Two-Level Grammars

Augment a context free grammar with a higher-level rules (rules about rules) to allow the testing of context sensitive properties.

A BNF grammar involves a finite number of terminal symbols and a finite number of production rules (a one-level grammar)

A two-level grammar has rules that can produce an infinite number of production rules.

Begin with the part of a two-level grammar that is equivalent to BNF.

A **protonotion** is a sequence of lower case, boldface characters that define a nonterminal.

If followed by the protonotion **symbol,** then corresponds to a terminal in the target language as defined in a representation table.

Examples

block stands for nonterminal <block>.

begin symbol stands for keyword **begin**.

Representation table shows correspondence between protonotions and terminal symbols.

A **hyper-rule** is a production rule that may contain protonotions and metanotions, each of which stands for a set of protonotions.

- Colon separates the left-hand and right-hand side of a rule.
- Comma indicates the juxtaposition of protonotions,
- Semicolon indicates alternatives for the right-hand side of a rule.
- Period terminates a hyper-rule.
- Substitutions of protonotions for metanotions must be consistent.
- If no metatnotions, then equivalent to BNF productions.

A Grammar for Wren

program:
program symbol, identifier, is symbol, block.

block: declaration seq, begin symbol,
command seq, end symbol.

declaration seq:
empty;
declaration, declaration seq.

empty: .

declaration: var symbol, variable list, colon
symbol, type, semicolon symbol.

type: integer symbol;
boolean symbol.

variable list:
variable;
variable, comma symbol, variable list.

command seq:
command;
command, semicolon symbol,
command seq.

command:
variable, assign symbol, expression;
read symbol, variable;
write symbol, integer expr;
skip symbol;
while symbol, boolean expr, do symbol,
command seq, end while symbol;
if symbol, boolean expr, then symbol,
command seq, end if symbol;
if symbol, boolean expr, then symbol,
command seq, else symbol,
command seq, end if symbol.

expression: integer expr;
boolean expr.

integer expr: term;
integer expr, weak op, term.

term: element;
term, strong op, element.

element: numeral;
variable;
left paren symbol, integer expr,
right paren symbol;
negation symbol, element.

boolean expr:
boolean term;
boolean expr, or symbol, boolean term.

boolean term:
boolean element;
boolean term, and symbol, boolean element.

boolean element:
true symbol;
false symbol;
variable;
comparison;
not symbol, left paren symbol, boolean expr, right paren symbol.

comparison: integer expr, relation, integer expr.

variable: identifier.

relation: less or equal symbol;
less symbol;
equal symbol;
greater symbol;
greater or equal symbol;
not equal symbol.

weak op: plus symbol;
minus symbol.

strong op: multiply symbol;
divide symbol.

identifier: letter;
letter, identifier;
letter, digit.

letter: a symbol; b symbol; c symbol;
d symbol; e symbol; f symbol;
g symbol; h symbol; i symbol;
j symbol; k symbol; l symbol;
m symbol; n symbol; o symbol;
p symbol; q symbol; r symbol;
s symbol; t symbol; u symbol;
v symbol; w symbol; x symbol ;
y symbol; z symbol.

numeral: digit;
digit, numeral.

digit: zero symbol; one symbol; two symbol;
three symbol; four symbol; five symbol;
six symbol; seven symbol;
eight symbol; nine symbol.

The Second Level

A **metanotion**, denoted by a string of upper case characters, represents a particular collection of protonotions.

A **metarule** is a production rule that defines single metanotion, given on left-hand side, in terms of a sequence of protonotions and metanotions on right-hand side.

- Double colon (**::**) separates the left- and right-hand sides.
- Space indicates the juxtaposition of items.
- Semicolon indicates alternatives for the right hand side.
- Period terminates a metarule.
- Metanotions may appear anywhere in a hyper-rule.

Examples

ALPHA :: a; b; c; d; e; f; g; h; i;
j; k; l; m; n; o; p; q; r;
s; t; u; v; w; x; y; z.

Metanotion ALPHA stands for twenty six different protonotions.

NOTION :: ALPHA; NOTION ALPHA.

A **NOTION** represents any sequence of bold, lowercase letters.

letter: ALPHA symbol.

