifications on the tree; more on that later.
- We'll discuss how to do this next week.

GEORGETOWN UNIVERSITY

Challenges in Parsing

A Serious Problem





Ambiguity

- A CFG is said to be ambiguous if there is at least one string with two or more parse trees.
- Note that ambiguity is a property of *grammars*, not *languages*.
- There is no algorithm for converting an arbitrary ambiguous grammar into an unambiguous one.

Some languages are inherently ambiguous, meaning that no unambiguous grammar exists for them.

There is no algorithm for detecting whether an arbitrary grammar is ambiguous.

Is Ambiguity a Problem?

- Depends on semantics.

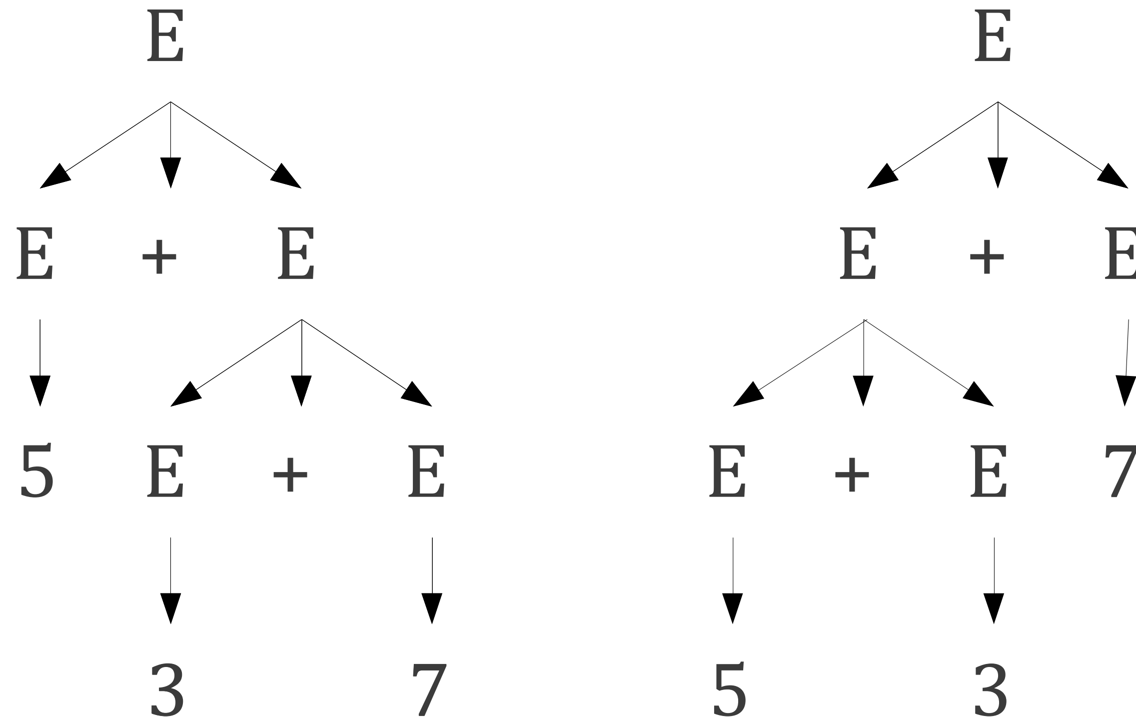
$$E \rightarrow \texttt{int} \mid E + E$$

GEORGETOWN UNIVERSITY

Is Ambiguity a Problem?

- Depends on semantics.

$E \rightarrow$ **int** | E + E



GEORGETOWN UNIVERSITY

Is Ambiguity a Problem?

- Depends on semantics.

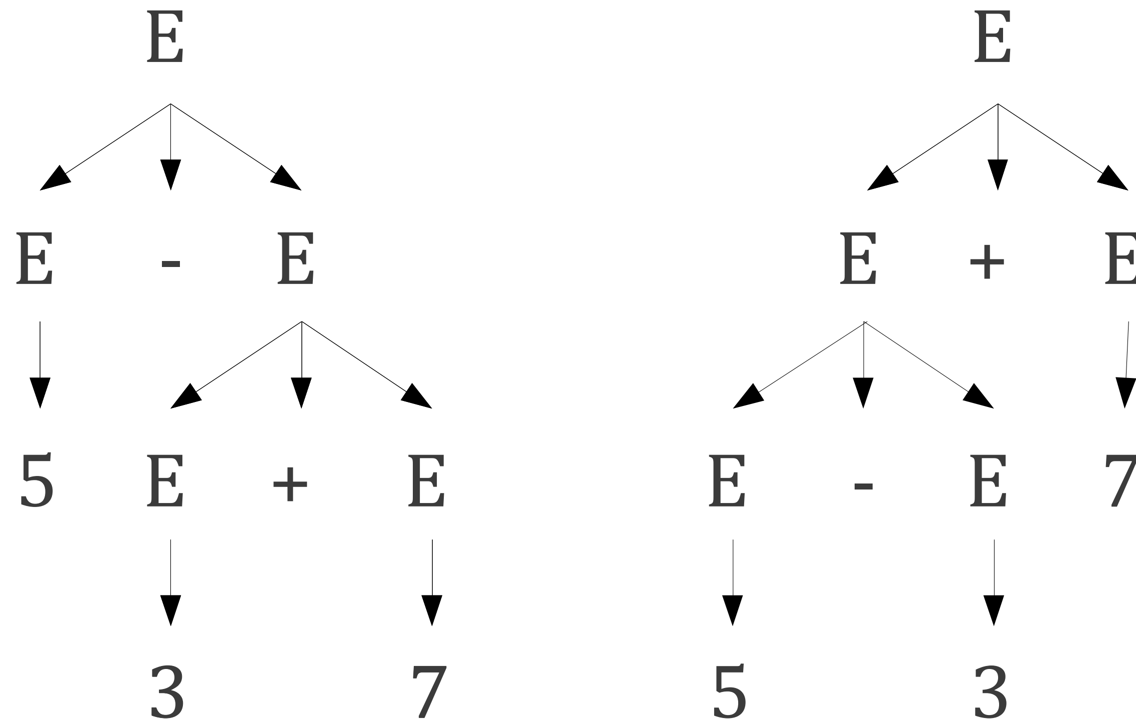
E → **int** | E + E | E - E

GEORGETOWN UNIVERSITY

Is Ambiguity a Problem?

- Depends on semantics.

$E \rightarrow$ **int** | E + E | E - E



GEORGETOWN UNIVERSITY

Resolving Ambiguity

- If a grammar can be made unambiguous at all, it is usually made unambiguous through layering.
- Have exactly one way to build each piece of the string.
- Have exactly one way of combining those pieces back together.

Example: Balanced Parentheses

- Consider the language of all strings of balanced parentheses.
- Examples:
  - ε
  - ( )
  - (( ) ( ))
  - ((( ))) (( )) ( )
- Here is one possible grammar for balanced parentheses:

$$P \rightarrow \varepsilon \mid PP \mid (P)$$

GEORGETOWN UNIVERSITY

Balanced Parentheses

- Given the grammar P → ε| PP | (P)
- How might we generate the string (()())?

GEORGETOWN UNIVERSITY

Balanced Parentheses

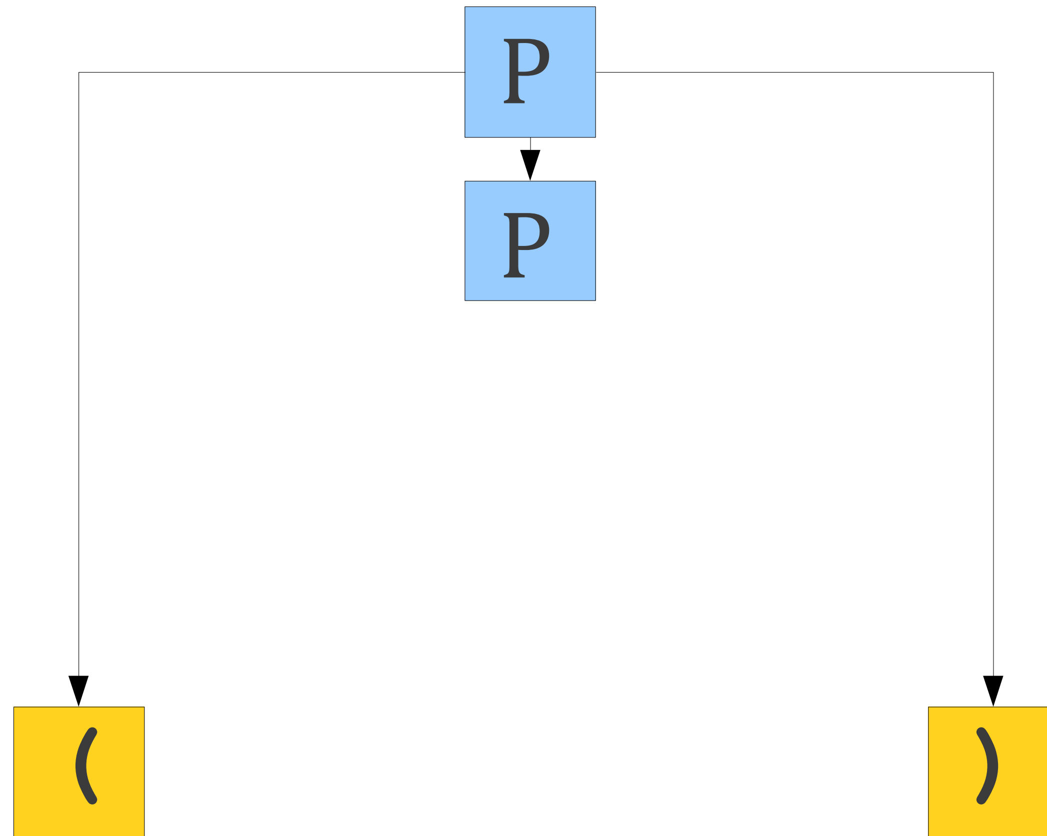
- Given the grammar $P \rightarrow \varepsilon \mid PP \mid (P)$
- How might we generate the string (()())?

P

GEORGETOWN UNIVERSITY

Balanced Parentheses

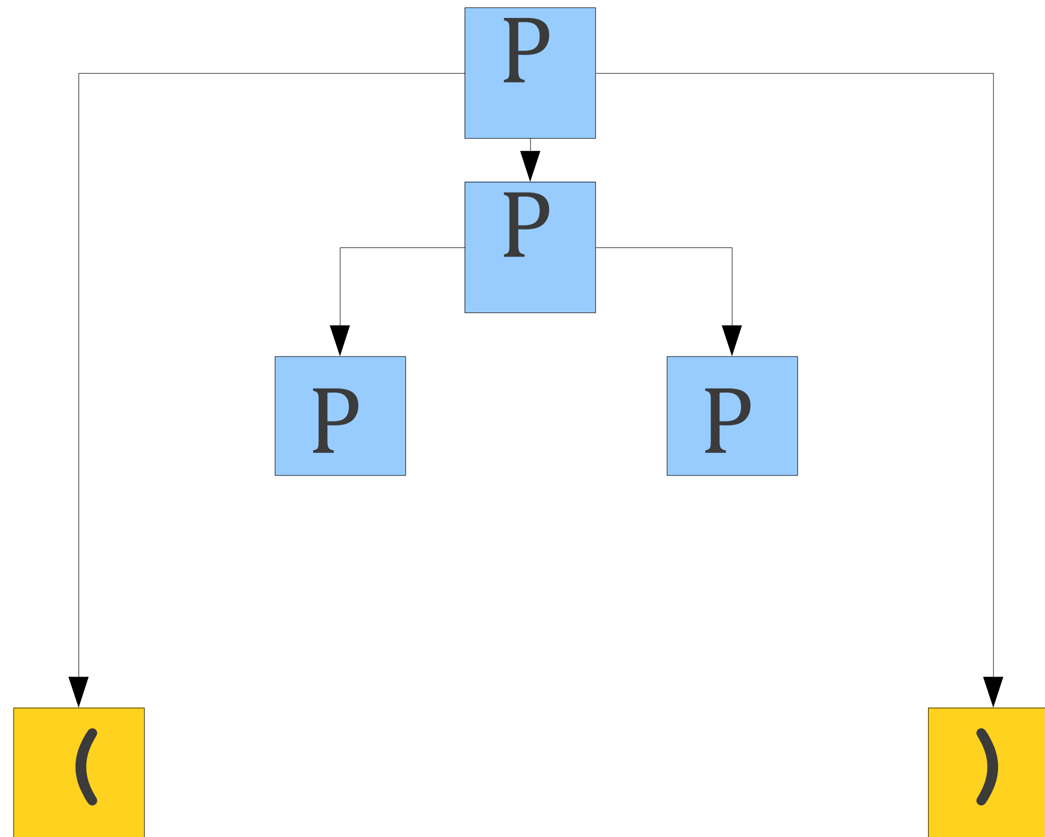
- Given the grammar P → ε| PP | (P)
- How might we generate the string (()())?



GEORGETOWN UNIVERSITY

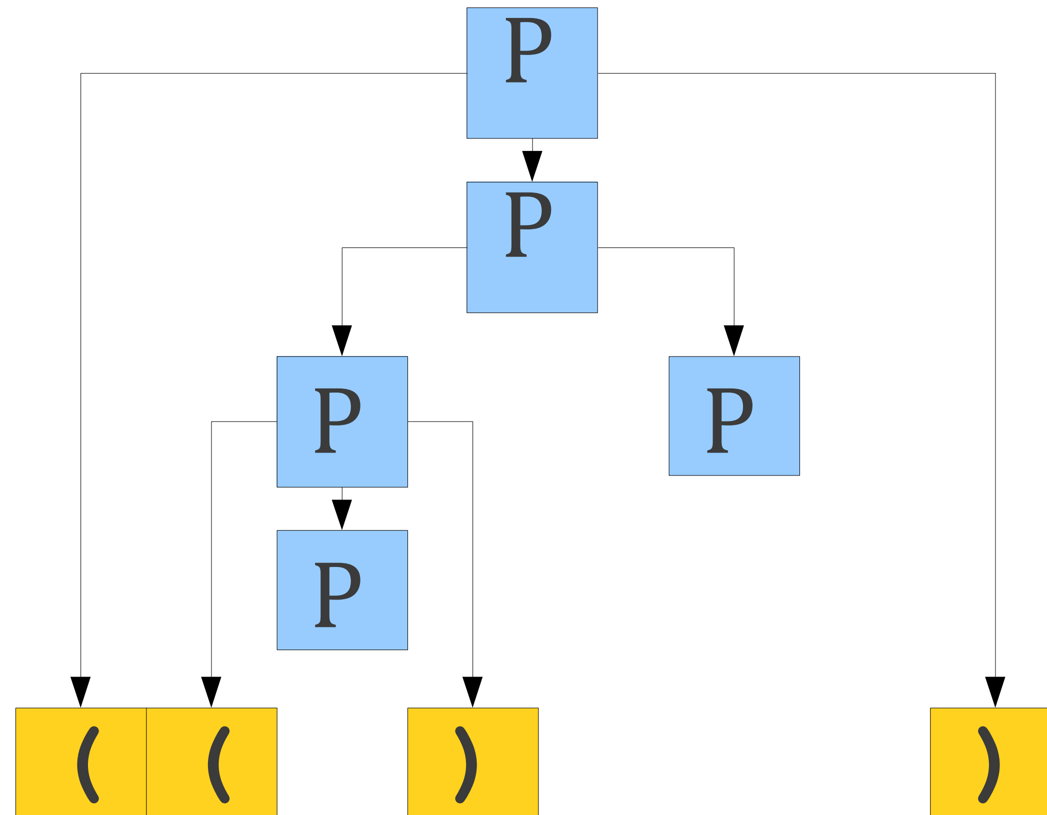
Balanced Parentheses

- Given the grammar $P \rightarrow \varepsilon \mid PP \mid (P)$
- How might we generate the string (()())?



