ISSN 0105-8517

THE MEANING OF LOGICAL PROGRAMS

by

Brian H. Mayoh

DAIMI PB-126

September 1980

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

THE MEANING OF LOGICAL PROGRAMS

Brian H. Mayoh
Computer Science Department
Aarhus University, Aarhus
Denmark

Abstract

The semantics of a programming language are given by a function $\mathfrak{m}$ from Programs to Meanings. In this paper we bring some uniformity into the definition of logical programming languages like LUCID and PROLOG by specifying $\mathfrak{m}$ in Logic $\rightarrow$ (Control $\rightarrow$ Meanings). We describe how a context-free grammar can be assigned to each logical program and we identify Control with the language generated by the grammar. After this reduction there is no difference between the semantics of logical and conventional programming languages.

THE MEANING OF LOGICAL PROGRAMS

In a recent paper Kowalski (4) has advocated replacing the slogan "Algorithm = Program + Data Structure" by the slogan "Algorithm = Logic + Control". If the semantics of a language are to give us a function $\mathfrak{m}$ from Algorithms to Meanings, the second slogan suggests defining this function as a member of

$$\text{Logic} \rightarrow (\text{Control} \rightarrow \text{Meanings})$$

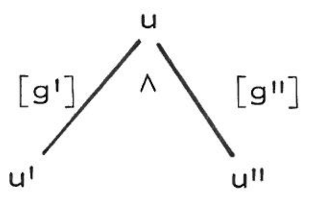
In this paper we use this factorization of $\mathfrak{m}$ to bring some uniformity in the definition of logical programming languages like LUCID and PROLOG. We describe how a context-free grammar can be assigned to each logical program and we identify Control with the language generated by the grammar. This reduces the problem of defining the function $\mathfrak{m}$ to the problem of defining the semantics of a traditional programming language, because the syntax of such a language is given by a grammar and the semantics gives a meaning to each of the "programs" generated by the grammar.

In the sections on AND/OR, PROLOG, LUCID and extended attribute grammar logical programming languages we show that our approach can give the "official" denotational semantics of these languages. In the last section we use data flow machines to give an operational semantics of a logical program and we prove the correctness of a data flow interpretation. The connection between data flow machines and logical programs should come as no surprise because both of them abandon assignment and Von Neumann stores.

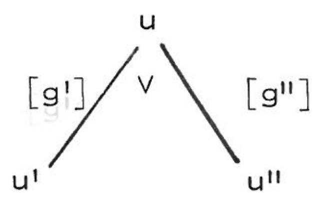
1. AND/OR logical programming

In (3) Harel introduced a precise logical programming language based on the AND/OR trees so common in work in artificial intelligence. There are three kinds of nodes in an AND/OR logical program:
(1) leaves,

(2) AND nodes



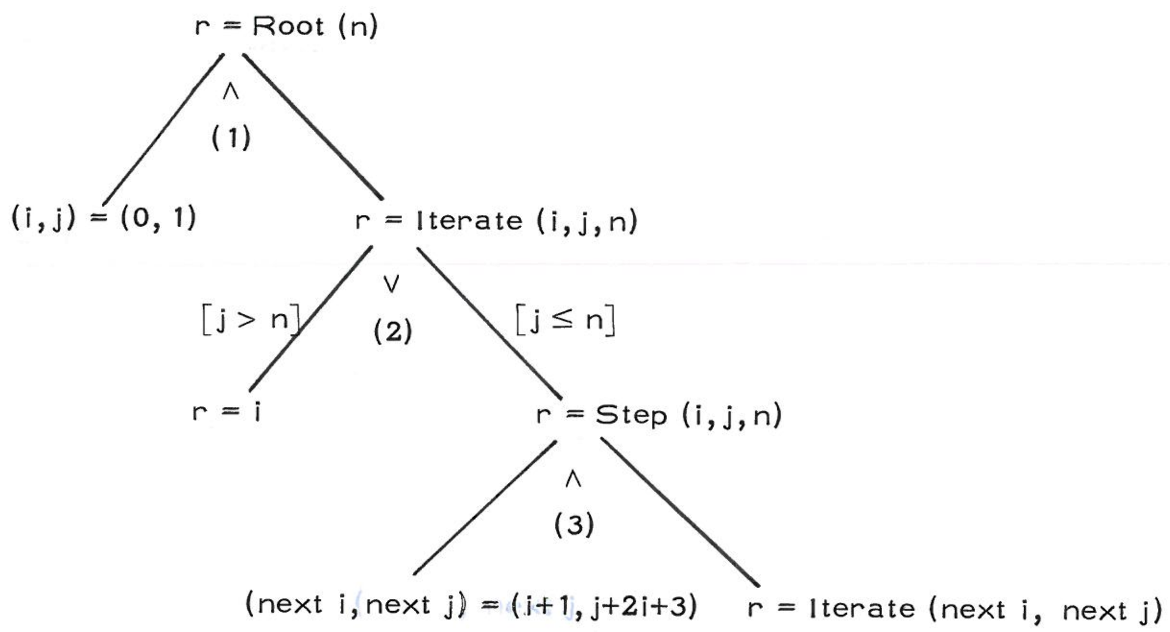
(3) OR nodes



where g' and g'' are conditional guards. Sequencing and concurrency are expressed by AND nodes; conditionals, non-determinism, repetition and recursion are expressed by OR nodes. Since AND/OR logical programs are designed for decomposing complicated problems into simpler problems, it is unfair of us to choose so simple an example as

$$\begin{aligned} \text{Root}(n) &= \text{Iterate}(0, 1, n) \\ \text{Iterate}(i, j, n) &= \underline{\text{if}}\ j > n\ \underline{\text{then}}\ i\ \underline{\text{else}}\ \text{Iterate}(i+1, j+2i+3, n) \end{aligned}$$

The AND/OR program for this way of finding the square root of a positive number n is



We notice (1) the variable lists for inputs and outputs, (2) the omission of guards that are always true, (3) all non-leaves are of the form $a = F(b)$. The rules for AND/OR programs require all non-leaves to have different F. If a leaf shares an F with a non-leaf, it is said to be a call leaf; if it does not, it is said to be a primitive leaf; in our square root program we have one call leaf and three primitive leaves.