This hyper-rule stands for 26 productions.

double: ALPHA symbol, ALPHA symbol.

Also stands for 26 productions.

List Structures

Sequence of one or more items separated by commas.

Introduce the metanotion **NOTION** to provide a template for list construction:

NOTION list:
NOTION;
NOTION, comma symbol,
NOTION list.

Possibilities for **NOTION**

NOTION ::
identifier; **digit**; **letter**; **numeral**;

Have specified lists of identifiers, lists of digits, lists of letters, and lists of numerals.

Require *consistent substitution* of protonotions for metanotions within a hyper-rule.

Notational Conventions

Metanotion **EMPTY**stands for the empty protonotion.

EMPTY :: .

Define the concept of number using a tally notation.

Protonotion **iiiiiiii** represents the number 8.

Specify a tally by the metarule:

TALLY :: i ; TALLY i.

Use the **EMPTY** protonotion for zero:

TALLETY :: EMPTY; TALLY.

The suffix **-ETY** implies the **EMPTY** protonotion as one of the alternatives.

Variation on Consistent Substitution

Relax (at least notationally) the consistent substitution principle.

Initially defined strings of double letters.

double: ALPHA symbol, ALPHA symbol.

Defines the strings: aa, bb, cc, dd, ….

Now mix letters in a single string.

mixed letters: ALPHA symbol,
ALPHA1 symbol.

ALPHA1 :: ALPHA. (assumed)

Defines the strings: aa, ab, ba, bb, ac, ca, …

A metanotion ending in a digit stands for the same set of protonotions as the metanotion without the digit.

Counting Items

Specify a sequence of items of length zero upto some fixed value.

upto TALLY i NOTION:
upto TALLY NOTION;
NOTION, upto TALLY NOTION.

upto NOTION: EMPTY.

Specity a sequence of zero to five digits:

upto iiiii digit

Derivation Tree

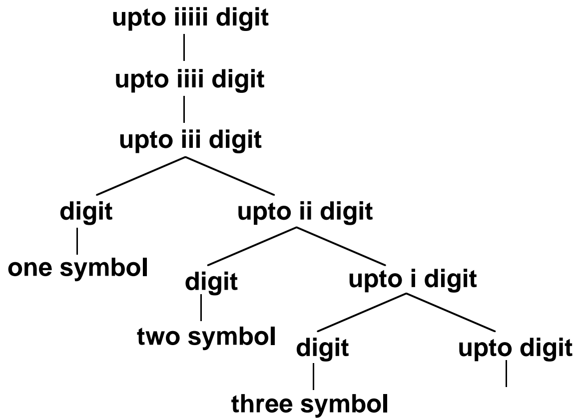
Nodes in the tree are label with protonotions.

Protonotions for all leaf nodes end with **symbol** or are **EMPTY**.

Traversing the leaf nodes from left to right, replacing the protonotion symbols with the corresponding tokens in the Representation Table, produces the string being parsed.

Empty leaves just disappear.

Derivation Tree for 123



Fortran String Literals

Older versions of Fortran defined character string literals using the notation

<string literal> ::= <numeral> **H** <string>

where the <numeral> is a positive base-ten integer, H is a keyword, and <string> is a sequence of characters.

Constraint

The numeric value of the base-ten numeral matches the length of the string.

Two-level Grammar

Initially, assume that numeral is single digit from 1 to 9.

Hyper-rules for the digit symbols:

i digit: digit one symbol.
ii digit: digit two symbol.
iii digit: digit three symbol.
iiii digit: digit four symbol.
iiiii digit: digit five symbol.
iiiiii digit: digit six symbol.
iiiiiii digit: digit seven symbol.
iiiiiiii digit: digit eight symbol.
iiiiiiiii digit: digit nine symbol.

A string literal is a sequence of lower case letters:

APLHA ::
a; b; c; d; e; f; g; h; i; j; k; l; m;
n; o; p; q; r; s; t; u; v; w; x; y; z.

LETTER :: letter ALPHA.

LETTERSEQ ::
LETTER;
LETTERSEQ LETTER.

A string literal, called **hollerith**, is specified by the following hyper-rule.

hollerith: TALLY digit,
hollerith symbol,
TALLY LETTERSEQ.