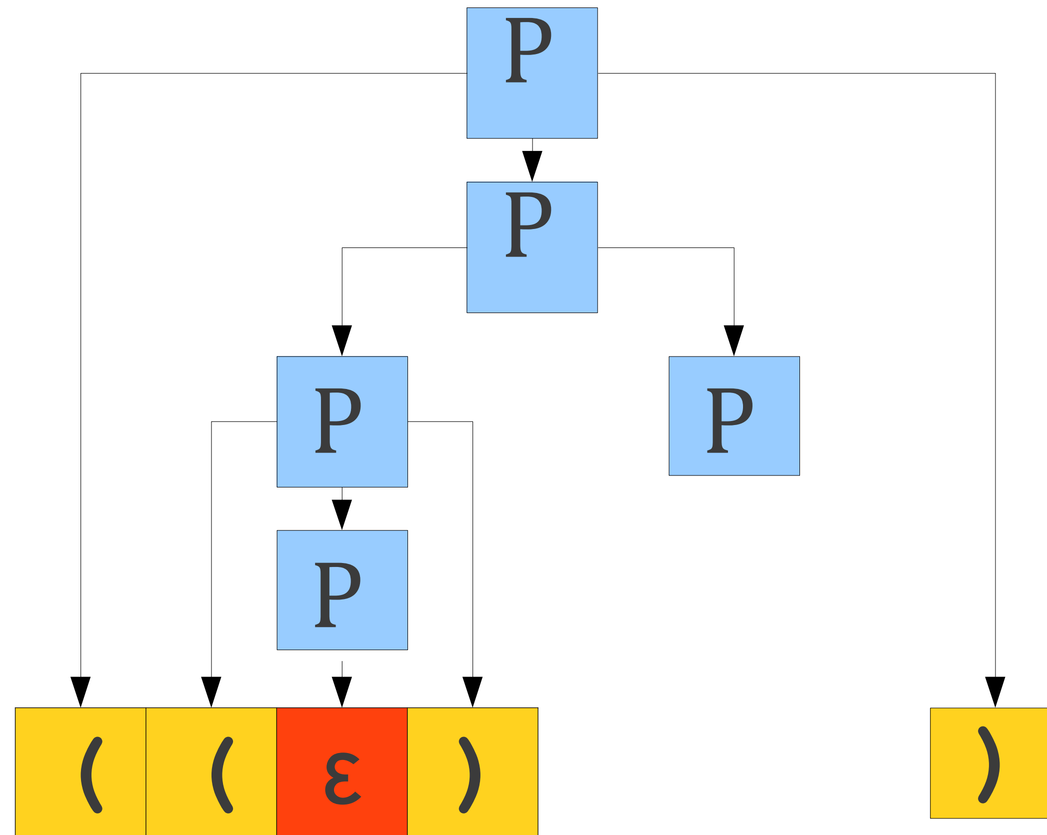
Balanced Parentheses

- Given the grammar $P \rightarrow \varepsilon \mid PP \mid (P)$
- How might we generate the string (()())?



Balanced Parentheses

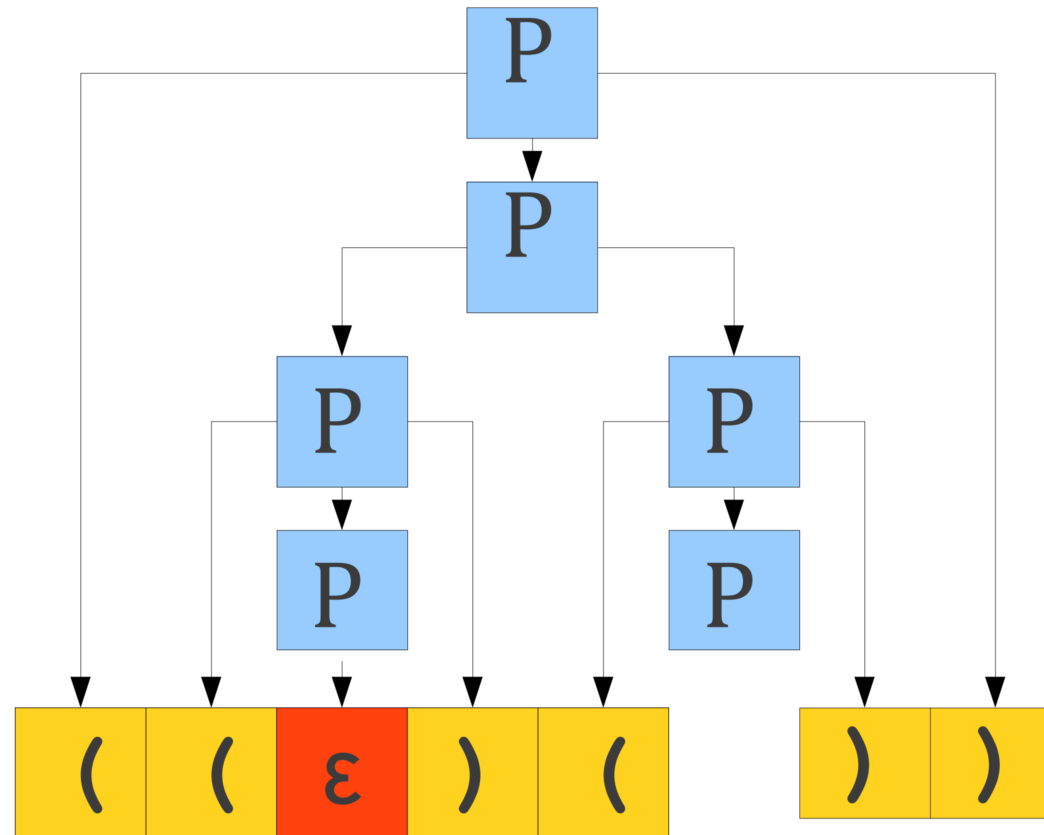
- Given the grammar P → ε| PP | (P)
- How might we generate the string (()())?



GEORGETOWN UNIVERSITY

Balanced Parentheses

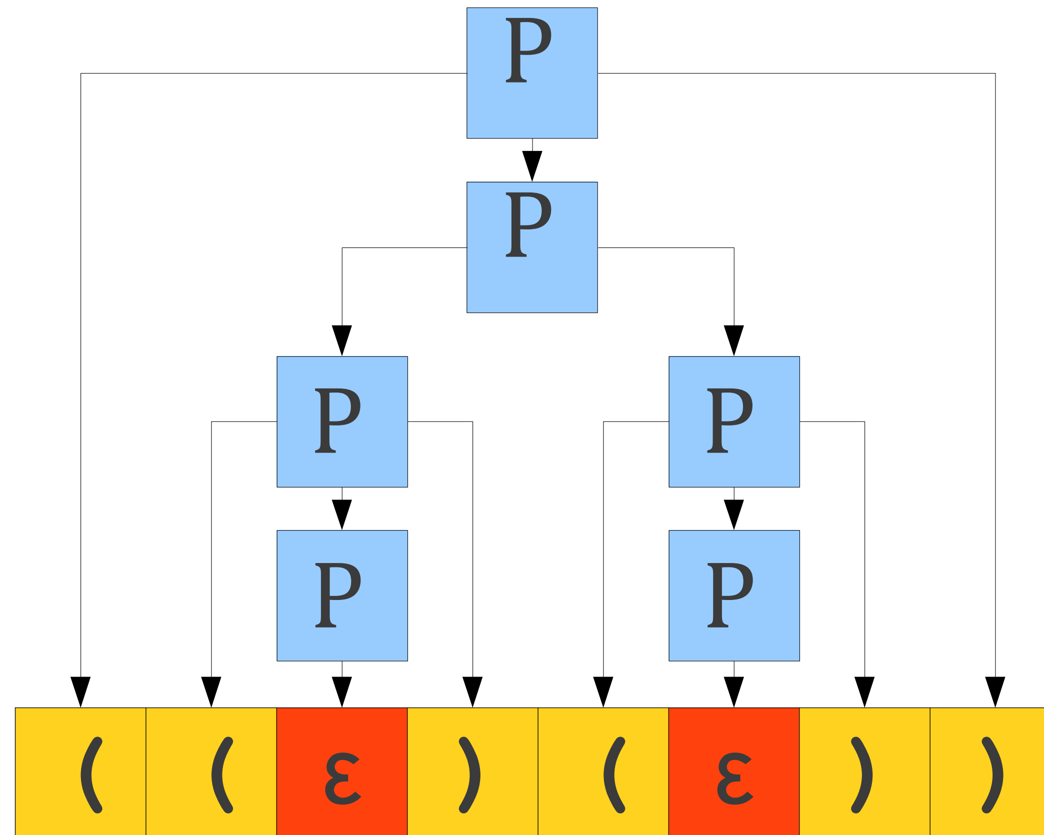
- Given the grammar P → ε| PP | (P)
- How might we generate the string (()())?



GEORGETOWN UNIVERSITY

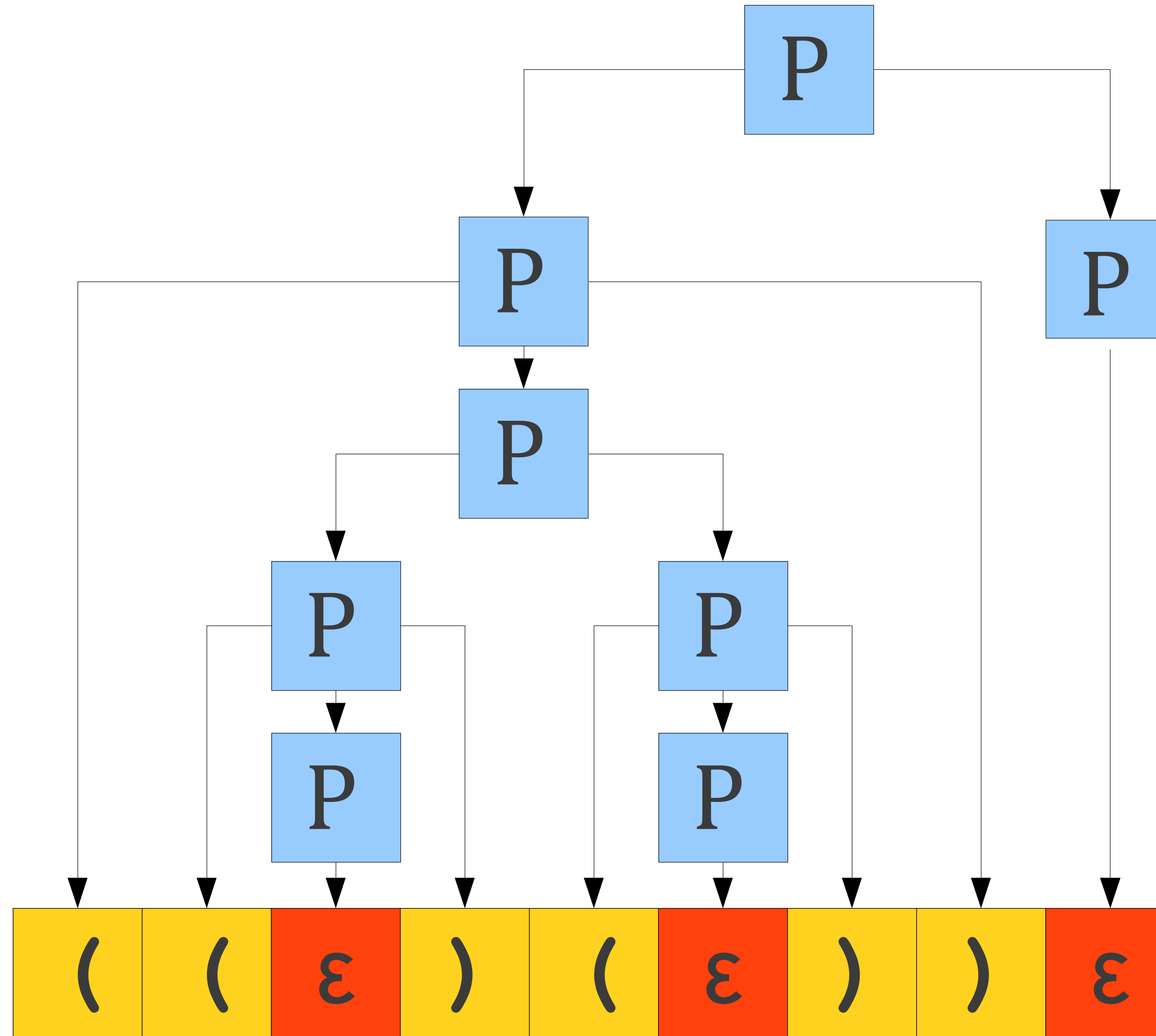
Balanced Parentheses

- Given the grammar P → ε| PP | (P)
- How might we generate the string (()())?



GEORGETOWN UNIVERSITY

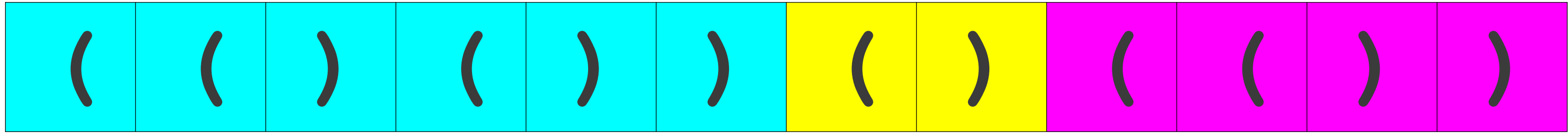
HOWEVER … we can also create this tree!

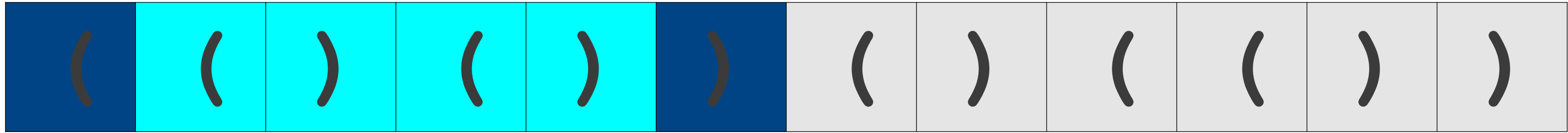


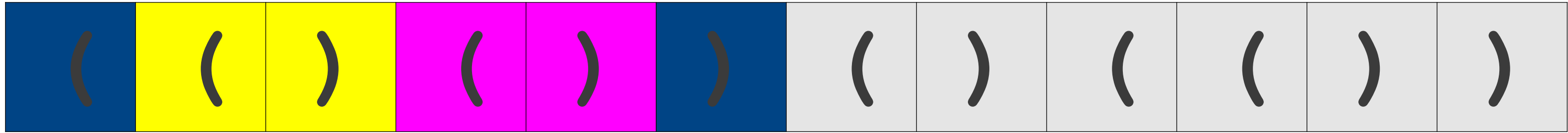
How to resolve this ambiguity?

(	(	)	(	)	)	(	)	(	(	)	)

GEORGETOWN UNIVERSITY







GEORGETOWN UNIVERSITY

Rethinking Parentheses

- A string of balanced parentheses is a sequence of strings that are themselves balanced parentheses.
- To avoid ambiguity, we can build the string in two steps:
  - Decide how many different substrings we will glue together.
  - Build each substring independently.

Do not allow for multiple recursive generations in the same rule!

Building Parentheses

- Spread a string of parentheses across the string. There is exactly one way to do this for any number of parentheses.
- Expand out each substring by adding in parentheses and repeating.

$$S \rightarrow P\ S \mid \varepsilon$$

$$P \rightarrow (\ S\ )$$

GEORGETOWN UNIVERSITY

Building Parentheses

$$S \rightarrow P\ S \mid \varepsilon$$

$$P \rightarrow (\ S\ )$$

$$
\begin{aligned}
& S \\
\Rightarrow\ & PS \\
\Rightarrow\ & PPS \\
\Rightarrow\ & PP \\
\Rightarrow\ & (S)P \\
\Rightarrow\ & (S)(S) \\
\Rightarrow\ & (PS)(S) \\
\Rightarrow\ & (P)(S) \\
\Rightarrow\ & ((S))(S) \\
\Rightarrow\ & (())(S) \\
\Rightarrow\ & (())()
\end{aligned}
$$

GEORGETOWN UNIVERSITY

Context-Free Grammars

- A regular expression can be
  - Any letter
  - ε
  - The concatenation of regular expressions.
  - The union of regular expressions.
  - The Kleene closure of a regular expression.
  - A parenthesized regular expression.

GEORGETOWN UNIVERSITY

Context-Free Grammars

- This gives us the following CFG:

  R → a | b | c | …

  R → “ε”

  R → RR

  R → R “|” R

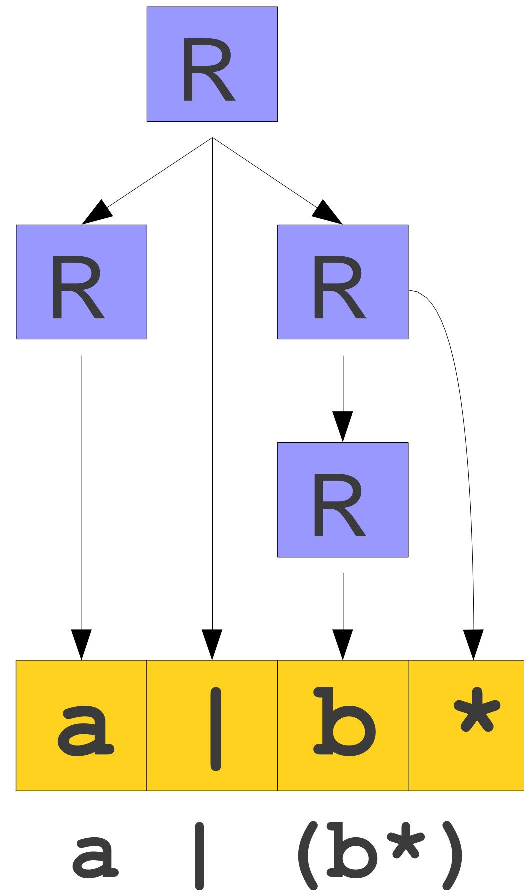
  R → R*

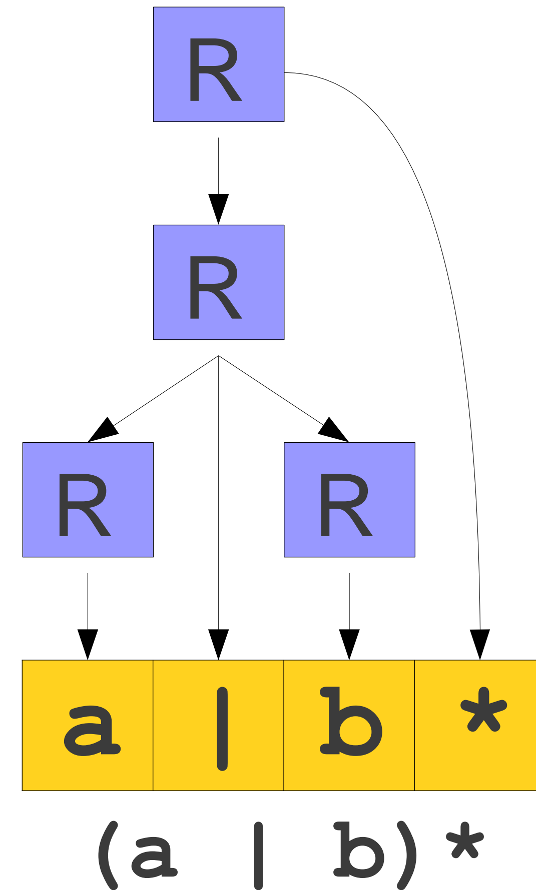
  R → (R)

GEORGETOWN UNIVERSITY

An Ambiguous Grammar

R → a | b | c | …
R → "ε"
R → RR
R → R "|" R
R → R*
R → (R)





GEORGETOWN UNIVERSITY

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → a | b | c | …

R → "ε"

R → RR

R → R "|" R

R → R*

R → (R)

a	a	|	b	*

GEORGETOWN UNIVERSITY

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → a | b | c | …

R → "ε"

R → RR

R → R "|" R

R → R*

R → (R)

a	a	|	b	*

GEORGETOWN UNIVERSITY

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → a | b | c | …

R → "ε"

R → RR

R → R "|" R

R → R*

R → (R)

a	a	|	b	*

GEORGETOWN UNIVERSITY

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → a | b | c | ...