In the official semantics the meaning of our AND/OR program is the relation $\mathfrak{m}_{Root}$ in the least solution of

$$\mathfrak{m}_{Root}(n,r) \;.\!\equiv\!.\; \exists\, i,j\, (\mathfrak{m}_{zeroone}(i,j) \wedge \mathfrak{m}_{Iterate}(i,j,n,r))$$

$$\mathfrak{m}_{Iterate}(i,j,n,r) \;.\!\equiv\!.\; (j > n \wedge \mathfrak{m}_{identity}(i,r))$$
$$\vee\; (j \leq n \wedge \mathfrak{m}_{Step})i,j,n,r))$$

$$\mathfrak{m}_{Step}(i,j,n,r) \;.\!\equiv\!.\; \exists \text{next } i, \text{ next } j\, (\mathfrak{m}_{increment}(i,j,\text{next } i,\text{next } j)$$
$$\wedge\; \mathfrak{m}_{Iterate}(\text{next } i, \text{ next } j, \; n\; r)$$

where the primitive relations $\mathfrak{m}_{zeroone}$, $\mathfrak{m}_{identity}$ and $\mathfrak{m}_{increment}$ are defined by

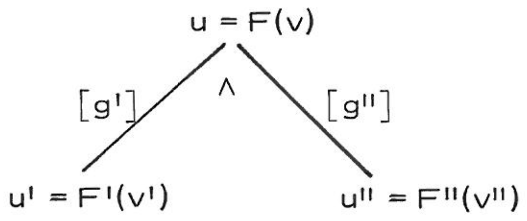
$$\mathfrak{m}_{\text{zeroone}}(i,j) \quad .\equiv. \quad i = 0 \wedge j = 1$$

$$\mathfrak{m}_{\text{identity}}(i,r) \quad .\equiv. \quad i = r$$

$$\mathfrak{m}_{\text{increment}}(i, j, \text{next } i, \text{next } j) \quad .\equiv. \quad (\text{next } i = i + 1) \wedge (\text{next } j = j+2i+3)$$

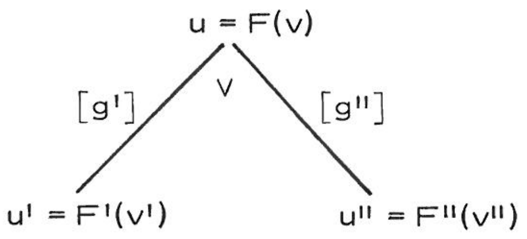
We shall give an equivalent definition of the official semantics of an AND/OR program.

First we describe how an AND/OR program can be transformed into a context-free grammar. The productions of the grammar are given by the non-leaf nodes of the program; an AND node gives one production



$\Rightarrow \qquad F ::= [g'] \; F' [g''] \; F''$

and an OR node gives a production with alternatives



$\Rightarrow \qquad F ::= [g'] \, F' \mid [g''] \, F''$

For our square root program we get the grammar

<u>nonterminals</u>	Root, Iterate, Step		
<u>terminals</u>	zeroone, identity, increment, $[j > n]$, $[j \leq n]$		
<u>productions</u>	(1) Root	::=	zeroone Iterate
	(2) Iterate	::=	$[j > n]$ identity $\mid$ $[j \leq n]$ Step
	(3) Step	::=	increment $\mid$ Iterate

Note how the numbering of the productions shows the corresponding non-leaf node in the program, and the convenience of using an initial capital to signal that a symbol is non-terminal.

Once we have a grammar for a logical program, we can use denotational semantics to give it a meaning. By the conventions in (2) a non-terminal symbol names the syntactic domain of derivation trees, its first letter printed normally names an arbitrary tree in this domain, and its first letter strangely names a semantic function

$\mathfrak{R}$: Root $\rightarrow$ Meanings	R : Root
$\mathcal{I}$: Iteration $\rightarrow$ Meanings	I : Iteration
$\mathcal{S}$: Step $\rightarrow$ Meanings	S : Step

We get the official meaning of our program from the denotation semantics

(1) $\mathfrak{R}[\text{zeroone I}](n,r) \;.\!\equiv\!.\; \exists i,j\,(i = 0 \wedge j = 1 \wedge \mathcal{I}[\text{I}](i,j,n,r))$

(2') $\mathcal{I}[[j > n]\ \text{identity}](i,j,n,r) \;.\!\equiv\!.\; j > n \wedge i = r$

(2'') $\mathcal{I}[[j \leq n]\ \text{S}](i,j,n,r) \;.\!\equiv\!.\; j \leq n \wedge \mathcal{S}[\text{S}](i,j,n,r)$

(3) $\mathcal{S}[\text{increment I}](i,j,n,r) \;.\!\equiv\!.\exists\, \text{next } i,\ \text{next } j\ (\text{next } i = i{+}1 \wedge \text{next } j = j{+}2i{+}3$
$\wedge\ \mathcal{I}[\text{I}](\text{next } i, \text{next } j, n, r))$

We have deliberately left this semantics in a crude form that shows how the conversion from an AND/OR program to such a denotational semantics is a mechanical non-creative process. The resulting semantics is equivalent to the official semantics because there is a bijection between the successful computations of an AND/OR program and the language generated by the corresponding context-free grammar.

