Abstract

A description is given of Gedanken, a simple typeless language, developed by John Reynolds, and an implementation of the language on the IBM 360/44 computer at St. Andrews University is presented.

The formal definition of Gedanken is based on work by Landin on his SECD Machine and his later Sharing Machine, and a summary of these is given.

Reynolds has specified a formal definition of Gedanken, based on the Vienna definition method. A description of this is given.

A description is given of the author's implentation of Gedanken using BCPL as the defining language. The implementation proved to be inefficient, and a critical examination of it is made in an attempt to discover the sources of the major inefficiencies. A number of changes are suggested to remedy these inefficiencies.

AN IMPLEMENTATION OF GEDANKEN

BY

STEPHANIE J. HOWLETT

I hereby declare that this thesis has been composed by myself; that the work of which it is a record has not been accepted in any previous application for a higher degree and was undertaken on 1st October 1973, the date of my admission as an M.Sc. student under Ordinance No. 51.

I hereby declare that the conditions of the Ordinances and Regulations for the degree of Master of Science have been fulfilled by the candidate, Stephanie J. Howlett.

ACKNOWLEDGEMENT

It is my pleasure to acknowledge the financial support of an SRC course award. The BCPL system used was supplied by Martin Richards of Cambridge and maintained by Patrick Currivan of the Computing Laboratory in St. Andrews.

Table of Contents

CHAPTER 1 - INTRODUCTION

Gedanken (refs 1,2) is a language unusual for its generality and completeness, and in which it is thereby easy to model many of the features of other high-level languages. In particular it may conveniently be used to model the more unusual and subtle concepts in high-level programming, and would therefore make a very good tool for teaching computer semantics to more advanced students. The aim of the work described in this thesis was to produce an implementation of Gedanken which would be of use in such courses.

A working and complete implementation was produced but it became apparent that this version was extremely inefficient. In fact the implementation failed to achieve its aim, since all but the simplest programs ran so slowly that it was of no use as a teaching tool.

In the thesis a description of the implementation is given, and then an attempt is made to assess the major inefficiencies and to suggest how these might be remedied.

A brief introduction to Gedanken itself is given in the latter part of this chapter.

The definitional machine for Gedanken is based closely on work done by Landin in which he defines his SECD machine, and his later Sharing Machine. An account of this work is given in Chapter 2.

The formal definition of Gedanken, based on the Vienna definition method, specifies an abstract syntax for Gedanken, and lays down the manner in which concrete programs should be translated into abstract programs. The semantics of the language are then defined by an interpreter for these abstract programs. These aspects of the formal semantics are described in Chapter 3 and compared with the SECD and Sharing Machine systems.

Chapter 4 describes the author's implementation of Gedanken on the IBM 360/44 computer at St Andrews University, using BCPL as the defining language. This implementation adheres closely to Reynolds' definitional system, and a description is given of the way in which this is modelled in BCPL.

The final chapter takes a critical look at the implementation in an attempt to discover the sources of the major inefficiencies. A number of changes are suggested to remedy these inefficiencies, from which it is believed, a useable BCPL implementation would result.

The Philosophy of Gedanken

Gedanken has been described by its author as an attack on the problem of "the simultaneous achievement of simplicity and generality in language design"*. These points are exposed using the method due to Strachey (ref 3) This method uses the two related criteria of types and domain structures for the description and comparison of programming languages.

Gedanken follows the philosophy of dynamic types, whereby each object has a type, but these types are not tested at compile time. Types are recognised by differences in the internal representation of objects permitting explicit type-testing of the values of identifiers. It is left to the user to provide his own feasibility checks on the data within the functions he designs, or to allow the run-time system to complain if asked to perform impossible tasks.

The domain structure of Gedanken is :-

D=V=N+Ch+T+J+A+F+L

F=V-(S-(V S))

L=L+(Updater loader)

Updater=V-(S-S)

Loader=S-(V S)

*Throughout this thesis all items quoted but not credited are taken from ref 1, "Gedanken - A Simple Typeless Language Which Permits Functional Data Structures and Co-routines" by John C.Reynolds

where the following abbreviations are used :-

N - integers

Ch - characters

T - truth values

J - labels (jump-points)

A - atoms

F - functions

L - locations

V - stored objects,i.e. objects which may be assigned

D - denotations, i.e. objects which may be named

S - stores or machine states

Atoms are special objects, similar to LISP atoms. Examples of their use appear below.

