بسم الله الرحمن الرحيم

Parsing:

Top-Down Parsing,
Recursive Descent & Predictive Parser & LL(1)

Steps for developing a predictive recursive descent parser

- Input: Context free grammar (non-ambiguous)
- Remove left-recursion
- Left-factoring: postponing the decision time to the time it is needed
- Calculate First and Follow for non-terminals
- Building LL(1) parse table
- Develop the parser

Elimination of the left-recursion

Elimination of Left Recursion (recall)

- Consider the left-recursive grammar

$$S \to S\,\alpha \mid \beta$$

- S generates all strings starting with a β and followed by a number of α

- Can rewrite using right-recursion

$$S \to \beta\, S'$$
$$S' \to \alpha\, S' \mid \varepsilon$$

More Elimination of Left-Recursion (recall)

- In general

$$S \rightarrow S\,\alpha_1 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

$$S \rightarrow \beta_1\,S' \mid \ldots \mid \beta_m\,S'$$

$$S' \rightarrow \alpha_1\,S' \mid \ldots \mid \alpha_n\,S' \mid \varepsilon$$

General Left Recursion (recall)

- The grammar

$$S \rightarrow A\ \alpha \mid \delta$$

$$A \rightarrow S\ \beta$$

is also left-recursive because

$$S \rightarrow^{+} S\ \beta\ \alpha$$

- This left-recursion can also be eliminated

- See Dragon Book for general algorithm
  - Section 4.3.3

Left factoring

Left-Factoring Example

- Recall the grammar

$$E \rightarrow T + E \mid T$$

$$T \rightarrow int \mid int * T \mid (E)$$

- Factor out common prefixes of productions

$$E \rightarrow T X$$

$$X \rightarrow + E \mid \varepsilon$$

$$T \rightarrow (E) \mid int\ Y$$

$$Y \rightarrow * T \mid \varepsilon$$

Calculating first and follow

Definitions

Basic Tools:

First: Let α be a string of grammar symbols. First(α) is the set that includes every terminal that appears leftmost in α or in any string originating from α.

NOTE: If $\alpha \Rightarrow \varepsilon$, then ε is First(α).

Follow: Let *A* be a non-terminal. Follow(*A*) is the set of terminals a that can appear directly to the right of *A* in some sentential form. ($S \Rightarrow \alpha A a \beta$, for some α and β).

NOTE: If $S \Rightarrow \alpha A$, then \$ is Follow(*A*).

Nullable Non-Terminal

- Non-terminal X is Nullable only if the following constraints are satisfied
  - base case:
    - if ($X \rightarrow \varepsilon$) then X is Nullable
  - inductive case:
    - if ($X \rightarrow ABC...$) and A, B, C, ... are all Nullable then X is Nullable

Computing Nullable Sets

- Compute X is Nullable by iteration:
  - Initialization:
    - Nullable := { }
    - if ($X \rightarrow \varepsilon$) then Nullable := Nullable U {X}
  - While Nullable different from last iteration do:
    - for all X,
      - if ($X \rightarrow ABC\ldots$) and A, B, C, ... are all Nullable then Nullable := Nullable U {X}

First Sets

- First(X) := First'(X) U E
    - E = {epsilon} if X is nullable
- First'(X) is specified like this:
    - base case:
        - if T is a terminal symbol then First'(T) = {T}
    - inductive case:
        - if X is a non-terminal and (X → ABC…) then
            - First'(X) = First'(ABC...)
              where First'(ABC...) = F1 U F2 U F3 U ... and
                - » F1 = First'(A)
                - » F2 = First'(B), if A is Nullable; emptyset otherwise
                - » F3 = First'(C), if A is Nullable & B is Nullable; emp...
                - » ...

Computing First Sets

- Compute First(X):
  - initialize:
    - if T is a terminal symbol then First (T) = {T}
    - if T is non-terminal then First(T) = { }
  - while First(X) changes (for any X) do
    - for all X and all rules (X → ABC...) do
      - First (X) U= First(X) U First (ABC...)
        where First(ABC...) := F1 U F2 U F3 U ... and
        - » F1 := First'(A)
        - » F2 := First'(B), if A is Nullable; { } otherwise
        - » F3 := First'(C), if A is Nullable & B is Nullable; { } …
        - » ...

