TYPES OF PARSING TECHNIQUES

Building Parsers

- In theory classes, you might have learned about general mechanisms for parsing all CFGs
  - algorithms for parsing all CFGs are expensive
    - actually, with computers getting faster and bigger year over year, researchers are beginning to dispute this claim.
  - for 1/10/100 million-line applications, compilers must be fast.
    - even for 20 thousand-line apps, speed is nice
  - sometimes 1/3 of compilation time is spent in parsing
- compiler writers have developed specialized algorithms for parsing the kinds of CFGs that you need to build effective programming languages
  - LL(k), LR(k) grammars can be parsed.

Recursive Descent Parsing

- Recursive Descent Parsing (Appel Chap 3.2):
  - aka: predictive parsing; top-down parsing
  - simple, efficient
  - can be coded by hand in ML quickly
  - parses many, but not all CFGs
    - parses LL(1) grammars
      - Left-to-right parse; Leftmost-derivation; 1 symbol lookahead
  - key ideas:
    - one recursive function for each non terminal
    - each production becomes one clause in the function

non-terminals: S, E, L

terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, ;, =

rules:

1. S ::= IF E THEN S ELSE S
2. | BEGIN S L
3. | PRINT E
4. L ::= END
5. | ; S L
6. E ::= NUM = NUM

non-terminals: S, E, L
terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, ;, =
rules:

1. S ::= IF E THEN S ELSE S
2. | BEGIN S L
3. | PRINT E
4. L ::= END
5. | ; S L
6. E ::= NUM = NUM

Step 1: Represent the tokens

```
datatype token = NUM | IF | THEN | ELSE | BEGIN | END
               | PRINT | SEMI | EQ
```

Step 2: build infrastructure for reading tokens from lexing stream

function supplied by lexer

```
val tok = ref (getToken ())

fun advance () = tok := getToken ()

fun eat t = if (! tok = t) then advance () else error ()
```

non-terminals: S, E, L

terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, ;, =

rules:

1. S ::= IF E THEN S ELSE S
2. | BEGIN S L
3. | PRINT E
4. L ::= END
5. | ; S L
6. E ::= NUM = NUM

```
datatype token = NUM | IF
 | THEN | ELSE | BEGIN | END
 | PRINT | SEMI | EQ
```

```
val tok = ref (getToken ())
fun advance () = tok := getToken ()
fun eat t = if (! tok = t) then advance () else error ()
```

Step 3: write parser => one function per non-terminal; one clause per rule

```
fun S () = case !tok of
             IF    =>  eat IF; E (); eat THEN; S (); eat ELSE; S ()
           | BEGIN =>  eat BEGIN; S (); L ()
           | PRINT =>  eat PRINT; E ()

and L () = case !tok of
             END   =>  eat END
           | SEMI  =>  eat SEMI; S (); L ()

and E () = eat NUM; eat EQ; eat NUM
```

non-terminals: S, A, E, L

rules:

1. S ::= A EOF
2. A ::= ID := E
3. | PRINT (L)
4. E ::= ID
5. | NUM
6. L ::= E
7. | L , E

```
fun S () = A (); eat EOF

and A () = case !tok of
             ID       =>  eat ID; eat ASSIGN; E ()
           | PRINT =>  eat PRINT; eat LPAREN; L (); eat RPAREN

and E () = case !tok of
             ID       =>  eat ID
           | NUM    =>  eat NUM

and L () = case !tok of
             ID       =>  ???
           | NUM    =>  ???
```

problem

- predictive parsing only works for grammars where the first terminal symbol in the input provides enough information to choose which production to use
  - LL(1)
- when parsing L, if !tok = ID, the parser cannot determine which production to use:

```
6. L ::= E          (E could be ID)
7.       | L , E    (L could be E could be ID)
```

solution

- eliminate left-recursion
- rewrite the grammar so it parses the same language but the rules are different:

```
S ::= A EOF
A ::= ID := E
    | PRINT ( L )

E ::= ID
    | NUM

L ::= E
    | L , E
```

```
S ::= A EOF
A ::= ID := E
    | PRINT ( L )

E ::= ID
    | NUM
```

solution

- eliminate left-recursion
- rewrite the grammar so it parses the same language but the rules are different:

```
S ::= A EOF
A ::= ID := E
    | PRINT ( L )

E ::= ID
    | NUM

L ::= E
    | L , E
```

```
S ::= A EOF
A ::= ID := E
    | PRINT ( L )

E ::= ID
    | NUM

L ::= E M

M ::= , E M
    |
```

eliminating single left-recursion

- Original grammar form:

```
X ::= base
    | X repeat
```

Strings: base repeat repeat ...