The Updater Loader pairs are unusual, and relate to the implicit reference system, also to be described later.

The domain structure reveals a number of points concerning the language :-

(a) Real numbers are excluded from the current version of the language.

(b) There are no data structures included in the basic domains. This is because all data structures are treated as functions.

(c) All functions in Gedanken take a single value as argument and produce a new machine state and a single value result. However, both the argument and the result may be the functional equivalent of sequences of any length (including zero).

(d) Labels have the same status as other identifiers and may be assigned or returned as the result of a function. It is this that is responsible for many of the powers peculiar to Gedanken.

(e) The general structure of Gedanken is fairly simple, and all objects may be named, stored or assigned. It is also very clean, in the sense that any value permitted in some context

of the language is permissible in any other meaningful context.

Gedanken is also a language based directly upon the λ-calculi of Alonzo Church (ref 4). Actual "λ" symbols are used, and functional application is effected by means of β-substitutions.

A Brief Introduction to Gedanken as a Programming Language*

(a) Functions and Functional Data-Structures

Functions in Gedanken are created by the evaluation of λ-expressions. When a λ-expression is evaluated any variables occurring free in it are bound to the values they possess at the time of evaluation. This binding is the same as that used in Algol.

It has been mentioned that all data-structures in Gedanken are treated as functions. A data-structure is any function which, when applied to an appropriate argument, will yield the corresponding field or component value.

For example, consider a vector to be a one-dimensional array with lower and upper bounds the integers b_1 and b_2 respectively. Such a structure could be represented by a function applicable to the integers from b^1 to b^2 inclusive, and yielding for each integer the appropriate component of the vector.

A semi-basic function VECTOR is provided, which takes a function and two integers as arguments, and produces a functional vector whose bounds are the two given integers, and whose components are the results of applying the function to all the integers in the given range. The vector may also be applied to the atoms UL or LL, and then yield its upper or lower bound respectively.

* For more detail references 1 and 2 should be consulted.

A sequence is a vector with a lower bound of 1. A sequence may be created by a function like any other vector, but they mayalso be created by the evaluation of a special expression -form, a sequence expression. These are either empty or take the form

expression , expression , expression *

These are very similar to the collateral clauses of Algol 68.

Sequence expressions are parallelled by sequence parameter forms , which are either empty or take the form

pform , pform , pform

"In general, if a is any value and p is any parameter form, then the binding of p to a is defined recursively as follows:

(1) If p is an identifier, then p is bound to a.

(2) If p has the form (p'), then p' is bound to a.

(3) If p is a sequence parameter form, p_1, , p_n (n≠1), then a, which must be a function, is applied to each integer from 1 to n, and each p_i is bound to the value of a(i).

The combined syntax of sequence expressions and sequence parameter forms is designed to preserve conventional notation for functions of several arguments".

This approach to data structures means that a function expecting an argument of a given form will accept any function producing a logically equivalent structure,regardless of the internal representation.

(b) Assignment

Assignment in Gedanken is possible only to special entities called references, each of which possesses a value, and

* Braces are used around a syntactic entity which may occur any number of time (including zero) in a given syntactic form.

which may themselves be possessed by either an identifier or another reference. Assignment is defined, as it is in Algol 68, to alter the relationship between a reference and its value, rather than between an identifier and its value.

There are three basic functions for reference manipulation:

REF X returns a distinct new reference, initialised to possess the value X.

SET (R,X) (which may be abbreviated to R:=X) causes the reference R to possess the value X, and also returns the value of X.

VAL R returns the value possessed by the reference R.

To reduce the need for frequent use of the VAL function a coercion convention is introduced, whereby references are automatically replaced by their values in contexts where they would otherwise be meaningless. A basic function COERCE is also provided for use elsewhere by the programmer.

Rather more unusual than the above explicit references are Gedanken's <u>implicit references.</u> These are functional references, analogous to the functional data structures described above. Associated with each implicit reference are two functions. The first, of one argument, is evaluated each time the reference is set, i.e. assignment to the reference(SET (R,X) or R:=X) causes this function to be applied to the argument of the assignment. The other is a function of zero arguments, and this is evaluated whenever the implicit reference is evaluated.

