Compilation 2024

Parsing

Amin Timany
timany@cs.au.dk

Revised from slides by Aslan Askarov, E. Ernst, J. Midtgaard, and M. I. Schwartzbach

Where do ASTs come from?

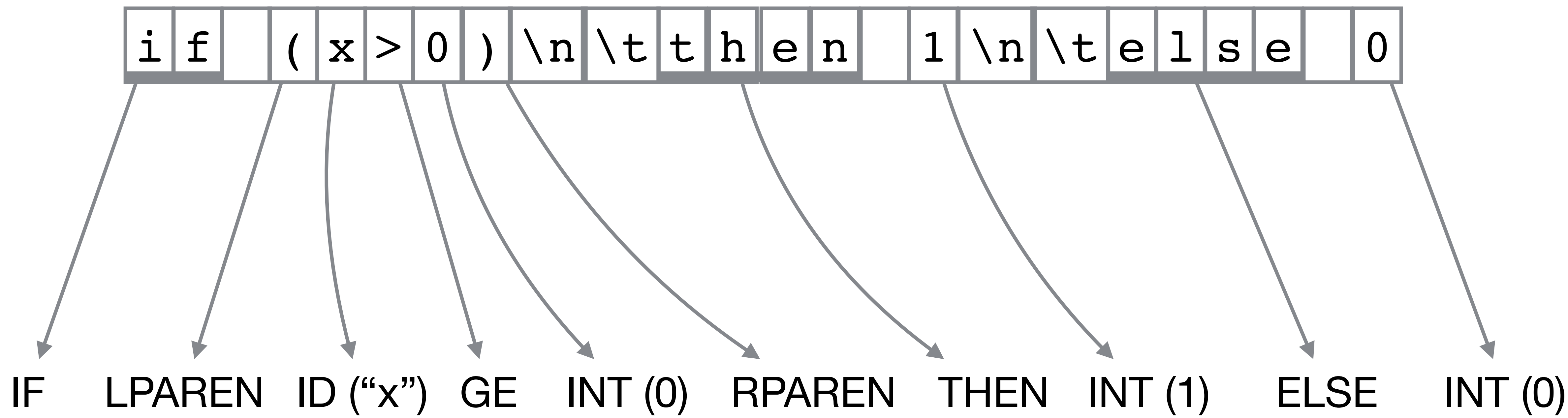
- So far we have worked with ASTs as input to our compilers
- Programmers write source code (sequence of characters)
- Where do ASTs come from?

Lexical Analysis & Parsing

Parsing

High-level source code
Stream of characters
Lexing
Stream of *tokens*
Parsing

Recall lexical analysis: from streams of characters to streams of tokens



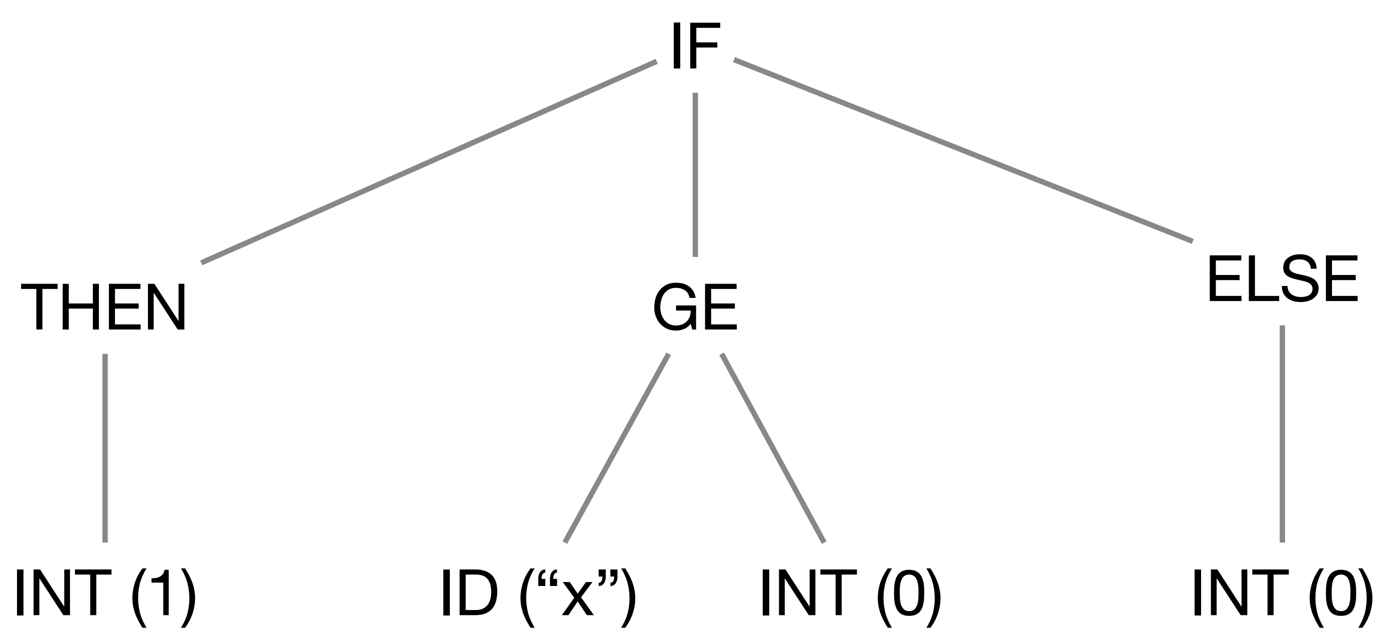
Tool: regular expressions

Parsing: from streams of tokens to trees

IF LPAREN ID (“x”) GE INT (0) RPAREN THEN INT (1) ELSE INT (0)

Parsing: from streams of tokens to trees

IF LPAREN ID (“x”) GE INT (0) RPAREN THEN INT (1) ELSE INT (0)



Tool: context-free grammars

Formal definition of CFG

A context-free grammar (CFG) is a 4-tuple $G = (V, \Sigma, S, P)$

- V is a finite set of *nonterminal* symbols
- Σ is an alphabet of *terminal* symbols and $V \cap \Sigma = \varnothing$
- $S \in V$ is a *start* symbol
- P is a finite set of *productions* of the form $A \to \alpha$, where
  - A is a nonterminal and α is a possibly empty string of nonterminals or terminals
    - formally: $A \in V$, and $\alpha \in (V \cup \Sigma)^*$

Context-Free Grammars by example

A CFG contains 4 components

$$
\begin{aligned}
S &\rightarrow S + T \\
&\mid S - T \\
&\mid T \\
T &\rightarrow T * F \\
&\mid T / F \\
&\mid F \\
F &\rightarrow x \\
&\mid y \\
&\mid z \\
&\mid (S)
\end{aligned}
$$

Context-Free Grammars by example

A CFG contains 4 components

1.Terminals – think tokens

$$
\begin{aligned}
S &\rightarrow S + T \\
&\mid S - T \\
&\mid T \\
T &\rightarrow T * F \\
&\mid T / F \\
&\mid F \\
F &\rightarrow x \\
&\mid y \\
&\mid z \\
&\mid (S)
\end{aligned}
$$

Context-Free Grammars by example

A CFG contains 4 components

1. *Terminals – think tokens*

$$
\begin{aligned}
S &\rightarrow S\ \boxed{+}\ T \\
&\quad |\ S\ \boxed{-}\ T \\
&\quad |\ T \\
T &\rightarrow T\ \boxed{*}\ F \\
&\quad |\ T\ \boxed{/}\ F \\
&\quad |\ F \\
F &\rightarrow \boxed{x} \\
&\quad |\ \boxed{y} \\
&\quad |\ \boxed{z} \\
&\quad |\ \boxed{(}\ S\ \boxed{)}
\end{aligned}
$$

Context-Free Grammars by example

A CFG contains 4 components

1. Terminals – think tokens
2. *Nonterminals – impose the hierarchical structure*

$$
\begin{aligned}
S &\rightarrow S + T \\
&\mid S - T \\
&\mid T \\
T &\rightarrow T * F \\
&\mid T / F \\
&\mid F \\
F &\rightarrow x \\
&\mid y \\
&\mid z \\
&\mid (S)
\end{aligned}
$$

Context-Free Grammars by example

A CFG contains 4 components

1. Terminals – think tokens
2. *Nonterminals – impose the hierarchical structure*

$$
\begin{aligned}
\boxed{S} &\rightarrow S + T \\
&\mid S - T \\
&\mid T \\
\boxed{T} &\rightarrow T * F \\
&\mid T / F \\
&\mid F \\
\boxed{F} &\rightarrow x \\
&\mid y \\
&\mid z \\
&\mid (S)
\end{aligned}
$$