Computing Follow Sets

- Follow(X) is computed iteratively
  - base case:
    - initially, we assume nothing in particular follows X
      - (when computing, Follow (X) is initially { })
      - Follow(S) = {$}
  - inductive case:
    - if (Y := s1 X s2) for any strings s1, s2 then
      - Follow (X) U= First'(s2)
      - Follow (X) U= Follow(Y), if s2 is Nullable

Example

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z			
Y			
X			

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z			
Y			
X			

base case

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first’	follow
Z	no		
Y			
X			

base case

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first’	follow
Z	no		
Y	yes		
X			

base case

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no		
Y	yes		
X	no		

base case

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first’	follow
Z	no		
Y	yes		
X	no		

after one round of induction, we realize we have reached a fixed point

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	{ }	
Y	yes	{ }	
X	no	{ }	

base case

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d	
Y	yes	c	
X	no	a,b	

round 1

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d	
Y	yes	c	
X	no	a,b	

round 1

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d	
Y	yes	c	
X	no	a,b	

round 1

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first’	follow
Z	no	d	
Y	yes	c	
X	no	a,b	

round 1

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d,a,b	
Y	yes	c	
X	no	a,b	

round 2

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d,a,b	
Y	yes	c	
X	no	a,b	

round 2

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d,a,b	
Y	yes	c	
X	no	a,b	

round 2

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d,a,b	
Y	yes	c	
X	no	a,b	

round 2

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d,a,b	
Y	yes	c	
X	no	a,b	

after three rounds of iteration, no more changes ==> fixed point

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first’	follow
Z	no	d,a,b	$
Y	yes	c	{ }
X	no	a,b	{ }

base case

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

after one round of induction, no fixed point

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first’	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

after one round of induction, no fixed point

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

after one round of induction, no fixed point

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first’	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

after one round of induction, no fixed point

building a predictive parser

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

	nullable	first'	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

after two rounds of induction, fixed point
(but notice, computing Follow(X) before Follow (Y) would have required 3rd round)

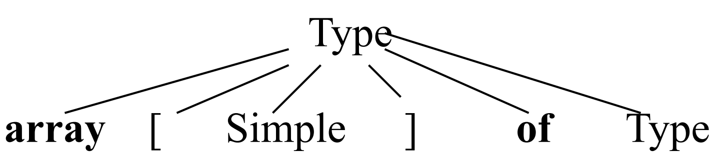
Building LL(1) parse table

Predictive Parsing Example

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**

Type
?

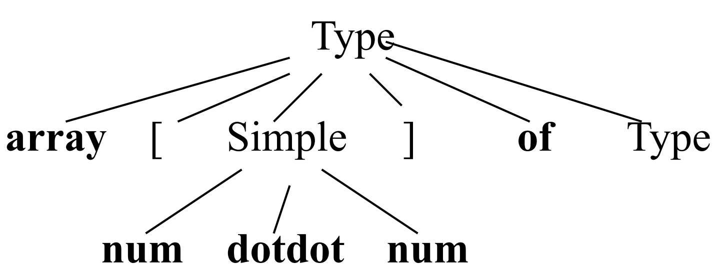


Start symbol

Type → simple
| ↑ **id**
| **array** [Simple] **of** Type
Simple → **integer**
| **char**
| **num dotdot num**

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**



Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

	a	b	c	d	e
Z					
Y					
X					

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if T ∈ First(s) then
  enter (X ::= s) in row X, col T
- if s is Nullable and T ∈ Follow(X)
  enter (X ::= s) in row X, col T

	a	b	c	d	e
Z					
Y					
X					

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if T ∈ First(s) then
  enter (X ::= s) in row X, col T
- if s is Nullable and T ∈ Follow(X)
  enter (X ::= s) in row X, col T

	a	b	c	d	e
Z	Z ::= XYZ	Z ::= XYZ			
Y					
X					

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if T ∈ First(s) then
  enter (X ::= s) in row X, col T
- if s is Nullable and T ∈ Follow(X)
  enter (X ::= s) in row X, col T

