

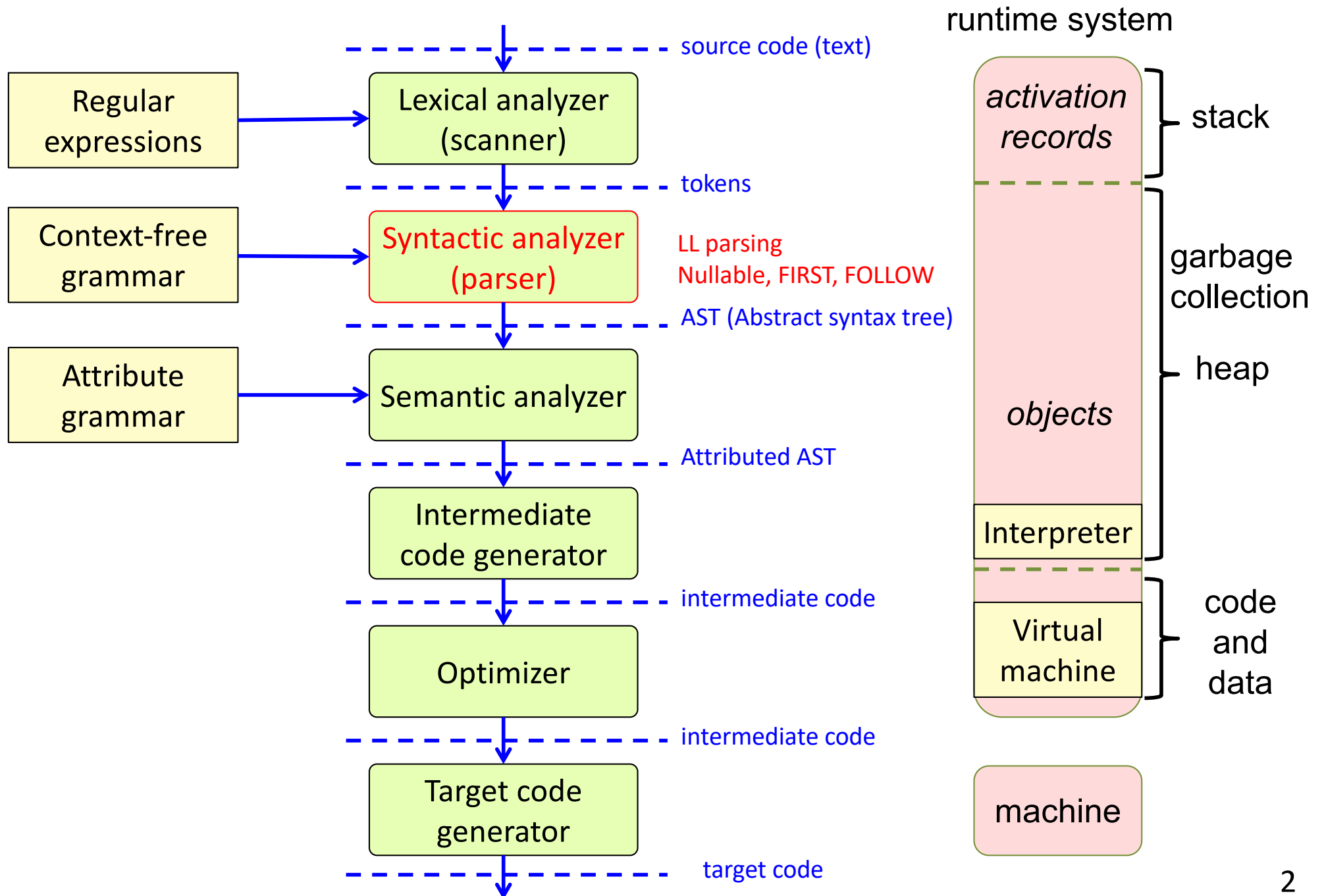
EDAN65: Compilers, Lecture 05 A

LL parsing

Nullable, FIRST, and FOLLOW

Görel Hedin

Revised: 2025-09-09

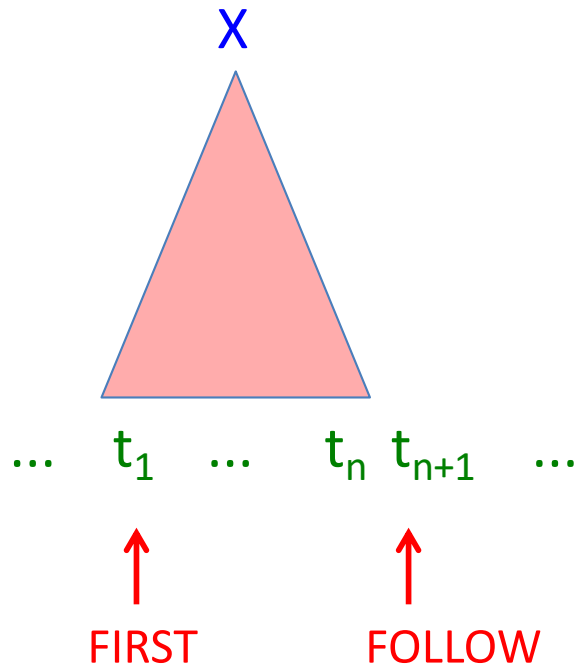


Algorithm for constructing an LL(1) parser

Fairly simple. The non-trivial part:

how to select the correct production p for X , based on the lookahead token.

p1: $X \rightarrow \gamma_1$
p2: $X \rightarrow \gamma_2$



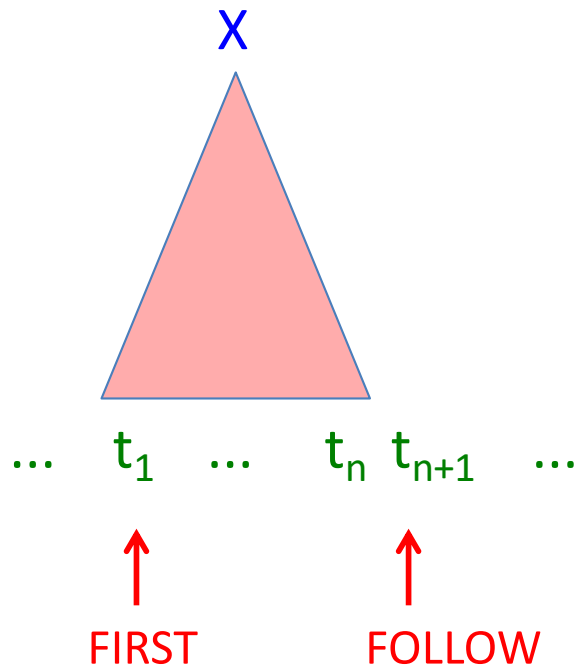
Algorithm for constructing an LL(1) parser

Fairly simple. The non-trivial part:

how to select the correct production p for X , based on the lookahead token.

$p1: X \rightarrow \gamma_1$

$p2: X \rightarrow \gamma_2$



- Which tokens can occur in the **FIRST** position?
- Can one of the productions derive the empty string? I.e., is it "**Nullable**"?
- If it is Nullable, which tokens can occur in the **FOLLOW** position?

Steps in constructing an LL(1) parser

1. Write the grammar on canonical form
2. Compute Nullable, FIRST, and FOLLOW.
3. Use them to construct a table. It shows what production to select, given the current lookahead token.
4. Conflicts in the table? The grammar is not LL(1).
5. No conflicts? Straightforward implementation using table-driven parser or recursive descent.

	t_1	t_2	t_3	t_4
x_1	p1	p2		
x_2		p3	p3	p4

Example:

Construct the LL(1) table for this grammar:

p1: statement -> assignment
p2: statement -> compoundStmt
p3: assignment -> ID "=" ID ";"
p4: compoundStmt -> "{" statements "
p5: statements -> statement statements
p6: statements -> ϵ

	ID	"="	";"	"{"	"}"
statement					
assignment					
compoundStmt					
statements					

Example:

Construct the LL(1) table for this grammar:

p1: statement -> assignment
p2: statement -> compoundStmt
p3: assignment -> ID "=" ID ";"
p4: compoundStmt -> "{" statements "
p5: statements -> statement statements
p6: statements -> ϵ

	ID	"="	";"	"{"	"}"
statement					
assignment					
compoundStmt					
statements					

For each production $p: X \rightarrow \gamma$, we are interested in:

FIRST(γ) – the tokens that occur first in a sentence derived from γ .

