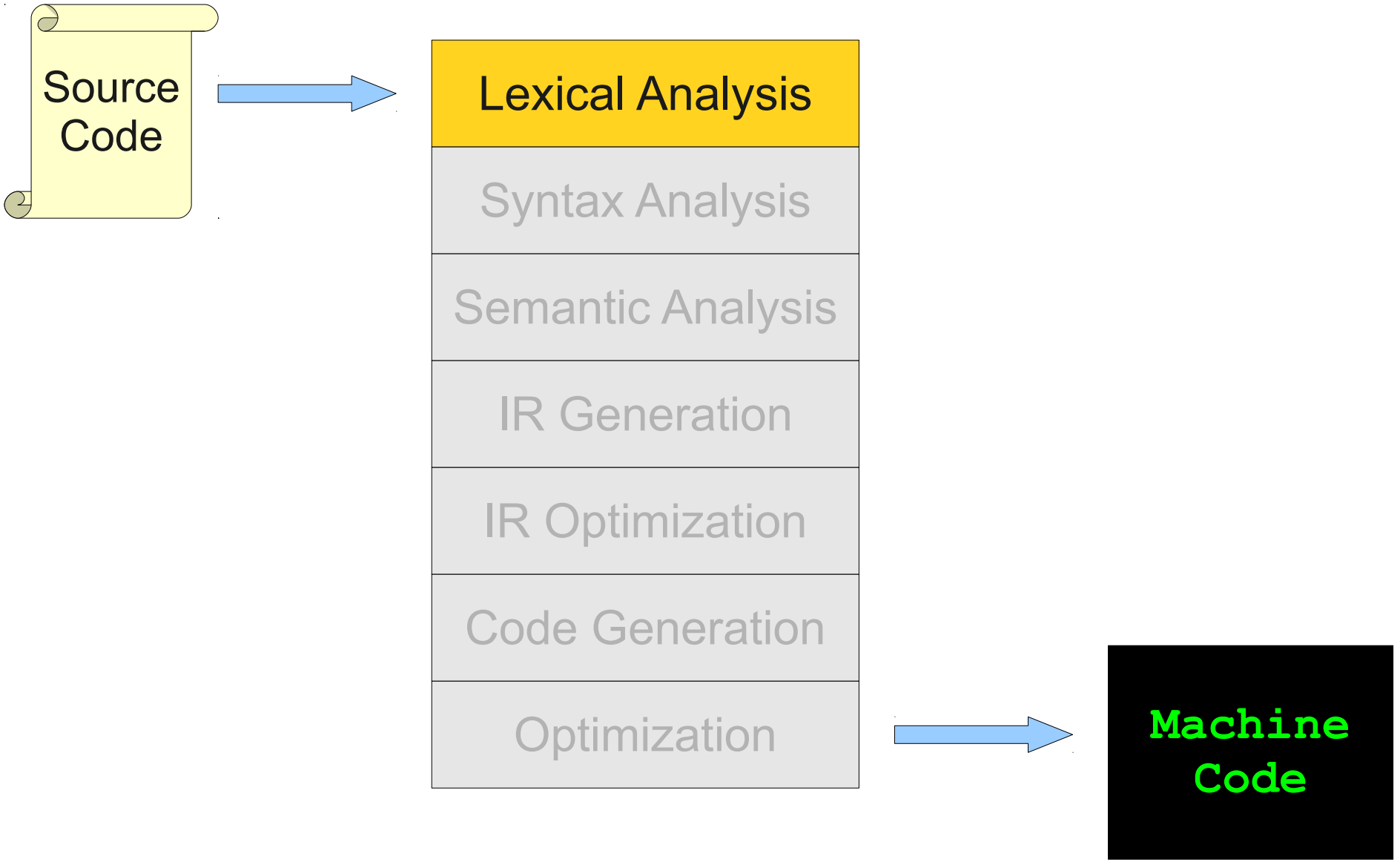
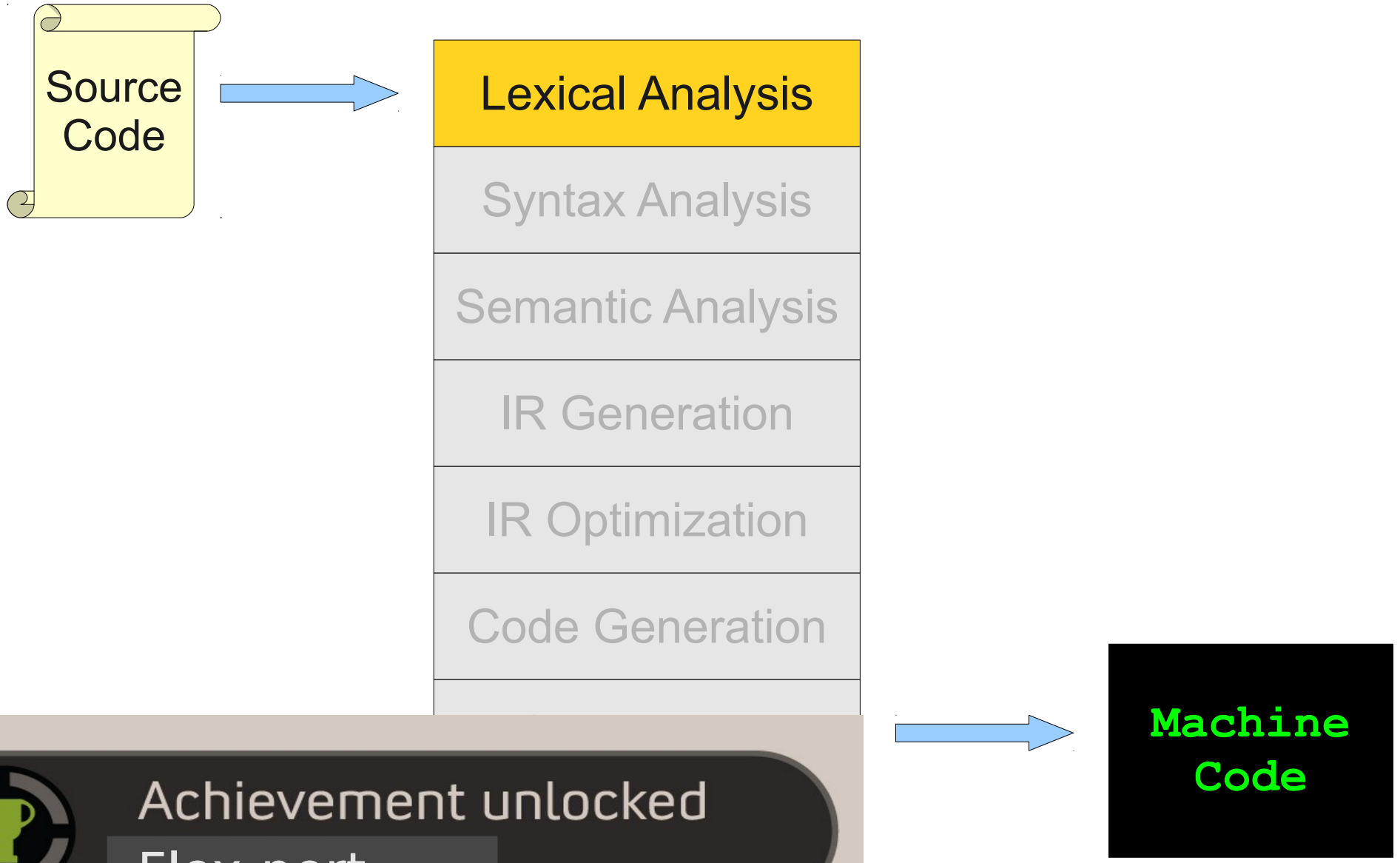
Syntax Analysis

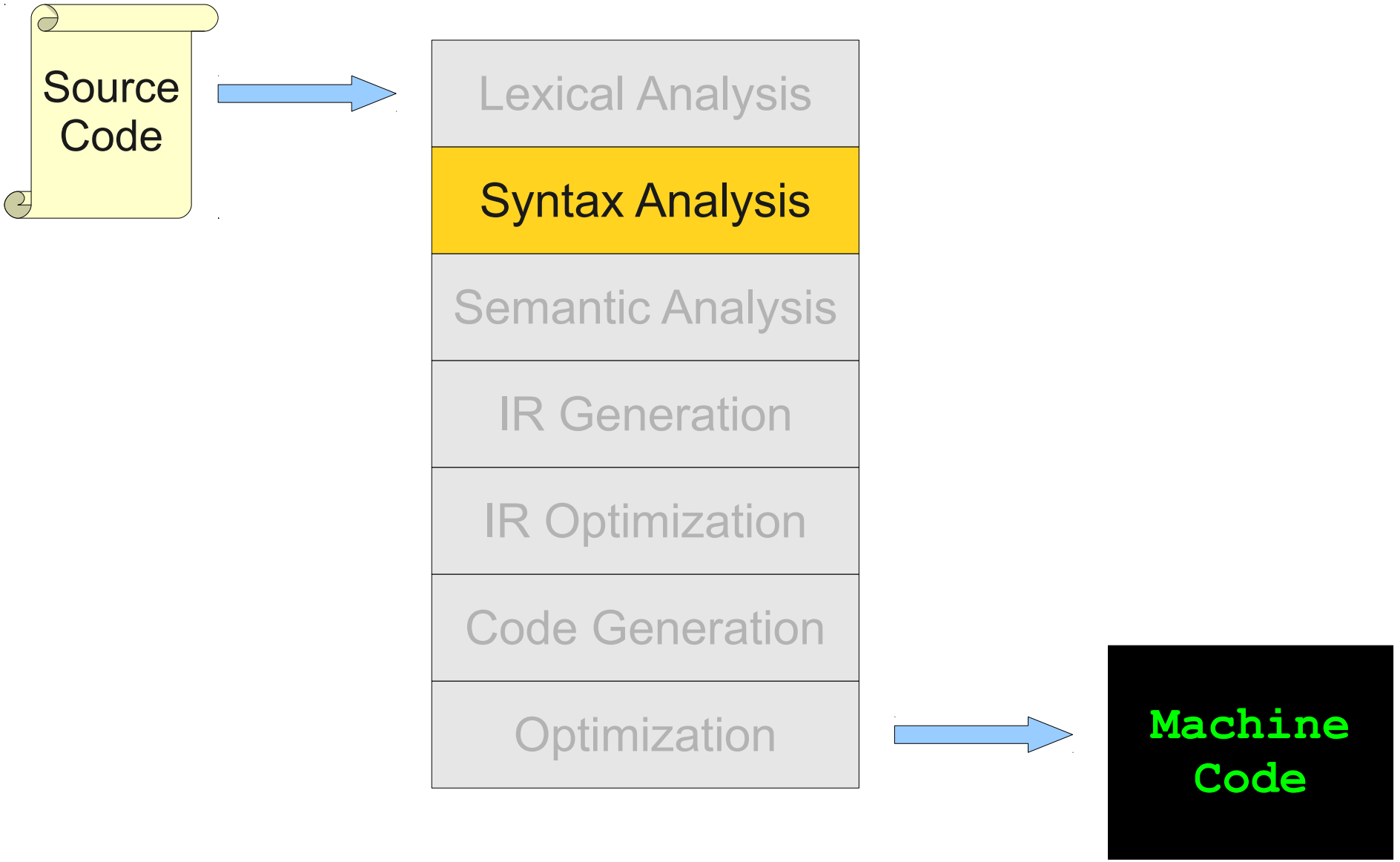
Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

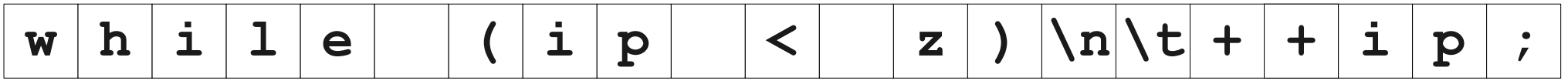
Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Machine Code
Achievement unlocked
Flex-pert

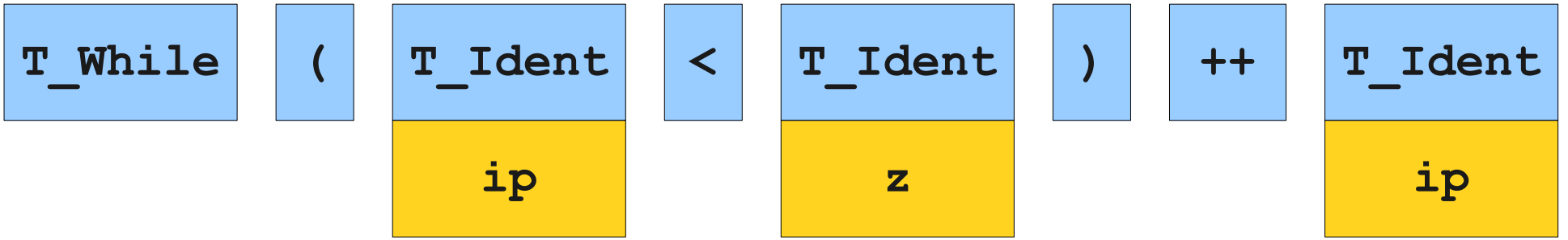
Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