Context-Free Grammars by example

A CFG contains 4 components

1. Terminals – think tokens
2. *Nonterminals – impose the hierarchical structure*

$$
\begin{aligned}
S &\rightarrow S + T \\
  &\mid\ S - T \\
  &\mid\ T \\
T &\rightarrow T * F \\
  &\mid\ T \,/\, F \\
  &\mid\ F \\
F &\rightarrow x \\
  &\mid\ y \\
  &\mid\ z \\
  &\mid\ (\ S\ )
\end{aligned}
$$

Context-Free Grammars by example

A CFG contains 4 components

1. Terminals – think tokens
2. Nonterminals – impose the hierarchical structure
3. *Start symbol*

$$
\begin{aligned}
S &\rightarrow S + T \\
  &\mid S - T \\
  &\mid T \\
T &\rightarrow T * F \\
  &\mid T / F \\
  &\mid F \\
F &\rightarrow x \\
  &\mid y \\
  &\mid z \\
  &\mid (S)
\end{aligned}
$$

Context-Free Grammars by example

A CFG contains 4 components

1. Terminals – think tokens
2. Nonterminals – impose the hierarchical structure
3. Start symbol
4. *Production rules*

$$
\begin{aligned}
S &\rightarrow S + T \\
  &\mid\ S - T \\
  &\mid\ T \\
T &\rightarrow T * F \\
  &\mid\ T / F \\
  &\mid\ F \\
F &\rightarrow x \\
  &\mid\ y \\
  &\mid\ z \\
  &\mid\ (S)
\end{aligned}
$$

Context-Free Grammars by example

A CFG contains 4 components

1. Terminals – think tokens
2. Nonterminals – impose the hierarchical structure
3. Start symbol
4. *Production rules*

$$
\begin{aligned}
S &\to S + T \\
&\mid S - T \\
&\mid T \\
T &\to T * F \\
&\mid T \,/\, F \\
&\mid F \\
F &\to x \\
&\mid y \\
&\mid z \\
&\mid (S)
\end{aligned}
$$

Context-Free Grammars by example

att: notation shortcut: left and right grammars are the same

A CFG contains 4 components

1. Terminals – think tokens
2. Nonterminals – impose the hierarchical structure
3. Start symbol
4. Production rules

$$
\begin{array}{lll}
S & \rightarrow & S + T \\
  & | & S - T \\
  & | & T \\
T & \rightarrow & T * F \\
  & | & T / F \\
  & | & F \\
F & \rightarrow & x \\
  & | & y \\
  & | & z \\
  & | & (S)
\end{array}
\qquad\qquad
\begin{array}{lll}
S & \rightarrow & S + T \\
S & \rightarrow & S - T \\
S & \rightarrow & T \\
T & \rightarrow & T * F \\
T & \rightarrow & T / F \\
T & \rightarrow & F \\
F & \rightarrow & x \\
F & \rightarrow & y \\
F & \rightarrow & z \\
F & \rightarrow & (S)
\end{array}
$$

Context-Free Grammars by example

att: notation shortcut: left and right grammars are the same

A CFG contains 4 components

1. Terminals – think tokens
2. Nonterminals – impose the hierarchical structure
3. Start symbol
4. Production rules

$$
\begin{aligned}
S &\rightarrow S + T \\
&\mid\ S - T \\
&\mid\ T \\
T &\rightarrow T * F \\
&\mid\ T \,/\, F \\
&\mid\ F \\
F &\rightarrow x \\
&\mid\ y \\
&\mid\ z \\
&\mid\ (S)
\end{aligned}
\qquad\qquad
\begin{aligned}
S &\rightarrow S + T \\
S &\rightarrow S - T \\
S &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow T \,/\, F \\
T &\rightarrow F \\
F &\rightarrow x \\
F &\rightarrow y \\
F &\rightarrow z \\
F &\rightarrow (S)
\end{aligned}
$$

Context-Free Grammars by example

A CFG contains 4 components

1. Terminals – think tokens
2. Nonterminals – impose the hierarchical structure
3. Start symbol
4. Production rules
   a) head of the production – left side

$$
\begin{aligned}
S &\rightarrow S + T \\
  &\mid S - T \\
  &\mid T \\
T &\rightarrow T * F \\
  &\mid T \,/\, F \\
  &\mid F \\
F &\rightarrow x \\
  &\mid y \\
  &\mid z \\
  &\mid (S)
\end{aligned}
$$

Context-Free Grammars by example

A CFG contains 4 components

1. Terminals – think tokens
2. Nonterminals – impose the hierarchical structure
3. Start symbol
4. Production rules
   a) head of the production – left side
   b) body of the production – right side

$$
\begin{aligned}
S &\rightarrow S + T \\
  &\mid S - T \\
  &\mid T \\
T &\rightarrow T * F \\
  &\mid T / F \\
  &\mid F \\
F &\rightarrow x \\
  &\mid y \\
  &\mid z \\
  &\mid (S)
\end{aligned}
$$

Example: algebraic expressions

- A CFG $G = (V, \Sigma, S, P)$
- $V = \{S\}$ // only one nonterminal
- $\Sigma = \{\, + \,, - \,, * \,, /, (,), x, y, z\}$ // a set of terminals
- $P =$ // productions set

$$\{S \rightarrow S + S, S \rightarrow S - S, S \rightarrow S * S,$$
$$S \rightarrow S/S, S \rightarrow (S), S \rightarrow x, S \rightarrow y, S \rightarrow z\}$$

alt notation:
S → S + S | S - S | S * S
| S / S | (S) | x | y | z

- Example: $x * y + z - (z/y + x)$

Derivations

- We write $\alpha \Rightarrow \beta$ when
  - $\alpha, \beta \in (V \cup \Sigma)^*$
  - $\alpha = \alpha_1 A \alpha_2$, where $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$ and $A \in V$
  - $\beta = \alpha_1 \gamma \alpha_2$
  - the grammar contains production $A \to \gamma$
- Here, "$\Rightarrow$" denotes a single derivation step when a nonterminal is rewritten according to some production
  - "$\Rightarrow$" is a relation on set $(V \cup \Sigma)^*$

The language of a CFG

- Define $\Rightarrow^*$ as the reflexive transitive closure of $\Rightarrow$

$$\alpha \Rightarrow^* \beta \text{ when } \underbrace{\alpha \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \beta}_{\text{0 or more steps}}$$

- Define the *language* of G as

$$\mathbf{L}(G) = \{ x \in \Sigma^* \mid S \Rightarrow^* x \}$$

  set of all (possibly empty) terminal strings that can be derived from start symbol S in zero or more steps
- We say that a language $L \subseteq \Sigma^*$ is *context-free* if there is a CFG G such that $\mathbf{L}(G) = L$

Example 1

Language $L = \{ a^n b^n \mid n \geq 0 \}$ is described by CFG $G = (V, \Sigma, S, P)$ where

- $V = \{ S \}$
- $\Sigma = \{ a, b \}$
- $P = S \rightarrow aSb \mid \varepsilon$

- That is, **L** $(G) = L$
- Example derivations:

  $S \Rightarrow^* \varepsilon$ $\qquad\qquad$ $S \Rightarrow^* aaabbb$

Example 1

Language $L = \{ a^n b^n \mid n \geq 0 \}$ **is described by CFG** $G = (V, \Sigma, S, P)$ **where**

- $V = \{ S \}$
- $\Sigma = \{ a, b \}$
- $P = S \rightarrow aSb \mid \varepsilon$

- That is, $\mathbf{L}(G) = L$
- Example derivations:

$$S \Rightarrow^* \varepsilon$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

$$S \Rightarrow^* aaabbb$$

Example 2

- Language pal = $\{ x \in \{0,1\}^* \mid x = \text{reverse}(x) \}$ is described by a CFG $G = (V, \Sigma, S, P)$ where

- $V = \{ S \}$
- $\Sigma = \{ 0, 1 \}$
- $P = S \to \varepsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1$

- Example derivations:

$$S \Rightarrow^* \varepsilon \qquad S \Rightarrow^* 100001$$

Example 2

- Language pal = $\{ x \in \{0,1\}^* \mid x = \text{reverse}(x) \}$ is described by a CFG $G = (V, \Sigma, S, P)$ where

- $V = \{ S \}$
- $\Sigma = \{ 0, 1 \}$
- $P = S \rightarrow \varepsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1$

$S \Rightarrow 1S1 \Rightarrow 10S01 \Rightarrow 100S001 \Rightarrow 100001$

- Example derivations:

$$S \Rightarrow^* \varepsilon \qquad S \Rightarrow^* 100001$$

Why “context-free”?

