DIKU-TRYK
AUGUST "84

DIKU

RAPPORT
NR: 84/7
ISSN 0107-8283

DATALOGI 2 NOTES: FUNCTIONS, EXPRESSIONS, PROGRAMMING LANGUAGES, COMPUTABILITY

Neil D. Jones

July 1984

Datalogisk Institut, Københavns Universitet
Institute of Datalogy, University of Copenhagen
Sigurdsgade 41 DK-2200 København N

DATALOGI 2 NOTES: FUNCTIONS, EXPRESSIONS, PROGRAMMING LANGUAGES, COMPUTABILITY

Neil D. Jones

July 1984

Contents

cR 5,20

I FUNCTIONS AND EXPRESSIONS

The first three chapters of these notes provide a semiformal introduction to several fundamental concepts found in modern programming languages: evaluation of expressions, the binding of names to values, recursively defined functions and flow of control. After developing some necessary notation, these concepts are explored by giving "operational semantics" (= rules for program execution) of three minilanguages: the lambda calculus, systems of recursion equations and a flowchart language. The same principles for program execution are found in more sophisticated languages (Pascal, LISP, etc.) and should aid in understanding the runtime structures and translation methods found later in DAT 2.

The treatment is "semiformal" in that on the one hand, the various terms and algorithms used are precisely defined; for example programs in the minilanguages may be unambiguously executed by hand, a feature unfortunately not present in the usual programming language manuals. On the other hand formalized mathematical reasoning or machine-executable programs will not be used.

What is a programming language? From a user's viewpoint a language P is a "black box" which can run programs, and so can be thought to consist of:

1. P-programs: The set of all admissible programs in the language P.

2. For each program p ∈ P-programs,

P-input(p): The set of possible runtime inputs to p (this may be empty)

P-output(p): A set including all the possible runtime outputs resulting from running program p

P-eval(p): P-input(p) $\overset{P}{\rightarrow}$ P-output(p), the input-output function computed by p. This may be a *partial* function (i.e. undefined on some inputs) due to nonterminating computations by p

This description is not all-encompassing, as it lacks some advanced programming language features (e.g. communication with other programs or user terminals, parallelism, etc.), but it will suffice for this part of DAT 2.

A precise description of these advanced features would require methods beyond the scope of the course.

1. NOTATIONS FOR FUNCTIONS AND TYPES

The concept "function" and the notations used to denote and manipulate functions are central to both mathematics and computer science, and form the inner core of nearly all programming languages. Informally, a function is a correspondence between elements of two sets, that is a mapping from each element of the first set to exactly one element of the second. More formally, a function f from A to B is a set of pairs (a,b) from $A \times B$ with the properties:

1. $\forall a \in A \quad \forall b,b' \in B$

   $(a,b) \in f$ and $(a,b') \in f$ implies $b = b'$

2. $\forall a \in A \quad \exists b \in B \; (a,b) \in f$

Property 1 says f is <u>single-valued</u>, i.e. each a in A is mapped to at most one $b \in B$ (this b is usually written f(a)). Property 2 says that f is <u>total</u>, i.e. that f(a) exists for every a in A. In computer science we will also have use for <u>partial</u> functions, that is, subsets f of $A \times B$ which satisfy 1 but not 2. One example is the input-output function computed by a program, which may not be defined on some inputs due to infinite loops during program execution.

It is important not to confuse the concept of a <u>function</u>, i.e. a single-valued input/output relation with the concept of an <u>algorithm</u>, which specifies a way to compute a function. For example the facturial function n! is abstractly just the set {(0,1, (1,1), (2,2), (3,6), (4,24) ...} while there exist many algorithms for computing n! , e.g. Pascal programs using iteration or recursion, LISP programs, etc.

The Pascal <u>function</u> declaration is thus an algorithm rather than a mathematical function. Further it is not even single-valued since f(5) may return different values on different calls due to changes in global variables!

In case f is a function from A to B we write $f: A \to B$, and say that f has <u>type</u> $A \to B$. For example, $N!: N \to N$ where $N = \{0,1,2, \ldots\}$ is the set of natural numbers. More generally we can form <u>type expressions</u>, each of which denotes a set, by the following rules:

1. The names of various standard sets are *atomic* type expressions, including

   N denoting $\{0,1,2, \ldots\}$

   Z denoting $\{\ldots, -2,-1,0,1,2, \ldots\}$

   Bool denoting {true, false}

2. If S and T are type expressions denoting sets A and B (respectively) then

   $S \times T$ denotes $\{(a,b) \mid a \in A \text{ and } b \in B\}$

   $S \to T$ denotes the set of total functions from A to B

   $S \overset{p}{\to} T$ denotes the set of partial functions from A to B

   $S + T$ denotes $\{(a,1) \mid a \in A\} \cup \{(b,2) \mid b \in B\}$

<u>Examples</u> Let A = {red,green} and B = {red,blue}. Then

$A \times B$ = {(red,red),(red,blue),(green,red),(green,blue)}

$A + B$ = {(red,1),(green,1),(red,2),(blue,2)}

Letting #A denote the number of elements in set A, we see that $\#(A \times B) = (\#A) \times (\#B)$ while $\#(A + B) = (\#A) + (\#B)$. The (presumably well-known) set $A \times B$ is called the *cartesian product* of A and B, while the less familiar $A + B$ is called the *disjoint sum* of A and B. Note that although $A + B$ resembles the ordinary set union $A \cup B$, it is not the same since elements of the intersection "appear twice", e.g. red in the example above.

Consider the following nine partial functions $f_1, f_2, \ldots, f_9$ from A to B

Function f_i	f_i(red)	f_i(blue)
f_1	f_1(red) = red	f_1(blue) = red
f_2	f_2(red) = red	f_2(blue) = blue
f_3	f_3(red) = blue	f_3(blue) = red
f_4	f_4(red) = blue	f_4(blue) = blue
f_5	f_5(red) = undefined	f_5(blue) = undefined
f_6	f_6(red) = undefined	f_6(blue) = red
f_7	f_7(red) = undefined	f_7(blue) = blue
f_8	f_8(red) = red	f_8(blue) = undefined
f_9	f_9(red) = blue	f_9(blue) = undefined

The sets of total and partial functions are:

$$A \rightarrow B = \{f_1, f_2, f_3, f_4\}$$
$$A \overset{p}{\rightarrow} B = \{f_1, f_2, \ldots, f_9\}$$

1.1 Type Constructors in Pascal

The product and sum type notations can be extended in an obvious way to more than two arguments: $A_1 \times \ldots \times A_n$ and $A_1 + \ldots + A_n$. In a Pascal context the atomic types are

Boolean, integer, real, char

and the product and sum types could be declared as records and variant records (respectively):

```
type  product  =  record a1 : A1  ;
                         a2 : A2  ;
                         ...
                         an : An
                  end ;
      tag      =  1 .. n ;
      sum      =  record
                    case i : tag of
                      1: (a1 : A1)  ;
                      2: (a2 : A2)  ;
                       ...
                      n: (an : An)
                  end
```

1.2 Examples of Type Expressions for Functions

In general we write t : T to indicate that expression t is in the set denoted by T, for example (5,6) : N × N . Following are several examples of function definitions and their corresponding types.

Function Definition		Type	
f(x)	$= x^2$	f	: N → N
g(n)	= <u>if</u> n = 0 <u>then</u> 1 <u>else</u> n * g(n − 1)	g	: N → N
h(m,n)	= m + n	h	: N × N → N
k(m,n)	= (m + n, m − n)	k	: N × N → N × N
sum(n,f)	= f(0) + f(1) + ... + f(n)	sum	: N × (N → N) → N
twice(f,n)	= f(f(n))	twice	: (N → N) × N → N
divide(a,b)	= a/b	divide	: $N \times N \overset{p}{\rightarrow} N$ (partial)
P-eval(p) where p ∈ P-programs		P-eval(p)	: P-input(p) $\overset{p}{\rightarrow}$ P-output(p)
add(n)	= ℓ, where $\ell(x) = x + n$	add	: N → (N → N)
power(n)	= p, where $p(x) = x^n$	power	: N → (N → N)

The list above contains

- partial functions divide and P-eval(p)
- a recursively defined function g (note: g(n) = n!)
- higher-order functions sum and twice, which have functions as asguments
- higher-order functions add and power, which yield functions as values. For instance, the value of add(1) is the function which, given any argument value x, returns x + 1, and power(2) is the function which, given argument x, returns x^2. Thus

  add(1) (7) = 8 and
  power(2) (7) = 49

Notational Conventions

A → B → C means A → (B → C). if f: A → B → C then f(a): B → C for all a ∈ A, so f is a function-producing function

f a means f(a) provided f: A → B
f a b means (f(a))(b) provided f: A → B → C

1.3 Notations Used to Define Functions

We have just seen several examples defining functions $f\colon A_1 \times A_2 \times \ldots \times A_n \to B$ by means of equations of the form

$$f(x_1, \ldots, x_n) = e$$

where e is an expression built from constants and the variables $x_1, \ldots, x_n$ by use of arithmetic operators and the application of functions to arguments. The types of e and its subexpressions are straightforwardly determined, for example

1. $x_1\colon A_1, \ldots, x_n\colon A_n$
2. $e : B$
3. if e contains subexpression $e_1(e_2)$ where

   $e_1\colon C \to D$ and $e_2\colon C$, then $e_1(e_2)\colon D$
4. if $e_1\colon N$ and $e_2\colon N$ then $e_1 + e_2\colon N$

Following are the types of the right sides of the equations for k and twice:

$$\overbrace{(\overbrace{\overset{N}{m} + \overset{N}{n}}^{N},\quad \overbrace{\overset{N}{m} - \overset{N}{n}}^{N})}^{N \times N} \qquad\qquad \overbrace{\underbrace{(\ f}_{}^{N \to N}\quad \overbrace{\overbrace{(\ f}^{N \to N}\quad \overset{N}{(n)}}^{N}\ )}^{N}$$

Relation to Pascal

In Pascal an algorithm to compute a function $f\colon A_1 \times \ldots \times A_n \to B$ would naturally be represented by

```
function F (a₁: A₁ ; ... ; aₙ: Aₙ): B;
  .... F := ...
end
```

While Pascal allows arguments $a_1, \ldots, a_n$ of any definable type (including functions), the result type B is severely restricted (to pointer, subrange or scalar types). Consequently functions k, add and power cannot be directly expressed in Pascal, although they can in LISP.

Lambda Notation

Lambda notation was developed by Alonzo Church [Chu36] as a way to write expressions denoting functions. Since then it has been developed into a formalism for studying computability theory (the "lambda calculus". See [Chu51]) and was the basis for the design of the programming language LISP [McC62].

Arithmetic expressions alone are not suitable for defining functions due to various ambiguities. For one example, it is not clear whether the expression n! denotes the factorial function as a whole (of type $n!\colon N \to N$) or its value given the current value of n (type $n!\colon N$). Another problem: if $y^2 + x$ is regarded as a function of two variables, should $(y^2 + x)(3,4)$ have value $13 = 3^2 + 4$ or $19 = 4^2 + 3$?

These problems are resolved by using notation $\lambda x \,.\, e$ to denote a function f of one variable. The value of f(5), for example, is the value of the expression got by substituting 5 for the occurrences of x in e which are bound by the λx. Thus functions add and power can be defined by:

$$\mathrm{add}(n) = \lambda x \,.\, x + n$$
$$\mathrm{power}(n) = \lambda x \,.\, x^n$$

so $\mathrm{add}(5) = \lambda x \,.\, x + 5$, i.e. the function which adds 5 to its argument. Consequently $\mathrm{add}(5)\ (7) = (\lambda x.x+5)(7) = 7 + 5 = 12$.

The notation may be extended by giving the type of x, in the form $\lambda x : T \,.\, e$. If e has type B then $\lambda x : A \,.\, e$ will have type $A \to B$. A function of several arguments may be denoted as follows:

$$\lambda x_1{:}A_1, x_2{:}A_2, \ldots, x_n{:}A_n \,.\, e$$

which has type $A_1 \times \ldots \times A_n \to B$, provided e has type B. If clear from context the types A_i may be dropped. Note that according to these rules;

$\lambda x{:}A, y{:}B \,.\, e{:}C$ has type $(A \times B) \to C$ while

$\lambda x{:}A \,.\, \lambda y{:}B \,.\, e{:}C$ has type $A \to (B \to C)$.

These are *not* the same (e.g. h and add have different types).

Thus some of the examples can be written:

$$\begin{aligned}
f &= \lambda x\,.\,x^2 && \text{type } N \to N \\
k &= \lambda m,n\,.\,(m+n,\ m-n) && \text{type } N \times N \to N \times N \\
h &= \lambda m,n\,.\,m+n && \text{type } N \times N \to N \\
add &= \lambda n\,.\,\lambda x\,.\,x+n && \text{type } N \to (N \to N)
\end{aligned}$$

Note that $\lambda x\,.\,x^2$ is not a function in itself; rather, it is an expression which denotes a function. More generally, a clear understanding of the relation between a *textual* object (such as a program) and its *meaning* (such as the input--output function it denotes) is essential in the study of programming languages. This relation is usually called the semantics of the programming language ([Gor79],[Sto77]). Analogously, the subject *mathematical logic* concerns the relation between mathematical notations such as formulas and the objects they denote.

Notation for Updating Functions

The notation $[a_1 \mapsto b_1,\ a_2 \mapsto b_2,\ \dots,\ a_n \mapsto b_n]$ is used to denote the finite function $f\colon A \to B$ such that

$$f(a_1) = b_1,\ f(a_2) = b_2,\ \dots,\ f(a_n) = b_n$$

If $g\colon A \to B$ is a function, the notation $g[a_1 \mapsto b\ ,\ \dots,\ a_n \mapsto b_n]$ is used to denote the function $h\colon A \to B$ such that

$$h(a_n) = b_1$$

$$\dots$$

$$h(a_n) = b_n$$

$$h(a) = g(a) \text{ for any } a \in A \setminus \{a_1, a_2, \dots, a_n\}$$

In other words, $g[a_1 \mapsto b_1,\ \dots,\ a_n \mapsto b_n]$ is the result of *updating* or *overwriting* g with the finite function $[a_1 \mapsto b_1,\ \dots,\ a_n \mapsto b_n]$.

1.4 A Type Constructor for Sequences

One often needs sequences of values, of undetermined or varying length, on order to describe programming languages. Examples include program input or output files, and the texts of programs themselves. For this we may introduce a new type constructor as follows:

Definition Let A be a type expression. Then A^* is also a type expression, which denotes the set of all finite sequences of elements of the set denoted by A. Following is some notation for operators on A^*.

1. NIL denotes the empty list, namely a sequence containing no elements.
2. $[a_1, a_2, \dots, a_n]$ denotes the sequence containing $a_1, \dots, a_n$ in that order, for $n \geq 0$. Consequently NIL = [].
3. If $[a_1, \dots, a_m]$ is a sequence and a an element then

   $$a :: [a_1, \dots, a_m] \quad = \quad [a, a_1, \dots, a_m]$$

   consequently (note that :: associates from the right)

   $$[a_1, \dots, a_n] \quad = \quad a_1 :: a_2 :: \dots :: a_n :: \mathrm{NIL}$$

Relation to LISP

LISP's data structures are binary lists generable by the productions

```
<list> ::= <atom> | (<list> . <list>)

<atom> ::= a string of letters, digits and other symbols
```

Any sequence as defined above (for example [3,1,4]) can be represented by a LISP list (for example (3 . (1 . (4 . NIL)))). Clearly :: is just an infix notation for the CONS operator. A *constant* sequence such as [3,1,4] can be represented in LISP's "list notation" as (3 1 4), whereas an *expression* such as $[x+y, x-y]$ requires explicit operators in LISP, for example

```
     (CONS(PLUS x y) (CONS(DIFFERENCE x y) NIL))

or   (LIST(PLUS x y) (DIFFERENCE x y))
```

Operations on Sequences or Binary Lists

Sequences	Binary Lists
x :: y	(CONS x y)
$\text{head}([a_1, a_2, \ldots, a_n]) = a_1$	$(\text{HEAD } (\ell_1 . \ell_2)) = \ell_1$
$\text{tail}([a_1, a_2, \ldots, a_n]) = [a_2, \ldots, a_n]$	$(\text{TAIL } (\ell_1 . \ell_2)) = \ell_2$
$\text{atom}(x) = \begin{cases} \text{false if } x \text{ is a sequence} \\ \text{true if not} \end{cases}$	$(\text{ATOM } x) = \begin{cases} \text{NIL if } x = (\ell_1 . \ell_2) \\ \text{T if } x \text{ is an atom} \end{cases}$
$\text{equal}(x,y) = \begin{cases} \text{true if } x = y \\ \text{false if not} \end{cases}$	$(\text{EQUAL } x\ y) = \begin{cases} \text{T if } x = y \\ \text{NIL if not} \end{cases}$

2. EVALUATION OF EXPRESSIONS

2.1 Expressions without Variables

Expressions denote values, for example $3 + 4$, $5 + 2$, and $17 - 5 * 2$ all denote the number 7. The rules according to which expressions denote values are called the *semantics* of the (very simple) programming language of expressions. Some example semantic rules follow, where the semantic function N-eval has functionality

N-eval : Numeric expressions → N

Note that N-eval maps a *textual* object which might appear in a Pascal program, e.g. "$17 - 5 * 2$" into a *mathematical* object, e.g. the natural number 7. In order to keep this distinction clear, we have extended the functional notation by writing textual arguments in special brackets, for example N-eval$[\![17 - 5 * 2]\!] = 7$.

N-eval$[\![d]\!]$	$= d$	for digits $d = 0, 1, \ldots, 9$
N-eval$[\![n\,d]\!]$	$= 10 *$ N-eval$[\![n]\!] + d$	where d is a digit and n a numeral
N-eval$[\![e_1 + e_2]\!]$	$=$ N-eval$[\![e_1]\!]$ + N-eval$[\![e_2]\!]$	
N-eval$[\![e_1 * e_2]\!]$	$=$ N-eval$[\![e_1]\!]$ * N-eval$[\![e_2]\!]$	

The last two rules are not as trivial as they look, since they explain the connection between the *syntactical* symbols +, * to the left in terms of the *mathematical* operations of addition and multiplication. In this way new syntactical constructions may be given meanings, for example

$$\text{N-eval}[\![e^2]\!] = \text{N-eval}[\![e]\!] * \text{N-eval}[\![e]\!]$$
$$\text{N-eval}[\![\text{double } e]\!] = 2 * \text{N-eval}[\![e]\!]$$

2.2 Expressions with Variables

The value of expression $x + 2 * y$ clearly depends on the current values of x and y. More generally an expression e can only be evaluated if the values of all variables occurring in e are known. The "current values" of all variables can be represented by a socalled "environment", namely a function env : Variables → Values. If, for example, $\text{env}(x) = 5$ and $\text{env}(y) = 7$ the value of $x + 2 * y$ *in environment env* is $5 + 2 * 7 = 19$.