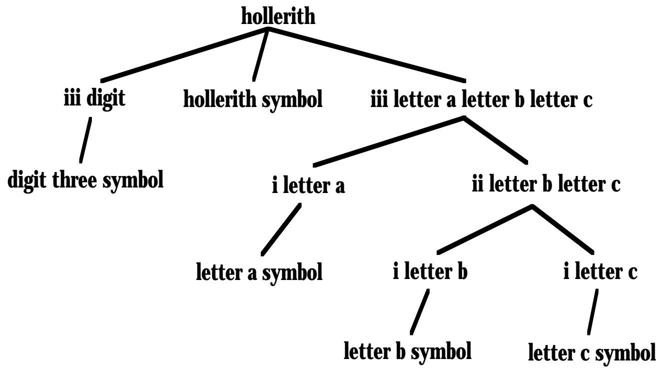
Every time we remove **i** from **TALLY,** we also remove a **LETTER** from **LETTERSEQ**.

Must eventually reach a single **i** followed by a single **LETTER**.

TALLY i LETTER LETTERSEQ:
i LETTER, TALLY LETTERSEQ.

i LETTER: LETTER symbol.

Derivation Tree for 3Habc



Multidigit Numerals

If the leaves of the subtree for a numeral are **digit two symbol** followed by **digit five symbol**, the **TALLY** will be **iiiiiiiiiiiiiiiiiiiiiiiii**.

Allow for a zero digit at any position, other than the leading digi.

EMPTY digit: digit zero symbol.

TALLETY :: TALLY; EMPTY.

Develop a hyper-rule for **TALLY constant**.

For a multiple digit numeral $d_1d_2 \ldots d_{n-1}d_n$:

$tally(d_1d_2 \ldots d_{n-1}d_n)$ is
ten copies of $tally(d_1d_2 \ldots d_{n-1})$
followed by $tally(d_n)$.

where Clauses (predicates)

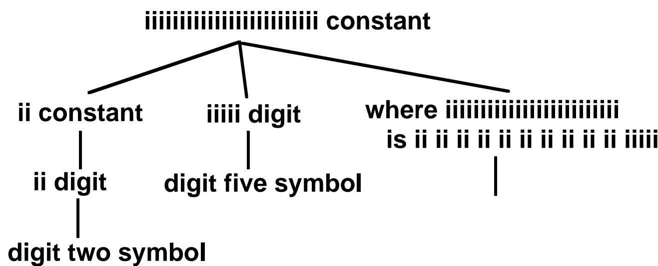
A mechanism for expressing equality of protonotions.

```
TALLETY constant:
   TALLETY digit;
   TALLETY1 constant,
                TALLETY2 digit,
      where TALLETY is
         TALLETY1 TALLETY1 TALLETY1
         TALLETY1 TALLETY1 TALLETY1
         TALLETY1 TALLETY1 TALLETY1
         TALLETY1 TALLETY2.
```


where TALLETY is TALLETY: EMPTY.

If condition holds, leaf on the tree is **EMPTY** protonotion; otherwise, the parse fails.

Derivation Tree for Numeral 25



Now Use

```
hollerith: TALLY constant,
             hollerith symbol,
                TALLY LETTERSEQ.
```


A Two-Level Grammar for Wren

Two-level grammar performs necessary context checking.

- Identifiers are not multiply declared.
- Variables have been declared, and that their usage is consistent with their types.

A program is syntactically correct if we can build a complete two-level grammar tree for it.

Declaration information is formed in a metanotion called DECLSEQ, which is associated with the context sensitive portions of commands.

Metanotions that serve as basic building blocks in Wren.

(m1) **ALPHA :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u; v; w; x; y; z.**

(m2) **NUM :: zero; one; two; three; four; five; six; seven; eight; nine.**

(m3) **ALPHANUM :: ALPHA; NUM.**

(m4) **LETTER :: letter ALPHA.**

(m5) **DIGIT :: digit NUM.**

(m6) **LETTERDIGIT :: LETTER; DIGIT.**

A **NAME** starts with a letter followed by any sequence of letters or digits.

(m7) **NAME ::**
LETTER; NAME LETTERDIGIT.