Nullable(γ) – is it possible to derive ϵ from γ ? And if so:

FOLLOW(X) – the tokens that can occur immediately after an X -sentence.

Example:

Construct the LL(1) table for this grammar:

p1: statement -> assignment
p2: statement -> compoundStmt
p3: assignment -> ID "=" ID ";"
p4: compoundStmt -> "{" statements "
p5: statements -> statement statements
p6: statements -> ϵ

	ID	"="	";"	"{"	"}"
statement					
assignment					
compoundStmt					
statements					

To construct the table, look at each production $p: X \rightarrow \gamma$.

Compute the token set $FIRST(\gamma)$. Add p to each corresponding entry for X .

Then, check if γ is **Nullable**. If so, compute the token set $FOLLOW(X)$, and add p to each corresponding entry for X .

Example:

Construct the LL(1) table for this grammar:

p1: statement -> assignment
p2: statement -> compoundStmt
p3: assignment -> ID "=" ID ";"
p4: compoundStmt -> "{" statements "
p5: statements -> statement statements
p6: statements -> ϵ

	ID	"="	";"	"{"	"}"
statement	p1			p2	
assignment	p3				
compoundStmt				p4	
statements	p5			p5	p6

To construct the table, look at each production $p: X \rightarrow \gamma$.

Compute the token set $FIRST(\gamma)$. Add p to each corresponding entry for X .

Then, check if γ is **Nullable**. If so, compute the token set $FOLLOW(X)$, and add p to each corresponding entry for X .

Example:

Dealing with End of File:

p1: varDecl -> type ID optInit
p2: type -> "integer"
p3: type -> "boolean"
p4: optInit -> "=" INT
p5: optInit -> ϵ

	ID	integer	boolean	"=	;"	INT	
varDecl							
type							
optInit							

Example:

Dealing with End of File:

p0: S -> varDecl \$
p1: varDecl -> type ID optInit
p2: type -> "integer"
p3: type -> "boolean"
p4: optInit -> "=" INT
p5: optInit -> ϵ

	ID	integer	boolean	"=	";"	INT	\$
S							
varDecl							
type							
optInit							

Example:

Dealing with End of File:

p0: S -> varDecl \$
p1: varDecl -> type ID optInit
p2: type -> "integer"
p3: type -> "boolean"
p4: optInit -> "=" INT
p5: optInit -> ϵ

	ID	integer	boolean	"=	";"	INT	\$
S		p0	p0				
varDecl		p1	p1				
type		p2	p3				
optInit				p4			p5

Example:

Ambiguous grammar:

p1: E $\rightarrow$ E "+" E
p2: E $\rightarrow$ ID
p3: E $\rightarrow$ INT

	"+"	ID	INT
E			

Example:

Ambiguous grammar:

p1: E -> E "+" E
p2: E -> ID
p3: E -> INT

	"+"	ID	INT
E		p1, p2	p1, p3

Collision in a table entry!
The grammar is not LL(1)

An ambiguous grammar is not even LL(k) –
adding more lookahead does not help.

Example:

Unambiguous, but left-recursive grammar:

p1: E -> E "*" F
p2: E -> F
p3: F -> ID
p4: F -> INT

	"*"	ID	INT
E			
F			

Example:

Unambiguous, but left-recursive grammar:

p1: E -> E "*" F
p2: E -> F
p3: F -> ID
p4: F -> INT

	"*"	ID	INT
E		p1,p2	p1,p2
F		p3	p4

Collision in a table entry!
The grammar is not LL(1)

A grammar with left-recursion is not even LL(k) –
adding more lookahead does not help.

Example:

Grammar with common prefix:

p1: E -> F "*" E
p2: E -> F
p3: F -> ID
p4: F -> INT
p5: F -> "(" E ")"

	"*"	ID	INT	"("	")"
E					
F					

Example:

Grammar with common prefix:

p1: E -> F "*" E
p2: E -> F
p3: F -> ID
p4: F -> INT
p5: F -> "(" E ")"

	"*"	ID	INT	"("	")"
E		p1,p2	p1,p2	p1,p2	
F		p3	p4	p5	

Collision in a table entry!
The grammar is not LL(1)

A grammar with common prefix is not LL(1).
Some grammars with common prefix are LL(k), for some k, –
but not this one.