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

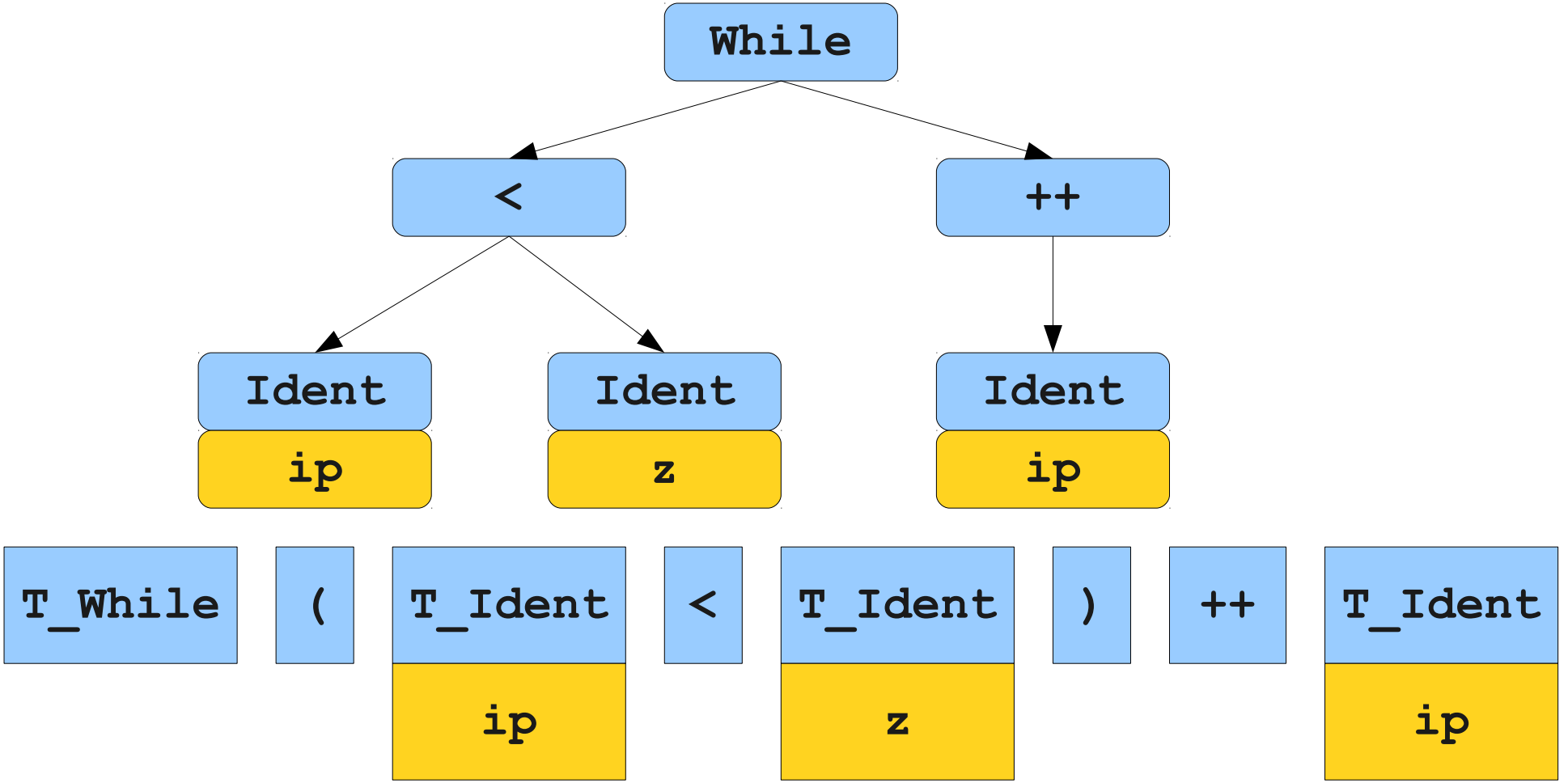



```
while (ip < z)
    ++ip;
```



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

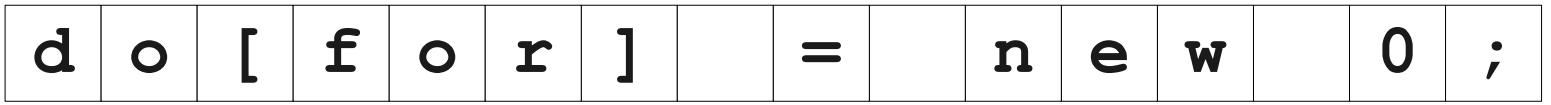
```
while (ip < z)
    ++ip;
```



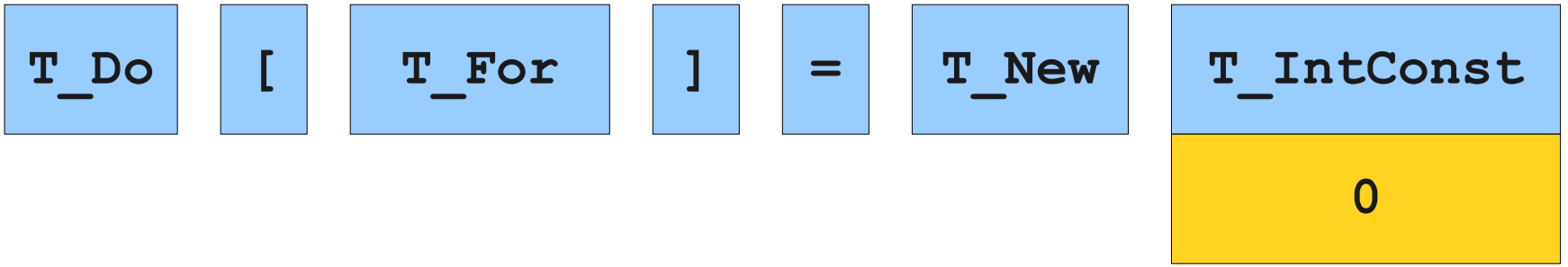
w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

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

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

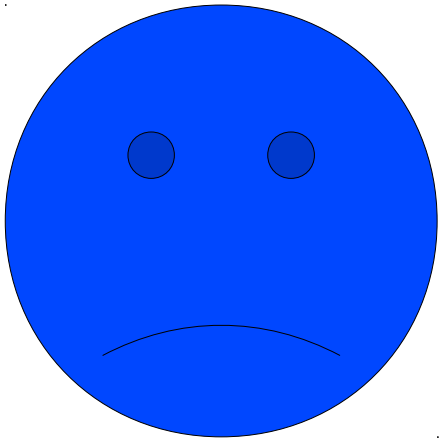



```
do[for] = new 0;
```



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

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



T_Do	[	T_For	]	=	T_New	T_IntConst
						0

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

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

What is Syntax Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- Goal: Recover the **structure** described by that series of tokens.
- Goal: Report **errors** if those tokens do not properly encode a structure.

Outline

- Today:
  - Formalisms for syntax analysis.
- Next Week:
  - Parsing algorithms.
  - Parser generators.

Recall: Languages

- Given an alphabet Σ, a **language** over Σ is a set of strings over Σ.
- During scanning, we were concerned with languages defining **tokens**:
  - Identifiers: {a, b, c, …, aa, ab, … }
  - Numbers: {0, 1, 2, … }
  - etc.
- However, we can think of a programming language as a language as well.
  - Each string is a complete program.

Describing Programming Languages

- When dealing with tokens, we used regular expressions.
- Can we do the same for programming languages?
- In general, **no**.
- Intuition: **Automata have finite memory**.
  - The only information retained at any point is what state(s) the automaton is is.
  - No way of remembering unbounded paths through the automaton (**pumping lemma**).

Non-Regular Languages

- Balanced parentheses.
  - { ε, (), (()), ()(), ((())), (())(), (()()), etc. }
- Scheme programs.
  - {1, 2, 3, …, (lambda (x) (+ x 1)), etc. }
- **Regular expressions.**
  - { ε, a, aa, a | a, a | ε, a*, ε*, (a), (ε), etc. }

Context-Free Grammars

- A tool for describing languages that is strictly more powerful than regular languages.
- Excellent way of capturing recursive structure.
- Example: Regular expressions
    - A regular expression can be
        - Any letter
        - ε
        - The concatenation of regular expressions.
        - The disjunction of regular expressions.
        - The Kleene closure of a regular expression.
        - A parenthesized regular expression.

Context-Free Grammars

- A tool for describing languages that is strictly more powerful than regular languages.
- Excellent way of capturing recursive structure.
- Example: Regular expressions

$$R \rightarrow a$$
$$R \rightarrow \varepsilon$$
$$R \rightarrow RR$$
$$R \rightarrow R \mid R$$
$$R \rightarrow R^*$$
$$R \rightarrow (R)$$

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | R

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | R

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$\mathbf{R \rightarrow RR}$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$\mathbf{R \rightarrow RR}$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | RR

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | RR

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | R**R**

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | R**R**

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$\mathbf{R \rightarrow R^*}$

$R \rightarrow (R)$

R

R | **R**

R | R**R**

R | RR*

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | R**R**

R | RR*

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | R**R**

R | **R**R*

CFG for Regular Expressions

R → a

R → ε

R → RR

R → R | R

R → R*

R → (R)

R

R | **R**

R | R**R**

R | **R**R*

CFG for Regular Expressions

R → a

R → ε

R → RR

R → R | R

R → R*

R → (R)

R

R | **R**

R | R**R**

R | **R**R*

R | cR*

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | R**R**

R | **R**R*

R | cR*

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | R**R**

R | **R**R*

R | cR*

CFG for Regular Expressions

R → a

R → ε

R → RR

R → R | R

R → R*

R → (R)

R

R | **R**

R | R**R**

R | **R**R*

R | cR*

CFG for Regular Expressions

R → a

R → ε

R → RR

R → R | R

R → R*

R → (R)

R

R | **R**

R | R**R**

R | **R**R*

R | cR*

a | cR*

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | R**R**

R | **R**R*

R | cR*

a | cR*

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | R**R**

R | **R**R*

R | cR*

a | c**R***

CFG for Regular Expressions

R → a

R → ε

R → RR

R → R | R

R → R*

R → (R)

R

R | **R**

R | R**R**

R | **R**R*

R | cR*

a | c**R***

CFG for Regular Expressions

R → a

R → ε

R → RR

R → R | R

R → R*

R → (R)

R

R | **R**

R | R**R**

R | **R**R*

R | cR*

a | c**R***

a | cb*

CFG for Regular Expressions

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

R | **R**

R | R**R**

R | **R**R*

R | cR*

a | c**R***

a | cb*

More Context-Free Grammars

- Example: Chemical Formulae

  $C_{19}H_{14}O_5S$

  $Cu_3(CO_3)_2(OH)_2$

  MnO_4^-

  S^{2-}

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | **O** | …
- *Ion* → **+** | **-** | *IonNum* **+** | *IonNum* **-**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

Cmp Ion

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

Cmp Ion

Cmp Cmp Ion

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

Cmp Ion

Cmp Cmp Ion

Cmp Term Num Ion

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

Cmp Ion

Cmp Cmp Ion

Cmp Term Num Ion

Term Term Num Ion

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

Cmp Ion

Cmp Cmp Ion

Cmp Term Num Ion

Term Term Num Ion

Elem Term Num Ion

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

Cmp Ion

Cmp Cmp Ion

Cmp Term Num Ion

Term Term Num Ion

Elem Term Num Ion

Mn *Term Num Ion*

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

Cmp Ion

Cmp Cmp Ion

Cmp Term Num Ion

Term Term Num Ion

Elem Term Num Ion

Mn *Term Num Ion*

Mn *Elem Num Ion*

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

Cmp Ion

Cmp Cmp Ion

Cmp Term Num Ion

Term Term Num Ion

Elem Term Num Ion

Mn *Term Num Ion*

Mn *Elem Num Ion*

MnO *Num Ion*

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

Cmp Ion

Cmp Cmp Ion

Cmp Term Num Ion

Term Term Num Ion

Elem Term Num Ion

Mn *Term Num Ion*

Mn *Elem Num Ion*

MnO *Num Ion*

$\mathbf{MnO_4}$ *Ion*

CFGs for Chemistry

- *Form* → *Cmp* | *Cmp Ion*
- *Cmp* → *Term* | *Term Num* | *Cmp Cmp*
- *Term* → *Elem* | **(***Cmp***)**
- *Elem* → **H** | **He** | **Li** | **Be** | **B** | **C** | **N** | …
- *Ion* → **+** | **–** | *IonNum* **+** | *IonNum* **–**
- *IonNum* → **2** | **3** | **4** | ...
- *Num* → **1** | *IonNum*

Form

Cmp Ion

Cmp Cmp Ion

Cmp Term Num Ion

Term Term Num Ion

Elem Term Num Ion

Mn *Term Num Ion*

Mn *Elem Num Ion*

MnO *Num Ion*

$\mathbf{MnO_4}$ *Ion*

$\mathbf{MnO_4^-}$

CFGs in Java

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
| ...

Formal Definition of CFGs

- A **context-free grammar** (**CFG**) is
  - A set of terminals **T**
  - A set of nonterminals **N**
  - A start symbol **S** in **N**
  - A set of productions **P**
- A production is a rule of the form

  $A \rightarrow B_1 B_2 \ldots B_n$

  where A is a nonterminal and the B_i's are either terminals or nonterminals.
- We write $A \rightarrow \varepsilon$ if A can be replaced by no symbols.

Formal Definition of Derivations

- Given a CFG, a **derivation** is a transformation

  $A_1 A_2 A_3 \ldots A_n \rightarrow A_1 A_2 \ldots A_{j-1} B_1 B_2 \ldots B_m A_{j+1} \ldots A_n$

  if there is a production

  $A_j \rightarrow B_1 B_2 \ldots B_m$

- We say that

  $A_1 A_2 \ldots A_n \rightarrow^* B_1 B_2 \ldots B_m$

  if

  $A_1 A_2 \ldots A_n \rightarrow \ldots \rightarrow \ldots \rightarrow B_1 B_2 \ldots B_m$

  using zero or more derivations.

Leftmost Derivations

BLOCK → STMT
| `{` STMTS `}`

STMTS

STMTS → ε
| STMT STMTS

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

EXPR → **`id`**
| **`constant`**
| EXPR **=** EXPR
| EXPR **+** EXPR
| EXPR **–** EXPR
| EXPR ***** EXPR
| ...

Leftmost Derivations

BLOCK → STMT
| `{` STMTS `}`

STMTS → ε
| STMT STMTS

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

EXPR → **`id`**
| **`constant`**
| EXPR **=** EXPR
| EXPR **+** EXPR
| EXPR **–** EXPR
| EXPR ***** EXPR
| ...

STMTS

→ *STMT STMTS*

Leftmost Derivations

BLOCK → STMT
| `{` STMTS `}`

STMTS → ε
| STMT STMTS

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

EXPR → **`id`**
| **`constant`**
| EXPR **=** EXPR
| EXPR **+** EXPR
| EXPR **–** EXPR
| EXPR ***** EXPR
| ...

STMTS

→ *STMT STMTS*

→ *EXPR`;` STMTS*

Leftmost Derivations

BLOCK → STMT
| `{` STMTS `}`

STMTS → ε
| STMT STMTS

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

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

STMTS

→ *STMT STMTS*

→ *EXPR`;` STMTS*

→ *EXPR* **`=`** *EXPR`;` STMTS*

Leftmost Derivations

BLOCK → STMT
| `{` STMTS `}`

STMTS → ε
| STMT STMTS

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

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

STMTS
→ *STMT STMTS*
→ *EXPR`;` STMTS*
→ *EXPR* **`=`** *EXPR`;` STMTS*
→ **`id =`** *EXPR`;` STMTS*

Leftmost Derivations

BLOCK → STMT
| `{` STMTS `}`

STMTS → ε
| STMT STMTS

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

EXPR → **`id`**
| **`constant`**
| EXPR **`=`** EXPR
| EXPR **`+`** EXPR
| EXPR **`-`** EXPR
| EXPR **`*`** EXPR
| …

STMTS

→ *STMT STMTS*

→ *EXPR`;` STMTS*

→ *EXPR* **`=`** *EXPR`;` STMTS*

→ **`id =`** *EXPR`;` STMTS*

→ **`id =`** *EXPR* **`+`** *EXPR`;` STMTS*

Leftmost Derivations

BLOCK → STMT
| `{` STMTS `}`

STMTS → ε
| STMT STMTS

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

EXPR → **`id`**
| **`constant`**
| EXPR `=` 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*

Leftmost Derivations

BLOCK → STMT
| `{` STMTS `}`

STMTS → ε
| STMT STMTS

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

EXPR → **`id`**
| **`constant`**
| EXPR **`=`** 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*

Leftmost Derivations

BLOCK → STMT
| `{` STMTS `}`

STMTS → ε
| STMT STMTS

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

EXPR → **`id`**
| **`constant`**
| EXPR = 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;`**

Leftmost Derivations

- A **leftmost derivation** expands the leftmost nonterminal first.
- A **rightmost derivation** expands the rightmost nonterminal first.
- These will be of great importance when we talk about parsing next week.