A **NUMERAL** is a sequence of digits.

(m8) **NUMERAL ::**
DIGIT; NUMERAL DIGIT.

Declaration / Declaration Sequence

A declaration associates a name with a type.

Consider:

var s1 : **integer**;
var d2 : **boolean**;

Represented in two-level grammar as:

letter s digit 1 type integer
letter d digit 2 type boolean

Metarules

(m9) **DECL :: NAME type TYPE.**

(m10) **TYPE ::**
integer; boolean; program.

(m11) **DECLSEQ :: DECL; DECLSEQ DECL.**

(m12) **DECLSEQETY :: DECLSEQ; EMPTY.**

(m13) **EMPTY :: .**

Note that a valid Wren program may have no declarations.

These metanotions are sufficient to begin construction of the declaration information for a Wren program.

Declarations

var x**: integer;**
var y**: integer;**
var z **: integer;**

Should produce the following **DECLSEQ**:

letter x type integer
letter y type integer
letter z type integer

A **DECLSEQ** permits three alternatives:

- A sequence followed by a single declaration

(h3) **DECLSEQ DECL declaration seq:**
DECLSEQ declaration seq,
DECL declaration.

- A single declaration

(h4) **DECLSEQ declaration seq:**
DECLSEQ declaration.

- An empty declaration

(h5) **EMPTY declaration seq: EMPTY.**

Single Declaration

(h6) **NAME type TYPE declaration:**
var symbol, NAME symbol,
colon symbol, TYPE symbol,
semicolon symbol.

Specification of **NAME symbol** restricts identifier names to single characters for now.

Declaration with a Variable List

The same declaration sequence should be produced by the single declaration:

var x**,** y**,** z **: integer;**

- First variable in the list is preceded by **var symbol** and followed by **comma symbol**.

(h9) **NAME1 type TYPE NAME2**
type TYPE var list :
var symbol, NAME1 symbol,
comma symbol.

- Last variable in the list is followed by **colon symbol**, the type, and **semicolon symbol**.

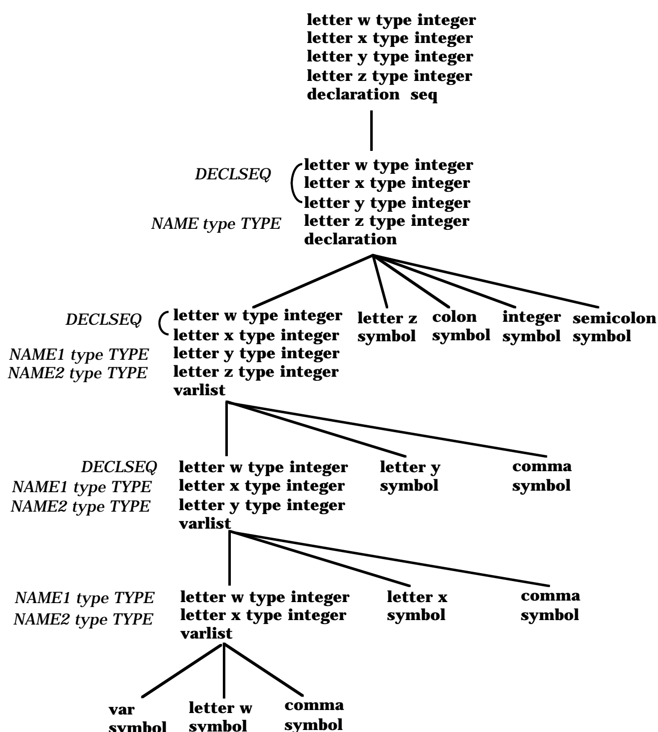
(h7) **DECLSEQ NAME type TYPE declaration:**
DECLSEQ NAME type TYPE var list,
NAME symbol, colon symbol,
TYPE symbol, semicolon symbol.

- "In between" variables are followed by **comma symbol**.

(h8) **DECLSEQ NAME1 type TYPE NAME2**
type TYPE var list :
DECLSEQ NAME1 type TYPE var list,
NAME1 symbol, comma symbol.

