Compiler Construction

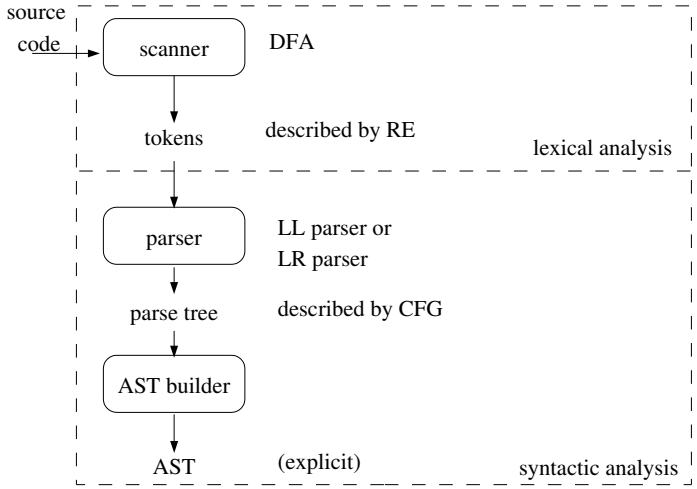
LR parsing
Fixed point equations

Lennart Andersson

Revision 2012-02-07

2012

Syntactic analysis



LR parsers

- build the tree bottom-up
- construct a rightmost derivation
- select rule after seeing all symbols matched by the right hand side and possibly some lookahead beyond that.
- many ambiguities can be treated by additional precedence rules
- are more powerful than LL parsers, the grammar seldom needs to be rewritten
- have better error recovery methods than LL parsers
- are more complex to understand
- are too complicated to build by hand

Grammar and (rightmost) derivation

$$S \rightarrow E\ \$$$
$$E \rightarrow E - x$$
$$E \rightarrow x$$

$$S \Rightarrow E\ \$ \Rightarrow E - x\ \$ \Rightarrow E - x - x\ \$ \Rightarrow x - x - x\ \$$$

LR execution

stack	input	action
	x - x - x $	shift
x	- x - x $	reduce by E → x
E	- x - x $	shift
E -	x - x $	shift
E - x	- x - x $	reduce by E → E - x
E	- x - x $	shift
E -	x - x $	shift
E - x	- x $	reduce by E → E - x
E	- x $	shift
E -	x $	shift
E - x	$	reduce by E → E - x
E	$	shift
E $		reduce by S → E $
S		accept

LR items

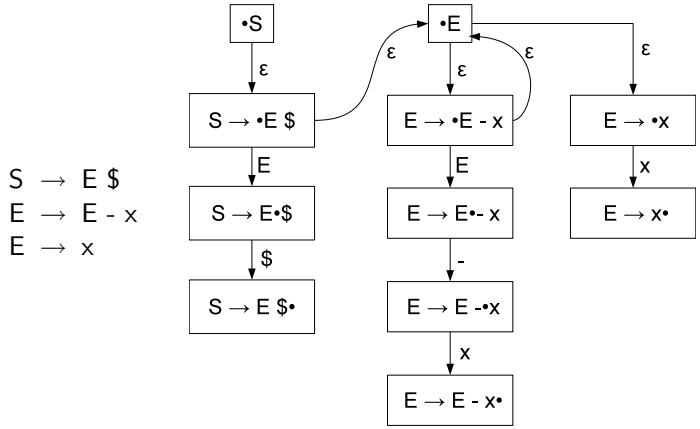
$$E \rightarrow E - x$$

gives rise to four different items:

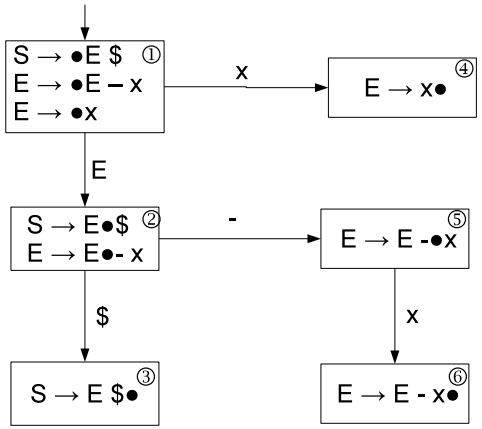
$$E \rightarrow \bullet E - x \qquad E \rightarrow E \bullet - x \qquad E \rightarrow E - \bullet x \qquad E \rightarrow E - x \bullet$$