R → "ε"

R → RR

R → R "|" R

R → R*

R → (R)

a	a	|	b	*

GEORGETOWN UNIVERSITY

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

  R → a | b | c | …

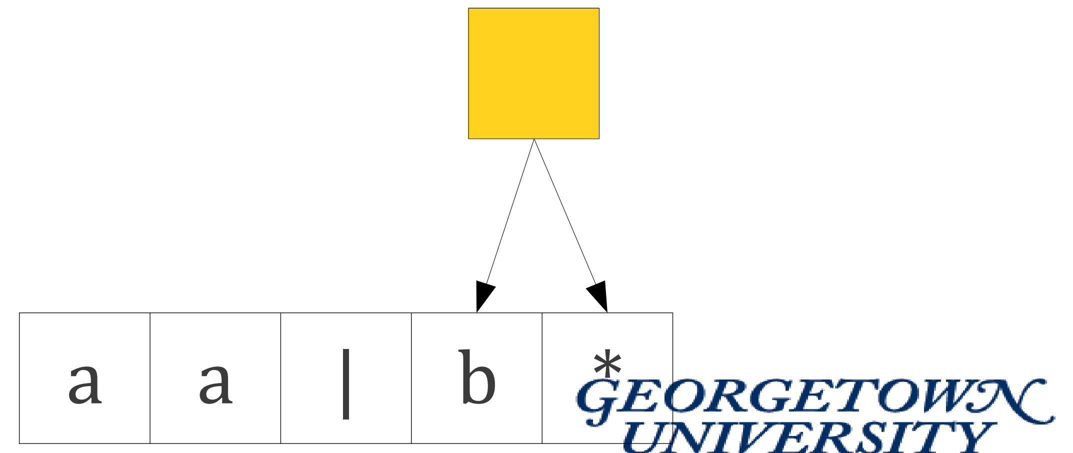
  R → "ε"

  R → RR

  R → R "|" R

  R → R*

  R → (R)



GEORGETOWN UNIVERSITY

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → **a** | **b** | **c** | …

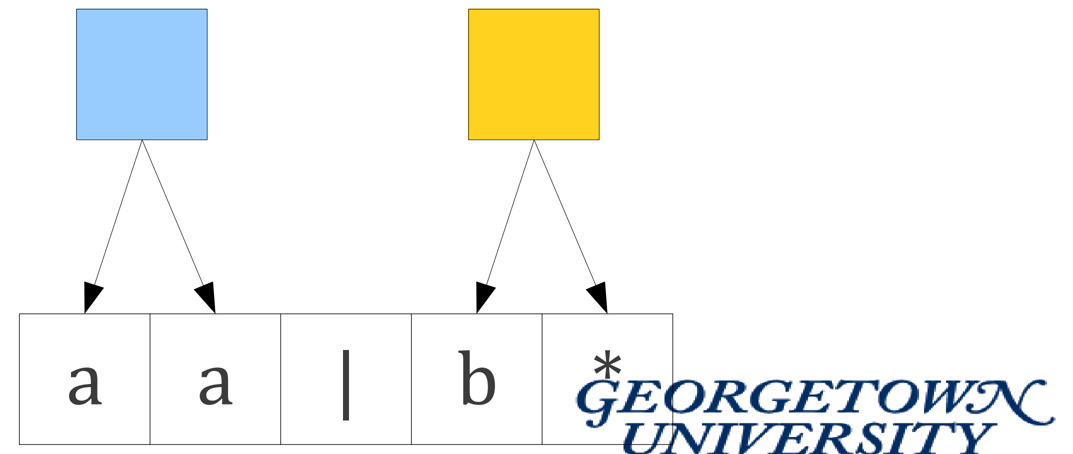
R → "**ε**"

R → RR

R → R "**|**" R

R → R*****

R → **(**R**)**



GEORGETOWN UNIVERSITY

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → **a** | **b** | **c** | …

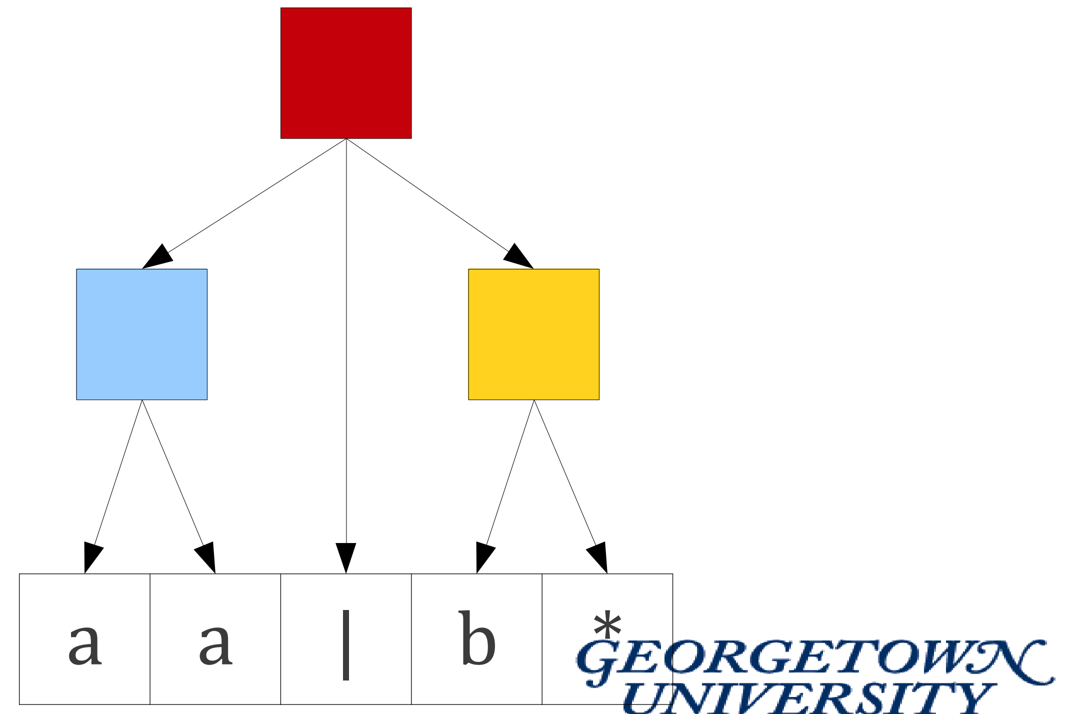
R → "**ε**"

R → RR

R → R "**|**" R

R → R*****

R → **(**R**)**



GEORGETOWN UNIVERSITY

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → a | b | c | …

R → "ε"

R → RR

R → R "|" R

R → R*

R → (R)

R → S | R "|" S

S → T | ST

T → U | T*

U → a | b | c | …

U → "ε"

U → (R)

GEORGETOWN UNIVERSITY

Why is this unambiguous?

R → S | R "|" S

S → T | ST

T → U | T*

U → a | b | …

U → "ε"

U → (R)

GEORGETOWN UNIVERSITY

Why is this unambiguous?

R → S | R "|" S

S → T | ST

T → U | T*

U → a | b | …

U → "ε"

U → (R)

Only generates "atomic" expressions

GEORGETOWN UNIVERSITY

Why is this unambiguous?

R → S | R "|" S

S → T | ST

T → U | T*

U → a | b | …

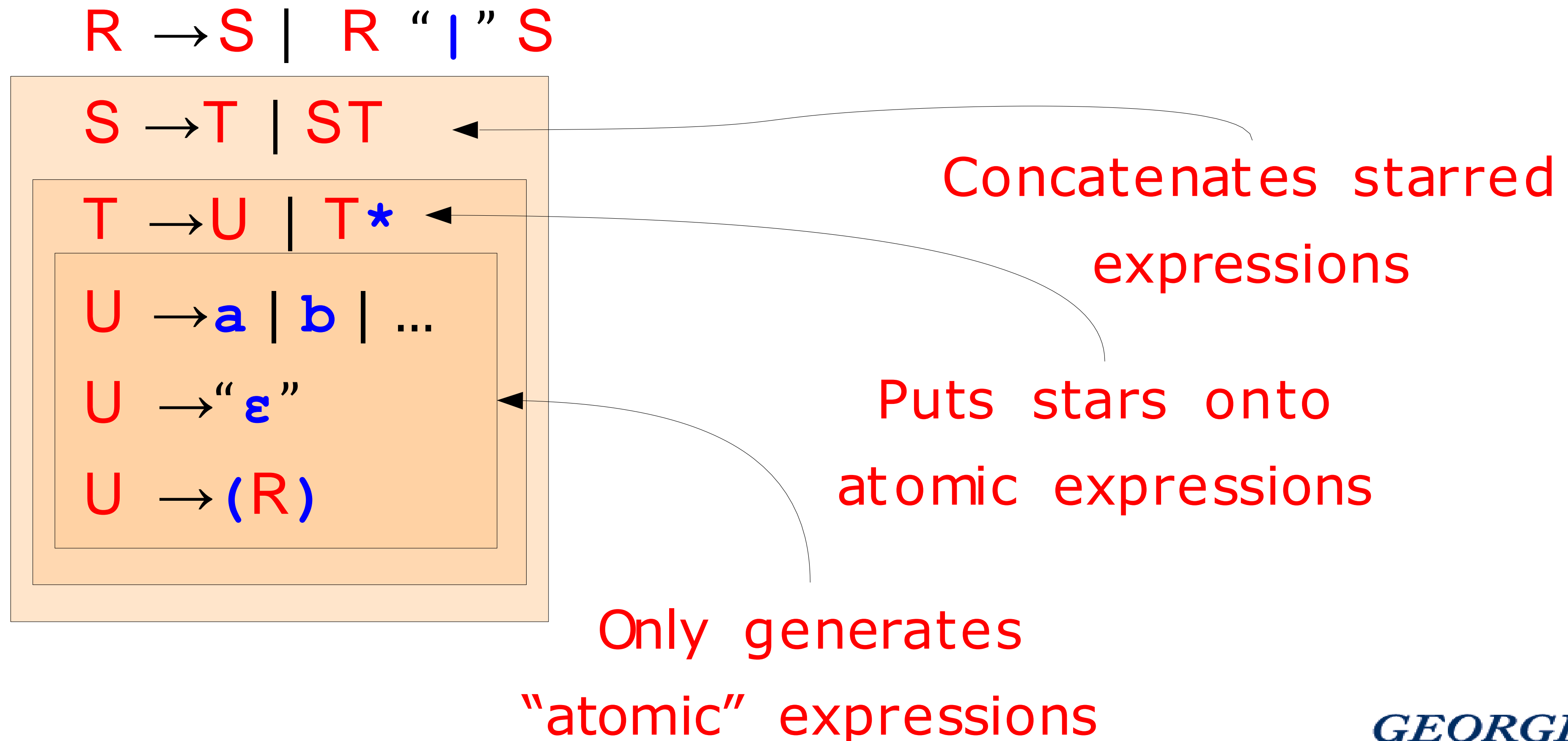
U → "ε"

U → (R)

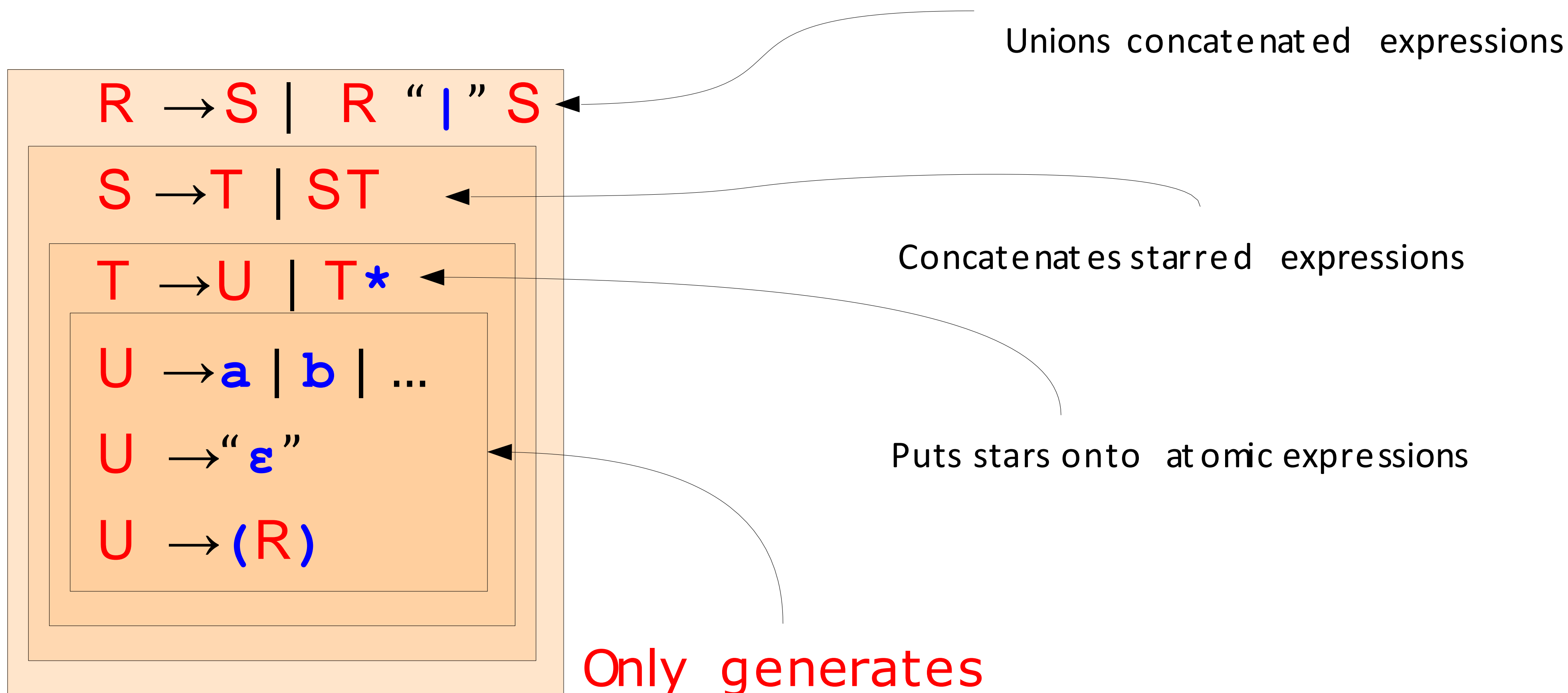
Puts stars onto atomic expressions

Only generates "atomic" expressions

Why is this unambiguous?



Why is this unambiguous?



R

$R \rightarrow S \mid R$ "|" S

$S \rightarrow T \mid ST$

$T \rightarrow U \mid T*$

$U \rightarrow$ a | b | c | ...

