J.C. VAN VLIET
THE PROGRAMS "RELATIONS CONCERNING A CF-GRAMMAR"
AND "LL(1)-CHECKER"
The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.
Abstract

In a previous report by Grune a.o. [1] the practical use of two algorithms is illustrated. The first algorithm determines which relations, such as "contains" or "may follow", may hold between the terminal productions of notions and/or symbols of a given context-free grammar, the second one verifies the LL(1)-ness of context-free grammars. In this report the ALGOL 60 programs that perform the corresponding calculations are given, together with some small examples.
# Contents

1. Introduction 1
   1.1 Reading the input 2
   1.2 Other machine-dependent features 2
2. Relations concerning a cf-grammar 5
   2.1 The program 7
   2.2 An example 22
3. LL(1)-checker 32
   3.1 The program 34
   3.2 Two examples 51
References 55
1. Introduction

In [1], some methods are explained that are employed by our group in the construction of an ALGOL 68 compiler. Amongst others, an algorithm is given to determine which relations, such as "contains" or "may follow", may hold between the terminal productions of notions and/or the symbols of a given context-free grammar, and an algorithm to verify the LL(1)-ness of context-free grammars. In fact, both these algorithms are well-known and have already been investigated by many others (see, e.g., [2,3]). Whereas in [1] the emphasis lies on the illustration of the practical use of those algorithms, in this report the actual programs (written in ALGOL 60) are given that perform the corresponding calculations. They are termed: "Relations concerning a cf-grammar" and "LL(1)-checker", respectively.

The input of both programs is a context-free grammar of the following form: first, all terminal symbols of the grammar are given, each one separated from the next by a semicolon, the last one followed by a point. Second, the production rules of the grammar are given, one rule for each non-terminal. Those rules should be given according to the following grammar:

```
rule: left hand side, colon, alternatives, point.
left hand side: notion.
alternatives: alternative option; alternative option, semicolon,
            alternatives.
alternative option: alternative;.
alternative: member; member, comma, alternative.
member: notion; open, notion list, close.
notion list: notion; notion, comma, notion list.
```

Whenever a list of notions is placed between the symbols "open" ("(") and "close" ")", this makes the list optional. Layout characters and comment (placed between the braces "[" and "]") are skipped, except for layout characters within notions, in which case 1 space is honoured to increase readability of the (quite bulky) output.
The author would like to thank D. Grune and L. Meertens for their valuable suggestions.

1.1 Reading the input

The input is read by a simple parser, which replaces each notion by some number for further use. A correct grammar is parsed correctly and no incorrect grammar passes unnoticed. As a notion may consist of only 27 distinct characters (26 letters and the space symbol), a 32-radix number system is used (i.e., 5 bits per symbol). On our machine (an EL X8 of Philips-Electrologica) one word contains 27 bits, so 5 symbols are packed per word. On other machines, some adaptations may be necessary: max int (= maximum integral number), part of the routine next symbol, which performs the packing, and the routine get notion at, which does the unpacking.

Reading the input is done through the external procedure resymbol. Each time resymbol is called, the character code of the next symbol of the input stream is delivered. Our character code is such that the letters a through z are represented by a contiguous set of integers (from the value of letter a up to letter a + 25). This property is used by the routines next symbol (packing), letter and get notion at (unpacking). All other character codes used are represented by variables with suggestive identifiers, initialized at the beginning to the proper values.

1.2 Other machine-dependent features

Both programs make use of the following output procedures:

procedure printtext (s); string s;
The string s is printed.

procedure absfixt (n, m, a); value n, m, a; integer n, m; real a;
For m = 0, the integral value of a is printed in n positions without a sign.

procedure nocr;
A new line is generated.
procedure carriage (n); value n; integer n;
             This procedure generates n new lines.

procedure newpage;
             A new page is generated.

procedure space (n); value n; integer n;
             This procedure generates n spaces.

procedure prsym (n); value n; integer n;
             The printing equivalent of resymbol. The character corresponding to
             the value of n, is printed.

integer procedure printpos;
             An informative procedure, delivering the current position on the line.
             The value delivered is 0 in case the line is empty. The positions on
             the line are numbered from 0 up to "line width". Thus, it indicates
             the first free position.

integer procedure linenumber;
             The value of the function designator linenumber is the number of the
             line we are currently printing on. Thus, its value is 1 after a new
             page is given.

In addition, the programs make use of four other machine-dependent proce-
dures:

procedure exit;
             A call of this procedure terminates the program.

integer procedure available;
             The value of this function designator is the number of words in memory
             that is still available.
integer procedure stringsymbol (i,s); value i; integer i; string s;
The character code of the i-th symbol of the string s is delivered,
where i counts from 0.

procedure fix (n, m, x, a); value n, m, x; integer n, m; real a;
integer array a;
For m = 0, the digits of the number in x are placed in consecutive
elements of a, starting with a [2] (a [1] contains the sign of x),
up to a [n + 2] (which always contains a space).

On the output of the program "Relations concerning a cf-grammar", the
linewidth is adjusted to 70. This can be changed by adjusting the integer
variable line width.

Some remarks should be given about transferring those programs to
other machines. On our machine, booleans are packed, i.e., the word length
is 27 bits, so 27 booleans are packed in one word. But still, a grammar with
say 250 distinct notions could not be run without more (in the initialization
part of both programs, the integer variable max notion is given a
value indicating an upperbound of the number of distinct notions according
to which the size of the arrays is determined; the remaining space is used
for storing the notions and the grammar). In one of the programs, as listed
here, ten huge boolean matrices are used. They are, however, not necessary
in parallel, so some of them may share memory, and can be swapped in and
out whenever necessary. However, on a machine which does not pack booleans,
it will, even then, probably be impossible to digest such a grammar. In
this case the user has to do the packing himself and rewrite the routines
transitive closure, mul 2 and mul 3 as well, and adjust several others.
2. Relations concerning a cf-grammar

The relations, dealt with in this program, are "may contain", "may be contained in", "may begin with", "may be the begin of", "may end with", "may be the end of" (, all of which are transitive,) and "may follow" and "may precede". A short description of the outcome of this program may suffice here; a more extensive description, together with an elaborate example, can be found in [1].

The first relation given is "may contain". We say a "WITHIN" \( \beta \) if \( \beta \) occurs in the right hand side of the production rule for \( \alpha \). "may contain" may then be defined as: may contain = WITHIN\(^+\), where the + indicates the transitive closure of the considered relation.

We say a "FIRST" \( \beta \) if \( \beta \) occurs as the first member of some direct production of \( \alpha \). "may begin with" is then given by: may begin with = FIRST\(^+\). Similarly, we define "LAST", so that: may end with = LAST\(^+\). We say a "FOLLOW" \( \beta \) if, in some production rule the succession of members \( \beta, \gamma_1, \ldots, \gamma_n, \) \( \alpha \) occurs, \( n \geq 0 \) and \( \gamma_i \) produces empty \( (1 \leq i \leq n) \). "may follow" can then be computed as follows: may follow = FIRST\(^T\) FOLLOW LAST\(^*\), that is, \( \alpha \) may follow \( \beta \) if there exist \( \gamma \) and \( \delta \) such that \( \gamma \) FOLLOW \( \delta, \gamma = \alpha \) or \( \gamma \) may begin with \( \alpha \), and \( \delta = \beta \) or \( \delta \) may end with \( \beta \). (The * indicates the reflexive transitive closure.) The other relations given are simply the transpose of one of the relations treated above.

For each relation \( R \) a list is given of entries for each of the notions, where an entry of the form "notion - \( s_1; \ldots; s_n \)" indicates that \( \{s|\text{notion } R \ s \} = \{s_1, \ldots, s_n\} \). An entry of the form "notion\(_i\), \ldots, notion\(_m\) - \( s_1; \ldots; s_n \)" is an abbreviation of \( m \) entries of the form "notion\(_i\) - \( s_1; \ldots; s_n \)", \( 1 \leq i \leq m \).

In each entry, the part "\( s_1; \ldots; s_n \)" is split up into two parts, the first one of the form "\( s_1; \ldots; s_j \)\)", containing only terminals (and preceded by the string (terminals:)), the second one of the form "\( s_{j+1}; \ldots; s_n \)\)", containing only non-terminals (and preceded by "(non-terminals:)\)". If one of those parts is empty, the string preceding it is also omitted.

On the output, the entries of each relation are numbered, this number, together with a two-letter code preceding it, being a reference for the alphabetical listing of the notions given at the end.
In an entry, one or more of the $s_i$ are marked with an asterisk. If, for a relation $R$, some $s_j$ is unmarked, this implies that there is a marked $s_i$ such that $s_i Q s_j$, where $Q$ is $R$ (is "may begin with", is "may end with") if $R$ is a transitive relation (is "may follow", is "may precede"). E.g., for the relation \textsc{FIRST}*, the subset which is \textsc{FIRST} is marked. Also, one or more of the notion $i$ may be marked with a plus, indicating that this relation $R$ is reflexive with respect to notion $i$, i.e., notion $i \ R$ notion $i$ holds.