	a	b	c	d	e
Z	Z ::= XYZ	Z ::= XYZ		Z ::= d	
Y					
X					

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if T ∈ First(s) then
  enter (X ::= s) in row X, col T
- if s is Nullable and T ∈ Follow(X)
  enter (X ::= s) in row X, col T

	a	b	c	d	e
Z	Z ::= XYZ	Z ::= XYZ		Z ::= d	
Y			Y ::= c		
X					

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if T ∈ First(s) then
  enter (X ::= s) in row X, col T
- if s is Nullable and T ∈ Follow(X)
  enter (X ::= s) in row X, col T

	a	b	c	d	e
Z	Z ::= XYZ	Z ::= XYZ		Z ::= d	
Y	Y ::=	Y ::=	Y ::= c	Y ::=	Y ::=
X					

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

- if T ∈ First(s) then
  enter (X ::= s) in row X, col T
- if s is Nullable and T ∈ Follow(X)
  enter (X ::= s) in row X, col T

	a	b	c	d	e
Z	Z ::= XYZ	Z ::= XYZ		Z ::= d	
Y	Y ::=	Y ::=	Y ::= c	Y ::=	Y ::=
X	X ::= a	X ::= b Y e			

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

What are the blanks?

	a	b	c	d	e
Z	Z ::= XYZ	Z ::= XYZ		Z ::= d	
Y	Y ::=	Y ::=	Y ::= c	Y ::=	Y ::=
X	X ::= a	X ::= b Y e			

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

What are the blanks? --> syntax errors

	a	b	c	d	e
Z	Z ::= XYZ	Z ::= XYZ		Z ::= d	
Y	Y ::=	Y ::=	Y ::= c	Y ::=	Y ::=
X	X ::= a	X ::= b Y e			

Grammar:

Z ::= X Y Z
Z ::= d

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Is it possible to put 2 grammar rules in the same box?

	a	b	c	d	e
Z	Z ::= XYZ	Z ::= XYZ		Z ::= d	
Y	Y ::=	Y ::=	Y ::= c	Y ::=	Y ::=
X	X ::= a	X ::= b Y e			

Grammar:

Z ::= X Y Z
Z ::= d
Z ::= d e

Y ::= c
Y ::=

X ::= a
X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	$
Y	yes	c	d,a,b,e
X	no	a,b	c,d,a,b

Is it possible to put 2 grammar rules in the same box?

	a	b	c	d	e
Z	Z ::= XYZ	Z ::= XYZ		Z ::= d Z ::= d e	
Y	Y ::=	Y ::=	Y ::= c	Y ::=	Y ::=
X	X ::= a	X ::= b Y e			

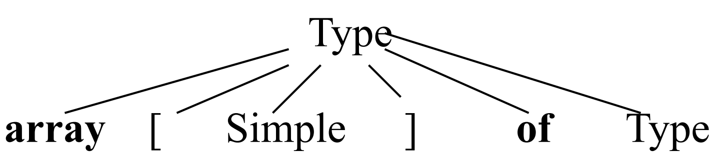
The parser

Predictive Parsing Example

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**

Type
?



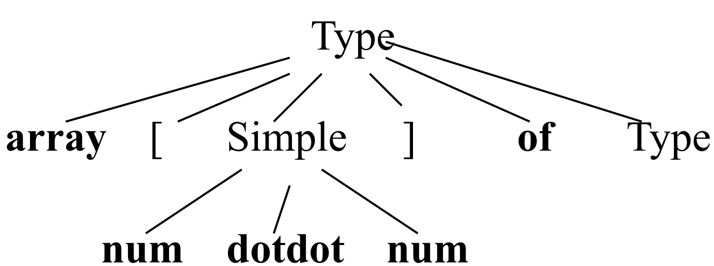
Start symbol

Type → simple
| ↑ **id**
| **array** [Simple] **of** Type

Simple → **integer**
| **char**
| **num dotdot num**

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**



LL(1) Parsing Table Example

- Left-factored grammar

$E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

LL(1) Parsing Table Example

- Left-factored grammar

$E \rightarrow T\,X$ $\qquad$ $X \rightarrow +\,E \mid \varepsilon$

$T \rightarrow (\,E\,) \mid \text{int}\;Y$ $\qquad$ $Y \rightarrow *\,T \mid \varepsilon$