$U \rightarrow$ "ε"

$U \rightarrow$ (R)

a	b	|	c	|	a	*

GEORGETOWN UNIVERSITY

R → S | R "|" S

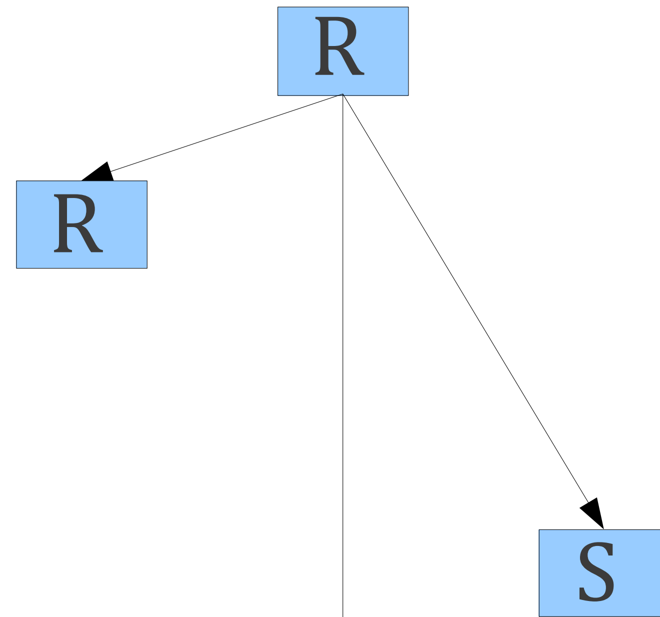
S → T | ST

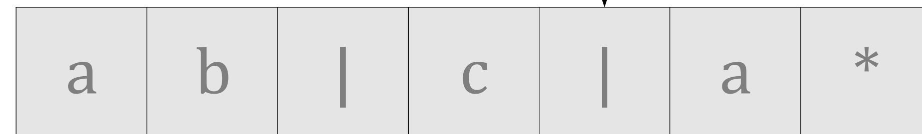
T → U | T*

U → a | b | c | …

U → "ε"

U → (R)





GEORGETOWN UNIVERSITY

R → S | R "|" S

S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)

GETOWN UNIVERSITY

R → S | R “|” S

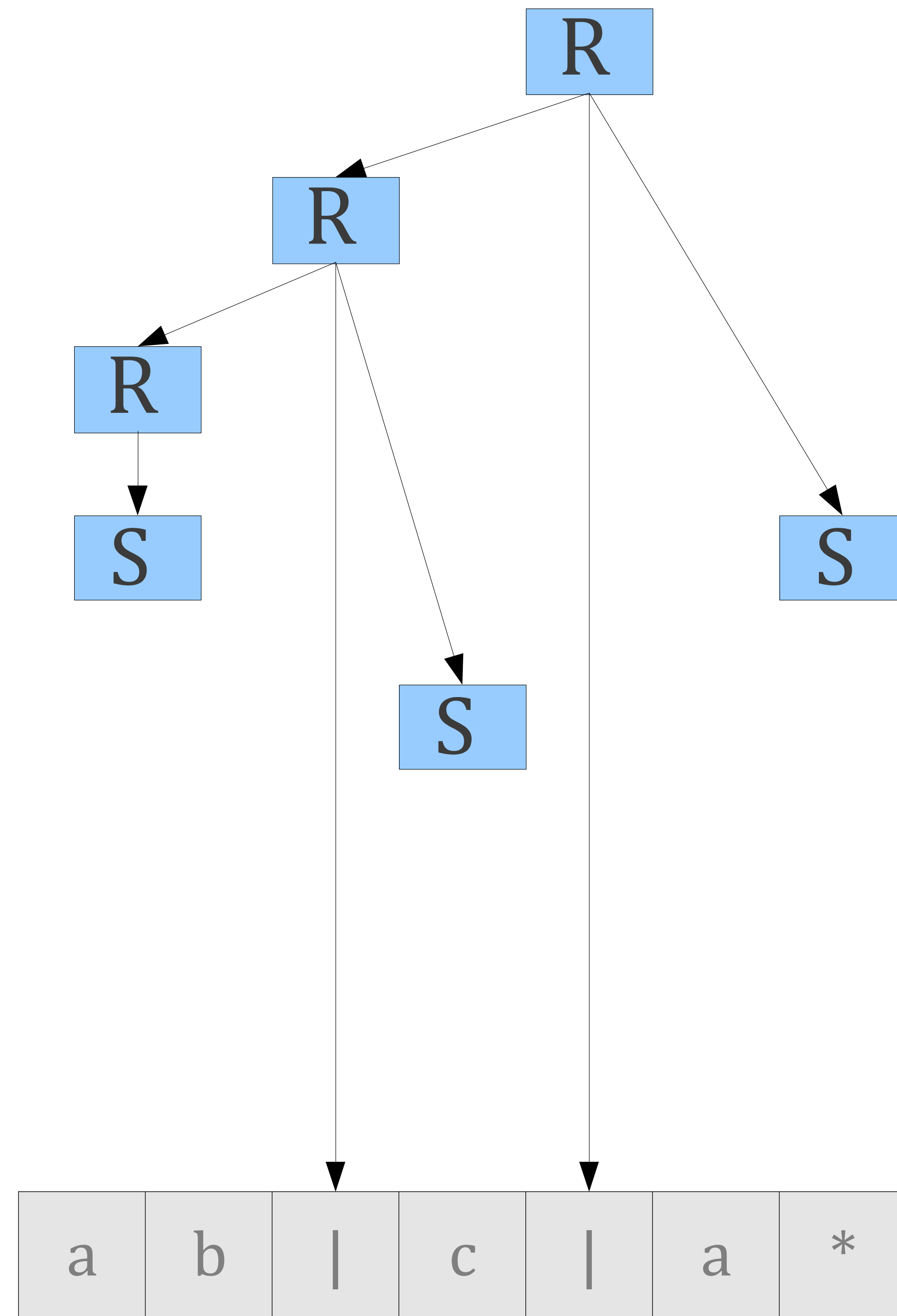
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R "|" S

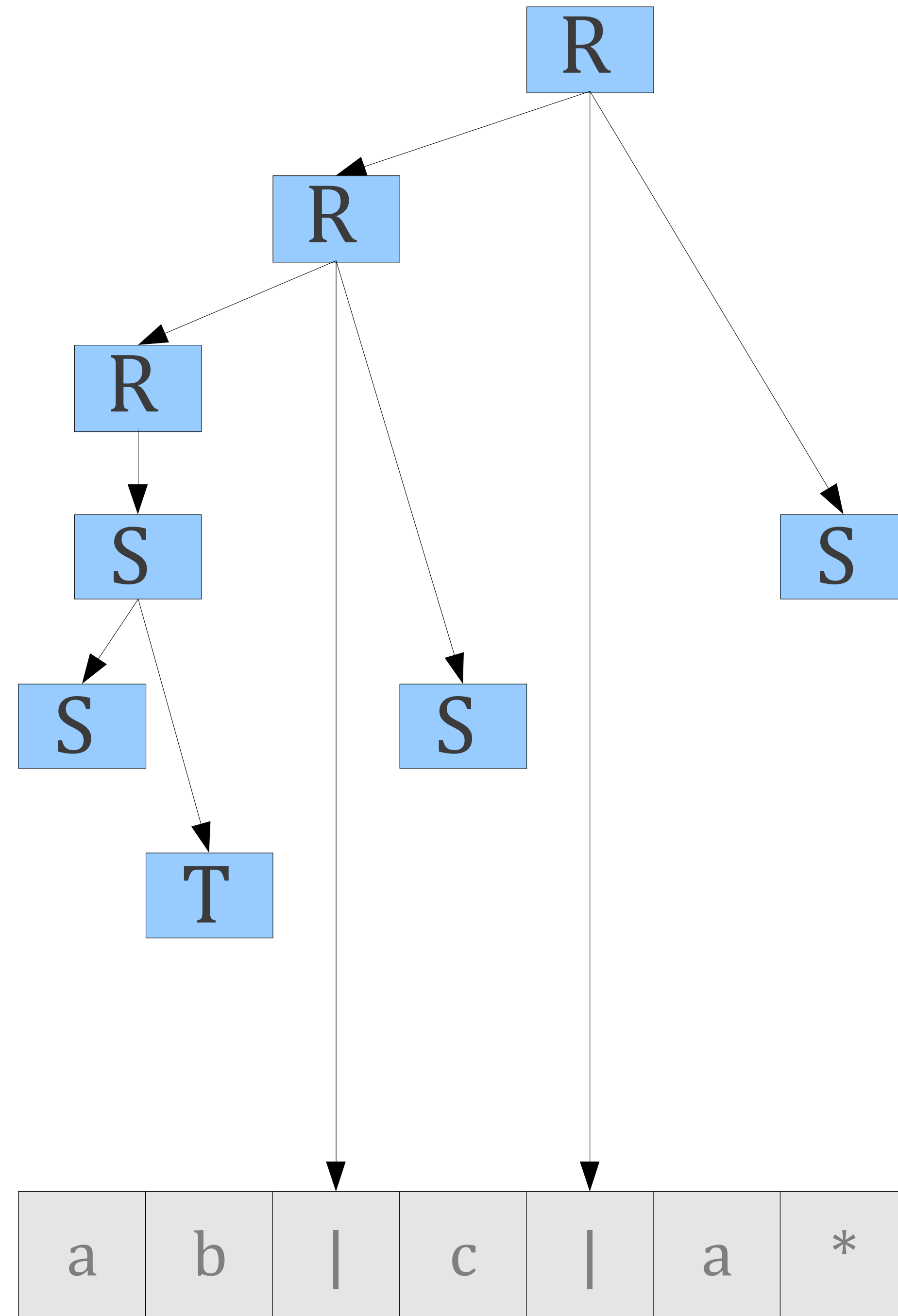
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



GEORGETOWN UNIVERSITY

R → S | R "|" S

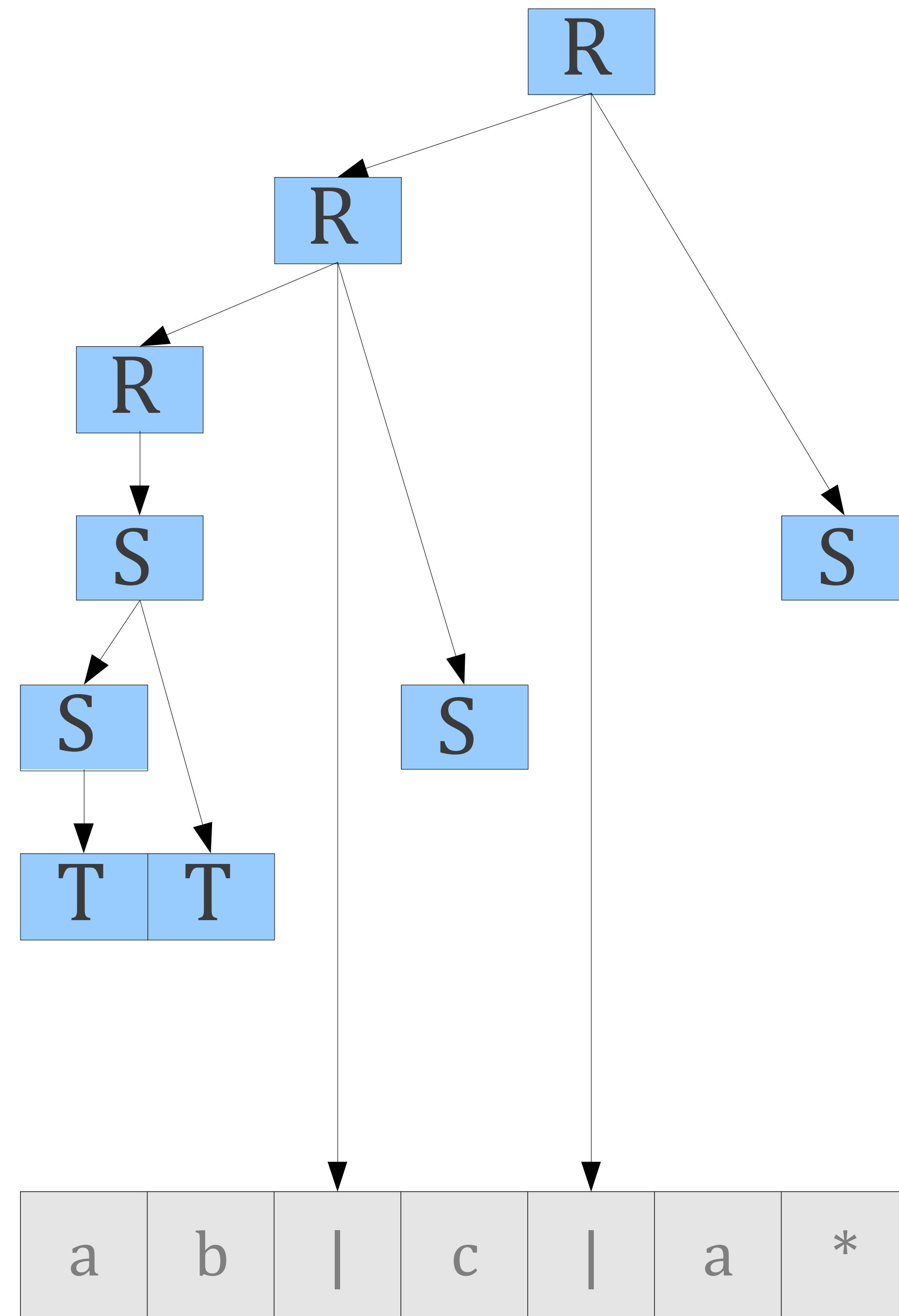
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



GEORGETOWN UNIVERSITY

R → S | R “|” S

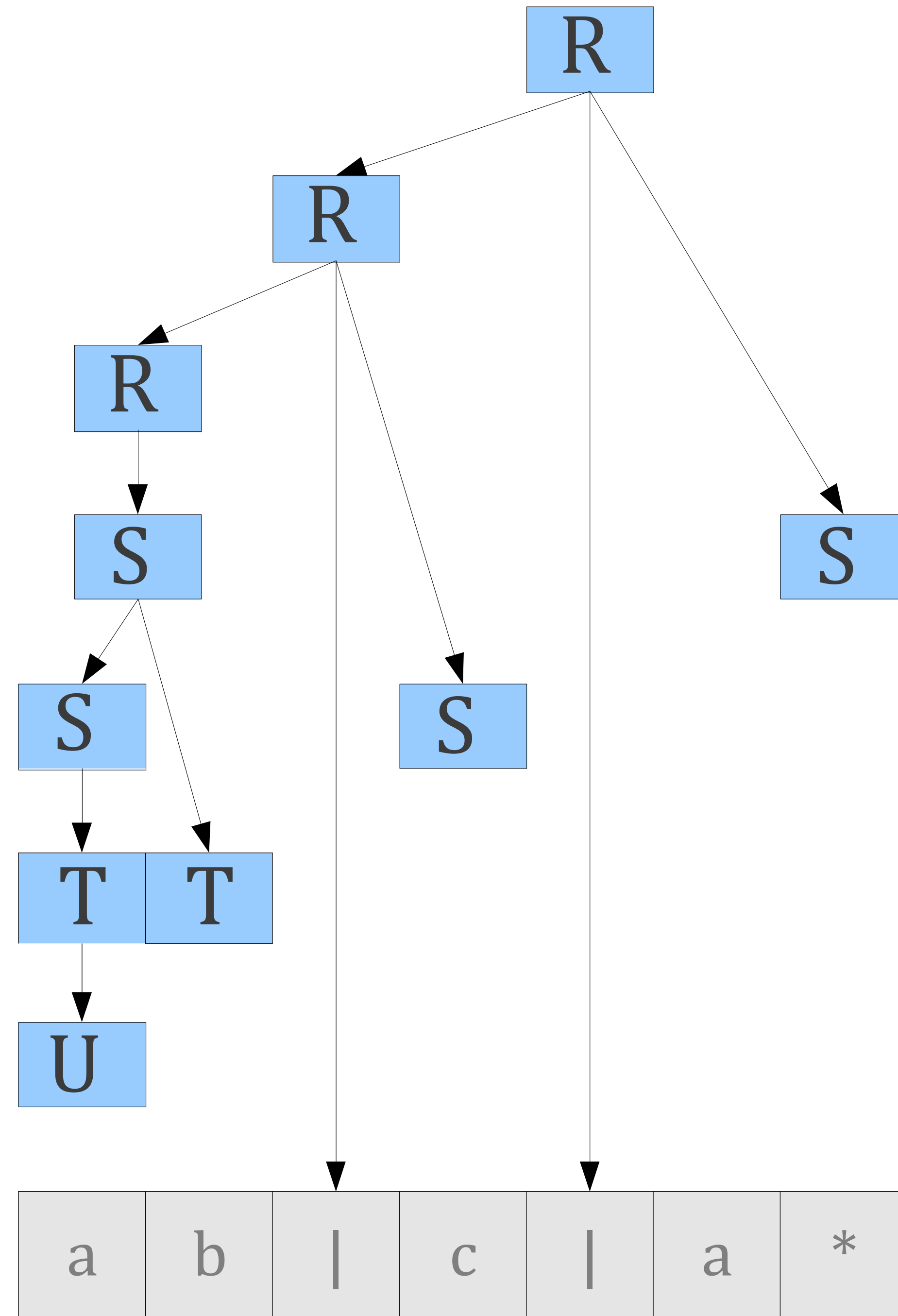
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