Parse Trees

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

Parse Trees

$R \rightarrow a$

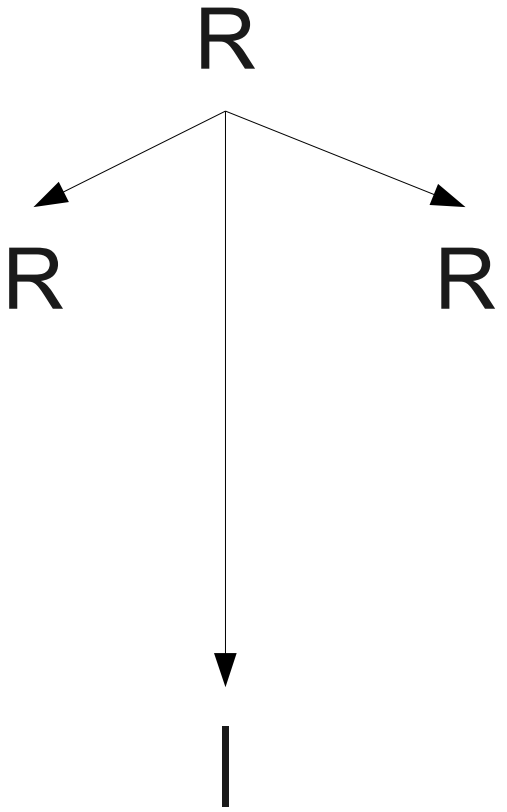
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

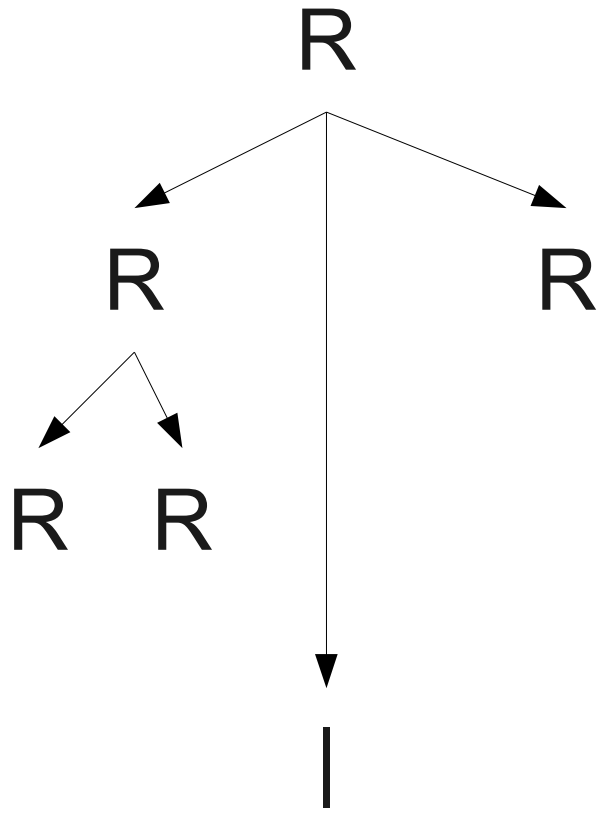
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

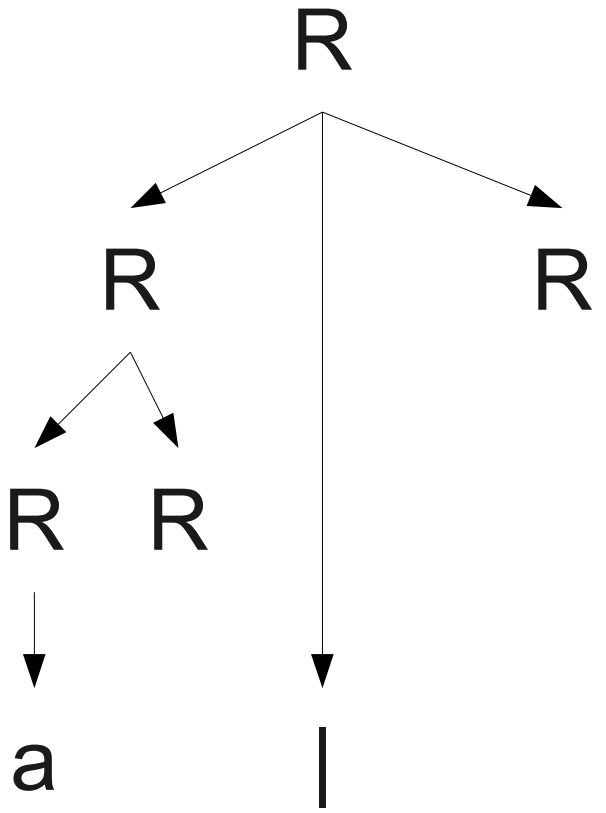
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

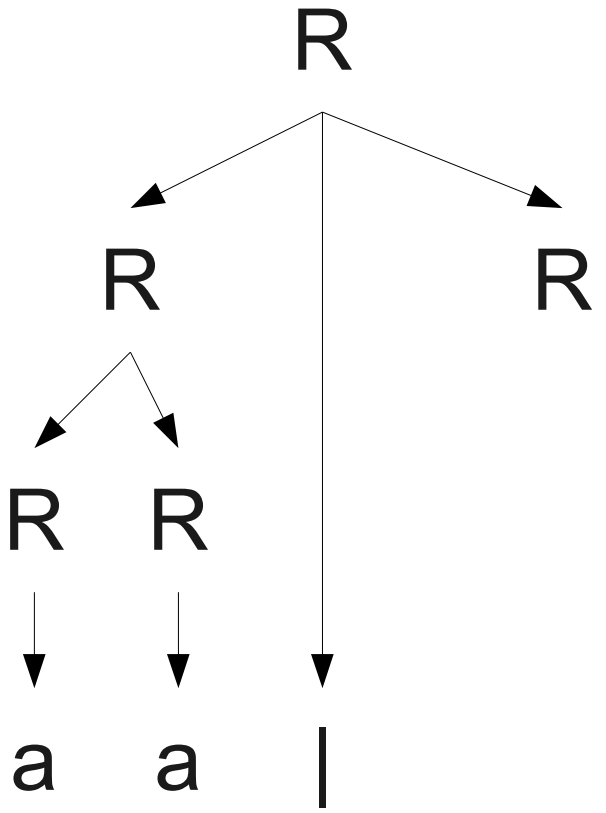
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

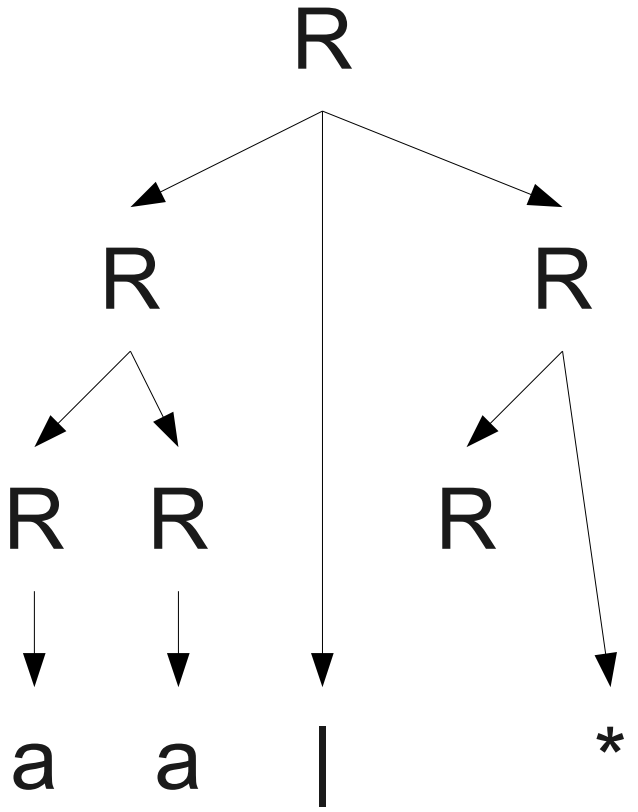
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

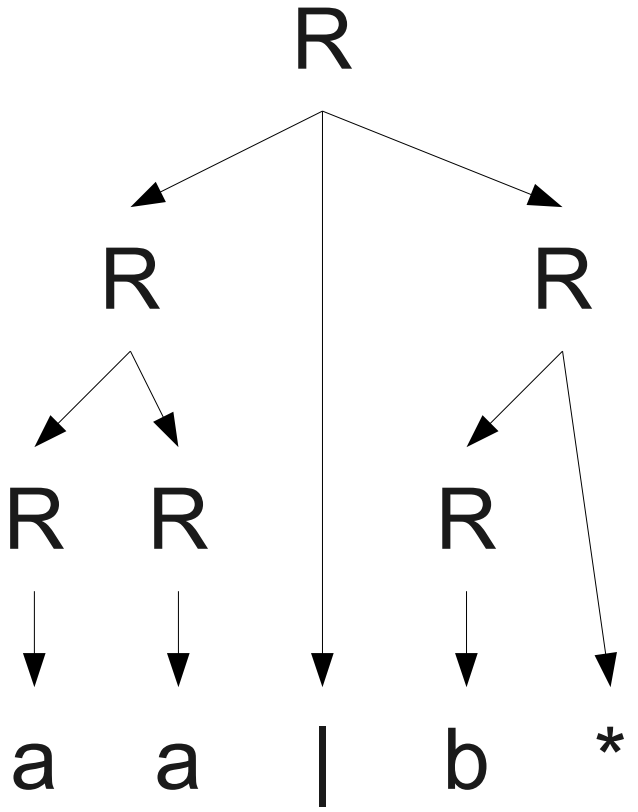
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

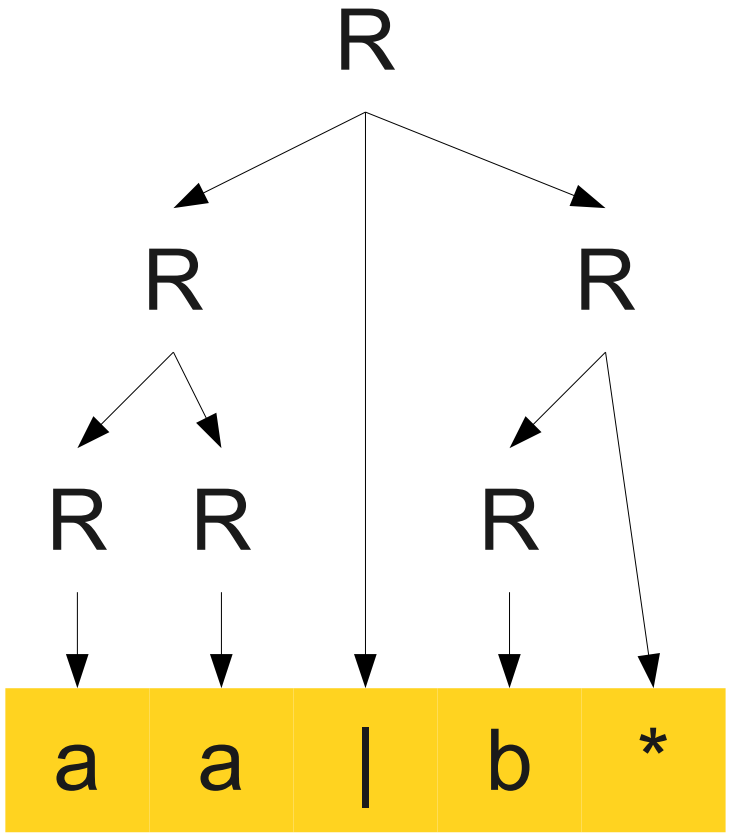
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

- Tree structure encoding the steps in a derivation.
- Inorder walk of the leaves contains the generated string.
- A derivation encodes **how** to produce the input.
- A parse tree encodes the **structure** of the input.

Syntax Analysis

- Goal of syntax analysis: Recover the **structure** described by a series of tokens.
- Use a context-free grammar:
  - Define a CFG for the language whose terminals are the tokens emitted by the scanner.
  - Obtain a parse tree for those tokens to recover the underlying structure.

Parse Trees

$R \rightarrow a$

$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$

R

Parse Trees

$R \rightarrow a$

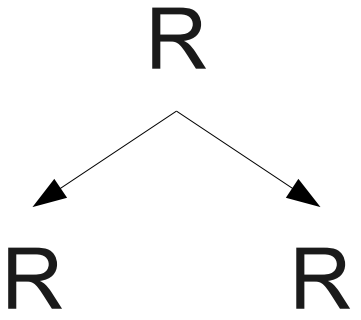
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

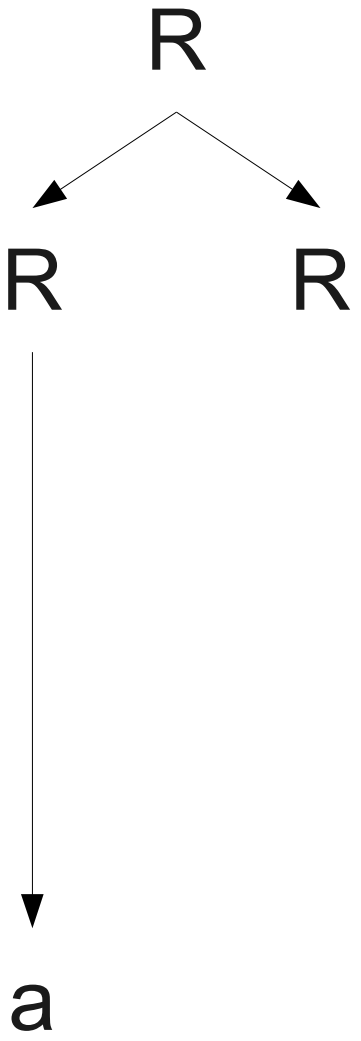
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

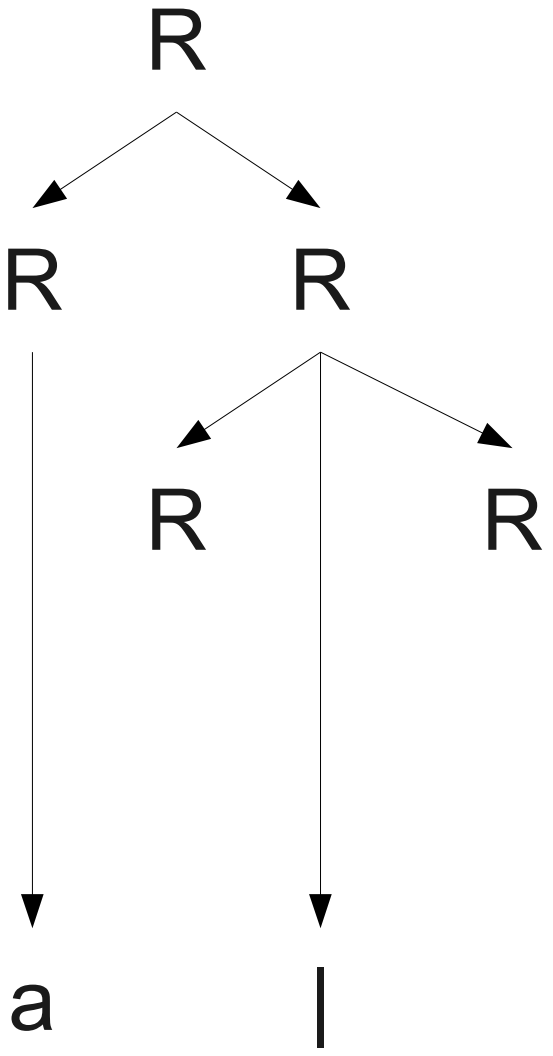
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