An implicit reference is created by the execution of the basic function IMPREF, which requires for its arguments two functions, say SETF and VALF, of one and zero arguments respectively. The application of IMPREF to its arguments creates a new implicit reference which satisfies the predicate ISREF, and may

be coerced like an explicit reference. However the effects of SET and VAL on an implicit reference are to execute the functions SETF and VALF respectively.

This is demonstrated in the following example where it should be considered that Y is an explicit reference global to V, whose value is a sequence.

```
V IS IMPREF (λX Y:=CONS(X,Y),
             λ() (Z IS Y 1; Y:=TAIL Y; Z))
```

Then SET(V,P) would be equivalent to

```
(λX Y:=CONS(X,Y)) P
```

i.e. P would be added to the front of the sequence possessed by Y.

VAL V would remove the first item from this sequence and return it as a result.

In this way an implicit reference could be used to operate a stack.

(c) Program Control

There exist certain semantic differences between labels in Gedanken and those used in most other languages. It is usual that the statement GOTO L should transfer control to the statement following the label L. In Gedanken, evaluation of the statement GOTO L will also cause the state of the computation to revert to that at the time when L was evaluated. In effect machine states may be stored by the use of labels. All identifier bindings holding at the time of the evaluation of L come into play again, and the information necessary to complete block exit becomes available again, although references do not revert to their former values.

Another feature peculiar to Gedanken, which makes the above assume great importance, is the fact that labels are treated as first-class objects, and may be assigned, or returned

as the result of an expression or function. These two features together present the capability of jumping back into a block after it has been exited, thereby producing many interesting and unconventional programming possibilities. For example, in such things as parsers it is often convenient to return to a previous control state. In Gedanken it is possible to do this using a simple GOTO command, all back-tracking being rendered unnecessary

Labels may be assigned and stored, so that succeeding control states in a block which is iterated may be saved. Another capability presented is the use of co-routines, i.e. programs which can relinquish control to a calling program, and later be re-activated to continue computation.

CHAPTER 2 - HISTORIC BACKGROUND

The definitional machine for Gedanken has been based upon the work done by Landin in the mid-60's, on programming languages (refs 5,6 and 7). A brief account of this work is given here as a basis for a discussion of the Gedanken abstract machine.

The AE/SECD Machine System

The basis of the work was the demonstration that certain of the expression forms used in programming languages can be expressed in terms of λ-expressions, operator/operand combinations and expressions, and then be evaluated mechanically. An abstract machine to perform this evaluation was specified.

The class of applicative expressions (referred to hereafter as AE's) is defined to denote the expressions constructed using these three expression-types, and structure definitions are introduced as a means of describing and manipulating composite information-structures such as AE's.

A structure definition for a given class of object will specify for each alternative format:-

(a) The number of components.
(b) The type of each component.
(c) The identifiers to be used for the predicates and component selectors appropriate to that class.

For example, the structure definition for an AE is given as:-

"An AE is either
    an identifier
or a λ-expression (λexp) and has a

bound-variable (bv) which is an identifier-list

and a λ-body (body) which is an AE

or a combination and has an operator (rator) which is an AE

and an operand (rand) which is an AE".

Here 'identifier', 'λ-expression' and 'combination' (and any abbreviations after them) are the predicates, and 'bound-variable', 'λ-body ', 'operator' etc are the selectors.

It is possible to evaluate an AE mechanically if the values of all its free identifiers are known, and if all arguments in it are compatible with their functions. If mathematical expressions or computer programs could be expressed as single AE's then these could be evaluated mechanically in the same way. It is demonstrated that this may in fact be done for certain features important in programming languages, namely function definitions, lists, conditional expressions and recursive definitions.

Function Definitions

Function definitions may be expressed very easily as AE's in such a way that there is no dummy variable on the left-hand side, e.g.

$$f(x) = x^2+2 = x.x^2+2$$

This paves the way for expressing as an AE an expression which uses an auxiliary function definition, e.g.

$$f(2) - f(4) = (f.f(2)-f(4))(x.x^2+2)$$

where $f(x)=x^2+2$

Lists

An expression list may be rendered as a single AE by considering commas as binary infixed operators with the following

effect.

Let the two operands be a_1 and a_2, then

(1) If a_2 is a list, it is transformed into a list of length one more by being prefixed by a_1.

(2) If a_2 is not a list then a list is made by prefixing a_2 to the empty list, and prefixing a_1 to the list so formed.

A list may have components which are lists, thus forming a list-structure, and this may be expressed as an AE in the same way.