One of the ideas behind our approach is that unofficial semantics based on the underlying grammar may be valuable and informative. Our example is typical of a functional AND/OR logical program because the primitive relations $\mathfrak{m}_{\text{zeroone}}$, $\mathfrak{m}_{\text{iterate}}$, $\mathfrak{m}_{\text{increment}}$ are functional. This suggests the denotational semantics

(1) $\mathcal{R}\ [\text{zeroone } I]\ (n)\ \ .{=}.\ \ \mathcal{I}\ [I]\ (0, 1, n)$

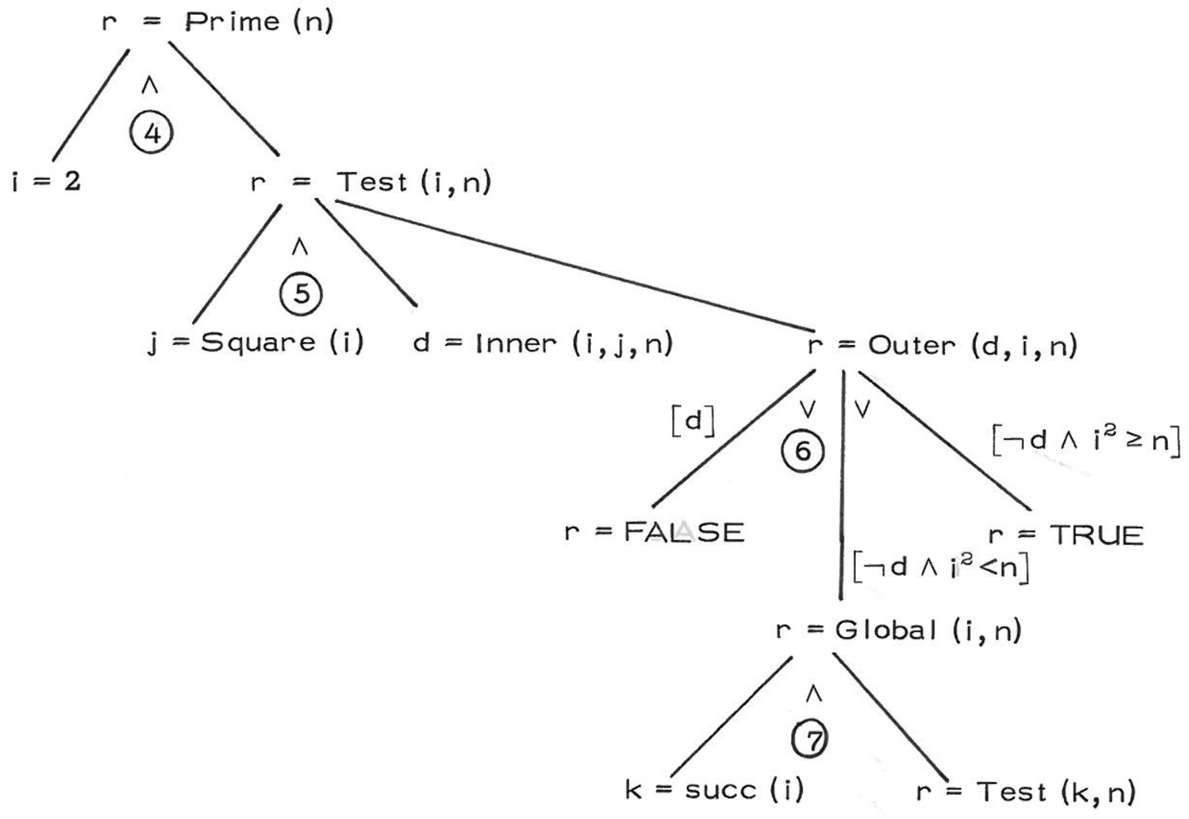
(2') $\mathcal{I}\ [[j>n]\ \text{identity}]\ (i, j, n)\ \ .{=}.\ \ j > n \rightarrow i, \perp$

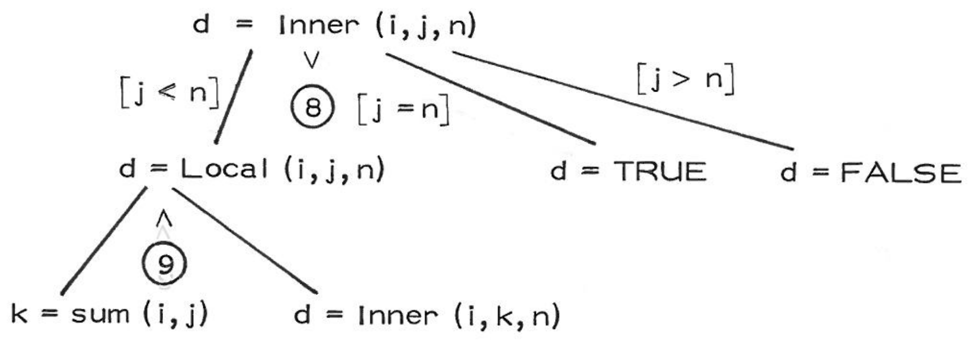
(2'') $\mathcal{I}\ [[j \leq n]\ S]\ (i, j, n)\ \ .{=}.\ \ j \leq n \rightarrow \mathcal{S}\ (S)\ (i, j, n), \perp$

(3) $\mathcal{S}\ [\text{increment } I]\ (i, j, n)\ \ .{=}.\ \ \mathcal{I}[I]\ (i+1,\ j+2i+3, n)$

where we use the symbol $\perp$ for undefined function values.