- Because it holds that $\alpha_1 A \alpha_2 \Rightarrow \alpha_1 \gamma \alpha_2$

  whenever the grammar contains the production $A \rightarrow \gamma$

- In other words, γ may be substituted for A independently of the context (α_1 and α_2)

Why "context-free"?

context

- Because it holds that $\alpha_1 \; A \; \alpha_2 \Rightarrow \alpha_1 \; \gamma \; \alpha_2$

  whenever the grammar contains the production $A \rightarrow \gamma$

- In other words, γ may be substituted for A independently of the context (α_1 and α_2)

Exercise

$S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid x \mid y \mid z$

Write a derivation for x * y + z and draw the corresponding parse tree

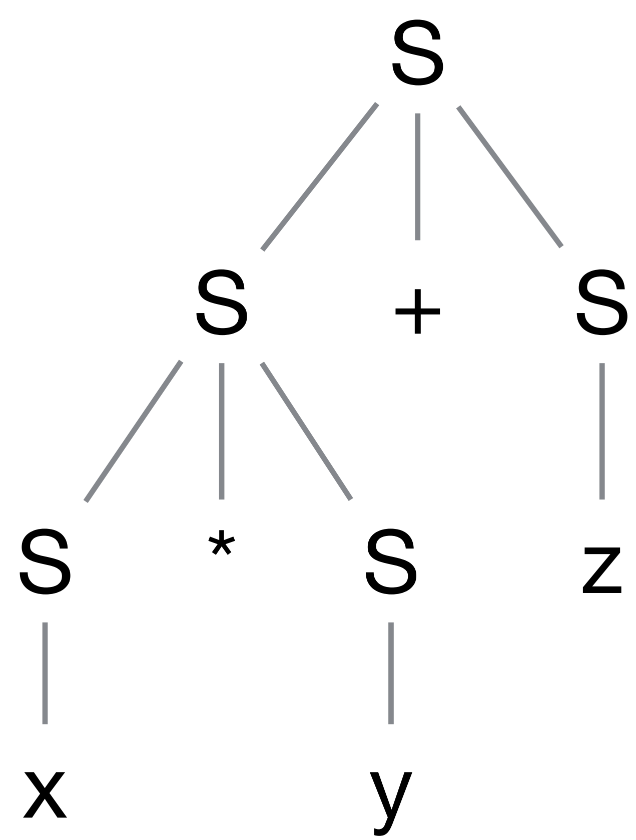
Given grammar $S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid x \mid y \mid z$

Consider three possible derivations of string $x * y + z$

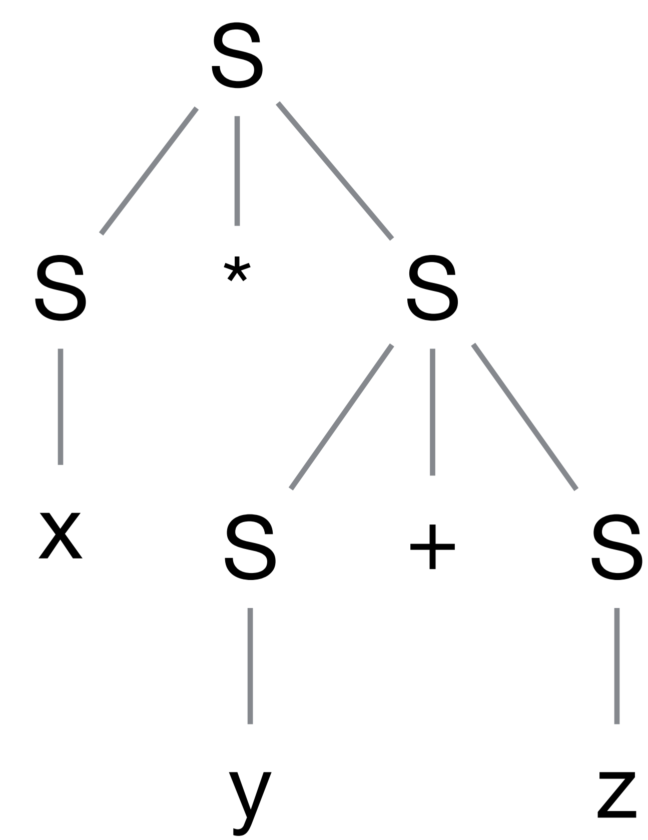
1. $S \Rightarrow S + S \Rightarrow S * S + S \Rightarrow x * S + S \Rightarrow x * y + S \Rightarrow x * y + z$
2. $S \Rightarrow S + S \Rightarrow S + z \Rightarrow S * S + z \Rightarrow S * y + z \Rightarrow x * y + z$
3. $S \Rightarrow S * S \Rightarrow S * S + S \Rightarrow x * S + S \Rightarrow x * y + S \Rightarrow x * y + z$

Derivation tree shows the *structure* of a derivation but not the detailed order

Derivations 1 and 2



Derivation 3



The goal of the parser is to find a derivation tree for a given string

Ambiguous CFGs

- A CFG G is *ambiguous* if there exists a string $x \in \mathbf{L}(G)$ with more than one derivation tree
- Example: our CFG for algebraic expressions is ambiguous
- The decision problem of whether an arbitrary CFG is ambiguous is *undecidable*

Removing ambiguity by rewriting the grammar

- The ambiguous grammar

  $S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid x \mid y \mid z$

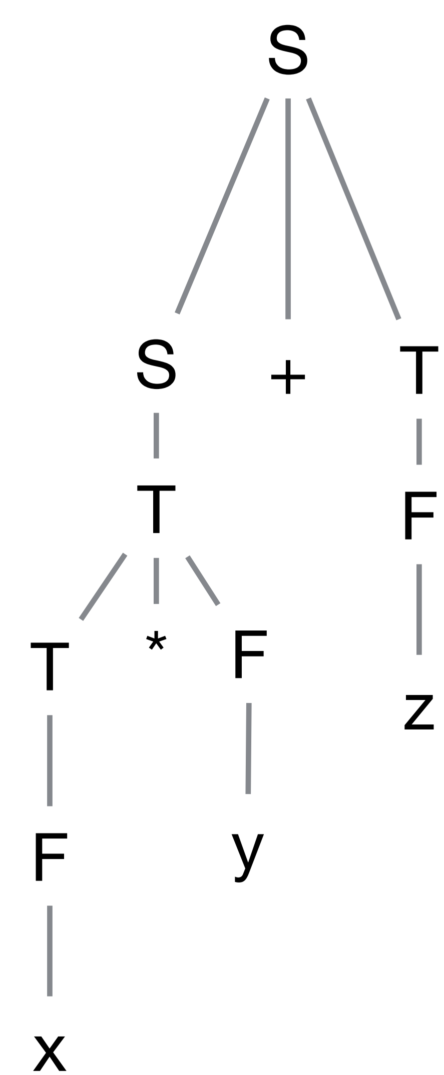
  may be rewritten to become unambiguous

  $S \rightarrow S + T \mid S - T \mid T$
  $T \rightarrow T * F \mid T / F \mid F$
  $F \rightarrow x \mid y \mid z \mid (S)$