If lists are considered in this way, any function may properly appear with a single argument if this is a list of appropriate length.

Conditional Expressions

A conditional expression may be rendered as an AE by considering 'if' as a function-producing function such that:

if (true) = 1st

and if (false)= 2nd

The function produced as its result will expect as an argument a 2-list, whose two components represent the two branches of the conditional. In this way, a conditional expression having the form

if e_1 then e_2 else e_3

would be expressed as the AE

if $(e_1)(e_2, e_3)$

However this rendering is not acceptable in the case where one or other of the arms of the conditional may be undefined since it would be necessary to evaluate both e_2 and e_3 in order to form the 2-list.

For example consider the AE

if (a=0) (1, 1/a)

1/a is undefined when a=0, and thus the list could not be formed.

It is necessary that only the appropriate branch of

the conditional should be evaluated. This may be effected by means of λ-expressions with dummy arguments. An expression of the form

if e_1 then e_2 else e_3 is expressed as

if (e_1) $(\lambda()e_2, \lambda()e_3)$ ()

Both arms of the conditional are now always defined, their values being functions of no arguments, but only the appropriate one is actually applied and evaluated in any case.

Recursive Definitions

A method of rewriting recursive definitions so that they are not formally circular by means of the 'fixed point function' is used to demonstrate how recursive definitions may be expressed as AE's.

The Evaluation of AE's

The evaluation of an AE is defined as taking place relative to an environment which provides the value for each identifier occurring free in the AE. An environment may be thought of as a function which takes an identifier as argument and returns its value, which may be a number, or a list of numbers, or a function, or a list of functions.

To find the value of any AE in a given environment the function 'val' is defined, such that val E X is the value of expression X relative to environment E. The function val is specifiedby the following rules:

1. If X is an identifier then val E X=E X
2. If X is a -expression, then val E X is that function whose result for any given argument x may be found by evaluating the body of X in a new environment E', where E' may be derived from E by pairing the bound-variables of X with the components of x (where x must be a list of appropriate length).

3. If X is a combination, val E X may be found by subjecting both the operator and the operand to val E, and applying the result of the former to the result of the latter.

These rules may be used to evaluate AE's mechanically, and an abstract machine to do so is described. In this machine, the state of progress of the evaluation at any time may be defined entirely in terms of four components. Thus a machine-state may be described by the following structure definition:-

"A state consists of a stack, which is a list, each of whose items is an intermediate result of evaluation, awaiting subsequent use;
and an environment, which is a list-structure made up of name/value pairs;
and a control, which is a list, each of whose items is either an AE awaiting evaluation, or a special object designated by 'ap', distinct from all AE's;
and a dump, which is a complete state, i.e. comprising four components as listed here".

A state may be denoted by (S,E,C,D).

Defining:

(a) a closure to comprise a λ-expression and the environment relative to which it was evaluated,

(b) h and t as being the head and tail functions respectively on lists

then the procedure for the evaluation of an AE, say X, is as follows:

Evaluation begins from a state with environment E, and control X. h C is examined at each step of the evaluation, and a new state is produced, in a manner depending on h C.

The transition function applied at each step may be

defined by:-

1. If C is null then:

   let current dump be (S',E',C',D')

   The current state is replaced by hS:S',E',C',D'

   (where x:L is used to denote prefixing x to L)

2. If C is not null then:

   (a) If hC is an identifier then

   S:=location E X E:S (where the function 'location' is defined such that location E X denotes the selector which selects the value of X from E)

   C:=tC

   (b) If hC is a λ-expression X, the closure derived from E and X is loaded onto the stack.

   (c) If hC is the special object 'ap' then if hS is a closure derived from E' and X' then

   S:= the empty list

   E:= the environment derived from E' by associating the bound variables of X with the component(s) of 2nd S.

   C:=unitlist(body X')

   D:=(t(tS),E,tC,D)

   (d) If hC is a combination X then

   C:=rand X:(rator X:(ap:tC))

If X is an AE, and E is an environment such that val E X is defined, then starting at any state S,E,X:C,D, and repeatedly applying the transition function, eventually the state val E X:S,E,C,D will be reached, i.e. the value of X in the current environment will have been loaded onto the stack.

In order for the above transition rule to suffice for

the evaluation of AE's it is necessary to assume that the initial environment binds the values of constants to their representations, and also that any basic functions used, such as 'if', 'prefix' etc. are defined in the environment as λ-expressions.