The general strategy is to have the type information passed from right to left.

var w, x, y, z : **integer**;



Program and Block

(h1) **program: program symbol,**
NAME symbol, is symbol,
block with NAME type
program DECLSEQETY,
where
NAME type program
DECLSEQETY unique.

(h2) **block with NAME type program**
DECLSEQETY:
DECLSEQETY declaration seq,
begin symbol,
NAME type program DECLSEQETY
command seq, end symbol.

Program identifier name is added to the declaration sequence with type **program**.

Information is passed to the command sequence.

The **where** rule checks for the uniqueness of declarations.

All leaf nodes will be **EMPTY** if the identifiers are unique using the following cases.

A single declaration is unique.

(h22) **where DECL unique: EMPTY.**

Multiple declarations: separate the last declaration in the list and insure the name is not contained in any declarations to the left.

(h23) **where DECLSEQ NAME type TYPE unique:**
where DECLSEQ unique,
where NAME not in DECLSEQ.

To insure a name is not contained in a declaration sequence.

(h24) **where NAME not in DECLSEQ DECL:**
where NAME not in DECLSEQ,
where NAME not in DECL.

Check that two names are not the same.

(h25) **where NAME1 not in NAME2 type TYPE:**
where NAME1 is not NAME2.

Names are separated into the a sequence of characters (possibly empty) followed by a final character.

Use **NOTION** and **NOTETY**.

(m14) **NOTION :: ALPHA; NOTION ALPHA.**

(m15) **NOTETY :: NOTION; EMPTY.**

The identifiers contain either alphabetic characters or digit characters.

Characters of different kind are not equal.

If characters are of the same kind but not the same, then one character appears before the other character in appropriate character set.

Test is applied to all characters in sequence until a mismatch is found.

(h26) **where NOTETY1 NOTION1 ALPHANUM1**
is not NOTETY2 NOTION2 ALPHANUM2:
where NOTETY1 is not NOTETY2;
where NOTION1 different
kind NOTION2;
where ALPHANUM1
precedes ALPHANUM2
in abcdefghijklmnopqrstuvwxyz;
where ALPHANUM2
precedes ALPHANUM1
in abcdefghijklmnopqrstuvwxyz;
where ALPHANUM1
precedes ALPHANUM2
in zero one two three four
five six seven eight nine;
where ALPHANUM2
precedes ALPHANUM1
in zero one two three four
five six seven eight nine.

The phrase **NOTETY1 NOTION1 ALPHANUM1** refers to an identifier the form:

NOTETY1	NOTION1	ALPHANUM1
letter s letter u	letter	m

The first clause in hyper-rule H26 acts as the recursive case; the others are the basis.

A **letter** is always different from a **digit**.

(h27) **where letter different kind digit: EMPTY.**

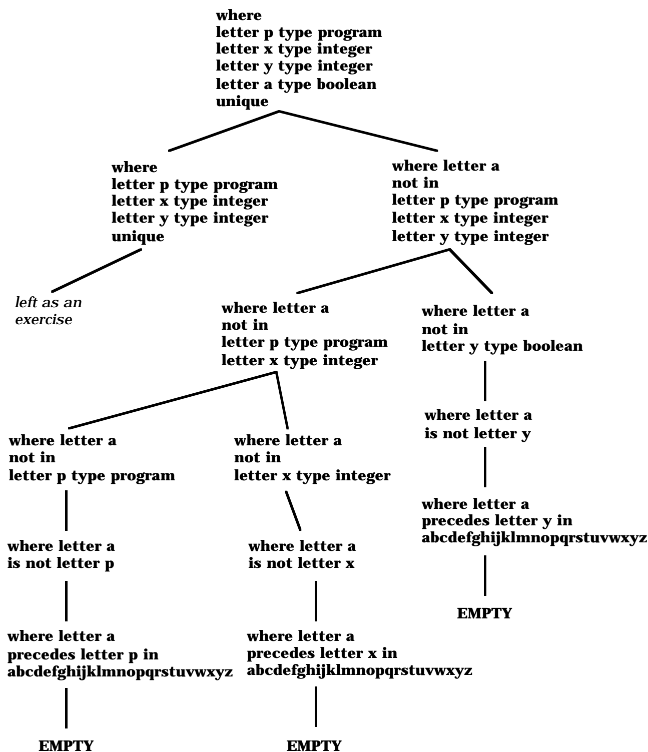
(h28) **where digit different kind letter: EMPTY.**

Check whether a character precedes another or if a digit precedes another.