In section 2.1, the ALGOL-60-text of the program is given, in section 2.2, a small example of a grammar and the resulting output is presented.
2.1 The program

begin comment relations concerning a cf-grammar, september 1973;
  integer end of file, eof, space char, tab char, n clr char,
  sub char, bus char, stock, nil, comma char, comma,
  point char, point, colon char, colon, semicolon char,
  semicolon, open char, open, close char, close, empty,
  list pointer, max int, first notion, new defined,
  head of tree, top, error count, sym, number, numb,
  notion, max notion, start notion, harmless, minus char,
  n of adm cells, upperbound, i, star, symbols, letter a,
  space repr, bits per word, line width, plus char,
  zero char;
  integer array t32[0 : 4];

initialization part:
end of file:= eof:= -4096; space char:= 93; tab char:= 118;
 n clr char:= 119; sub char:= 100; bus char:= 101; nil:= -1;
 comma char:= comma:= 87; point char:= 88; point:= -5;
 colon char:= colon:= 90; semicolon char:= 91; semicolon:= -4;
 open char:= 98; open:= -2; close char:= 99; close:= -3; empty:= 0;
 max int:= 67 108 865; first notion:= 1; new defined:= -1;
 t32[0]: = 1 048 576; t32[1]: = 32 768; t32[2]: = 1 024; t32[3]: = 32;
 t32[4]: = 1; list pointer:= 1; star:= 66; error count:= 0;
 harmless:= -1; n of adm cells:= 10; letter a:= 10; zero char:= 0;
 minus char:= 65; space repr:= 27; bits per word:= 27;
 line width:= 70; plus char:= 64;
 max notion:= 300;
 upperbound:= top:= available - max notion - 10 \times max notion \times
 max notion / bits per word - 2000;
 printtext(4(upperbound array list:4)); absfixt(5, 0, upperbound);
 n clr; n clr;

begin integer array list[1 : upperbound],
  notion link[1 : max notion];
  comment 'notion link' contains pointers to notions, stored in
  'list', as follows:
  left
  right
  7 places for the index
  number(also used for the index)
  pointer: text 1
  :
  text last - max int,
 starting from 1 up to 'list pointer'.
 the essential part of the grammar is stored in 'list',
 from 'upperbound' to 'top' (i.e. backwards);
 boolean array first, first star, last, last star, follow,
 within, within star, aux1, aux2, aux3
 [1 : max notion, 1 : max notion],
 possibly empty[1 : max notion];
comment reading department;

integer procedure char;
begin integer i, j;
repeat: i:= char:= resymbol;
  if i ≠ end of file then prsym(i);
  if i = tab char v i = nlcr char then goto repeat else
  if i = space char then
    begin j:= resymbol;
      for i:= j while i = space char v i = tab char v
        i = nlcr char v i = sub char do
        begin if i = sub char then
            begin for i:= sub char, j while j ≠ bus char
            j ≠ end of file do
              begin prsym(i); j:= resymbol end;
              if j = bus char then
                begin prsym(j); j:= resymbol end
              else error(4 premature end of file)
            end
          else begin prsym(i); j:= resymbol end
        end
        if j ≠ end of file then prsym(j);
        if letter(j) then stock:= j else char:= j
      end
      else if i = sub char then
        skip comment:
        begin i:= char:= resymbol; prsym(i);
          if i = bus char then goto repeat;
          if i ≠ end of file then goto skip comment
        end
    end char;

procedure next symbol;
comment this procedure yields a symbol in "sym", and,
if it is a notion, yields its number in "numb";
begin
again: if sym = end of file then else
  if stock = nil then sym:= char else
    begin sym:= stock; stock:= nil end;
  if sym = space char then goto again;
  if sym = comma char then sym:= comma else
  if sym = point char then sym:= point else
  if sym = colon char then sym:= colon else
  if sym = semicolon char then sym:= semicolon else
  if sym = open char then sym:= open else
  if sym = close char then sym:= close else
  if sym = end of file then sym:= eof else
  if letter(sym) then
    begin integer j, aux, sm;
      number:= number + 1;
for j := nil, nil, empty, empty, empty, empty, empty, empty, empty, empty, empty, number do store(j);
aux := j := 0; start notion := list pointer;
for sm := sym, sm while letter(sm) do
begin if j = 5 then
  begin store(aux); aux := j := 0 end store in word;
aux := t32[j] \times (if sm = space char then space repr
  else sm = letter a + 1) + aux; j := j + 1;
  if stock = letter then sm := char else
  begin sm := stock; stock := nil end
end pack symbols;
store( - max int + aux); stock := sm;
numb := in tree(start notion, head of tree);
if numb = new defined then
begin numb := new(number); notion link[number] := - start
  notion
end not yet defined
else
  if defined(numb) then numb := new(numb);
sym := notion
end read and store a notion
else
begin error(4illegal character; skipped); goto again end
end next symbol;

integer procedure in tree(start, last ref);
value start, last ref; integer start, last ref;
comment this procedure considers the notion at ”start”. if this
  notion is in the tree, it yields its number and removes
  the notion, otherwise, it adds the notion to the tree
  and yields ”new defined”;
begin integer node, comp;
  boolean found;
  integer procedure compare(one, two); value one, two;
  integer one, two;
  comment compare := sign(one’, -’ two);
  begin integer o, t, so, st, i;
    o := list[one]; t := list[two];
    for i := 1 while o = t \& o > 0 do
      begin one := one + 1; o := list[one]; two := two + 1;
        t := list[two]
    end;
    comment discriminating item found, now analyze it;
    if o < 0 then
      begin o := o + max int; so := - 1 end
    else so := 1;
    if t < 0 then
      begin t := t + max int; st := - 1 end
    else st := 1;
    compare := sign(if o = t then so - st else o - t)
  end compare;
found := false;
comment search tree;
for node := list[last ref] while node # nil ^ ~ found do
begin comp := compare(node, start);
    if comp = 0 then found := true else
    last ref := node - n of adm cells +
        (if comp < 0 then 1 else 0)
end node;
comment analyze result: ;
if found then
begin listpointer := start - n of adm cells;
    number := number - 1; comment item removed;
    in tree := list[node - 1]
end list[head - 1] contains the number of the notion
else
begin list[last ref] := start; in tree := new defined
end item added
end list in tree;

boolean procedure letter(sym); value sym; integer sym;
letter := letter a <= sym < sym < letter a + 26 v sym = space char;

procedure store(sym); value sym; integer sym;
if list pointer > top then
begin carriage(2); printtext(\"space exhausted\"); exit end
else
begin list[list pointer] := sym; list pointer := list pointer + 1
end store;

procedure store on top(sym); value sym; integer sym;
if top < list pointer then
begin carriage(2); printtext(\"space exhausted\"); exit end
else
begin list[top] := sym; top := top - 1 end store on top;

integer procedure new(number); value number; integer number;
new := - number;

boolean procedure is new(number); value number;
integer number;
is new := number < 0;

procedure define(number); value number; integer number;
notion link[number] := abs(notion link[number]);

boolean procedure defined(number); value number;
integer number;
defined := notion link[number] > 0;
comment the syntax of the input is given by:

```
grammar: terminals, rules, eof.
terminals: notion, point|
    notion, semicolon, terminals.

rules: rule, (rules).
rule: notion, colon, tail, point.
tail: (alternative), (semicolon, tail).
alternative: member, (comma, member).
member: notion|
    open, notion list, close.
notion list: notion, (comma, notion list).
```

the next part reads in such a grammar, puts its structure in the back end of the array "list", and puts the notions in a tree, stored at the front end of "list", obviously, the | stands for:

```plaintext
procedure grammar;
begin number := 0; sym := stock := nil; head of tree := list pointer;
    store(nil); next symbol; terminals; rules
end grammar;

procedure terminals;
terminals:
begin req notion(false);
    if is(new(num)) then define(new(num)) else
        error("terminal occurs twice");
    if is(semicolon, false) then goto terminals else
        if is(point, false) then symbols := number - 1 else
            if is(eof, false) then else
                begin error("error in terminals"); next symbol;
                goto terminals
        end
    end
end terminals;
```
procedure rules;
  rules:
    begin req notion(true);
      if is new(num) then define(new(num)) else
        error("notion already defined");
      if is(colon, false) then else error("colon missing");
    tail: if is(point, true) then
      begin if is(eof, false) then else goto rules end
    else if is(semicolon, true) then goto tail else
      if member then
        rest list:
          begin if is(comma, false) then
            begin if "member then error("alternative wrong");
              goto rest list
            end
            else goto tail
          end
        else if is(eof, false) then
          begin store on top(point); error("premature end of file");
            end
          else
            begin error("incorrect rule"); next symbol; goto tail end
        end
      rules;
    boolean procedure member;
      if is(notion, true) then member:= true else
        if is(open, true) then
          member:= true;
        rest member: req notion(true);
          if is(comma, false) then goto rest member else
            if is(close, true) then else error("option wrong");
          end
        else member:= false;
    procedure req notion(copy); value copy; boolean copy;
      if "is(notion, copy) then
        begin error("notion missing"); numb:= harmless end req notion;
    boolean procedure is(s, copy); value s, copy; integer s;
      boolean copy;
      if s = sym then
        begin is:= true; if copy then
          store on top(if sym = notion then abs(numb) else sym);
          next symbol
        end
      else is:= false;
    procedure error(s); string s;
      begin integer pos;
        pos:= printpos; error count:= error count + 1; carriage(2);
        printtext("error: "); printtext(s); carriage(3); space(pos)
      end error;
comment compute "possibly empty", "within", "within star", "first", "first star", "last", "last star" and "follow";