GETOWN UNIVERSITY

R → S | R "|" S

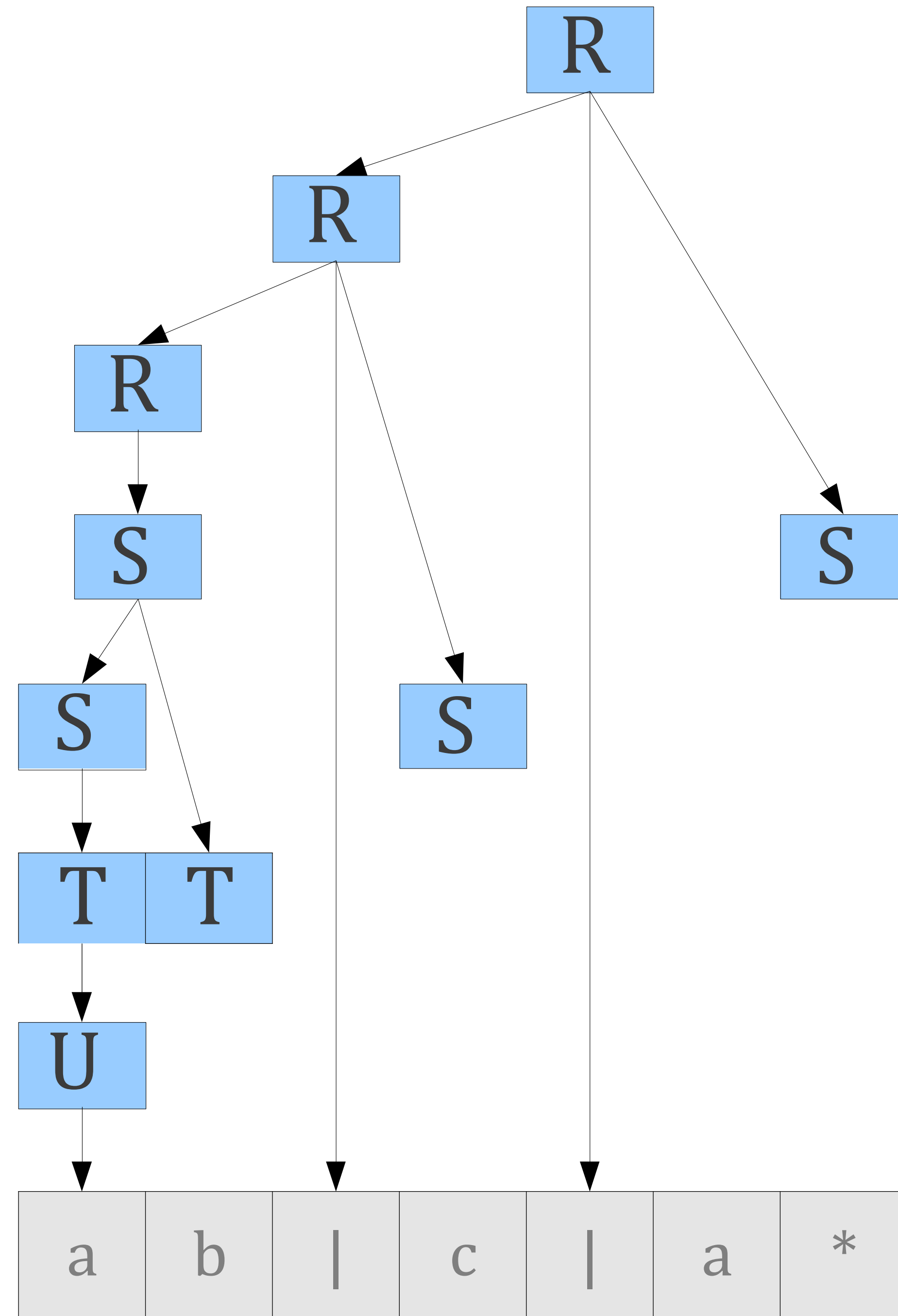
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



R → S | R "|" S

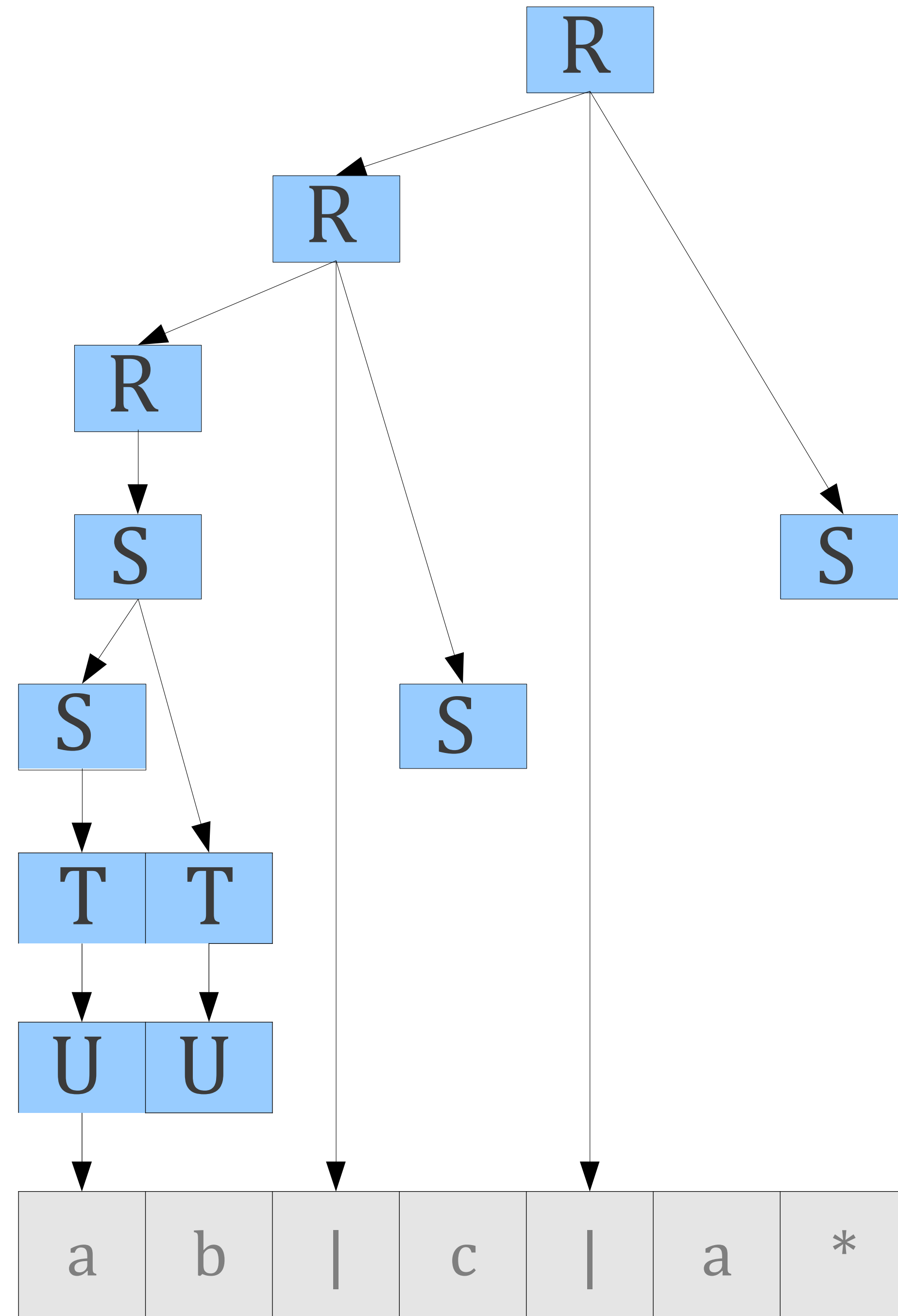
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



R → S | R “|” S

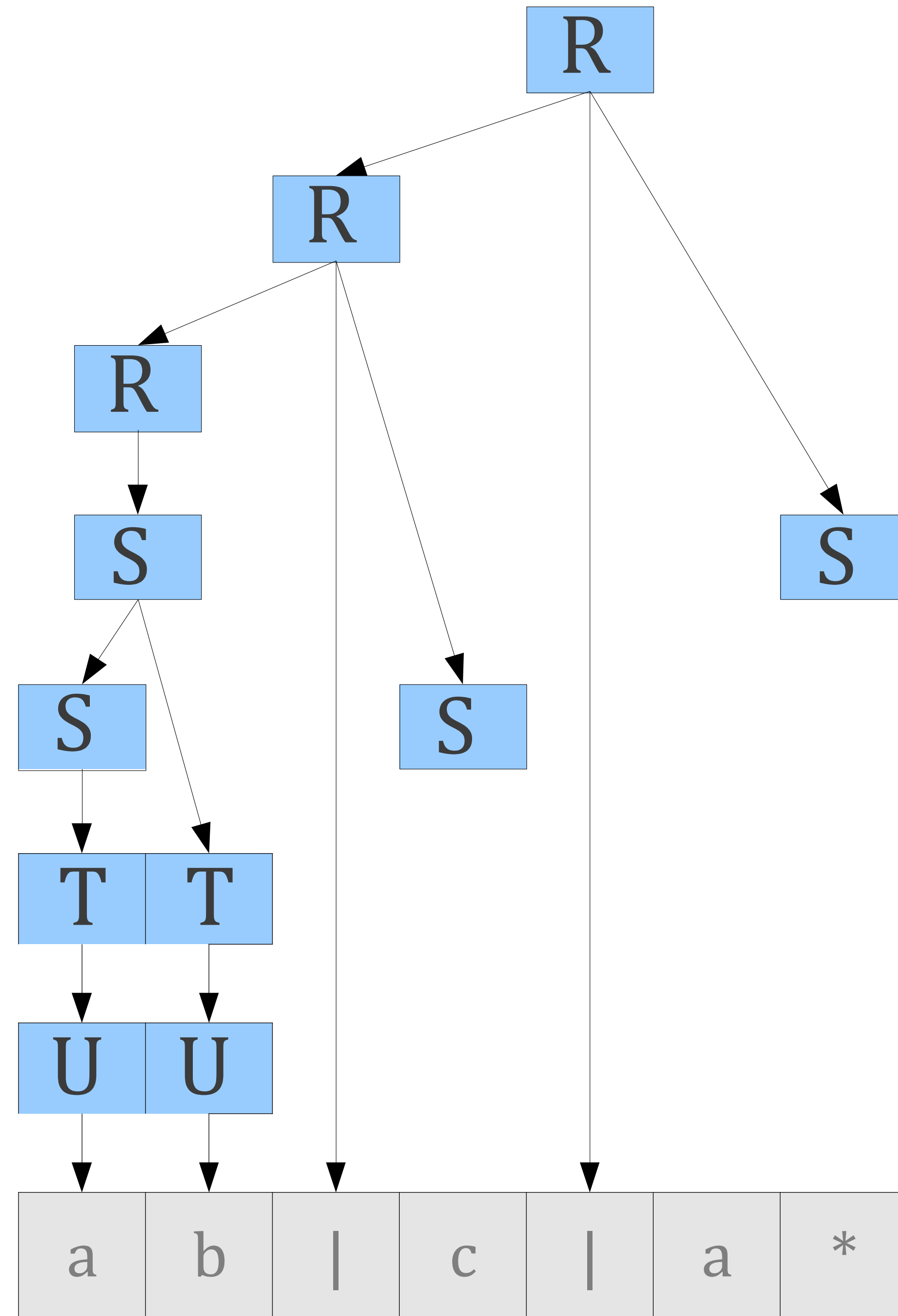
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R "|" S

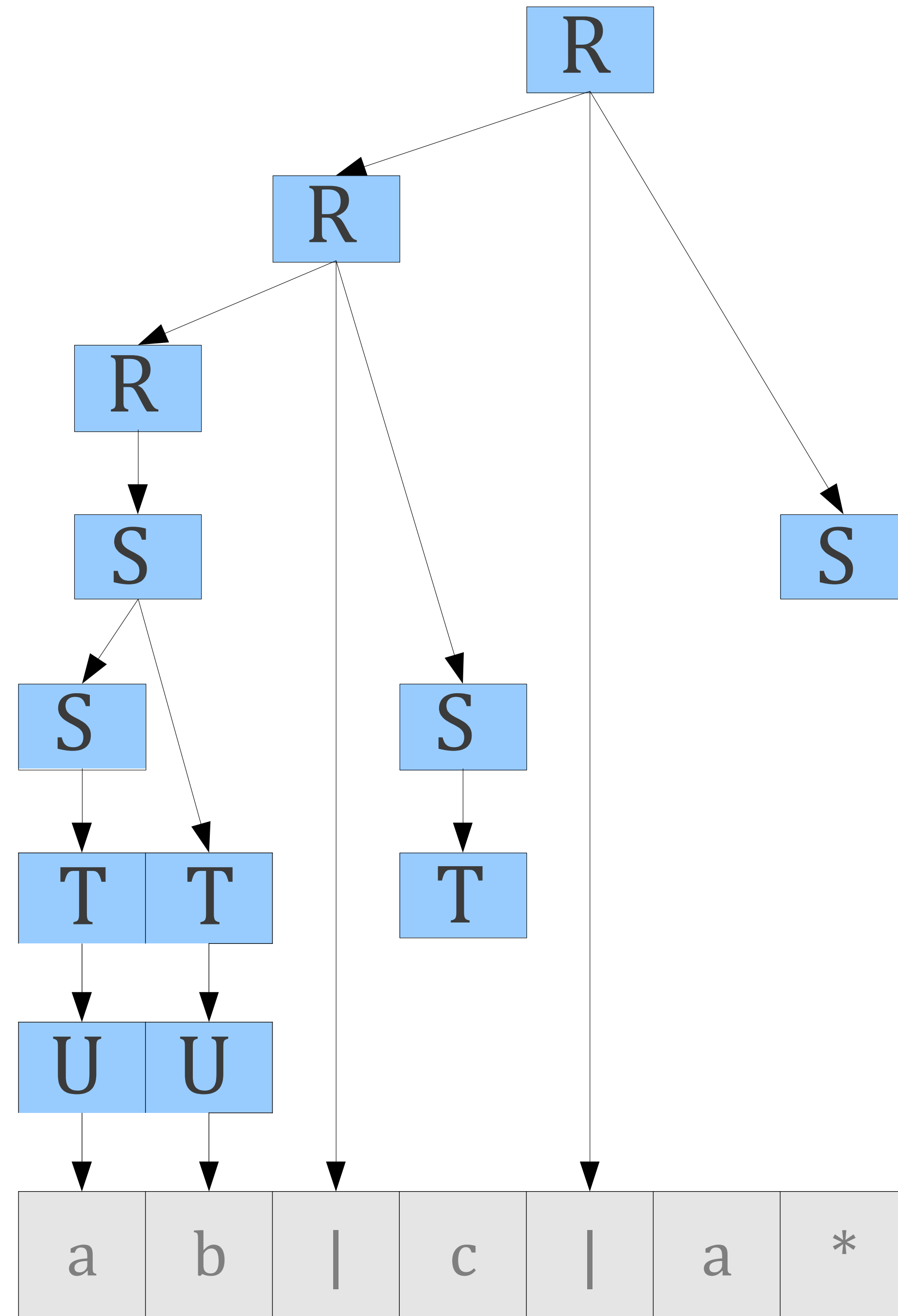
S → T | ST

T → U | T*

U → a | b | c | …

U → "ε"

U → (R)



R → S | R “|” S

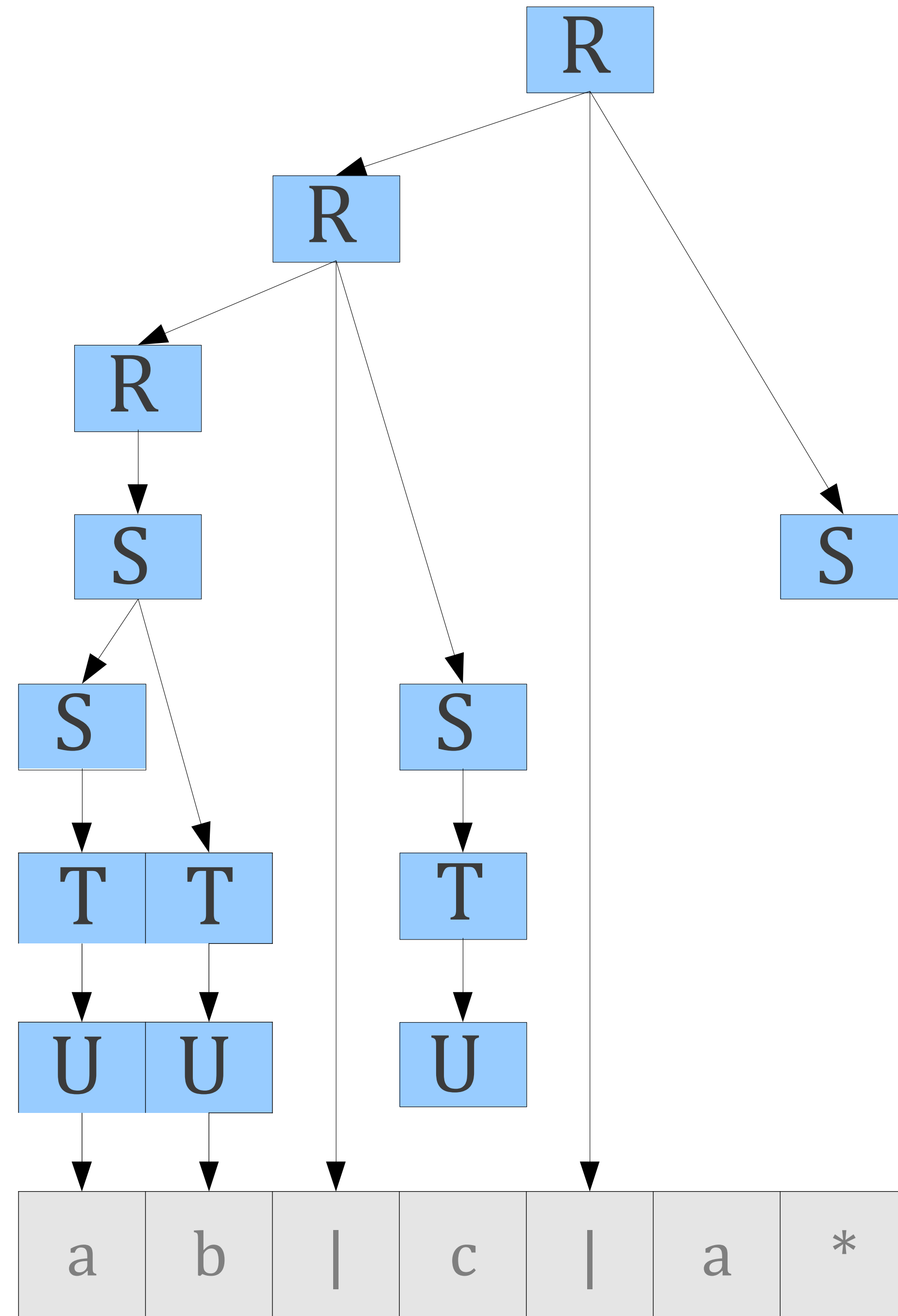
S → T | ST

T → U | T*

U → a | b | c | ...