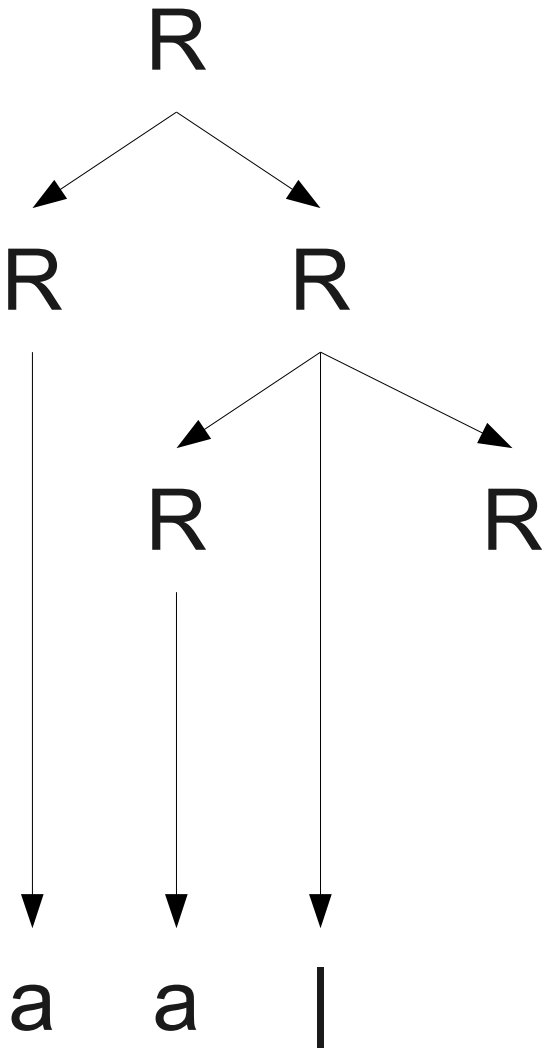
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

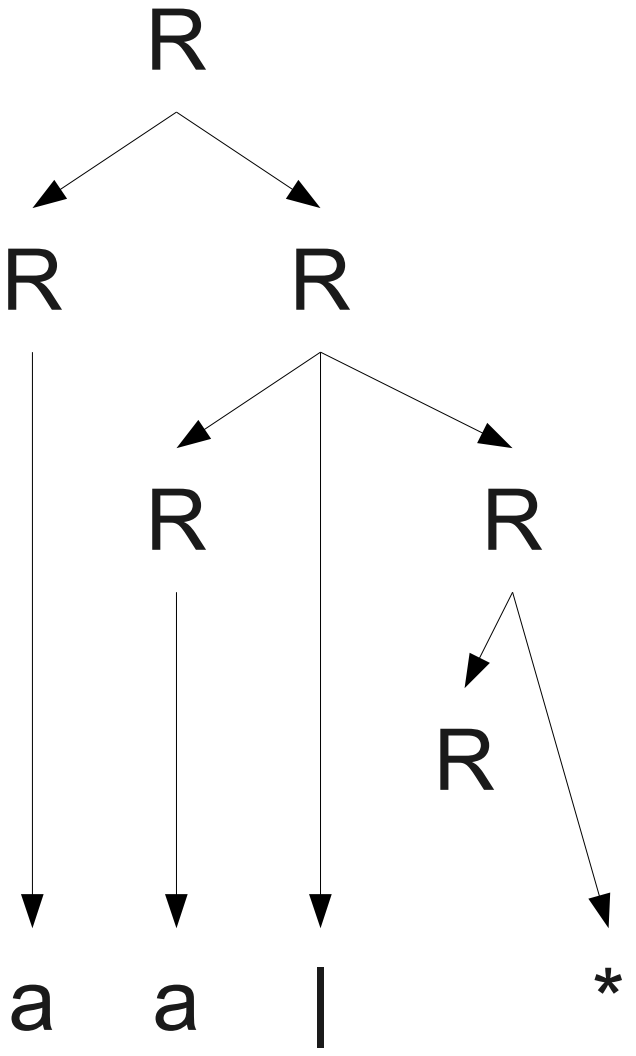
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

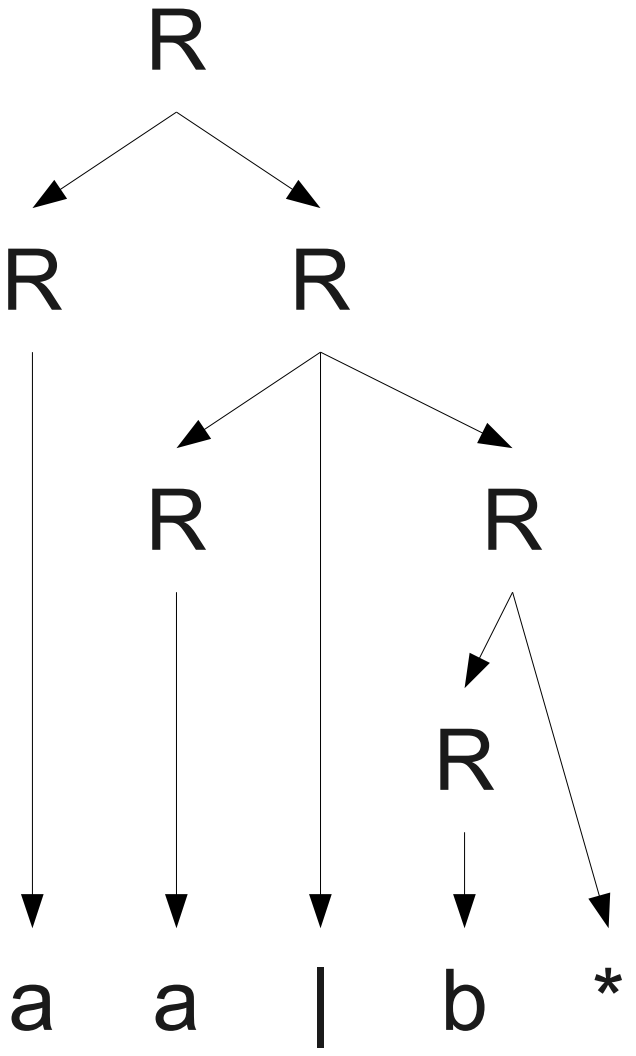
$R \rightarrow \varepsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Parse Trees