- This imposes an *operator precedence & associativity*

Unambiguous parsing

- String x * y + z now only admits a single parse tree

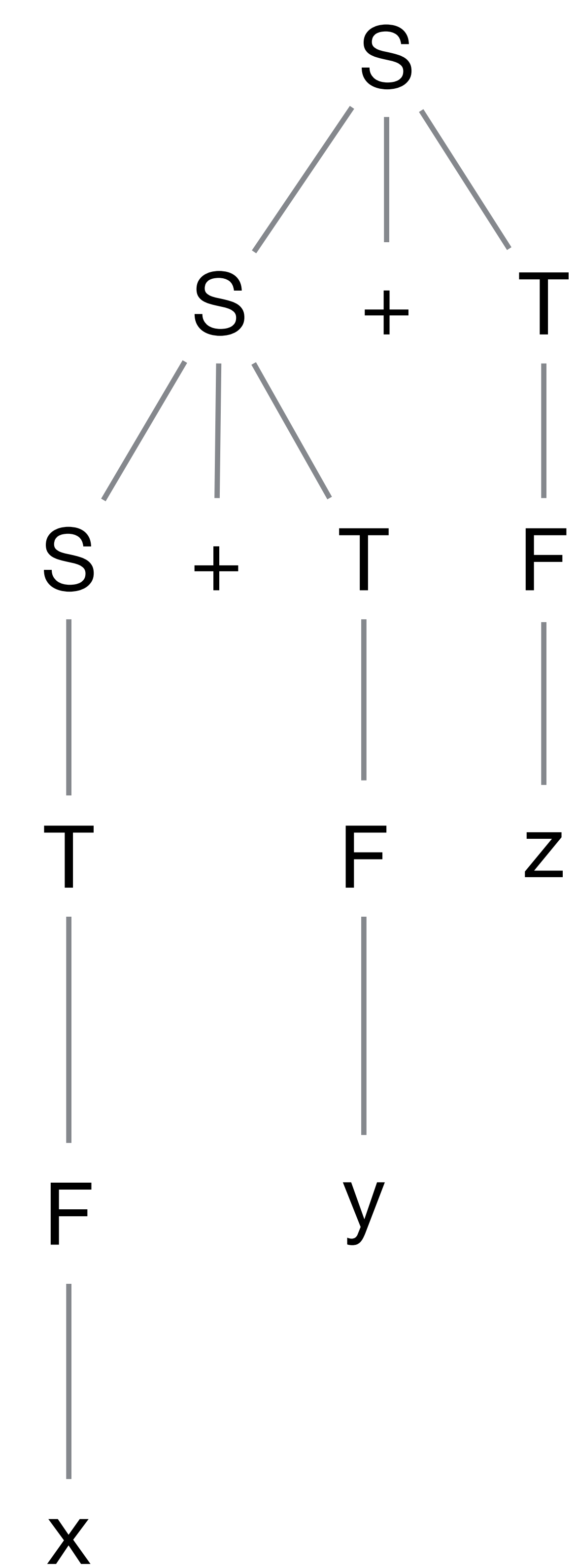


Q: what is the associativity and precedence in this grammar?

$$
\begin{aligned}
S &\rightarrow S + T \mid S - T \mid T \\
T &\rightarrow T * F \mid T / F \mid F \\
F &\rightarrow x \mid y \mid z \mid (S)
\end{aligned}
$$

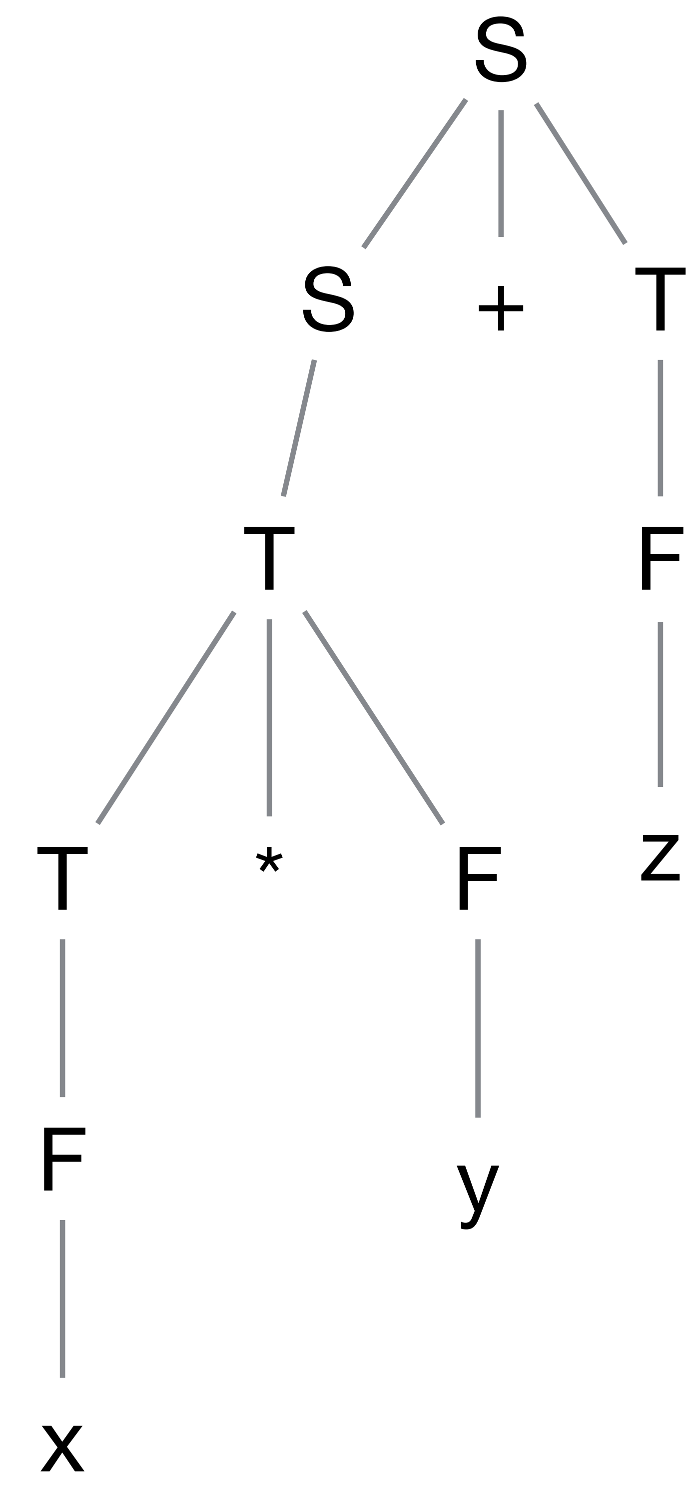
Answer

- left associative: x+y+z => (x+y)+z



Answer

- * and / are higher than + and - x*y+z =(x*y)+z



Including end-of-file

Common transformation: make EOF (denoted by $) explicit.
Adding a fresh nonterminal S’ and rule S’ → S$

Example

$S \to S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid x \mid y \mid z$

becomes

$S' \to S\$$

$S \to S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid x \mid y \mid z$

Parsing techniques in compilers

We have two main parsing techniques in compilers

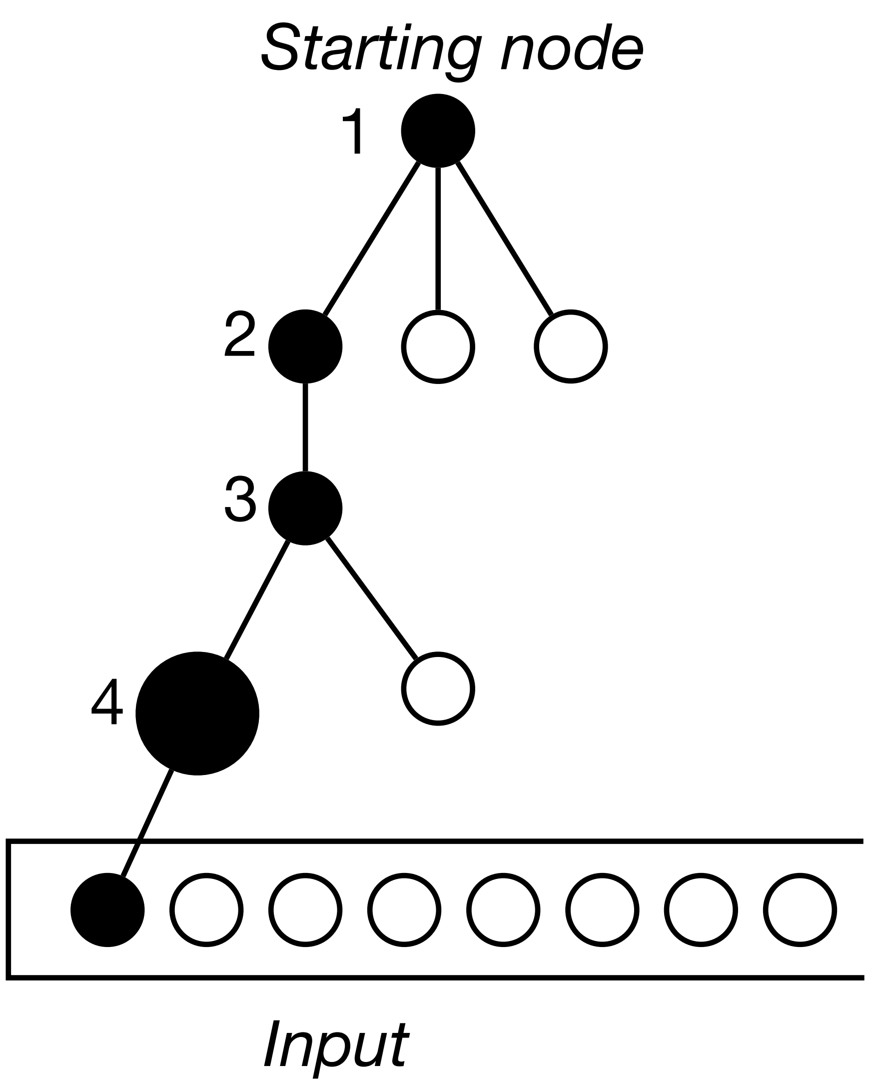
1. Top-down (aka predictive) parsing
   - often can be used for writing parsers by hand, also amenable to tool support
2. Bottom-up
   - only tool-based

[Fig source: D. Grune et al. Modern Compiler Design, 2nd ed.]