- The LL(1) parsing table:

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Table Example

- Left-factored grammar

$E \rightarrow T\,X$ $\qquad$ $X \rightarrow +\,E \mid \varepsilon$

$T \rightarrow (\,E\,) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\,T \mid \varepsilon$

- The LL(1) parsing table:

next input token

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Table Example

- Left-factored grammar

$E \rightarrow T\ X$

$T \rightarrow (\ E\ )\ |\ \text{int}\ Y$

$X \rightarrow +\ E\ |\ \varepsilon$

$Y \rightarrow *\ T\ |\ \varepsilon$

- The LL(1) parsing table:

next input token

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

rhs of production to use

LL(1) Parsing Table Example

- Left-factored grammar

$E \rightarrow T\,X$

$T \rightarrow (\,E\,) \mid \text{int}\ Y$

$X \rightarrow +\,E \mid \varepsilon$

$Y \rightarrow *\,T \mid \varepsilon$

- The LL(1) parsing table:

next input token

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

leftmost non-terminal

rhs of production to use

LL(1) Parsing Table Example (Cont.)

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \rightarrow T X$"
  - This can generate an int in the first position

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

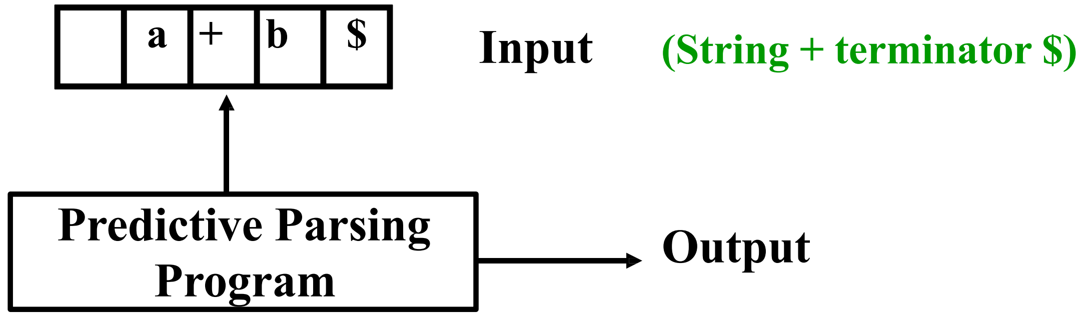
LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \rightarrow T\ X$"
  - This can generate an int in the first position

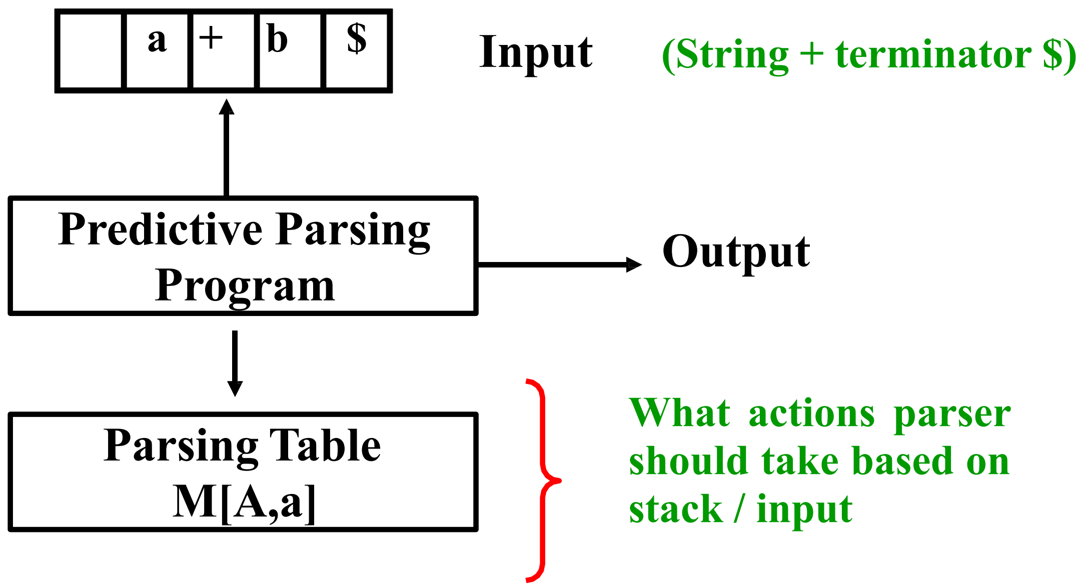
- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by +
    - only if $Y \rightarrow \varepsilon$