$R \rightarrow a$

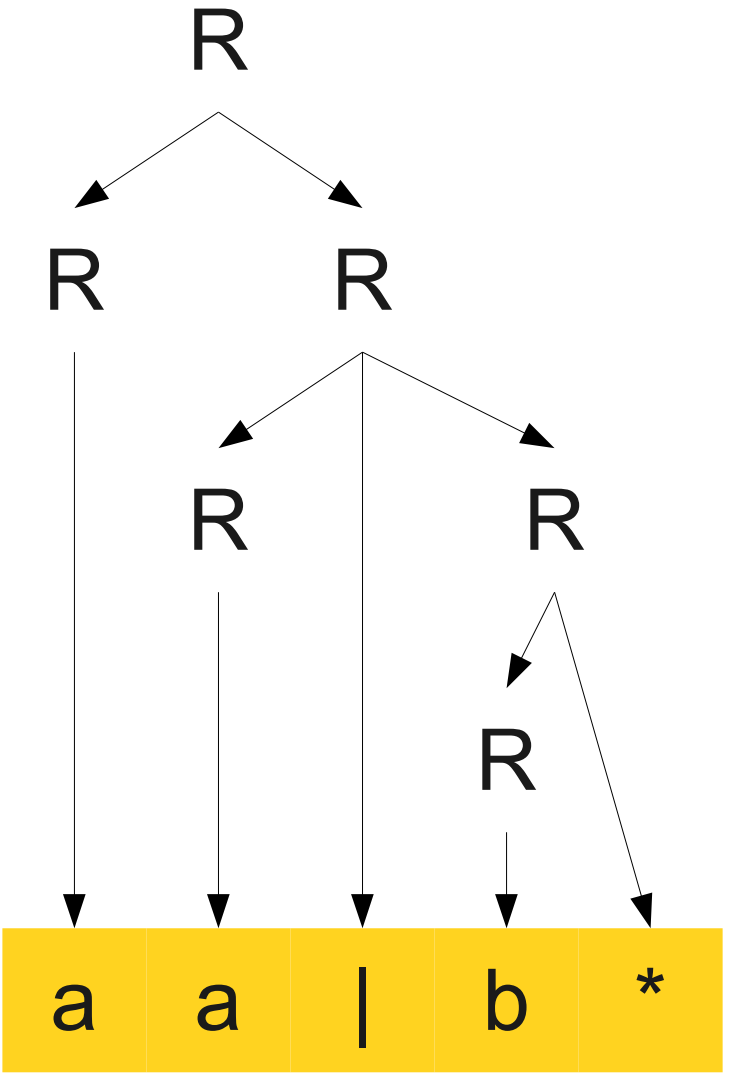
$R \rightarrow \varepsilon$

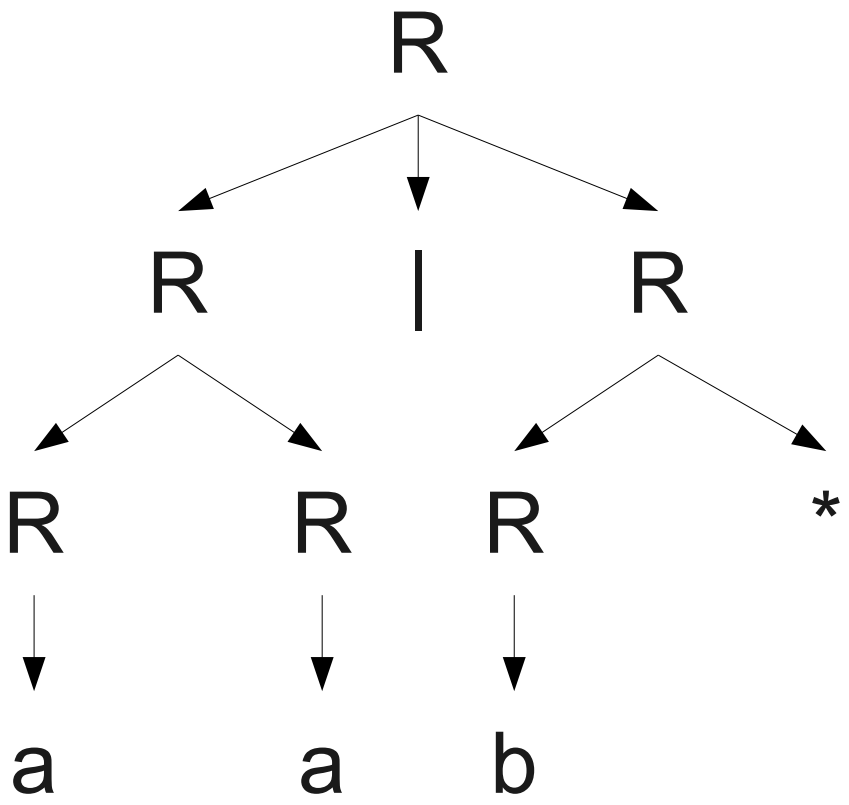
$R \rightarrow RR$

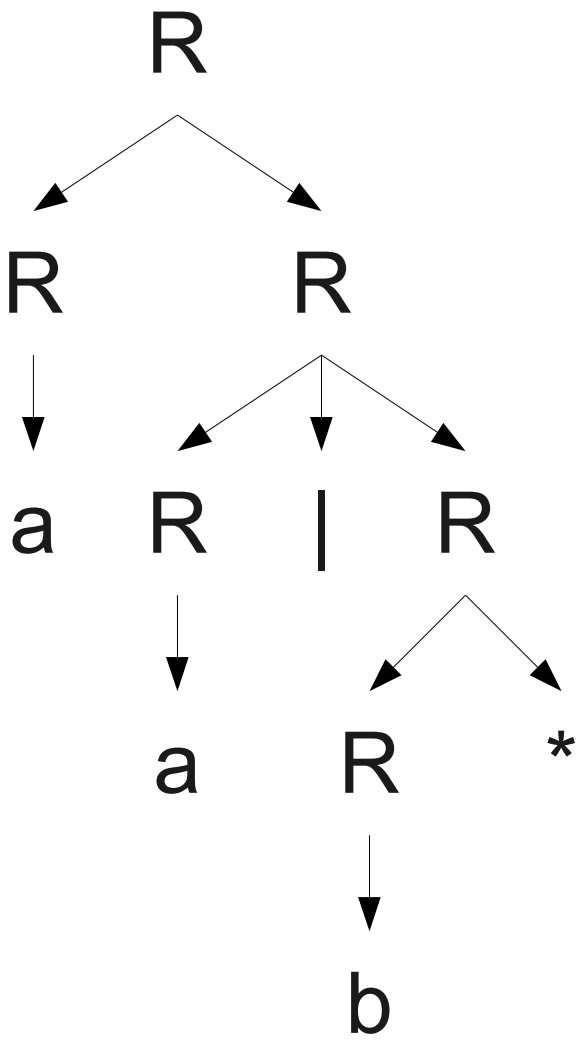
$R \rightarrow R \mid R$

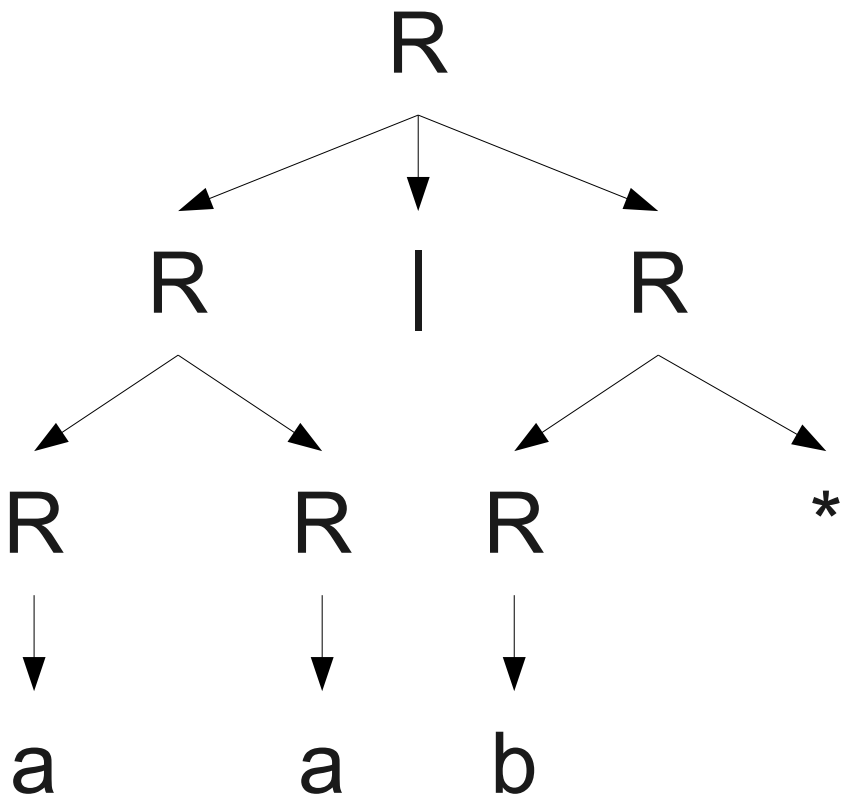
$R \rightarrow R^*$

$R \rightarrow (R)$

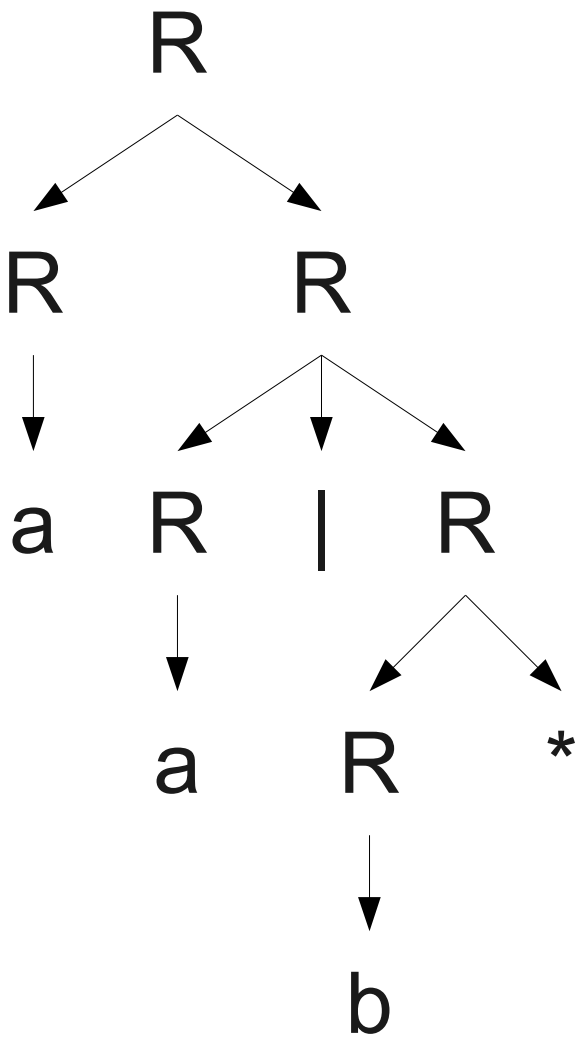



R
R
|
R
R
R
R
*
a
a
b


R
R
R
a
R
|
R
a
R
*
b

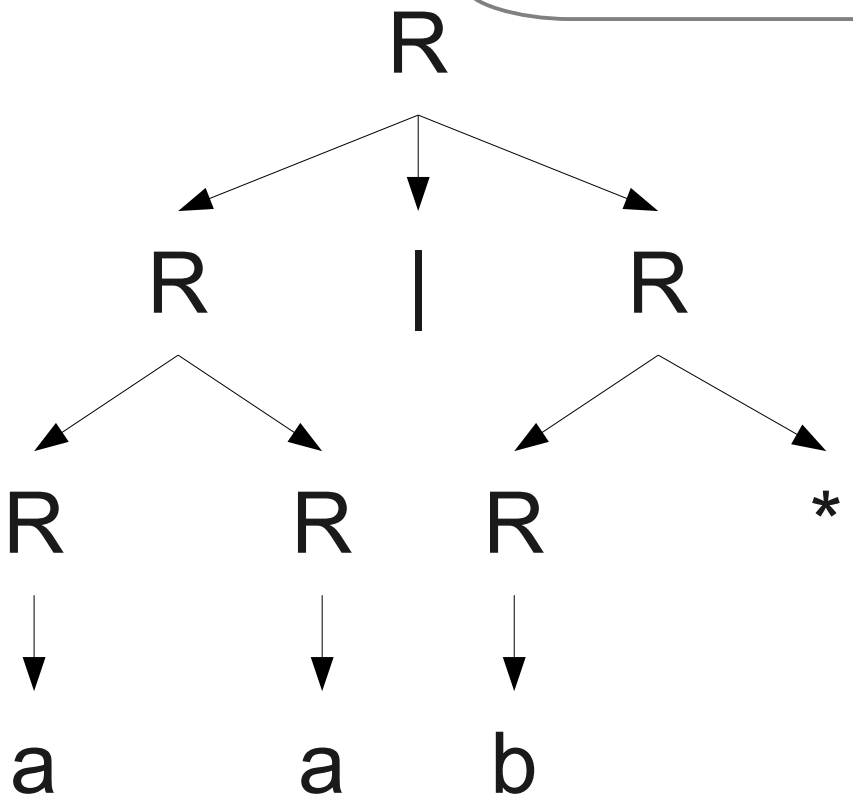


(aa) | (b*)

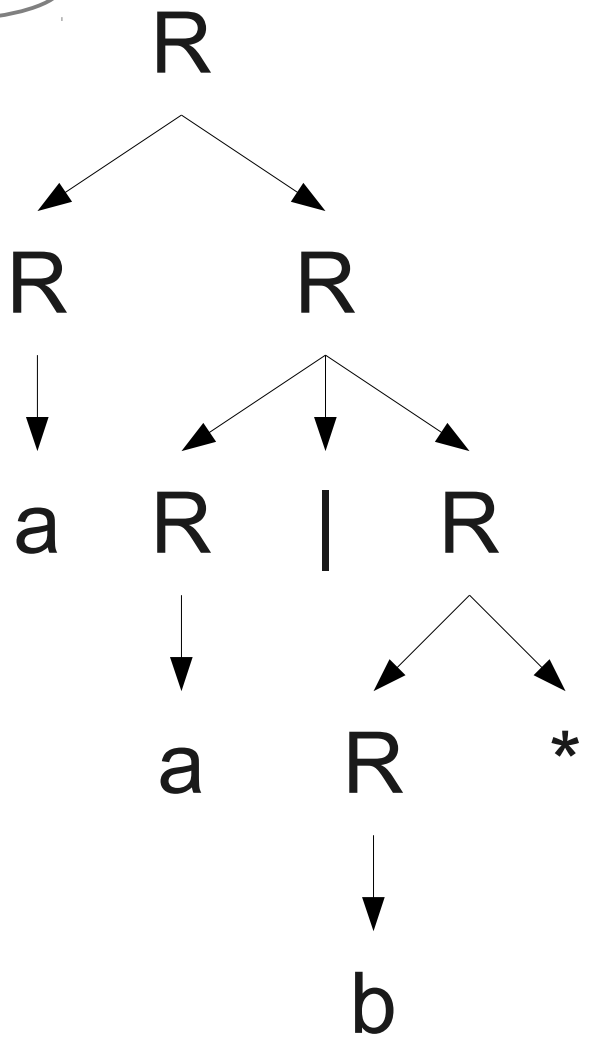


a(a | (b*))

Problem?



(aa) | (b*)



a(a | (b*))

Ambiguity

- A CFG is said to be **ambiguous** if there is at least one string with two or more derivations.
- Unfortunately:
  - It is **undecidable** whether a given grammar is ambiguous.
  - There are some languages that are **inherently** ambiguous.
- However, in most cases, the ambiguity is in the **grammar** and not the **language**.

Is Ambiguity a Problem?

Is Ambiguity a Problem?

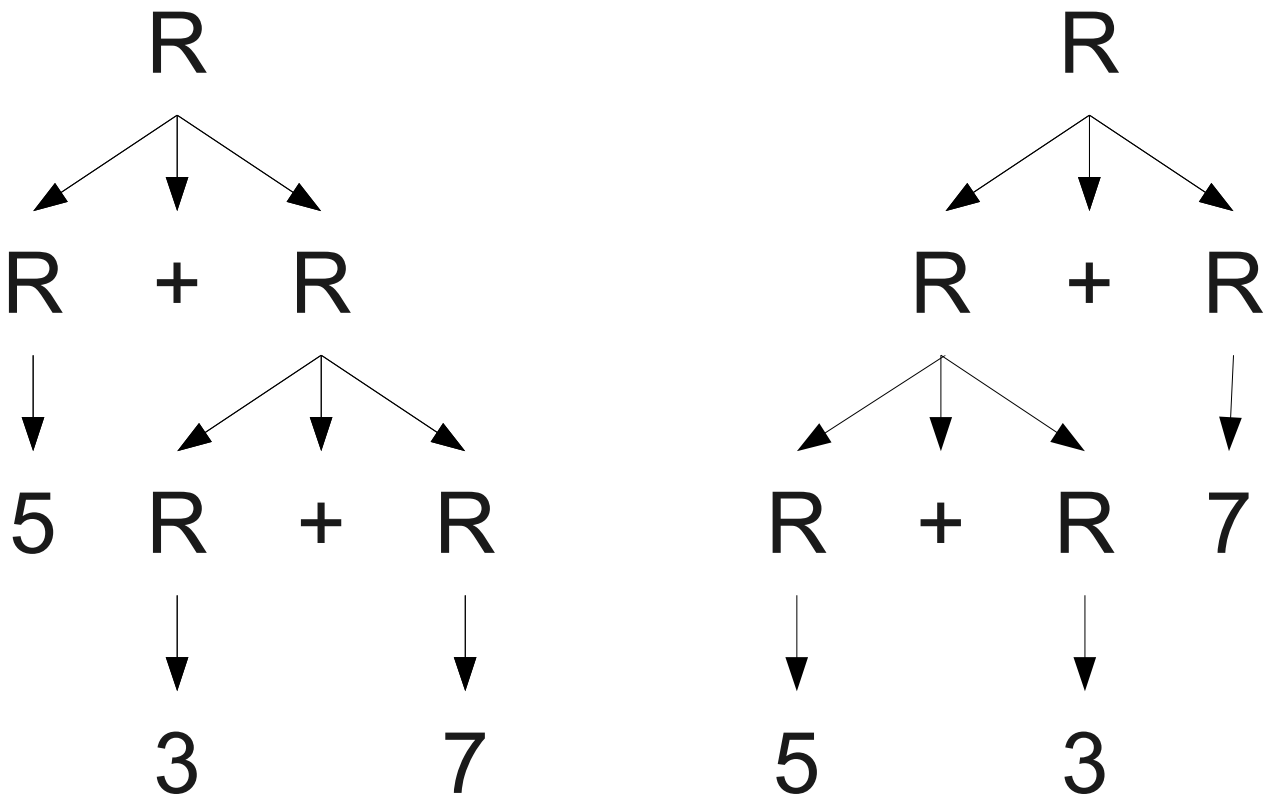
- Depends on **semantics**.

Is Ambiguity a Problem?

- Depends on **semantics**.
- E → **int** | E + E

Is Ambiguity a Problem?

- Depends on **semantics**.
- E → **int** | E + E

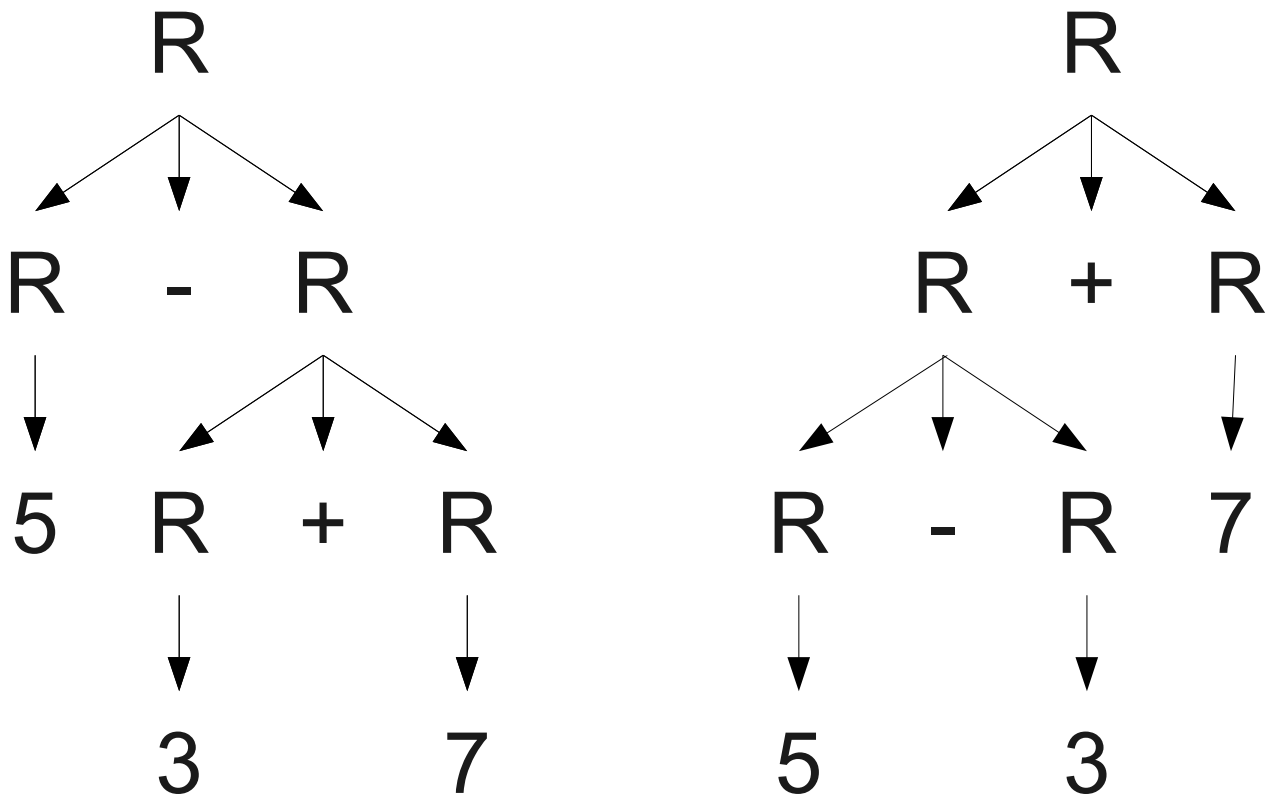


Is Ambiguity a Problem?

- Depends on **semantics**.
- E → **int** | E + E | E – E

Is Ambiguity a Problem?

- Depends on **semantics**.
- E → **int** | E + E | E – E



Resolving Ambiguity

- Return to our CFG for regular expressions:

  $R \rightarrow R \mid R$

  $R \rightarrow RR$

  $R \rightarrow R^*$

  $R \rightarrow a$

  $R \rightarrow \varepsilon$

  $R \rightarrow (R)$

- Kleene closure should have highest precedence, then concatenation, then disjunction.

Resolving Ambiguity

- Return to our CFG for regular expressions:

  R → R "|" R

  R → RR

  R → R*

  R → "a"

  R → "ε"

  R → (R)

- Kleene closure should have highest precedence, then concatenation, then disjunction.

