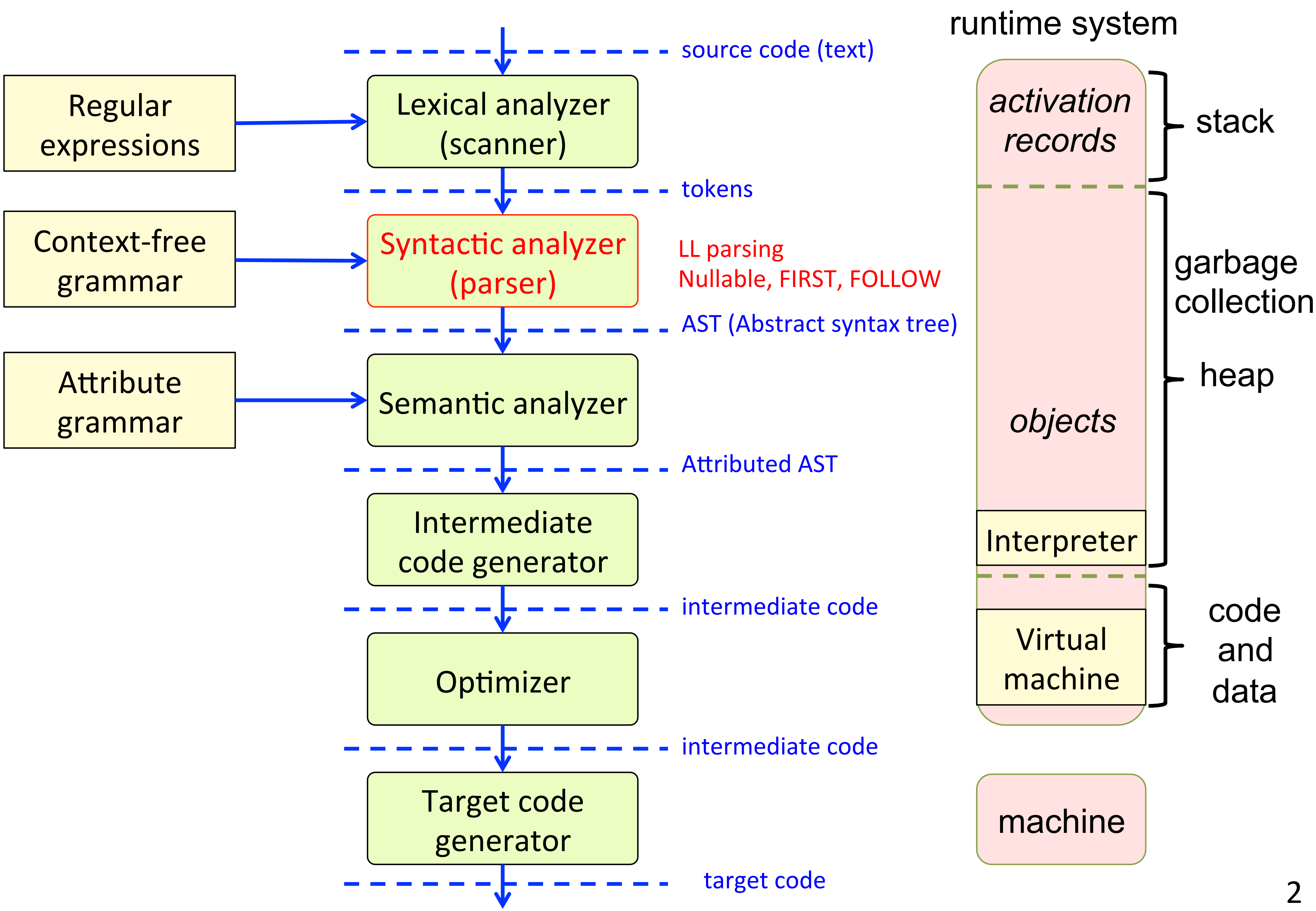
EDAN65: Compilers, Lecture 05 A

LL parsing
Nullable, FIRST, and FOLLOW

Görel Hedin

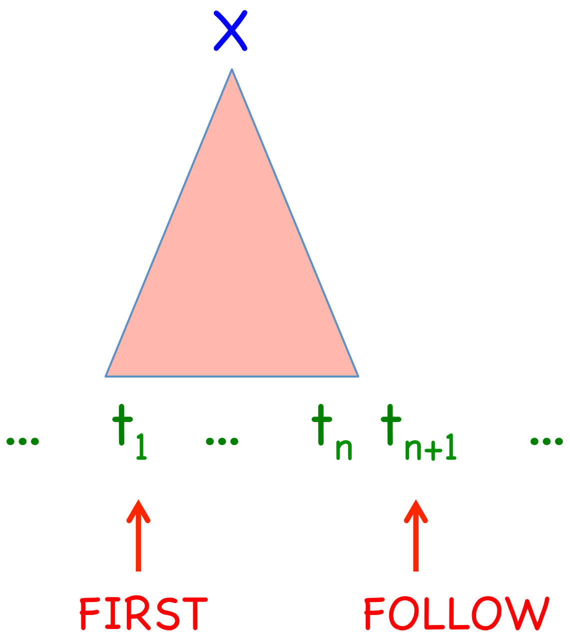
Revised: 2015-09-14


runtime system
source code (text)
Regular expressions
Lexical analyzer (scanner)
tokens
Context-free grammar
Syntactic analyzer (parser)
LL parsing
Nullable, FIRST, FOLLOW
AST (Abstract syntax tree)
Attribute grammar
Semantic analyzer
Attributed AST
Intermediate code generator
intermediate code
Optimizer
intermediate code
Target code generator
target code
activation records
stack
garbage collection
heap
objects
Interpreter
code and data
Virtual machine
machine

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 -> ...
p2: X -> ...

Which tokens can occur in the FIRST position?

Can one of the productions derive the empty string? I.e., is it "NULLABLE"?
If so, which tokens can occur in the FOLLOW position?

Steps in constructing an LL(1) parser

1. Write the grammar on canonical form

2. Analyze the grammar to construct a table.
   The table shows what production to select, given the current lookahead token.

3. Conflicts in the table? The grammar is not LL(1).

4. No conflicts? Straight forward 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 "=" expr ";"
p4: compoundStmt -> "{" statements "}"
p5: statements -> statement statements
p6: statements -> ε
```

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

For each production p: X -> γ, 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 "=" expr ";"
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 -> γ.
Compute the token set FIRST(γ). 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" "=" expr ";"
p4: optInit -> "=" INT
p5: optInit -> ε
```

	ID	integer	boolean	"="	";"	INT	
varDecl							
type							
optInit							

Example:

Dealing with End of File:

```
p0: S -> varDecl EOF
p1: varDecl -> type ID optInit
p2: type -> "integer"
p3: type -> "boolean" "=" expr ";"
p4: optInit -> "=" INT
p5: optInit -> ε
```

	ID	integer	boolean	"="	";"	INT	EOF
S							
varDecl							
type							
optInit							

Example:

Dealing with End of File:

```
p0: S -> varDecl EOF
p1: varDecl -> type ID optInit
p2: type -> "integer"
p3: type -> "boolean" "=" expr ";"
p4: optInit -> "=" INT
p5: optInit -> ε
```

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

Example:

Ambiguous grammar:

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

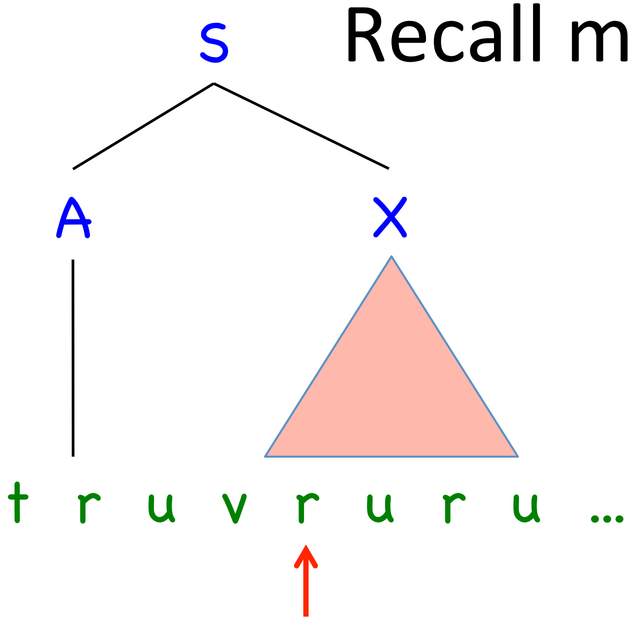
Collision in a table entry!
The grammar is not LL(1)

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

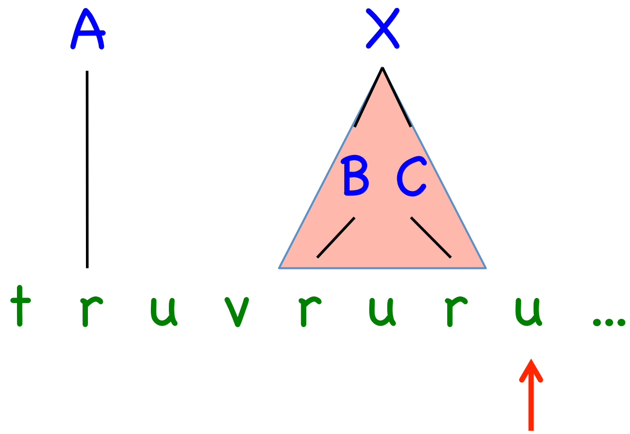
Summary: constructing an LL(1) parser

1. Write the grammar on canonical form
2. Analyze the grammar using FIRST, NULLABLE, and FOLLOW.
3. Use the analysis to construct a table.
   The table 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.

Recall main parsing ideas



LL(1): decides to build X after seeing the first token of its subtree.
The tree is built top down.



LR(1): decides to build X after seeing the first token following its subtree.
The tree is built bottom up.

For each production X -> γ we need to compute

FIRST(γ): the tokens that can appear first in a γ derivation

NULLABLE(γ): can the empty string be derived from γ?

FOLLOW(X): the tokens that can follow an X derivation

Algorithm for constructing an LL(1) table

```
initialize all entries table[X_i, t_j] to the empty set.

for each production p: X -> γ
  for each t ∈ FIRST(γ)
    add p to table[X, t]
  if NULLABLE(γ)
    for each t ∈ FOLLOW(X)
      add p to 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 -> d
Z -> X Y Z
Y -> ε
Y -> c
X -> Y
X -> a
```

	NULLABLE
X	
Y	
Z	

Solution: what is NULLABLE(X)

```
Z -> d
Z -> X Y Z
Y -> ε
Y -> c
X -> Y
X -> a
```

	NULLABLE
X	true
Y	true
Z	false

Definition of NULLABLE

Informally:

NULLABLE(γ): true if the empty string can be derived from γ
where γ is a sequence of terminals and nonterminals

Formal definition, given G=(N,T,P,S)

NULLABLE(ε) == true (1)

NULLABLE(t) == false (2)
where t $\in$ T, i.e., t is a terminal symbol

NULLABLE(X) == NULLABLE(γ_1) || … || NULLABLE(γ_n) (3)
where X -> γ_1, … X -> γ_n are all the productions for X in P

NULLABLE(sγ) == NULLABLE (s) && NULLABLE (γ) (4)
where s $\in$ N U T, i.e., s is a nonterminal or a terminal