NFA grammar

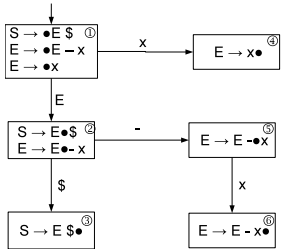
$S \rightarrow E\ \$$
$E \rightarrow E - x$
$E \rightarrow x$



DFA grammar



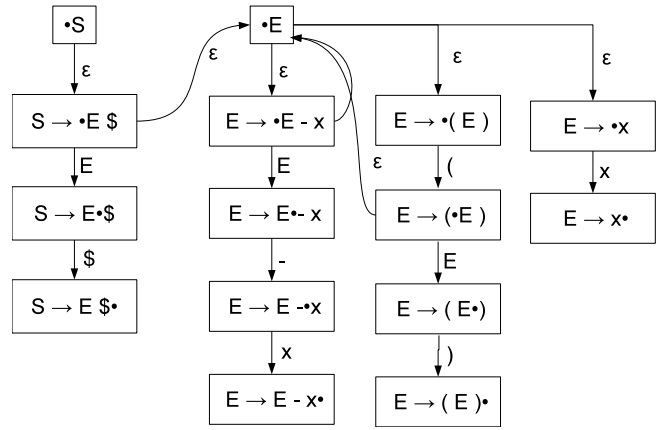
LR execution



stack	input	state	action
	x - x - x $	1	shift
x	- x - x $	4	E → x
E	- x - x $	2	shift
E -	x - x $	5	shift
E - x	- x - x $	6	E → E - x
E	- x - x $	2	shift
E -	x - x $	5	shift
E - x	- x $	6	E → E - x
E	- x $	2	shift
E -	x $	5	shift
E - x	$	6	E → E - x
E	$	2	shift
E $		3	S → E $
S			accept

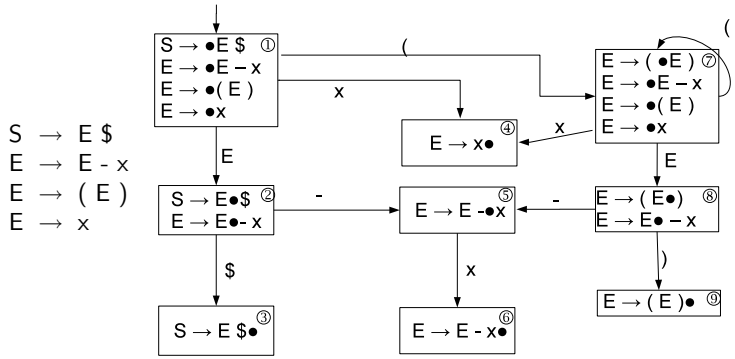
Another grammar

S → E $
E → E - x
E → (E)
E → x



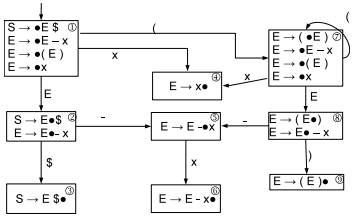
DFA

S → E $
E → E - x
E → (E)
E → x



LR execution

[0] S → E $
[1] E → E - x
[2] E → (E)
[3] E → x



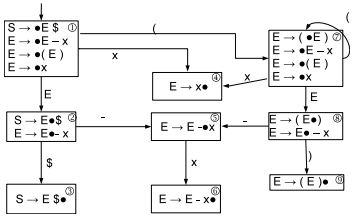
stack	input	state	action
	((x - x)) $	1	shift
(	(x - x)) $	7	shift
((	x - x)) $	7	shift
((x	- x)) $	4	red [3]
((E	- x)) $	8	shift
((E -	x)) $	5	shift
((E - x	)) $	6	red [1]
((E	)) $	8	shift
((E)	) $	9	red [2]
(E	) $	8	shift
(E)	$	9	red [2]
E	$	2	shift
E $		3	red [0]
S			accept