Resolving Ambiguity

- Return to our CFG for regular expressions:

R → R "|" R	R → S | R "|" S
R → RR	S → T | ST
R → R*	T → U | T*
R → "a"	U → "a"
R → "ε"	U → "ε"
R → (R)	U → (R)

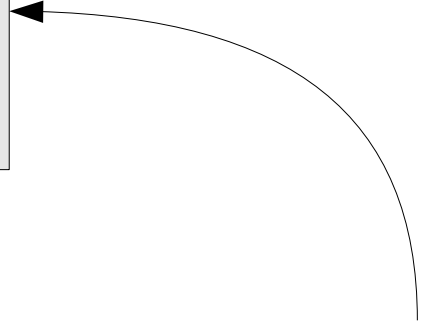
- Kleene closure should have highest precedence, then concatenation, then disjunction.

Why is this unambiguous?

R → S | R "|" S
S → T | ST
T → U | T*
U → "a"
U → "ε"
U → (R)

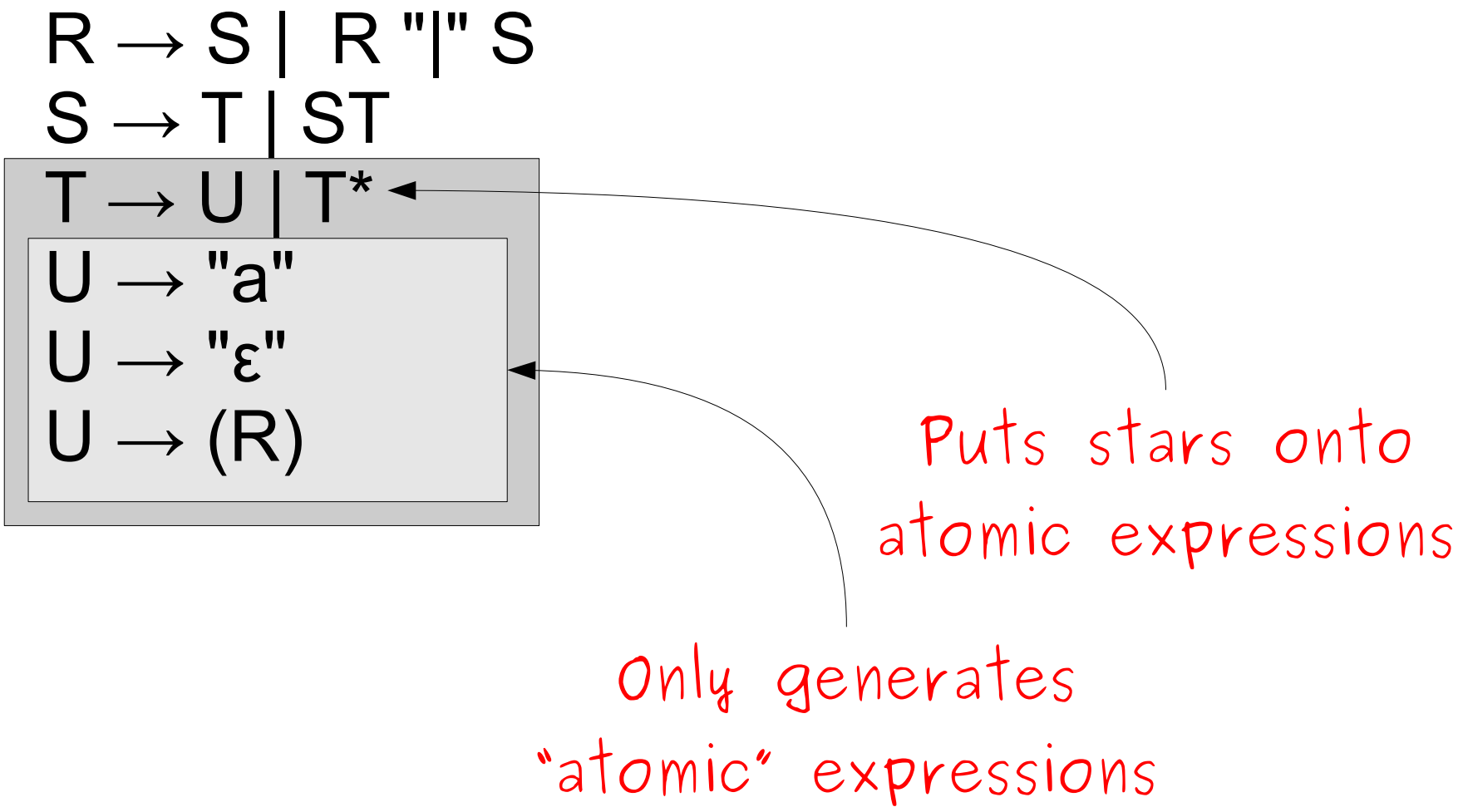
Why is this unambiguous?

R → S | R "|" S
S → T | ST
T → U | T*
U → "a"
U → "ε"
U → (R)

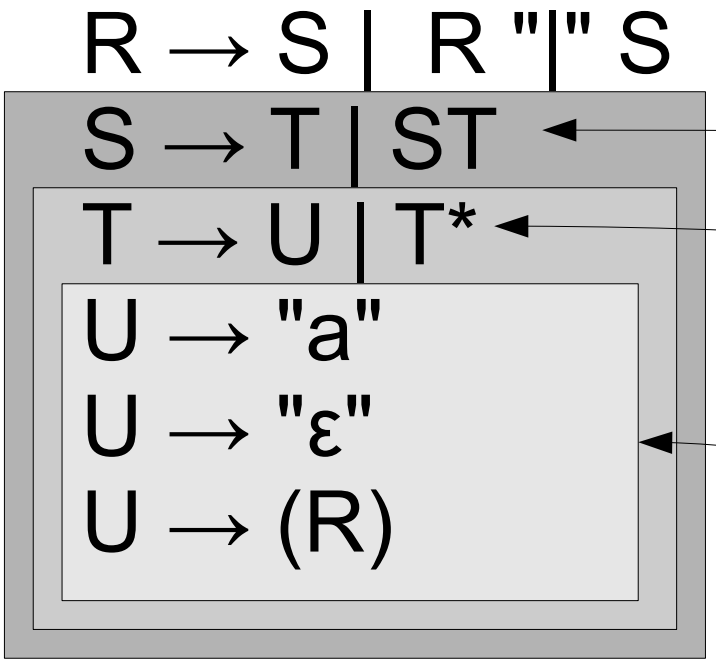


Only generates
"atomic" expressions

Why is this unambiguous?



Why is this unambiguous?

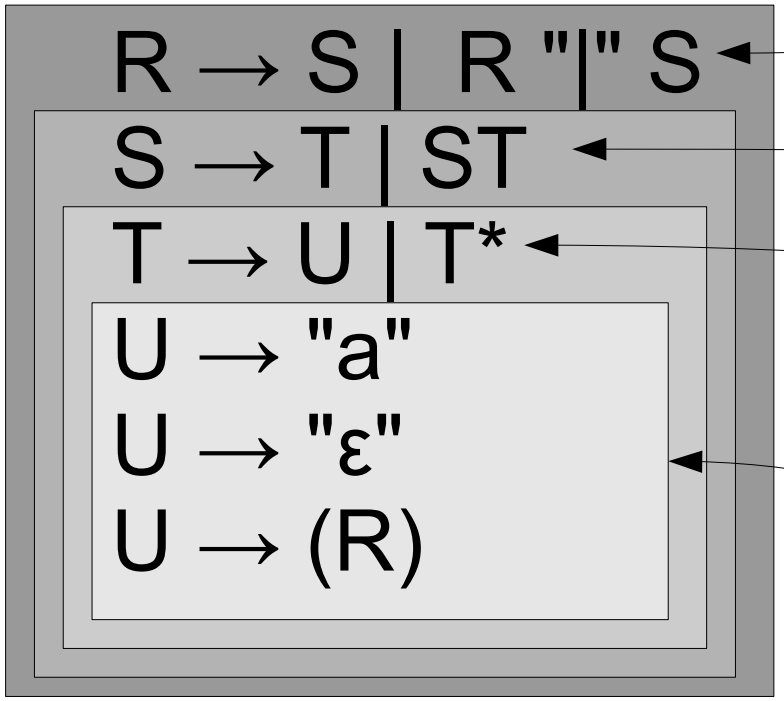


Concatenates starred expressions

Puts stars onto atomic expressions

Only generates "atomic" expressions

Why is this unambiguous?



Ors concatenated expressions

Concatenates starred expressions

Puts stars onto atomic expressions

Only generates "atomic" expressions

Resolving Ambiguity: Precedence

- If we leave the world of pure CFGs, we can often resolve ambiguities through **precedence declarations**.
- Used in many parser generators, including **`bison`**.
- We'll see how these are implemented later on.

The Structure of a Parse Tree

R → S | R "|" S
S → T | ST
T → U | T*
U → "a"
U → "ε"
U → (R)

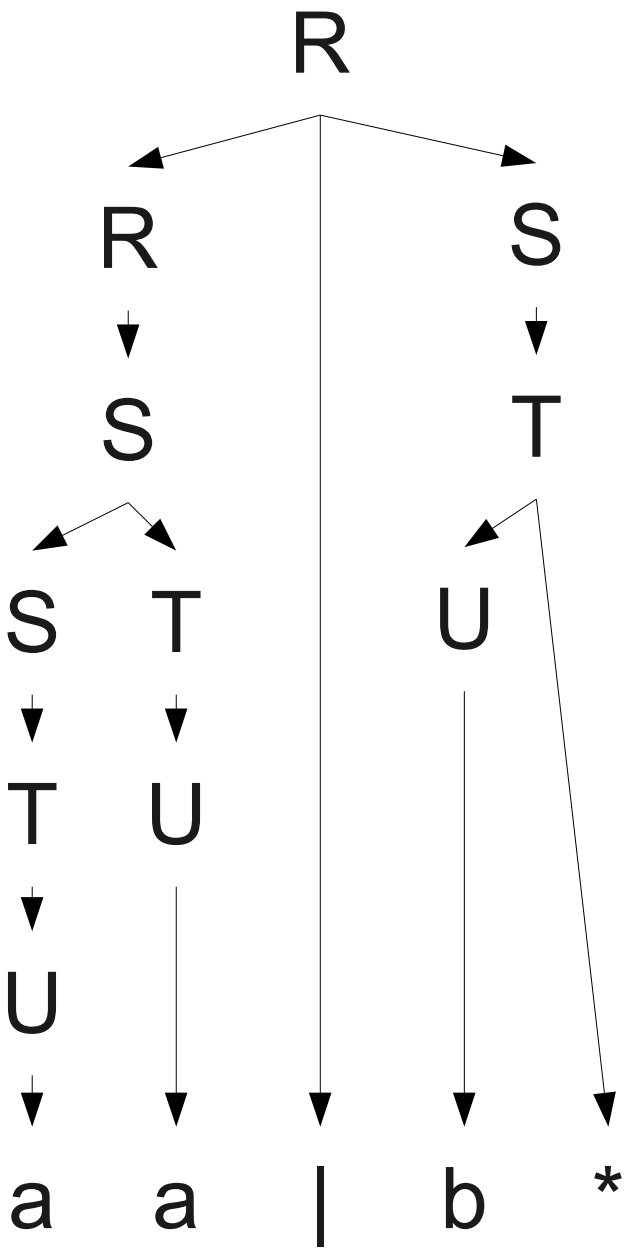
The Structure of a Parse Tree

R → S | R "|" S
S → T | ST
T → U | T*
U → "a"
U → "ε"
U → (R)

a a | b *

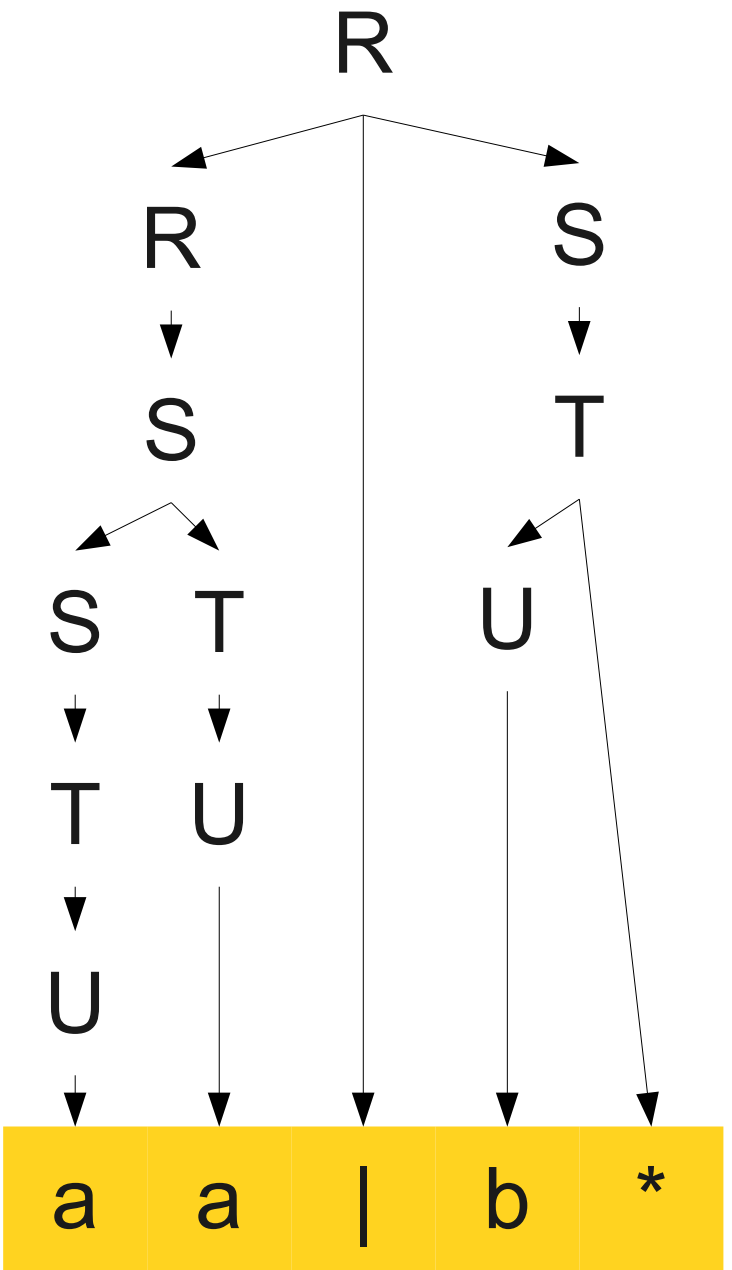
The Structure of a Parse Tree

$R \rightarrow S \mid R \text{ "|" } S$
$S \rightarrow T \mid ST$
$T \rightarrow U \mid T^*$
$U \rightarrow \text{"a"}$
$U \rightarrow \text{"}\varepsilon\text{"}$
$U \rightarrow (R)$