Let us look at a more complicated example - an AND/OR program for testing if a number n is prime:





The grammar for this AND/OR program is

<u>non terminals</u> Prime, Test, Outer, Global, Inner, Local

<u>terminals</u> two, square, [d], FALSE, TRUE, succ, sum, $[\neg d \wedge i^2 < n]$, $[\neg d \wedge i^2 \geq n]$, $[j < n]$, $[j = n]$, $[j > n]$

<u>productions</u>

(4)	Prime	::=	two Test
(5)	Test	::=	square Inner Outer
(6)	Outer	::=	[d] FALSE | $[\neg d \wedge i^2 < n]$ Global | $[\neg d \wedge i^2 \geq n]$ TRUE
(7)	Global	::=	succ Test
(8)	Inner	::=	$[j < n]$ Local | $[j = n]$ TRUE | $[j > n]$ FALSE
(9)	Local	::=	sum Inner

The official semantics is given by

(4) $\mathcal{P}[\text{two } T]\ (n,r) \ .\!\equiv.\ \exists i\ (i = 2 \wedge \mathfrak{T}[T]\ (i,n,r))$

(5) $\mathfrak{T}[\text{square } I\ O]\ (i,n,r) \ .\!\equiv.\ \exists j,d\ (j = i^2 \ \wedge \mathfrak{I}[I]\ (i,j,n,d)$
$\wedge\ \mathfrak{O}[O]\ (d,i,n,r))$

(6') $\mathfrak{O}[[d]\ \text{FALSE}]\ (d,i,n,r) \ .\!\equiv.\ (d \wedge \neg r)$

(6'') $\mathfrak{O}[[\neg d \wedge i^2 < n]\ G]\ (d,i,n,r) \ .\!\equiv.\ (\neg d \wedge i^2 < n \wedge \mathfrak{G}[G]\ (i,n,r))$

(6''') $\mathfrak{O}[[\neg d \wedge i^2 \geq n]\ G]\ (d,i,n,r) \ .\!\equiv.\ (\neg d \wedge i^2 \geq n \wedge r)$

(7) $\mathfrak{G}[\text{succ } T]\ (i,n,r) \ .\!\equiv.\ \exists k\ (k = i+1 \wedge \mathfrak{T}[T]\ (k,n,r))$

(8') $\mathfrak{I}[[j < n]\ L]\ (i,j,n,d) \ .\!\equiv.\ (j < n \wedge \mathfrak{L}[L]\ (i,j,n,d))$

(8'') $\mathfrak{I}[[j = n]\ \text{TRUE}]\ (i,j,n,d) \ .\!\equiv.\ d$

(8''') $\mathfrak{I}[[j > n]\ \text{FALSE}]\ (i,j,n,d) \ .\!\equiv.\ \neg d$

(9) $\mathfrak{L}[\text{sum } I]\ (i,j,n,d) \ .\!\equiv.\ \exists k\ (k = i+j \wedge \mathfrak{I}[I]\ (i,k,n,d))$

As this AND/OR program is functional we also have the more natural semantics

(4) $\mathcal{P}[\text{two } T]\ (n) \ .\!=.\ \tau[T]\ (2,n,r)$

(5) $\tau[\text{square } I\ O]\ (i,n) \ .\!=.\ \underline{\text{let}}\ d = \mathfrak{I}[I]\ (i, i^2, n)$
$\underline{\text{in}}\ \mathfrak{O}[O]\ (d,i,n)$

(6') $\mathfrak{O}[[d]\ \text{FALSE}]\ (d,i,n) \ .\!=.\ d \rightarrow \text{FALSE}, \perp$

(6'') $\mathcal{O}[[\neg d \wedge i^2 < n]\ G]\ (d, i, n)\ .=.\ (\neg d \wedge i^2 < n) \to \mathcal{G}[G]\ (i, n), \perp$

(6''') $\mathcal{O}[[\neg d \wedge i^2 \geq n]\ \mathrm{TRUE}]\ (d, i, n)\ .=.\ (\neg d \wedge i^2 \geq n) \to \mathrm{TRUE}, \perp$

(7) $\mathcal{G}[\mathrm{succ}\ T]\ (i, n)\ .=.\ \mathcal{T}[T]\ (i+1, n)$

(8') $\mathcal{I}[[j < n]\ L]\ (i, j, n)\ .=.\ (j < n) \to \mathcal{L}[L]\ (i, j, n), \perp$

(8'') $\mathcal{I}[[j = n]\ \mathrm{TRUE}]\ (i, j, n)\ .=.\ (j = n) \to \mathrm{TRUE}, \perp$

(8''') $\mathcal{I}[[j > n]\ \mathrm{FALSE}]\ (i, j, n)\ .=.\ (j > n) \to \mathrm{FALSE}, \perp$

(9) $\mathcal{L}[\mathrm{sum}\ I]\ (i, j, n)\ .=.\ \mathcal{I}[I]\ (i, i+j, n)$

In spite of the fact that TRUE and FALSE are the only defined values of these functions, the functions are not relations because they can have undefined values. The domain specification for our functions is

$$\begin{array}{lllll}
\mathcal{P}: & \text{Prime} & \to & \text{Number} & \to \text{Boolean} \\
\mathcal{T}: & \text{Test} & \to & \text{Number}^2 & \to \text{Boolean} \\
\mathcal{O}: & \text{Outer} & \to & \text{Boolean} \times \text{Number}^2 & \to \text{Boolean} \\
\mathcal{G}: & \text{Global} & \to & \text{Number}^2 & \to \text{Boolean} \\
\mathcal{I}: & \text{Inner} & \to & \text{Number}^3 & \to \text{Boolean} \\
\mathcal{L}: & \text{Local} & \to & \text{Number}^3 & \to \text{Boolean}
\end{array}$$

and there is an undefined value $\perp$ in the set Boolean.

