COMPUTING FIRST AND FOLLOW SETS AND DEVELOPING AN LL(1) PARSER

Robert Heckendorn
University of Idaho

To compute the first and follow sets for use in make parse tables
you must pretreat the grammar by removing alternations and
then prefixing the grammar with a production that
attaches an End of Input token (often denoted by a $) to start
symbol:

```
Pgm = start '$'
```

If we have productions of the form some with alternations:

```
A_1 = X_1 X_2 X_3...X_n_1a | X_1 X_2 X_3...X_n_1b
A_2 = X_1 X_2 X_3...X_n_2
A_3 = X_1 X_2 X_3...X_n_3a  | X_1 X_2 X_3...X_n_3b
           .
           .
           .
A_k = X_1 X_2 X_3...X_n_k
```

Then we convert to this equivalent grammar with no alternations:
Note the duplicates on the left hand side. Underscore denotes a
subscript. X_n_1 reads X sub (n sub 1).

```
A_0 = A_1 $
A_1 = X_1 X_2 X_3...X_n_1a
A_1 = X_1 X_2 X_3...X_n_1b
A_2 = X_1 X_2 X_3...X_n_2
A_3 = X_1 X_2 X_3...X_n_3a
A_3 = X_1 X_2 X_3...X_n_3b
           .
           .
           .
A_k = X_1 X_2 X_3...X_n_k
```

The productions can be numbered P_1, ...P_m.

Then we proceed with the algorithms.