procedure check for empty productions;
comment check which notions produce empty. the procedure continues until there are no more changes;
begin integer aux, notion, el, i;
   boolean changed, any;
   any:= false;
   for i:= 1 step 1 until number do
      possibly empty[i]:= false;
   check for empty productions: changed:= false;
   for aux:= upperbound, aux - 1 while aux > top do
      notion:= list[aux];
      if possibly empty[notion] then
         begin for el:= list[aux] while el ≠ point do
            aux:= aux - 1
         end was already empty
      else
         begin
            aux:= aux - 1; el:= list[aux]; if el = open then
               begin for el:= list[aux] while el ≠ close do
                  aux:= aux - 1; goto next
               end optional part
            else if el = semicolon v el = point then
               begin for el:= list[aux] while el ≠ point do
                  aux:= aux - 1;
               possibly empty[notion]:= changed:= true
            end empty production found
            else if possibly empty[el] then goto next else
               for el:= list[aux] while el ≠ point do
               if el = semicolon then goto next else
               aux:= aux - 1
         end production rule
end grammar;
if changed then goto check for empty productions;
if any then
begin printtext("the following rules may produce empty:");
nlcr;
   for i:= 1 step 1 until number do
      if possibly empty[i] then
         begin nlcr; print notion(i) end
end
else printtext("no rule produces empty"); newpage
end check for empty productions;
procedure may contain;
comment the relation "directly contains" is stored in "within",
its transitive closure in "within star";
begin integer i, j, aux, el, lhs;
for i := 1 step 1 until number do
  for j := 1 step 1 until number do
    within[i, j] := within star[i, j] := false;
    for aux := upperbound, aux - 1 while aux > top do
      begin lhs := list[aux]; aux := aux - 1;
        for el := list[aux] while el # point do
          begin if el > 0 then
            within[lhs, el] := within star[lhs, el] := true;
            aux := aux - 1
          end
        end
      end
    end
  end
end rule
end grammar;
transitive closure(within star)
end may contain;

procedure may begin with;
comment this procedure determines the relation "may begin with".
the result is stored in 'first' (directly beginning with),
its transitive closure in 'first star';
begin integer i, j, el, aux, notion;
begin boolean optional, errors;
  errors := false;
  for i := 1 step 1 until number do
    for j := 1 step 1 until number do
      first[i, j] := first star[i, j] := false;
      for aux := upperbound, aux - 1 while aux > top do
        begin notion := list[aux]; aux := aux - 1; optional := false;
          for el := list[aux] while el # point do
            begin aux := aux - 1;
              if el = open then optional := true else
                if el = close then optional := false else
                  if el # semicolon then
                    begin first[notion, el] := first star[notion, el] :=
                      true;
                      if possibly empty[el] then else
                        if optional then
                          begin for el := list[aux] while el # close
                            do aux := aux - 1
                          end
                        end
                      end
                    end
                  end
                end
            end
          end
        end
      end
    end
  end
end grammar;
transitive closure(first star)
end may begin with;
procedure may end with;
comment this procedure determines the relation "may end with";
begin integer i, j, n, p, aux, notion, el;
  boolean optional;
  for i:= 1 step 1 until number do
    for j:= 1 step 1 until number do
      last[i, j]:= last star[i, j] := false;
      for aux:= upperbound, aux while aux > top do
        begin notion:= list[aux]; n:= aux; optional:= false;
          for el:= list[aux] while el ≠ point, el do
            aux:= aux - 1; p:= aux + 2;
            for el:= list[p] while p < n do
              begin p:= p + 1;
                if el = open then optional:= false else
                if el = close then optional:= true else
                if el ≠ semicolon then
                  begin last[notion, el]:= last star[notion, el]:= true;
                    if possibly empty[el] then else
                    if optional then
                      begin for el:= list[p] while el ≠ open do
                        p:= p + 1
                      end skip rest of option
                    else
                      for el:= list[p] while el ≠ semicolon ^ p < n
                        do p:= p + 1
                  end
              end production rule
          end grammar;
          transitive closure(last star)
  end may end with;

procedure transitive closure(r); boolean array r;
comment warshall algorithm, see "grammar handling tools", p48;
begin integer i, j, k;
  for i:= 1 step 1 until number do
    for j:= symbols + 1 step 1 until number do
      if r[j, i] then
        begin for k:= 1 step 1 until number do
          if r[i, k] then r[j, k]:= true
        end
  end transitive closure;
procedure mul2(f, v, r); boolean array f, v, r;
comment v transposed x f is computed;
begin integer a, b, c;
  for a:= 1 step 1 until number do
  for b:= 1 step 1 until number do r[a, b]:= false;
  for a:= 1 step 1 until number do
  for c:= 1 step 1 until number do
    if f[c, a] then
      begin for b:= 1 step 1 until number do
        if v[c, b] then r[a, b]:= true
      end
  end mul2;

procedure mul3(f, v, l, r); boolean array f, v, l, r;
comment ”multiplication” of 3 boolean arrays, the result is 
stored in ”r”. see ”grammar handling tools”, p. 49;
begin integer a, b, c, d;
  for a:= 1 step 1 until number do
  for b:= 1 step 1 until number do r[a, b]:= false;
  for c:= 1 step 1 until number do
  for d:= 1 step 1 until number do
    if v[c, d] then
      begin for a:= 1 step 1 until number do
        if f[c, a] then
          begin for b:= 1 step 1 until number do
            if l[d, b] then r[a, b]:= true
          end
        end
    end mul3;

procedure transpose(r) in:(r transp);
boolean array r, r transp;
begin integer i, j;
  for i:= 1 step 1 until number do
  for j:= 1 step 1 until number do r transp[i, j]:= r[j, i]
end transpose;

procedure follow within;
comment this procedure determines the successions of notions 
within the production rules;
begin integer i, j, stackpointer, lwb stack, upb stack, el, 
previous, aux;
  integer array stack[1 : 100];
  comment the upperbound 100 may be erroneous, a good one is 
the maximum number of notions in an alternative, 
but this will very likely be less than 100;
  for i:= 1 step 1 until number do
  for j:= 1 step 1 until number do follow[i, j]:= false;
for aux := upperbound, aux - 1 while aux > top do
  for el := semicolon, list[aux] while el ≠ point do
    begin aux := aux - 1;
      if el = open then
        begin if previous ≠ nil then
            begin stackpointer := stackpointer + 1;
                stack[stackpointer] := previous
            end;
            upb stack := stackpointer + 1
        end
      else if el = close then lwb stack := upb stack := 1
      else if el = semicolon then
        begin stackpointer := 0; lwb stack := upb stack := 1;
            previous := nil
        end
    end
  else
    begin for i := lwb stack step 1 until stackpointer do
        follow[el, stack[i]] := true;
        if previous ≠ nil then
          begin follow[el, previous] := true;
                if possibly empty[el] then
                  begin stackpointer := stackpointer + 1;
                      stack[stackpointer] := previous
                  end
                else
                  begin stackpointer := upb stack - 1;
                      lwb stack := upb stack
                  end
            end;
      previous := el
    end
  end rule and grammar
end follow within;

comment output department;

procedure print list of notions;
begin integer i;
  print text(\text{"list of notions, each notion preceded by its number\text{"}); nlcr;
  print text(\text{"and, if not defined by an asterisk\text{"}); nlcr;
  for i := 1 step 1 until number do
    begin nlcr;
      if defined(i) then
        begin prsym(\text{star}); define(1); error count := error count + 1
        end
      else prsym(spacechar);
        absfix(4, 0, i); print notion(i)
    end
  end print list of notions;
procedure print notion(number); value number; integer number;
  print notion at(notion link[number]);

procedure print notion at(index); value index; integer index;
begin integer i, n;
  integer array a[1 : 100];
  comment this upperbound is rather arbitrary, notions of
    length > 100 are considered rare;
  get notion at(index) of length:(n) in:(a);
  for i := 1 step 1 until n do prsym(a[i])
end print notion at;

procedure get notion at(index) of length:(n) in:(a);
value index; integer index, n; integer array a;
begin integer el, j, k;
  n := 0;
  for el := list[index] while el > 0, el + max int do
    begin for k := 0, k + 1 while k < 5 ^ el > 0 do
      begin j := el div t32[k]; el := el - j * t32[k]; n := n +1;
        a[n] := if j = space repr then space char else
          j + letter a - 1
      end;
      index := index + 1
    end
end get notion;

procedure print relation(r, marked, heading, index,
  kind of relation);
value index; integer index; boolean array r, marked;
string heading, kind of relation;
begin integer i, notion, count, aux;
  boolean in lhs, first terminal, first nonterminal;
  boolean array printed, union marked[1 : number];

procedure newline;
begin nlcr; if linenumber = 1 then
  begin prnttext(heading); nlcr; nlcr end
end heading;

procedure print notion(number); value number;
integer number;
begin integer n, i;
  integer array a[1 : 100];
  comment see remark at "print notion at";
  get notion at(notion link[number]) of length:(n) in:(a);
  if if printpos < 8 then false else
    printpos + n > line width then
    begin newline; space(if in lhs then 5 else 8) end;
  for i := 1 step 1 until n do prsym(a[i])
end print notion;
for i := 1 step 1 until number do printed[i] := false; nln; printtext(heading); nln; nln; count := 0; for notion := 1 step 1 until number do if ~printed[notion] then begin aux := count := count + 1; in lhs := true; space(if count < 10 then 2 else if count < 100 then 1 else 0); printtext(kind of relation); if aux > 99 then begin i := aux div 100; prsym(i + zero char); aux := aux - i * 10 end; if aux > 9 then begin i := aux div 10; prsym(i + zero char); aux := aux - i * 10 end; prsym(aux + zero char); prsym(space char); print notion(notion); list[notion link[notion] - index] := count; if r[notion, notion] then prsym(plus char); for i := 1 step 1 until number do union marked[i] := marked[notion, i]; for i := notion + 1 step 1 until number do if ~printed[i] then begin boolean the same; the same := true; aux := 0; for aux := aux + 1 while aux < number ~ the same do the same := r[notion, aux] >= r[i, aux]; if the same then begin prsym(comma char); prsym(space char); print notion(i); list[notion link[i] - index] := count; if r[i, i] then prsym(plus char); for aux := 1 step 1 until number do if marked[i, aux] then union marked[aux] := true; printed[i] := true end the same rhs, so taken together end grouping; prsym(minus char); newline; in lhs := false; first terminal := first nonterminal := true;
for i := 1 step 1 until number do
if $r[notion, 1]$ then
  begin if first terminal $^\wedge i \leq$ symbols then
    begin printtext($\{ (\text{terminals:}) \}$);
      first terminal := false
    end first terminal
  else if first nonterminal $^\wedge i >$ symbols then
    begin if $^\wedge$ first terminal then
      begin prsym(point char); newline end close ;
        printtext($\{ (\text{nonterminals:}) \}$);
      first nonterminal := false
    end first nonterminal
  else
    begin prsym(semicolon char); prsym(space char) end;
    print notion(i);
    if union marked[i] then prsym(star)
  end print notion i;
    prsym(point char); newline; newline
  end notion
end print relation;