Summary: constructing an LL(1) parser

1. Write the grammar on canonical form
2. Compute Nullable, FIRST, and FOLLOW.
3. Use them to construct a table. It shows what production to select, given the current lookahead token.
4. Conflicts in the table? The grammar is not LL(1).
5. No conflicts? Straight forward implementation using table-driven parser or recursive descent.

Algorithm for constructing an LL(1) table

initialize all entries $\text{table}[X_i, t_j]$ to the empty set.

for each production $p: X \rightarrow \gamma$

for each $t \in \text{FIRST}(\gamma)$

add p to $\text{table}[X, t]$

if $\text{Nullable}(\gamma)$

for each $t \in \text{FOLLOW}(X)$

add p to $\text{table}[X, t]$

	t_1	t_2	t_3	t_4
X_1	p1	p2		
X_2		p3	p3	p4

If some entry has more than one element, then the grammar is not LL(1).

Exercise: what is Nullable(**X**)?

$Z \rightarrow d$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

	Nullable
X	
Y	
Z	

Solution: what is Nullable(**X**)

$Z \rightarrow d$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

	Nullable
X	true
Y	true
Z	false

$X \Rightarrow Y \Rightarrow \epsilon$

yes, **X** is Nullable

$Y \Rightarrow \epsilon$

yes, **Y** is Nullable

$Z \Rightarrow XYZ \Rightarrow YYZ \Rightarrow^* Z \Rightarrow XYZ \dots$

no, **Z** is not Nullable, we cannot derive ϵ

Definition of Nullable

Definition of Nullable

Definition

Nullable(γ) is true iff the empty sequence can be derived from γ , i.e.,
iff there exists a derivation $\gamma \Rightarrow^* \epsilon$
(γ is a sequence of terminals and nonterminals)

Definition of Nullable

Definition

Nullable(γ) is true iff the empty sequence can be derived from γ , i.e.,
iff there exists a derivation $\gamma \Rightarrow^* \epsilon$
(γ is a sequence of terminals and nonterminals)

*Do case analysis to get equation system for **Nullable**, given $G=(N,T,P,S)$*

$$\text{Nullable}(\epsilon) == \text{true} \quad (1)$$

$$\text{Nullable}(t) == \text{false} \quad (2)$$

where $t \in T$, i.e., t is a terminal symbol

$$\text{Nullable}(X) == \text{Nullable}(\gamma_1) \mid \mid \dots \mid \mid \text{Nullable}(\gamma_n) \quad (3)$$

where $X \rightarrow \gamma_1, \dots, X \rightarrow \gamma_n$ are all the productions for X in P

$$\text{Nullable}(s\alpha) == \text{Nullable}(s) \ \&\& \ \text{Nullable}(\alpha) \quad (4)$$

where $s \in N \cup T$, i.e., s is a nonterminal or a terminal
and α is the rest of the sequence

*The equations for **Nullable** are recursive.*

*How would you write a program that computes **Nullable**(X)?*

Just using recursive functions could lead to nontermination!

Fixed-point problems

Fixed-point problems

Computing `Nullable(X)` is an example of a *fixed-point problem*.

These problems have the form:

$$x == f(x)$$

Can we find a value `x` for which the equation holds (i.e., a solution)?

`x` is then called a *fixed point* of the function `f`.

Fixed-point problems can (sometimes) be solved using iteration:

Guess an initial value x_0 , then apply the function iteratively, until the fixed point is reached:

$x_1 := f(x_0);$

$x_2 := f(x_1);$

...

$x_n := f(x_{n-1});$

until $x_n == x_{n-1}$

This is called a fixed-point iteration, and x_n is the fixed point.