We will begin by describing the algorithms finding the values of the
arrays First and Follow. First(A) is an array indexed by a terminal or
nonterminal and its value is a set of terminals and/or \epsilon.

```
Follow(A) is an array indexed by a nonterminal and its value is a set of
terminals and does not contain \epsilon.

COMPUTING First[A]
------------------

// Procedure computeFirst
//
// Input: productions P_1, P_2, ..., P_m where P_i = A::=X_1 X_2 X_3 ...X_n
//    with no alternations allowed in the productions.
//
// Output: Computes first of a term or nonterm accounting for nullability
// and multiple productions for the same nonterm.
//
//  First is an array indexed by a terminal or
//  nonterminal and its value is a set of terminals and/or \epsilon.
//
// First[A] for nonterminal A is the set of all possible tokens that
//    can occur as the first token of a sentence derived from A.
// First[A] for terminal A is simply the set { A }.
//
// Compute the first sets for all tokens from productions P_1, P_2, ..., P_m
//  where no production contains an alternation
//
// CALLS: computeFirstOfList(X_1, X_2, ... X_n)
procedure computeFirst({P_1, P_2, ...P_m})  // works on a list of productions
    // initial value for the First of anything
    foreach A \elemof TERMS do First[A] = {A}
    foreach A \elemof NONTERMS do First[A] = {}

    // loop until nothing new happens updating the First sets
    while stillchanging any First[A] do
        foreach production P_i = A::=X_1, X_2, ... X_n do
             First[A] <- First[A] \union computeFirstOfList(X_1, X_2, ... X_n)
        end foreach
    end while
end


// Procedure computeFirstOfList
//
// Computes the First of a rhs rather than just a token!
//
// This computes the set of tokens that can occur as the first
// token of a sentence derived from this rhs (right hand side) of
// of production.  That is X_1, X_2, ... X_n is a concatenation of
// terminals and nonterminals often found on the right hand side
// of a production.  This is nontrivial because some of the leading
```

```
// nonterminals on the rhs can go to epsilon.
//
// REFS: First[X_i]  (does not use Follow)
//
procedure computeFirstOfList(X_1, X_2, ... X_n)
    Tmp = {}
    k=0
    do
          k++
          Tmp <- Tmp \union First[X_k]-{\epsilon}
    while k<n & \epsilon isin First[X_k]

    // \epsilon only if X_1, X_2, ... X_n -> \epsilon
    // Note: this test can only possibly work if k==n:
    if \epsilon isin First[X_k] then Tmp <- Tmp \union {\epsilon}

    return Tmp
end

Note:

1. IMPORTANT: if grammar has no \epsilon then the procedure
   computeFirstOfList(X_1, X_2, ... X_n) simply returns First[X_1]

2. since \epsilon is removed when adding to First inside the do/while
\epsilon can only appear when the entire argument list can be
replaced by \epsilon (called this production is called NULLABLE).

3. First Sets can contain \epsilon as an element.  Follow Sets cannot as we'll see.

4. Conceptually, computeFirst generates a relation of the form:
First[A] = First[\alpha] \union First[\beta] \union ... \union { \epsilon },
for each production where A occurs on the left hand side (lhs).
This is based on the next point.

5. Conceptually, computeFirstOfList generates a relation of the form:
computeFirstOfList(X_1, X_2, ... X_n) = First[\alpha] \union First[\beta]
\union ... \union { \epsilon } where terms are added based on if
all of the terms before it in the rhs are nullable.


COMPUTING Follow[A]
-------------------

// Procedure computeFollow
//
// Input: productions P_1, P_2, ..., P_m where P_i = A::=X_1 X_2 X_3 ...X_n
//    with no alternations allowed in the productions.
```

```
//
// Output: Follow is an array indexed by a nonterminal and its value
// is a set of terminals.
//
// Follow[A] is the set of all possible tokens that
//    can occur after nonterminal A.   This procedure assumes you
//    have computed the First sets
//
// CALLS: computeFirstOfList(X_i+1,X_i+2...) which reqires First
// REFS: Follow[]
//
procedure computeFollow({P_1, P_2, ...P_m})   // works on a list of productions
    // initialize all the follow sets
    foreach A \elemof NONTERMS do Follow[A] = {}
    Follow[<start>]={$}

    // loop until nothing new happens updating the Follow sets
    while stillchanging any Follow[A] do {
        foreach P_i do {
            foreach X_i do           // over elements in right hand side!
                if X_i \elemof NONTERMS then {   // the body of this loop is over all the nonterms or
                     Follow[X_i] <- Follow[X_i] \union
                         computeFirstOfList(X_i+1 X_i+2 ...)-{\epsilon}
                     if \epsilon \elemof computeFirstOfList(X_i+1,X_i+2...) then
                         Follow[X_i] <- Follow[X_i] \union Follow[A]
                     end if
                end if
            end foreach
        end foreach
     end while
end

Note:
1. that since \epsilon is subtracted from First[X_i+1 X_i+2 ...] before
adding to Follow[X_i], \epsilon CANNOT OCCUR IN A FOLLOW SET.  This is
unlike the first set.

2. Only follow sets contain the end of input symbol '$'.

3. Conceptually, computeFollow generates a relation of the form:
Follow[X_i] = First[\alpha] \union First[\beta]
\union ... \union Follow[A] - { \epsilon } where any of these terms may be absent
depending on the grammar.  This is based on the next point.

PREDICT SET
-----------

The Predict Set of a production tells what lookahead tokens predict the
```

```
use of that production A::=X_1, X_2, ... X_n
This is simply computeFirstOfList but if that is empty then use Follow.

// compute the predict set of a production
procedure computePredict(A::=X_1, X_2, ... X_n)
    Tmp <- computeFirstOfList(X_1, X_2, ... X_n)
    if \epsilon \elemof Tmp then
         Tmp <- Tmp \union Follow[A]   // only need Follow if there is epsilon in computeFirstOfList
    endif
    return Tmp - { \epsilon }
end


If there is NO \epsilon in the grammar:

procedure computePredict(A::=X_1, X_2, ... X_n)
     return First[X_1]
end


Summary:

Function           Uses         TakesThisTypeOfArgument
--------           ----         -----------------------
computeFirst       First,Follow SetOfProductions
computeFollow      First,Follow SetOfProductions
computeFirstOfList First        RHSofProduction
computePredict     First,Follow Production



CONSTRUCTING THE LL PARSE TABLE
-------------------------------

If P_i are productions then we want
M(A, t) where A is \elemof NONTERMS and t \elemof computePredict(P_i) should
contain a reference to production P_i as the action to take.

This means:

An LL(1) parse table can be built if for every pair of productions P_i,
P_j with lhs(P_i) = lhs(P_j) that it is the case that
computePredict(P_i) intersect computePredict(P_j) = \emptyset

In other words:

there will be two productions in some M(A, t) which means
we don't know which to do in that case.
```

CAREFUL STEPS TO AUTOMATICALLY FINDING THE LL PARSE TABLE
---

1. remove the alternation and list the terms and nonterms
2. compute first sets for nonterminals
3. compute the follow sets (only needed if \epsilon is in grammar)
4. compute the predict sets.
5. create LL Parse Table

now you are ready to parse.