- Transformed grammar:

```
X ::= base Xnew

Xnew ::= repeat Xnew
       |
```

Strings: base repeat repeat ...

Think about: what if you have mutually left-recursive variables X,Y,Z?
What's the most general pattern of left recursion? How to eliminate it?

Recursive Descent Parsing

- Unfortunately, can’t always eliminate left recursion
- Questions:
  - how do we know when we can parse grammars using recursive descent?
  - Is there an algorithm for generating such parsers automatically?

Constructing RD Parsers

- To construct an RD parser, we need to know what rule to apply when
  - we are trying to parse a non terminal X
  - we see the next terminal a in input
-
  -
  -

Constructing RD Parsers

- To construct an RD parser, we need to know what rule to apply when
  - we are trying to parse a non terminal X
  - we see the next terminal a in input
- We apply rule X ::= s when
  - a is the first symbol that can be generated by string s, OR
  - s reduces to the empty string (is nullable) and a is the first symbol in any string that can follow X

Constructing Predictive Parsers

1. Y ::=
2. | bb

3. X ::= c
4. | Y Z

5. Z ::= d

non-terminal seen	next terminal	rule
X	c	
X	b	
X	d	

Constructing Predictive Parsers

1. Y ::=
2. | bb

3. X ::= c
4. | Y Z

5. Z ::= d

non-terminal seen	next terminal	rule
X	c	3
X	b	
X	d	

Constructing Predictive Parsers

1. Y ::=
2. | bb

3. X ::= c
4. | Y Z

5. Z ::= d

non-terminal seen	next terminal	rule
X	c	3
X	b	4
X	d	

Constructing Predictive Parsers

1. Y ::=
2. | bb

3. X ::= c
4. | Y Z

5. Z ::= d

non-terminal seen	next terminal	rule
X	c	3
X	b	4
X	d	4

Constricting Predictive Parsers

- in general, must compute:
  - for each production X ::= s, must determine if s can derive the empty string.
    - if yes, X $\in$ Nullable
  - for each production X := s, must determine the set of all first terminals Q derivable from s
    - Q $\subseteq$ First(X)
  - for each non terminal X, determine all terminals symbols Q that immediately follow X
    - Q $\subseteq$ Follow(X)

Iterative Analysis

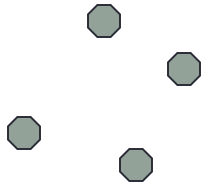
- Many compilers algorithms are iterative techniques.
- Iterative analysis applies when:
  - must compute a set of objects with some property P
  - P is defined inductively. ie, there are:
    - base cases: objects o1, o2 “obviously” have property P
    - inductive cases: if certain objects (o3, o4) have property P, this implies other objects (f o3; f o4) have property P
  - The number of objects in the set is finite
    - or we can represent infinite collections using some finite notation & we can find effective termination conditions

Iterative Analysis

- general form:
  - initialize set S with base cases
  - applied inductive rules over and over until you reach a fixed point
- a fixed point is a set that does not change when you apply an inductive rule (function)
  - Nullable, First and Follow sets can be determined through iteration
  - many program analyses & optimizations use iteration
  - worst-case complexity is bad
  - average-case complexity can be good: iteration “usually” terminates in a couple of rounds