procedure print index(head); value head; integer head;
begin integer i, j;
  boolean first;
  integer array a[1 : 5];
  if list[head - n of adm cells] $\neq$ nil then
    print index(list[head - n of adm cells]);
  print notion at(head);
  if printpos < 28 then space(28 - printpos) else
    begin nlcr; space(28) end;
  first := true;
  for j := 0 step 1 until 7 do
    begin prsym(comma char); prsym(space char) end
    else first := false;
    for i := 4 * j, i + 1 do
      prsym(stringsymbol(i, \text{mc, ci, bw, bo, ew, eo, mf, mp}));
      fix(3, 0, list[head + j - 8], a);
      for i := 1 step 1 until 5 do
        if a[i] < letter a then prsym(a[i] + zero char)
  end;
  nlcr; if list[head - n of adm cells + 1] $\neq$ nil then
    print index(list[head - n of adm cells + 1])
end print index;
main program:
    grammar;
    newpage;
    print list of notions;
    if error count > 0 then
        begin carriage(3); absfixt(3, 0, error count);
            printtext(4 errors in grammar); exit
        end;
    newpage;
    check for empty productions;
    may contain;
    print relation(within star, within, \$may contain (mc): ab - a; b.,
    8, \$mc$);
    transpose(within star) in:(aux1);
    transpose(within) in:(aux2);
    print relation(aux1, aux2, \$may be contained in (ci): a, b - ab.,
    7, \$ci$);
    may begin with;
    print relation(first star, first, \$may begin with (bw): ab - a.,
    6, \$bw$);
    transpose(first star) in:(aux1);
    transpose(first) in:(aux2);
    print relation(aux1, aux2, \$may be the begin of (bo): a - ab.,
    5, \$bo$);
    may end with;
    print relation(last star, last, \$may end with (ew): ab - b.,
    4, \$ew$);
    transpose(last star) in:(aux1);
    transpose(last) in:(aux2);
    print relation(aux1, aux2, \$may be the end of (eo): b - ab.,
    3, \$eo$);
    follow within;
    for i:= 1 step 1 until number do
        first star[i, i]:= last star[i, i]:= true;
        mul3(first star, follow, last star, aux1);
        mul2(first star, follow, aux2);
        print relation(aux1, aux2, \$may follow (mf): b - a., 2, \$mf$);
        transpose(aux1) in:(aux3);
        transpose(follow) in:(aux1);
        mul2(last star, aux1, aux2);
        print relation(aux3, aux2, \$may precede (mp): a - b., 1, \$mp$);
    newpage;
    print index(list[head of tree])
end
end
2.2 An example

UPPERBOUND ARRAY LIST: 20431

[INPUT:]

[Terminal Symbols]
COLON; POINT; SEMICOLON; COMMA; OPEN; CLOSE; NOTION; EOF.

Grammar: TERMINALS, RULES, EOF.

TERMINALS: NOTION, REST TERMINALS.
REST TERMINALS: POINT; SEMICOLON, TERMINALS.

RULES: RULE, (RULES).
RULE: LEFT HAND SIDE, COLON, ALTERNATIVES, POINT.
LEFT HAND SIDE: NOTION.
ALTERNATIVES: (ALTERNATIVE), (SEMICOLON, ALTERNATIVES).
ALTERNATIVE: MEMBER, (COMMA, ALTERNATIVE).
MEMBER: NOTION; OPEN, NOTION LIST, CLOSE.
NOTION LIST: NOTION, (COMMA, NOTION LIST).
MAY CONTAIN (MC): AB - AJ B.

MC1 COLON, POINT, SEMICOLON, COMMA, OPEN, CLOSE, NOTION, EOF*. 

MC2 GRAMMAR-
(TERMINALS): COLON; POINT; SEMICOLON; COMMA; OPEN; CLOSE; NOTION; EOF*.
(NONTERMINALS): TERMINALS*; RULES*; REST TERMINALS; RULE; LEFT HAND SIDE; ALTERNATIVES; ALTERNATIVE; MEMBER; NOTION LIST.

MC3 TERMINALS+, REST TERMINALS+-
(TERMINALS): POINT*; SEMICOLON*; NOTION*.
(NONTERMINALS): TERMINALS*; REST TERMINALS*.

MC4 RULES++
(TERMINALS): COLON; POINT; SEMICOLON; COMMA; OPEN; CLOSE; NOTION.
(NONTERMINALS): RULES*; RULE*; LEFT HAND SIDE; ALTERNATIVES; ALTERNATIVE; MEMBER; NOTION LIST.

MC5 RULE-
(TERMINALS): COLON; POINT; SEMICOLON; COMMA; OPEN; CLOSE; NOTION.
(NONTERMINALS): LEFT HAND SIDE*; ALTERNATIVES*; ALTERNATIVE; MEMBER; NOTION LIST.

MC6 LEFT HAND SIDE-
(TERMINALS): NOTION*.

MC7 ALTERNATIVES++
(TERMINALS): SEMICOLON*; COMMA; OPEN; CLOSE; NOTION.
(NONTERMINALS): ALTERNATIVES*; ALTERNATIVE*; MEMBER; NOTION LIST.

MC8 ALTERNATIVE--
(TERMINALS): COMMA*; OPEN; CLOSE; NOTION.
(NONTERMINALS): ALTERNATIVE*; MEMBER*; NOTION LIST.

MC9 MEMBER-
(TERMINALS): COMMA; OPEN*; CLOSE*; NOTION*.
(NONTERMINALS): NOTION LIST*.

MC10 NOTION LIST++
(TERMINALS): COMMA*; NOTION*.
(NONTERMINALS): NOTION LIST*.
MAY BE CONTAINED IN (C1): A, B - AB.

C1 COLON, LEFT HAND SIDE-
  (NONTERMINALS1) GRAMMAR; RULES; RULE*.

C12 POINT-
  (NONTERMINALS1) GRAMMAR; TERMINALS; RULES; REST TERMINALS*; RULE*.

C13 SEMICOLON-
  (NONTERMINALS1) GRAMMAR; TERMINALS; RULES; REST TERMINALS*; RULE;
  ALTERNATIVES*.

C14 COMMA, NOTION LIST-
  (NONTERMINALS1) GRAMMAR; RULES; RULE; ALTERNATIVES; ALTERNATIVE*;
  MEMBER*; NOTION LIST*.

C15 OPEN, CLOSE-
  (NONTERMINALS1) GRAMMAR; RULES; RULE; ALTERNATIVES; ALTERNATIVE;
  MEMBER*.

C16 NOTION-
  (NONTERMINALS1) GRAMMAR; TERMINALS*; RULES; REST TERMINALS; RULE;
  LEFT HAND SIDE*; ALTERNATIVES; ALTERNATIVE; MEMBER*;
  NOTION LIST*.

C17 EOF-
  (NONTERMINALS1) GRAMMAR*.

C18 GRAMMAR-

C19 TERMINALS*, REST TERMINALS*-
  (NONTERMINALS1) GRAMMAR*; TERMINALS*; REST TERMINALS*.

C10 RULES*, RULE-
  (NONTERMINALS1) GRAMMAR*; RULE*.

C11 ALTERNATIVES*-
  (NONTERMINALS1) GRAMMAR; RULES; RULE*; ALTERNATIVES*.

C12 ALTERNATIVE*, MEMBER-
  (NONTERMINALS1) GRAMMAR; RULES; RULE; ALTERNATIVES*; ALTERNATIVE*,
  MEMBER.
MAY BEGIN WITH (BW): AB - A.

BW1 COLON, POINT, SEMICOLON, COMMA, OPEN, CLOSE, NOTION, EOF-

BW2 GRAMMAR-
(TERMINALS:) NOTION,
(NONTERMINALS:) TERMINALS*.

BW3 TERMINALS, LEFT HAND SIDE, NOTION LIST-
(TERMINALS:) NOTION*.

BW4 RULES-
(TERMINALS:) NOTION,
(NONTERMINALS:) RULE*; LEFT HAND SIDE.

BW5 REST TERMINALS-
(TERMINALS:) POINT*; SEMICOLON*.

BW6 RULE-
(TERMINALS:) NOTION,
(NONTERMINALS:) LEFT HAND SIDE*.

BW7 ALTERNATIVES-
(TERMINALS:) SEMICOLON*; OPEN; NOTION,
(NONTERMINALS:) ALTERNATIVE*; MEMBER.

BW8 ALTERNATIVE-
(TERMINALS:) OPEN; NOTION,
(NONTERMINALS:) MEMBER*.

BW9 MEMBER-
(TERMINALS:) OPEN*; NOTION*.
MAY BE THE BEGIN OF (BO): A - AB.

BO1 COLON, COMMA, CLOSE, EOF, GRAMMAR, RULES, REST TERMINALS,
ALTERNATIVES, NOTION LIST-

BO2 POINT-
  (NONTERMINALS:) REST TERMINALS*.