An appropriate semantic function is Eval, with functionality

Eval : Expression → (Environment → N) where
env : Environment = Variables → N

There are two equivalent ways to evaluate expressions: *substitution*, in which N-eval of Section 2.1 is applied to the result of substituting values for variables, and *direct* evaluation, in which Eval is recursively defined using argument env.

Evaluation by Substitution

$\text{Eval}\llbracket e\rrbracket\ \text{env} = \text{N-eval}\llbracket e'\rrbracket$ where e' is the result of replacing every variable x in e by its value $\text{env}\llbracket x\rrbracket$

Direct Evaluation

$$\begin{aligned}
&\text{Eval}\llbracket d\rrbracket\ \text{env} &&= d && \text{for } d = 0,1,\ldots,9\\
&\text{Eval}\llbracket nd\rrbracket\ \text{env} &&= (10 * \text{Eval}\llbracket n\rrbracket\ \text{env}) + d && \text{for } d = 0,1,\ldots,9 \text{ and number } n\\
&\text{Eval}\llbracket x\rrbracket\ \text{env} &&= \text{env}\llbracket x\rrbracket && \text{for variable } x\\
&\text{Eval}\llbracket e_1 + e_2\rrbracket\ \text{env} &&= (\text{Eval}\llbracket e_1\rrbracket\ \text{env}) + (\text{Eval}\llbracket e_2\rrbracket\ \text{env}) &&\\
&\text{Eval}\llbracket e_1 * e_2\rrbracket\ \text{env} &&= (\text{Eval}\llbracket e_1\rrbracket\ \text{env}) * (\text{Eval}\llbracket e_2\rrbracket\ \text{env}) &&
\end{aligned}$$

If for example $\text{env}\llbracket x\rrbracket=5$ and $\text{env}\llbracket y\rrbracket=7$ then

$$\begin{aligned}
\text{Eval}\llbracket x + 2 * y\rrbracket\ \text{env} &= \text{Eval}\llbracket x\rrbracket\ \text{env} + \text{Eval}\llbracket 2 * y\rrbracket\ \text{env}\\
&= \text{env}\llbracket x\rrbracket + (\text{Eval}\llbracket 2\rrbracket\ \text{env}) * (\text{Eval}\llbracket y\rrbracket\ \text{env})\\
&= 5 + 2 * \text{env}\llbracket y\rrbracket\ = 5 + 2 * 7 = 19
\end{aligned}$$

Semantic rules of this sort thus define the meanings of expressions and programs in one language by mapping them into expressions of another "semantic" language. We shall see, for example, that it is easy to use LISP as a semantic language for the purpose of implementing other languages or even to extend LISPs own facilities.

II PROGRAMMING LANGUAGES

3. THREE MINILANGUAGES

A large part of DAT2 concerns practical aspects of the semantics of programming languages, including

- Understanding the finer points of fundamental concepts such as dynamic data structures, recursive procedures, name binding and parameter transmission.
- Methods for implementing these concepts on traditional hardware.
- Construction of interpreters and compilers.
- Construction of programs from specifications.

In order to describe the essentials of several alternative semantic concepts and implementation techniques, we now describe three very simple languages. The first, the *lambda calculus,* is both the simplest and the oldest, dating back to 1936. A program is simply an expression; computation proceeds stepwise by rewriting this expression, eventually transforming it into the final answer. In spite of its simplicity, this archetypical "applicative language" can compute all computable functions and clearly illustrates several fundamental concepts including call-by-value, call-by-name, and the possibility of parallel computation.

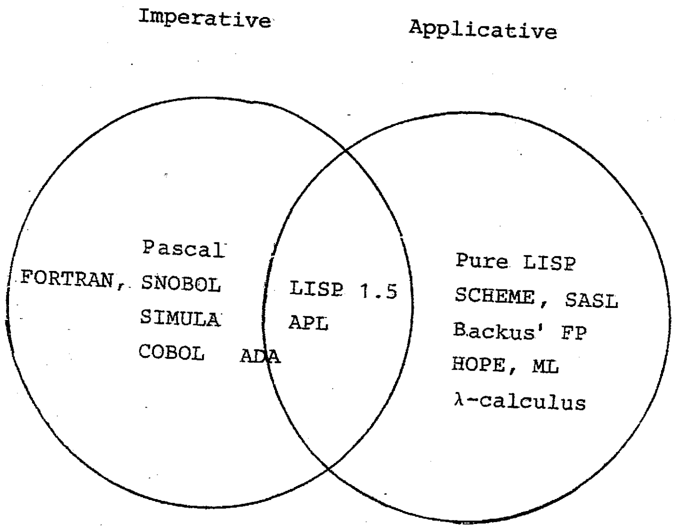
The second language is that of *recursive systems of equations,* also an applicative, expression-oriented language. Unlike the λ-calculus, the concepts of "program" and "computed value" are separated. The programming language LISP originated from lambda calculus, and in use resembles a combination of it with recursion equations.

The third is a more traditional "imperative" language of flowcharts, with assignment statements and goto's. It is introduced for comparison with the first two, and to illustrate methods for translating applicative programs into imperative form and vice versa.

3.1 Applicative and Imperative

Programming languages can roughly be classified into two groups:

The *applicative* (or functional) and the *imperative* (or state--transition) languages. The diagram below shows the placement of several familiar programming languages in this dichotomy.



Typical Characteristics of Applicative Languages

1. Expression-oriented: program execution is done by expression evaluation.
2. Weak or no time concept - no single "point of control". Sequentiality comes only from dependency on data values.
3. Much use of recursion. Functions can be used as data objects.
4. Complex data Objects (fx. trees and strings) may be both operands to and results of operations.
5. Suited to parallel execution.
6. Well-suited to formal verification that a program satisfies its specification, i.e. does what it is intended to do.
7. Well-suited to program transformation.

Typical Characteristics of Imperative Languages

1. Storage - or state-oriented (where a "state" maps variables to their current values). Computation is done by updating the state, changing variables' values one at a time.
2. Linear time - at each moment there is one point of control.
3. Much use of iteration (which repeatedly updates the state).
4. Complex data structures are built stepwise, by sequences of operations which modify individual components, i.e. by selective updating.
5. Programs reflect current machine architectures, and are consequently efficient on sequential machines, but are not well suited to parallel execution.

3.2 The Lambda Calculus

The lambda calculus is an extremely simple programming language, but is powerful enough so that any computable function can be computed by a lambda expression [Chu36]. Further, the lambda calculus illustrates in a simple context the fundamental concepts of

- Binding of variables to values.
- Call-by-value and call-by-name.
- Parallel or nondeterministic evaluation.

3.2.1 Syntax

The notation is simply a formalization of the lambda and function notation described earlier, and is given by the following Backus-Naur Form productions. For the sake of generality, the set of possible constant values <Const> and operations on them <op> have not been specified, since various dialects of the lambda calculus will have their own data domains. In our examples <Const> will be the natural numbers, and <op> can be any of the usual operations on them, for example +, -, *, or / (written in infix, prefix, or suffix in the customary way). By contrast, in LISP constants are binary trees with numbers or atoms as leaves, and operations are included which build and decompose trees.

```
<Lam> ::= <Const>                                Constants
        | <var>                                  Variables
        | <Lam0>(<Lam1>,...,<Lamn>)              Function application
        | λ<Var1>....<Varn> . <Lam>              Abstraction
        | if <Lam> then <Lam>                    Conditional
                   else <Lam>
        | <op>                                   Operation on
                                                    constant values
```

Lambda Calculus Syntax

Examples

x, 5, +	(note that + by itself is a lambda expression)
+(5,6)	(or 5 + 6 in ordinary notation)
λx . x + 1	
(λy . (λx . x / y)6)2	
<u>if</u> x = y <u>then</u> x + y <u>else</u> (λy . y * y * y)(3 + 4)	

As in the earlier discussion we will allow parentheses to be dropped when the meaning is clear, and write for example fx for f(x) and f x y for (f(x))(y) . In expression ---λx . --- the λx is understood to apply to the *longest* complete expression to the right of the "." so for example

λx . x + y	is	equivalent to λx . (x + y)
λx . (λy . x + y) + 5	is	" " λx . ((λy . (x + y)) + 5)

Scopes of Names

An abstraction λx . M is by itself a legal "program" in the lambda calculus language. Intuitively λx . M corresponds to an unnamed PASCAL *function* declaration with parameter x

$$\lambda x\ .\ \text{expression} \equiv \left\{\begin{array}{ll}\underline{\text{function}} & \text{noname(x) : valuetype;}\\ \underline{\text{begin}} & \text{noname := expression}\\ \underline{\text{end}} & \end{array}\right.$$

Consequently a function application such as (λx . x * 2 + 1)5 is computed by first binding x to 5 and then computing 5 * 2 + 1 = 11. (Evaluation rules will be formally defined below).

The *scope* of x in λx . M consists

- All subexpressions of M
- *Except* for any subexpression of M which is contained in an expression λx . N

Examples

$$\lambda x\ .\ \overbrace{x + 5}^{\text{scope of } x} \qquad \lambda x\ .\ \overbrace{x + (\lambda y\ .\ \overbrace{y * x}^{\text{scope of } y})7}^{\text{scope of } x} \qquad \overset{\text{scope of the outer } x}{\lambda x\ .\ \overbrace{x + (\lambda x\ .}^{}\ \overbrace{x * y}^{\text{scope of inner } x})\ \overbrace{* x + z}^{}}$$

An occurrence of a variable x in an expression M is said to be *bound* if it is contained in an expression of the form λx . N ; otherwise the occurrence is said to be *free*. Note that x may have both free *and* bound occurrences in M.

Examples

λx . x + 5	x is bound
λx . x + (λy . y * x)7	y and both x occurrences are bound.
λx . x + (λx . x * y) * x + z	all x's are bound, but y and z are free.
x + (λx . x * y) * x + z	the x in (λx . x * y) is bound, while y, z, and the other x's are free.

Clearly the scope rules of the lambda calculus are essentially the same as those of PASCAL, and free variables would be called "undeclared" in PASCAL. Finally, a *closed* lambda expression is one which contains no free variables (corresponding to a PASCAL program in which every variable is declared). The first two of the examples are closed, while the last two are not.

3.2.2 Computation in the Lambda Calculus

In traditional programming languages computation is done by a series of local changes to a runtime state, directed by the current controlpoint in the program being executed. Modelling computation thus requires three components:

1. The program being executed.
2. The current runtime state (current values of variables).
3. The current point of control in the program.

In contrast, computation with lambda expressions is much simpler: the "program" is a lambda expression, which is itself transformed into the "final answer" by a series of *reductions*. Consequently components 1 and 2 are merged. Surprisingly, we will see that component 3 is not needed at all.

The reduction rules are given in the table below. The first two rules use a special notation for substitution:

[N/x]M = the result of substituting N for all free occurrences of x in M .

Example Reductions

$+(5,6)$	⇒	11	by δ reduction
$5 + 6 = 11$	⇒	true	by two δ reductions
$\lambda a . a + 5$	⇒	$\alpha b . b + 5$	by α reduction
$(\lambda a . a + 5)6$	⇒	$6 + 5$	by β reduction
$(\lambda a . a + 5)6$	⇒	11	by repeated reductions
$(\lambda x . (\lambda y . x + y)6)7$	⇒	$(\lambda x\ \ x + 6)7$	by reduction in context
if true then 13 else 14	⇒	13	by conditional reduction
if 5 + 6 = 11 then 13 else 14	⇒	13	by conditional and δ reductions
$(\lambda x . ((\lambda x . x + x)5) + x + (\lambda x . x * x)3)4$	⇒	23	by repeated reductions

LAMBDA CALCULUS REDUCTION RULES

α reduction, or renaming

$$\lambda x . M \Rightarrow \lambda y . [y/x]M$$

β reduction, or parameter binding

$$(\lambda x . M)(N) \Rightarrow [N/x]M$$

δ reduction, or constant calculation

$op(a_1, \ldots, a_n) \Rightarrow b$ where b is the result of applying op to constants $a_1, \ldots, a_n$ (note: $a_1, \ldots, a_n$ *must* be constants)

Conditional Reduction

if true then M else N ⇒ M

if false then M else N ⇒ N

Reduction in Context

$\ldots M \ldots \Rightarrow \ldots N \ldots$ provided $M \Rightarrow N$

Repeated Reduction

Suppose $M_1 \Rightarrow M_2$, $M_2 \Rightarrow M_3, \ldots,$ and $M_{n-1} \Rightarrow M_n$, where $n \geq 1$. Then

$$M_1 \Rightarrow M_n$$

Explanation of the Reduction Rules

1. α (alpha) reduction allows the current program (i.e. λ-expression) to be modified by renaming a bound variable. For example $\lambda a . a + 5$ can be transformed to $\lambda b . b + 5$.
2. All computation with atomic values is done by δ (delta) reduction. For example +(5,6) can be transformed to 11 by one δ reduction.
3. A λ-expression of form $(\lambda x . M)N$ specifies the application of a function f (denoted by $\lambda x . M$) to a single argument denoted by N. This is done by β (beta) reduction: the value f(N) is obtained by substituting N for each free occurrence of the formal parameter x in M, and evaluating the result. For example, $(\lambda x . x + x * x)5$ is transformed to $5 + 5 * 5$ by a β reduction, and then further to 30 by two δ reductions.
4. A conditional <u>if</u> B <u>then</u> M <u>else</u> N may be evaluated as follows:
   a) evaluate B to an atomic value b
   b) if b = true then evaluate M, else if b = false then evaluate N, else stop.
5. An expression occurring inside another λ-expression may be reduced without changing the rest of the expression in which it appears.

A computation may consist of a sequence of reductions applied to an initial λ-expression. If the expression is a constant then no further reductions may be performed.

A Remark: On Metalanguages

We have used a set of reduction rules expressed in one language, English, to define the computations by programs in another language, the lambda calculus. In other words we are using English as a <u>metalanguage</u> to define computation rules for lambda calculus.

We have tried to be precise and unambiguous in our use of English, but it is of course not itself a formally defined metalanguage. A program execution algorithm for language L which is written in another programming language M is called an *interpreter* for L, symbolized by $\boxed{\begin{matrix}L\\M\end{matrix}}$. The subject of denotational semantics ([Gor79], [Sto77]) is concerned with the development of mathematically well-defined metalanguages and their applications to programming language definition.

We will see later that a simple language called LISP0 can serve as its own metalanguage.

A Technical Restriction on Substitution

Restriction:
The substitution [N/x]M may only be computed if no free variables in N become bound as a result of the substitution.

The restriction on [N/x]M prevents free variables of N being "captured" by lambdas inside M. For example consider

$$(\lambda x . \overbrace{2 + (\ y . x + y)5)}^{M}\ \overbrace{(y + 1)}^{N}$$

Clearly the y's in M and N are distinct, so it would be illogical to apply β reduction blindly and get $2 + (\lambda y . y + 1 + y)5$. However $(\lambda x . M)N$ *can* be reduced with the aid of an α reduction as follows:

$$\begin{aligned}(\lambda x . M)N &\Rightarrow (\lambda x . 2 + (\lambda z . x + z)5(y + 1)\\ &\Rightarrow 2 + (\lambda z . y + 1 + z)5\end{aligned}$$

Fortunately this slightly complicated restriction can be safely ignored in practice provided a reduction sequence starts with a closed lambda expression and uses call-by-value or call-by-name evaluation.

3.2.3 Order of Evaluation

Clearly reductions may be done in many different orders, for example

<u>call-by-value</u>: evaluate arguments first

$$\begin{array}{ll}(\lambda x . x + 5)((\lambda y . y * y)3) & \Rightarrow\\ (\lambda x . x + 5)(3 * 3) & \Rightarrow\\ (\lambda x . x + 5)9 & \Rightarrow\\ \quad 9 + 5 & \Rightarrow\ 14\end{array}$$

<u>call-by-name</u>: evaluate arguments last

$$\begin{array}{ll}(\lambda x . x + 5)((\lambda y . y * y)3) & \Rightarrow\\ (\lambda y . y * y)3 + 5 & \Rightarrow\\ \quad 3 * 3 \quad + 5 & \Rightarrow\\ \quad\quad 9 \quad + 5 & \Rightarrow\ 14\end{array}$$

An obvious question is: can the result of two different reduction sequences produce different constant values? The answer is "no" as a consequence of the well-known.

Church-Rosser Theorem [HLS72]. Let M, N, P be lambda expressions such that M ⇒ N and M ⇒ P. Then there exists an expression Q such that N ⇒ Q and P ⇒ Q.

$$\begin{tikzcd}
& M \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
N \arrow[dr, Rightarrow, dashed] & & P \arrow[dl, Rightarrow, dashed] \\
& Q &
\end{tikzcd}$$

Consequently if M can be reduced to constants c and d by two reduction sequences, the theorem implies c ⇒ d and d ⇒ c, so c and d must be equal.

This well-known "Church-Rosser Property" is very interesting because it opens the possibility for *parallel* evaluation, since evaluation of a large λ-expression $M = \ldots M_1 \ldots M_2 \ldots M_n \ldots$ can be done by reducing components $M_1, M_2, \ldots, M_n$ simultaneously. By the theorem, the result will be the same regardless of the order of processing or the number of processors. This idea has been used by C. Paulsen to implement a LISP-like language on a 9-processor parallel machine in Århus [Pau83], and is the subject of much current research in several countries.

Traditional imperative languages such as Pascal, on the other hand, do not easily admit parallel implementation, because the time concept is so strongly built into their semantics.

3.2.4 Call-by-Name, Call-by-Value, Call-by-Need

These well-known evaluation strategies differ in the time that β-reduction is applied. Given expression (λx . M)N, with call-by-value the argument N will be evaluated *before* it is substituted into M, whereas with call-by-name N is substituted without first evaluating it. More precise definitions can be given by specifying two evaluation functions:

CBV, CBN: Lambda-expressions $\overset{P}{\to}$ Lambda-expressions

Recursive evaluation algorithms for CBV and CBN are found below. We have described them using an informal metalanguage containing a structure-oriented *case* statement like that in [Hoa75]. The statement's form is

```
case form of M of
pattern$_1$: expression$_1$
    ...
pattern$_n$: expression$_n$
end
```

This has the following effect: first, M's value is matched against the various patterns. If the first that matches is pattern$_i$, then any variables appearing in pattern$_i$ are bound to the corresponding part of M's value, and expression$_i$ is then evaluated. As a simpler example of the notation, the simple expression evaluation algorithm of Section 2.1 can be expressed as

N-Eval: Numeric expressions → N

```
N-Eval(e) =
  case form of e of
  $e_1 + e_2$  : N-eval($e_1$) + N-eval($e_2$)
  $e_1 * e_2$  : N-eval($e_1$) * N-eval($e_2$)
   nd      : 10 * N - eval(n) + d
    d      : d
  end
```

```
CBN, CBV: Lambda-expressions $\overset{P}{\to}$ Lambda-expressions

CBN(M) =
case form of M of
        constant : M
        variable : M
          λx . N : M
  op($N_1$,...,$N_n$) : value of op(CBN($N_1$),...,CBN($N_n$))

  "if N then P else Q": if CBN(N) then CBN(P) else CBN(Q)

    operator(operand) : if CBN(operator) has the form λx . N
                        then CBN([operand/x]N)
                        else error
end

CBV(M) =
case form of M of
  ... same pattern as for CBN, except:
    operator(operand) : if CBV(operator) has the form λx . N
                        then CBV([CBV(operand)/x]N)
end
```

Remarks

1. The following all denote λ-expressions: M, N, P, Q, N_i, operator, operand.
2. "op" denotes a λ-calculus constant operator, for example "+".
3. The pattern "if N then P else Q" describes a λ-expression in the interpreted language. We have used quotes to avoid confusion with the use of if in the interpreting or meta-language.