2. PROLOG programming

In (5) Kowalski introduced the logical programming language PROLOG. A PROLOG program consists of statements of the forms

$A \leftarrow$	assertions
$B \leftarrow C_1 \ldots C_m$	definitions
$\leftarrow D$	goals

and the execution of the program is an attempt to refute its goals using its assertions and definitions. The natural PROLOG solution of most problems requires an infinite list of assertions, but each implementation of the language adopts conventions for avoiding these infinite lists. Our conventions can be seen in the program for finding square roots

(1)	Root (n, r)	$\leftarrow$	Iterate (0, 1, n, r)
(2')	Iterate (i, j, n, i)	$\leftarrow$	$[j > n]$
(2'')	Iterate (i, j, n, r)	$\leftarrow$	$[j \leq n]$ Step (i, j, n, r)
(3)	Step (i, j, n, r)	$\leftarrow$	Iterate (i+1, j+2i+3, n, r)

Let us follow the execution of this program with the goal

$\leftarrow$ Root (7, r)

The pattern match on (1) gives the new goal

$\leftarrow$ Iterate (0, 1, 7, r)

This cannot be matched on (2') because the conditional (guard) $[j > n]$ fails so we match on (2'') to get

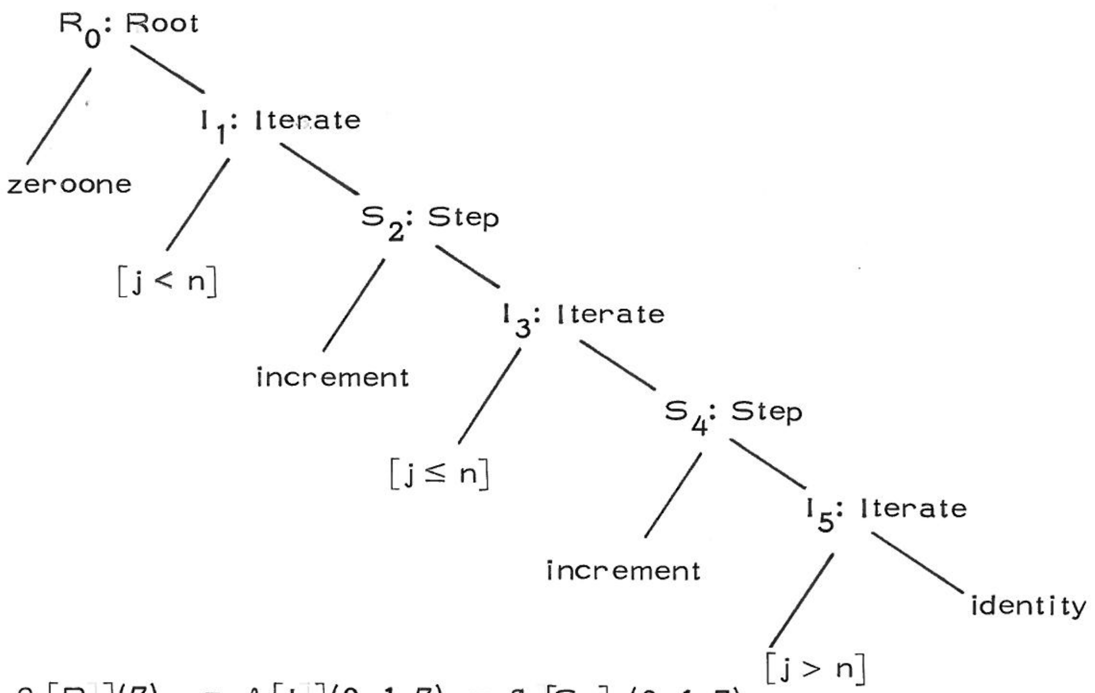
$\leftarrow$ Step (0, 1, 7, r)

Matching on (3), (2'') and (3) gives the goals

$\leftarrow$ Iterate (1, 4, 7, r)
$\leftarrow$ Step (1, 4, 7, r)
$\leftarrow$ Iterate (2, 9, 7, r)

and the final match on (2') gives the solution r = 2.

Compare this computation with the way the semantics of the grammar in the last section gives a meaning to the derivation tree



$$\begin{aligned}\mathcal{R}[R_0](7) &= \mathcal{I}[I_1](0,1,7) = \mathcal{S}[S_2](0,1,7)\\ &= \mathcal{I}[I_3](1,4,7) = \mathcal{S}[S_4](1,4,7)\\ &= \mathcal{I}[I_5](2,9,7) = 2\end{aligned}$$

Our semantics mimics the computation of the PROLOG program; it is defined for a grammar that is derived from the program by introducing terminal symbols for "primitive" operations. You have already seen the semantics and the grammar for the PROLOG program.

(4) Prime (n, r) $\leftarrow$ Test (2, n, r)

(5) Test (i, n, r) $\leftarrow$ Inner (i, i^2, n, d) Outer (d, i, n, r)

(6') Outer (d, i, n, FALSE) $\leftarrow$ $[d]$

(6'') Outer (d, i, n, r) $\leftarrow$ $[\neg d \wedge i^2 < n]$ Global (i, n, r)

(6''') Outer (d, i, n, TRUE) $\leftarrow$ $[\neg d \wedge i^2 \geq n]$

(7) Global)i, n, r) $\leftarrow$ Test (i+1, n, r)

(8') Inner (i, j, n, d) $\leftarrow$ $[j < n]$ Local (i, j, n, d)

(8'') Inner (i, j, n, TRUE) $\leftarrow$ $[j = n]$

(8''') Inner (i, j, n, FALSE) $\leftarrow$ $[j > n]$

(9) Local (i, j, n, d) $\leftarrow$ Inner (i, i+j, n, d)

As before the computations of this PROLOG program are mimicked by the semantics for the corresponding grammar in the last section.