==========
EXAMPLE 1: NO \epsilon EXAMPLE

Given the following grammar with 5 productions which include alternation

<exp> ::= <exp> <addop> <term> | <term>
<addop> ::= + | -
<term> ::= <term> <mulop> <factor> | <factor>
<mulop> ::= *
<factor> ::= (<exp>) | num

STEP 1: REMOVE ALTERNATIONS (accept for some terminals) and
list the terms and nonterms. This is done for clarity

list of productions without alternation:

1) <exp> ::= <exp> <addop> <term>
2) <exp> ::= <term>
3) <addop> ::= + | - <-- cheating here
4) <term> ::= <term> <mulop> <factor>
5) <term> ::= <factor>
6) <mulop> ::= *
7) <factor> ::= (<exp>)
8) <factor> ::= num

TERMS = {+, -, *, (,), num}
NONTERMS = {<exp>, <addop>, <term>, <mulop>, <factor>}

Important Observation:

* The discovery of the first sets will be driven by the nonterminals in the lhs of the productions.
* The discovery of the follow sets will be driven by the nonterminals in the rhs of the productions.

STEP 2: COMPUTE THE FIRST SET

One way to think of the computeFirst algorithm is that it sets up relationships

First[exp] = First[exp] \union First[term]

	pass 1	pass 2	pass 3
<exp>	First[exp], First[term]	First[term]	(,num
<addop>	+, -	+,-	+,-
<term>	First[term], First[factor]	First[factor]	(,num
<mulop>	*	*	*
<factor>	(,num	(,num	(,num

STEP 3: COMPUTE THE FOLLOW SET. Not really needed because there are no \epsilons, but we do it here for practice. We will do this two ways. Note: only productions 1,2,4,5,7 affect the follow sets since the rhs of 3, 6, 8 are nothing but terminals.

1) <exp> ::= <exp> <addop> <term>
2) <exp> ::= <term>
4) <term> ::= <term> <mulop> <factor>
5) <term> ::= <factor>
7) <factor> ::= (<exp>)

Because this grammar has no \epsilon, more complex flow through the algorithm is avoided and the process is simply a collection of set dependencies. I will list out the dependencies and fill in the data in the three steps below. fst stands for First and fol for Follow to save room. The first line below essentially means Follow[exp] = First[addop] \union First["]").

Let's sketch out what will happen.
For simplicity, just list out the relationships from each production:

	First	prod 1	prod 2	prod 4	prod 5	prod 7
<exp>	(,num	fst(addop)				fst(")")
<addop>	+,-	fst(term)				
<term>	(,num	fol(exp)	fol(exp)	fst(mulop)		
<mulop>	*			fst(factor)		
<factor>	(,num			fol(term)	fol(term)	

IMPORTANT: Initialize fol(exp) = $

First[] replaces the First sets AND each row represents the follow set. e.g.
Follow[<exp>] = {$,+,-,)} initially below:

	First	prod 1	prod 2	prod 4	prod 5	prod 7	Follow
<exp>	(,num	+,-				)	$,+,-,)
<addop>	+,-	(,num					(,num
<term>	(,num	fol(exp)	fol(exp)	*			
<mulop>	*			(,num			(,num
<factor>	(,num			fol(term)	fol(term)		

Then iterate over the follow sets:

	First	prod 1	prod 2	prod 4	prod 5	prod 7	Follow
<exp>	(,num	+,-				)	$,+,-,)
<addop>	+,-	(,num					(,num
<term>	(,num	$,+,-,)	$,+,-,)	*			$,+,-,*,)
<mulop>	*			(,num			(,num
<factor>	(,num			$,+,-,*,)	$,+,-,*,)		$,+,-,*,)

Let's see that again only this time we compute the Follow set by
running through the algorithm. If a first or follow set is mentioned
here it stands for the empty set (not the empty string) and is just
there to be informative about what information we are using.

1) <exp> ::= <exp> <addop> <term>
2) <exp> ::= <term>
4) <term> ::= <term> <mulop> <factor>
5) <term> ::= <factor>
7) <factor> ::= (<exp>)

	First	prod 1	prod 2	prod 4	prod 5	prod 7
<exp>	(,num	fst(addop)				fst(")")
<addop>	+,-	fst(term)				
<term>	(,num	fol(exp)	fol(exp)	fst(mulop)		
<mulop>	*			fst(factor)		
<factor>	(,num			fol(term)	fol(term)	

pass 0:
Initialize fol(exp) = $

Pass 1:
Under each prod is what is ADDED for the production given at the top
of the column. The TOTAL column is what is in each follow set at the

end of the pass. Order of evaluation in each nonterminal is determined by the order of the productions (from prod 1 to prod 7).