Some Example Evaluations:

$$\begin{aligned}
\mathrm{CBV}(5) &= 5\\
\mathrm{CBV}(\lambda x . x + 1) &= \lambda x . x + 1\\
\mathrm{CBV}(5 + 1) &= 6\\
\mathrm{CBV}((\lambda x . x + 1)5) &= \mathrm{CBV}([\mathrm{CBV}(5)/x]x + 1)\\
&= \mathrm{CBV}([5/x]x + 1)\\
&= \mathrm{CBV}(5 + 1)\\
&= 6
\end{aligned}$$

$$\begin{aligned}
\mathrm{CBV}((\lambda y . yy)(\lambda y . yy)) &= \mathrm{CBV}([\mathrm{CBV}(\lambda y . yy)/y]yy)\\
&= \mathrm{CBV}((\lambda y . yy)(\lambda y . yy))\\
&= \dots \quad \text{undefined}
\end{aligned}$$

In principle, call-by-name is to be preferred due to the following

Completeness Property if M can be reduced to a constant c, then CBN(M) = c.

In other words if there is *any* way to reduce M to a constant, then call-by-name will do it. The same does *not* hold for call-by-value as shown by the example:

$$M = \overbrace{(\lambda x . 1 + 2)}^{N}\overbrace{(\lambda y . yy)(\lambda y . yy))}^{P}$$

Clearly $M \Rightarrow 1 + 2 \Rightarrow 3$ (since no x's appear in 1 + 2), and CBN(M) = 3 is easily seen. However, CBV(M) requires evaluation of CBV(P), which is undefined since the only possible reduction sequence is $P \Rightarrow P \Rightarrow P \Rightarrow \dots$.

In practice, however (for example in LISP) call-by-value is used because it avoids the need for repeated argument evaluation which occurs with call by name (for an example consider evaluation of $(\lambda x . x + x * x)(3 + 4 * 5)$).

A third alternative is *call-by-need*, which resembles call-by-name except that after the first evaluation of the argument, the argument's value is physically substituted for the argument, thus avoiding re-evaluation. (In practice execution is done using expressions represented as directed graphs, and substitution is done by changing the information present at a graph node). In summary we have

Method	Number of Argument Evaluations
call-by-value	1
call-by-name	any number $n \geq 0$
call-by-need	0 or 1

3.3 Recursive Systems of Equations

Recursively defined functions and procedures are used in a wide variety of computer science applications including

- tree and data structure manipulation
- formula manipulation and symbolic computation
- artificial intelligence
- computation with very large numbers
- algorithms on directed and undirected graphs
- compilers and interpreters

Further, the widely used paradigm for program design and construction called "divide-and-conquer" leads naturally to recursive programs.

Recursive algorithms can (surprisingly) be expressed in the lambda calculus with the aid of the so-called Y combinator

$$Y = \lambda h . (\lambda x . h(xx))(\lambda x . h(xx))$$

so for example the factorial function can be defined by

$$fac = Y(\lambda f . \lambda n . \underline{if}\ n = 0\ \underline{then}\ 1\ \underline{else}\ n * f(n-1))$$

but the resulting programs are long and not easily understood.

Thus in order to explain simply how recursive programs are executed, and how they can be implemented on traditional computers, we introduce a second minilanguage whose programs consist of sets of recursion equations.

LISP [McC60] is essentially a language of this type, as is the language called HOPE recently developed at Edinburgh (HOPE also uses Hoare's recursive data structures [HOPE], [Hoa75].) For the sake of simplicity we consider only programs which compute and operate on atomic values. HOPE and LISP allow functions and data structures as both arguments to and results of functions.

Like the lambda calculus, systems of recursion equations form an applicative language with the attendant properties described earlier.

3.3.1 Syntax

```
<program>    ::= <equation>...<equation>
<equation>   ::= <leftside> = <expression>
<leftside>   ::= <identifier>(<identifier>,...,<identifier>)
<expression> ::= <constant> | <variable>
               | <op>(<expression>,...,<expression>)
               | <functionname>(<expression>,...,<expression>)
               | if <expression> then <expression>
                                 else <expression>
```

As in the lambda calculus we leave unspecified the exact constants and operators <op> to be used, and allow expressions to be written using customary precedence and associativity rules, infix notation, etc. A very simple example is the familiar factorial function

$$fac(n) = \underline{if}\ n = 0\ \underline{then}\ 1\ \underline{else}\ n * fac(n-1)$$

Following are a simple way and a more efficient way to compute the exponential function $exp(x,y) = x^y$

$$exp1(x,y) = \underline{if}\ y = 0\ \underline{then}\ 1\ \underline{else}\ x * exp1(x, y-1)$$

$$\begin{array}{lll} exp2(x,y) = \underline{if}\ y = 0\ \underline{then}\ 1 & & \\ \qquad \underline{else}\ \underline{if}\ even(y) & \underline{then}\ exp2(x, y/2)^2 & \\ & \underline{else}\ x * exp2(x, (y-1)/2)^2 & \end{array}$$

For a simple example of a system of two equations, the following computes $f_1(x) = true$ if x's binary representation contains an even number of 1's, else $f_1(x) = false$.

```
f₁(x) = if x = 0 then true
        else if even(x) then f₁(x/2)
                        else f₂((x - 1)/2)

f₂(x) = if x = 0 then false
        else if even(x) then f₂(x/2)
                        else f₁((x - 1)/2)
```

Scopes of Names

In an equation

$f(x_1,\dots,x_n)$ = <expression>

the scope of $x_1,\dots,x_n$ is the <expression>, whose variables must all lie in $\{x_1,\dots,x_n\}$. Consequently no "cross-references" between equations are allowed. For obvious reasons it is further required that $x_1,\dots,x_n$ all be different identifiers.

3.3.2 Semantics: Computation with Equation Systems

Suppose we are given a program

```
prog     f₁(x₁,...,xₘ) = exp₁
         f₂(  ...    ) = exp₂
          ...             ...
         fₙ(  ...    ) = expₙ
```

Conceptually each function name f_i ($1 \le i \le n$) denotes a mathematical function $\varphi_i: A_1 \times \dots \times A_p \to B$ where A_j is the type of the j-th argument of f_i and B is the type of exp_i . The whole program defines the function φ_1. In the previous example

$\varphi_1(x)$ = true if x has an even number of 1's, else false

$\varphi_2(x)$ = false if x has an even number of 1's, else true

Clearly in order to compute, for example, $f_5(5,3,8)$ we must evaluate exp_5 with its variables bound to 5,3, and 8,

respectively. This suggests use of an evaluation algorithm like the environment based function Eval of Section 2.2. Following is an operational semantics for equation systems based on this idea. Note that an extra variable, prog, is used. The reason is that if a function call $f_j(\dots)$ occurs in some exp_i, then the definition of f_j must be found in order to evaluate $f_j(\dots)$. Note that we use the notation for updating described in Section 1.3, for example $f[x \mapsto 1, y \mapsto 2]$.

An operational Semantics for Equation Systems

$$\begin{aligned}
&\text{env: Environment} = \text{Variables} \to \text{Value}\\
&\quad\text{Interpret: Program} \times \text{Input} \overset{P}{\to} \text{Value}\\
&\quad\text{Eval: Expression} \times \text{Environment} \times \text{Program} \overset{P}{\to} \text{Value}
\end{aligned}$$

```
Interpret(prog,a1,...,an) = Eval(exp1,env,prog)
        where the first equation in prog is f1(x1,...,xn) = exp1 and
        env = [x1↦a1,...,xn↦an]
```

```
Eval(exp,env,prog)  =
case form of exp of
  constant           : exp
  variable(x)        : env(x)
  op(exp1,...,expp)  : op(Eval(exp1,env,prog),...,Eval(expn,env,prog))

  "if exp1
   then exp2         : if Eval(exp1,env,prog)
   else exp3"          then Eval(exp2,env,prog) else Eval(exp3,env,prog)

   f(exp1,...,expp)  : Eval(exp',env',prog)    where
                       "... f(y1,...,yp) = exp'..." = prog   and
                       env' = [y1↦Eval(exp1,env,prog),...,
                               yp↦Eval(expp,env,prog)]
end
```

Remarks

1. Eval clearly follows the pattern of Section 2.2, although we have used a syntax more like that of Hoare.
2. The semantics is more complex than that of the lambda calculus due to the use of environments and the need for "prog" as a parameter. Note that this is a *fixed-program* semantics (i.e. prog remains unchanged) in contrast to the reduction semantics of the lambda calculus which repeatedly rewrites prog.
3. The environments env and env' are functions defined by update notation. Note that according to the definition of env', the arguments of a function call $f(exp_1,...,exp_n)$ are evaluated *before* the function is applied. Hence this is a *call-by-value* semantics.
4. Call-by-name semantics can also be defined, by letting the environment bind variables to unevaluated expressions, and evaluating these expressions only when the variables must be evaluated.

   Technicality: Actually, it is not enough to bind variables to expressions alone, since the values of the variables appearing in the expressions must also be recorded. The standard solution is to bind variables to *closures* of the form (expression, call-time-environment). With this modification we obtain

   ```
   Environment = Variables → Values
   Values      = Numbers + closures
   Closures    = Expression × Environment.
   ```
5. We have not allowed functions as arguments to or results of functions, but this is easily incorporated using the mechanism of closures.
6. The semantics just given could be directly programmed in LISP, and in fact resembles the LISP definition of a LISP evaluator which is found in [McC62].

3.4 The Third Minilanguage: Flow Charts

Flow charts form a simple imperative language, much closer to traditional machine architectures than the applicative languages just discussed. Their essential characteristic is that computation proceeds sequentially, by execution of a (usually long) sequence of commands each of which updates some component of an implicit program state. The exact form of the state varies widely from computer to computer, so for the sake of generality we will devise a scheme for flowchart semantics which can be parametrized by different choices of commands and states. The state usually consists of the values of certain registers (accumulators, index registers, ...) together with a store or memory which maps storage locations to their current contents.

We will first give a syntax and semantics which is common to all flow charts, and then specify the detailed states and command sets for two natural flow chart languages.

3.4.1 Syntax

BASIC COMMON SYNTAX

```
<program>   ::= read <variable₁>,...,<variable_m>;
                <commands>
<commands>  ::= 1: <command₁> 2: <command₂>... n: <command_n>
<command>   ::= goto <label>
              | if <test> goto <label> else <label>
              | <halt>
              | <other command>
<label>     ::= 1 | 2 | ... | n
```

Since flow chart programs may be executed with various forms of data (lists, numbers, etc.) we have specified only a minimal syntax which is common to all flow charts, and so have left <test>, <halt>, and <other command> unspecified. If they were chosen to describe numeric computation the following could be a flow chart program:

Example A program to compute the greatest common divisor of natural numbers x and y .

```
    read x,y;
    1: if x = y goto 7 else 2
    2: if x < y goto 5 else 3
    3: x := x - y
    4: goto 1
    5: y := y - x
    6: goto 1
    7: return x
```

This syntax is cumbersome for practical use due to the numerous labels. For the sake of readability in examples and constructions we allow some Pascal-like control flow structures.

EXTENDED COMMON SYNTAX

```
<command>   ::=  <command>; <command>
               | [<command>*]
               | if <test> then <command>
               | if <test> then <command> else <command>
               | while <test> do <command>
               | repeat <command> until <test>
```

It should be clear that a program using these constructions can be rewritten without them. Note: "[" and "]" play the role of begin and end in Pascal.

SPECIALIZATION 1: NUMERIC COMPUTATION

```
<other command> ::= <variable> := <expression>
<expression>    ::= <number>  <variable>
                  | <expression> + <expression>
                  | <expression> - <expression>
                  | ... (other numeric operators)
<test>          ::= <expression><relation><expression>
<relation>      ::= < | ≤ | = | ≠ | ≥ | <
<halt>          ::= return <expression>
```

Example The program above, rewritten:

```
    read x,y;
    while x ≠ y do
    [if x > y then x := x - y else y := y - x];
    return x
```

SYNTAX FOR SPECIALIZATION 2: LIST COMPUTATION

```
<other command>  ::=  <variable> := <expression>
<test>           ::=  <expression>    (test on empty list)
<halt>           ::=  return <expression>
<expression>     ::=  (QUOTE <list>)
                   |  <variable>
                   |  (HEAD  <expression>)
                   |  (TAIL  <expression>)
                   |  (ATOM  <expression>)
                   |  (CONS  <expression> <expression>)
                   |  (EQUAL <expression> <expression>)
<list>           ::=  <atom> | (<list> . <list>)
```

Example A program to find the first element of a list.

```
read x;
while true do
  if (ATOM x) then return (x)
              else x := (HEAD x)
```

3.4.2 Semantics

The following interpretive semantics is parametrized in the same way as the syntax. In order to be complete, one needs definitions of how to perform tests, how to terminate execution, and how to execute commands with syntax <other command>. The kernel of the interpreter is the function Run, such that

Run (s,i,program) = the final output (if any) which results from executing "program", beginning at ~~its~~ i-th command with state s

i,j,k	: IC = N		(Instruction Counter)
s	: State		- varies with the machine type -
	Input = Value*		(sequences of values)
	Output		- varies with the machine type -

Interpret	:	Program × Input	$\overset{P}{\rightarrow}$	Value
Run	:	State × IC × Program	$\overset{P}{\rightarrow}$	Output

Do-command	:	Command × State	$\overset{P}{\rightarrow}$	State
Eval-test	:	Test × State	→	Bool
Init-state	:	Variables × Input	→	State
Final-answer	:	State	→	Output

Interpret(program,$a_1,\dots,a_n$) = Run(s,1,program)
  where "read $x_1,\dots,x_n$; ..." = program
    and s = Init-state($x_1,\dots,x_n,a_1,\dots,a_n$)

```
Run(s,i,program) = let "... i: command ..." = program in
case form of command of
 "goto j"        : Run(s,j,program)
 "if test
  goto j         : Run(s,ℓ,program) where
  else k"          ℓ = if Eval-test(test,s) then j else k
 "halt"          : Final-answer(halt,s)
 "other
    command"     : Run(s',i+1,program) where
                         s' = Do-command(command,s)
end
```

Do-command(c,s)	=	- varies with the machine type -
Init-state($x_1,\dots,x_n,a_1,\dots,a_n$)	=	" " " " "
Eval-test(t,x)	=	" " " " "
Final-answer(h,x)	=	" " " " "

Languages with Assignment

As with the applicative languages, imperative languages operate by binding variables to computed values. Applicative programs bind values implicitly during function calls, while imperative programs bind variables to values by means of the *assignment statement*

variable := expression

which causes the expression to be evaluated with the current bindings, and then causes the left side variable to be bound to the expression's value, overwriting any previous value to which the variable may have been bound.

The essential difference is that in an applicative program the binding is done *once*, at function call, remains in effect during evaluation of the functions body, and is then lost once the functions value has been computed. In contrast an imperative program can (and usually must) bind the same variable to many different values during execution (for example I := I - 1 has no natural applicative counterpart). Computation is typically done by means of loops, while applicative programs typically use recursion. Note: suppose, for example, a call to fac(3) binds n to 3 and then calls fac(2) which binds n to 2, etc. It is not the case that n is being *rebound* as n := n - 1 would do, since this would mean that the value n = 3 would be lost (and it will be needed later to compute fac(3) = 3 * fac(2)). Rather, *new* bindings are created at function calls and abandoned after function values have been computed.

In our formalization the runtime state may thus be represented by a store, a function

Store = Variables → Value

This is mathematically equivalent to an environment as used in the previous section, but is used differently.

INTERPRETER HELP FUNCTIONS FOR SPECIALIZATION 1: NUMERIC COMPUTATION

```
Value = N
State = Variables → Value
                          p
Eval : Expression × State → Value

Init-state(x1,...,xn,a1,...,an) = [x1 ↦ a ,...,xn ↦ an]
Do-command(x := exp,s)   = s[x ↦ Eval(exp,s)]
Eval-test(exp1 < exp2,s) = if Eval(exp1,s) < Eval(exp2,s)
                           then true else false
Final-answer(return exp,s) = Eval(exp,s)

Eval(exp,s)   =
case form of exp of
 "variable(x)          : s(x)
 "constant"            : constant
 "op(exp1,...,expn)"   : Do-op(eval(exp1,s),...,Eval(expn,s))
end
```

Explanation

1. Init-state yields the initial state, in which variables $x_1,\ldots,x_n$ are bound to $a_1,\ldots,a_n$, respectively.
2. Eval(exp,s) returns the value of expression exp, given store s.
3. Do-command(x := exp,s) returns a new state which is identical to s except that x is bound to the value of exp.
4. Do-op(v_1,v_2) returns v_1 op v_2 as value, where op is an operator, e.g. +, -, *, etc. For the sake of definiteness we let

   $0 - n = 0$

   for any n in N .

SPECIALIZATION 2: LIST COMPUTATION

Value = List

State = Variables → Value

Eval : Expression × State → Value

Init-state, Do-command, Eval-test, Eval } as for the numeric specialization

Eval-test(exp,s) = if Eval(exp,s) = () [the empty list] then false else true

AN EXAMPLE: GCD COMPUTATION

With "program" as in the first example from Section 3.4.1 we have an example computation

Interpret(program,5,10)

= Run(s_0,1,program) where s_0 = [x↦5,y↦10]

= Run(s_0,2,program) since Eval(x = y,s_0) = false and Eval(2 ,s_0) = 2

= Run(s_0,5,program) since x < y

= Run(s_1,6,program) where s_1(x) = 5, s_1(y) = 5

= Run(s_1,1,program)

= Run(s_1,7,program) since x = y

= Eval(x,s_1)

= 5

Note that return x causes program termination (since Run is not called again) in contrast to the other commands. State s_0 above could be represented more explicitly by

$$s_0 = \lambda v\,.\;\underline{\text{if}}\; v = x\; \underline{\text{then}}\; 5\; \underline{\text{else}}\; \underline{\text{if}}\; v = y\; \underline{\text{then}}\; 10\; \underline{\text{else}}\; \text{error: unbound variable } v$$

3.5 Pascal and the Three Minilanguages

Pascal is a relatively complex language with features from all three minilanguages which is well suited to efficient execution on contemporary machines. Following are some essential characteristics.