LR

stack	input	state	action
1	((x - x)) $	1	shift
1 (7	(x - x)) $	7	shift
1 (7 (7	x - x)) $	7	shift
1 (7 (7 x 4	- x)) $	4	reduce by E → x
1 (7 (7 E 8	- x)) $	8	shift
1 (7 (7 E 8 - 5	x)) $	5	shift
1 (7 (7 E 8 - 5 x 6	)) $	6	reduce by E → E - x
1 (7 (7 E 8	)) $	8	shift
1 (7 (7 E 8) 9	) $	9	reduce by E → (E)
1 (7 E 8	) $	8	shift
1 (7 E 8) 9	$	9	reduce by E → (E)
1 E 2	$	2	shift
1 E 2 $ 3		3	reduce by S → E $
1 S			accept

LR execution with states

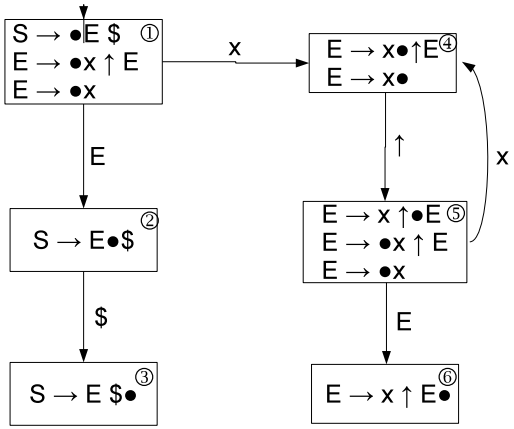
[0]: S → E $
[1]: E → E - x
[2]: E → (E)
[3]: E → x



state	action	x	-	(	)	$	E
1	shift	4		7			2
2	shift		5			3	
3	red by [0]						
4	red by [3]						
5	shift	6					
6	red by [1]						
7	shift	4		7			8
8	shift		5		9		
9	red by [2]						

LR with right recursion

S → E $
E → x ↑ E
E → x

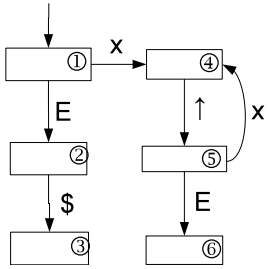


LR erroneous execution

stack	input	state	action
	x ↑ x $	1	shift
x	↑ x $	4	reduce
E	↑ x $	1	shift
E ↑	x $	?	error

Constructing the SLR table

	x	↑	$	E
1				
2				
3				
4				
5				
6				

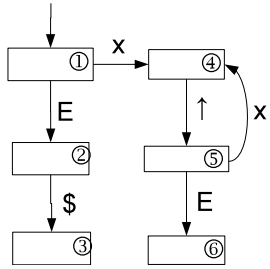


- for each transition $I \rightarrow J$ on a terminal T, put sJ in action[I,T].
- for each transition $I \rightarrow J$ on a nonterminal N, put gJ in action[I,N].
- for every state I containing an item $A \rightarrow \gamma$. from production [K] put rK in action[I,T] for all terminals T in follow(A).

I and J are states.

Constructing the SLR table

	x	↑	$	E
1	s4			g2
2			s3	
3	r1	r1	r1	
4		s5	r3	
5	s4			g6
6		r2		

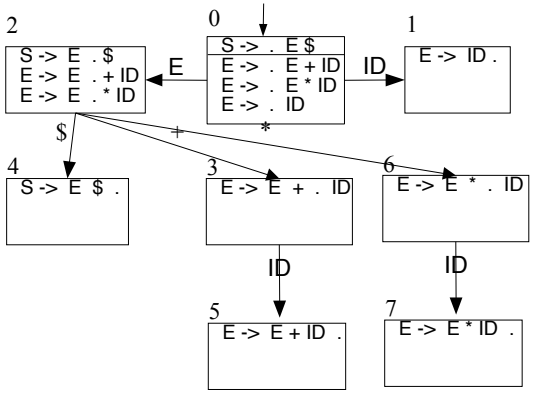


- for each transition $I \rightarrow J$ on a terminal T, put sJ in action[I,T].
- for each transition $I \rightarrow J$ on a nonterminal N, put gJ in action[I,N].
- for every state I containing an item $A \rightarrow \gamma$. from production [K] put rK in action[I,T] for all terminals T in follow(A).

I and J are states.