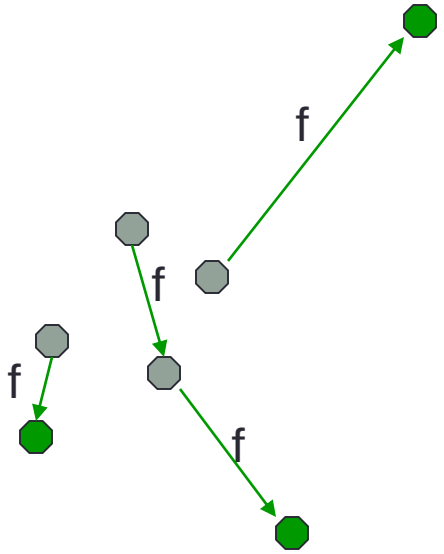
Iterative Analysis

Base Cases
“obviously”
have the property



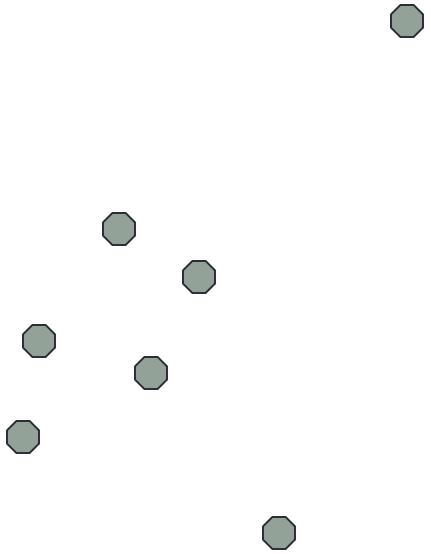
Iterative Analysis

apply function
(rule) to things
that have the
property to
produce new things
that have the property