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

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

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 -> γ1, …, X -> γn do
    newValue = nlbl(γ1) || … || nlbl(γn)
    if newValue != nlbl[X] then
       nlbl[X] = newValue
       changed = true
    fi
  do
until !changed
```

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

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 -> d
Z -> X Y Z
Y -> ε
Y -> c
X -> Y
X -> 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 -> γ_1, …, X -> γ_n
newValue = nlbl(γ_1) || … || nlbl(γ_n)

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

Solution: compute NULLABLE(X)

Z -> d
Z -> X Y Z
Y -> ε
Y -> c
X -> Y
X -> 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 -> γ_1, …, X -> γ_n
    newValue = nlbl(γ_1) || … || nlbl(γ_n)

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

Definition of FIRST

Informally:
FIRST(γ): the tokens that can occur as the *first* token in sentences derived from γ

Formal definition, given G=(N,T,P,S)

FIRST(ε) == $\varnothing$ (1)

FIRST(t) == { t } (2)
where t $\in$ T, i.e., t is a terminal symbol

FIRST(X) == FIRST(γ_1) $\cup$ … $\cup$ FIRST(γ_n) (3)
where X -> γ_1, … X -> γ_n are all the productions for X in P

FIRST(sγ) == FIRST(s) $\cup$ (if NULLABLE(s) then FIRST(γ) else $\varnothing$ fi) (4)
where s $\in$ N $\cup$ T, i.e., s is a nonterminal or a terminal

The equations for FIRST are recursive.
Compute using 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 -> γ1, ..., X -> γn do
    newValue = FIRST(γ1) ∪ ... ∪ FIRST(γn)
    if newValue != FIRST[X] then
       FIRST[X] = newValue
       changed = true
    fi
  do
until !changed

where FIRST(γ) 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 -> d
Z -> X Y Z
Y -> ε
Y -> c
X -> Y
X -> a
```

	NULLABLE
X	t
Y	t
Z	f

FIRST[ ]

	$iter_0$	$iter_1$	$iter_2$	$iter_3$
X	$\varnothing$			
Y	$\varnothing$			
Z	$\varnothing$			

In each iteration, compute:

for each nonterminal X with productions X -> γ_1, …, X -> γ_n
newValue = FIRST(γ_1) ∪ … ∪ FIRST(γ_n)

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

Exercise: compute FIRST(X)

Z -> d
Z -> X Y Z
Y -> ε
Y -> c
X -> Y
X -> 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 -> γ_1, …, X -> γ_n
  newValue = FIRST(γ_1) $\cup$ … $\cup$ FIRST(γ_n)

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

Definition of FOLLOW

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

Formal definition, given G=(N,T,P,S)

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

Let Y -> γ X δ denote such an occurrence, where γ and δ are arbitrary sequences of terminals and nonterminals.

FOLLOW(X) == $\bigcup$ FOLLOW(Y -> γ X δ), (1)
over all occurrences Y -> γ X δ

and where
FOLLOW(Y -> γ X δ) == (2)
FIRST(δ) $\cup$ (if NULLABLE(δ) then FOLLOW(Y) else $\varnothing$ fi)

The equations for FOLLOW are recursive.
Compute using 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 == ∪ FOLLOW(Y -> γ X δ ), for each occurrence Y -> γ X δ
    if newValue != FOLLOW[X] then
       FOLLOW[X] = newValue
       changed = true
    fi
  do
until !changed

where FOLLOW(Y -> γ X δ ) 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 -> Z $
Z -> d
Z -> X Y Z
Y -> ε
Y -> c
X -> Y
X -> 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 == U FOLLOW(Y -> γ X δ), for each occurrence Y -> γ X δ

where FOLLOW(Y -> γ X δ) is computed using the current values in FOLLOW[ ].

Solution: compute FOLLOW(X)

S -> Z $
Z -> d
Z -> X Y Z
Y -> ε
Y -> c
X -> Y
X -> a

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

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

FOLLOW[ ]

	$iter_0$	$iter_1$	$iter_2$	$iter_3$
X	∅	{c}	{a, c, d}	{a, c, d}
Y	∅	{a, c, d}	{a, c, d}	{a, c, d}
Z	∅	{$}	{$}	{$}

In each iteration, compute:

newValue == U FOLLOW(Y -> γ X δ), for each occurrence Y -> γ X δ

where FOLLOW(Y -> γ X δ) is computed using the current values in FOLLOW[ ].

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?