Yet another grammar

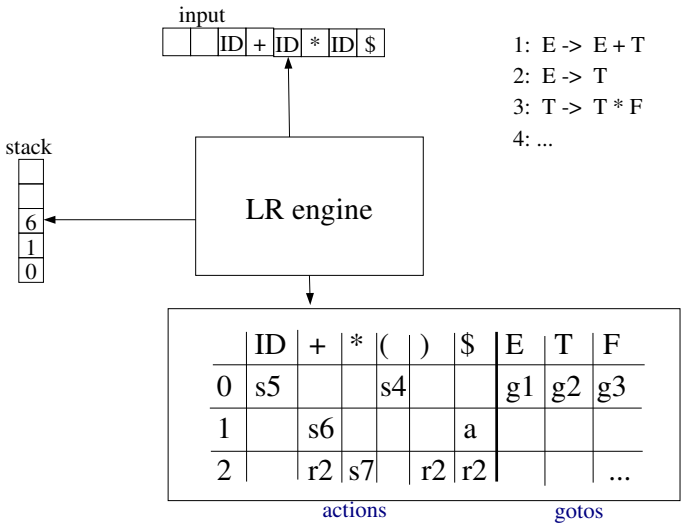
[0] S -> E $ {}
[1] E -> E + ID {+,*, $}
[1] E -> E * ID {+,*, $}
[2] E -> ID {+,*, $}



LR engine data structures

- uses an input stream of tokens
- uses a stack of DFA states
- uses two tables indexed by the DFA states and symbols:
  - an *action* table with entries
    - shift, s*n*
    - reduce, r*n*
    - accept, a
    - error
  - a *goto* table with state entries
    - goto, g*n*

Syntactic analysis



LR engine

```
stack.push(1); t = nextToken(); s = stack.top();
while(action[s][t] != a) {
  switch(action[s][t]) {
    case sn:
      stack.push(n);
      t = nextToken();
      break;
    case rn: // pn: X -> gamma
      pop() k times; // k is number of symbols in gamma;
      s' = stack.top();
      stack.push(goto[s',X]);
      break;
    default:
      error();
  }
  s = stack.top();
}
```


Parser generators

name	type	langauge
JavaCC	RD	Java
ANTLR	LL	several
yacc	LALR	C
bison	LALR	C
cup	LALR	Java
beaver	LALR	Java

An LALR parser is obtained from an LR parser by merging states that only differ in lookahead sets. The number of states and tables are much smaller. Some more conflicts can appear.

Cup example – specification 1

```
import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner.  */
init with {: scanner.init();                :};
scan with {: return scanner.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal            ID, PLUS, TIMES;
terminal            LEFTPAR, RIGHTPAR;

/* Non terminals */
non terminal            expr, term, factor;
```

Cup example – specification 2

```
/* The grammar */
 expr   ::= expr PLUS term;
 expr   ::= term;
 term   ::= term TIMES factor;
 term   ::= factor;
 factor ::= LEFTPAR expr RIGHTPAR;
 factor ::= ID;
```


Cup – execution

```
Opening files...
Parsing specification from standard input...
Checking specification...
Building parse tables...
  Computing non-terminal nullability...
  Computing first sets...
  Building state machine...
  Filling in tables...
  Checking for non-reduced productions...
Writing parser...
------- CUP v0.10k Parser Generation Summary -------
  0 errors and 0 warnings
  7 terminals, 4 non-terminals, and 7 productions declared,
  producing 13 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
  Code written to "parser.java", and "sym.java".
---------------------------------------------- (v0.10k)
```



```
===== Terminals =====
[0]EOF [1]error [2]ID [3]PLUS [4]TIMES
[5]LEFTPAR [6]RIGHTPAR

===== Non terminals =====
[0]$START [1]expr [2]term [3]factor

===== Productions =====
[0] expr ::= expr PLUS term
[1] $START ::= expr EOF
[2] expr ::= term
[3] term ::= term TIMES factor
[4] term ::= factor
[5] factor ::= LEFTPAR expr RIGHTPAR
[6] factor ::= ID
```