1. Pascal is imperative and includes all the facilities of the flowchart language. The Pascal runtime state's structure is determined by the declarations and parameters appearing in the program (for example var x: integer).

2. Pascal also includes declarations of functions and procedures which may call one another recursively as in the second minilanguage. Note, however, that a "function" may not be functional! since the same arguments to a call may yield different results due to changes in global variables. A Pascal procedure call's effect is to change the state without returning a value.

3. As in the lambda calculus, Pascal allows functions to be arguments to other functions. Functions may not, however, be assigned or be the results returned by functions.

4. Pascal's runtime state consists of file values and a stack of frames (also called activation records), one for each environment as seen in § 3.3.2. At each procedure or function call a new frame is pushed onto the stack, and is popped upon exit.

5. An essential characteristic of Pascal's operational semantics is that each frame may be given a linear structure in storage. Further, every declared variable may be allocated a fixed relative storage location in the frame corresponding to the procedure in which the variable was declared or a parameter. Consequently runtime variable addressing may be done by a "base + offset" method (see Dat2 kursusbog, Søren Olsen's Section), so that variable names need not be present during execution. This results in significantly less storage and time for variable access than in LISP, whose semantics requires that variables' names appear in storage as well as their values.

III COMPILATION AND INTERPRETATION

4. TRANSLATIONS BETWEEN THE MINILANGUAGES

In view of our discussion of applicative versus imperative languages it is natural to ask: which is more powerful? In this section we will show that in fact each can simulate the other, so there is no difference in principle. In practice it seems easier to simulate imperative programs applicatively than *vice versa*, and applicative programs do seem to have a certain semantic elegance not shared by imperative programs.

A beneficial side effect of the comparison is that the method introduced in Section 4.3 is essentially the traditional way to implement block structured languages, including the Pascal storage scheme to be discussed later in the course. We begin with a general description of compilers and interpreters.

4.1 Compilers and Interpreters

There are two essentially different ways to execute programs written in a language L which is not directly machine-executable: compilation and interpretation.

Interpretation is conceptually the simplest, and the various operational semantics we have given have all been interpreters. An interpreter accepts two inputs: a program ℓ in language L, and the data that program is to be applied to. We use the symbol $\boxed{\begin{matrix}L\\M\end{matrix}}$ to denote the set of all programs *written in* language M, which are correct *interpreters for* language L. Using the definition of a programming language from the start of these notes, we have:

$$m \in \boxed{\begin{matrix}L\\M\end{matrix}} \quad \text{if and only if}$$

$$\text{M-eval}(m)(\ell,x_1,\ldots,x_n) = \text{L-eval}(\ell)(x_1,\ldots,x_n)$$

$$\text{for all L-programs } \ell \text{ and all } (x_1,\ldots,x_n) \in \text{L-input}(\ell)$$

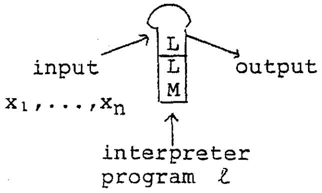
An interpreter typically simulates the behavior of an L program step by step by running an M program, and contains representations of both the L program ℓ and its program state. Consequently interpretation involves a fairly high overhead in space and time, which can often be substantially reduced by first translating ℓ into a target language and then running that target program separately. (Of course to be worth while in practice the total cost of translating and running the target program has to be less than that of running the interpreter).

The symbol $\begin{array}{c}\boxed{L\to T}\\ \boxed{M}\end{array}$ will be used to denote the set of all program written in language M, which are correct translators from source language L to target language T. More precisely,

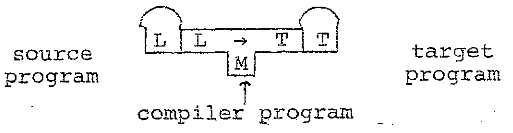
> $m \in \begin{array}{c}\boxed{L\to T}\\ \boxed{M}\end{array}$ if and only if
>
> 1. M-input(m) = L-programs
> 2. M-output(m) = T-programs
>
> $$\text{L-eval}(\ell)(x_1,\ldots,x_n) = \text{T-eval}(\text{M-eval}(m)(\ell))(x_1,\ldots,x_n)$$
>
> for all ℓ in L-programs and all $(x_1,\ldots,x_n) \in$ L-input(ℓ)

4.1.1 Combinations of Compilers and Interpreters

Suppose we are given a collection of L programs, nature unspecified. This set can be denoted by a symbol . If we are given an interpreter $m \in \begin{array}{|c|}\hline L\\ \hline M\\ \hline\end{array}$ and if we have a processor for language M at our disposal (e.g. M could be a machine language) then we can execute any $\ell \in$ with the aid of m. This situation can be described by the diagram

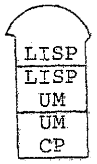


Similarly, if we have a compiler m' from L to T written in M, we can perform translations (again assuming M programs can be executed). This situation can be described by



which clearly specified the translation of any program in into an equivalent T program.

The diagram notation can be applied in case the L programs are themselves compilers or interpreters. For example, the local LISP is processed interpretively by a program written in UNIVAC machine code (call this UM). The machine code is itself interpreted by the UNIVAC central processor (call this CP), so two levels of interpretation are involved, as described by

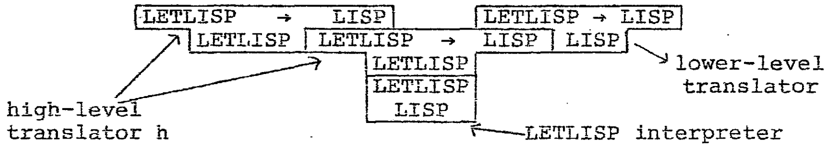


4.1.2 An Example of Bootstrapping

The LETLISP system used in DAT2 is in fact a translator, since the "deflet" command converts LETLISP programs into ordinary LISP. The translator, here called "h", was itself written in LETLISP and so has T diagram:

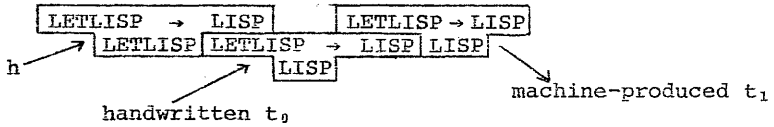
$$h \in \begin{array}{c}\boxed{\text{LETLISP} \to \text{LISP}}\\ \boxed{\text{LETLISP}}\end{array}$$

Such a component is not very useful by itself - one also needs a lower-level LETLISP processor. One way to get this would have been to write a LETLISP interpreter in LISP. With the aid of this a much more efficient compiler could have been obtained by the following (note: we have dropped $\begin{array}{|c|}\hline \text{LISP}\\ \hline \text{UM}\\ \hline\end{array}$ and $\begin{array}{|c|}\hline \text{UM}\\ \hline \text{CP}\\ \hline\end{array}$ for the sake of simplicity.)

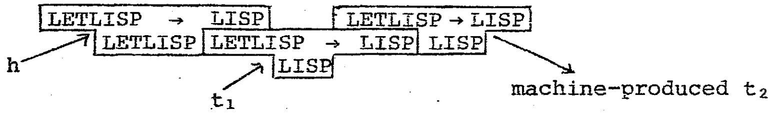


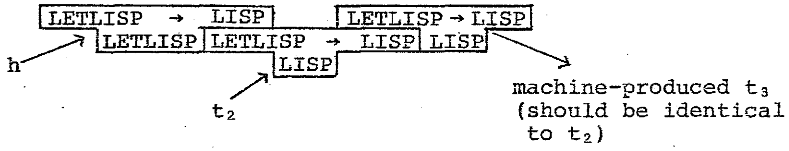
An alternative (and the way it was done) is to write a LETLISP to LISP translator in LISP. This was a straightforward hand translation of the previously written translator in LETLISP. Call this translator t_0 . The following runs were then done:

1. To produce the compiler



2. To test the compiler's correctness: two runs





It's clear that t_1 and t_2 should be behaviorally identical since they are translations of the same source program h, and so their outputs t_2 and t_3 should be textually identical. Note that t_1 may differ textually from t_2, though, since t_1 is output from a handwritten program while t_2 is machine-produced.

Future improvements to LETLISP may now be carried out in LETLISP, and t_0 and t_1 may be discarded. In this way the language's power may be improved stepwise. The usual term for this is bootstrapping, coming from the phrase "to lift one's self by one's own bootstraps".

4.2 Implementing Flow Charts by Recursion Equations

We now see that for every flow chart there is an equation system which computes the same input-output function. In other words, we will show that *goto*, iteration and assignment may be simulated by recursive function calls and binding of formal parameters to actual parameters.

Note: The operational semantics of Section 3.4.2 defines a flowchart *interpreter* in an informal extension of the mini-language of recursion equations, so we have in fact already seen that recursion can simulate iteration. The following describes a *compiler* which provides more efficient and compact simulation, in which the program itself is not present.

The compiling algorithm below may be applied to any flow chart program

$$\boxed{\begin{array}{l} \text{prog} = \underline{\text{read}}\ x_1,\ldots,x_m; \\ \quad 1{:}\ \text{cmd}_1 \quad 2{:}\ \text{cmd}_1 \ \ldots\ n{:}\ \text{cmd}_n \end{array}}$$

whose state is a store of form

$$\text{State} = X_1 \times X_2 \times \ldots \times X_p$$

Here we let $x_1, x_2, \ldots, x_p$ be a list of all variables appearing in "prog" with input variables occurring first (so $p \geq m$). Set X_i represents the set of possible values for variable x_i.

We obtain a recursive "target program" by specializing the operational semantics of Section 4.2 to the fixed, known source program "prog".

The basic idea is to replace the general-purpose function

$$\boxed{\text{Run: State} \times \text{IC} \times \text{Program} \rightarrow \text{Output}}$$

by a collection of n special-purpose functions

$$\boxed{\text{Run}_i\text{: } X_1 \times \ldots \times X_o \rightarrow \text{Output}} \qquad \text{for } 1 = 1,2,\ldots,n$$

Note that both "IC" and "Program" have vanished from Run, and "State" is represented by $X_1 \times \ldots \times X_p$. The following scheme describes translation of "Specialization 1" of Section 3.4.1.

Translation Scheme: Flow Charts to Equation Systems

$$\text{Execute}: X_1 \times \ldots \times X_m \rightarrow \text{Output}$$
$$\text{Run}_i: X_1 \times \ldots \times X_p \rightarrow \text{Output} \qquad \text{for } i = 1,2,\ldots,n$$

$$\text{Execute}(x_1,\ldots,x_m) = \text{Run}_1(x_1,\ldots,x_m,0,0,\ldots,0)$$

If cmd_i is "x_j := exp" and $i < n$ then

$$\text{Run}_i(x_1,\ldots,x_p) = \text{Run}_{i+1}(x_1,\ldots,x_{j-1}, \text{exp},x_{j+1},\ldots,x_p)$$

If cmd_i is "<u>goto</u> j" then

$$\text{Run}_i(x_1,\ldots,x_p) = \text{Run}_j(x_1,\ldots,x_p)$$

If cmd_i is "<u>if</u> exp <u>goto</u> j <u>else</u> k" then

$$\text{Run}_i(x_1,\ldots,x_p) = \underline{\text{if}}\ \text{exp}\ \underline{\text{then}}\ \text{Run}_j(x_1,\ldots,x_p)\ \underline{\text{else}}\ \text{Run}_k(x_1,\ldots,x_p)$$

If cmd_i is "<u>return</u> exp" then

$$\text{Run}_i(x_1,\ldots,x_p) = \text{exp}$$

For example, if the scheme is applied to the GCD algorithm of Section 3.4.1 we obtain the system of equations:

$$\begin{aligned}
\text{Execute}(x,y) &= \text{Run}_1(x,y)\\
\text{Run}_1(x,y) &= \underline{\text{if}}\ x = y\ \underline{\text{then}}\ \text{Run}_7(x,y)\ \underline{\text{else}}\ \text{Run}_2(x,y)\\
\text{Run}_2(x,y) &= \underline{\text{if}}\ x < y\ \underline{\text{then}}\ \text{Run}_5(x,y)\ \underline{\text{else}}\ \text{Run}_3(x,y)\\
\text{Run}_3(x,y) &= \text{Run}_4(x - y,y)\\
\text{Run}_4(x,y) &= \text{Run}_1(x,y)\\
\text{Run}_5(x,y) &= \text{Run}_5(x,y - x)\\
\text{Run}_6(x,y) &= \text{Run}_1(x,y)\\
\text{Run}_7(x,y) &= x
\end{aligned}$$

This can obviously be simplified drastically, for example to:

$$\begin{aligned}
\text{Run}_1(x,y) &= \underline{\text{if}}\ x = y\ \underline{\text{then}}\ x\ \underline{\text{else}}\ \text{Run}_2(x,y)\\
\text{Run}_2(x,y) &= \underline{\text{if}}\ x < y\ \underline{\text{then}}\ \text{Run}_1(x,y - x)\\
&\qquad \underline{\text{else}}\ \text{Run}_1(x - y,x)
\end{aligned}$$

4.3 Implementing Recursion Equations by Flow Charts

We now describe an implementation technique which is the basis for nearly all methods for compiling block-structured languages (Pascal, SIMULA, ADA, etc.) into machine language. Our starting point is a system of recursion equations.

$$\begin{aligned}
f_1(x_1,\ldots,x_m) &= \text{exp}_1\\
f_2(\ -\ \ -\ \ -\ ) &= \text{exp}_2\\
\ldots\\
f_n(\ -\ \ -\ \ -\ ) &= \text{exp}_n
\end{aligned}$$

This will be translated into an equivalent flow chart program (list variant) as described in Section 3.4.2, of form

```
read x₁,...,xₙ j
   1: firstframe(x₁,...,xₙ) ┐   these call f₁ with
   2: pushreturn (4)        }   the input data.
   3: goto 5                ┘
   4: returntop                 write value of f₁.
   5:  - instructions for ┐      Flow chart code
             f₁,f₂,...,fₙ ┘     for the equations.
```

Recursive calls to f_j(exp) will be handled by the use of push-down stacks. For this purpose we use flow charts which manipulate lists as values. The runtime actions "firstframe" etc. will be explained shortly, after we describe the target programs' runtime state.

4.3.1 The Runtime State

```
      State = Valuestack × Bindings × Returnstack
 Valuestack = Value*  -  used in expression evaluation
   Bindings = Frame*  -  holds values of all variables
      Frame = Value*  -  variable values for one equation
Returnstack = Label*  -  stack of "return addresses"
                           used for function calls
```

Valuestack

This is a computation stack used to evaluate the expressions on the equations' right sides. For example the target flow chart code for 3 ÷ 4 * 5 would be

```
push(3); push(4); push(5); do - *; do - + .
```

Each of these is an operation on Value-stack, for example push(3) can be done by:

```
Value-stack := (CONS 3 Valuestack)
```

and do - * can be done by

```
Value-stack := (CONS (TIMES (HEAD Valuestack)
                            (HEAD (TAIL Valuestack)))
                     (TAIL (TAIL Valuestack)))
```

which pops the top two elements off valuestack, and pushes their product on.

Generation of target code is based on the following simple property:

> **Net Effect Property** Suppose the target instructions corresponding to an expression "exp" are executed. If execution terminates, then
> a) Valuestack = v :: oldvs where v is the value of "exp" and oldvs is the value that valuestack had before expression evaluation began.
> b) The runtime state components "bindings" and "returnstack" are unchanged.

In other words the net effect of expression evaluation is to push the expression's value onto valuestack.

For another example, if e_1 then e_2 else e_3 may be coded as

```
1:  - code to evaluate e1 and leave the result on the top
      of Valuestack
2:  falsejump(5)
3:  - code to evaluate e2
4:  goto 6
5: - code to evaluate e3
```

This target code is easily seen to have the net effect property if "falsejump(4)" is realized by

```
2: Temp := (HEAD Valuestack);
   Valuestack := (TAIL Valuestack);
   if Temp = NIL then goto 4
```

(Assuming of course that the code for e_2 and e_3 already have the net effect property).

Variable Binding

During expression evaluation the stack Bindings will always contain as its topmost entry a "frame" containing the values of the arguments to the function whose right side is currently being evaluated. Thus during evaluation of the right side of equation

$$f_j(z_1,\ldots,z_p) = exp_i$$

the value of z_i may be found as the i-th component of the topmost frame in Bindings. The initial call to f is handled the same way - if the input values of $x_1,\ldots,x_n$ are $v_1,\ldots,v_n$, then "first-frame" initializes

$$\text{Bindings} = [v_1,v_2,\ldots,v_n] :: \text{NIL}$$

A recursive call to, for example, f_3 (8,9+10,11) is done by first evaluating the arguments and then pushing a new frame, yielding

$$\text{Bindings} = [8,19,11] :: [v_1,v_2,\ldots,v_n] :: \text{NIL}$$

The right side of the equation defining f_3 is then evaluated.

(Note that f_3's variables are identified by their positions in the topmost frame, and not by their names.) By the "net effect property" the value is to be left on top of Valuestack and the new frame must be popped in order to restore Bindings to its previous form.

Function Calls

"Returnstack" is used to handle control flow during evaluation of a function call $f(exp_1,\ldots,exp_n)$. Such a call is realized as follows:

1. $exp_1,\ldots,exp_n$ are evaluated, so their values $v_1,\ldots,v_n$ appear on top of Valuestack (in reverse order).
2. These are popped and combined into a new frame $[v_1,\ldots,v_n]$ which is pushed onto Bindings.
3. A return address (a label) is pushed onto Returnstack.
4. Control is transferred to the start of the code for equation

   $$f(x_1,\ldots,x_n) \quad = \quad exp$$

5. Once f's value has been computed and lies on top of Valuestack, Bindings and Returnstack are popped and control is transferred to the label just popped from Returnstack.

It is easy to see that this sequence has the "net effect" mentioned before.

4.3.2 An Example Computation

Consider a one-equation system for the factorial function:

$$\boxed{fac(n) \;=\; \underline{if}\; n \le 1 \;\underline{then}\; 1 \;\underline{else}\; n * fac(n-1)}$$

Following is a series of "snapshots" showing the runtime state at various points during the computation of 3! . This computation is in fact the one which will be performed by the target program produced by the compiling algorithm of Section 4.3.5. Note that the neteffect property holds for each expression evaluation, including calls.