	First	prod 1	prod 2	prod 4	prod 5	prod 7	Follow
<exp>	(,num	$,+,-				)	$,+,-,)
<addop>	+,-	(,num					(,num
<term>	(,num	$,+,-	$,+,-	*			$,+,-,*
<mulop>	*			(,num			(,num
<factor>	(,num			$,+,-,*	$,+,-,*		$,+,-,*

Pass 2:

	First	prod 1	prod 2	prod 4	prod 5	prod 7	Follow
<exp>	(,num	$,+,-				)	$,+,-,)
<addop>	+,-	(,num					(,num
<term>	(,num	$,+,-,)	$,+,-,)	*			$,+,-,*,)
<mulop>	*			(,num			(,num
<factor>	(,num			$,+,-,*,)	$,+,-,*,)		$,+,-,*,)

Pass 3: no change

STEP 4. Compute the predict sets for each production. In this case it is essentially the first set of the first symbol on the right hand side:

1)	<exp> ::= <exp> <addop> <term>	(,num
2)	<exp> ::= <term>	(,num
3)	<addop> ::= + | -	+,-
4)	<term> ::= <term> <mulop> <factor>	(,num
5)	<term> ::= <factor>	(,num
6)	<mulop> ::= *	*
7)	<factor> ::= (<exp>)	(
8)	<factor> ::= num	num

The sets on the LEFT for an expression on the RIGHT must have empty intersections.

STEP 5: compute the LL parse table.

Remember this is a top-down parser so given the nonterm on left of table and terminal at top of table what production should we use which is M(A, t) in the table.

M(A, t) where A is \elemof NONTERMS and t \elemof computePredict(P_i)

M	$	+	-	*	(	)	num

```
<exp>    stop                          1,2          1,2
<addop>          3      3
<term>                                 4,5          4,5
<mulop>                        6
<factor>                               7            8

Note:
1. since no \epsilons are present Follow sets are not needed but
it was good practice.  The next example is more complicated.

IMPORTANT:
Because M(<exp>, '(') can be a 1 OR 2 the machine is not well
defined!!!  This means that the grammar we gave is NOT an LL(1)
grammar.  The problem that the LL(1) parser is suffering is the same
one we had with recursive descent parsing.   We can fix this.  Let's
look at an example of the fix before we generalize.



------------------------------------------------------------------------
========================================================================
EXAMPLE 2: an example with \epsilon in the grammar

Take the same grammar as in example 1:

<exp> ::= <exp> <addop> <term> | <term>
<addop> ::= + | -
<term> ::= <term> <mulop> <factor> | <factor>
<mulop> ::= *
<factor> ::= ( <exp> ) | num

We will begin by removing left recursion as in section 4.2.3
creating two new nonterminals: expx and termx.

Step 0: remove left recursion

<exp>   ::= <term> <expx>
<expx>  ::= <addop> <term> <expx> | \epsilon
<addop> ::= + | -
<term>  ::= <factor> <termx>
<termx> ::= <mulop> <factor> <termx> | \epsilon
<mulop> ::= *
<factor> ::= ( <exp> ) | num


STEP 1: remove alternation and list the terms and nonterms for clarity

0) <start> ::= <exp> $
```

```
1) <exp>   ::= <term> <expx>
2) <expx>  ::= <addop> <term> <expx>
3) <expx>  ::= \epsilon
4) <addop> ::= + | -
5) <term>  ::= <factor> <termx>
6) <termx> ::= <mulop> <factor> <termx>
7) <termx> ::= \epsilon
8) <mulop> ::= *
9) <factor> ::= ( <exp> )
a) <factor> ::= num

TERMS = {+, -, *, (, ), num}
NONTERMS = {<exp>, <expx>, <addop>, <term>, <termx>, <mulop>, <factor>}


STEP 2: compute the first set

          pass 1          pass 2          pass 3
<start>   first(exp)      first(exp)      (,num
<exp>     first(term)     first(factor)   (,num
<expx>    +,-,\epsilon    +,-,\epsilon    +,-,\epsilon
<addop>   +,-             +,-             +,-
<term>    first(factor)   (,num           (,num
<termx>   *,\epsilon      *,\epsilon      *,\epsilon
<mulop>   *               *               *
<factor>  (,num           (,num           (,num

Note: \epsilon occurs only where the nonterminal can disappear by
application of a production.

STEP 3: compute the follow set
Note: only productions 0,1,2,5,6,9 affect the follow sets (ignoring nullable NTs)

0) <start> ::= <exp> $
1) <exp>   ::= <term> <expx>
2) <expx>  ::= <addop> <term> <expx>
3) <expx>  ::= \epsilon
5) <term>  ::= <factor> <termx>
6) <termx> ::= <mulop> <factor> <termx>
7) <termx> ::= \epsilon
9) <factor> ::= ( <exp> )

The following shows what happens to the follow sets as each production
is analyzed.  Note for example that we add in fol(exp) for term because
in prod 1 <expx> can go to epsilon in production 3!

Account for each production:
```

```
           First          prod 0&1  prod 2          prod 5      prod 6       prod 9
<start>    (,num
<exp>      (,num          $                                                  )
<expx>     +,-,\epsilon   fol(exp)  fol(expx)
<addop>    +,-                      fst(term)
<term>     (,num          fst(expx) fst(expx)
                          & fol(exp) & fol(expx)
<termx>    *,\epsilon                                 fol(term)  fol(termx)
<mulop>    *                                                     fst(factor)
<factor>   (,num                                      fst(termx) fst(termx)
                                                      & fol(term) & fol(termx)
```