States

```
START lalr_state [0]: {
  [factor ::= (*) LEFTPAR expr RIGHTPAR , {EOF PLUS TIMES }]
  [expr ::= (*) term , {EOF PLUS }]
  [term ::= (*) factor , {EOF PLUS TIMES }]
  [$START ::= (*) expr EOF , {EOF }]
  [factor ::= (*) ID , {EOF PLUS TIMES }]
  [term ::= (*) term TIMES factor , {EOF PLUS TIMES }]
  [expr ::= (*) expr PLUS term , {EOF PLUS }]
}
transition on LEFTPAR to state [5]
transition on expr to state [4]
transition on term to state [3]
transition on factor to state [2]
transition on ID to state [1]

-------------------
lalr_state [1]: {
  [factor ::= ID (*) , {EOF PLUS TIMES RIGHTPAR }]
}

-------------------
lalr_state [2]: {
  [term ::= factor (*) , {EOF PLUS TIMES RIGHTPAR }]
}

-------------------
lalr_state [3]: {
  [expr ::= term (*) , {EOF PLUS RIGHTPAR }]
  [term ::= term (*) TIMES factor , {EOF PLUS TIMES RIGHTPAR }]
}
transition on TIMES to state [10]

...
```

LR parser conflicts

- A reduce/reduce conflict appears when two or more different productions may be used. By default the first rule is chosen.
- A shift/reduce conflict appears when there is choice of shifting or reducing. Shift is chosen by default; this will give left association for operators.

Precedence rules (cup, BNF grammar)

expr → expr '==' expr
expr → expr '**' expr
expr → expr '*' expr
expr → expr '/' expr
expr → expr '+' expr
expr → expr '-' expr
expr → ID
expr → INT

precedence nonassoc EQ
precedence left PLUS, MINUS
precedence left TIMES, DIV
precedence right POWER

Cup – shift/reduce conflict

```
/* Terminals (tokens returned by the scanner). */
terminal            ID, PLUS;

/* Non terminals */
non terminal            expr;

/* The grammar */
 expr ::= expr PLUS expr
        | ID;
```


Cup – conflict report

```
Opening files...
Parsing specification from standard input...
Checking specification...
Building parse tables...
  Computing non-terminal nullability...
  Computing first sets...
  Building state machine...
  Filling in tables...
*** Shift/Reduce conflict found in state #5
  between expr ::= expr PLUS expr (*)
  and     expr ::= expr (*) PLUS expr
  under symbol PLUS
  Resolved in favor of shifting.

  Checking for non-reduced productions...
Writing parser...
Closing files...
```

Set equations

Let S be a set of tokens that satisfies

$$S = \{ID\} \cup S$$

Solution:

Any set containing ID.

Solution by iteration

Equation:

$$S = \{ID\} \cup S$$

Iteration:

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= \{ID\} \cup S_i \end{aligned}$$

converges to the smallest solution

$$S_0 = \emptyset, \qquad S_1 = \{ID\}, \qquad S_2 = \{ID\}, \qquad \ldots$$

Generalization

Let F be a function on sets. Equation:

$$S = F(S)$$

Iteration:

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= F(S_i) \end{aligned}$$

$$S_0 = \emptyset, \qquad S_1 = F(\emptyset), \qquad S_2 = F(S_1) = F(F(\emptyset)) = F^2(\emptyset), \qquad \ldots$$

Will it converge?

Monotone functions

F is *monotone* if

$$A \subseteq B \text{ implies that } F(A) \subseteq F(B)$$

for all sets A and B.

Compare with real functions:

$f(x)$ is *increasing* if $x \leq y$ implies that $f(x) \leq f(y)$ for all x and y.

A monotone function

$F(S) = \{ID\} \cup S$

If $A \subseteq B$ then $\{ID\} \cup A \subseteq \{ID\} \cup B$

Most such functions derived from grammars are monotone.

Convergence

If F is monotone then $\emptyset \subseteq F(\emptyset) \subseteq F(F(\emptyset)) \subseteq \ldots$.

If the sets under consideration are finite, e.g. the set of all terminal symbols in a grammar, this sequence will always converge.

It will converge to the least set satisfying the equation $S = F(S)$.
A set satisfying this equation is called a *fixed point* of F.

A system of equations

$$\begin{aligned} S &= \{ID\} \cup S \cup T \\ T &= \{INT\} \cup T \end{aligned}$$

Try iterating!

A "general" system of equations

$$\begin{aligned} S &= F_1(S, T) \\ T &= F_2(S, T) \end{aligned}$$

If $S_1 \subseteq S_2$ and $T_1 \subseteq T_2$ implies that

$$\begin{aligned} F_1(S_1, T_1) &\subseteq F_1(S_2, T_2) \\ F_2(S_1, T_1) &\subseteq F_2(S_2, T_2) \end{aligned}$$

corresponding convergence properties are true.