C becomes empty whenever a function has been applied and the result loaded on the stack. This result may be the final result or it may be the component of another AE, the evaluation of which is continued by installing the most recent dump as the new state, and using the result on the stack during the evaluation. Thus the dump is used to store the state of the machine at the start of evaluation of an AE, at any level of nesting. This corresponds to the block-structure of certain high-level languages, such as Algol-60, a new dump element being added each time a new 'block' is entered.

The IAE/Sharing Machine System

In subsequent work an attempt was made to model Algol-60 in a similar manner, using a development of the AE/SECD system, the IAE/'sharing machine' system which could deal with imperative features.

To model Algol-60 it was necessary to add jumps and assignment to the AE/SECD system. This made the evaluation of expressions much more difficult, since the value of any expression was no longer dependent solely on the values of its subexpressions, but also on the side-effects produced by their evaluation. A variable declared and used in the evaluation of an expression might be changed by assignment in that expression itself or one of its sub-expressions. Also there is the problem that two variables might be declared as equivalent, so that assignment to one will also change the value of the other.

Landin's sharing machine models the fact that distinct

'positions' in the machine, with equal occupants might share the same representation, and thus get updated collectively.

The expressions evaluated by the machine are called imperative applicative expressions (IAE's) and consist of AE's with the addition of

an assigner which consists of a lefthand side (LHS),
which is an IAE
and a righthand side (RHS),
which is an IAE

Each state in the system is characterised by both an SECD state, and also an equivalence relation, which specifies the sharing among its component positions. Each time the transition rule is applied it is necessary to specify how the equivalence relation changes. This means that while the semantics of AE's may be specified formally independently of any machine, it is impossible to describe the semantics of IAE's other than in terms of a machine.

There are four rules which govern whether or not two state-positions share:

1. When an identifier is scanned, the stack-head is left sharing with the environment position which holds the value of the identifier.

2. When a closure is applied, the newest member of the new environment shares with any surviving co-sharers of the old stack-head. Thus a function can have non-local effects by assigning to arguments that are called by reference. This also means that there is no need to use a special procedure to scan an identifier on the LHS of an assigner.

3. When a closure is applied, components of older levels of the new environment share with the corresponding components of the environment from which it was drawn. This means that

'positions' in the machine, with equal occupants might share the same representation, and thus get updated collectively.

The expressions evaluated by the machine are called imperative applicative expressions (IAE's) and consist of AE's with the addition of

an assigner which consists of a lefthand side (LHS),
which is an IAE
and a righthand side (RHS),
which is an IAE

Each state in the system is characterised by both an SECD state, and also an equivalence relation, which specifies the sharing among its component positions. Each time the transition rule is applied it is necessary to specify how the equivalence relation changes. This means that while the semantics of AE's may be specified formally independently of any machine, it is impossible to describe the semantics of IAE's other than in terms of a machine.

There are four rules which govern whether or not two state-positions share:

1. When an identifier is scanned, the stack-head is left sharing with the environment position which holds the value of the identifier.

2. When a closure is applied, the newest member of the new environment shares with any surviving co-sharers of the old stack-head. Thus a function can have non-local effects by assigning to arguments that are called by reference. This also means that there is no need to use a special procedure to scan an identifier on the LHS of an assigner.

3. When a closure is applied, components of older levels of the new environment share with the corresponding components of the environment from which it was drawn. This means that

non-local effects can also be achieved if a function assigns to its free variables.

4. When a control-string is exhausted, the new stack-head is left sharing with any remaining co-sharers of the old stack-head, so that an application of a function is appropriate as the LHS of an assigner.

Execution of an Assigner

It has been mentioned that the left and right-hand sides of an assigner may both be evaluated in the same way. However, a LHS must refer to a previously named object, or a component thereof, already occupying a certain position in the current state. Execution of an assigner will change the value of this state-position, and hence of all positions sharing with it, and leave a nugatory result on the stack.

Jumps

In order to effect jumps in the system an operation "J" is introduced, applicable to functions, which has the effect of forcing an exit from functions under certain conditions,

e.g. f(x)=.....g(.....,.....).....

where g=Jλ(u,v).....

g may occur anywhere in the definiens.

If ever this sub-expression is evaluated while f is being applied, there will be an immediate return from f , and the value of the subexpression will be given as the result of f.