Instruction Counter	Valuestack	Bindings	Return-stack	Comments
1	[ ]	[ ]	[ ]	Before execution starts
2	[ ]	[[3]]	[ ]	Bind n to 3
5	[ ]	[[3]]	[4]	Initial call: fac(3)
6 7 8	[3] [1,3] [false]	" " "	" " "	fetch n push 1 test: n ≤ 1 ?
11	[ ]	"	"	False so compute n * fac(n-1)
12 13 14 15	[3] [3,3] [1,3,3] [2,3]	" " " "	" " " "	fetch n fetch n push n compute n-1
16 17	[3] [3]	[[2] [3]] [[2] [3]]	[4] [18,4]	Prepare to call fac(n-1) push new frame push return address
5 6,7,8	[3] [false,3]	" "	" "	Second call: fac(2) Test: n ≤ 1 ?
11,12 13,14,15	[2,3] [1,2,3]	" "	" "	fetch n compute n-1
16,17	[2,3]	[[1],[2],[3]]	[18,18,4]	Push new frame and return for call
5	[2,3]	"	"	Third call: fac(1)
6 7 8	[1,2,3] [1,1,2,3] [true,2,3]	" " "	" " "	fetch n test: n ≤ 1 ?
9 10 19	[2,3] [1,2,3] [1,2,3]	" " [[1],[2],[3]]	" " [18,18,4]	 Push 1 = 1! Prepare to return
20,18	[1,2,3]	[[2],[3]]	[18,4]	Return to 18
19	[2,3]	[[2],]3]]	[18,4]	Push 2 * 1 = 21
20,18	[2,3]	[[3]]	[[4]]	Return to 18
19	[6]	[[3]]	[4]	Push 3 * 2 = 3!
20,4	[6]	[ ]	[ ]	Return to 4
-				Return 6 = 3!

4.3.3 New Commands

As mentioned earlier the flow chart language will be extended by adding several special-purpose commands which aid the iterative execution of the recursive equation system. Their effects will only be described informally in terms of their effects on the runtime state and point of control IC. The runtime state's form will always be

$$s \;=\; (v^*, f^*, \ell^*)$$

where v^* is the value stack, f^* is the frame stack "Bindings", and ℓ^* is the return stack. Control always passes from one command to the following one, unless the contrary is explicitly stated. Recall that a :: list denotes the result of attaching a to the front of list.

Commands for Expression Evaluation

push(v) takes (v^*, f^*, ℓ^*) to $(v::v^*,\ f^*,\ \ell^*)$

fetch(i) takes (v^*, f^*, ℓ^*) to $(v_i::v^*,\ f^*,\ \ell^*)$
    where the top frame in f^* is $[v_1, v_2, \ldots, v_n]$

do-+ takes $(v_1::v_2::v^*,\ f^*,\ \ell^*)$ to $((v_1+v_2)::v^*,\ f^*,\ \ell^*)$

do-* takes (" , f^*, ℓ^*) to $((v_1 * v_2)::v^*,\ f^*,\ \ell^*)$

and similarly for all other operations "do-op" on atomic values.

Commands for Binding and Unbinding

firstframe$(x_1, x_2, \ldots, x_n)$ yields the initial state

$$(\text{NIL},\ [x_1, \ldots, x_n]::\text{NIL},\ \text{NIL})$$

makeframe(n) takes $(v_n:: \ldots ::v_2::v_1::v^*,\ f^*,\ \ell^*)$
    to $(v^*,\ [v_1, \ldots, v_n]::f^*,\ \ell^*)$

popbindings takes $(v^*,\ f::f^*,\ \ell^*)$ to $(v^*,\ f^*,\ \ell^*)$

Commands for Function Call and Return

pushreturn(ℓ) takes $(v^*,\ f^*,\ \ell^*)$ to $(v^*,\ f^*,\ \ell::\ell^*)$

returnjump takes $(v^*,\ f^*,\ \ell::\ell^*)$ to $(v^*,\ f^*,\ \ell^*)$
    and transfers control to the instruction labeled ℓ

Miscellaneous

returntop takes $(v::v^*,\ f^*,\ \ell^*)$ and terminates computation
    producing v as the program's final answer

falsejump(ℓ) takes $(v::v^*,\ f^*,\ \ell^*)$ to $(v^*,\ f^*,\ \ell^*)$, and transfers control to the instruction labeled ℓ if
    v = false, else control goes to the next command.

4.3.4 Construction of Target Code

Target programs are flow charts and so have form

$$\boxed{\text{program} = \underline{\text{read}}\ x_1, \ldots, x_m;\quad 1{:}\ \text{cmd}_1\ \ldots\quad n{:}\ \text{cmd}_n}$$

In order to describe the compiling algorithm concisely we use the notation c @ c' to denote concatenation of labeled command sequences. We assume that in forming c @ c' all labels and label references in c' are appropriately renumbered, and that any references in c to labels *not* in c are replaced by the label of the first instruction in c'. For example

$$[1{:}\ x := x+1] \;\;@\;\; \left[\begin{array}{l} 1{:}\ \ \underline{\text{if}}\ x > 0\ \ \underline{\text{goto}}\ 2 \\ \qquad\qquad\qquad \underline{\text{else}}\ \ 1 \\ 2{:}\ \ y := y-1 \end{array}\right] \;=\; \left[\begin{array}{l} 1{:}\ \ x := x+1 \\ 2{:}\ \ \underline{\text{if}}\ x > 0\ \ \underline{\text{goto}}\ 3 \\ \qquad\qquad\qquad \underline{\text{else}}\ \ 2 \\ 3{:}\ \ y := y-1 \end{array}\right]$$

,

$$\left[\begin{array}{l} 1{:}\ \ \underline{\text{if}}\ x = 0\ \ \underline{\text{goto}}\ 1492 \\ \qquad\qquad\qquad \underline{\text{else}}\ \ 2 \\ 2{:}\ \ y := 0 \end{array}\right] \;\;@\;\; [1{:}\ y := y+1] \;=\; \left[\begin{array}{l} 1.\ \ \underline{\text{if}}\ x = 0\ \ \underline{\text{goto}}\ 3 \\ \qquad\qquad\qquad \underline{\text{else}}\ \ 2 \\ 2{:}\ \ y := 0 \\ 3{:}\ \ y := y+1 \end{array}\right]$$

4.3.5 The Compilation Algorithm

```
   Compile: Recursive Program → Flow Chart Program
      Code: Equation + Expression → Commands
```

```
Compile($eqn_1$ $eqn_2$ ...$eqn_m$)  =
"read $x_1$,...,$x_n$; startcode @ eqncode"
where
      "$f_1$($x_1$,...,$x_n$) = exp" = $eqn_1$
      startcode = [1: firstframe($x_1$,...,$x_n$)  2: pushreturn (4)
                   3: goto 5                     4: returntop]
        eqncode = Code($eqn_1$) @ ... @ Code($eqn_n$)
```

```
Code("f($x_1$,...,$x_n$) = exp")  =
        Code(exp) @ [1: popbindings  2: returnjump]
```

```
Code("constant")         = "push(constant)"
Code(variable $x_i$)      = "fetch(i)"
Code("op($exp_1$,...,$exp_n$)*) =
      Code($exp_1$) @ ... Code($exp_n$) @ do-op
```

```
Code("if $exp_1$ then $exp_2$ else $exp_3$")  =
      Code($exp_1$) @ Branchcode where
Branchcode = [1: falsejump(4)  2: Code$exp_2$)  3: goto 5  4: Code($exp_3$)]
```

```
Code("f($exp_1$,...,$exp_n$)")  =
    Code($exp_1$) @ ... @ Code($exp_n$) @ callcode   where
Callcode = [1: makeframe(n)
            2: pushreturn(4)
            3: goto first label of code for equation
                    "f($x_1$,...,$x_n$) = exp"]
```

Example   Code for the factorial function

$$\boxed{fac(n) = \underline{if}\ n \leq 1\ \underline{then}\ 1\ \underline{else}\ n * fac(n-1)}$$

The target program is

```
read n;

 1: firstframe(n)       Bindings := [n]::NIL = [[n]]
 2: pushreturn(4)       perform initial call
 3: goto 5              goto start of fac code
 4: returntop           final result = Valuestack to p
 5: fetch(1)            start of code for fac
                           push n on Valuestack
 6: push(1)             push constant one
 7: do-≤                evaluate n ≤ 1
 8: falsejump(11)       do "if" test
 9: push(1)             return 1 if true
10: goto 19             exit to caller
11: fetch(1)            push n (from n * fac(n - 1))
12: fetch(1)         ┐
13: push(1)          │  compute n - 1
14: do--             ┘
15: makeframe(1)     ┐
16: pushreturn(18)   │  recursive call to fac
                     │     (result on Valuestack top)
17: goto 5           ┘
18: do-*                compute n * fac(n - 1)
19: popbindings      ┐
                     │  return from fac
20: returnjump       ┘
```

4.3.6 Single-stack Implementation

Notice that the binding and return stacks are pushed and popped synchronously at call time and return time. They both remain constant during a call, and according to the "net effect property" the Valuestack at exit equals its entry value, plus the function value as a new top. The three stacks are thus pushed and popped so consistently that they could be combined into a single stack.

This is in fact the traditional stack implementation of block-structured languages, and naturally yields

State = (Value + Frame)*
Frame = Label × Value*

The new commands of 4.3.3 can easily be implemented on this new data structure. Practical implementation details may be found in the notes on "Lageradministration" in the DAT2 kursusbog I.

IV UNSOLVABILITY OF THE HALTING PROBLEM

5. OVERVIEW: COMPUTABILITY AND UNSOLVABILITY

Following three sections explain three of the most important results from the theory of computability. Their importance lies in the fact that they together argue strongly that the class of "all problems solvable by computer" is in fact well-defined class with sharp boundaries. Thus a computational problem may be classified without ambiguity as "computer solvable" or "unsolvable by computer". Further, we will display several concrete and simple problems which can be precisely defined, but which cannot be solved by any computer whatever, no matter how much time or memory is available, and no matter how rich the computer's instruction set is.

First, some comments on fundamental assumptions: first, we will not concern ourselves at all with computational efficiency in time or space: we are concerned only with the existence of programs which solve a given problem *correctly*, given sufficient resources. (In fact some of the constructions to be shown are enormously inefficient, but this is quite irrelevant to our goals.) Second, we are interested mainly in problems which have an *infinite* number of instances (for example: given arbitrary integers x and y, find the least prime number larger than xy + y.) The reason is that if a problem has only finitely many combinations of input data, a "solution procedure" could take the form of a finite table. This table would contain the problem's answer for each combination of input data, so that any problem instance could be solved by a table lookup. Consequently *all* finite problems are, at least in principle, algorithmically solvable.

The first result is that for a particularly simple programming language, called LISP0, there is a problem which is not solvable by any LISP0 program: the halting problem.

The second result is not a theorem (as was the first) but a collection of arguments supporting the socalled

> Church-Turing Thesis
> Any process which could naturally be called an effective procedure or algorithm can be realised by a Turing machine.

This thesis asserts that one particular computing device, the Turing machine, is at least as powerful as *any other* device (past, present or imaginable) whose computations are "effective", that is algorithmic. Such a thesis cannot be formally proved since it contains an informal phrase: "effective procedure"; its real significance lies in that it asserts that a formal concept - the Turing machine - is an adequate and complete formulation of an informal concept, that of algorithm or effective procedure.

Evidence for the Church-Turing thesis is of two sorts: that a wide variety of computing devices have turned out to be exactly equivalent to the Turing machine in computational power; and that no convincing counterargument has been put forth since the topic first was studied in the early 1930's. Its consequences are wide, as seen in the following simple application.

An application

We have asserted that the LISP0 halting problem (call it "HALT") cannot be solved by any LISP0 program. We will see later that *any* Turing machine can be simulated by a LISP0 program.

Consequently HALT would be LISP0 solvable if it were solvable by a Turing machine, leading to the conclusion that HALT is Turing unsolvable as well as LISP0 unsolvable.

By the Church-Turing thesis, HALT cannot be solved by any effective procedure whatever, since this would imply its Turing solvability and hence its LISP0 solvability.

The third topic we introduce is the idea of *reduction* of one problem to another. By definition problem A can be reduced to problem B (written $A \leq B$) if an algorithm to solve problem A can be constructed, provided one assumes the existence of an algorithm to solve B (that is, there exists a "B subroutine" which may be called as an aid during the solution of A). Using this concept we show that the halting problem's algorithmic unsolvability is not an isolated phenomenon, and that in fact *many* problems concerning program behavior are algorithmically unsolvable.

Reduction is also a fundamental concept for studying the complexity of algorithmically solvable problems.

These notes are organized as follows. In Section 6 the language LISP0 is introduced (essentially an applicative subset of LISP) and its semantics is described informally. Section 7 contains a LISP0 interpretation algorithm Eval written in LISP0. Section 8 shows that the halting problem for LISP0 programs cannot be solved by any LISP0 program.

Chapter V contains arguments that several computing devices are exactly equivalent to LISP0 in computing power. The method is to show that for every program a in programming language A, a program b in language B may be constructed which faithfully simulates a. Representing this relation by an arrow A ← B, the simulations sketched in Chapter V can be diagrammed as follows:

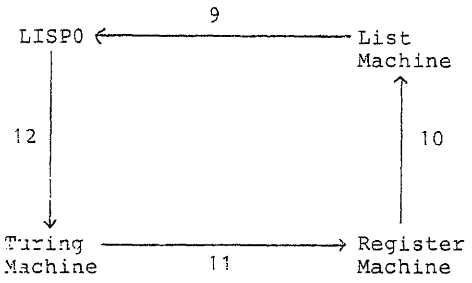


Figure 1. Simulations

Chapter VI deals with reducibility between problems, and shows a variety of problems unsolvable by reducing the halting problem to them. In particular, it is shown that it is impossible by computer to decide whether a context-free grammar is ambiguous.

6. A SIMPLE APPLICATIVE LANGUAGE: LISP0

6.1 Syntax and Informal Semantics

LISP0 is essentially a subset of applicative LISP, restricted for technical convenience to functions of one argument. A LISP0 program manipulates data in the form of a "list", an ordered binary tree with atoms as leaves, where an atom is a nonempty sequence of letters or digits. Lists are written in linear form according to the following syntax

<list> ::= <atom> | (<list> · <list>)

<atom> ::= {<letter> | <digit>}$^{+}$

For brevity in writing we adopt the LISP convention of writing the short form

$(d_1\ d_2\ \cdots\ d_n)$

to represent the following right linear tree ending in the atom "NIL" :

$(d_1 \cdot (d_2 \cdot (d_3 \cdot (\cdots(d_n \cdot \mathrm{NIL})\ \cdots\ ))))$

Lists are constructed with the binary operator CONS, so that CONS applied to lists d_1 and d_2 yields list $(d_1 \cdot d_2)$. Using the short form just mentioned, note that the

CONS of d and $(d_1\ d_2\ \cdots\ d_n)$ equals $(d\ d_1\ d_2\ \cdots\ d_n)$

If list d equals $(d_1 . d_2)$, then the HEAD operator applied to d yields d_1 and the TAIL of d is d_2 , while both are undefined (erroneous) is d is an atom. The operator ATOM applied to a list yields the atom "T" if the list is atomic, and "NIL" otherwise. A LISP0 program, p, consists of a collection of recursively defined functions of form

$$p = \left\{ \begin{array}{l} ((\mathrm{DEFINE}\ f_1(x_1)\ \mathrm{expression}_1) \\ (\mathrm{DEFINE}\ f_2(x_2)\ \mathrm{expression}_2) \\ \quad \cdots \\ (\mathrm{DEFINE}\ f_n(x_n)\ \mathrm{expression}_n)) \end{array} \right.$$

The result of applying program p to data d is the value of expression$_1$, after replacement of all occurrences of variable x_1 in the expression by the value d.

An expression of the form (QUOTE d) is a *constant* expression, with value d. An expression may also be a variable (for example x_1 above), or it may be constructed from other expressions by the operators HEAD, TAIL, ATOM or CONS mentioned above. A conditional expression has the form

(IF exp_1 = exp_2 THEN exp_3 ELSE exp_4)

the values of exp_1 and exp_2 are compared (these may be arbitrary lists). If equal, the conditional expression's value is the value of exp_3, else the value of exp_4. Finally, an expression may take the form (f_i expression), specifying a call to function f_i . The argument is evaluated and then function f_i is applied to the resulting list, so evaluation is "call by value".

Figure 6.1 contains a context free syntax for LISP0 programs and data. Notice that every LISP0 program, expression, etc. is also a list. This makes it possible to regard LISP0 programs *as data objects* which can be processed by other LISP0 programs, a capability which will be important in further developments.

```
<program>    ::=  (<definition>+)

<definition> ::=  (DEFINE <fname> (<vname>) = <expression>)

<expression> ::=  (QUOTE <list>)                    constant
              |   <vname>                           variable reference
              |   (HEAD <expression>)
              |   (TAIL <expression>)
              |   (ATOM <expression>)
              |   (CONS <expression> <expression>)

              |   (IF   <expression>   =   <expression>
                   THEN <expression> ELSE <expression>)
              |   (<fname> <expression>)            function application

<fname>      ::=  <atom>                            function name
<vname>      ::=  <atom>                            variable name

<list>       ::=  <atom> | (<list> . <list>)
<atom>       ::=  {<letter> | <digit>}*
```

Figure 6.1 Contex-Free Syntax of LISP0

Context-Sensitive Syntax In a program

p = ((DEFINE $f_1(x_1)$ = exp_1) ... (DEFINE $f_n(x_n)$ = exp_n))

it is further required that

1. Function names are distinct ($i \neq j$ implies $f_i \neq f_j$) and not contained in the set

   {QUOTE,HEAD,TAIL,ATOM,CONS,IF}

2. The only variable name which may appear in expression exp_i is x_i .

Example

The following program may be applied to a pair gt = (goal.table), where

table = ((key$_1$. value$_1$) (key$_2$. value$_2$) ...(key$_n$. value$_n$))

Its purpose is to compare goal to key$_1$, key$_2$, ... in succession, searching for a match. If the first match is goal = key$_i$, the corresponding value$_i$ is returned, while if no match is found, the value NIL is returned.

```
p = ((DEFINE Search (gt) =
       (IF (TAIL gt) = (QUOTE NIL)
        THEN (QUOTE NIL)
        ELSE
        (IF (HEAD gt) = (HEAD(HEAD(TAIL gt)))
         THEN (TAIL(HEAD(TAIL gt)))
         ELSE (Search (CONS (HEAD gt)
                                 (TAIL(TAIL gt))
    ))))     )        )
```

For instance

```
1. Search of (B . ((A.1)(B.2)(C.3))) =
   Search of (B . ((B.2)(C.3)))      =  2
2. Search of (D . ((A.1)(B.2)(C.3))) =
   Search of (D . ((B.2)(C.3)))      =
   Search of (D . ((C.3)))           =
   Search of (D . ()) = Search of (D.NIL) = NIL
```

6.2 A More Precise Semantics for LISP0

The evaluation of LISP0 expressions and the running of programs can be more completely and precisely defined by specifying program execution and expression evaluation functions:

$$\text{Run} : \langle\text{program}\rangle \times \langle\text{list}\rangle \overset{p}{\rightarrow} \langle\text{list}\rangle$$
$$\text{Eval} : \langle\text{expression}\rangle \times \langle\text{list}\rangle \overset{p}{\rightarrow} \langle\text{list}\rangle$$

The intention is that Eval(exp,d) equal the value of "exp", given that d is the value of the variable possibly occurring in exp. Eval will always be applied to subexpressions occurring in a program

$$p = \begin{cases} \text{((DEFINE } f_1(x_1) = \mathrm{exp}_1\text{)} \\ \quad \ldots \\ \quad \text{(DEFINE } f_n(x_n) = \mathrm{exp}_n\text{))} \end{cases}$$

so exp_i can contain at most one variable x_i . The result of running program p on input d will be Run(p,d) = Eval(exp$_1$,d).