BO3 SEMICOLON-
  (NONTERMINALS:) REST TERMINALS*; ALTERNATIVES*.

BO4 OPEN-
  (NONTERMINALS:) ALTERNATIVES; ALTERNATIVE; MEMBER*.

BO5 NOTION-
  (NONTERMINALS:) GRAMMAR; TERMINALS*; RULES; RULE; LEFT HAND SIDE*;
  ALTERNATIVES; ALTERNATIVE; MEMBER*; NOTION LIST*.

BO6 TERMINALS-
  (NONTERMINALS:) GRAMMAR*.

BO7 RULE-
  (NONTERMINALS:) RULE*.

BO8 LEFT HAND SIDE-
  (NONTERMINALS:) RULES; RULE*.

BO9 ALTERNATIVE-
  (NONTERMINALS:) ALTERNATIVES*.

BO10 MEMBER-
  (NONTERMINALS:) ALTERNATIVES; ALTERNATIVE*.
MAY END WITH (EW): AB = B.

EW1 COLON, POINT, SEMICOLON, COMMA, OPEN, CLOSE, NOTION, EOF.

EW2 GRAMMAR-
   (TERMINALS:) EOF*.

EW3 TERMINALS+, REST TERMINALS+  
   (TERMINALS:) POINT*, 
   (NONTERMINALS:) TERMINALS*; REST TERMINALS*.

EW4 RULES+-
   (TERMINALS:) POINT, 
   (NONTERMINALS:) RULES*; RULE*.

EW5 RULE-
   (TERMINALS:) POINT*.

EW6 LEFT HAND SIDE-
   (TERMINALS:) NOTION*.

EW7 ALTERNATIVES+-
   (TERMINALS:) SEMICOLON*; CLOSE, NOTION, 
   (NONTERMINALS:) ALTERNATIVES*; ALTERNATIVE*; MEMBER.

EW8 ALTERNATIVE+-
   (TERMINALS:) CLOSE; NOTION, 
   (NONTERMINALS:) ALTERNATIVE*; MEMBER*.

EW9 MEMBER-
   (TERMINALS:) CLOSE*; NOTION*.

EW10 NOTION LIST+-
   (TERMINALS:) NOTION*, 
   (NONTERMINALS:) NOTION LISTS*.
MAY BE THE END OF (EO): B = AB.

E01 COLON, COMMA, OPEN, GRAMMAR, LEFT HAND SIDE-

E02 POINT-
  (NONTERMINALS!) TERMINALS; RULES; REST TERMINALS*; RULE*.

E03 SEMICOLON, ALTERNATIVES+-
  (NONTERMINALS!) ALTERNATIVES*.

E04 CLOSE-
  (NONTERMINALS!) ALTERNATIVES; ALTERNATIVE; MEMBER*.

E05 NOTION-
  (NONTERMINALS!) LEFT HAND SIDE*; ALTERNATIVES; ALTERNATIVE; MEMBER*;
  NOTION LIST*.

E06 EOF-
  (NONTERMINALS!) GRAMMAR*.

E07 TERMINALS+, REST TERMINALS+-
  (NONTERMINALS!) TERMINALS*; REST TERMINALS*.

E08 RULES+, RULE-
  (NONTERMINALS!) RULES*.

E09 ALTERNATIVE+, MEMBER-
  (NONTERMINALS!) ALTERNATIVES*; ALTERNATIVE*.

E010 NOTION LIST+-
  (NONTERMINALS!) NOTION LIST*.
MAY FOLLOW (MF): B - A.

MF1 COLON-
  (TERMINALS: ) NOTION,
  (NONTERMINALS: ) LEFT HAND SIDE*.

MF2 POINT-
  (TERMINALS: ) COLON*; SEMICOLON; CLOSE; NOTION*.
  (NONTERMINALS: ) ALTERNATIVES*; ALTERNATIVE; MEMBER.

MF3 SEMICOLON+-
  (TERMINALS: ) COLON*; SEMICOLON*; CLOSE; NOTION*.
  (NONTERMINALS: ) ALTERNATIVE*; MEMBER.

MF4 COMMA-
  (TERMINALS: ) CLOSE; NOTION*.
  (NONTERMINALS: ) MEMBER*.

MF5 OPEN, ALTERNATIVE, MEMBER-
  (TERMINALS: ) COLON*; SEMICOLON*; COMMA*.

MF6 CLOSE-
  (TERMINALS: ) NOTION,
  (NONTERMINALS: ) NOTION LIST*.

MF7 NOTION-
  (TERMINALS: ) COLON*; POINT; SEMICOLON*; COMMA*; OPEN*.
  (NONTERMINALS: ) TERMINALS*; REST TERMINALS; RULE*.

MF8 EDF-
  (TERMINALS: ) POINT.
  (NONTERMINALS: ) RULE*; RULE.

MF9 GRAMMAR-

MF10 TERMINALS-
  (TERMINALS: ) SEMICOLON*.

MF11 RULES; RULE*, LEFT HAND SIDE-
  (TERMINALS: ) POINT,
  (NONTERMINALS: ) TERMINALS*; REST TERMINALS; RULE*.

MF12 REST TERMINALS-
  (TERMINALS: ) NOTION*.

MF13 ALTERNATIVES-
  (TERMINALS: ) COLON*; SEMICOLON*.

MF14 NOTION LIST-
  (TERMINALS: ) COMMA*; OPEN*. 

MAY PRECEDE (MP): A → B.

MP1 COLON-
  (TERMINALS:) POINT*; SEMICOLON; OPEN; NOTION.
  (NONTERMINALS:) ALTERNATIVES*; ALTERNATIVE; MEMBER.

MP2 POINT, RULE+-
  (TERMINALS:) NOTION; EOF*.
  (NONTERMINALS:) RULES*; RULE; LEFT HAND SIDE.

MP3 SEMICOLON+-
  (TERMINALS:) POINT*; SEMICOLON; OPEN; NOTION.
  (NONTERMINALS:) TERMINALS*; ALTERNATIVES*; ALTERNATIVE; MEMBER.

MP4 COMMA-
  (TERMINALS:) OPEN; NOTION.
  (NONTERMINALS:) ALTERNATIVE*; MEMBER; NOTION LIST*.

MP5 OPEN-
  (TERMINALS:) NOTION,
  (NONTERMINALS:) NOTION LIST*.

MP6 CLOSE, MEMBER-
  (TERMINALS:) POINT*; SEMICOLON*; COMMA*.

MP7 NOTION-
  (TERMINALS:) COLON*; POINT*; SEMICOLON*; COMMA*; CLOSE*.
  (NONTERMINALS:) REST TERMINALS*.

MP8 EOF, GRAMMAR-

MP9 TERMINALS, REST TERMINALS-
  (TERMINALS:) NOTION.
  (NONTERMINALS:) RULES*; RULE; LEFT HAND SIDE.

MP10 RULES-
  (TERMINALS:) EOF*.

MP11 LEFT HAND SIDE-
  (TERMINALS:) COLON*.

MP12 ALTERNATIVES-
  (TERMINALS:) POINT*.

MP13 ALTERNATIVE-
  (TERMINALS:) POINT*; SEMICOLON*.

MP14 NOTION LIST-
  (TERMINALS:) CLOSE*.
ALTERNATIVE, ALTERNATIVES
CLOSE
COLON
COMMA
EOF
GRAMMAR
LEFT HAND SIDE
MEMBER
NOTION
NOTION LIST
OPEN
POINT
REST TERMINALS
RULE
RULES
SEMICOLON
TERMINALS

MC8, C12, B8, B09, E8, E09, M0, M13
MC7, C11, B7, B00, E7, E00, M13, M12
MC0, C1, B5, B0, E1, E04, M6, M6
MC1, C11, B1, B01, E1, E01, M1, M1
MC0, C14, B0, B01, E1, E01, M4, M4
MC0, C17, B0, B01, E1, E06, M8, M8
MC2, C18, B2, B00, E2, E00, M9, M0
MC6, C10, B1, B08, E6, E0, M1, M11
MC0, C1, B9, B10, E9, E00, M9, M5
MC0, C16, B0, B05, E0, E05, M7, M7
MC0, C10, B4, B03, E10, E010, M14, M14
MC0, C15, B0, B04, E0, E01, M5, M5
MC0, C12, B0, B02, E0, E02, M2, M2
MC0, C13, B5, B00, E1, E03, M12, M0
MC5, C10, B6, B07, E5, E00, M8, M11
MC4, C110, B4, B00, E4, E08, M11, M10
MC0, C13, B0, B03, E0, E03, M3, M3
MC3, C19, B3, B06, E3, E07, M10, M9
3. LL(1)-checker

The necessary requirements for a grammar, written in a form without optional parts and without useless non-terminals (i.e., without non-terminals which either do not produce any finite string, or do not depend on the root of the grammar), to be of type LL(1), are:

1. for any rule of the form $A : a_1 \ldots a_n$, the sets first ($a_1$), ... , first ($a_n$), where first ($a_i$) = {s | $a_i \in \text{FIRST}(s)$}, are mutually disjoint;

2. at most one of the $a_1, \ldots , a_n$ can produce the null string ($\epsilon$);

3. if $a_p$ produces $\epsilon$, then first (A) has no elements in common with follow (A), where follow (A) = {s | s "may follow" A}.

This third requirement is slightly different from, but equivalent to, the one given by Knuth [3], and somewhat easier to check. The equivalence is proved in [1].

The check for LL(1)-ness is done in several steps, corresponding to the requirements listed above:

i. Requirement 1 is checked in two steps: during the computation of FIRST it is tested that no elements of the array "first" are filled twice, i.e., that there are no non-terminals for which two alternatives "directly start" with the same notion (either terminal or non-terminal).