A program-closure is introduced as another possible result of evaluation. If J is applied to a closure, it is transformed to a program-closure. This differs from a closure in that in addition to including the current environment (for subsequent installation) and an expression (for evaluation) it also includes the current dump, which is installed when the

program-closure comes to be activated. By contrast, when a closure is activated, the dump is used to record the current state for later resumption. An expression of the form J(λL.S) is known as a 'program-point'.

CHAPTER 3 - THE FORMAL SEMANTICS OF GEDANKEN

The semantic interpretation of Gedanken programs is formally defined by the specification of:-

(a) A set of rules by which a concrete Gedanken program may be translated into an abstract data structure.

(b) An abstract machine to evaluate such data-structures.

Abstract Gedanken

Before it can be evaluated a parsed Gedanken program must be translated into an abstract form, an information structure amenable to evaluation by the interpreter.

An abstract Gedanken program is made up of sequences and records, where:-

(a) A sequence is the functional equivalent of a one-dimensional array.

(b) A record is the functional equivalent of a finite collection of fields, each identified by a field-name. Each record belongs to a particular class such that all records in the same class have the same set of fields.

For a particular field of all records in a particular class, the range of possible field values is specified by a class definition, of the form:-

(CLASS,⟨classname⟩,(⟨fieldname⟩,⟨range descriptor⟩))

where

⟨range descriptor⟩::=⟨set name⟩|SEQ,⟨setname⟩

If the range descriptor is a set-name, S, then the corresponding field is a member of the set denoted by S. If the range descriptor has the form SEQ, S then the corresponding field is a sequence, the components of which are all members of the set denoted by S.

"A set name may be any of the following: (1) INTCLASS, BOOLCLASS, and CHARCLASS, denoting the sets of integers, boolean values, and characters respectively; (2) UNIVERSAL, denoting the universal value set; (3) A class name, denoting a class of records; (4) A union name, denoting the union of sets denoted by other set names. The meaning of union names is described by expressions called union definitions, with the form:

⟨union definition⟩::=(UNION,⟨set name⟩{,⟨set name⟩}*)

The union definition $(UNION, s_0, s_1, \ldots, s_n)$ implies that the set name s_0 denotes the union of the sets denoted by $s_1, \ldots, s_n$. Circular union definitions, e.g. (UNION,X,X) are not permitted.

A collection of interrelated class and union definitions defining various record classes and other sets, is called an abstract syntax definition."*

An abstract syntax is equivalent to Landin's structure definitions. Thus the three definitions:

$(UNION, c, c_1, c_2)$
$(CLASS, c_1, (f_1, s_1))$
$(CLASS, c_2, (f_2, s_2), (f_3, s_3))$

are equivalent to the structure definition

a c is either a c_1 and has an f_1 which is an s_1
or a c_2 and has an f_2 which is an s_2
and an f_3 which is an s_3

* N.B. in this chapter those sections in quotation marks are taken from John C. Reynolds' "Gedanken: A Simple Typeless Language Which Permits Functional Data Structures and Coroutines" 1969, Argonne national Laboratory, ANL-7621.

The abstract syntax for abstract Gedanken data-structures is given below.

```
(UNION,EXP,CONSTANT,IDENT,FUNCTDES,LAMBDAEXP,CONDEXP,CASEEXP,
     BLOCK),
(CLASS,CONSTANT,(VALUE,VALUEDEN)),
(UNION,IDENT,PROGIDENT,INTRIDENT),
(CLASS,PROGIDENT,(STRING,SEQ,CHARCLASS)),
(CLASS,INTRIDENT,(NAME,INTCLASS)),
(CLASS,FUNCTDES,(FUNCTPART,EXP),(ARGPART,EXP)),
(CLASS,LAMBDAEXP,(PARAMPART,IDENT),(BODY,EXP)),
(CLASS,CONDEXP,(PREMISS,EXP),(CONCLUSION,EXP),(ALTERNATIVE,EXP)),
(CLASS,CASEEXP,(INDEX,EXP),(BODY,SEQ,EXP)),
(CLASS,BLOCK,(RDECLPART,SEQ,RDECL),(LDECLPART,SEQ,LDECL),
     (BODY,SEQ,EXP)),
(CLASS,RDECL,(LEFT,IDENT),(RIGHT,LAMBDAEXP)),
(CLASS,LDECL,(LEFT,IDENT),(RIGHT,SEQ,EXP)),
(UNION,PFORM,IDENT,SEQPFORM),
(CLASS,SEQPFORM,(BODY,SEQ,PFORM))
```