Parsing techniques in compilers

We have two main parsing techniques in compilers

1. Top-down (aka predictive) parsing
   - often can be used for writing parsers by hand, also amenable to tool support
2. Bottom-up
   - only tool-based



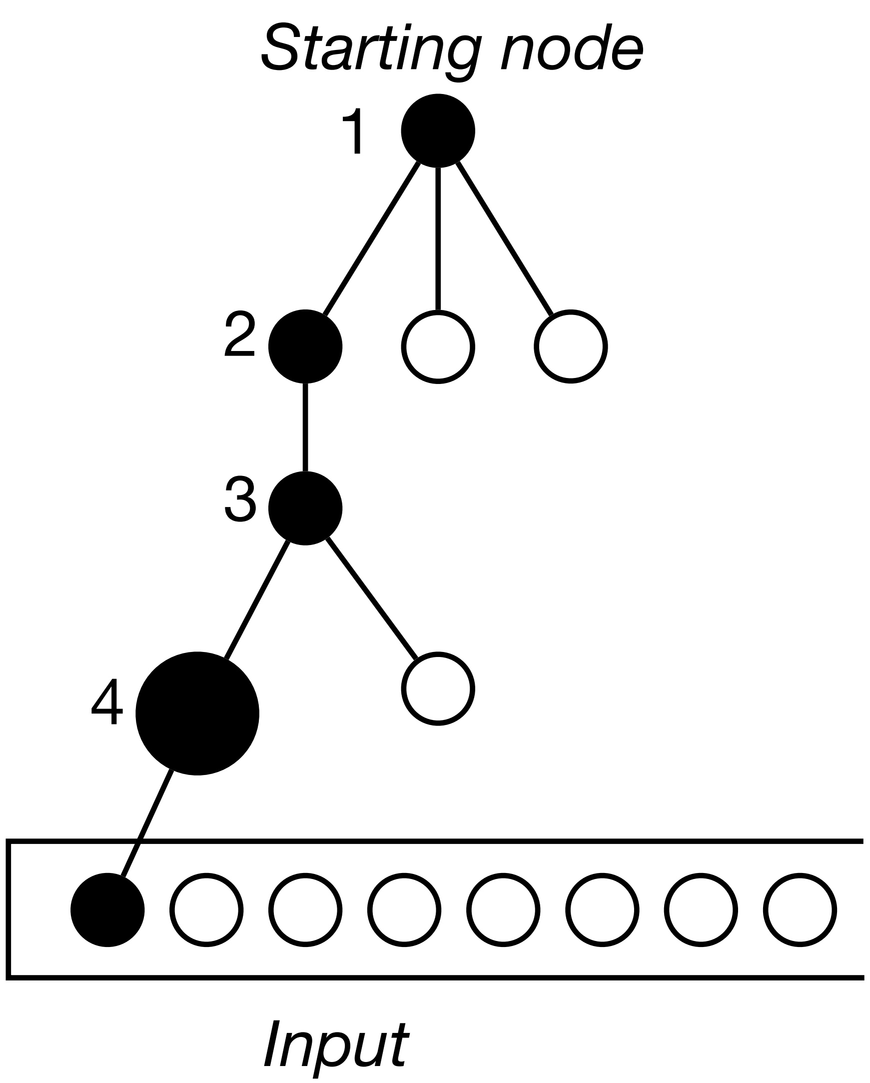
Top-down parser

[Fig source: D. Grune et al. Modern Compiler Design, 2nd ed.]

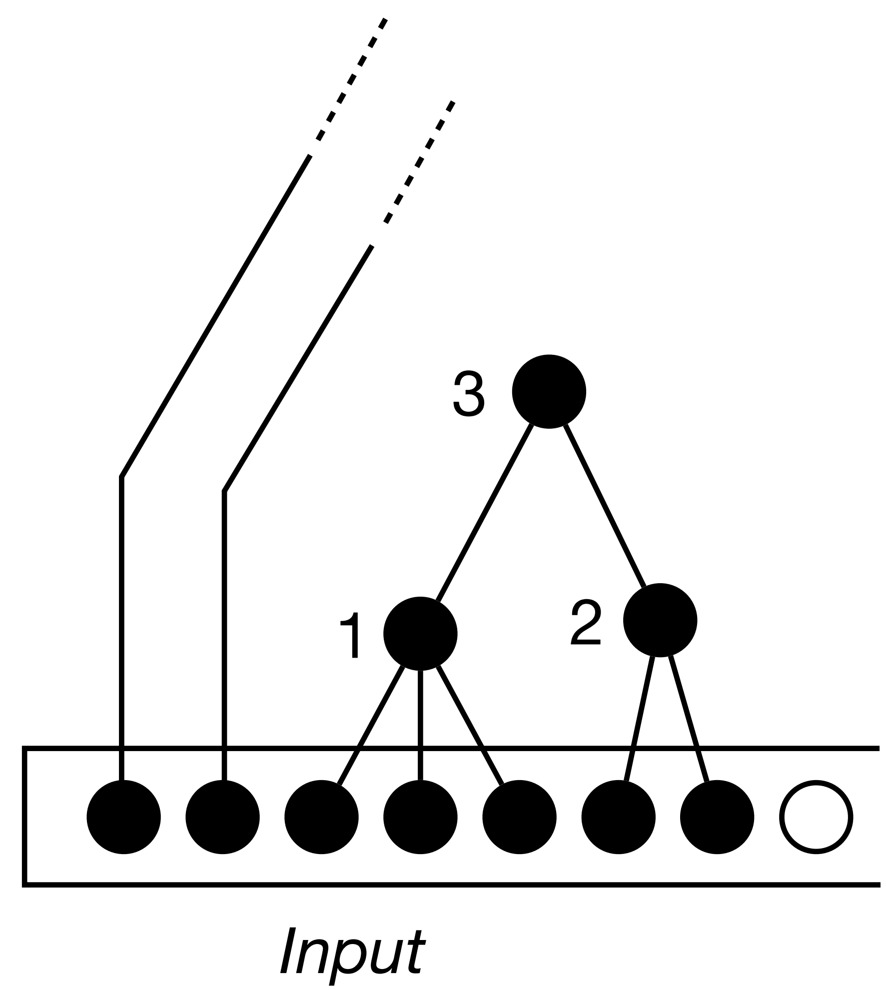
Parsing techniques in compilers

We have two main parsing techniques in compilers

1. Top-down (aka predictive) parsing
   - often can be used for writing parsers by hand, also amenable to tool support
2. Bottom-up
   - only tool-based



Top-down parser



Bottom-up parser

[Fig source: D. Grune et al. Modern Compiler Design, 2nd ed.]

Predictive parsing, i.e., LL(k)

- Basic idea: top-down process, leftmost derivation
- “see what’s coming”: look at k tokens, guess what made it
- Example grammar:

$$
\begin{aligned}
S &\rightarrow \text{if } E \text{ then } S \text{ else } S \\
S &\rightarrow \text{begin } S\ L \\
S &\rightarrow \text{print } E \\
L &\rightarrow \text{end} \\
L &\rightarrow \text{; } S\ L \\
E &\rightarrow \text{num} = \text{num}
\end{aligned}
$$

- Example above is LL(1)

Recursive descent parser

One function per nonterminal, one case per rule

```
type token = IF | THEN | ELSE | BEGIN | END | PRINT | SEMI | NUM |EQ
let tok = ref (getToken())
let advance() = (tok := getToken())
let eat t = if (!tok=t) then advance() else error()

let S() = match !tok with
            IF -> (eat(IF); E(); eat(THEN); S(); eat(ELSE); S())
          | BEGIN -> (eat(BEGIN); S(); L())
          | PRINT -> (eat(PRINT); E())
          | _ -> error ()
and L() = match !tok with
            END -> (eat(END))
          | SEMI -> (eat(SEMI); S(); L())
          | _ -> error ()
and E() = (eat(NUM); eat(EQ); eat(NUM))
```

S → if E then S else S
S → begin S L
S → print E
L → end
L → ; S L
E → num = num

Limitations of recursive descent

- “One function per nonterminal, one case per rule” only works when we can choose the right case
- May immediately break down, e.g.:
  $S \rightarrow S + x \mid S - x \mid x$