Secondly, all "direct starts", i.e., notions that are the beginning of some non-terminal, are looked at to test that there are no direct starts that may (directly or indirectly) begin with the same terminal. (The test for non-terminals is, obviously, superfluous here.)

ii. Requirement 2 is also checked in two steps: first, it is tested that non-terminals that produce empty ($\epsilon$), do not have more than one alternative producing $\epsilon$ (which is exactly the requirement as stated before).

However, as our grammar has optional parts, the full test is slightly more complicated. If each optional part is replaced by some non-terminal, the production rule for which has two alternatives: $\epsilon$, and the enclosed list of notions, requirement 2 is fulfilled if the enclosed list
of notions does not produce $\varepsilon$. This is tested along with requirement 3.

iii. Requirement 3 is checked more or less similarly to requirement 2. The check is done for notions that produce $\varepsilon$, and for the implicit notions for which the enclosed parts stand.

The test for useless non-terminals is not done. Notions which do not depend on the root of the grammar are of no importance at all, notions which do not produce any finite string are not taken into account.

In section 3.1 a listing of the program is given, section 3.2 shows two small examples, one for a grammar which is not of type LL(1), and one for a grammar which is LL(1) and generates the same language.
3.1 The program

begin comment ll(1)-checker, august-september 1973;
  integer end of file, eof, space char, tab char, n1cr char,
  sub char, bus char, stock, nil, comma char, comma,
  point char, point, colon char, colon, semicolon char,
  semicolon, open char, open, close char, close,
  list pointer, max int, first notion, new defined,
  head of tree, top, error count, sym, number, numb,
  notion, max notion, start notion, harmless, minus char,
  n of adm cells, upperbound, star, symbols, letter a,
  space repr, bits per word;

  boolean ll1;
  integer array t32[0 : 4];

initialization part:
  end of file:= eof:= -4096; space char:= 93; tab char:= 118;
  n1cr char:= 119; sub char:= 100; bus char:= 101; nil:= -1;
  comma char:= comma:= 87; point char:= 88; point:= -5;
  colon char:= colon:= 90; semicolon char:= 91; semicolon:= -4;
  open char:= 98; open:= -2; close char:= 99; close:= -3;
  max int:= 67 108 863; first notion:= 1; new defined:= -1;
  t32[0]:= 1 048 576; t32[1]:= 32 768; t32[2]:= 1 024; t32[3]:= 32;
  t32[4]:= 1; list pointer:= 1; star:= 66; error count:= 0;
  harmless:= -1; n of adm cells:= 3;
  minus char:= 65; letter a:= 10; space repr:= 27; bits per word:= 27;
  max notion:= 300;
  upperbound:= top:= available - max notion - 5 x max notion x
  max notion / bits per word - 2000;
  printf('upperbound array list:'); absfix(5, 0, upperbound);
  n1cr; n1cr;

begin integer array list[1 : upperbound],
  notion link[1 : max notion];

  comment 'notion link' contains pointers to notions, stored in
  'list', as follows:
    left
    right
    number

  pointer: text 1
    text last - max int,
    starting from 1 up to 'listpointer'.
  the essential part of the grammar is stored in 'list'
  from 'upperbound' to 'top' (i.e. backwards);
  boolean array first, first star, last, last star, follow
  [1 : max notion, 1 : max notion],
  possibly empty[1 : max notion];
comment reading department;

integer procedure char;
begin integer i, j;
repeat: i:= char:= resymbol;
if i ≠ end of file then prsym(i);
if i = tab char v i = nlcr char then goto repeat else
if i = space char then
begin j:= resymbol;
for i:= j while i = space char v i = tab char v
i = nlcr char v i = sub char do
begin if i = sub char then
begin for i:= sub char, j while j ≠ bus char ^
j ≠ end of file do
begin prsym(i); j:= resymbol end;
if j = bus char then
begin prsym(j); j:= resymbol end
else error("premature end of file")
end
else
begin prsym(i); j:= resymbol end
end;
if j ≠ end of file then prsym(j);
if letter(j) then stock:= j else char:= j
end
else if i = sub char then
skip comment:
begin i:= char:= resymbol; prsym(i);
if i = bus char then goto repeat;
if i ≠ end of file then goto skip comment
end
end char;

procedure next symbol;
comment this procedure yields a symbol in "sym", and,
if it is a notion, yields its number in "num";
begin
again: if sym = end of file then else
if stock = nil then sym:= char else
begin sym:= stock; stock:= nil end;
if sym = space char then goto again else
if sym = comma char then sym:= comma else
if sym = point char then sym:= point else
if sym = colon char then sym:= colon else
if sym = semicolon char then sym:= semicolon else
if sym = open char then sym:= open else
if sym = close char then sym:= close else
if sym = end of file then sym:= eof else
if letter(sym) then
begin integer j, aux, sm;
number:= number + 1;
for j := nil, nil, number do store(j);
aux := j := 0; start notion := list pointer;
for sm := sym, sm while letter(sm) do
begin if j = 5 then
begin store(aux); aux := j := 0 end store in word;
aux := t32[j] \times (if sm = space char then space repr
else sm - letter a + 1) + aux; j := j + 1;
if stock = nil then sm := char else
begin sm := stock; stock := nil end
end pack symbols;
store(- max int + aux); stock := sm;
numb := in tree(start notion, head of tree);
if numb = new defined then
begin numb := new(number); notion link[number] := - start
notion
end not yet defined
else
if defined(numb) then numb := new(numb);
sym := notion
end read and store a notion
else
begin error(4 illegal character; skipped4); goto again end
end next symbol;

integer procedure in tree(start, last ref);
value start, last ref; integer start, last ref;
comment this procedure considers the notion at "start". if this
notion is in the tree, it yields its number and removes
the notion, otherwise, it adds the notion to the tree
and yields "new defined";

begin integer node, comp;
boolean found;

integer procedure compare(one, two); value one, two;
integer one, two;
comment compare := sign(one - two);
begin integer o, t, so, st, i;
o := list[one]; t := list[two];
for i := 1 while o = t \^ o > 0 do
begin one := one + 1; o := list[one]; two := two + 1;
t := list[two]
end;
comment discriminating item found, now analyze it;
if o < 0 then
begin o := o + max int; so := - 1 end
else so := 1;
if t < 0 then
begin t := t + max int; st := - 1 end
else st := 1;
compare := sign(if o = t then so - st else o - t)
end compare;
found := false;
comment search tree;
for node := list[last ref] while node # nil ^ ~ found do
begin comp := compare(node, start);
   if comp = 0 then found := true else
      last ref := node - n of adm cells +
      (if comp < 0 then 1 else 0)
end node;
comment analyze result:
if found then
begin listpointer := start - n of adm cells;
   number := number - 1; comment item removed;
   in tree := list[node - 1]
end list[head - 1] contains the number of the notion
else
begin list[last ref] := start; in tree := new defined
item added
end in tree;

boolean procedure letter(sym); value sym; integer sym;
letter := letter a < sym ^ sym < letter a + 26 v sym = space char;

procedure store(sym); value sym; integer sym;
if list pointer > top then
begin carriage(2); printtext(4space exhausted); exit
end else
begin list[list pointer] := sym; list pointer := list pointer + 1
end store;

procedure store on top(sym); value sym; integer sym;
if top < list pointer then
begin carriage(2); printtext(4space exhausted); exit
end else
begin list[top] := sym; top := top - 1 end store on top;

integer procedure new(number); value number; integer number;
new := - number;

boolean procedure is new(number); value number; integer number;
is new := number < 0;

procedure define(number); value number; integer number;
notion link[number] := abs(notion link[number]);

boolean procedure defined(number); value number; integer number;
defined := notion link[number] > 0;
the syntax of the input is given by:

grammar: terminals, rules, eof.

terminals: notion, point |
          notion, semicolon, terminals.

rules: rule, (rules).
rule: notion, colon, tail, point.
tail: (alternative), (semicolon, tail).
alternative: member, (comma, member).
member: notion |
     open, notion list, close.
notion list: notion, (comma, notion list).

the next part reads in such a grammar, puts its
structure in the back end of the array "list", and puts
the notions in a tree, stored at the front end of "list".
obviously, the | stands for ;

procedure grammar;
begin number:= 0; sym:= stock:= nil; head of tree:= list pointer;
store(nil); next symbol; terminals; rules
end grammar;

procedure terminals;
terminals:
begin req notion(false);
    if is new(number) then define(new(number)) else
        error(\textit{terminal occurs twice});
    if is(semicolon, false) then goto terminals else
    if is(point, false) then symbols:= number - 1 else
    if is(eof, false) then else
        begin error(\textit{error in terminals}); next symbol;
        goto terminals
    end
end terminals;
procedure rules;
    rules:
    begin req notion(true);
        if is new(num) then define(new(num)) else
            error('notion already defined');
        if is (colon, false) then else error('colon missing');
    tail: if is(point, true) then
        begin if (eof, false) then else goto rules end
        else if is (semicolon, true) then goto tail else
        if member then
    rest list:
        begin if is (comma, false) then
            begin if ' member then error('alternative wrong');
                goto rest list
            else goto tail
        end
        else if (eof, false) then
            begin store on top(point); error('premature end of file');
                end
        else
            begin error('incorrect rule'); next symbol; goto tail end
        end rules;
    boolean procedure member;
    if is(notion, true) then member := true else
    if is(open, true) then
        begin member := true;
            rest member := req notion(true);
        if is (comma, false) then goto rest member else
        if is(close, true) then else error('option wrong');
        end
    else member := false;