U → “ε”

U → (R)



GEORGETOWN UNIVERSITY

R → S | R “|” S

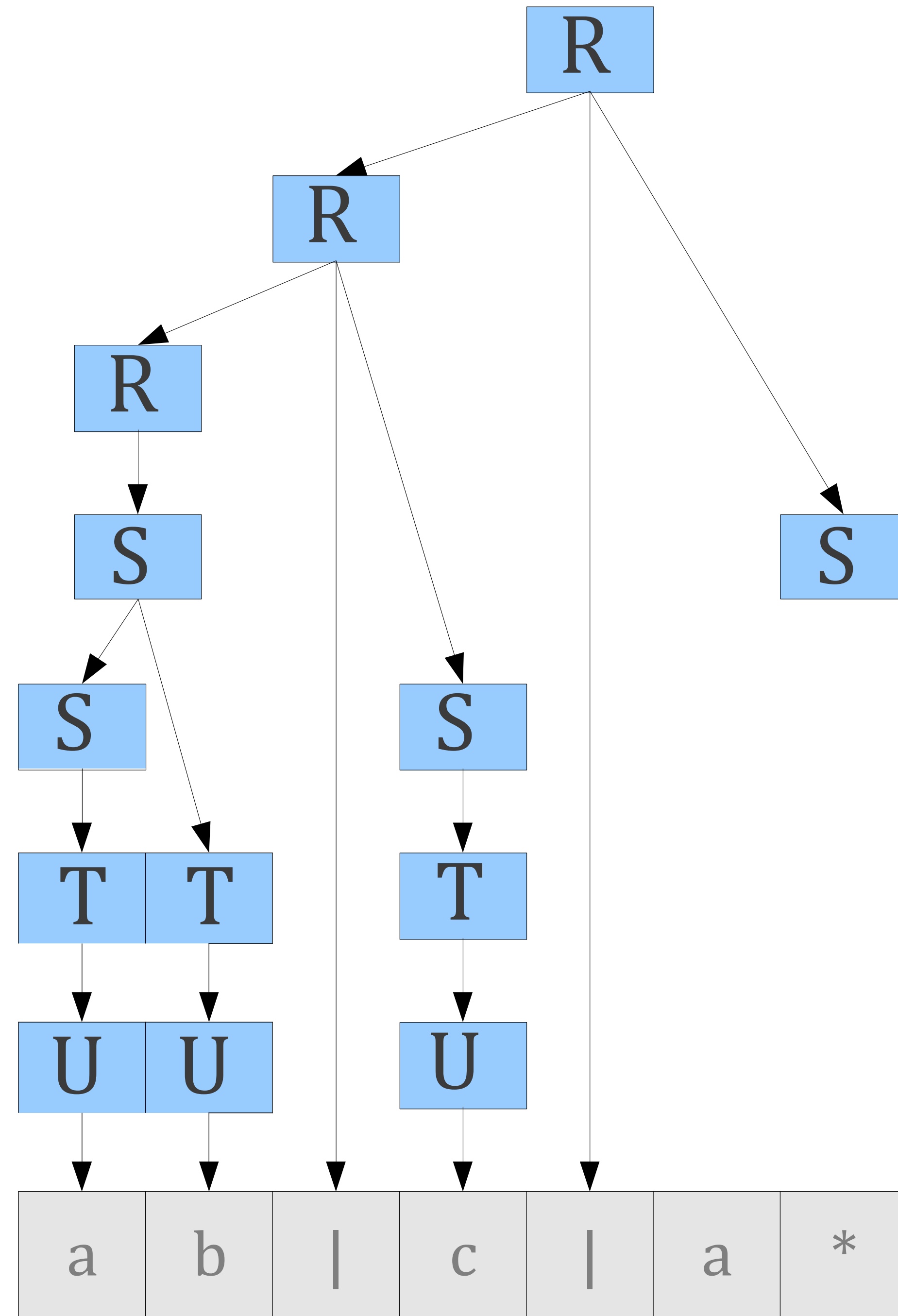
S → T | ST

T → U | T*

U → a | b | c | ...

U → “ε”

U → (R)



GETOWN UNIVERSITY

R → S | R "|" S

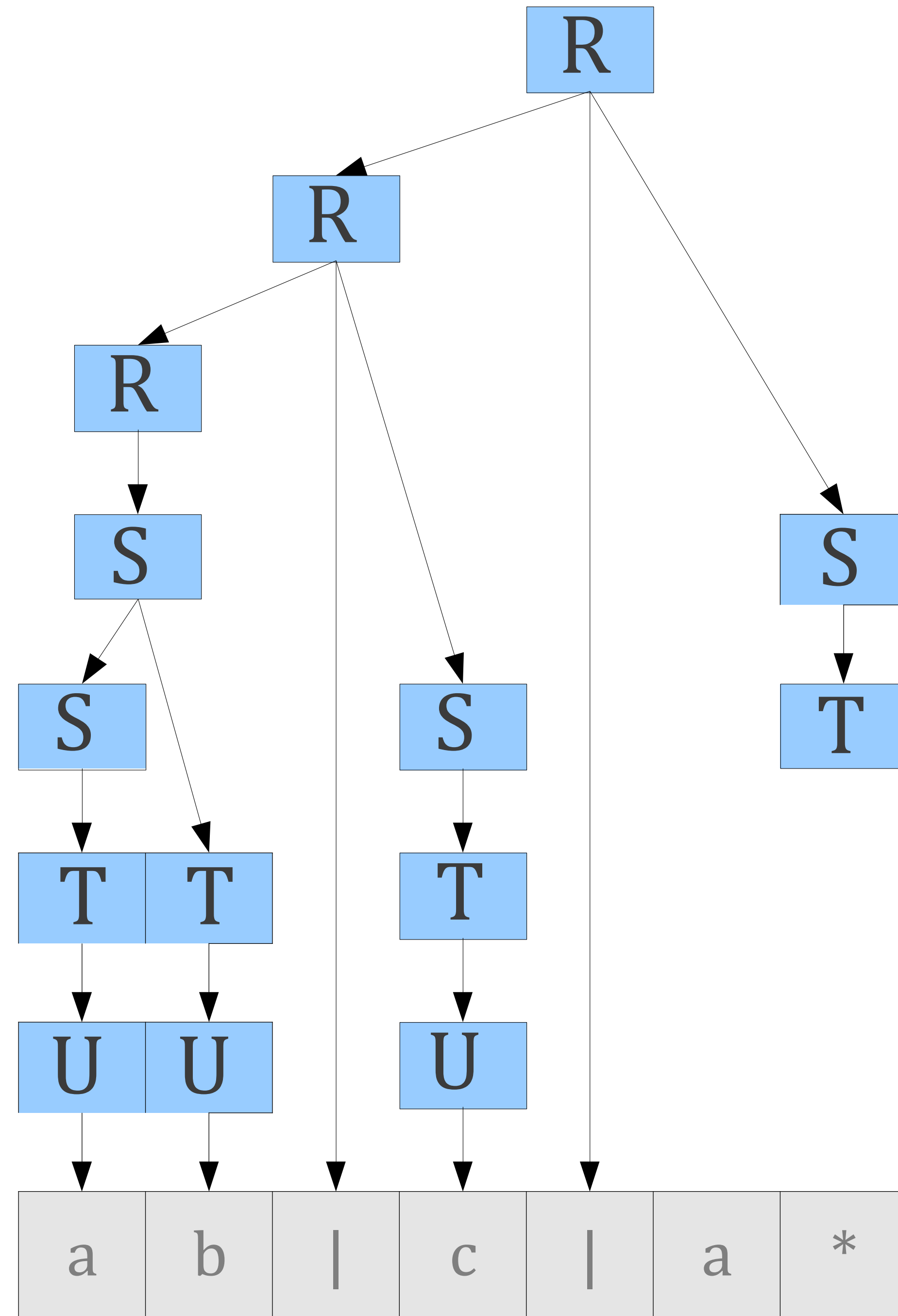
S → T | ST

T → U | T*

U → a | b | c | …

U → "ε"

U → (R)



R → S | R "|" S

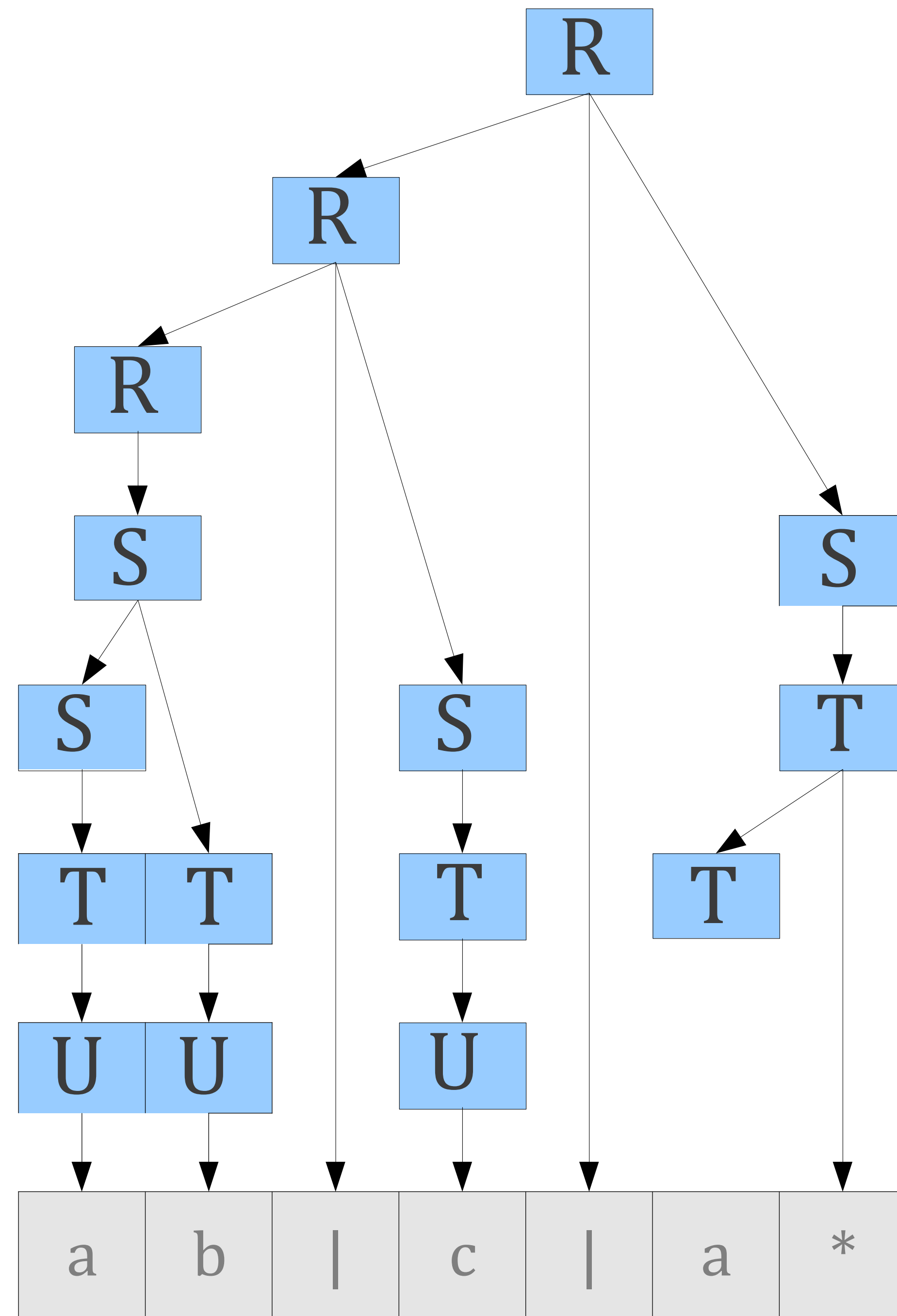
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



R → S | R "|" S

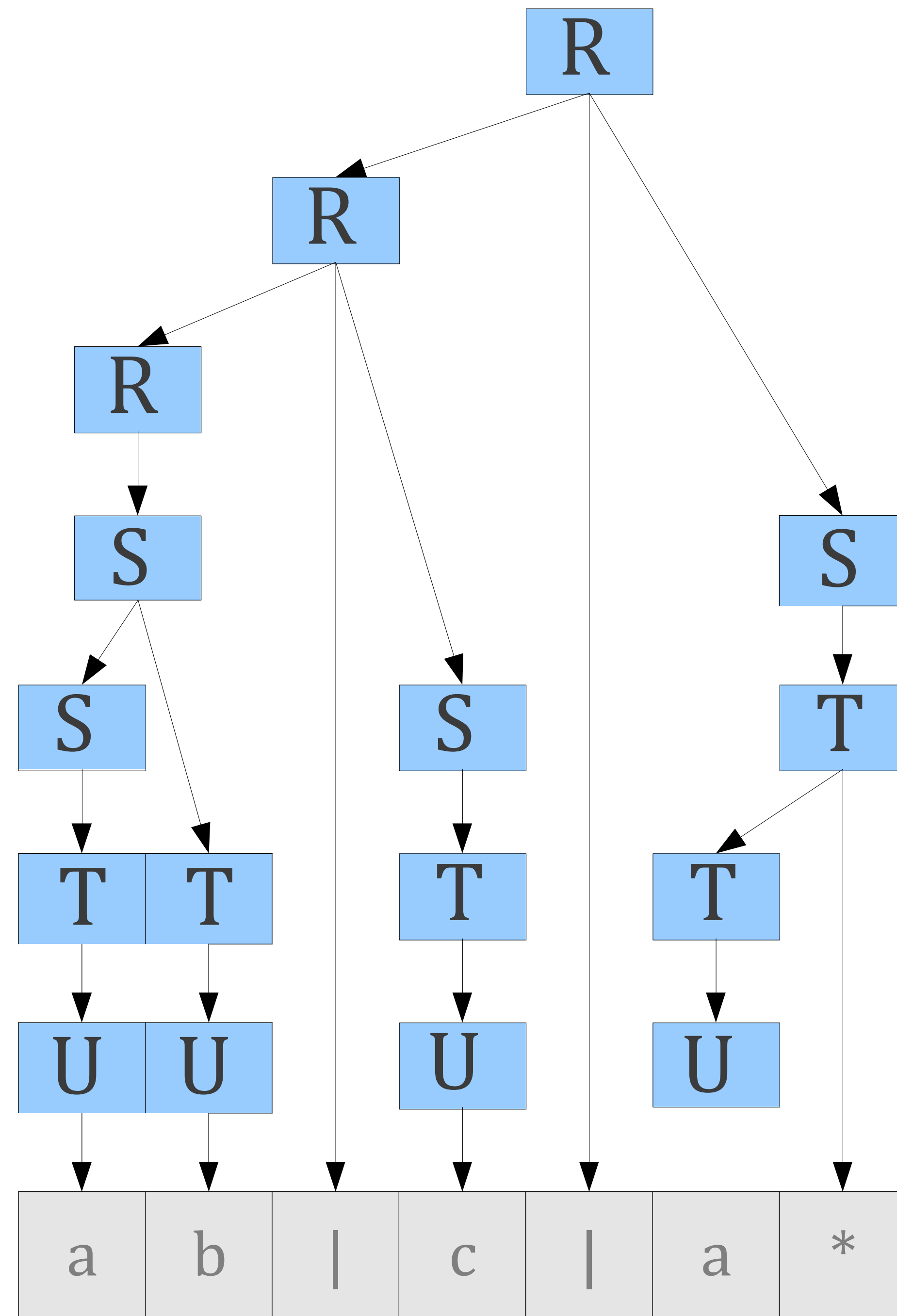
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