Example

p_1: program $\rightarrow$ 'begin' body 'end'
p_2: body $\rightarrow$ optStmts
p_3: optStmts $\rightarrow$ statement moreStmts
p_4: optStmts $\rightarrow$ ϵ
p_5: moreStmts $\rightarrow$ ';' statement moreStmts
p_6: moreStmts $\rightarrow$ ϵ
p_7: statement $\rightarrow$ 'if' expr 'then' statement
p_8: statement $\rightarrow$ ID '=' expr
p_9: expr $\rightarrow$ ID
p_{10}: expr $\rightarrow$ INT

FIRST sets

FIRST(X) = the set of terminal symbols that may appear as the first token in an X derivation.

System of equations for $FIRST(X)$

For each nonterminal X

$$FIRST(X) = FIRST(\gamma_1) \cup \ldots \cup FIRST(\gamma_n)$$

where $X \rightarrow \gamma_1, \ldots, X \rightarrow \gamma_n$ are all productions for the nonterminal X.

$$\begin{aligned} FIRST(\epsilon) &= \emptyset \\ FIRST(t) &= \{t\} \\ FIRST(s\gamma) &= \begin{cases} FIRST(s) & \text{if s is not nullable} \\ FIRST(s) \cup FIRST(\gamma) & \text{if s is nullable} \end{cases} \end{aligned}$$

where t is a terminal,
s is a terminal or a nonterminal,
and γ is a sequence of terminals and nonterminals

Algorithm for computation of $FIRST(X)$

Represent FIRST(X) as a vector first[ ] of token sets

```
Set first[]; -- initially empty sets
repeat
  changed = false
  for each nonterminal X with X → γ1,... X →
γn do
    old = first[X]
    first[X] = first[X] ∪ FIRST(γ1) ∪ ... ∪ FIRST(γ n)
    if (old != first[X]) then changed = true fi
  od
until not changed
```

where each $FIRST(\gamma_i)$ is computed using the current values in $first[\,]$

Example computation of FIRST

p_1: program $\rightarrow$ 'begin' body 'end'
p_2: body $\rightarrow$ optStmts
p_3: optStmts $\rightarrow$ statement moreStmts
p_4: optStmts $\rightarrow$ ϵ
p_5: moreStmts $\rightarrow$ ';' statement moreStmts
p_6: moreStmts $\rightarrow$ ϵ
p_7: statement $\rightarrow$ 'if' expr 'then' statement
p_8: statement $\rightarrow$ ID '=' expr
p_9: expr $\rightarrow$ ID
p_{10}: expr $\rightarrow$ INT

		first[ ]			
	nlbl	start value	iter 1	iter 2	iter 3
program	F	$\emptyset$			
body	T	$\emptyset$			
optStmts	T	$\emptyset$			
moreStmts	T	$\emptyset$			
statement	F	$\emptyset$			
expr	F	$\emptyset$			

FOLLOW

FOLLOW(X) = the set of terminal symbols that may occur *immediately* after an X derivation.

System of equations for *FOLLOW*(X)

Let *FOLLOW*$(X, Y \rightarrow \gamma X \delta)$ denote the set of all terminal symbols that may occur immediately after X in a derivation using this production

$$FOLLOW(X, Y \rightarrow \gamma X \delta) = \begin{cases} FIRST(\delta) & \text{if } \delta \text{ is not nullable} \\ FIRST(\delta) \cup FOLLOW(Y) & \text{if } \delta \text{ is nullable} \end{cases}$$

For each nonterminal X

$$FOLLOW(X) = \bigcup FOLLOW(X, Y \rightarrow \gamma X \delta)$$

where the union is taken over all occurrences of X in the right-hand sides of the productions.