Systematic analysis of limitation

- Introduce *sets of terminals* FIRST(γ), FOLLOW (X)
- Intuition:
  - FIRST (α) : set of terminals that begin strings derived from α
  - FOLLOW(X): set of terminals a that can appear immediately to the right of X in some derivable string, e.g., $S \Rightarrow^* \alpha X a \beta$
- Let nullable(X) be true when X can derive empty string ε
- Define FIRST and FOLLOW to be the smallest sets such that

```
forall terminals t, FIRST(t) = {t}

forall productions X → Y1..Yk
  if nullable(Y1),…,nullable(Yk) then nullable(X) = true
  for i∈1..k, j∈i+1..k
    if nullable(Y1),…,nullable(Yi-1) (or i=1) then FIRST(Yi) ⊆ FIRST(X)
    if nullable(Yi+1),…,nullable(Yk) (or i=k) then FOLLOW(X) ⊆ FOLLOW(Yi)
    if nullable(Yi+1),…,nullable(Yj-1) (or i+1=j) then FIRST(Yj) ⊆ FOLLOW(Yi)
```

- Computed by fixpoint iteration

Example

Grammar:

S → if E then S else S
S → begin S L
S → print E
L → end
L → ; S L
E → num = num

- FIRST (α) : set of terminals that begin strings derived from α
- FOLLOW(X): set of terminals a that can appear immediately to the right of X in some derivable string, e.g., S ⇒* αXaβ
- Let nullable(X) be true when X can derive empty string ε

```
forall terminals t, FIRST(t) = {t}

forall productions X → Y1..Yk
  if nullable(Y1),…,nullable(Yk) then nullable(X) = true
  for i∈1..k, j∈i+1..k
    if nullable(Y1),…,nullable(Yi-1) (or i=1) then FIRST(Yi) ⊆ FIRST(X)
    if nullable(Yi+1),…,nullable(Yk) (or i=k) then FOLLOW(X) ⊆ FOLLOW(Yi)
    if nullable(Yi+1),…,nullable(Yj-1) (or i+1=j) then FIRST(Yj) ⊆ FOLLOW(Yi)
```

Example

Grammar:

S → if E then S else S
S → begin S L
S → print E
L → end
L → ; S L
E → num = num

- FIRST (α) : set of terminals that begin strings derived from α
- FOLLOW(X): set of terminals a that can appear immediately to the right of X in some derivable string, e.g., $S \Rightarrow^* \alpha X a \beta$
- Let nullable(X) be true when X can derive empty string ε

Nonterminal	Nullable?	First set	Follow set
S		if, begin, print	else, end, ;, $
L		end, ;	else, end, ;, $
E		num	then, else, end, ; $

Systematic analysis of limitation

- Use FIRST(γ) for every rule X ➞ γ
- In *parsing table*, let cell (X, t) include X ➞ γ iff
  t ∈ FIRST (γ) or, nullable(γ) and t ∈ Follow (X)
- Declare conflict if there is a cell with > 1 rule
- Tricks: left recursion elimination, left factoring

Systematic analysis of limitation

- Use FIRST(γ) for every rule X → γ
- In *parsing table*, let cell (X, t) include X → γ iff
  t ∈ FIRST (γ) or, nullable(γ) and t ∈ Follow (X)
- Declare conflict if there is a cell with > 1 rule
- Tricks: left recursion elimination, left factoring

Example parsing table for our grammar from the previous slide

	if	then	else	begin	print	end	;	num	=	$
S	$S \to$ **if** E **then** S **else** S			$S \to$ **begin** $S\ L$	$S \to$ **print** E					
L						$L \to$ **end**	$L \to$ **;** $S\ L$			
E								$E \to$ **num = num**		

Systematic analysis of limitation

- Use FIRST(γ) for every rule X → γ
- In *parsing table*, let cell (X, t) include X → γ iff t ∈ FIRST (γ) or, nullable(γ) and t ∈ Follow (X)
- Declare conflict if there is a cell with > 1 rule
- Tricks: left recursion elimination, left factoring

Example parsing table for our grammar from the previous slide

	if	**then**	**else**	**begin**	**print**	**end**	**;**	**num**	**=**	**$**
S	$S \rightarrow$ **if** E **then** S **else** S			$S \rightarrow$ **begin** $S\ L$	$S \rightarrow$ **print** E					
L						$L \rightarrow$ **end**	$L \rightarrow$ **;** $S\ L$			
E								$E \rightarrow$ **num = num**		

Example parsing table for the grammar S → S + x | S - x | x

	+	**x**	**-**	**$**
S		$S \rightarrow S$ **+ x** $S \rightarrow S$ **- x** $S \rightarrow$ **x**		

Systematic analysis of limitation

- Use FIRST(γ) for every rule X → γ
- In *parsing table*, let cell (X, t) include X → γ iff t ∈ FIRST (γ) or, nullable(γ) and t ∈ Follow (X)
- Declare conflict if there is a cell with > 1 rule
- Tricks: left recursion elimination, left factoring

Example parsing table for our grammar from the previous slide

	if	**then**	**else**	**begin**	**print**	**end**	**;**	**num**	**=**	**$**
S	$S \rightarrow$ **if** E **then** S **else** S			$S \rightarrow$ **begin** $S\ L$	$S \rightarrow$ **print** E					
L						$L \rightarrow$ **end**	$L \rightarrow$ **;** $S\ L$			
E								$E \rightarrow$ **num = num**		

Example parsing table for the grammar S → S + x | S - x | x

	+	**x**	**-**	**$**
S		$S \rightarrow S$ **+ x** $S \rightarrow S$ **- x** $S \rightarrow$ **x**		

more than 1 rule in a cell!

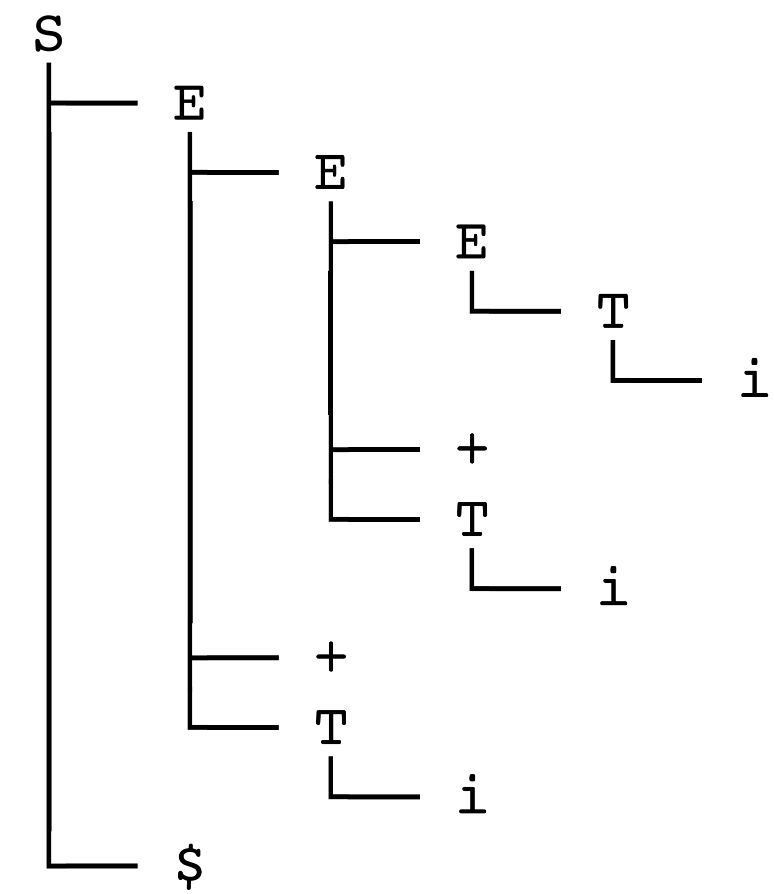
LR parsing

- LR parsers work ***bottom-up*** using a stack to track derivations; perform *rightmost* reduction
- Grammar includes EOF symbol $
- The main task of a bottom-up parser is to find the leftmost node that has not yet been constructed but all of whose children have been constructed.

```
S
├── E
│   ├── E
│   │   ├── E
│   │   │   └── T
│   │   │       └── i
│   │   ├── +
│   │   └── T
│   │       └── i
│   ├── +
│   └── T
│       └── i
└── $
```

LR parsing

- LR parsers work ***bottom-up*** using a stack to track derivations; perform *rightmost* reduction
- Grammar includes EOF symbol $
- The main task of a bottom-up parser is to find the leftmost node that has not yet been constructed but all of whose children have been constructed.