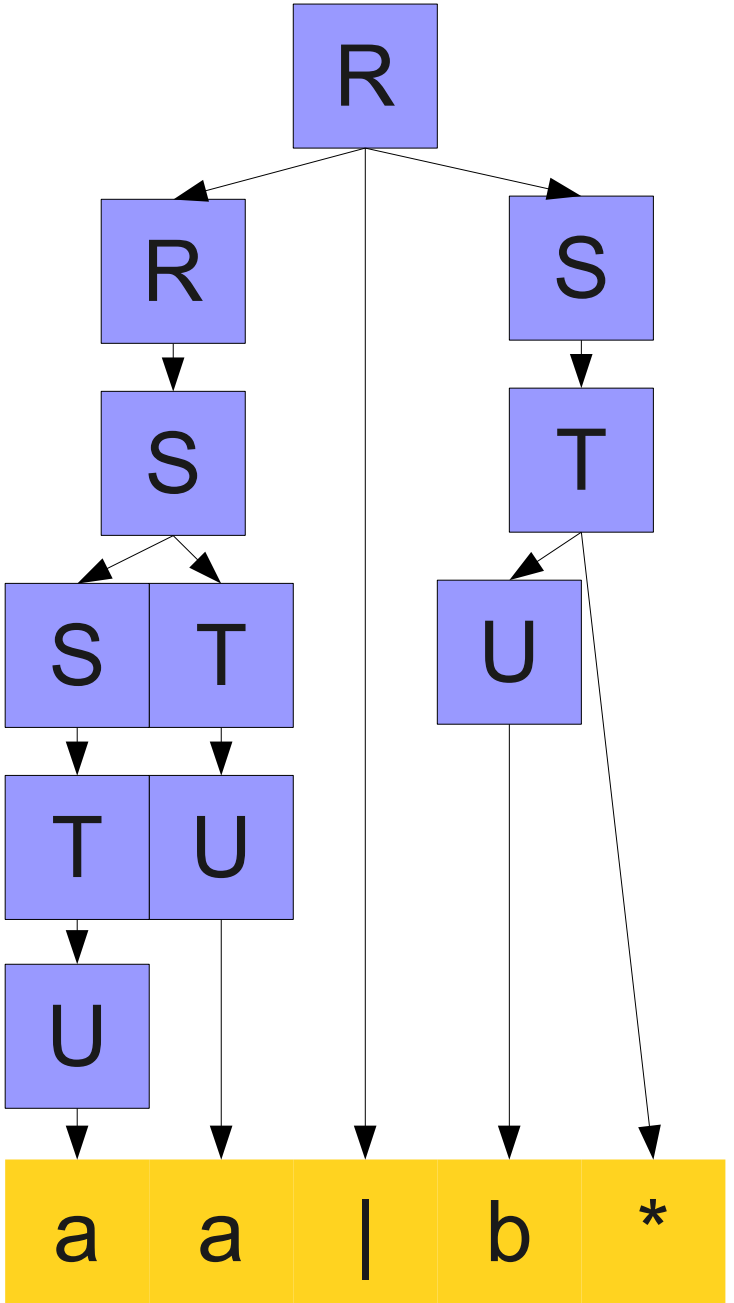
The Structure of a Parse Tree

$R \rightarrow S \mid R \text{ "|" } S$
$S \rightarrow T \mid ST$
$T \rightarrow U \mid T^*$
$U \rightarrow \text{"a"}$
$U \rightarrow \text{"}\varepsilon\text{"}$
$U \rightarrow (R)$



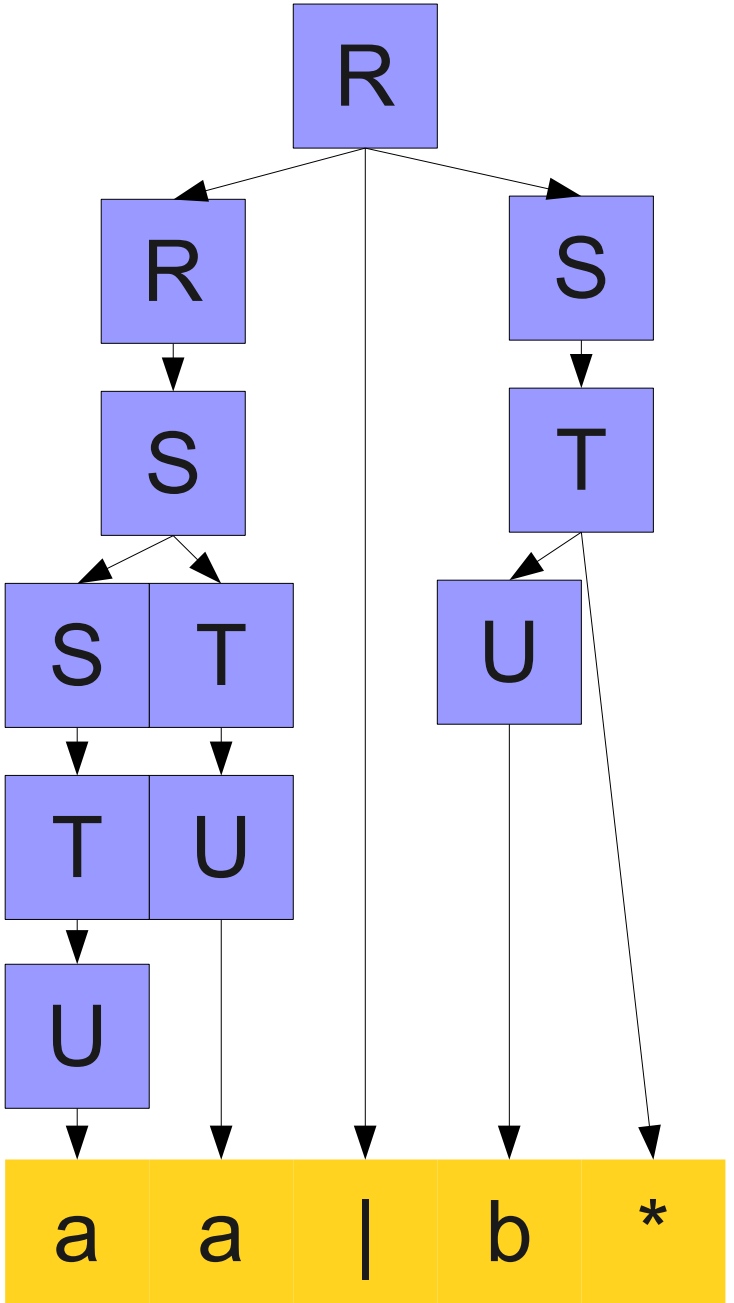
The Structure of a Parse Tree

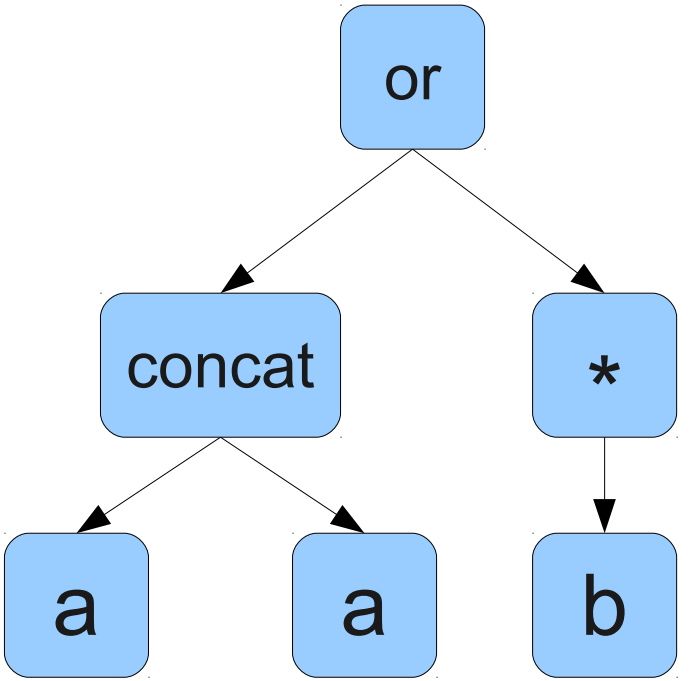
R → S | R "|" S
S → T | ST
T → U | T*
U → "a"
U → "ε"
U → (R)



The Structure of a Parse Tree

R → S | R "|" S
S → T | ST
T → U | T*
U → "a"
U → "ε"
U → (R)



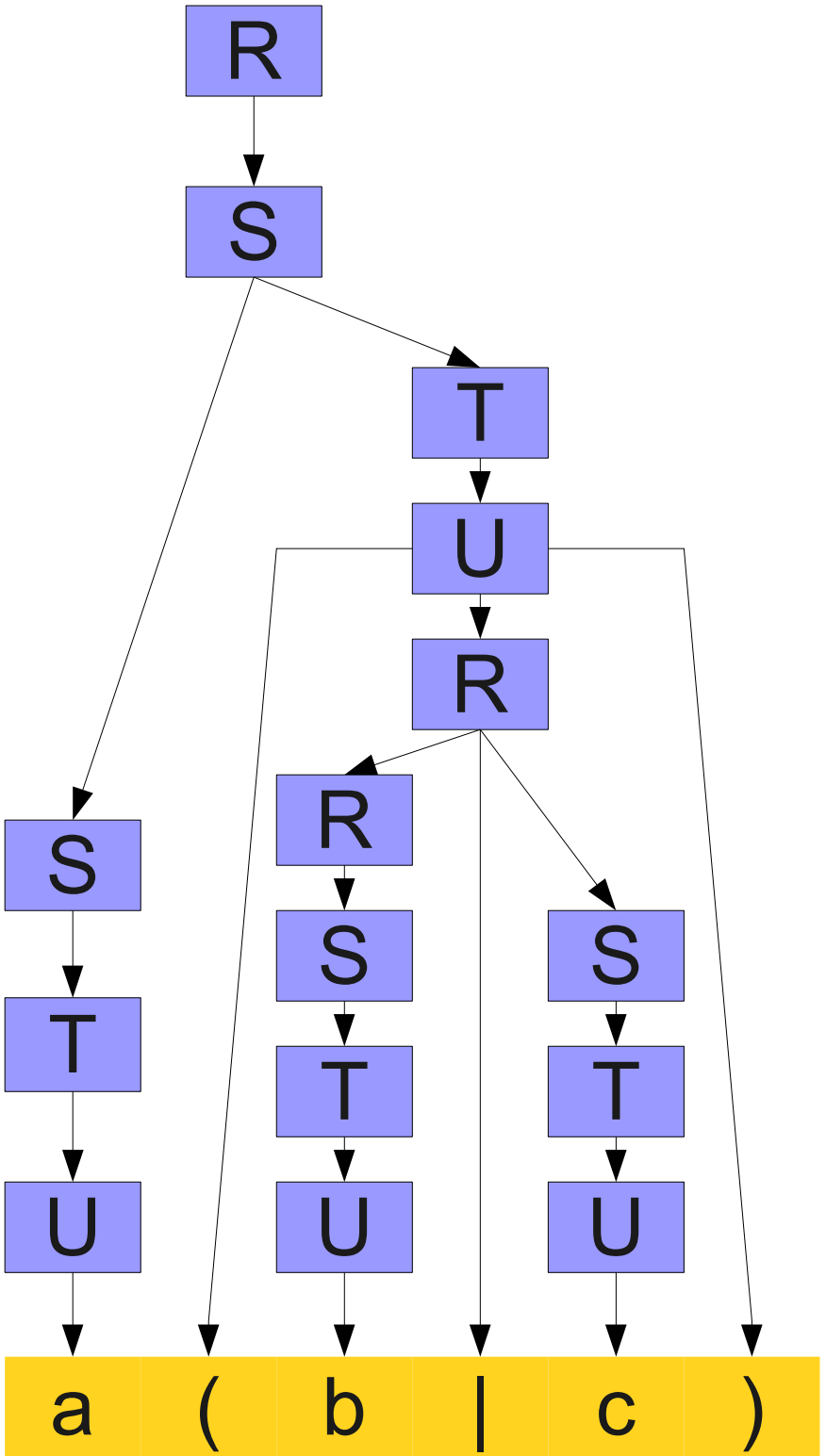


R → S | R "|" S
S → T | ST
T → U | T*
U → "a"
U → "ε"
U → (R)

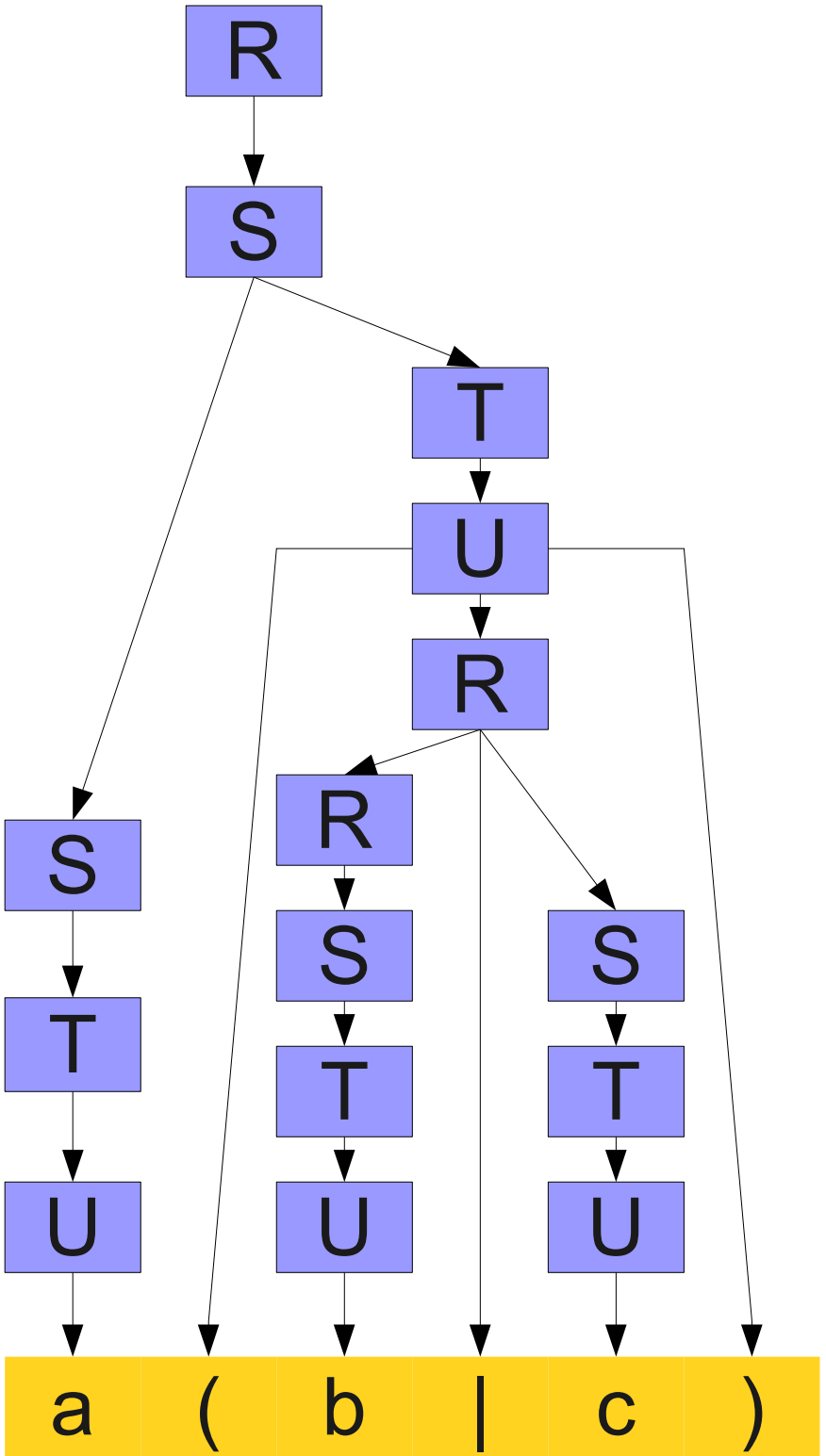
R → S | R "|" S
S → T | ST
T → U | T*
U → "a"
U → "ε"
U → (R)