Eval is defined in Figure I.2. Note that according to rule 7, the argument to a function call is evaluated before the function is applied (call by value). By rule 6, either the THEN or ELSE branch of a conditional is evaluated, but not both.

0. Run(((DEFINE $f_1(x_1)$ = exp_1) ...), d) = Eval(exp_1, d)

1. Eval (variable name, d) = d

2. Eval ((QUOTE v) , d) = v

3. Suppose Eval(exp,d) = a nonatomic list (v_1 . v_2).
Then
Eval ((HEAD exp), d) = v_1 and
Eval ((TAIL exp), d) = v_2

4. Eval ((ATOM exp), d) $= \begin{cases} \text{T if Eval (exp,d) is atomic} \\ \text{NIL if Eval (exp,d) is of form } (v_1 . v_2) \end{cases}$

5. If Eval (exp_1,d) = v_1 and Eval (exp_2,d) = v_2, then
Eval ((CONS exp_1 exp_2), d) = (v_1 . v_2)

6. Let *IF* ≡ (IF exp_1 = exp_2 THEN exp_3 ELSE exp_4),
and suppose Eval (exp_1, d) = v_1 and Eval (exp_2, d) = v_2.
then
Eval (*IF*, d) $= \begin{cases} \text{Eval } (exp_3, d) \text{ IF } v_1 = v_2 \\ \text{Eval } (exp_4, d) \text{ IF } v_1 \neq v_2 \end{cases}$

7. Suppose Eval (exp, d) = v, and that program p contains
(DEFINE $f_i(x_i)$ = exp_i). Then
Eval ((f_i exp), d) = Eval (exp_i, v)

Figure 6.2 Semantics of LISP0

6.3 Some Syntactic Sugar

In order to make LISP0 programs more readable, we introduce three forms of "syntactic sugar". These are alternate notations which make programs easier to read and write. Programs

containing these notations may be easily (in fact, mechanically) transformed into pure LISP0 syntax, so the language has not really been changed.

1. '<list> may be written instead of (QUOTE <list>)
2. [exp_1,exp_2,...,exp_n] may be written for (CONS exp_1 (CONS exp_2 .... (CONS exp_n 'NIL) ...)) . This expression is used to construct lists (like the "LIST" function in ordinary LISP). If exp_1,...,exp_n have values d_1,d_2,...,d_n then [exp_1,exp_2,..,exp_n] will have the list (d_1 d_2 ... d_n) as value.
3. Long HEAD-TAIL sequences (as seen on a small scale in "Search") are hard to read and write and thus prone to cause errors. To alleviate this problem we introduce a way to define local abbreviations for HEAD-TAIL sequences, namely the construction (LET pattern = variable IN expression).

The pattern may contain names for various substructures of the value of "variable", and these names may be used within the "expression". Thus, for example

(LET (a.b) = x in (CONS b a))

is precisely equivalent to

(CONS (TAIL x) (HEAD x))

Pattern expressions may also be nested, and the shorthand list notation (d_1 d_2 ... d_n) may also be used as indicated in these examples.

LET Example	Equivalences
(LET (a.b) = x IN ...)	a = (HEAD x), b = (TAIL x)

(LET ((a.b) . (c.d)) = x IN ...)	a = (HEAD(HEAD x)),
	b = (TAIL(HEAD x))
	c = (HEAD(TAIL x))
	d = (TAIL(TAIL x))

(LET (a b c) = x IN ...)	a = (HEAD x)
	b = (HEAD(TAIL x))
	c = (HEAD(TAIL(TAIL x)))

(LET (a(b.c)) = x IN ...)	a = (HEAD x)
	b = (HEAD(HEAD(TAIL x)))
	c = (TAIL(HEAD(TAIL x)))

With this notation, the "Search" example is more readable:

```
((DEFINE Search(gt)
   (LET (goal . table) = gt IN
    (LET ((key.value) . rest) = table IN
     (IF table = 'NIL THEN 'NIL
      ELSE
      (IF goal = key THEN value
                    ELSE (Search (CONS goal rest))
))))))
```

6.4 Computability of Functions and Decidability of Sets

A last example serves also to show that the restriction to single argument functions is not significant in principle, since a multiple-argument function may be expressed in LISP0 by CONS'ing the arguments into a single list. The example is the "append" function; it takes as argument a list $((a_1\ a_2\ \dots\ a_m)(b_1\ b_2\ \dots\ b_n))$ and returns the concatenated list $(a_1\ a_2\ \dots\ a_m\ b_1\ b_2\ \dots\ b_n)$.

```
((DEFINE append(xy) =
   (LET (x y) = xy IN
    (LET (first . rest) = x IN
     (IF x = 'NIL THEN y
      ELSE (CONS first (append [rest,y]))      ))))
```

Definition A partial function f: List $\xrightarrow{p}$ List is LISP0-computable if there exists a LISP0-program p such that for every $x \in$ List, either f(x) is defined and

$$f(x) = \text{Run}(p,x)$$

or both f(x) and Run(p,x) are undefined. A partial multiple-argument function f: $\text{List}^n \xrightarrow{p}$ List is (by definition) LISP - computable if there is a one-argument LISP0-computable function g such that for any $x_1,\dots,x_n \in$ List

$$g.(\ (x_1,\dots,x_n)\ ) = f(x_1,\dots,x_n)$$

(where again the two sides must be both defined and equal,, or both undefined)

□

Many questions concerning solvable and unsolvable problems are most naturally expressed in terms of deciding membership in a given set. For example, primality corresponds to the test:

is $x \in \{2,3,5,7,11,13,17,\dots\}$?

We say such a question is *LISP0-decidable* if it can be answered by a LISP0-program *which always terminates*. More formally, we have the following

Definition A set $A \subseteq$ List is LISP0-decidable if there exists a LISP0-program p such that p terminates for every $x \in$ List, and

$$\text{Run}(p,x) = \begin{cases} \text{T} & \text{if } x \in A \\ \text{NIL} & \text{if } x \notin A \end{cases}$$

7. A SELF-INTERPRETER FOR LISP0

Recall that every LISP0 program is also a LISP0 data object. The rules of Figure 6.2 may be expressed in LISP0, yielding a LISP0 interpreter written in LISP0. The following may be stated more briefly as: Run (as defined in 6.2) is LISP0-computable, or symbolically as:

$$\overline{\text{SI}} \in \boxed{\begin{array}{c}\text{LISP0}\\ \text{LISP0}\end{array}}$$

Theorem. There is a LISP0 program SI such that for any LISP0 program p and list d

1. If Run (p,d) = y, then application of SI to the list (p d) also produces y.
2. If Run (p,d) is undefined, then the result of applying SI to (p d) is also undefined.

□

In case 2, SI may either go into an infinite computation, or attempt to apply HEAD or TAIL to an atomic value. We will not prove the theorem, but just present SI and hope its similarity with Figure 6.2 is evident. If Eval(exp,d) = y according to Figure 6.2, then function "Eval" from Figure 7.1 will also yield y if applied to the list (exp d p). The extra argument p is needed in order to find the definition of f in a function call (f expression), as performed by function lookup.

```
((DEFINE SI(pd) =                          {Run program p on data d}

  (LET (p d)                        =  pd IN
   (LET ((define f1(x1) = exp1))    =  p  IN

    (Eval [exp1, d, p])
)))

(DEFINE Eval(x) =                          {As in Figure I.2}

 (LET (exp d p)   =  x   IN                {p = entire program
  (LET (op e1 e2) =  exp IN                      (used in function call)}

   (IF (ATOM exp) =   'T THEN d
    ELSE
   (IF op = 'QUOTE THEN e1
    ELSE
   (IF op = 'HEAD THEN (HEAD(Eval [e1,d,p]))
    ELSE
   (IF op = 'TAIL THEN (TAIL(Eval [e1,d,p]))
    ELSE
   (IF op = 'ATOM THEN (ATOM(Eval [e1,d,p]))
    ELSE
   (IF op = 'CONS THEN (CONS (Eval [e1,d,p])
                             (Eval [e2,d,p]))
    ELSE
   (IF op = 'IF   THEN

    (LET (if e1 = e2 then e3 else e4) = exp IN

     (IF   (Eval [e1,d,p])   =   (Eval [e2,d,p])
      THEN (Eval [e3,d,p]) ELSE (Eval [e4,d,p])))

    ELSE
   (IF   (Lookup [op,p]) = 'NIL
    THEN ['BAD, 'SYNTAX: , exp]

    ELSE (Eval [(Lookup [op,p]), (Eval [e1,d,p]),p])

 )))))))))))

(DEFINE Lookup (fp) =                      {Find definition of function
                                                 named f in program p}
 (LET (f p) = fp IN
  (LET ((define fi(xi) = expi) . prest) = p IN

   (IF p  = 'NIL THEN 'NIL
    ELSE
    (IF f = fi   THEN expi
     ELSE
     (Lookup [f,prest])
))))))
```

Figure 7.1 LISP0 Self-Interpreter SI

7.1 A Digression: Metacircular Interpreters

A natural question is: why not regard Figure 7.1 as the definition of LISP0 and not bother with Figure 6.2 at all? This method for language definition is called "Metacircular Interpretation", and can be used to clarify many fine points about program behavior. The original LISP report contained two such interpreters, one for basic LISP and one for an extension which was closer to machine implementation.

The problem is that such a definition may actually define nothing at all! For an extreme example, note that SI is built up by use of CONS, QUOTE, IF, etc.. These operators are only defined in terms of each other in Figure 7.1 and no definitions independent of LISP0 are given. Thus if we assumed all these primitive functions returned the value 17 regardless of input, p would also return 17 regardless of input!

Even if we insist that CONS, HEAD, and TAIL behave as expected there are still problems in the use of Figure 7.1 as a language definition. In the discussion before Figure 6.2 it was stated that "IF" could be used to compare arbitrary lists. This is not true in conventional LISP - only atoms may be compared directly and recursion must be used to compare lists. What happens if, for the sake of argument, we assume only atomic values can be compared by the interpretation algorithm SI ? The result is that only atomic values may be compared in the *interpreted* language, and SI is still a correct interpreter, but for a different version of LISP0 than that defined by Figure 6.2.

A more subtle problem is that if SI itself is executed using call-by-name (or with a "lazy CONS"), the same will hold for the language it interprets.

8. UNSOLVABILITY OF THE HALTING PROBLEM

Consider the following two functions, where p is a LISP0 program and d its data.

$$\text{PHALT(p,d)} = \begin{cases} \text{T} & \text{if program p halts on data d} \\ \text{undefined} & \text{if p does not halt on d} \end{cases}$$

$$\text{HALT(p,d)} = \begin{cases} \text{T} & \text{if program p halts on data d} \\ \text{NIL} & \text{if p does not halt on d} \end{cases}$$

In spite of their obvious similarity, PHALT is LISP0 computable while there exists no LISP0 program whatever which correctly computes HALT.

Lemma PHALT is LISP0 computable

Proof Consider the program

```
((DEFINE PHALT(pd) =
    (HEAD (CONS 'T (SI pd))))
  (DEFINE SI(pd)      = ... )
  (DEFINE Eval(x)     = ... )        [From Figure 7.1].
  (DEFINE Lookup(fp)  = ... ))
```

If p halts on d then SI will halt on pd = (p d) and produce some answer y. PHALT then returns (HEAD (CONS 'T y)) = T. If p does not halt on d then (SI pd) doesn't halt either, so PHALT(pd) is undefined. □

Remark We have used "PHALT" both to designate a certain function from lists to lists, and as a name in the LISP0 program just constructed. Strictly speaking this is an abuse of notation since the two meanings are entirely distinct.

Theorem There is no LISP0 program which correctly computes HALT

Proof We will show that no LISP0 program can solve the self-halting problem

$$\text{SH(p)} = \begin{cases} \text{T} & \text{if program p halts on input p} \\ \text{NIL} & \text{if p does not halt on itself as input} \end{cases}$$

This is enough, since SH(p) = HALT(p,p), so SH would be LISP0 computable if HALT were. Assume for the sake of argument that there *does* exist a program

```
((DEFINE SH(p) = ... ) ... )
```

which correctly computes SH. We will show that the assumption leads to an impossible situation, and so must be false (this type of argument is known as *reductio ad absurdum*).

Construct the following LISP0 program.

```
r = ((DEFINE R(p) =
        (IF (SH p) = 'T THEN (R p) ELSE 'T))
      (DEFINE SH(p) = ... ) ... )
```

Program r clearly computes a partial function R: LISP0 programs $\xrightarrow{p}$ List, where

$$\text{R(p)} = \begin{cases} \text{undefined} & \text{if SH(p) = T, i.e. if p halts on itself as input data} \\ \text{T} & \text{if SH(p) = NIL} \end{cases}$$

Question What is the value of R(r)? There are only two possibilities:

1. R(r) is defined. Then the call (SH r) returns T by definition of SH. This causes a recursive call to R(r) and so an infinite computation. Thus R(r) is undefined.
2. R(r) is undefined. Then (SH r) returns NIL, so R(r) = T and so is defined.

Thus R(r) can neither be defined nor undefined. Consequently the unjustified assumption in the argument must be false - so SH is not LISP0 computable. □

V. SUPPORT FOR THE CHURCH-TURING THESIS

It could be argued that the uncomputability of HALT by LISP0 programs simply indicates that LISP0 is too weak - perhaps HALT *could* be computed in a more powerful programming language. However the Church-Turing thesis argues that the Turing machine (and LISP0) are "maximally powerful" - that a Turing machine can compute as much as any other computing device. Consequently the halting problem for LISP0 is in a sense *absolutely uncomputable,* since it cannot be computed by any computing device at all.

The Church-Turing thesis asserts the equivalence between the intuitive concept "effectively computable" and the formal concept "Turing machine computable". The purpose of the next several sections is to give evidence for this thesis, by showing that Turing machines, in spite of their simplicity, can both simulate and be simulated by LISP0 programs. Further, the ideas behind the constructions we give are quite clearly generalizable, and can be used to show the equivalence of other computing devices with the Turing machine.

As mentioned in the introduction we will show equivalence among programs of LISP0, a "List Machine", a "Register Machine", and the Turing machine. The latter three are variants of the imperative flowchart programs of Section 3.4. List machine programs contain assignment statements, and their variables range over LISP0 lists. Register machines are similar except that variables may only take natural numbers as values. Turing machines are even simpler, with a tape for memory and no variables at all.

The three machines have the common program syntax introduced in Section 3.4.1. The reader is advised to review those notations, since the following sections are closely based on them.

9. LIST MACHINES CAN SIMULATE LISP0 PROGRAMS

We now argue that given any LISP0 program, a flow chart program may be found whose data values are lists and whose instructions manipulate lists, and which computes the same input-output function.

Definition A *list machine* program is a flow chart program as described in Section 3.4.1, Specialization 2: List Computation.

Semantics: As described in Section 3.4.2, a list machine state is a function State: Variables → Lists. The function EQUAL returns atom T if its two arguments are equal, else NIL. In a <test>, NIL is consideres to be false and all other values true. □

List Machines Can Simulate LISP0 Programs

Apart from inessential syntactic differences, LISP0 is clearly just a special case of the recursive equation systems discussed in Section 3.3, limited to functions of one variable, data ranging over binary lists, and operators in the set

{HEAD, TAIL, ATOM, CONS, EQUAL}

Consequently the compilation algorithm of Section 4.3 can be used to translate any LISP0 program into an equivalent flow chart program containing "special-purpose commands" whose effects were to update a runtime state (v*, f*, ℓ*). In this state and for LISP0

```
v* = computation stack
   = a stack of list values

f* = frame stack
   = a stack of actual parameter values (= lists)
     of called functions

ℓ* = label stack, for "return addresses"
```

The stack v* can clearly be represented by the list-valued variable v* = (v_1 v_2 ... v_n), where v_1 is the top (so v* = NIL corresponds to an empty stack). Pushing and popping of v* can be done by v* := (CONS newtop, v*) and v* := (TAIL v*), respectively, and v_1 = (HEAD v*). In the same way f* and ℓ* may be regarded as variables in a list machine program.

The "special-purpose commands" of Section 4.3 are specified by their effects on the runtime state, for example

```
push(con)   takes (v*, f*, ℓ*)     to (con::v*, f*, ℓ*)
fetch(1)    takes (v*, f::f*, ℓ*) to (f::v*, f::f*, ℓ*)
do-EQUAL    takes (a::b::v*, f*, ℓ*) to
   (T :: v*, f*, ℓ*)   if  a = b
   (NIL::v*, f*, ℓ*)   if  a ≠ b
```

It is easy to verify that all the commands from Section 4.3 can be implemented by short sequences of list machine code (the only tricky case is "returnjump"). For the examples above:

```
push(con)  : v* := (CONS con v*)
fetch(1)   : v* := (CONS(HEAD f*) v*)
do-EQUAL   : tem := (EQUAL(HEAD v*) (HEAD(TAIL v*)))
             v* := (CONS tem (TAIL(TAIL v*)))
```

Consequently we have the following:

Theorem For every LISP0 program, a list machine program may be constructed which computes the same input-output function.

In the next section we will show that list machines can be simulated by the apparently much simpler register machine. As an aid to this development we show that the form of list machine commands may be limited without loss of computing abilities.

Lemma For any list machine program, there is an equivalent one whose commands are restricted to the following forms (where X, Y and Z are variables and L, M labels):

```
X := (QUOTE <list>)
X := Y
X := (HEAD Y), (TAIL Y), (CONS Y Z),
     (ATOM Y) or (EQUAL Y Z)
return X
goto L
if X goto L else M                              □
```

This can obviously be done by adding some extra assignment statements to break complex list machine commands into sequences of the forms above.

10. REGISTER MACHINES

Definition A *register machine* program is a program as described in Section 3.4.1, Specialization 1: Numeric Computation.