Example

```
S -> E $
E -> T
E -> E + T
T -> i
T -> ( E )
```

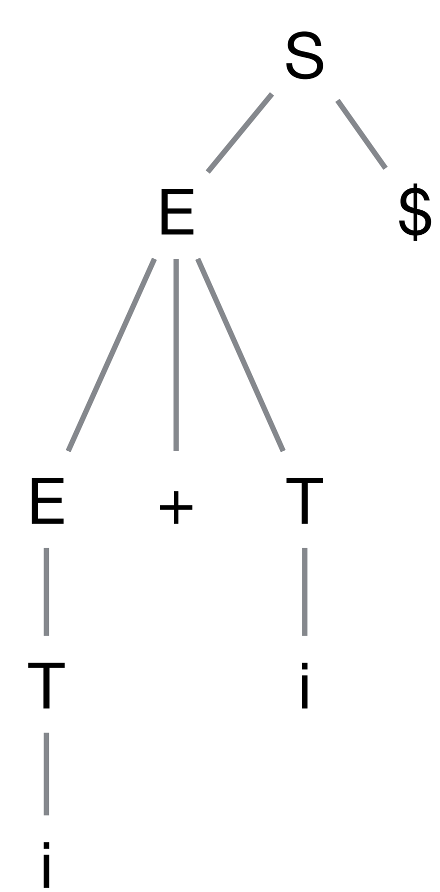
Quick example

- Let us consider the following grammar and string

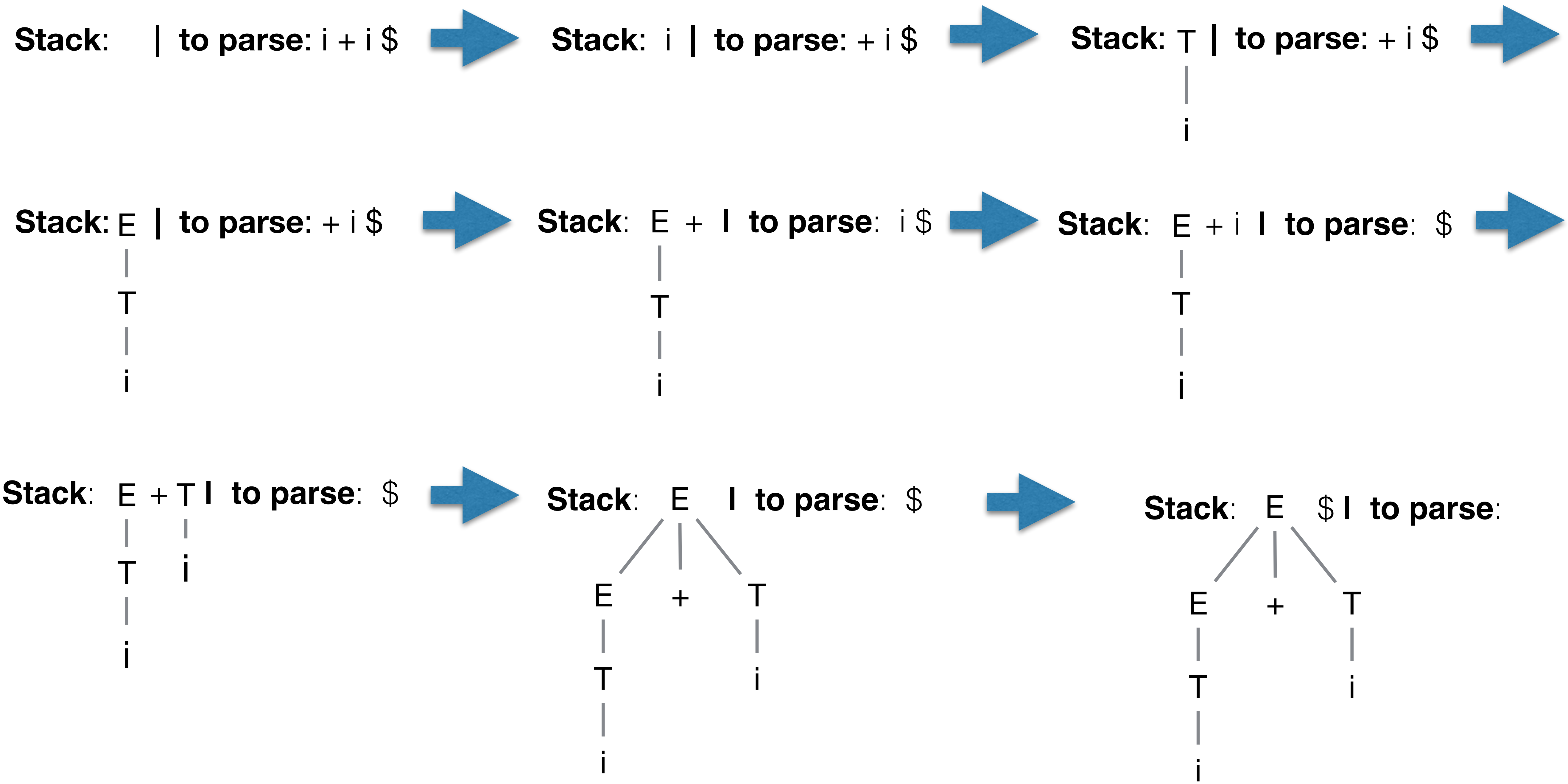
Grammar:

```
S -> E $
E -> T
E -> E + T
T -> i
T -> ( E )
```