Group by nonTerminal. Remember Follow sets do not have \epsilons

```
           First          Follow
<start>    (,num
<exp>      (,num          $,)
<expx>     +,-,\epsilon   fol(exp)
<addop>    +,-            (,num
<term>     (,num          +,- & fol(exp) & fol(expx)
<termx>    *,\epsilon     fol(term) & fol(termx)
<mulop>    *              (,num
<factor>   (,num          * & fol(term) & fol(termx)
```

```
           First          Follow
<start>    (,num
<exp>      (,num          $,)
<expx>     +,-,\epsilon   $,)
<addop>    +,-            (,num
<term>     (,num          +,-,$,)
<termx>    *,\epsilon     +,-,$,)
<mulop>    *              (,num
<factor>   (,num          *,+,-,$,)
```

pass 0:

Initialize fol(exp) = $

```
           First          Follow
<exp>      (,num          $,)
<expx>     +,-,\epsilon   $,)
<addop>    +,-            (,num
<term>     (,num          +,-,$,)
<termx>    *,\epsilon     +,-,$,)
```

```
<mulop>    *             (,num
<factor>   (,num         *,+,-,$,)
```

STEP 4. Compute the predict sets:

	production	Predict Set	Predict Set
1)	<exp> ::= <term> <expx>	First[term]	(,num
2)	<expx> ::= <addop> <term> <expx>	First[addop]	+,-
3)	<expx> ::= \epsilon	Follow[expx]	$,)
4)	<addop> ::= + | -	First[+|-]	+,-
5)	<term> ::= <factor> <termx>	First[factor]	(,num
6)	<termx> ::= <mulop> <factor> <termx>	First[mulop]	*
7)	<termx> ::= \epsilon	Follow[termx]	+,-,$,)
8)	<mulop> ::= *	First[*]	*
9)	<factor> ::= (<exp>)	First["("]	(
a)	<factor> ::= num	First[num]	num

STEP 5. Create M(NONTERMS, TERMS)

M(A, t) where A is \elemof NONTERMS and t \elemof computePredict(A::=X_1 X_2...X_n)

M	$	+	-	*	(	)	num
<exp>					1		1
<expx>	3	2	2			3	
<addop>		4	4				
<term>					5		5
<termx>	7	7	7	6		7	
<mulop>				8			
<factor>					9		a

RUN THE EXAMPLE ON SOME INPUT

PARSE STACK	INPUT	production
exp $	3+4*5$	
		1
<term> <expx> $	3+4*5$	
		5
<factor><termx><expx>$	3+4*5$	
		a
num<termx><expx>$	3+4*5$	

```
                                        match
<termx><expx>$               +4*5$
                                        7
<expx>$                      +4*5$
                                        2
<addop><term><expx>$         +4*5$
                                        4
(+|-)<term><expx>$           +4*5$
                                        match
<term><expx>$                 4*5$
                                        5
<factor><termx><expx>$        4*5$
                                        a
num<termx><expx>$             4*5$
                                        match
<termx><expx>$                *5$
                                        6
<mulop><factor><termx><expx>$ *5$
                                        8
*<factor><termx><expx>$       *5$
                                        match
<factor><termx><expx>$         5$
                                        a
num<termx><expx>$              5$
                                        match
<termx><expx>$                  $
                                        7
<expx>$                         $
                                        3
$                               $
                                        match

Holy cow!  It works!


Postscript:

A grammar is LL(1) if:
1. For every production  A::=a_1 | a_2 | ...
     forall i, j i\neq j: First[a_i] \intersect First[a_j] is empty
AND
2. if \eps \elemof First[A] then First[A] \intersect Follow[A] is empty
```