Implement **Nullable** by a fixed-point iteration

Implement Nullable by a fixed-point iteration

represent Nullable as an array `nlbl[ ]` of boolean variables
initialize all `nlbl[X]` to false

```
repeat
  changed = false
  for each nonterminal  $X$  with productions  $X \rightarrow \gamma_1, \dots, X \rightarrow \gamma_n$  do
    newValue = nlbl( $\gamma_1$ ) || ... || nlbl( $\gamma_n$ )
    if newValue != nlbl[X] then
      nlbl[X] = newValue
      changed = true
  fi
do
until !changed
```

where `nlbl( $\gamma$ )` is computed using the current values in `nlbl[ ]`.

Implement **Nullable** by a fixed-point iteration

```
represent Nullable as an array nbl[ ] of boolean variables
initialize all nbl[X] to false

repeat
  changed = false
  for each nonterminal X with productions  $X \rightarrow \gamma_1, \dots, X \rightarrow \gamma_n$  do
    newValue = nbl( $\gamma_1$ ) || ... || nbl( $\gamma_n$ )
    if newValue != nbl[X] then
      nbl[X] = newValue
      changed = true
  fi
do
until !changed

where nbl( $\gamma$ ) is computed using the current values in nbl[ ].
```

The computation will terminate because

- the variables are only changed monotonically (from false to true)
- the number of possible changes is finite (from all false to all true)

Exercise: compute Nullable(X)

$Z \rightarrow d$
 $Z \rightarrow XYZ$
 $Y \rightarrow \varepsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

$nlbl[]$

	$iter_0$	$iter_1$	$iter_2$	$iter_3$
X	f			
Y	f			
Z	f			

In each iteration, compute:

for each nonterminal X with productions $X \rightarrow \gamma_1, \dots, X \rightarrow \gamma_n$
newValue = $nlbl(\gamma_1) \mid \dots \mid nlbl(\gamma_n)$

where $nlbl(\gamma)$ is computed using the current values in $nlbl[]$.

Solution: compute Nullable(X)

$Z \rightarrow d$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

$nlbl[]$

	$iter_0$	$iter_1$	$iter_2$	$iter_3$
X	f	f	t	t
Y	f	t	t	t
Z	f	f	f	f

In each iteration, compute:

for each nonterminal X with productions $X \rightarrow \gamma_1, \dots, X \rightarrow \gamma_n$
newValue = $nlbl(\gamma_1) \mid \dots \mid nlbl(\gamma_n)$

where $nlbl(\gamma)$ is computed using the current values in $nlbl[]$.

Definition of FIRST

Definition of FIRST

FIRST(γ) is the set of tokens that can occur *first* in sentences derived from γ :

$$\text{FIRST}(\gamma) = \{t \in T \mid \gamma \Rightarrow^* t \delta\}$$

Definition of FIRST

FIRST(γ) is the set of tokens that can occur *first* in sentences derived from γ :

$$\text{FIRST}(\gamma) = \{t \in T \mid \gamma \Rightarrow^* t \delta\}$$

Do case analysis to get equation system for FIRST, given $G=(N,T,P,S)$

$$\text{FIRST}(\varepsilon) == \emptyset \quad (1)$$

$$\text{FIRST}(t) == \{t\} \quad (2)$$

where $t \in T$, i.e., t is a terminal symbol

$$\text{FIRST}(X) == \text{FIRST}(\gamma_1) \cup \dots \cup \text{FIRST}(\gamma_n) \quad (3)$$

where $X \rightarrow \gamma_1, \dots, X \rightarrow \gamma_n$ are all the productions for X in P

$$\text{FIRST}(s\alpha) == \text{FIRST}(s) \cup (\text{if } \text{Nullable}(s) \text{ then } \text{FIRST}(\alpha) \text{ else } \emptyset \text{ fi}) \quad (4)$$

where $s \in N \cup T$, i.e., s is a nonterminal or a terminal

and α is the rest of the sequence

The equations for FIRST are recursive.

Compute using fixed-point iteration.