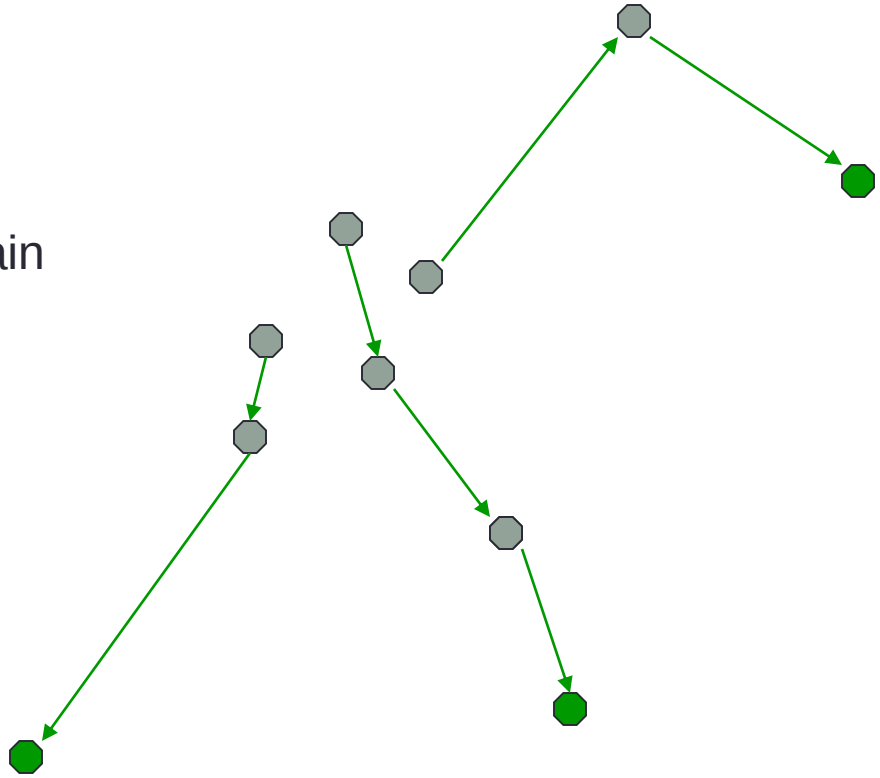
Iterative Analysis

Base Cases
+ things you get
by applying rule
to base cases
have the property



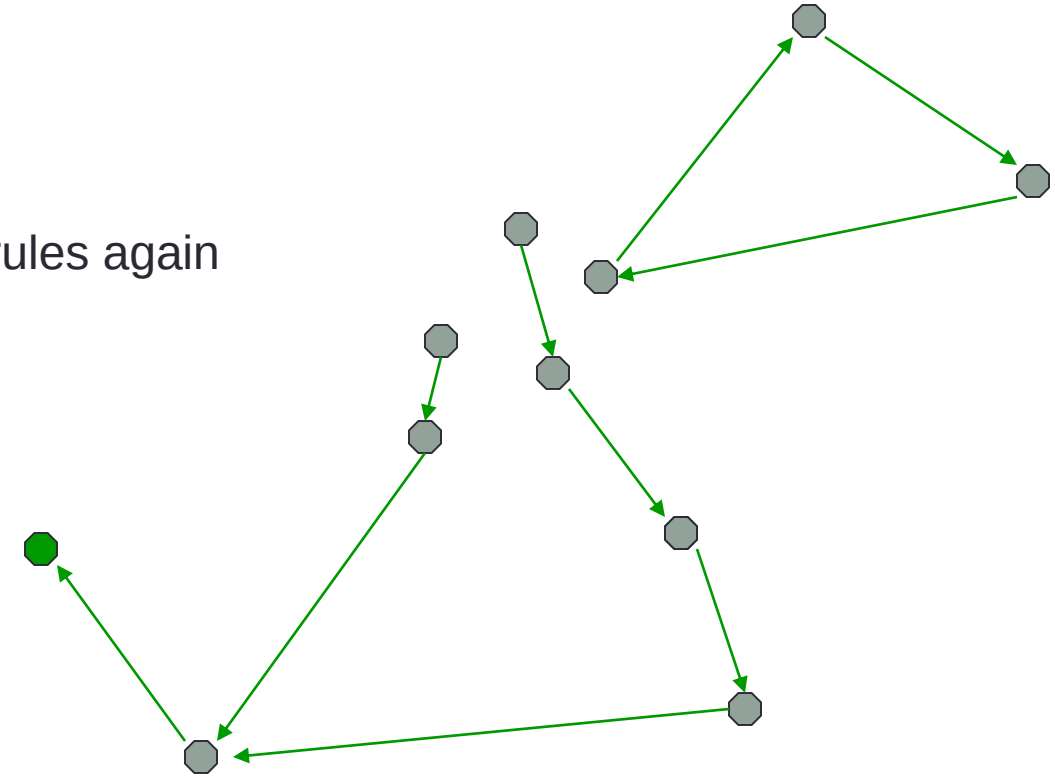
Iterative Analysis

Apply rules again



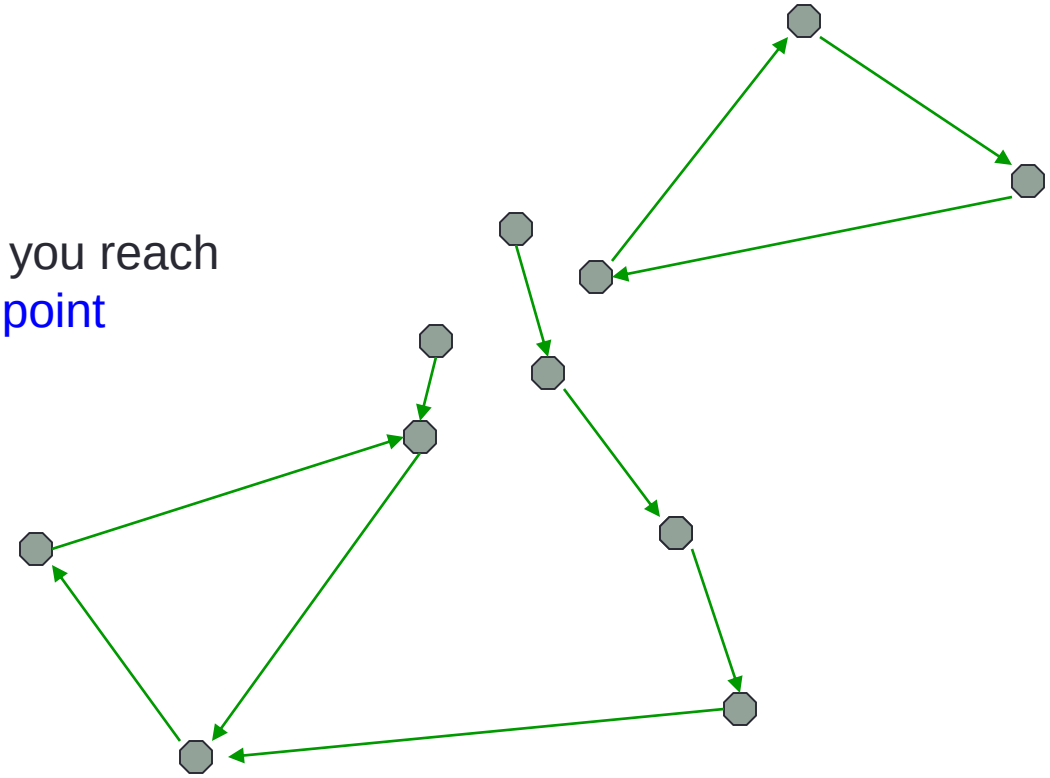
Iterative Analysis

Apply rules again



Iterative Analysis

Finally, you reach
a fixed point



Iterative Analysis

Example:
- axioms are “obviously true”/taken for granted
- rules of logic take basic axioms and prove new things are true

axioms:

“Dave teaches cos 441”

“Dave teaches cos 320”

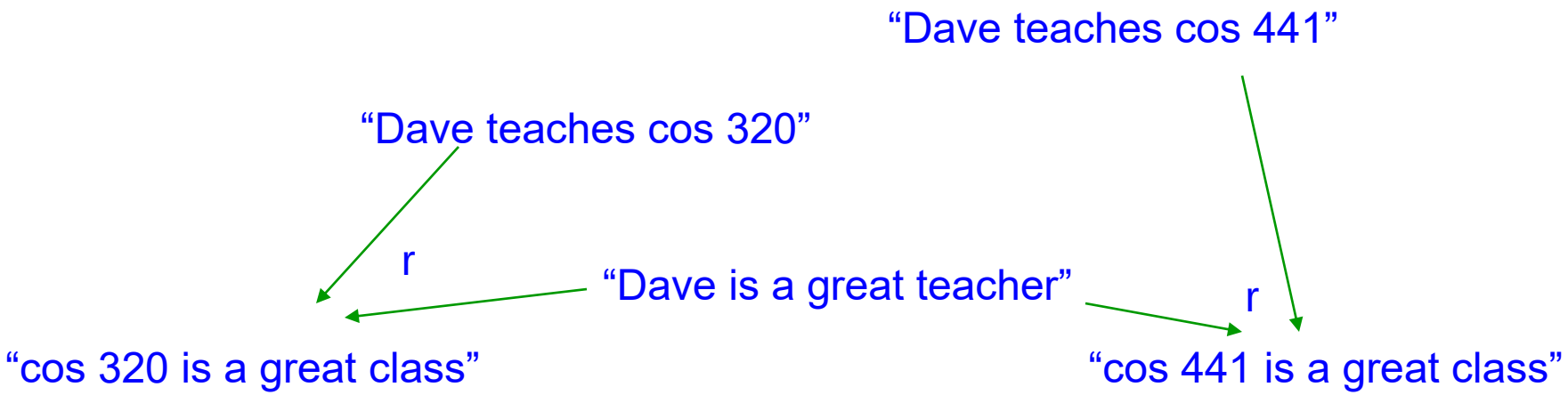
“Dave is a great teacher”

rule r: X is a great teacher /\ X teaches Y => Y is a great class

Iterative Analysis

Example:
- axioms are “obviously true”/taken for granted
- rules of logic take basic axioms and prove new things are true

axioms:



rule r: X is a great teacher /\ X teaches Y => Y is a great class

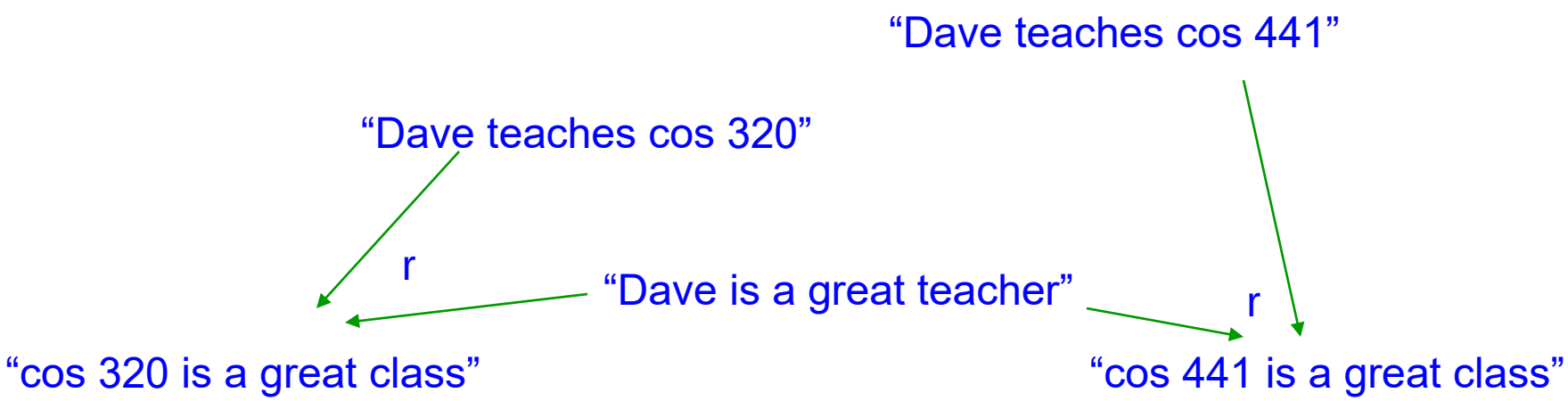
Iterative Analysis

Example:

- axioms are “obviously true”/taken for granted
- rules of logic take basic axioms and prove new things are true

Fixed Point Reached!

axioms:



rule r: X is a great teacher /\ X teaches Y => Y is a great class

Nullable Sets

- Non-terminal X is Nullable only if the following constraints are satisfied
  - base case:
    - if (X :=) then X is Nullable
  - inductive case:
    - if (X := 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 :=) then Nullable := Nullable U {X}
  - While Nullable different from last iteration do:
    - for all X,
      - if (X := ABC...) and A, B, C, ... are all Nullable then Nullable := Nullable U {X}

First Sets

- 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) := 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; emptyset otherwise
        - F3 := First (C), if A is Nullable & B is Nullable; emp...
        - ...

Computing Follow Sets

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

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			

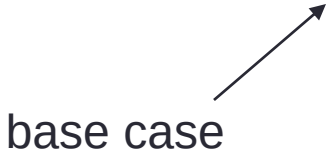
building a predictive parser

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

Y ::= c
Y ::=

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

	nullable	first	follow
Z	no		
Y	yes		
X	no		



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,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	e,d,a,b
X	no	a,b	c,e,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	e,d,a,b
X	no	a,b	c,e,d,a,b

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

Grammar:

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

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	{ }
Y	yes	c	e,d,a,b
X	no	a,b	c,e,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 Y ::= c X ::= a
Z ::= d Y ::= X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	{ }
Y	yes	c	e,d,a,b
X	no	a,b	c,e,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 Y ::= c X ::= a
Z ::= d Y ::= X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	{ }
Y	yes	c	e,d,a,b
X	no	a,b	c,e,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 Y ::= c X ::= a
Z ::= d Y ::= X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	{ }
Y	yes	c	e,d,a,b
X	no	a,b	c,e,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 Y ::= c X ::= a
Z ::= d Y ::= X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	{ }
Y	yes	c	e,d,a,b
X	no	a,b	c,e,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 Y ::= c X ::= a
Z ::= d Y ::= X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	{ }
Y	yes	c	e,d,a,b
X	no	a,b	c,e,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 Y ::= c X ::= a
Z ::= d Y ::= X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	{ }
Y	yes	c	e,d,a,b
X	no	a,b	c,e,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 Y ::= c X ::= a
Z ::= d Y ::= X ::= b Y e

Computed Sets:

	nullable	first	follow
Z	no	d,a,b	{ }
Y	yes	c	e,d,a,b
X	no	a,b	c,e,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 Y ::= c X ::= a
Z ::= d Y ::= X ::= b Y e

Computed Sets:

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

predictive parsing tables

- if a predictive parsing table constructed this way contains no duplicate entries, the grammar is called LL(1)
  - Left-to-right parse, Left-most derivation, 1 symbol lookahead
- if not, of the grammar is not LL(1)
- in LL(k) parsing table, columns include every k-length sequence of terminals:

aa	ab	ba	bb	ac	ca	...

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:

- how do we fix it?

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

Y ::= c
Y ::=

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

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:

- solution here is left-factoring:

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

Y ::= c
Y ::=

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

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

W ::=
W ::= e

Y ::= c
Y ::=

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

summary

- CFGs are good at specifying programming language structure
- parsing general CFGs is expensive so we define parsers for simple classes of CFG
  - LL(k), LR(k)
- we can build a recursive descent parser for LL(k) grammars by:
  - computing nullable, first and follow sets
  - constructing a parse table from the sets
  - checking for duplicate entries, which indicates failure
  - creating an ML program from the parse table
- if parser construction fails we can
  - rewrite the grammar (left factoring, eliminating left recursion) and try again
  - try to build a parser using some other method