procedure req notion(copy); value copy; boolean copy;
    if 'is(notion, copy) then
        begin error('notion missing'); numb := harmless end req notion;

boolean procedure is(s, copy); value s, copy; integer s;
    boolean copy;
    if s = sym then
        begin is := true; if copy then
            store on top(if sym = notion then abs(numb) else sym);
            next symbol
        end
    else is := false;

procedure error(s); string s;
    begin integer pos;
        pos := printpos; error count := error count + 1; carriage(2);
        printtext('error: '); printtext(s); carriage(3); space(pos)
    end error;
comment compute "possibly empty", "first", "first star", "last", "last star" and "follow";

procedure check for empty productions;
comment check which notions produce empty. the procedure continues
until there are no more changes;
begin integer aux, notion, el, i;
   boolean changed, any;
   any := false;
   for aux := 1 step 1 until number do
   possibly empty[aux] := false;
check for empty productions: changed := false;
   for aux := upperbound, aux - 1 while aux > top do
   begin notion := list[aux];
      if possibly empty[notion] then
      begin for el := list[aux] while el ≠ point do
         aux := aux - 1;
      end
      was already empty
      else
      begin
         next: aux := aux - 1; el := list[aux]; if el = open then
         begin for el := list[aux] while el ≠ close do
            aux := aux - 1; goto next
         end optional part
         else if el = semicolon v el = point then
         begin for el := list[aux] while el ≠ point do
            aux := aux - 1;
            possibly empty[notion] := changed := any := true
         end empty production found
         else if possibly empty[el] then goto next else
         for el := list[aux] while el ≠ point do
         if el = semicolon then goto next else
            aux := aux - 1
         end production rule
end grammar;
if changed then goto check for empty productions;
if any then
begin printtext("the following notions may produce empty:");
lcr;
   for i := 1 step 1 until number do
   if possibly empty[i] then
      begin nlc[i]; print notion(i) end
end
else printtext("no rule produces empty");
carriage(5);
check only one alternative yields empty
end check for empty productions;
procedure check only one alternative yields empty;
comment this procedure checks whether not more than one
alternative of a possibly empty notion yields "empty";
begin integer aux, notion, el, numb of empties;
for aux:= upperbound, aux - 1 while aux > top do
begin notion:= list[aux];
  if possibly empty[notion] then
    begin numb of empties:= 0;
    next: aux:= aux - 1; el:= list[aux];
    if el = open then
      begin for el:= list[aux] while el ≠ close do
          aux:= aux - 1; goto next
      end optional part
    else if el = semicolon v el = point then
      begin numb of empties:= numb of empties + 1;
        if numb of empties > 1 then
          begin lll:= false; message1(\in \, notion, +
                          \ two alternatives yield empty\)
          end;
        if el = semicolon then goto next
      end alternative
    else if possibly empty[el] then goto next else
      for el:= list[aux] while el ≠ point do
      if el = semicolon then goto next else
      aux:= aux - 1
    end empty notion
  else
    for el:= list[aux] while el ≠ point do aux:= aux-1
  end grammar
end check only one alternative yields empty;
procedure may begin with;

comment this procedure determines the relation "may begin with".
the result is stored in 'first' (directly beginning with),
its transitive closure in 'first star'.
while determining 'first', error messages are given for
the "direct initial uncertainties" found;

begin integer i, j, el, aux, notion;
boolean optional, errors, any;
errors:= false;
for i:= 1 step 1 until number do
for j:= 1 step 1 until number do
first[i, j]:= first star[i, j]:= false;
for aux:= upperbound, aux - 1 while aux > top do
begin notion:= list[aux]; aux:= aux - 1; optional:= false;
for el:= list[aux] while el ≠ point do
begin aux:= aux - 1;
if el = open then optional:= true else
if el = close then optional:= false else
if el ≠ semicolon then
begin if first[notion, el] then
begin errors:= true;
message2("two alternatives in ", notion,
+ start with ", el)
end direct uncertainty
else first[notion, el]:= first star[notion, el]:= true;
if possibly empty[el] then else
if optional then
begin for el:= list[aux] while el ≠ close
do aux:= aux - 1
end skip rest of option
else
for el:= list[aux] while el ≠ semicolon
el ≠ point do aux:= aux - 1
end
end production rule
end grammar;
if errors then l1l:= false else
printtext("no direct initial uncertainties found"); carriage(5);
transitive closure(first star);
any:= false;
for i:= 1 step 1 until number do
if first star[i, i] then
begin if ¬ any then
begin printtext("the following rules are left-recursive");
nlcr; any:= true
end;
nlcr; print notion(i)
end;
if ¬ any then printtext("no rule is left-recursive");
carriage(5);
end may begin with;
procedure may end with;
comment this procedure determines the relation "may end with";
begin integer i, j, n, p, aux, notion, el;
  boolean optional;
  for i := 1 step 1 until number do
  for j := 1 step 1 until number do
    last[i, j] := last star[i, j] := false;
  for aux := upperbound, aux while aux > top do
    begin notion := list[aux]; n := aux; optional := false;
      for el := list[aux] while el ≠ point, el do
        aux := aux - 1; p := aux + 2;
      for el := list[p] while p < n do
        begin p := p + 1;
          if el = open then optional := false else
          if el = close then optional := true else
          if el ≠ semicolon then
            begin last[notion, el] := last star[notion, el] :=
              true;
              if possibly empty[notion] then else
              if optional then
                begin for el := list[p] while el ≠ open do
                  p := p + 1
                end skip rest of option
              else
                for el := list[p] while el ≠ semicolon ^ p < n
                do p := p + 1
            end
        end
    end production rule
end grammar;
transitive closure(last star)
end may end with;

procedure transitive closure(r); boolean array r;
comment warshall algorithm, see "grammar handling tools", p48;
begin integer i, j, k;
  for i := 1 step 1 until number do
  for j := symbols + 1 step 1 until number do
    if r[j, i] then
      begin for k := 1 step 1 until number do
        if r[i, k] then r[j, k] := true
      end
end transitive closure;
procedure follow within;
comment this procedure determines the successions of notions within the production rules;
begin integer i, j, stackpointer, lwb stack, upb stack, el, previous, aux;
   integer array stack[1: 100];
comment the upperbound 100 may be erroneous, a good upperbound is the maximum number of notions in an alternative, but this will very likely be less than 100;
for i:= 1 step 1 until number do
for j:= 1 step 1 until number do follow[i, j]:= false;
for aux:= upperbound, aux - 1 while aux > top do
for el:= semicolon, list[aux] while el # point do
begin aux:= aux - 1;
   if el = open then
      begin if previous # nil then
         begin stackpointer:= stackpointer + 1;
            stack[stackpointer]:= previous
         end;
            upb stack:= stackpointer + 1
      end
   else if el = close then lwb stack:= upb stack:= 1
   else if el = semicolon then
      begin stackpointer:= 0; lwb stack:= upb stack:= 1;
         previous:= nil
      end
   end
begin for i:= lwb stack step 1 until stackpointer do follow[stack[i], el]:= true;
   if previous # nil then
      begin follow[previous, el]:= true;
         if possibly empty[el] then
            begin stackpointer:= stackpointer + 1;
               stack[stackpointer]:= previous
            end
      else
            begin stackpointer:= upb stack - 1;
               lwb stack:= upb stack
            end
      end;
   previous:= el
end notion
end rule and grammar
end follow within;
comment check for ll(1)-ness;

procedure report indirect initial uncertainties;
comment this procedure determines whether two alternatives of
one rule start with the same terminal symbol;
begin integer n1;
  boolean any, left;

procedure report(n1, n2); value n1, n2; integer n1, n2;
begin if "any then
  begin printtext("for the following notions, more than ");
    printtext("one alternative may"); nlc; space(5);
    printtext("start with a given notion:"); nlc;
    any:= true; lli:= false
  end;
  nlc; if left then
  begin nlc; print notion(n1); prsym(minus char); nlc;
    left:= false
  end;
  space(8); print notion(n2)
end report;

any:= false;
for n1:= symbols + 1 step 1 until number do
begin integer count, n2, i, si, j, sj;
  integer array direct start[1 : number];
  comment collect "direct start"s of n1;:
  left:= true; count:= 0;
  for n2:= 1 step 1 until number do
  if first[n1, n2] then
  begin count:= count + 1; direct start[count]:= n2 end;
  for i:= 1 step 1 until count do
  begin si:= direct start[i];
    for j:= i + 1 step 1 until count do
      begin sj:= direct start[j];
        if si < symbols then
        begin if sj < symbols then else
          if first star[sj, si] then report(n1, si)
        end si is basic, sj may start with si
        else
          for n2:= 1 step 1 until symbols do
          if first star[sj, n2]" first star[sj, n2] then
          report(n1, n2)
      end sj
    end si
end rules;
if "any then
printtext("no indirect initial uncertainties found");
carriage(5)
end report indirect initial uncertainties;
procedure report indirect uncertainties;
comment this procedure detects violations of requirement 3;
begin integer aux, el, handle;
 boolean optional, any;