Implement **FIRST** by a fixed-point iteration

Implement **FIRST** by a fixed-point iteration

```
represent FIRST as an array FIRST[ ] of token sets
initialize all FIRST[X] to the empty set

repeat
  changed = false
  for each nonterminal X with productions  $X \rightarrow \gamma_1, \dots, X \rightarrow \gamma_n$  do
    newValue = FIRST( $\gamma_1$ )  $\cup$  ...  $\cup$  FIRST( $\gamma_n$ )
    if newValue  $\neq$  FIRST[X] then
      FIRST[X] = newValue
      changed = true
    fi
  do
until !changed

where FIRST( $\gamma$ ) is computed using the current values in FIRST[ ].
```

Implement **FIRST** by a fixed-point iteration

```
represent FIRST as an array FIRST[ ] of token sets
initialize all FIRST[X] to the empty set

repeat
  changed = false
  for each nonterminal X with productions  $X \rightarrow \gamma_1, \dots, X \rightarrow \gamma_n$  do
    newValue = FIRST( $\gamma_1$ )  $\cup \dots \cup$  FIRST( $\gamma_n$ )
    if newValue  $\neq$  FIRST[X] then
      FIRST[X] = newValue
      changed = true
  fi
do
until !changed

where FIRST( $\gamma$ ) is computed using the current values in FIRST[ ].
```

The computation will terminate because

- the variables are changed monotonically (using set union)
- the largest possible set is finite: **T**, the set of all tokens
- the number of possible changes is therefore finite

Solution: compute **FIRST**(X)

$Z \rightarrow d$
 $Z \rightarrow XYZ$
 $Y \rightarrow \varepsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

	Nullable
X	t
Y	t
Z	f

FIRST[]

	iter ₀	iter ₁	iter ₂	iter ₃
X	∅			
Y	∅			
Z	∅			

In each iteration, compute:

for each nonterminal X with productions $X \rightarrow \gamma_1, \dots, X \rightarrow \gamma_n$
newValue = **FIRST**(γ_1) $\cup \dots \cup$ **FIRST**(γ_n)

where **FIRST**(γ) is computed using the current values in **FIRST**[].

Exercise: compute **FIRST**(X)

$Z \rightarrow d$
 $Z \rightarrow XYZ$
 $Y \rightarrow \varepsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

	Nullable
X	t
Y	t
Z	f

FIRST[]

	iter ₀	iter ₁	iter ₂	iter ₃
X	$\emptyset$	{a}	{a, c}	{a, c}
Y	$\emptyset$	{c}	{c}	{c}
Z	$\emptyset$	{a, c, d}	{a, c, d}	{a, c, d}

In each iteration, compute:

for each nonterminal X with productions $X \rightarrow \gamma_1, \dots, X \rightarrow \gamma_n$
newValue = **FIRST**(γ_1) $\cup \dots \cup$ **FIRST**(γ_n)

where **FIRST**(γ) is computed using the current values in **FIRST**[].

Definition of FOLLOW

Definition of FOLLOW

FOLLOW(*X*) is the set of tokens that can occur as the *first* token *following* *X*, in any sentential form derived from the start symbol *S*:

$$\text{FOLLOW}(X) = \{t \in T \mid S \Rightarrow^* \alpha X t \beta\}$$

Definition of FOLLOW

FOLLOW(X) is the set of tokens that can occur as the *first* token *following* X , in any sentential form derived from the start symbol S :

$$\text{FOLLOW}(X) = \{t \in T \mid S \Rightarrow^* \alpha X t \beta\}$$

The nonterminal X occurs in the right-hand side of a number of productions.

Let $Y \rightarrow \gamma X \delta$ denote such an occurrence, where γ and δ are arbitrary sequences of terminals and nonterminals.

Equation system for FOLLOW, given $G=(N,T,P,S)$

$$\text{FOLLOW}(X) == \bigcup \text{FOLLOW}(Y \rightarrow \gamma \underline{X} \delta), \quad (1)$$

over all occurrences $Y \rightarrow \gamma X \delta$

and where

$$\begin{aligned} \text{FOLLOW}(Y \rightarrow \gamma \underline{X} \delta) == & \quad (2) \\ \text{FIRST}(\delta) \cup (\text{if } \text{Nullable}(\delta) \text{ then } \text{FOLLOW}(Y) \text{ else } \emptyset \text{ fi}) \end{aligned}$$

*The equations for FOLLOW are recursive.
Compute using fixed-point iteration.*

Implement **FOLLOW** by a fixed-point iteration

Implement FOLLOW by a fixed-point iteration

represent FOLLOW as an array FOLLOW[] of token sets
initialize all FOLLOW[X] to the empty set

```
repeat
  changed = false
  for each nonterminal X do
    newValue ==  $\bigcup \text{FOLLOW}(Y \rightarrow \gamma \underline{X} \delta)$ , for each occurrence  $Y \rightarrow \gamma \underline{X} \delta$ 
    if newValue != FOLLOW[X] then
      FOLLOW[X] = newValue
      changed = true
    fi
  do
until !changed
```

where FOLLOW($Y \rightarrow \gamma \underline{X} \delta$) is computed using the current values in FOLLOW[].