Semantics: This is as in Section 3.4.1. In short, a register machine state is a function State: Variables → N where N = {0, 1, 2, ., .}. Variables thus range over the natural numbers. Expressions are evaluated as usual and with the usual interpretations of +, *, ** (exponentiation) and - (except that 0-n is taken to be 0, for any $n \in N$). MOD and DIV denote integer modulo (remainder) and truncated division, so for any $x, y \in N$, $y \neq 0$, one has

$$0 \leq (x \text{ MOD } y) < y \quad \text{and} \quad x = y * (x \text{ DIV } y) + (x \text{ MOD } y). \qquad \square$$

Register machine commands resemble ordinary machine codes, except that variables may contain arbitrarily large natural numbers as values. In the following we show that register machines can also simulate list machines (and thus any LISP0 program); and that this remains true even if limited to the command forms X := X $\pm$1, if X = 0 goto L else M and return X.

10.1 Pairing and Selection Functions

The functions first: N → N, second: N → N and pair: N × N → N will be used to simulate the LISP0 operations on lists.

$$\text{pair}(x,y) \;=\; (2x+1) * 2^y$$

$$\text{second}(z) \;=\; \text{the largest } y \text{ such that } 2^y \text{ divides } z$$

$$\text{first}(z) \;=\; \frac{1}{2}\left(\frac{z}{2^{\text{second}(z)}} - 1\right)$$

it is easily verified that pair(x,y), first(z), and second(z) are well-defined natural numbers for any x,y,z in N, and that

1. first(pair(x,y)) = x
2. second(pair(x,y)) = y
3. pair(x,y) = pair(x',y') only if x = x' and y = y'

A pairing function provides a way of representing a pair of data objects (numbers in this case) uniquely by a single data object, and the selector functions allow a pair to be decomposed. Integers 1,2,... represent the following pairs:

(x,y)	(0,0)	(0,1)	(1,0)	(0,2)	(2,0)	(1,1)	(3.0)	(0,3)	...
pair(x,y)	1	2	3	4	5	6	7	8	...

These functions can be computed by register machines, as follows.

```
Z := pair(X,Y)   ≡  Z := (2 * X + 1)  *  (2 ** Y)
Y := second(Z)   ≡  Y := 0; U := Z;
                    while U MOD 2 = 0
                    do [U := U DIV 2; Y := Y + 1]
X := first(Z)    ≡  X := [(Z DIV second(Z)) - 1] DIV 2
```

10.2 Coding List Structures As Numbers

The input to a list program, and the data manipulated by list programs, are binary lists, whereas in contrast a register machine manipulates only non-negative integers. Consequently a direct simulation of list by register machines is impossible, so we devise a way faithfully to represent lists by numbers, and to simulate operations on lists by operations on numbers.

Recall that lists have the syntax:

```
<list>   ::= <atom> | (<list> . <list>)
<atom>   ::= <symbol | <atom> <symbol>
<symbol> ::= 0 | 1 | .. | 9 | A | B | .. | Z
```

Our encoding uses the pairing functions just described.

List Representation

Let a = {1,2,...,9,0,A,B,...Z} = $\{s_1,\ldots,s_{36}\}$ be all the symbols which can appear in LISP0 atoms. We will code a particular atom

$$a = S_{i_n} \ldots S_{i_1} S_{i_0} \qquad (1 \leq i_j \leq 36)$$

By the number

$$\bar{a} = 2k + 1 \text{ where } k = i_0 + 36i_1 + 36^2 i_2 + \ldots + 36^n i_n .$$

For example $\bar{3} = 2 * 3 + 1 = 7$ and $\overline{A1} = 2*(11*36+1) + 1 = 795$. Note that every atom is represented by an odd number.

A compound list $\ell = (\ell_1 . \ell_2)$ will be encoded (recursively) by the number

$$\bar{e} = \text{pair}(\bar{\ell}_1, \bar{\ell}_2)$$

This will always be even (since pair(x,y) = $(2x+1)2^Y$ and $\bar{\ell}_2 > 0$), so a parity test is sufficient to distinguish atoms from compound lists.

Examples

1. Atomic list 7 has code $2 * 7 + 1 = 15$

2. Compound list (2.1) has code

   $\overline{(2.1)} = (2*\bar{2}+1) * 2^{\bar{1}} = (2*5+1) * 2^3 = 88$

3. Compound list ((2.1)·2) has code

   $\overline{((2\cdot 1)\cdot 2)} = (2 * \overline{(2\cdot 1)}+1) * 2^{\bar{2}} = (2*88+1) * 2^5 = 5664$

Remarks on the encoding scheme:

1. The scheme, while simple, is clearly inefficient: small list structures are encoded as very large numbers, and many numbers are not encodings of any list structures at all. These objections are immaterial, however, since our only interest in the present development is to investigate the outer limits of computability. If we were, for example, comparing efficiency of various machine types or algorithms, more efficient codings would be needed.

2. In fact any encoding will do, provided different lists have different codes and that one can program algorithms to simulate the list machine's operations. For this are needed: comparison of two list values; decomposition of a list into its head and tail; and construction of a list from two other lists.

10.3 Register Machines Can Simulate List Machines

Theorem For any list machine program p, there exists a register machine program r such that

if p, when applied to input lists $a_1, \ldots, a_n$, halts with list a as output

then r, when applied to input numbers $\bar{a}_1, \ldots, \bar{a}_n$, halts with number $\bar{a}$ as output

Further, r halts on input $\bar{a}_1, \ldots, \bar{a}_n$ *if and only if* p halts on $a_1, \ldots, a_n$ □

Proof We will construct program r so that it simulates p step by step. By the lemma at the end of Section 9, p may be assumed to contain at most one operator per command. For each command in p there will be a command sequence in r, constructed as follows:

command in p	commands in r

X := (QUOTE ℓ)
X := Y
X := (HEAD Y)
X := (TAIL Y)
X := (CONS Y Z)
	X := $\bar{\ell}$
	X := Y
	X := first(Y)
	X := second(Y)
	X := pair(Y,Z)

X := (ATOM Y)	if Y MOD 2 ≠ 0 then X := $\overline{\text{T}}$ else X := $\overline{\text{NIL}}$
X := (EQUAL Y Z)	if Y = Z then X := $\overline{\text{T}}$ else X := $\overline{\text{NIL}}$
return X goto L	return X goto L
if X goto L1 else L2	if X = $\overline{\text{NIL}}$ then goto L2 else goto L1

Corollary any LISP0 program may be simulated by a register machine.

10.4 A Simpler Version of the Register Machine

We now show that the register machine's command set may be drastically simplified without loss of computational power. This simpler machine will then be shown simulable by a Turing machine in the next section.

Lemma For any register machine, there is an equivalent simple register machine whose commands are restricted to the following forms (where X is a variable and L, M are labels):

```
X := X + 1, X := X - 1
return X
goto L, if X = 0 goto L else M
```

Proof We show that the more complex commands can be simulated by sequences of commands of the form above. First, X := 0 may be accomplished by

```
while X ≠ 0 do X := X - 1
```

or equivalently:

```
1: if X = 0 goto 4 else 2
2: X := X - 1
3: goto 1
4:
```

Similarly X := 1, X := 2, etc. may be accomplished by the sequence above followed by "X := X + 1" an appropriate number of times.

The following sequence has the same effect as "X := Y; Y := 0":

```
X := 0;
while Y ≠ 0 do [X := X + 1; Y := Y - 1]
```

In order to accomplish X := Y without the side effect of setting Y to 0 we use a new auxiliary variable Z. Further, we see that X := Y + Z may be done using the same idea:

```
X := Y      ≡ Z := 0; while y ≠ 0 do [Y:=Y-1; Z := Z+1];
              X := 0; while Z ≠ 0 do [Y:=Y+1; Z := Z-1;
                                      X := X+1]
X := X + Y ≡ - the same, but without X := 0
X := Y + Z ≡ [X := Y; X := X + Z]
```

The following show how Y * Z, Y MOD Z and Y DIV Z can be computed; Y - Z and Y**Z may be computed similarly:

```
X := Y * Z    ≡ W := Z; X := 0;
                while W ≠ 0 do [X := X + Y; W := W-1]
X := Y MOD Z    X := Y; while X ≥ Z do X := X - Z
X := Y DIV Z    W := Y; X := 0;
                while W ≥ Z do [W := W - Z; X := X + 1]
```

Now suppose one is given an unrestricted register machine program. This can be converted to the desired form as follows:

1. Replace tests involving <, ≤, >, ≥ by tests of equality with 0. For example, X ≤ Y if and only if X - Y = 0 (since 0 - Y = 0 for all Y, by our version of -).
2. Add extra assignment statements if necessary so that assignment commands contain at most one operator, and all expressions outside assignments are constants or variables.
3. The resulting program can now be converted to the desired form by substituting the sequences above for X * Y, X DIV Y, etc. □

Corollary Any LISP0 program may be simulated by a simple register machine.

11. TURING MACHINES

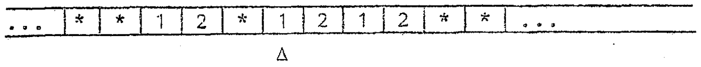
Syntax

A Turing machine program is a flow chart program as in 3.4.1, specialized to commands of the following simple forms:

Right	- move read head right one square
Left	- move read head left one square
Print a	- print a on the scanned square
goto L	
if a goto L else M	- if the scanned square contains a then go to L, else go to M.
halt	

Semantics

A Turing machine's state consists of its control point (the current "instruction counter") and a tape



Consisting of a two-way infinite string of symbols, together with a designated *scanned symbol* (marked by Δ in the diagram). The tape's symbols lie in A ∪ {*}, where A is the Turing machine's *input alphabet* and * ∉ A is called the *blank* symbol. In practice every tape will contain * in all but a finite number of symbols.

The commands given above should be self-explanatory (a more detailed description with examples may be found in [Jon73]). In the following, 1^x represents a sequence of x consecutive ones.

Definition Turing machine Z computes a partial function $f\colon N^n \overset{P}{\to} N$ (on the natural numbers) if for any $x_1,\ldots,x_n \in N$

(i) if $f(x_1,\ldots,x_n) = y$ and Z is started in the initial configuration:

$$\ldots * * 1^{x_1} * 1^{x_2} * \ldots * 1^{x_n} * * \ldots$$
$$\Delta$$

Then it will eventually execute a halt instruction in a configuration of the form

$$- \text{ anything } * 1^y * \text{ anything } -$$
$$\Delta$$

(ii) if $f(x_1,\ldots,x_n)$ is undefined then Z will not halt if started in the initial configuration. □

Theorem If $f\colon N^n \overset{p}{\rightarrow} N$ is computable by a simple register machine, then it is also computable by a Turing machine.

Proof Let f be computed by a register machine program of form

$r = [\text{read } x_1,\ldots,x_n;\ 1\colon C_1 \ \ldots\ m\colon C_m]$

By definition of simple register machines, each command is of the form $X := X+1$, $X := X-1$, return X, goto L or if $X = 0$ goto L else M . Following the pattern of previous proofs, we show how to construct from r a Turing machine Z which simulates r's actions step by step.

Let the variables of r be $X_1,\ldots,X_p$ (where $n \le p$ since $X_1,\ldots,X_n$ contain the values of the input). A computational state with $X = a_1,\ldots, X_p = a_p$ will be represented in standard form by the tape

$$\ldots * * 1^{a_1} * 1^{a_2} * \ldots * 1^{a_p} * * \ldots$$
$$\Delta$$

For the initial configuration, $a_1,\ldots,a_n$ come from input data and $X_{p+1} = \ldots = X_n = 0$.

As in the previous section, we introduce some "macros" to abbreviate frequently-occurring command sequences, and freely make use of PASCAL-like control structures. The tests in until and while are of course tests on the Turing machine's scanned symbol.

I. Right(*) Left (*) — move the scanner Δ to the first * to the right (left) of the current scanning position

```
Right(*)  ≡  until * do Right
Left (*)  ≡  until * do Left
```

II. Shift — changes configuration

from $$\ldots \text{ anything}_1 \ a \ 1^x * \text{anything}_2 \ \ldots$$
$$\Delta$$
(a = any symbol in $A \cup \{*\}$)

to $$\ldots \text{ anything}_1 1^x * * \text{ anything}_2 \ \ldots$$
$$\Delta$$

```
Shift  ≡  Print 1 ; Right(*);
          Left    ; Print * ; Right
```

The Turing machine is constructed from r by replacing each command by a sequence of Turing commands according to the following plan. Recall that p is the number og r's variables.

Command in r	Command Sequence in Z
$X_i := X_i+1$	Shift^i (i.e., shift repeated i times); Left ; Print 1 ; $\text{Left}(*)^i$

$X_i := X_i-1$	$\text{Right}(*)^{i-1}$; Right;
	if * then $\text{Left}(*)^i$
	else $[\text{Shift}^{p-i+1}, \text{Left}(*)^{p+1}]$

return X_I	$\text{Right}(*)^{i-1}$
goto L	goto L

if $X_i = 0$
goto L
else M
	$\text{Right}(*)^{i-1}$; Right;
	if * then [Left; $\text{Left}(*)^{i-1}$; goto L]
	else [Left; $\text{Left}(*)^{i-1}$; goto M]

Correctness of the simulation is easily verified (analyze the Z commands' effects on a standard configuration), so the desired result has been shown. □

Corollary Every LISP0 program may be simulated by a Turing machine.

12. COMPLETING THE LOOP

We have shown that, in spite of its extreme simplicity, the Turing machine can compute any function computable by LISP0 programs (provided the inputs are encoded in numeric form). In this section we show that LISP0 programs can also simulate Turing machines, so that (modulo data representations), LISP0, stack machines, register machines and Turing machines all have the same computational power.

The technique we use resembles that in Section 4.2. During the Turing machine's computation its tape will always be "almost everywhere blank", and so will have the form:

...	*	*	b_n	...	b_2	b_1	b_0	a_0	a_1	a_2	...	a_m	*	*	...
							Δ								

where each a_i, $b_j \in A \cup \{*\}$ and all symbols to right of a_m og to the left of b_n are * . This will be represented by a LISP0 list

$((b_0\ b_1\ \dots\ b_n)\ .\ (a_0\ a_1\ \dots\ a_m))$

Theorem Any Turing machine program can be simulated by a LISP0 program.

Proof Let the Turing machine program be

t = read $x_1,\dots x_n$; $1{:}C_1$; $2{:}C_2$ ... ; $k{:}C_k$

For the sake of simplicity we assume $n = 1$, so the initial tape is of form (p is the initial value of x_1):

*	*	...	*	1^p	*	*	...
			Δ				

whose LISP0 representation is thus the list

$$(\ (\,)\ .\ \underbrace{(1\ 1\ \dots\ 1)}_{p\ 1\text{'s}})$$

The simulating program will have form:

```
((DEFINE Execute (x1) = (Run1 (CONS 'NIL x1)))
 (DEFINE Run1 (x)       =
   (LET   (b . a)          x IN
    (LET  (a0 . arest)  =  a IN
     (LET (b0 . brest)  =  b IN Body1))))
                              ...
 ...
 (DEFINE Runn (x)       =
   (LET   (b . a)       =  x IN
    (LET  (a0 . arest)  =  a IN
     (LET (b0 . brest)  =  b IN Bodyk)))))

 (DEFINE Result(x)      =
   (LET   (a0 . arest)  =  x IN
    (IF a0 = '1 THEN (CONS '1 (Result arest))
                ELSE 'NIL)))
```

where each Turing command C_i corresponds to a LISP0 expression Body_i. Simulation of the Turing commands is straightforward:

1. goto L becomes (RunL x)

2. if sym goto L else M becomes
   (IF b0 = 'sym THEN (RunL x) ELSE (RunM x))

3. halt becomes (Result a), which returns the longest sequence of 1's which starts a (recall the definition of computation at the start of this section.)

4. "Right" in essence simply converts (b . (a0 . arest)) to ((a0 . b) . arest). The code is complicated by the possibility that a = NIL, resulting in the following. "Left" is analogous.

```
(Run_{i+1} (IF a = 'NIL
               THEN (CONS(CONS 'NIL b) 'NIL)
               ELSE (CONS(CONS a0 b) arest)
 ))
```

VI. REDUCTIONS BETWEEN COMPUTABLE PROBLEMS

13. USING REDUCTION TO PROVE UNSOLVABILITY

We have shown in Chapter IV that

1. Run is LISP0 computable, where for any LISP0 program p and input d

   Run(p,d) = the result of applying p to d (if defined)

2. The "self-halting" problem SH is not LISP0 computable, where

$$SH(p) = \begin{cases} T & \text{if Run(p,p) is defined} \\ NIL & \text{if Run(p,p) is not defined} \end{cases}$$

This can be expressed in a different way, using the idea of LISP0-decidability from Section 6.4:

Theorem. The following sets are not LISP0 decidable:

SH = {p | Run(p,p) is defined}
HALT = {(p.d) | Run(p,d) is defined}

(Note: we have used the same names for these sets as for the functions they correspond to.)

In fact a great many decision problems about LISP0 programs are not LISP0 decidable. This implies by the Church-Turing thesis that they are not decidable in any intuitive sense either. In the rest of this material we will just say "decidable" and not LISP0 decidable.

The argument for the incomputability of SH was somewhat subtle. In order to avoid having to duplicate the reasoning we introduce a new technique of reducing one problem to another.

<u>Definition</u> Let A and B be two sets (i.e. decision problems). We say that A is (LISP0) *reducible* to B, and write $A \leq B$, if there is a LISP0 computable total function f such that for all arguments

a ∈ A if and only if f(a) ∈ B □

The definition implies immediately that

1. If $A \leq B$ and B is decidable then A is decidable.
2. If $A \leq B$ and A is undecidable then B is undecidable

In case 1, an algorithm to decide membership in A can take the following form, where a is an arbitrary input:

```
compute b = f(a) ;
if b ∈ B then return T
          else return NIL
```

Case 2 is simply the contrapositive form of case 1. We now apply problem reduction to several natural questions about program behavior.

<u>Theorem</u> None of the following sets is decidable:

SH = {p | Run(p,p) is defined}

HALT = {(p d) | Run(p,d) is defined}

HALTNIL = {p | Run(p,NIL) is defined}

EMPTY = {p | Run(p,x) is not defined for any x}

ALL = {p | Run(p,x) is defined for all x}

EQUAL = {(p q) | ∀x Run(p,x) = Run(q,x)
(i.e. both are undefined or both are defined and equal) }

<u>Proof</u> Undecidability of SH and HALT have already been shown. We will first show that $SH \leq HALTNIL$, thus establishing the undecidability of HALTNIL. By the definition of reduction, we have to exhibit a LISP0 computable function such that for any LISP0 program p, p ∈ SH if and only if f(p) ∈ HALTNIL. In other words

p halts on p if f(p) halts on NIL

This can be done as follows. Let p have form

```
p = ((DEFINE P(x) = ...)...)
```

Now construct program q = f(p)

```
q = ((DEFINE Q(y) = (HEAD (CONS 'T (P 'p)))
       (DEFINE P(x) = ...)...)
```

Clearly q ignores its input, and tries to apply p to itself. Consequently q halts on NIL input (with T as output) if and only if p halts on p, or symbolically:

p ∈ SH ⇔ f(p) ∈ HALTNIL

Program q = f(p) is obviously easy to construct from p, so f is LISP0 computable. Thus $SH \leq HALTNIL$, and the undecidability of SH implies that HALTNIL is also undecidable.