R → S | R "|" S

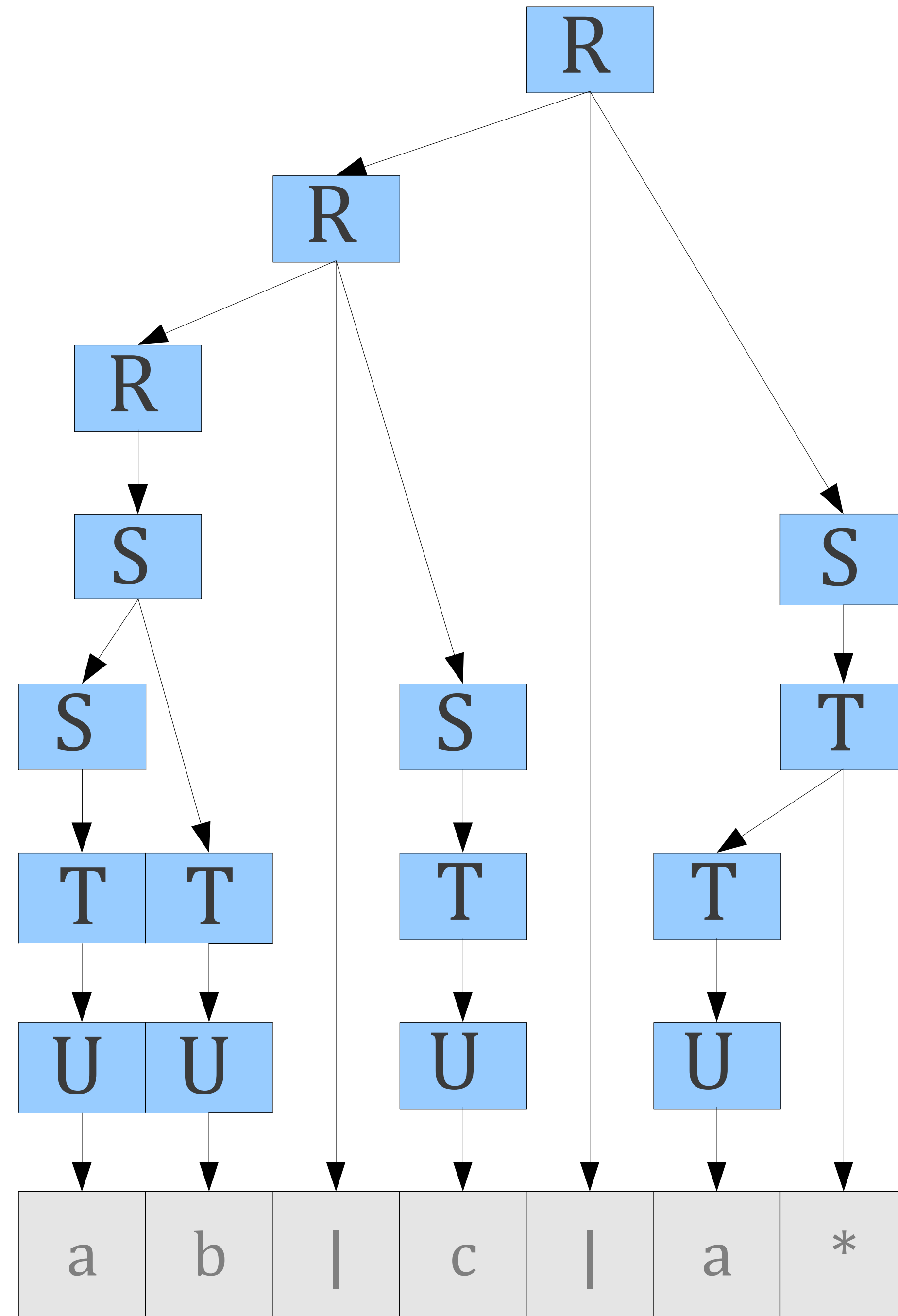
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



Precedence Declarations

- If we leave the world of pure CFGs, we can often resolve ambiguities through precedence declarations.
  - e.g. multiplication has higher precedence than addition, but lower precedence than exponentiation.
- Allows for unambiguous parsing of ambiguous grammars.

The Structure of a Parse Tree

R → S | R "|" S

S → T | ST

T → U | T*

U → a | b | …

U → "ε"

U → (R)

GEORGETOWN UNIVERSITY

The Structure of a Parse Tree

R → S | R "|" S

S → T | ST

T → U | T*

U → a | b | …

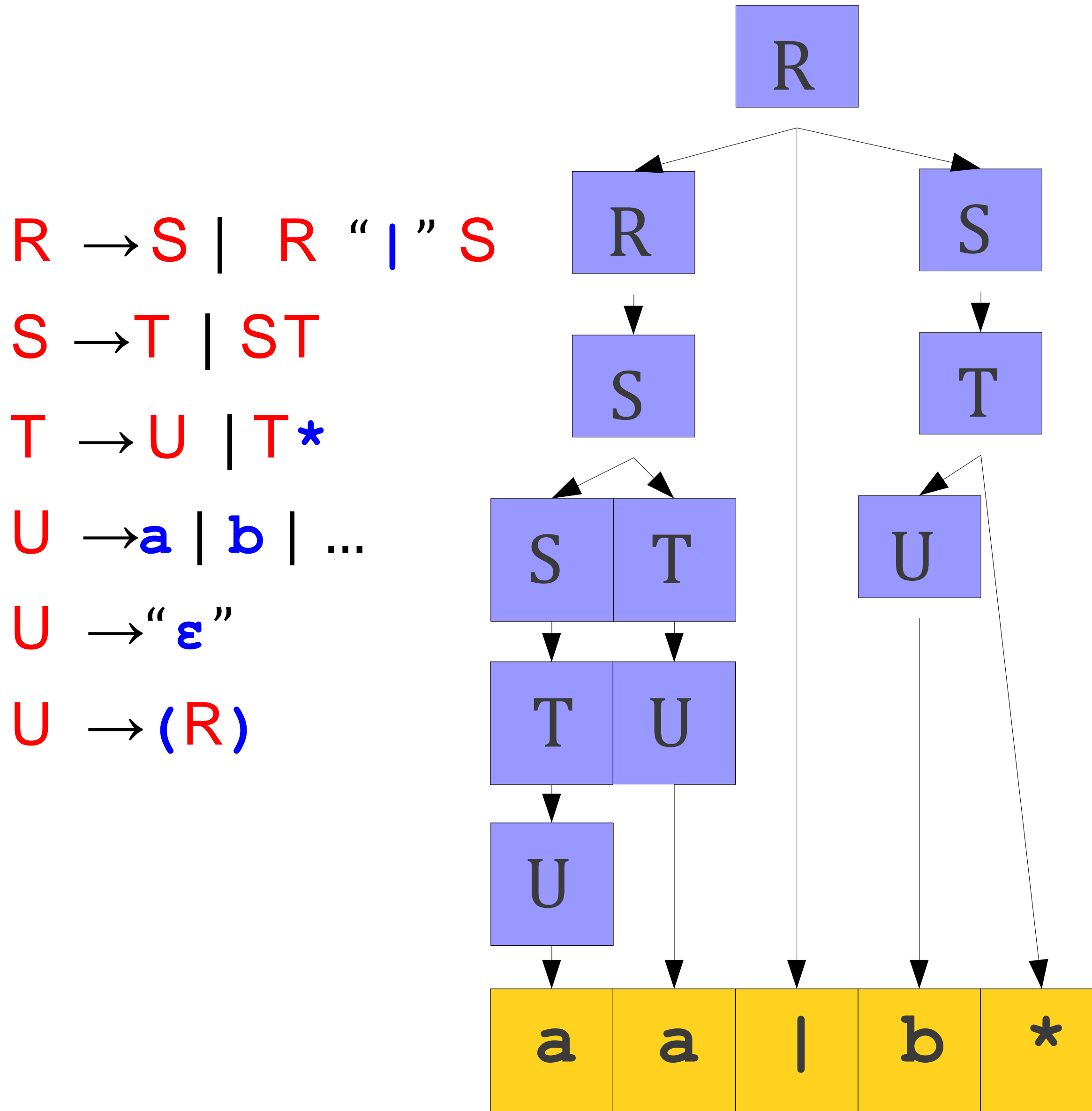
U → "ε"

U → (R)

a	a	|	b	*

GEORGETOWN UNIVERSITY

The Structure of a Parse Tree



GEORGETOWN UNIVERSITY

The Structure of a Parse Tree

R → S | R “|” S

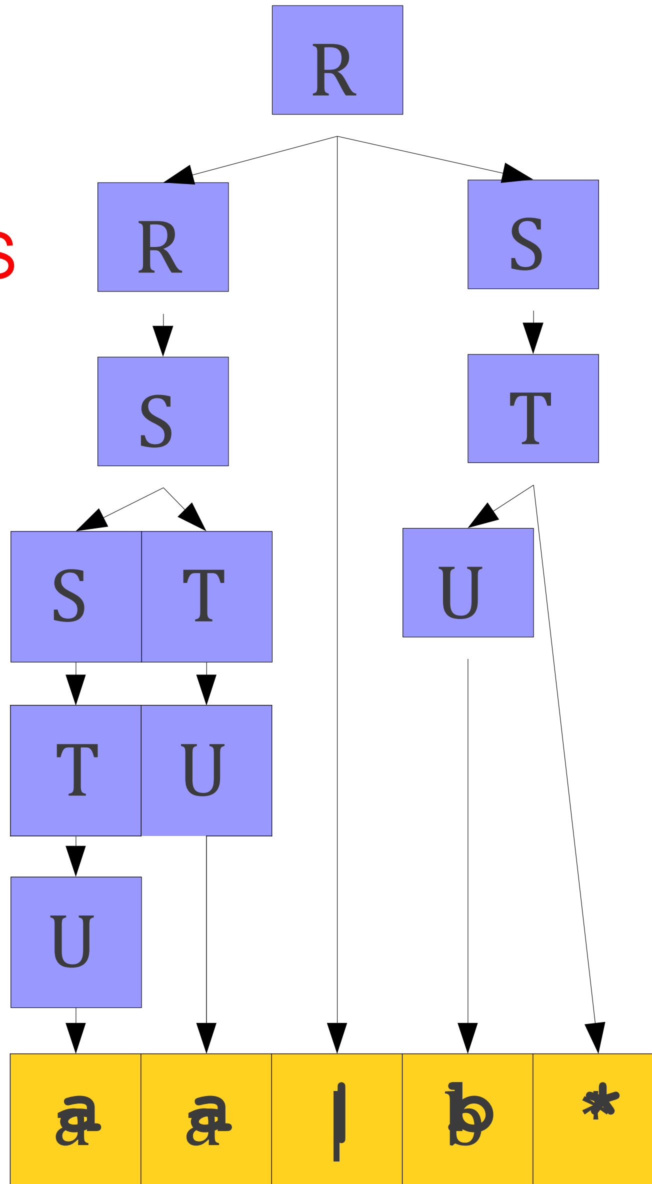
S → T | ST

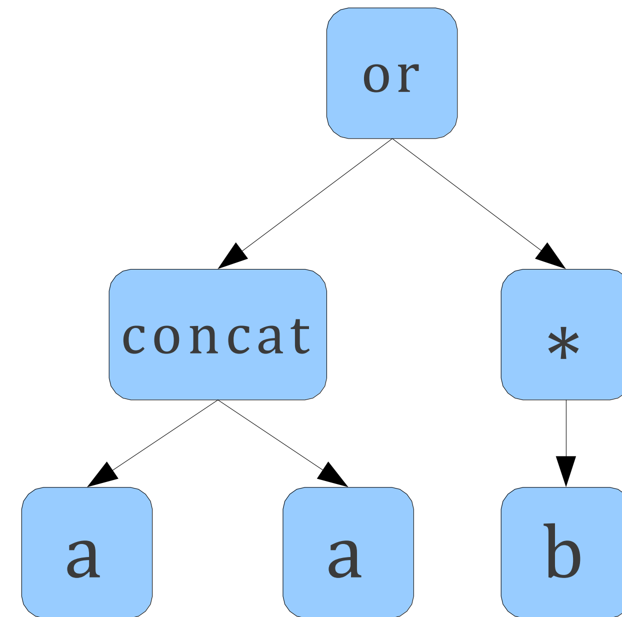
T → U | T*

U → a | b | …

U → “ε”

U → (R)





GEORGETOWN UNIVERSITY

R → S | R "|" S

S → T | ST

T → U | T*

U → a | b | …

U → "ε"

U → (R)

a	(	b	|	c	)

GEORGETOWN UNIVERSITY

R → S | R "|" S

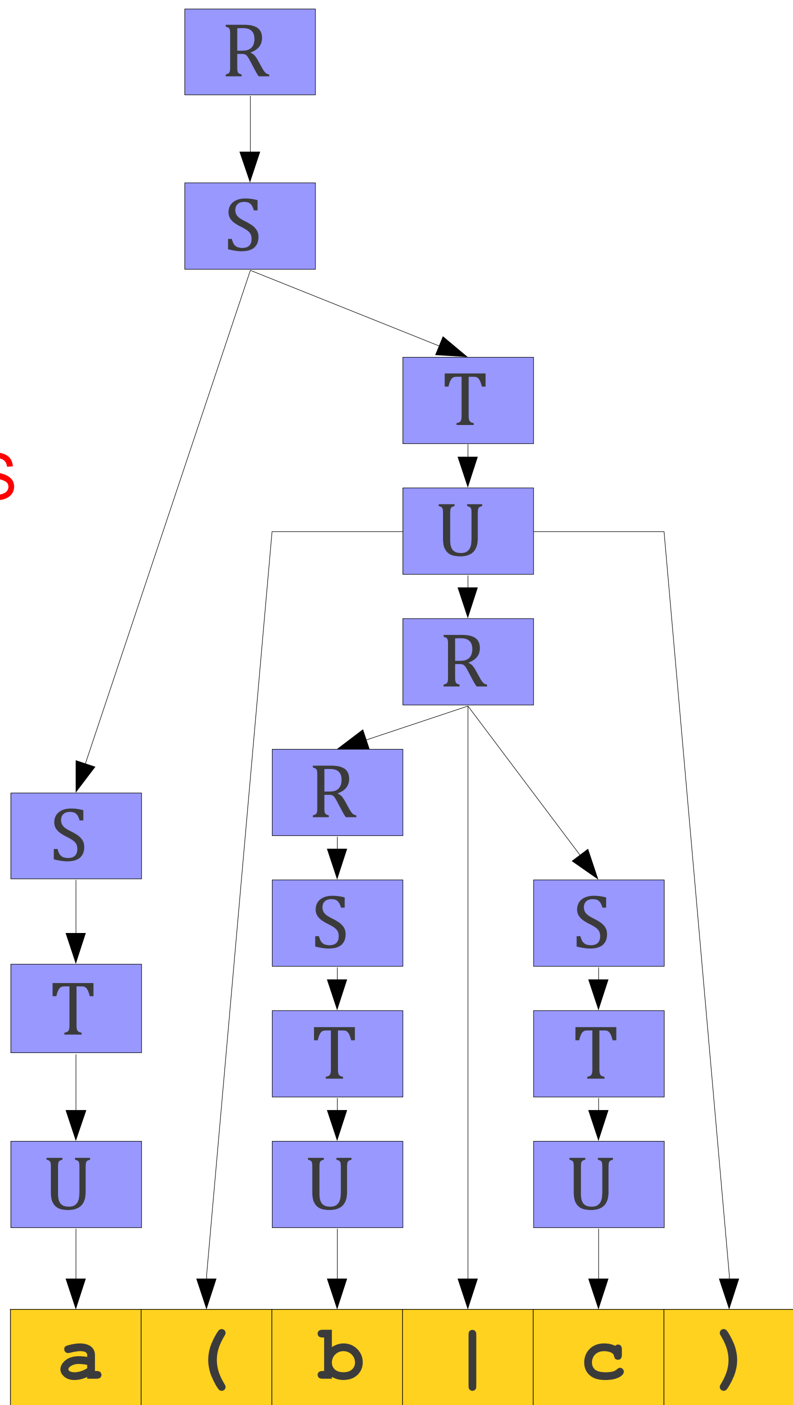
S → T | ST

T → U | T*

U → a | b | …

U → "ε"

U → (R)