Record Creation and Testing

Three functions are specified which test records for set membership by searching through the abstract syntax, In the following definitions the abstract syntax is the sequence D.

```
"T ISR λ(X,S)
   IF S=INTCLASS THEN ISINTEGER X ELSE IF S=BOOLCLASS
   THEN ISBOOLEAN X
   ELSE IF S=CHARCLASS THEN ISCHAR X
   ELSE IF S=UNIVERSAL THEN TRUE
   ELSE SEARCH(D UL, λI(D I)2=S,
     λI IF(D I)1=CLASS THEN ISFUNCTION X
                AND X TYPE=S
      ELSE TUNION(X,TAIL TAIL D I),
```

```
        λ() GOTO ERROR);
    TUNION ISR λ(X,U)
        SEARCH(U UL, λI T(X,U I), λI TRUE, λ() FALSE);
    TSEQ ISR λ(X,S)
        X LL=1 AND SEARCH(X UL, λI NOT T(X I,S),
          λI FALSE,λ() TRUE);
```

T(X,S) accepts a record or primitive datum X and a set name S, and tests whether X is a member of the set denoted by S. TUNION(X,U) accepts a sequence U of set names, and tests whether the record or primitive datum X is a member of any of the sets denoted by the components of U. TSEQ(X,S) accepts a vector X whose components are records or primitive data, and tests whether X is a sequence in which every component is a member of the set denoted by the set name S.

The function M is used to create records. It accepts as argument a sequence of which the first element is a class-name C, and the remaining elements are field values. M searches the abstract syntax to ensure that the correct number of field values have been provided, and uses the set membership functions to ensure that each of the field values satisfies its range descriptor.

M has the following definition:

```
M ISR λX
    (C,V IS IF ISATOM X THEN X, () ELSE (X 1,TAIL X);
     SEARCH(D UL, λI (D I)1=CLASS AND (D I)2=C,
       λI M1(C,V,TAIL TAIL D I), λ(') GOTO ERROR));
M1 ISR λ(C,V,F)
  IF V UL ≠ F UL AND SEARCH V UL,
      λI NOT(IF (F I)2=SEQ THEN
        TSEQ(V I,(F I)3) ELSE T(V I,(F I)2)),
```

```
    λI FALSE, λ() TRUE)
  THEN λX IF X=TYPE THEN C ELSE
    SEARCH(V UL, λI X=(F I)1,V,λ() GOTO ERROR)
  ELSE GOTO ERROR;
```

Translation into Abstract Form

Before a Gedanken program can be converted into abstract form it must be parsed according to the concrete grammar of Gedanken.

"The syntax of concrete programs is defined in two stages: (1) A unique partitioning of the program into a sequence of character strings called tokens is specified, and then (2) The set of well-formed programs is defined by a grammar over the infinite vocabulary of tokens.

The tokens themselves satisfy the following grammar

⟨character⟩::=" |⟨quotable character⟩

⟨quotable character⟩::=⟨letter or digit⟩|λ|,|=|:|(|)|;|␣|
⟨extra character⟩

⟨letter or digit⟩::=⟨letter⟩|⟨digit⟩

⟨letter⟩::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

⟨digit⟩::=0|1|2|3|4|5|6|7|8|9

⟨token⟩::=⟨integer token⟩|⟨quoted string token⟩|⟨word token⟩|
⟨punctuation token⟩

⟨integer token⟩::=⟨digit⟩{⟨digit⟩}

⟨quoted string token⟩::="{⟨quotable character⟩}"

⟨word token⟩::=⟨letter⟩{⟨letter or digit⟩}

⟨punctuation token⟩::=λ|,|=|:|(|)|;|:=

Here the symbol ␣ denotes a blank, and the undefined class ⟨extra character⟩ denotes the set of all input-representable characters not occurring elsewhere in the syntax.....

....The class of ⟨word token⟩'s is subdivided into ⟨reserved word token⟩'s, which are the strings AND, OR, IF, THEN, ELSE,

CASE, OF, IS, and ISR, and ⟨identifier tokens⟩'s which are all other ⟨word tokens⟩'s."