3. LUCID programming

In (1) Ashcroft and Wadge introduced the logical programming language LUCID. A LUCID program is a set of equations specifying a set of variables. A variable may be specified directly by an equation like V = E or indirectly by two equations

first V = E.   next V = E

every variable except "input" must be specified and no variable may be specified more than once. Our first LUCID program for finding a square root might be

```
n  =  first input
first i  =  0
next i  =  i + 1
first j  =  1
next j  =  j + 2i + 3
output  =  i as soon as j > n
```

In the official semantics each of the program variables acquires an infinite sequence of values when "input" is given an infinite sequence of values. Suppose the first value given to "input" is 7. The first line of the program ensures that the sequence for n is (7,7,7,7,...); the next two lines ensure that the sequence for i is (0,1,2,3,...). Because functions work pointwise, the next two lines ensure that the sequence for j is (1,4,9,16,...) so the sequence for j > n is

(FALSE, FALSE, TRUE, TRUE ...)

and the rule for as soon as ensures that the sequence for "output" is (2,2,2,...).

Another LUCID program for finding square roots is

```
        n = first input
(1)     first (i,j) = (0, 1)
(2)     output = i as soon as j ¬ n
(3)     next (i,j) = (i+1, j+2i+3)
```

This is so similar to the grammar

```
(1)     Root      ::=   zeroone Iterate
(2)     Iterate   ::=   [j > n] identity | [j ≤ n] Step
(3)     Step      ::=   increment Iterate
```

that the semantics of this grammar in section 1 give the essential part of the official LUCID meaning of the program – given an input they determine an output.

Now compare the LUCID program for testing if a number is prime with the grammar in section 1

```
        n = first input
(4)     first i = 2                           Prime = two Test
        begin
(5)         first j = $i^2$                     Test = square Inner Outer
(8)         d = j eq n as soon as j ≥ n       Inner ::= [j < n] Local | [j=n] TRUE
                                                                       | [j>n] FALSE
(9)         next j = i + j                    Local = sum Inner
        end
(6)     output = ¬ d as soon as d ∨ $i^2$ ≥ n   Outer = [d] FALSE
                                                      | [¬d ∧ $i^2$ < n] Global
                                                      | [¬d ∧ $i^2$ ≥ n] TRUE
(7)     next i = i + 1                        Global = succ Test
```

Again there is a production for every LUCID equation, and the natural semantics for the grammar gives the essential part of the official LUCID meaning of the program. However the process of devising a grammar for a LUCID program sometimes requires imagination - for bizarre programs that "reach into the future" there may not be an equivalent grammar.

4. Extended attribute programming

Attribute grammars are not usually thought of as logical programs, but this impression is misleading for extended attribute grammars (6). Compare the PROLOG program for finding square roots with

(1) $\text{Root}\ (\downarrow n \uparrow r) \quad = \quad \text{Iterate}\ (\downarrow 0 \downarrow 1 \downarrow n \uparrow r)$

(2') $\text{Iterate}\ (\downarrow i \downarrow j \downarrow n \uparrow i) \quad = \quad \text{greater than}\ (\downarrow j \downarrow n \uparrow \text{TRUE})$

(2'') $\text{Iterate}\ (\downarrow i \downarrow j \downarrow n \uparrow j) \quad = \quad \text{greater than}\ (\downarrow j \downarrow n \uparrow \text{FALSE})\ \text{Step}\ (\downarrow i \downarrow j \downarrow n \uparrow r)$

(3) $\text{Step}\ (\downarrow i \downarrow j \downarrow n \uparrow r) \quad = \quad \text{Iterate}\ (\downarrow i+1 \downarrow j+2i+3 \downarrow n \uparrow r)$

Those with access to attribute grammar based compiler systems should be able to execute this grammar as a logical program. Strictly speaking the underlying context free grammar for our square root attribute grammar is

```
Root      ::=   Iterate
Iterate   ::=   greater than | greater than | greater than Step
Step      ::=   Iterate
```

but the semantics in section 1 gives the same meaning to the program as the semantics for the underlying grammar

$$\mathscr{R}\,[\mathrm{I}]\,(n) = \mathscr{J}\,[\mathrm{I}]\,(0,1,n)$$
$$\mathscr{J}\,[\text{greaterthan}]\,(i,j,n) = j > n \rightarrow i, \perp$$
$$\mathscr{J}\,[\text{greaterthan S}]\,(i,j,n) = j \leq n \rightarrow \mathscr{S}\,[\mathrm{S}]\,(i,j,n), \perp$$
$$\mathscr{S}\,[\mathrm{I}]\,(i,j,n) = \mathscr{J}\,[\mathrm{I}]\,(i+1, j+2i+3, n)$$

Notice that this semantics is essentially functional; AND/OR and PROLOG are more powerful logical, programming languages than extended attribute grammars because they allow relations as primitives; LUCID is more powerful because it allows value sequences. This extra power is not needed for the problem of testing primality, so we have an attribute grammar solution

(4) Prime $(\downarrow n \uparrow r)$ ::= Test $(\downarrow 2 \downarrow n \uparrow r)$

(5) Test $(\downarrow i \downarrow n \downarrow r)$ ::= Inner $(\downarrow i \downarrow i^2 \downarrow n \uparrow d)$ Outer $(\downarrow d \downarrow i \downarrow n \uparrow r)$