procedure check follow(start, in first option);
 value start, in first option; integer start;
 boolean in first option;
 comment checks whether the notion "i" at "start" (which is
 possibly empty or the beginning of an optional part)
 may be followed (in the rule for "handle") by a notion
 which starts with a terminal symbol that may also be
 the beginning of "i";
begin integer aux, el;
 boolean test completed, in next option;
 aux := start - 1; start := list[start];
in next option := test completed := false;
for el := list[aux] while ¬ test completed do
 begin if el = open then in next option := true else
 if el = close then
 in next option := in first option := false else
 if el = semicolon v el = point then
 begin check end(start); test completed := true end
 else
 begin if possibly empty[start] v ¬ in first option
 then
 begin integer i;
 for i := 1 step 1 until symbols do
 if first star[start, i] × first star[el, i]
 then
 begin any := true; message6(in 4, handle,
, the possibly empty or optional notion 4,
start, 4 may be followed by 4, el, 4; 4,
4 both 4, start, 4 and 4, el,
4 may begin with 4, i); lli := false
end message
end compare the beginning of start and el;
if possibly empty[el] then else
if in first option v in next option then
begin for el := list[aux - 1] while el #
close do aux := aux - 1
end skip rest of option
else test completed := true
end notion;
 aux := aux - 1
end test
end check follow;
procedure check end(start); value start; integer start;
comment similar to "check follow", but now "\^" is the
begin integer i, j;
for i:= symbols + 1 step 1 until number do
if last start[i, handle] then
begin comment i may end with handle;
for j:= 1 step 1 until number do
if follow[i, j] then
begin comment j may follow i;
integer k;
for k:= 1 step 1 until symbols do
if first star[start, k] \^ first star[j, k] then
begin message(\^the notion \^, start,
\^ is the beginning of the possibly empty\^, 
\^or optional last member of \^, i, \^via \^, 
handle, \^\^, \^and may be followed by \^, j, \^;\^, 
\^both \^, start, \^ and \^, j, \^ may begin with \^, 
k); any:= lll:= false
end message
end compare the beginning of start and j
end
end check end;

procedure check against ambiguous option(start);
value start; integer start;
comment checks whether an option starting at "start" does not 
contain only empty-producing notions, i.e. is 
ambiguous;
begin integer aux;
boolean ok;
for aux:= start - 1, aux - 1 while ok do
begin el:= list[aux]; if el = close then
begin ok:= lll:= false; any:= true; message(\^in \^, 
handle, \^ the optional part starting with \^, 
start - 1, \^ produces empty in more than one way\^)
end
else ok:= possibly empty[el]
end option
end check against ambiguous option;

for el:= 1 step 1 until number do
first start[el, el]:= last start[el, el]:= true;
printtext(\^violations of requirement 3\^); nlc;
printtext(\^i.e., ambiguities arising from empty notions or \^);
printtext(\^optional parts\^); nlc; nlc; any:= false;
for aux:= upperbound, aux - 1 while aux > top do
begin handle:= list[aux]; aux:= aux - 1; optional:= false;
for e1:= list[aux] while e1 ≠ point do
    begin if e1 = open then
        begin optional:= true;
            check against ambiguous option(aux)
        end
    else
        if e1 = close then optional:= false else
            if e1 ≠ semicolon then
                begin if possibly empty[e1] then
                    check follow(aux, optional) else
                        if optional then
                            begin check follow(aux, true); optional:=false
                            end
                        first [not empty] notion of optional part
                end
        aux:= aux - 1
    end rule
    end grammar;
    if ¬ any then printtext("none")
end report indirect uncertainties;

comm

procedure message1(s1, n1, s2); value n1; integer n1;
    string s1, s2;
begin n1cr; printtext(s1); print notion(n1);
    printtext(s2); n1cr
end message1;

procedure message2(s1, n1, s2, n2); value n1, n2;
    integer n1, n2; string s1, s2;
begin n1cr; printtext(s1); print notion(n1);
    printtext(s2); print notion(n2); n1cr
end message2;

procedure message3(s1, n1, s2, n2, s3); value n1, n2;
    integer n1, n2; string s1, s2, s3;
begin n1cr; printtext(s1); print notion(n1); printtext(s2);
    print notion(n2); printtext(s3); n1cr
end message3;

procedure message6(s1, n1, s2, n2, s3, n3, s4, s5, n4, s6, n5, s7,
    n6);
    value n1, n2, n3, n4, n5, n6; integer n1, n2, n3, n4, n5, n6;
    string s1, s2, s3, s4, s5, s6, s7;
begin n1cr; n1cr; printtext(s1); print notion(n1);
    printtext(s2); print notion(n2); n1cr; space(5); printtext(s3);
    print notion(n3); printtext(s4); n1cr; space(5); printtext(s5);
    print notion(n4); printtext(s6); print notion(n5);
    printtext(s7); print notion(n6)
end message6;
procedure message7(s1, n1, s2, s3, n2, s4, n3, s5, s6, n4, s7, s8, n5, s9, n6, s10, n7);
value n1, n2, n3, n4, n5, n6, n7; integer n1, n2, n3, n4, n5, n6, n7; string s1, s2, s3, s4, s5, s6, s7, s8, s9, s10;
begin nlc; nlc; printtext(s1); print notion(n1); printtext(s2);
  nlc; space(5); printtext(s3); print notion(n2);
  nlc; space(5); printtext(s4); print notion(n3); printtext(s5);
  nlc; space(5); printtext(s6); print notion(n4); printtext(s7);
  nlc; space(5); printtext(s8); print notion(n5); printtext(s9);
  print notion(n6); printtext(s10); print notion(n7)
end message7;

procedure print list of notions;
begin integer i;
  printtext(4list of notions, each notion preceded by its number4);
  nlc; printtext(4and, if not defined by an asterisk4); nlc;
  for i:= 1 step 1 until number do
    begin nlc; if ~ defined(i) then
      begin prsym(star); define(i); error count:= error count+1
      end
    else prsym(spacechar);
      absfixt(4, 0, i); print notion(i)
    end
end print list of notions;

procedure print notion(number); value number; integer number;
begin integer i, n;
  integer array a[1 : 100];
  comment this upperbound is rather arbitrary, notions of
  length > 100 are considered rare;
  get notion at(notion link[number]) of length:(n) in:(a);
  for i:= 1 step 1 until n do prsym(a[i])
end print notion;

procedure get notion at(index) of length:(n) in:(a);
value index, integer index, n; integer array a;
begin integer el, j, k;
  n:= 0;
  for el:= list[index] while el > 0, el + max int do
    begin for k:= 0, k + 1 while k < 5 ^ el > 0 do
      begin j:= el div t32[k]; el:= el - j * t32[k]; n:= n +1;
        a[n]:= if j = space repr then space char else
          j + letter a - 1
      end;
      index:= index + 1
    end
end get notion at;
main program:
  grammar;
  newpage;
  print list of notions;
  if error count > 0 then
    begin carriage(3); absfixt(3, 0, error count);
    printtext(4 errors in grammar); exit
  end;
  newpage;
  l11 := true;
  check for empty productions;
  may begin with;
  report indirect initial uncertainties;
  may end with;
  follow within;
  report indirect uncertainties;
  carriage(5); printtext(4 the grammar is 4);
  if not l11 then printtext(4 of type l1(1) 4)
end
end
3.2 Two examples

UPPERBOUND ARRAY LIST: 20452

[INPUT:]

[TERMINAL SYMBOLS]
COLON; POINT; SEMICOLON; COMMA; OPEN; CLOSE; NOTION; EOF.

GRAMMAR: TERMINALS, RULES, EOF.

TERMINALS: NOTION, POINT; NOTION, SEMICOLON, TERMINALS.

RULES: RULE; RULE, RULES,
RULE: LEFT HAND SIDE, COLON, ALTERNATIVES, POINT,
LEFT HAND SIDE: NOTION,
ALTERNATIVES: ALTERNATIVE OPTION;
ALTERNATIVE OPTION, SEMICOLON, ALTERNATIVES.
ALTERNATIVE OPTION: ALTERNATIVE;
ALTERNATIVE: MEMBER; MEMBER, COMMA, ALTERNATIVE.
MEMBER: NOTION; OPEN, NOTION LIST, CLOSE,
NOTION LIST: NOTION; NOTION, COMMA, NOTION LIST.
THE FOLLOWING NOTIONS MAY PRODUCE EMPTY:

ALTERNATIVES
ALTERNATIVE OPTION

TWO ALTERNATIVES IN TERMINALS START WITH NOTION
TWO ALTERNATIVES IN RULES START WITH RULE
TWO ALTERNATIVES IN ALTERNATIVES START WITH ALTERNATIVE OPTION
TWO ALTERNATIVES IN ALTERNATIVE START WITH MEMBER
TWO ALTERNATIVES IN NOTION LIST START WITH NOTION

NO RULE IS LEFT-RECURSIVE

NO INDIRECT INITIAL UNCERTAINTIES FOUND

VIOLATIONS OF REQUIREMENT 3
(I.E., AMBIGUITIES ARISING FROM EMPTY NOTIONS OR OPTIONAL PARTS):
NONE

THE GRAMMAR IS NOT OF TYPE LL(1)
[INPUT]

[Terminal Symbols]
Colon; Point; Semicolon; Comma; Open; Close; Notion; EOF.

Grammar: Terminals, Rules, EOF.

Terminals: Notion, Rest Terminals.
Rest Terminals: Point; Semicolon, Terminals.

Rules: Rule, (Rules).
Rule: Left Hand Side, Colon, Alternatives, Point.
Left Hand Side: Notion.
Alternatives: (Alternative), (Semicolon, Alternatives).
Alternative: Member, (Comma, Alternative).
Member: Notion; Open, Notion List, Close.
Notion List: Notion, (Comma, Notion List).

Upperbound Array List: 20452
THE FOLLOWING NOTIONS MAY PRODUCE EMPTY ALTERNATIVES

NO DIRECT INITIAL UNCERTAINTIES FOUND

NO RULE IS LEFT-RECURSIVE

NO INDIRECT INITIAL UNCERTAINTIES FOUND

VIOLATIONS OF REQUIREMENT 3 (I.E., AMBIGUITIES ARISING FROM EMPTY NOTIONS OR OPTIONAL PARTS):

NONE

THE GRAMMAR IS OF TYPE LL(1)
References