(h29) **where ALPHA1 precedes ALPHA2**
in NOTETY1 ALPHA1 NOTETY2
ALPHA2 NOTETY3 : EMPTY.

(h30) **where NUM1 precedes NUM2**
in NOTETY1 NUM1 NOTETY2
NOTETY3 : EMPTY.

Checking Uniqueness of Identifiers



Commands and Expressions

Metanotions for operators:

(m16) **WEAKOP :: plus symbol; minus symbol.**

(m17) **STRONGOP :: multiply symbol;**
divide symbol.

(m18) **RELATION :: less or equal symbol;**
less symbol;
not equal symbol;
greater symbol;
greater or equal symbol;
equal symbol.

Declaration information is passed to each individual command.

No empty declaration sequences since every program must be named.

(h10) **DECLSEQ command seq:**
DECLSEQ command;

DECLSEQ command,
semicolon symbol,
DECLSEQ command seq.

Commands use the declaration information in different ways:

- **skip** uses no declaration information.
- **write**, **while**, and **if** commands pass the declaration information to their constituent parts.
- assignment requires that the target variable have the same type as the expression.
- **read** uses the declaration information to check that associated variable is of type integer.

Hyper-rule (H11) separates these cases.

(h11) **DECLSEQ command:**

TYPE NAME in DECLSEQ,
assign symbol,
TYPE expression in DECLSEQ;

skip symbol;

read symbol, integer NAME in DECLSEQ;

write symbol,
integer expression in DECLSEQ;

while symbol,
boolean expression in DECLSEQ,
do symbol, DECLSEQ command seq,
end while symbol;

if symbol,
boolean expression in DECLSEQ,
then symbol,
DECLSEQ command seq,
end if symbol;

if symbol,
boolean expression in DECLSEQ,
then symbol,
DECLSEQ command seq, else symbol,
DECLSEQ command seq, end if symbol.

The **read** command illustrates hyper-rules of the form **TYPE NAME in DECLSEQ** that

- produce the appropriate **NAME** symbol.
- check that the **NAME** appears in the **DECLSEQ** with the appropriate **TYPE**.

(h19) **TYPE NAME in DECLSEQ :**
NAME symbol, where NAME type
TYPE found in DECLSEQ.

The **DECLSEQ** is checked one declaration at a time from left to right.

The **EMPTY** notion is produced if appropriate declaration is found; otherwise, the parse fails.

(h20) **where NAME type TYPE found in**
NAME type TYPE DECLSEQETY:
EMPTY.

(h21) **where NAME1 type TYPE1 found in**
NAME2 type TYPE2 DECLSEQETY:
where NAME1 type TYPE1
found in DECLSEQETY.

Integer Expressions

(h12) **integer expression in DECLSEQ:**
term in DECLSEQ;
integer expression in DECLSEQ,
WEAKOP, term in DECLSEQ.

(h13) **term in DECLSEQ:**
element in DECLSEQ;
term in DECLSEQ, STRONGOP,
element in DECLSEQ.

(h14) **element in DECLSEQ:**
NUMERAL symbol;
integer NAME in DECLSEQ;
left paren symbol,
integer expression in DECLSEQ,
right paren symbol;
negation symbol,
element in DECLSEQ.

(h15) **boolean expression in DECLSEQ : …**

(h16) **boolean term in DECLSEQ : …**

(h17) **boolean element in DECLSEQ : …**

(h18) **comparison in DECLSEQ:**
integer expression in DECLSEQ,
RELATION,
integer expression in DECLSEQ.

TWO-LEVEL GRAMMARS AND PROLOG

Consistent substitution of protonotions for a metanotion within a hyper-rule is similar to binding of identifiers within a Prolog clause.

Hollerith String Literal Grammar

Top-level predicate:

hollerith(<list of digits>, hollerith, <list of letters>).

The program should print either "valid Hollerith string" or "invalid Hollerith string", as appropriate for the data.

We assume the list of digits and list of letters are syntactically correct.

Note the sample session in the text.