(6') Outer $(\downarrow \mathrm{TRUE} \downarrow i \downarrow n \uparrow \mathrm{FALSE})$ ::=

(6'') Outer $(\downarrow \mathrm{FALSE} \downarrow i \downarrow n \uparrow r)$ ::= less than $(\downarrow i^2 \downarrow n \uparrow \mathrm{TRUE})$ Global $(\downarrow i \downarrow n \uparrow r)$

(6''') Outer $(\downarrow \mathrm{FALSE} \downarrow i \downarrow n \uparrow \mathrm{TRUE})$ ::= less than $(\downarrow i^2 \downarrow n \uparrow \mathrm{FALSE})$

(7) Global $(\downarrow i \downarrow n \uparrow r)$ ::= Test $(\downarrow i+1 \downarrow n \uparrow r)$

(8') Inner $(\downarrow i \downarrow j \downarrow n \uparrow r)$ ::= less $(\downarrow j \downarrow n \uparrow \mathrm{TRUE})$ Local $(\downarrow i \downarrow j \downarrow n \uparrow r)$

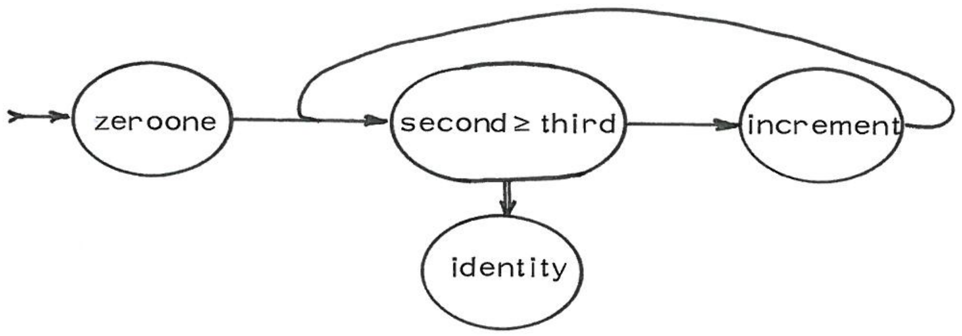
(8'') Inner $(\downarrow i \downarrow j \downarrow n \uparrow j=n)$ ::= less $(\downarrow j \downarrow n \uparrow \mathrm{TRUE})$

(9) Local $(\downarrow i \downarrow j \downarrow n \uparrow r)$ ::= Inner $(\downarrow i \downarrow j+1 \downarrow n \uparrow r)$

This attribute grammar is similar to the PROLOG solution, and its underlying context-free grammar is sufficiently similar to the grammar in section 1 that the semantics of both grammars give the same meaning to the program. Note that this would not have been so if we had used "less than" instead of "less" in (8') and (8''), because the underlying grammar would have been ambiguous.

5. Data Flow Machines

Unlike von Neumann machines, data flow machines have no store; data flows between processes and a process transmits its results as soon as it has received its arguments. The data flow machine for our square root problem might be



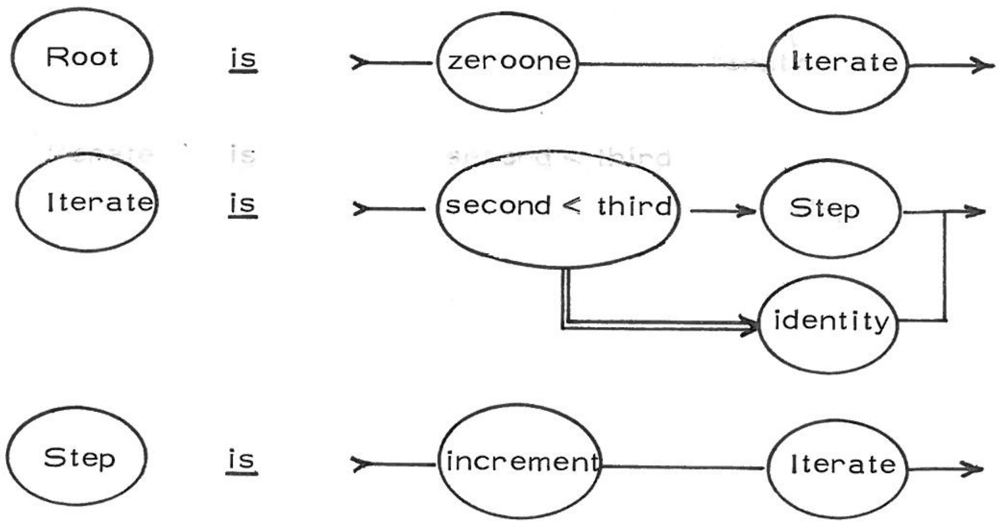
where one number is given to the zeroone machine, number triples flow around the internal arrows until a triple reaches the identity machine, and this machine then presents a single number as its result. In a variant of a notation being developed by R.F. Barton and his associates (7) this machine would be specified by the expression

$$\text{zeroone (increment identity)}^{(2 \geq 3)*}$$

which is very similar to the regular expression for the square root grammar in section 1

$$\text{zeroone } ([j < n] \text{ increment})^{*} \; [j \geq n] \text{ identity}$$

This similarity should come as no surprise because data flow machines, like logical programs, do not have assignment. We can use data flow machines to give an operational semantics of our grammar. We can define three data flow machines by



We give these machines an operational semantics by

$$\begin{aligned}
\mathfrak{R}\text{oot}\ (n) &= \mathfrak{I}\text{terate}\ (0, 1, n)\\
\mathfrak{I}\text{terate}\ (i, j, n) &= j > n \rightarrow i,\ \mathfrak{S}\text{tep}\ (i, j, n)\\
\mathfrak{S}\text{tep}\ (i, j, n) &= \mathfrak{I}\text{terate}\ (i+1, j+2i+3, n)
\end{aligned}$$