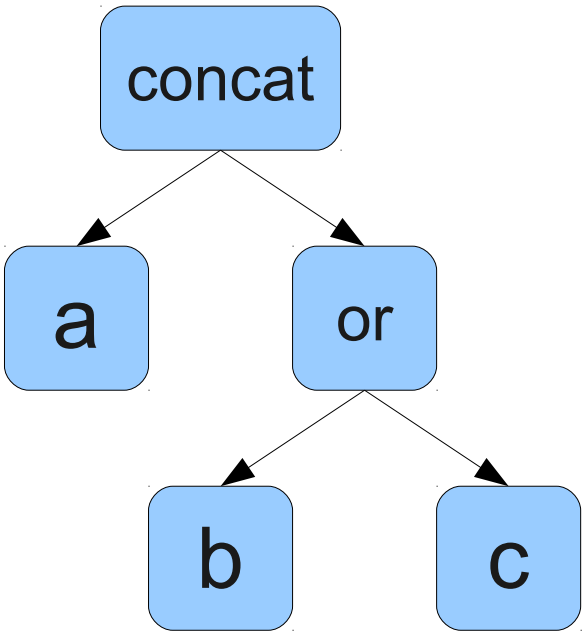
a (b | c)

R → S | R "|" S
S → T | ST
T → U | T*
U → "a"
U → "ε"
U → (R)



R → S | R "|" S
S → T | ST
T → U | T*
U → "a"
U → "ε"
U → (R)





Abstract Syntax Trees (ASTs)

- Tree structure encoding the **logical structure** of a piece of code.
  - Abstracts from the particular syntax being used.
- Eliminate unnecessary grammar symbols.
- Eliminate nodes from grammar internals.
- Ultimate output of syntax analysis.

How to build an AST?

- Typically done through **semantic actions**.
- Idea: **Associate code 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**.

Simple Semantic Actions

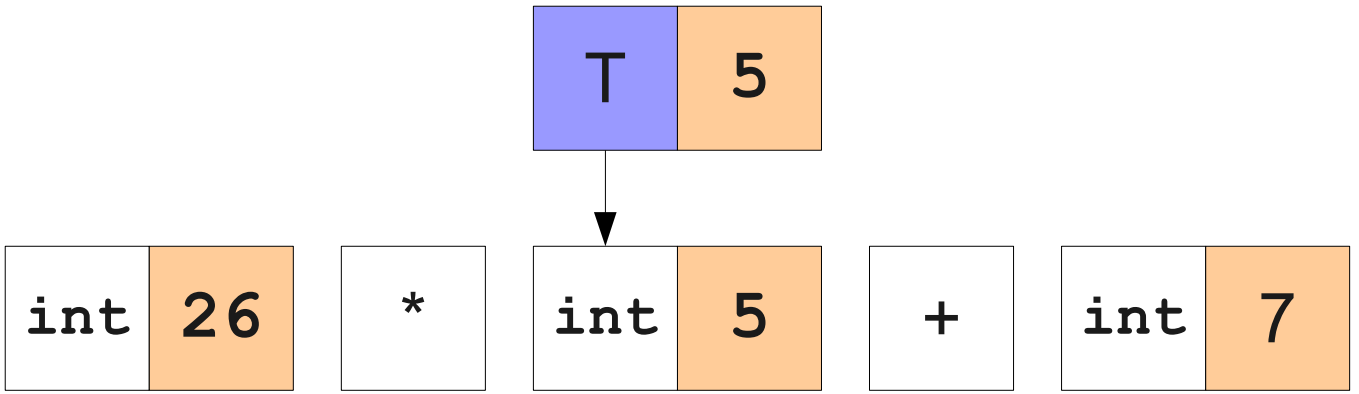
$E \rightarrow T + E$	$E_1.val = T.val + E_2.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow \textbf{int}$	$T.val = int.val$
$T \rightarrow \textbf{int} * T$	$T.val = int.val * T.val$
$T \rightarrow (E)$	$T.val = E.val$

Simple Semantic Actions

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

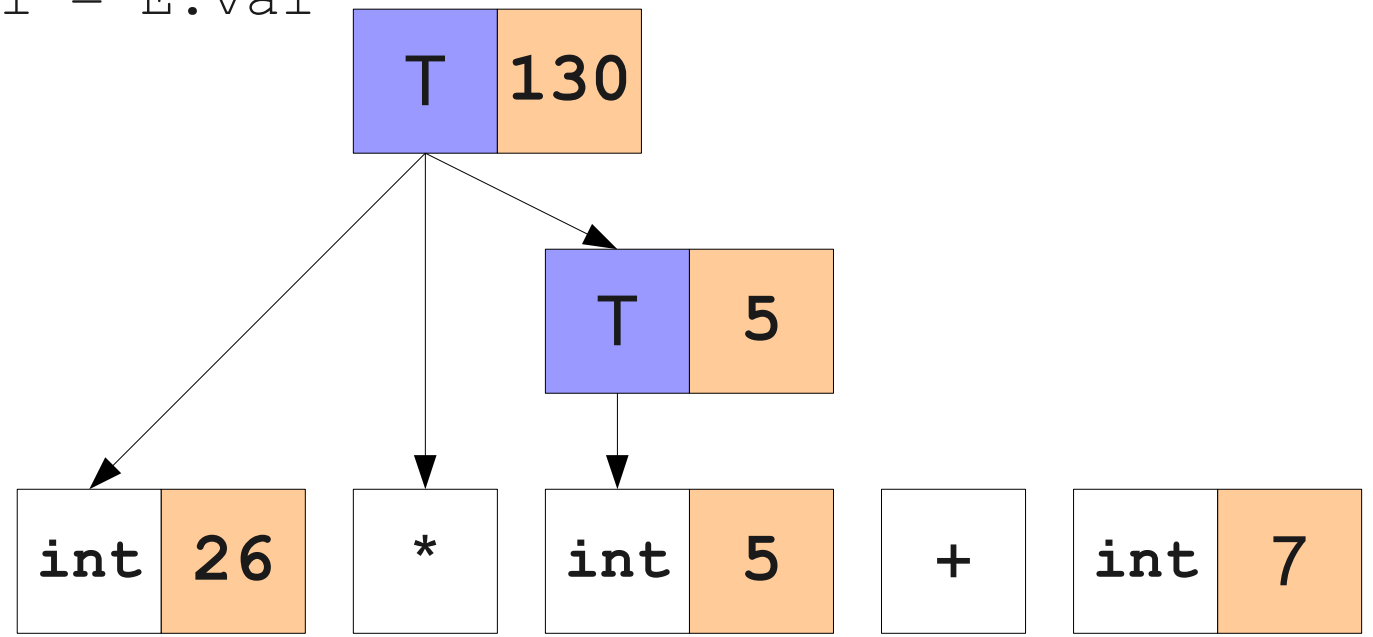
Simple Semantic Actions

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



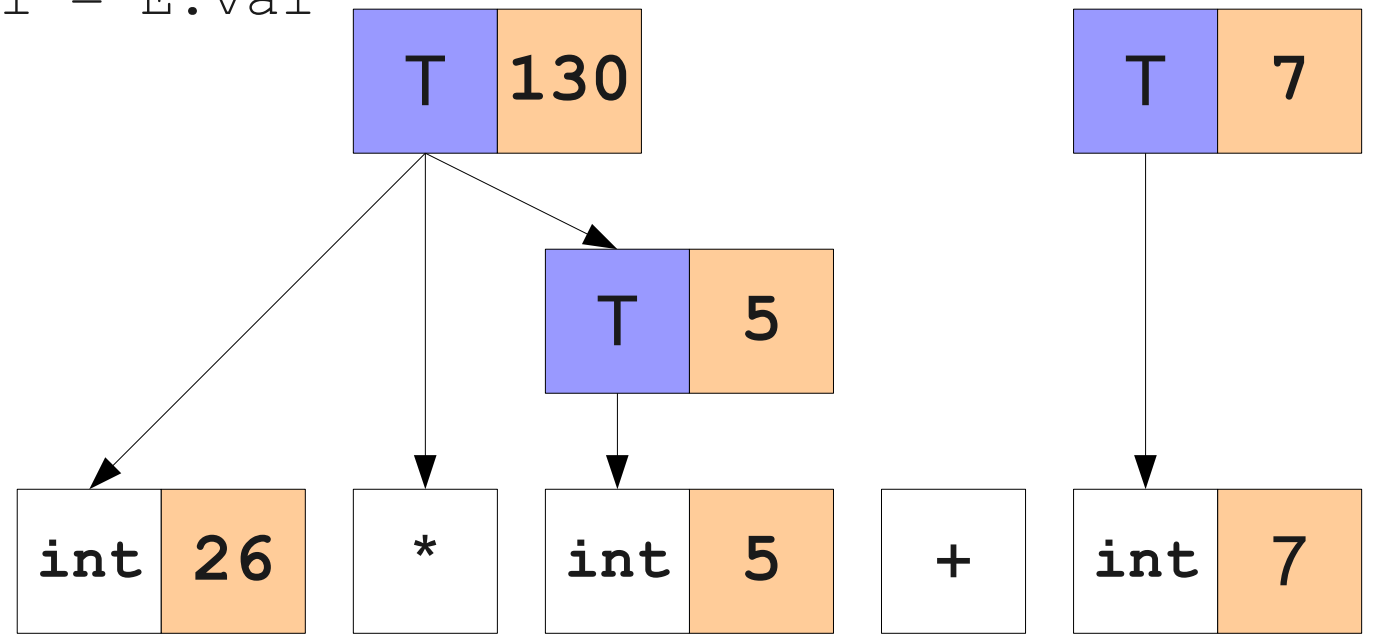
Simple Semantic Actions

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



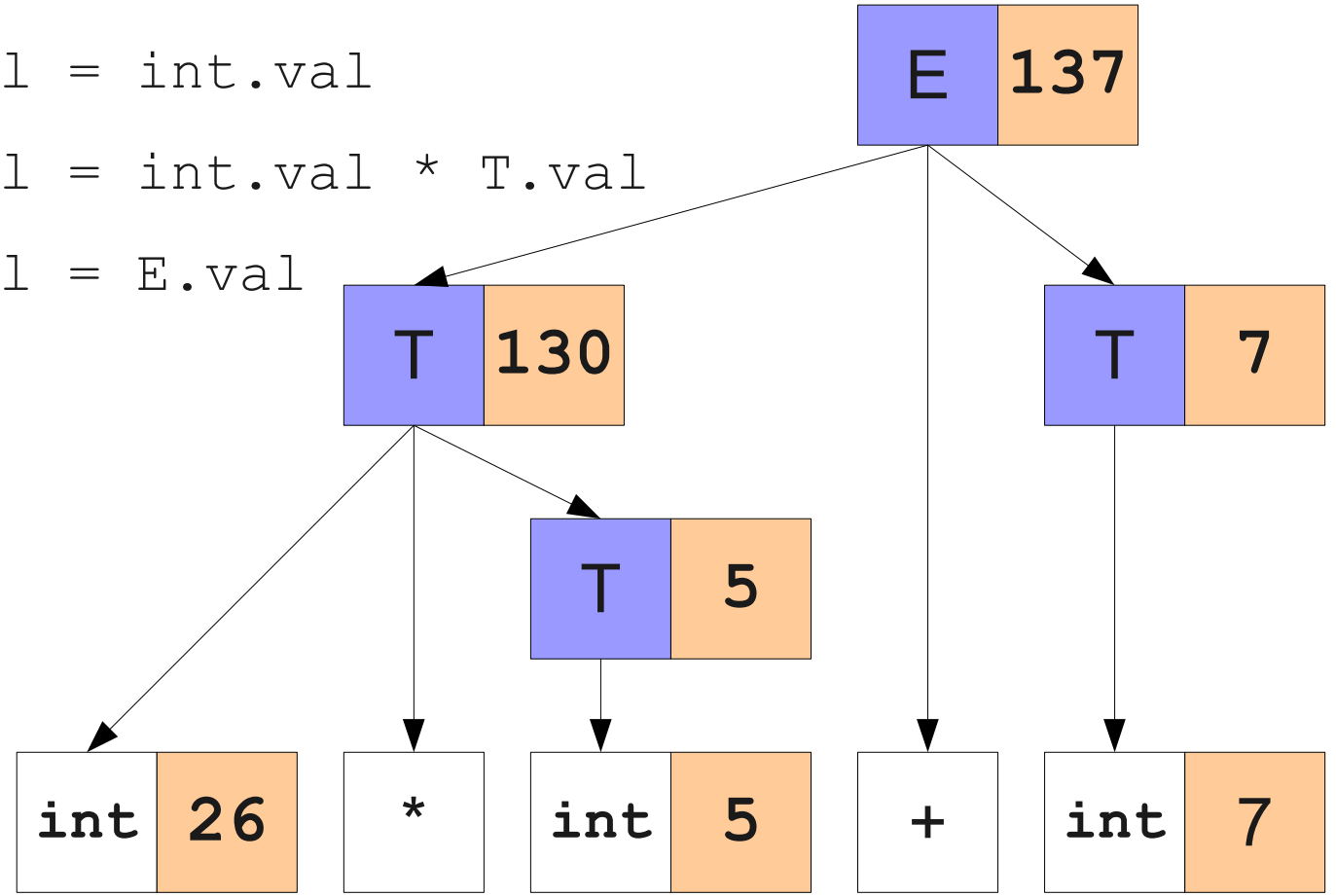
Simple Semantic Actions

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



Simple Semantic Actions

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



Semantic Actions to Build ASTs

$R \rightarrow S$	`R.ast = S.ast;`
$R \rightarrow R$ "|" S	`R1.ast = new Or(R2.ast, S.ast);`
$S \rightarrow T$	`S.ast = T.ast;`
$S \rightarrow ST$	`S1.ast = new Concat(S2.ast, T.ast);`
$T \rightarrow U$	`T.ast = U.ast;`
$T \rightarrow T^*$	`T1.ast = new Star(T2.ast);`
$U \rightarrow$ "a"	`U.ast = SingleChar('a');`
$U \rightarrow$ "ε"	`U.ast = new Epsilon();`
$U \rightarrow (R)$	`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** (**CFG**s).
- 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** (**AST**s) contain an abstract representation of a program's syntax.
- **Semantic actions** associated with productions can be used to build ASTs.

Next Time

- Top-Down Parsing
  - Recursive Descent
  - LL(1)