Allow for numerals of any size using **where** clauses in the grammar.

Two-level grammar for Hollerith string literals

hollerith: TALLY constant,
hollerithsymbol,
TALLY LETTERSEQ.

TALLY i LETTER LETTERSEQ:
i LETTER, TALLY LETTERSEQ.

i LETTER: LETTER symbol.

i digit: digit one symbol.
ii digit: digit two symbol.
iii digit: digit three symbol.
iiii digit: digit four symbol.
iiiii digit: digit five symbol.
iiiiii digit: digit six symbol.
iiiiiii digit: digit seven symbol.
iiiiiiii digit: digit eight symbol.
iiiiiiiii digit: digit nine symbol.

TALLETY constant:
TALLETY digit;
TALLETY1 constant,
TALLETY2 digit,
where TALLETY is
TALLETY1 TALLETY1 TALLETY1
TALLETY1 TALLETY1 TALLETY1
TALLETY1 TALLETY1 TALLETY1
TALLETY1 TALLETY2.

where TALLETY is TALLETY: EMPTY.

EMPTY digit: digit zero symbol.

TALLETY :: TALLY; EMPTY.

APLHA:: a; b; c; d; ... x; y; z.

LETTER:: letter ALPHA.

LETTERSEQ:: LETTER;
LETTERSEQ LETTER.

Clauses for single digits

```
digit([digitZeroSymbol], []).
digit([digitOneSymbol], [i]).
digit([digitTwoSymbol], [i,i]).
digit([digitThreeSymbol], [i,i,i]).
digit([digitFourSymbol], [i,i,i,i]).
digit([digitFiveSymbol], [i,i,i,i,i]).
digit([digitSixSymbol], [i,i,i,i,i,i]).
digit([digitSevenSymbol], [i,i,i,i,i,i,i]).
digit([digitEightSymbol], [i,i,i,i,i,i,i,i]).
digit([digitNineSymbol], [i,i,i,i,i,i,i,i,i]).
```

A Constant: Single digit or Sequence of digits.

```
constant(DIGIT, TALLETY) :-
        digit(DIGIT,TALLETY).

constant(DIGITS, TALLETY) :-
      splitDigits(DIGITS, LeadingDIGITS,
                                UnitDIGIT),
      constant(LeadingDIGITS, TALLETY1),
      digit(UnitDIGIT, TALLETY2),
      concatTenPlusDigit(TALLETY1,
                          TALLETY2, TALLETY).
```


Split Digits and Concatenate

```
splitDigits([D], [], [D]).

splitDigits([Head|Tail],[Head|Result],Unit) :-
                splitDigits(Tail, Result, Unit).

concatTenPlusDigit(TALLETY1,
                   TALLETY2, TALLETY) :-
    concat(TALLETY1, TALLETY1, TwoTimes),
    concat(TwoTimes, TwoTimes, FourTimes),
    concat(FourTimes, FourTimes, EightTimes),
    concat(EightTimes, TwoTimes, TenTimes),
    concat(TenTimes, TALLETY2, TALLETY).
```

The tally is generated from the number part and used to check the length of the letter sequence.

Each time a tally symbol is removed, a letter is removed.

Base cases check validity of hollerith string.

Hollerith

```
hollerith(Number,hollerith,Letters) :-
                constant(Number, TALLY),
                letterSeq(TALLY, Letters).

letterSeq([i],[Letter]) :-  alpha(Letter),
               write('Valid Hollerith string').

letterSeq([i|TALLETY],[Letter|Letters]) :-
                alpha(Letter),
                letterSeq(TALLETY, Letters).

letterSeq([],Letters) :-
          write('Invalid Hollerith string').

letterSeq(Number,[]) :-
          write('Invalid Hollerith string').

alpha(a).   alpha(b).   alpha(c).   alpha(d).
alpha(e).   alpha(f).   alpha(g).   alpha(h).
alpha(i).   alpha(j).   alpha(k).   alpha(l).
alpha(m).   alpha(n).   alpha(o).   alpha(p).
alpha(q).   alpha(r).   alpha(s).   alpha(t).
alpha(u     alpha(v).   alpha(w).   alpha(x).
alpha(y).   alpha(z).
```