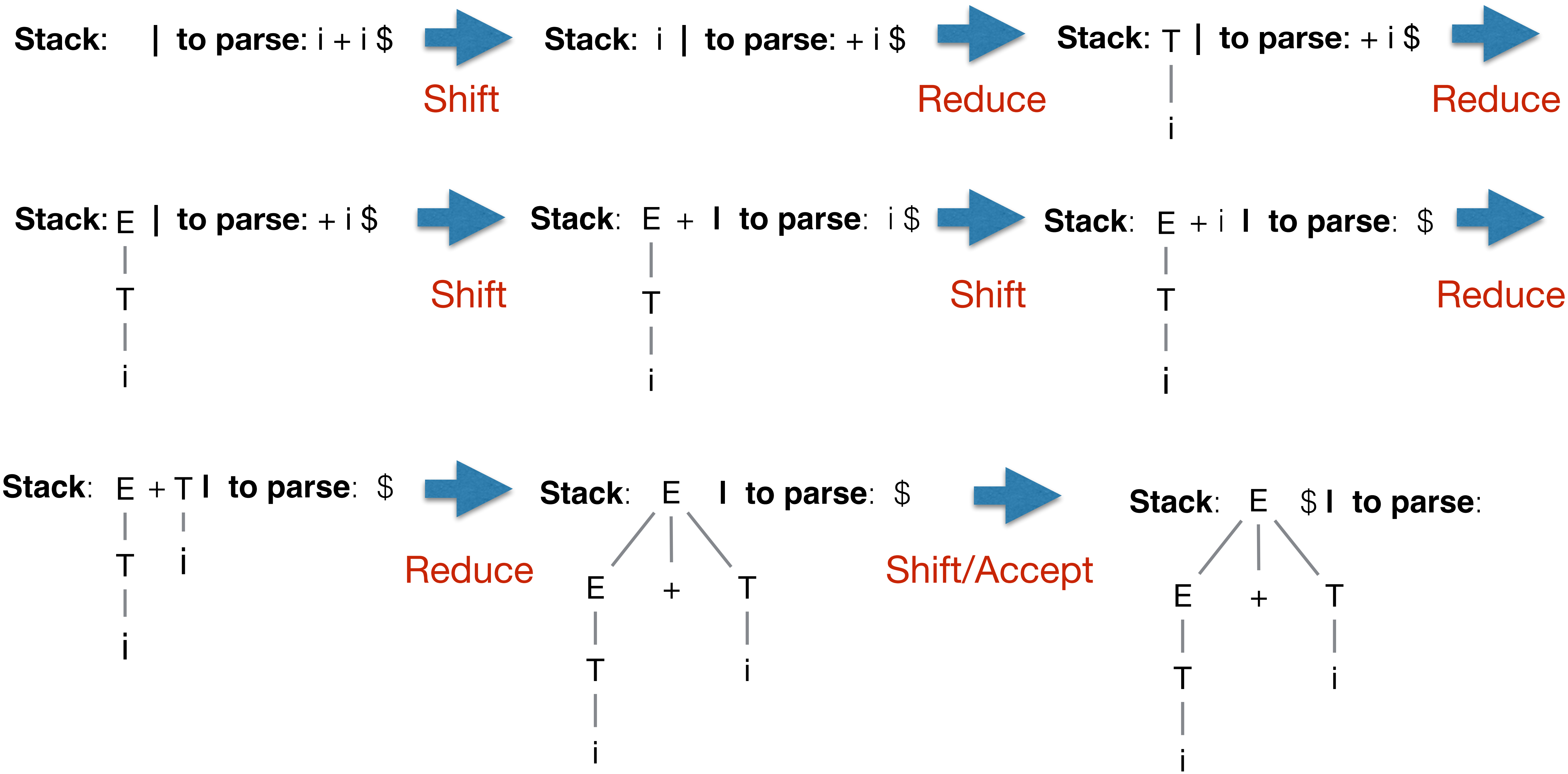
String: i + i $



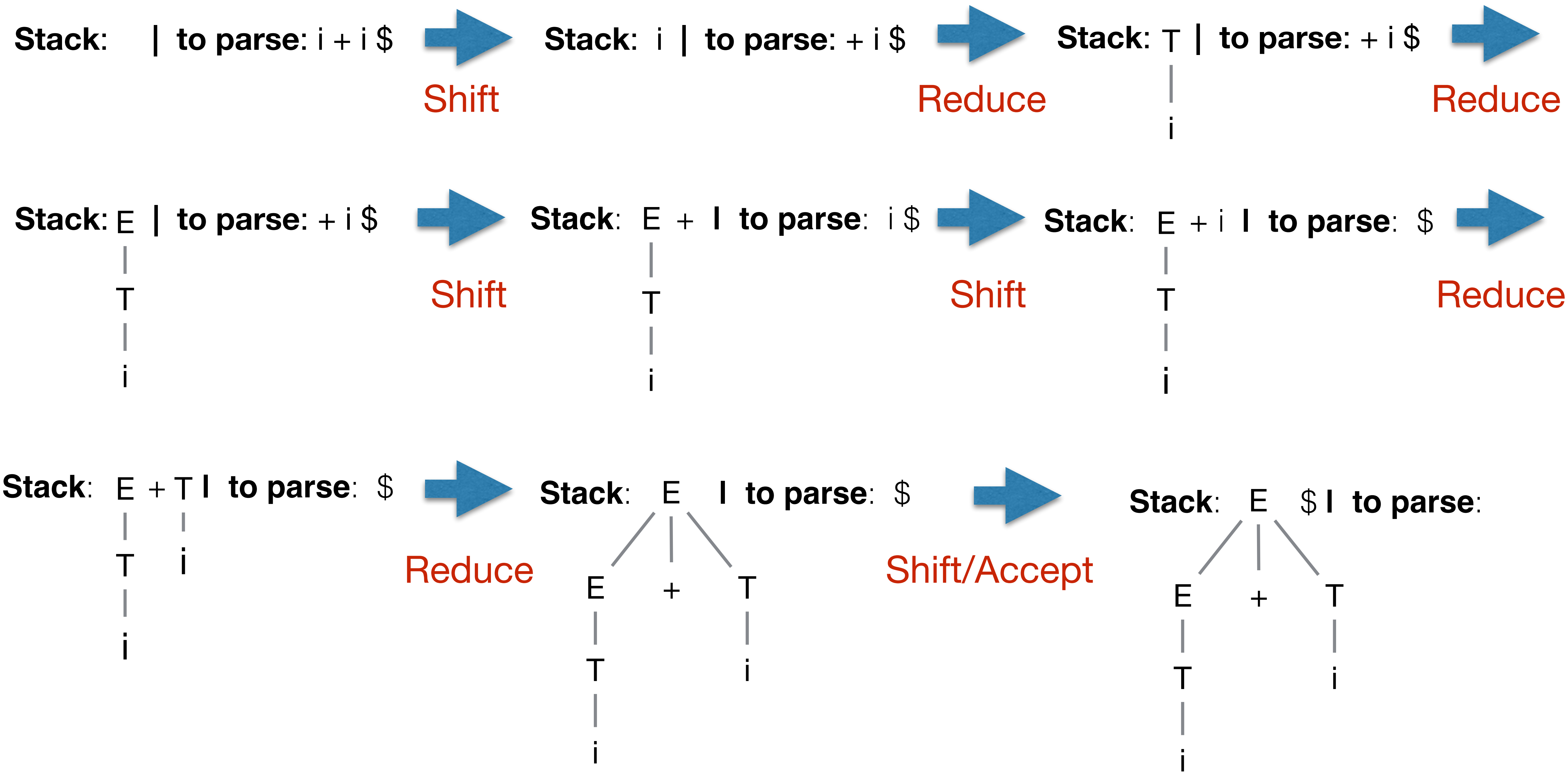
Quick example



Quick example

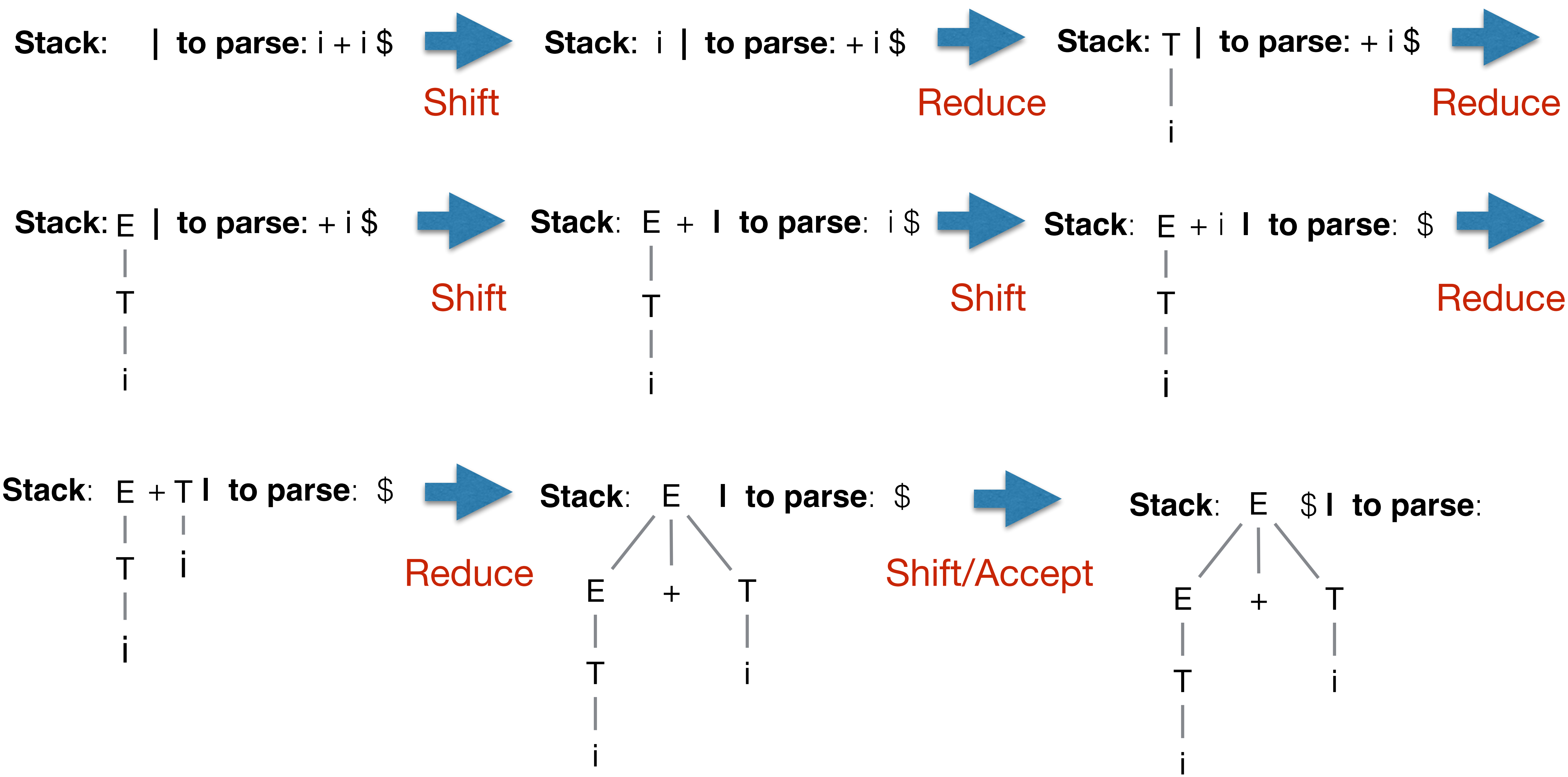


Quick example



How do we know when to shift and when to reduce?

Quick example



How do we know when to shift and when to reduce? LR parsing states and automaton

Grammar containment