This operational semantics is correct because we can prove:

(1) $\mathfrak{R}\text{oot}\ (n) = \mathfrak{R}[R]\ n$ where $\mathfrak{R}$ is the function in section 1 and R is the unique derivation tree determined by n;

(2) $\mathfrak{I}\text{terate}\ (i, j, n) = \mathfrak{I}[I]\ (i, j, n)$ where $\mathfrak{I}$ is the function in section 1 and I is the unique derivation tree determined by i, j, n;

(3) $\mathfrak{S}\text{tep}\ (i, j, n) = \mathfrak{S}[S]\ (i, j, n)$ where $\mathfrak{S}$ is the function in section 1 and S is the unique derivation tree determined by i, j, n.

We prove this by induction on the length of computations in the three data flow machines. Suppose (1), (2), (3) are true for computations of length $\leq$ k. If the computation of the Root machine from n has k+1 steps then the unique derivation tree R determined by n is

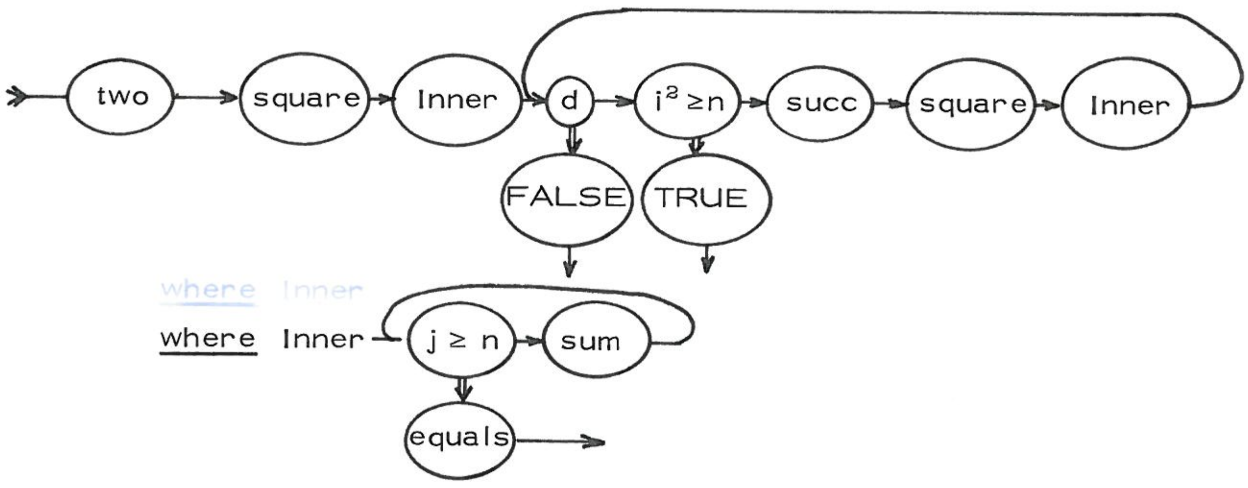
$$\begin{tikzcd}
 & \text{Root} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{zeroone} & & I
\end{tikzcd}$$

the induction hypothesis gives $\mathcal{I}$terate $(0, 1, n) = \mathcal{I}[I]\ (0, 1, n)$ and we have $\mathcal{R}$oot $(n) = \mathcal{R}[R]\ (n)$. Analogous argument give (3) for computations of the Step machine of length k+1 and (2) for computations of the Iterate machine of length k+1 when $j \leq n$. When $j > n$ there is a one step computation of the Iterate machine and we have

$$\mathcal{I}\text{terate}\ (i, j, n) \;=\; \mathcal{I}([j > n]\ \text{identity}) = i$$

Since there are no other one step computations, (1), (2), (3) hold for computations of length ≤ 1, so they always hold.

A similar argument shows that the data flow machine



is a correct solution of the primality problem. This machine can be specified by the expression

$$\text{two square Inner ((succ square Inner, not d)}^{(d \vee i^2 \geq n)*}$$
$$\underline{\text{where}}\ \text{Inner} = (\text{sum, equals})^{(j \geq n)*}$$

which is very close to the regular expression for the primality grammar in section 1

$$\text{two square Inner}\ ([\neg d \wedge i^2 < n]\ \text{succ square Inner})^*([d]\ \text{FALSE} \mid [\neg d \wedge i^2 \geq n]\ \text{TRUE})$$
$$\underline{\text{where}}\ \text{Inner} = ([j < n]\ \text{sum})^*\ ([j = n]\ \text{TRUE} \mid [j > n]\ \text{FALSE})$$

References

(1) E.A. Ashcroft, W.W. Wadge: Lucid: a non procedural language with iteration. CACM 20 (1977) 519-526.

(2) M.J.C. Gordon: The denotational description of programming languages. Springer Verlag, 1979.

(3) D. Harel: And/or programs, a new approach to structured programming. ACM Tr. Prog. Lang. 2 (1980) 1-17.

(4) R. Kowalski: Algorithm = Logic + Control. CACM 22 (1979) 425-436.

(5) R. Kowalski: Predicate logic as programming language. IFIP 74, Amsterdam, pp. 569-574.

(6) O.L. Madsen: On defining semantics by means of extended attribute grammars. Springer Lecture Notes in Computer Science 94, (1980).

(7) F.M. Tonge, R.M. Cowan: Structured process description. Univ. Calif. Irvine preprint (1980), submitted for publication.