Implement FOLLOW by a fixed-point iteration

```
represent FOLLOW as an array FOLLOW[ ] of token sets
initialize all FOLLOW[X] to the empty set

repeat
  changed = false
  for each nonterminal X do
    newValue ==  $\bigcup \text{FOLLOW}(Y \rightarrow \gamma \underline{X} \delta)$ , for each occurrence  $Y \rightarrow \gamma X \delta$ 
    if newValue != FOLLOW[X] then
      FOLLOW[X] = newValue
      changed = true
    fi
  do
until !changed

where FOLLOW( $Y \rightarrow \gamma \underline{X} \delta$ ) is computed using the current values in FOLLOW[ ].
```

Again, the computation will terminate because

- the variables are changed monotonically (using set union)
- the largest possible set is finite: T

Exercise: compute FOLLOW(X)

$S \rightarrow Z \$$
 $Z \rightarrow d$
 $Z \rightarrow X Y Z$
 $Y \rightarrow \varepsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

The grammar has been extended with end of file, \$.

	Nullable	FIRST
X	t	{a, c}
Y	t	{c}
Z	f	{a, c, d}

FOLLOW[]

	iter ₀	iter ₁	iter ₂	iter ₃
X	∅			
Y	∅			
Z	∅			

In each iteration, compute:

newValue == $\bigcup \text{FOLLOW}(Y \rightarrow \gamma \underline{X} \delta)$, for each occurrence $Y \rightarrow \gamma X \delta$

where $\text{FOLLOW}(Y \rightarrow \gamma \underline{X} \delta)$ is computed using the current values in FOLLOW[].

Solution: compute FOLLOW(X)

$S \rightarrow Z \$$
 $Z \rightarrow d$
 $Z \rightarrow X Y Z$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

The grammar has been extended with end of file, \$.

	Nullable	FIRST
X	t	{a, c}
Y	t	{c}
Z	f	{a, c, d}

FOLLOW[]

	iter ₀	iter ₁	iter ₂	iter ₃
X	∅	{a, c, d}	{a, c, d}	
Y	∅	{a, c, d}	{a, c, d}	
Z	∅	{a, c, d}	{a, c, d}	

In each iteration, compute:

newValue == $\bigcup \text{FOLLOW}(Y \rightarrow \gamma \underline{X} \delta)$, for each occurrence $Y \rightarrow \gamma X \delta$

where $\text{FOLLOW}(Y \rightarrow \gamma \underline{X} \delta)$ is computed using the current values in FOLLOW[].

Summary

Nullable(γ) is true iff the empty sequence can be derived from γ , i.e.,
iff there exists a derivation $\gamma \Rightarrow^* \epsilon$
(γ is a sequence of terminals and nonterminals)

FIRST(γ) is the set of tokens that can occur *first* in sentences derived from γ :
$$\text{FIRST}(\gamma) = \{t \in T \mid \gamma \Rightarrow^* t \delta\}$$

FOLLOW(X) is the set of tokens that can occur as the *first* token *following* X , in
any sentential form derived from the start symbol S :
$$\text{FOLLOW}(X) = \{t \in T \mid S \Rightarrow^* \alpha X t \beta\}$$

Summary questions

- Construct an LL(1) table for a grammar.
- What does it mean if there is a collision in an LL(1) table?
- Why can it be useful to add an end-of-file rule to some grammars?
- How can we decide if a grammar is LL(1) or not?
- What is the definition of Nullable, FIRST, and FOLLOW?
- What is a fixed-point problem?
- How can it be solved using iteration?
- How can we know that the computation terminates?