GEORGETOWN UNIVERSITY

R → S | R "|" S

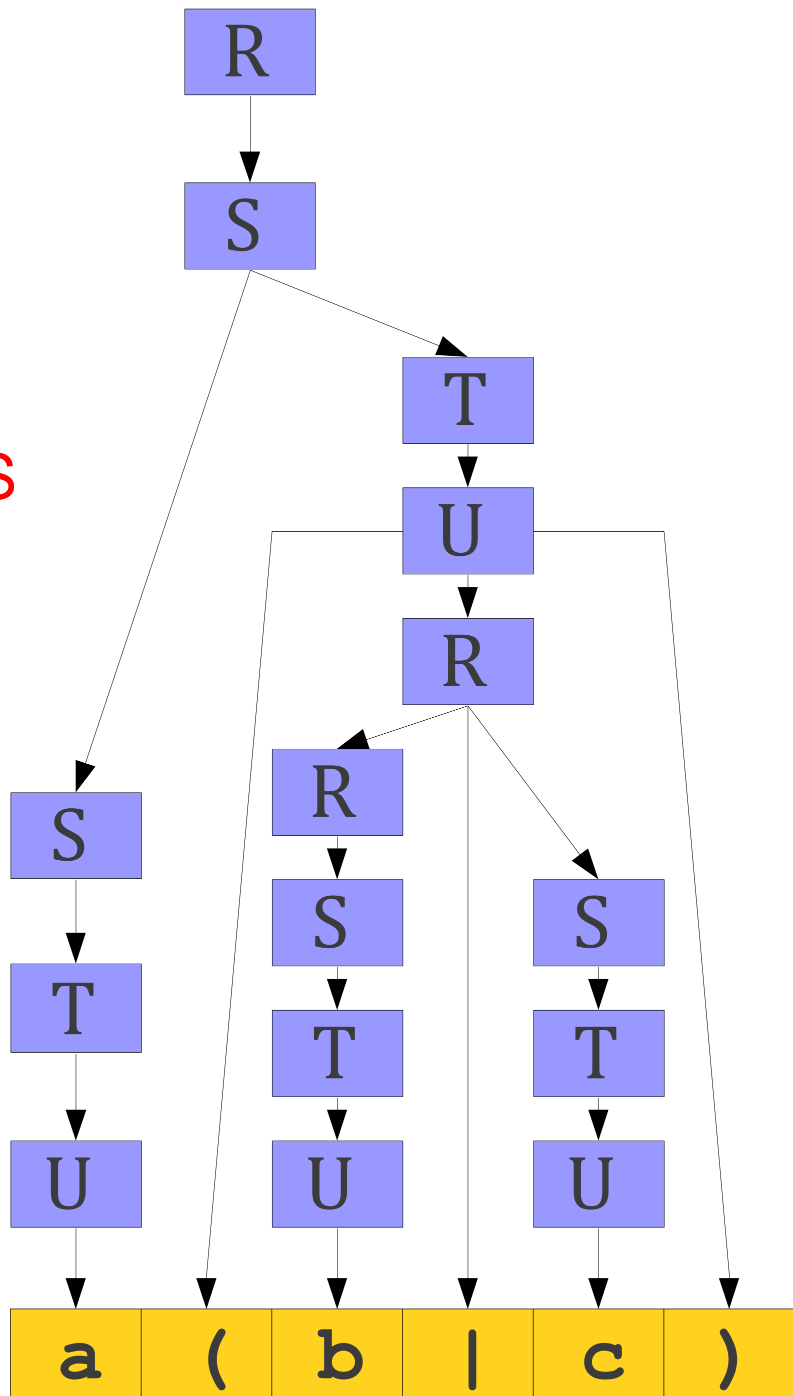
S → T | ST

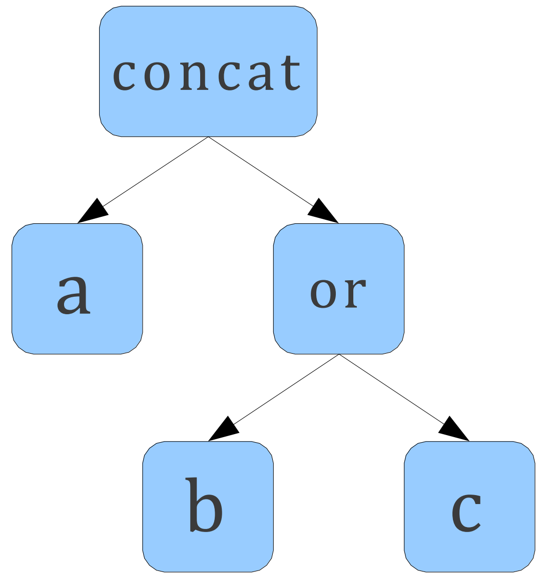
T → U | T*

U → a | b | …

U → "ε"

U → (R)





GEORGETOWN UNIVERSITY

Abstract Syntax Trees (ASTs)

- A parse tree is a concrete syntax tree; it shows exactly how the text was derived.
- A more useful structure is an abstract syntax tree, which retains only the essential structure of the input.
  - A *bridge* between Syntax and Semantics!

GEORGETOWN UNIVERSITY

How to build an AST?

- Typically done through semantic actions.
- Associate a piece of code to execute with each production.
- As the input is parsed, execute this code to build the AST.
  - Exact order of code execution depends on the parsing method used.
- This is called a syntax-directed translation.

GEORGETOWN UNIVERSITY

Simple Semantic Actions

E → T **+** E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → **int**	T.val = int.val
T → **int** ***** T	T.val = int.val * T.val
T → **(**E**)**	T.val = E.val

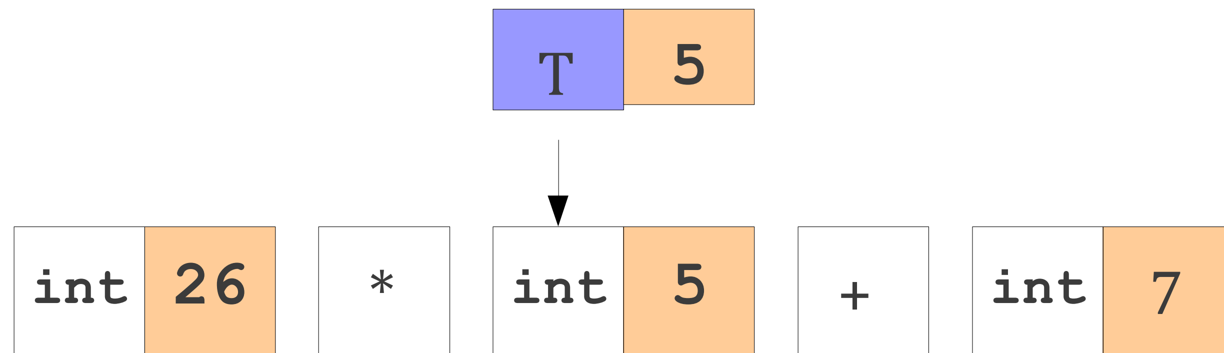
GEORGETOWN UNIVERSITY

Simple Semantic Actions

$E \rightarrow T + E$	E_1.val = T.val + E_2.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow$ **int**	T.val = int.val
$T \rightarrow$ **int** * T	T.val = int.val * T.val
$T \rightarrow$ (E)	T.val = E.val

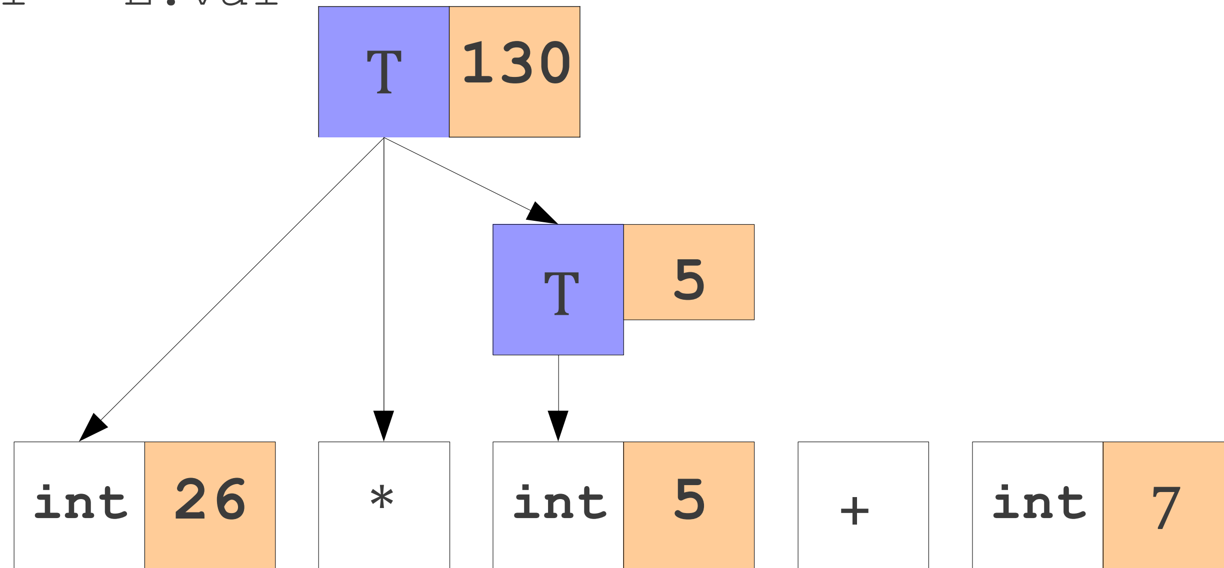
Simple Semantic Actions

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → **int**	T.val = int.val
T → **int** * T	T.val = int.val * T.val
T → (E)	T.val = E.val



Simple Semantic Actions

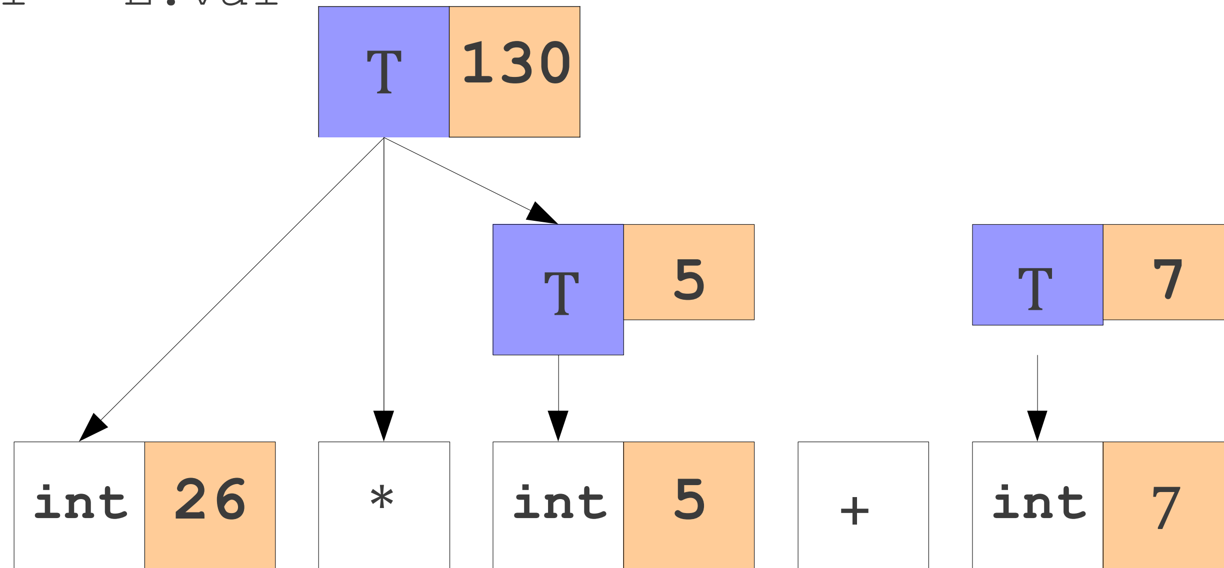
E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → **int**	T.val = int.val
T → **int** * T	T.val = int.val * T.val
T → (E)	T.val = E.val



GEORGETOWN UNIVERSITY

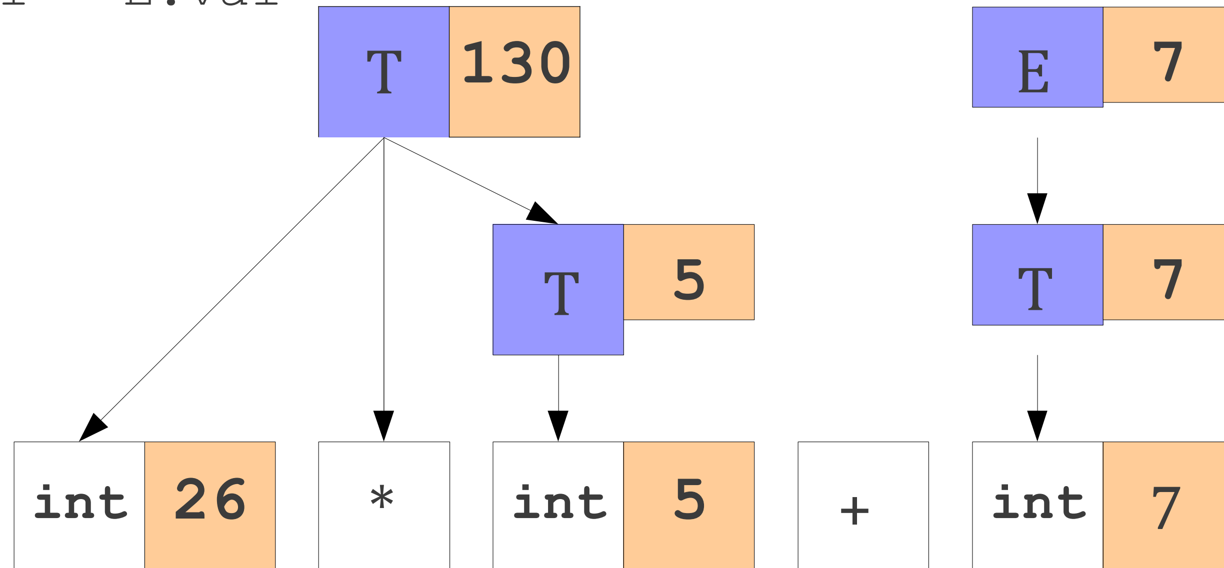
Simple Semantic Actions

Production	Semantic action
E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → **int**	T.val = int.val
T → **int** * T	T.val = int.val * T.val
T → (E)	T.val = E.val



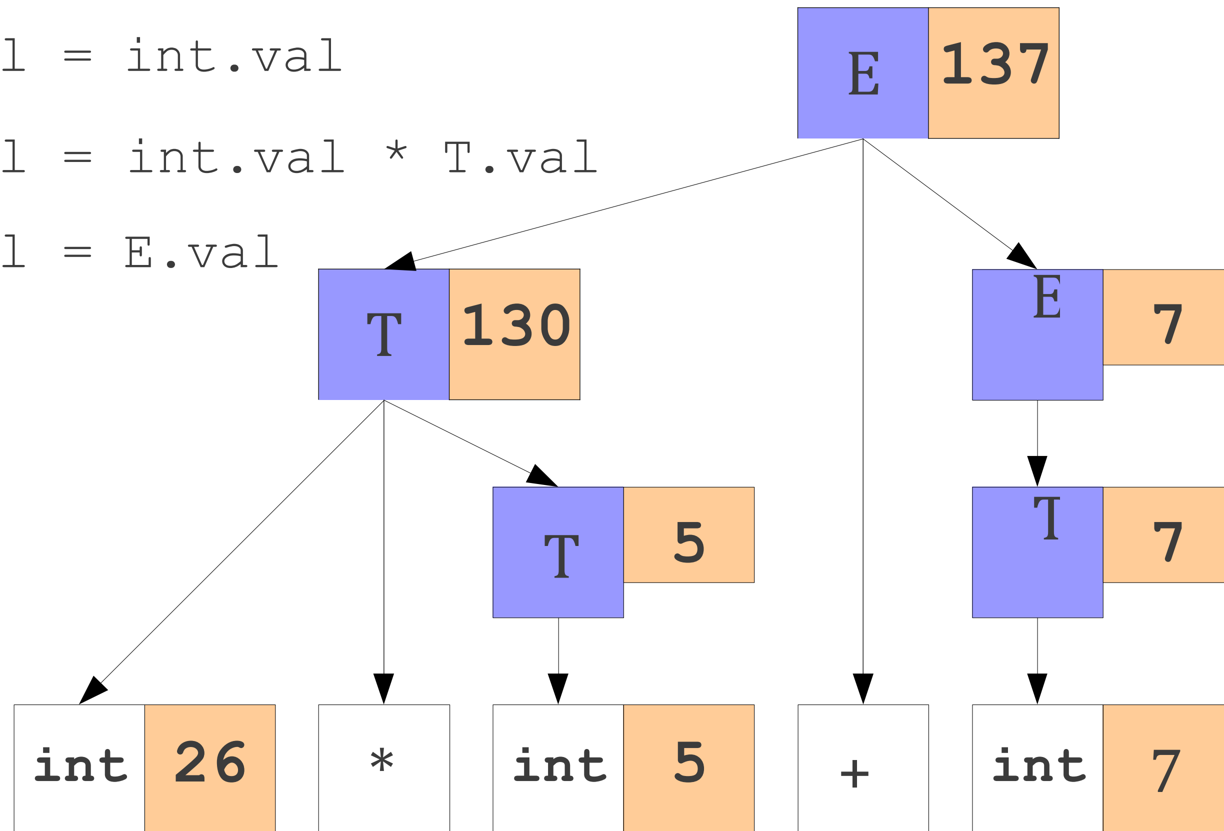
Simple Semantic Actions

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → int	T.val = int.val
T → int * T	T.val = int.val * T.val
T → (E)	T.val = E.val



Simple Semantic Actions

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → int	T.val = int.val
T → int * T	T.val = int.val * T.val
T → (E)	T.val = E.val



Semantic Actions to Build ASTs

Production	Semantic action
R → S	`R.ast = S.ast;`
R → R "|" S	R_1`.ast = new Or(`R_2`.ast, S.ast);`
S → T	`S.ast = T.ast;`
S → ST	S_1`.ast = new Concat(`S_2`.ast, T.ast);`
T → U	`T.ast = U.ast;`
T → T*	T_1`.ast = new Star(`T_2`.ast);`
U → a	`U.ast = new SingleChar('a');`
U → "ε"	`U.ast = new Epsilon();`
U → (R)	`U.ast = R.ast;`

Summary

- Syntax analysis (parsing) extracts the structure from the tokens produced by the scanner.
- Languages are usually specified by context-free grammars (CFGs).
- A parse tree shows how a string can be derived from a grammar.
- A grammar is ambiguous if it can derive the same string multiple ways.
- There is no algorithm for eliminating ambiguity; it must be done by hand.
- Abstract syntax trees (ASTs) contain an abstract representation of a program's syntax.
- Semantic actions associated with productions can be used to build ASTs.

GEORGETOWN UNIVERSITY

Appendix

GEORGETOWN UNIVERSITY