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

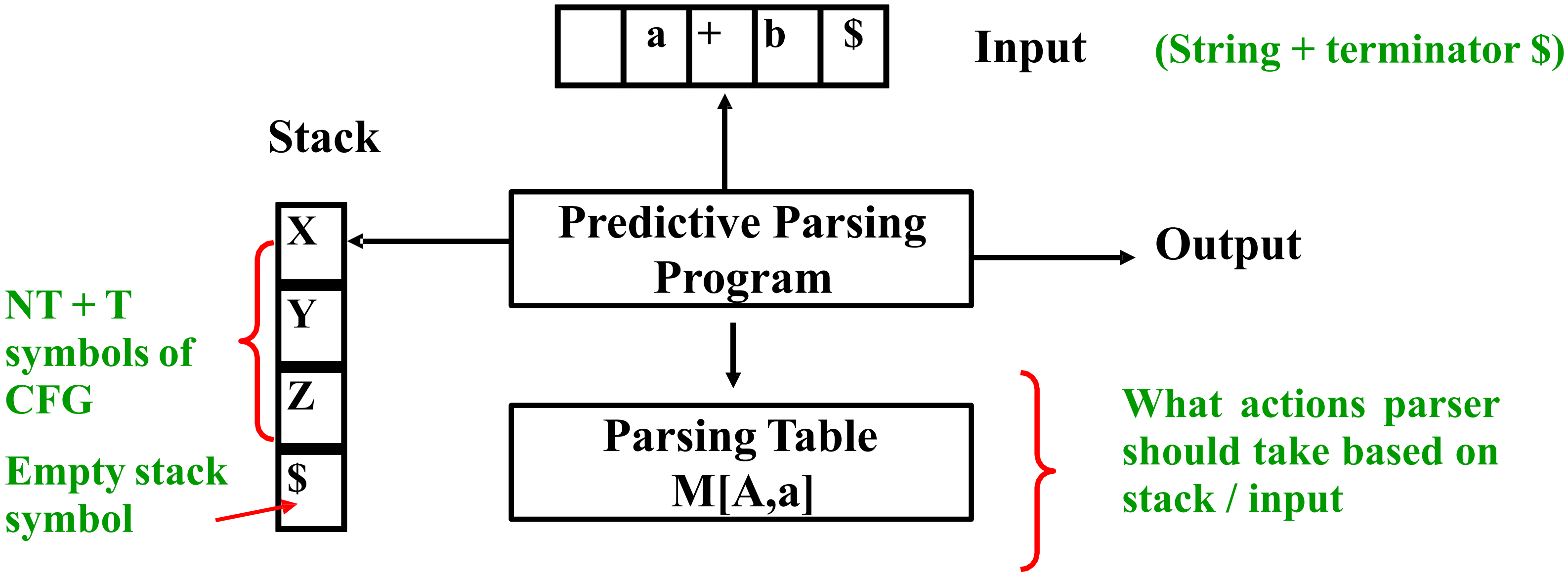
LL(1) Parsing Algorithm



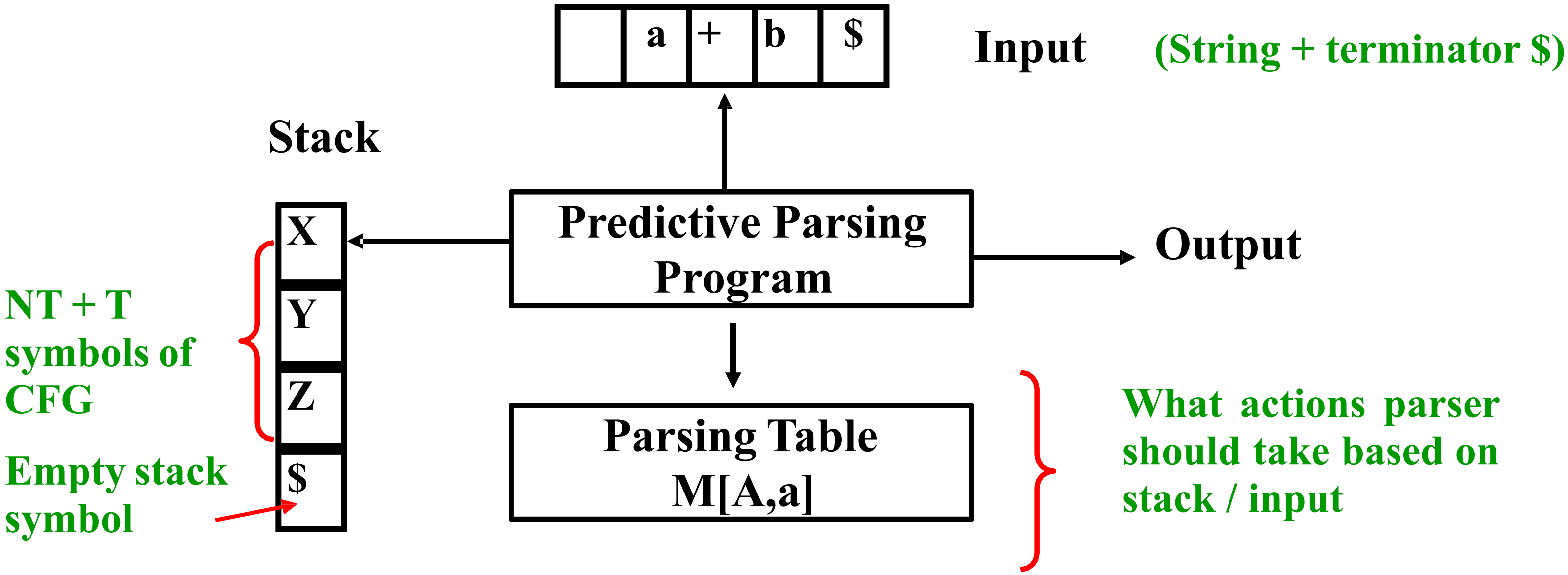
LL(1) Parsing Algorithm



LL(1) Parsing Algorithm

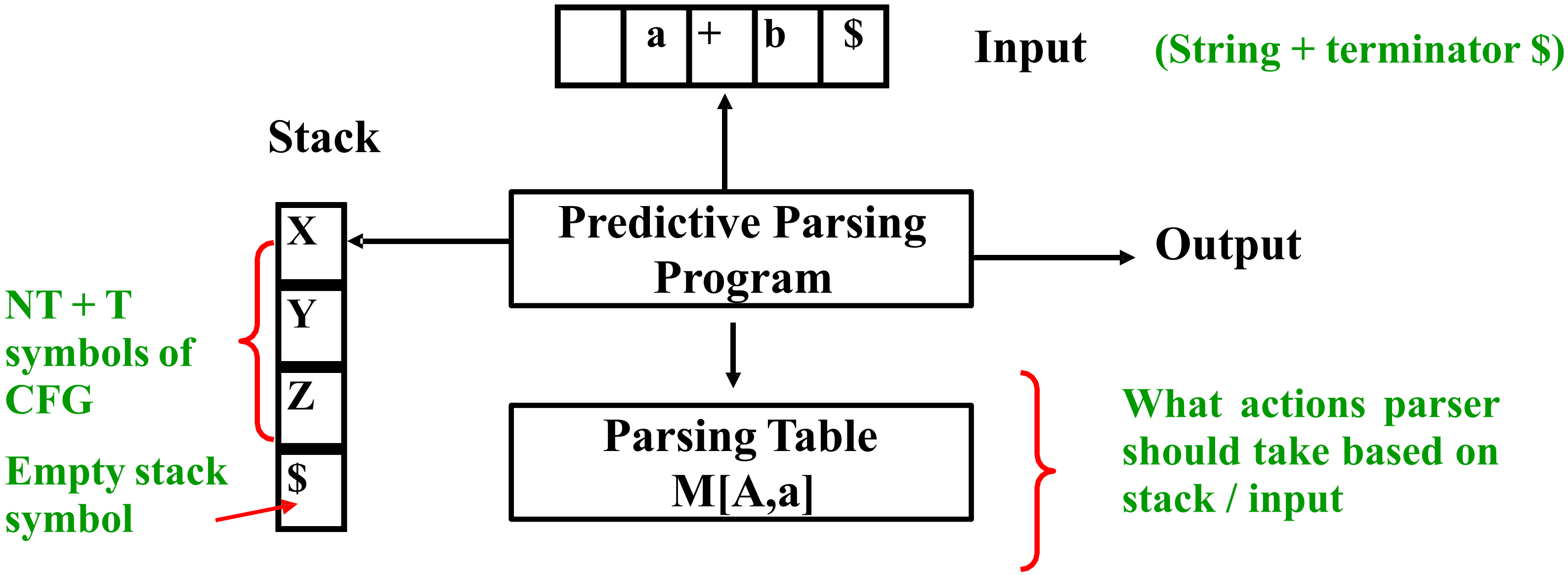


LL(1) Parsing Algorithm



General parser behavior: X : top of stack a : current token

LL(1) Parsing Algorithm



General parser behavior: X : top of stack a : current token

1. When X=a = $ halt, accept, success

2. When X=a ≠ $, POP X off stack, advance input, go to 1.

3.When X is a non-terminal, examine M[X, a], if it is an error, call recovery routine if M[X, a] = {UVW}, POP X, PUSH U,V,W, and **DO NOT** advance input

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action
E $	int * int $	T X

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε

LL(1) Parsing Example