Surprisingly, the same construction can be used for the other sets. The following are trivially true, where "ANY" is the LISP0 program which yields T for all input:

1. p ∈ SH ⇔ f(p) ∈ ALL
2. p ∈ SH ⇔ (f(p) ANY) ∈ EQUAL
3. p ∈ SH ⇔ f(p) ∉ EMPTY

By 1 $SH \leq ALL$ so ALL is undecidable. The function taking p to the list (f(p) ANY) is clearly LISP0 computable, so $SH \leq EQUAL$ and so EQUAL is undecidable. By 3, $SH \leq NONEMPTY$ = {p | ∃x Run(p,x) is defined}, so NONEMPTY is undecidable. Now if EMPTY were decidable, the LISP0 program which decided it could be trivially modified to decide NONEMPTY (by reversing T and NIL). contradicting the undecidability of NONEMPTY. □

The constructions above all have a program p as input. A natural question is whether it is the variability of p which leads to undecidability? or put in concrete terms, is there a *fixed* program p_0 whose halting problem is undecidable? It turns out that the answer is yes.

Theorem There is a program p_0 such that the following is an undecidable set

$$\mathrm{HALT}_{p_0} = \{x \mid \mathrm{Run}(p_0,x) \text{ is defined}\}$$

Proof Let p_0 = SI, the "self-interpreter" of Section 7. Then

$$\mathrm{Run}(p_0, (p\ d)) = \mathrm{Run}(p,d)$$

for any program p and input d. Thus $\mathrm{HALT} = \mathrm{HALT}_{p_0}$ for this p_0, so HALT_{p_0} is undecidable for at least one fixed p_0. □

14. UNSOLVABLE PROBLEMS CONCERNING CONTEXT-FREE GRAMMARS

We now use the reduction method to show that a well-known problem concerning syntax analysis cannot be solved by computer. Specifically, we will show that it is undecidable whether a context-free grammar is ambiguous, that is, whether at least one of its generated strings has two different parse trees.

First we describe Post's Correspondence Problem, or PCP for short. This is a very well-known problem which is easy to describe but which nevertheless cannot be solved algorithmically. We will show here that the PCP problem can be reduced to the ambiguity problem. Consequently, if one could solve the ambiguity problem algorithmically, one could also solve PCP algorithmically. Since this is known to be impossible ambiguity cannot be solved algorithmically either.

Post's Correspondence Problem

Given two arbitrary sequences $(x_1,x_2,\ldots,x_n)$ and $(y_1,y_2,\ldots,y_n)$ of nonempty strings of symbols from some alphabet A, the problem is

To Determine whether or not there is a nonempty sequence of integers $i_1,i_2,\ldots,i_k$ such that

$$x_{i_1}\, x_{i_2} \ldots x_{i_k} = y_{i_1}\, y_{i_2} \ldots y_{i_k}$$

Such an index sequence is called a *solution* to the correspondence problem.

Examples (all with A = {a,b}).

1.

i	1	2	3
x_i	b^3	ab	b^2
y_i	b^2	bab^2	a^2b^2

One solution sequence is 1,2,1 since $x_1x_2x_1 = b^3ab^4 = y_1y_2y_1$

2.

i	1	2	3	4
x_i	a	b^2a	a^2b	ab
y_i	ba	a^3	b	b^2a

Each pair (x_i, y_i) begins with different letters, so no solution is possible.

Theorem Post's Correspondence Problem is undecidable.

Proof outline: We have shown that the halting problem for LISP0 programs is undecidable. This implies that

$$\mathrm{TMHALT} = \{(r,n) \mid r \text{ is a Turing machine which halts on input } 1^n\}$$

is also undecidable. To see this, recall that we showed in Section 11 that any LISP0-program can be simulated by a Turing machine. If TMHALT were decidable then the LISP0 halting problem on empty input could be decided in two steps, by first constructing an equivalent Turing machine program r and then deciding whether r halts on an empty input tape.

Given that TMHALT is undecidable, PCP is shown undecidable by the reduction method of the previous section; one shows

$$\mathrm{TMHALT} \leq \mathrm{PCP}$$

Specifically, one can show that given any Turing machine program r, it is possible to construct a PCP problem which has a solution if and only if r halts on an empty input tape.

The details of the construction are somewhat involved, so we refer the interested reader to [Bir76]. □

Theorem The following two sets are undecidable:

$$AMB = \{G \mid G \text{ is an ambiguous context-free grammar}\}$$

$$INT = \{(G_1,G_2) \mid G_1,G_2 \text{ are context-free grammars with } L(G_1) \cap L(G_2) \neq \emptyset\}$$

Proof We show that PCP $\leq$ AMB and PCP $\leq$ INT; undecidability follows thereby from the undecidability of PCP. Recall that the input data to PCP is a pair of sequences of strings

$$(x_1,\dots,x_n) \quad \text{and} \quad (y_1,\dots,y_n) \qquad (x_i,y_i \in A^*)$$

We first show how one, if given $x_1,\dots,x_n,y_1,\dots,y_n$, can construct two context-free grammars G_1, G_2 such that

$L(G_1) \cap L(G_2) \neq \emptyset$ if and only if

$$\exists\, i_1,i_2,\dots,i_k \ (x_{i_1} x_{i_2} \dots x_{i_k} = y_{i_1} y_{i_2} \dots y_{i_k})$$

Construction Let G_1, G_2 contain the following productions, where c is a symbol not in A

G_1 :	$S \to aEa$	for every symbol $a \in A$
	$E \to c$	
	$E \to aEa$	for every $a \in A$

G_2 :	$F \to c$	
	$F \to x_i^R \ F \ y_i$	for $i = 1,2,\dots,n$ (Note: x^R = x written backwards)

It is quite clear that

$$L(G_1) = \{x^R c x \mid x \in A^+\}$$

$$L(G\ ) = \{x_{i_k}^R \dots x_{i_2}^R c\, y_{i_1} y_{i_2} \dots y_{i_k} \mid i_1,i_2,\dots,i_k \text{ is an index sequence}\}$$

Clearly if $L(G_1) \cap L(G_2) \neq \emptyset$ then there exists a string

$$x^R c x = x_{i_k}^R \dots x_{i_1}^R \dots x_i^R c\, y_{i_1} \dots y_{i_k}$$

such that

$$x = x_{i_1} \dots x_{i_k} = y_{i_1} \dots y_{i_k} \neq \varepsilon$$

In other words, this PCP problem has a solution if $L(G_1) \cap L(G_2) \neq \emptyset$. The converse is immediate, so

$$((x_1,\dots,x_n),\ (y_1,\dots,y_n)) \in PCP \text{ iff } (G_1,G_2) \in INT$$

Consequently PCP $\leq$ INT, and so INT is undecidable.

We now wish to show AMB undecidable, and do this by showing PCP $\leq$ AMB. The goal is thus to show that, given two sequences $(x_1,\dots,x_n)$ and $(y_1,\dots,y_n)$, one can construct a context-free grammar G which is ambiguous if and only if

$$\exists i_1,\dots,i_k \ (x_{i_1} \dots x_{i_k} = y_{i_1} \dots y_{i_k})$$

Construction G has productions

$S \to a\ E\ a$	for every $a \in A$
$S \to F$	
$E \to c,\ E \to a\ E\ a$	for every $a \in A$
$F \to c,\ F \to x_i^R\ F\ y_i$;	

It is clear that $L(G) = L(G_1) \cup L(G_2)$, and that a string $z \in A^*$ has two parse trees if and only if it lies in *both* $L(G_1)$ and $L(G_2)$. Consequently $G \in AMB$ if and only if $(G_1,G_2) \in INT$, which we have just seen to be true just in case the given PCP has a solution. Consequently PCP $\leq$ AMB. Since PCP is undecidable AMB must thus also be undecidable. □

Example

PCP-problem:

i	1	2	3
x_i	bbb	ab	bb
y_i	bb	babb	aabb

Corresponding Grammars:

G_1: $S \rightarrow a E a \mid b E b$

$E \rightarrow c \mid a E a \mid b E b$

G_2: $F \rightarrow c$

$F \rightarrow bbbFbb = x_1{}^R F y_1$

$F \rightarrow baFabb = x_2{}^R F y_2$

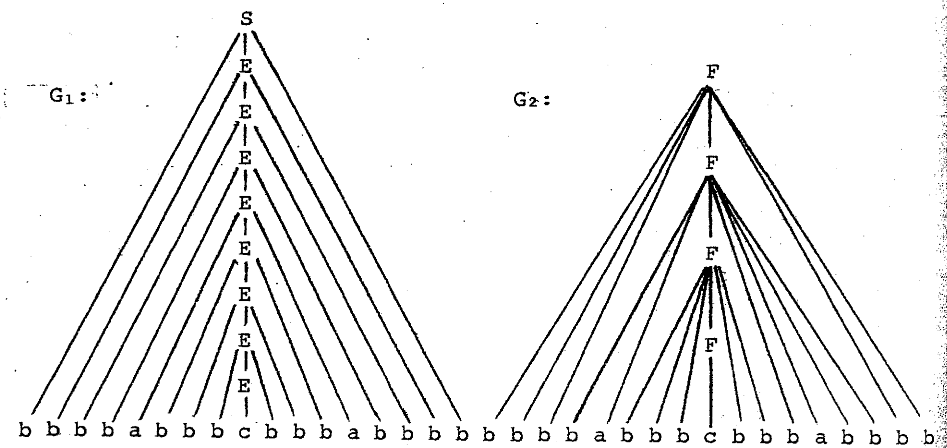
$F \rightarrow bbFaabb = x_3{}^R F y_3$

A Solution Sequence: 121

$y_1 y_2 y_1 = \boxed{bb \mid babb \mid bb}$

$= x_1 x_2 x_1 = \boxed{bbb \mid ab \mid bbb}$

Two Derivations for $(x_1 x_2 x_1)^R c y_1 y_2 y_1$:


G1:
S
E
E
E
E
E
E
E
E
b b b b a b b b c b b b a b b b b
G2:
F
F
F
F
b b b b a b b b c b b b a b b b b

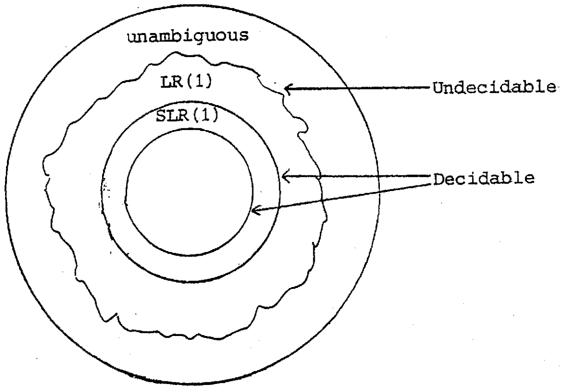
Remarks

1. The arguments above prove that there are no algorithms whatsoever which can solve the ambiguity problem for context-free grammars.
2. But then what about parser generator systems such as YACC and BOBS? According to [AhU77] (example 6.8) every SLR(1) grammar is unambiguous. Further, it is quite clear that it is decidable whether a grammar is SLR(1).

This apparent contradiction is not a real one, because *not all unambiguous grammars are SLR(1)*. The great advantage of the SLR(1) class is precisely this: it is simultaneously a class

1. which is *large enough* to include most of the natural grammars used in programming languages
2. which contains *only unambigous* grammars (and this is essential for practical use)
3. whose membership problem "is G SLR(1)?" can be decided with in *reasonable computation time*.

All Context-free Grammars



REFERENCES

[AhU77] Aho, A. and J. Ullman, *Principles of Compiler Design*, Addison Wesley (1977).

[Backus] Backus, J.W., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM* 21 (8), pp 613-641 (1978).

[Bir76] Bird, R., *Programs and Machines*, Wiley (1976).

[Chu36] Church, A., A note on the Entscheidungsproblem. *Jour. Symbolic Logic* 1, pp 40-41, 101-102 (1936).

[Chu51] Church, A., *The Calculi of Lambda-Conversion*, Annals of Math. Studies 6, Princeton University Press, New Jersey (1951).

[EaS70] Earley, J. and H. Sturgis, A formalism for translator interactions, *Comm. ACM* 13 (70), pp 607-616 (1970).

[Gor79] Gordon, M., *The Denotational Semantics of Programming Languages*, Springer Verlag (1979).

[HLS72] Hindley, J.R. and B. Lercher, J.P. Seldin, *Introduction to Combinatory Logic*, Cambridge University Press, England (1972).

[Hoa75] Hoare, C.A.R., Recursive data structures, *Int. J. Comput. Info, Sci.* Vol 4, No 2, pp 105-131 (1975).

[HOPE] R.M. Burstall, D.B. MacQueen, D.T. Sannella, HOPE: An experimental applicative language, Technical report, University Edinburgh (1979).

[Jon73] Jones, N., *Computability Theory: An Introduction*, Academic Press (1973).

[McC62] McCarthy, J. et al., *LISP 1.5 Programmer's Manual*, MIT Press (1962).

[Min67] Minsky, M. *Computation: Finite and Infinite Machines*, Prentice-Hall (1967).

[Pau83] Paulsen, C., A parallel implementation of a functional language, Report DAIMI IR-45, University Aarhus, Denmark (1983).

[Sto77] Stoy, J., *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, MIT Press (1977).

INDEX

Ordliste

abandoned	brudt
abbreviation	forkortelse
admissible	mulige
all-encompassing	altomfattende
application	anvendelse, funktionskald
applicative language	funktionsbaseret programmeringssprog
atom	grundsymbol
avoid	undgå
bootstraps	støvlehank
call-by-name	navneoverførsel
call-by-need	parametre beregnes kun hvis der er behov for det
call-by-value	værdioverførsel
captured	fanget
compatability	beregnelighed
concatenation	sammenkædning
concept	begreb
context	sammenhæng, omgivelser
continuation	fortsættelse
decidable	afgørlig
denote	betegne, være symbol for
dichotomy	tvedeling
environment	omgivelser
evaluation	udregning
flowchart	rutediagram
halting problem	standsningsproblem
imperative language	sprog med tildelingssætninger
implies	medfører
net-effect	netto effekt
operational semantics	operationel semantik
property	egenskab
scope	virkefelt, område, hvor variabelnavn er kendt
state-transition	tilstandsovergang
unsolvability	uløselighed

Fortegnelse over rapporter i 1984

84/1 Production and Location on a Network under Demand Uncertainty. Francois Louveaux and Jacques-Francois Thisse.

84/2 Typed Representation of Objects by Functions. Jørgen Steensgaard-Madsen.

84/3 Steiner Problem in Halin Networks. Pawel Winter.

84/4 An Algorithm for the Enumeration of Spanning Trees. Pawel Winter.

84/5 Open Problems Presented at the Copenhagen Workshop on Computer Vision. Knud Henriksen, Peter Johansen, Søren Olsen.

Fortegnelse over rapporter i 1983

83/1 Stepwise Development of Operational and Denotational Semantics for Prolog. Neil D. Jones and Alan Mycroft.

83/2 A Skeleton Interpreter for Specialized Languages. Jørgen Steensgaard-Madsen.

83/3 Naming Commands. An Analysis of Designers' Naming Behaviour. Anker Helms Jørgensen et al.

83/4 Gendannelse af forringede billeder ved invers - og Wienerfiltrering. Jørgen Bansler og Søren Olsen.

83/5 Stepwise Development of Logic Programmed Software Development Methods. Gregers Koch.

83/6 An Algorithm for the Steiner Problem in the Euclidean Plane. Pawel Winter.

83/7 Eksperimentelle teknikker i systemarbejdet. Jørgen Bansler og Keld Bødker.

83/8 Generering af en oversættergenerator. Mads Tofte.

83/9 Interval Arithmetic Implementations Using Floating Point Arithmetic. Michael Clemmesen.

83/10 Design practice and interface usability: evidence from interviews with designers. Anker Helms Jørgensen, N. Hammond, A. MacLean, P. Barnard, and J. Long.

83/11 En model for brugeres opfattelse af edb-baserede systemer. Jan Chr. Clausen.

83/12 Definition of the Programming Language MODEF. Jørgen Steensgaard-Madsen og Lars Møller Olsen.

83/13 The effect of task structure in interactive systems: a pilot experiment. Anker Helms Jørgensen, Phil Barnard, Nick Hammond, Allan MacLean.

83/14 The psychology of developing and using computer systems: five contributions. Anker Helms Jørgensen.

83/15 Systemudvikling som element i den kapitalistiske teknologiudvikling. Jørgen Bansler og Keld Bødker.

83/16 Oversætterteknik for programmeringssprog ved hjælp af PROLOG. Flemming Als, Carsten Hendriksen og Jens Johansen.

83/17 Generalized Steiner Problem in Outerplanar Networks. Pawel Winter.

Fortegnelse over rapporter i 1982

82/1 Worst-case Behavior of Simplex Algorithms: From Theory to Numerical Examples. Jens Clausen.

82/2 An Experimental Pattern Recognition System. Yu Song Yu and Peter Johansen.

82/3 Structuring of Dedicated Concurrent Programs Using Adaptable I/O Interfaces. Eric Jul.

82/4 Control Flow Treatment in a Simple Semantics-Directed Compiler Generator. Henning Christiansen and Neil Jones.

82/5 Edb-helhedsplanlægning. Elin Rønby Pedersen.

82/6 Definition af Concurrent Pascal-maskinen og dokumentation af CP-maskinsimulatoren på DIKU's PDP-11/45 anlæg. Jesper Lund og Lars LUnd Hansen.

82/7 A Survey of Literature on Modularization and Abstraction in Programming. Erik Reeh Nielsen.

82/8 Automatisk orddeling. Hasse Hansson.

82/9 Sagen om superdatamaten - anden runde. Edda Sveinsdottir og Erik Frøkjær.

82/10 CONCEPTUAL GRAPHS - A Denotational Semantics Approach. Jørgen Flensholt.

82/11 Brugervejledning for Pascal Class. Lars Møller Olsen.

82/12 A Module Development Discipline. Jørgen Steensgaard-Madsen.

82/13 Combined LP and Quasi-Newton Methods for Nonlinear ℓ_1 Optimization. Jørgen Hald and Kaj Madsen.