Algorithm for computation of *FOLLOW*(X)

Represent FOLLOW(X) as a vector follow[ ] of token sets

```
Set follow[]; -- initially empty sets
repeat
  changed = false
  for each occurrence of a nonterminal X in a prod
    Y → γXδ  do
    old = follow[X]
    follow[X] = follow[X] ∪ FOLLOW(X, Y → γXδ)
    if (old != follow[X]) then changed = true fi
  od
until not changed
```

where each *FOLLOW*$(X, Y \rightarrow \gamma X \delta)$ is computed using the current values in *follow*[ ]

Example computation of FOLLOW

			follow[ ]		
	nlbl	first	start value	iter 1	iter 2
program	F	$\{begin\}$	$\emptyset$		
body	T	$\{if, ID\}$	$\emptyset$		
optStmts	T	$\{if, ID\}$	$\emptyset$		
moreStmts	T	$\{';'\}$	$\emptyset$		
statement	F	$\{if, ID\}$	$\emptyset$		
expr	F	$\{ID, INT\}$	$\emptyset$		

Nullable

$NULLABLE(X)$ is true if X can derive the empty string and false otherwise.

$$NULLABLE(X) = \begin{cases} true & \text{if } X \Rightarrow^* \epsilon \\ false & \text{otherwise} \end{cases}$$

System of equations for $NULLABLE(X)$

For each nonterminal X

$NULLABLE(X) = NULLABLE(\gamma_1) \vee \ldots \vee NULLABLE(\gamma_n)$

where $X \rightarrow \gamma_1, \ldots, X \rightarrow \gamma_n$ are all productions for the nonterminal X. $\vee$ is logical or.

$$\begin{aligned} NULLABLE(\epsilon) &= true \\ NULLABLE(t) &= false \\ NULLABLE(s\gamma) &= NULLABLE(s) \wedge NULLABLE(\gamma) \end{aligned}$$

where t is a terminal,
s is a terminal or a nonterminal,
γ is a sequence of terminals and nonterminals,
$\wedge$ is logical and.

Boolean systems of equations

There is a similar theory when the unknowns are boolean.

Algorithm for computation of *nullable*(X)

Represent nullable(X) as a vector nlbl[ ] of boolean values

```
boolean nlbl[] = new boolean[...]; // initially fal
repeat
    changed = false
    for each nonterminal X with X → γ1,... X →
γn do
      old = nlbl[X]
      nlbl[X] = nlbl[X] || nlbl(γ1) || ... || nlbl(γn)
      if (old != nlbl[X]) then changed = true fi
    od
until not changed
```

where each $nlbl(\gamma_i)$ is computed using the current values in $nlbl[\,]$

Example computation of nullable

	nlbl[ ]				
	start value	iter 1	iter 2	iter 3	iter 4
program	F				
body	F				
optStmts	F				
moreStmts	F				
statement	F				
expr	F				

NULLABLE, FIRST, and FOLLOW

	nlbl	*FIRST*	*FOLLOW*
program	*F*	{*begin*}	{$}
body	*T*	{*if*, *ID*}	{*end*}
optStmts	*T*	{*if*, *ID*}	{*end*}
moreStmts	*T*	{';'}	{*end*}
statement	*F*	{*if*, *ID*}	{';', *end*}
expr	*F*	{*ID*, *INT*}	{';', *end*, *then*}

Fixed-point computations

The computations of nullable, FIRST, and FOLLOW are examples of a *fixed-point* computations.

- assign initial "smallest" values
- assign iteratively according to the equations until all values converge.

A *fixed point* is a solution to the system of equations
The *least fixed point* is the "smallest" solution (with no unnecessary information in it)

- the computation will terminate and compute the least fixed point if the values can be placed in a *lattice* of finite *height* and all assignments are *monotonic* on this lattice.

A grammar as a set of equations

$$\begin{array}{lcl} \mathsf{S} & \rightarrow & \mathsf{E}\ \text{'\$'} \\ \mathsf{E} & \rightarrow & \mathsf{E}\ \text{'-'}\ \text{'x'} \mid \text{'x'} \end{array}$$

Let $\mathcal{S}$ and $\mathcal{E}$ be languages on the alphabet $\{x, -, \$\}$.

$$\begin{cases} \mathcal{S} & = & \mathcal{E}\ \{\text{"\$"}\} \\ \mathcal{E} & = & \mathcal{E}\ \{\text{"-"}\}\ \{\text{"x"}\} \cup \{\text{"x"}\} \end{cases}$$

Fixed point theorem

- If the functions are monotone and continuous on a chain complete partially order then there is a least fixed point.
- A language is a chain complete partially order.
- The language operators concatenation and union are monotone and continuous.
- Set complement is not a monotone operator.