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

Some notes

How to deal with multiple entries in a cell?

	a	b	c	d	e
Z	Z ::= XYZ	Z ::= XYZ		Z ::= d Z ::= d e	
Y	Y ::=	Y ::=	Y ::= c	Y ::=	Y ::=
X	X ::= a	X ::= b Y e			

- the example non-LL(1) grammar we just saw:

Z ::= X Y Z
Z ::= d
Z ::= d e

Y ::= c
Y ::=

X ::= a
X ::= b Y e

- how do we fix it?

another trick

- Previously, we saw that grammars with left-recursion were problematic, but could be transformed into LL(1) in some cases
- the example non-LL(1) grammar we just saw:

```
Z ::= X Y Z        Y ::= c        X ::= a
Z ::= d            Y ::=          X ::= b Y e
Z ::= d e
```

- solution here is left-factoring:

```
Z ::= X Y Z
Z ::= d W        W ::=        Y ::= c        X ::= a
                 W ::= e      Y ::=          X ::= b Y e
```

LL(1) Predictive Parsers

- Parser can “predict” which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means “left-to-right” scan of input
  - L means “leftmost derivation”
  - k means “predict based on k tokens of lookahead”
  - In practice, LL(1) is used

predictive parsing tables

- LL(k) parsing table

aa	ab	ba	bb	ac	ca	...

- LL(k) parser
- LL(k) grammer: a grammer that can be parsed with an LL(k) parser

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - <u>And in other cases as well</u>

- Most programming language CFGs are not LL(1)

Notes on LL(1) Grammars

Grammar is LL(1) ==> for all $A \rightarrow \alpha \mid \beta$

1. $\text{First}(\alpha) \cap \text{First}(\beta) = \varnothing$;
   - besides, only one of α or β can derive ε
2. if α derives ε, then $\text{Follow}(A) \cap \text{First}(\beta) = \varnothing$

It may not be possible for a grammar to be manipulated into an LL(1) grammar

Parser

to be continued …