Once a program has been partitioned into tokens it is parsed according to the grammar on the left-hand of table 1, and transformed into a derivation tree with the following properties:

"(1) Each node is associated with either a token or a production of the grammar in Table 1.

(2) A node associated with a token is a terminal node.

(3) If a node is associated with a production which has n items on its right side, then the node has n subnodes. If the ith item on the right side of the production is a specific token (token class, syntactic class), then the ith subnode is associated with the specific token (some member of the token class, some production whose left side is the syntactic class)....

....With each production of the grammar in Table 1 we associate a GEDANKEN function which expresses the translation of any phrase which is an instance of that production in terms of the translations of its sub-phrases.

This process can be described more precisely as follows: The nodes of the derivation tree are translated in some order such that no node is translated until after all of its subnodes have been translated. If a node is associated with a token, its translation is a sequence whose components are the characters of the token. If a node is associated with a production its translation is obtained by applying the corresponding translation function to a sequence whose components are the translations of the immediate subnodes, excepting those subnodes which are associated with reserved word or punctuation tokens. In the special case where this sequence has a single component, the component itself, rather than the one-component sequence, is

used as the argument to the translation function....

....When the right side of a production is ⟨empty⟩, the associated translation function receives an empty sequence as its argument. The following subsidiary functions and other values are used by the translation functions:

```
IICOUNT IS REF 0;
CRIDENT IS λ() IICOUNT:=INC IICOUNT; M(INTRIDENT, VAL IICOUNT));
EQUALCON IS M(CONSTANT, GETVALPREDEF M(PROGIDENT,"EQUAL"));
SETCON IS M(CONSTANT, GETVALPREDEF M(PROGIDENT,"SET"));
COERCECON IS M(CONSTANT, GETVALPREDEF M(PROGIDENT,"COERCE"));
CONVERTINT ISR λX IF X UL=0 THEN 0 ELSE
     ADD(DIGITTOINT X X UL, MULTIPLY(10, CONVERTINT HEAD X));
TRANSTRING IS λX IF X UL=1 THEN M(CONSTANT, X 1) ELSE
     (I IS CRIDENT(); M(LAMBDAEXP, I, M(CASEEXP, M(FUNCTDES,
      COERCECON,I), VECTOR(1, X UL, λJ M(CONSTANT, X J)))));
TRANSDECL ISR λ(P,E,B) M(FUNCTDES, TRANSLAMBDA(P,B),E);
TRANSLAMBDA ISR λ(P,B) IF T(P,IDENT) THEN M(LAMBDAEXP,P,B)
     ELSE (I IS CRIDENT(); K IS REF(P BODY) UL; R IS REF B;
     LOOP: IF K=0 THEN GOTO DONE ELSE
           R:=TRANSDECL((P BODY) VAL K, M(FUNCTDES, M(FUNCTDES,
              COERCECON,I), M(CONSTANT, VAL K)), VAL R);
           K:=DEC K; GOTO LOOP;
     DONE: M(LAMBDAEXP,I,VAL R));
TRANSEQUEXP ISR λX
     (S IS VECTOR(1,X UL,λJ CRIDENT()); I IS CRIDENT();
      K IS REF X UL; R IS REF M(LAMBDAEXP,I,M(CASEEXP,
      M(FUNCTDES,COERCECON,I),S));
      LOOP: IF K=0 THEN GOTO DONE ELSE
            R:=TRANSDECL(S VAL K, X VAL K, VAL R);
            K:=DEC K; GOTO LOOP;
      DONE: VAL R);
```

The global reference IICOUNT maintains a count of the number of internal identifiers which have been created during the translation process. It is used by the function CRIDENT(), which returns a distinct internal identifier each time it is executed."

M is a function such that M(c,a_1,....,a_n) will create and return a record with class name c and field values a_1,...,a_n.

"The values of EQUALCON, SETCON, and COERCECON are constants denoting the basic functions EQUAL, SET, and COERCE. The value fields of these constants are obtained from the function GETVALPREDEF (to be defined later), which produces the value denotations of predefined identifiers. The use of these constants instead of the corresponding identifiers insures that redeclaration of the identifiers will not affect implicit coercion or the meaning of the operations = and:=.

CONVERTINT converts a sequence of digits into the corresponding integer.

TRANSTRING translates quoted string tokens. If n=1, the token "$c_1...c_n$" is translated into the abstract equivalent of i (CASE i OF "c_1",...,"c_n").

The three interconnected functions TRANSDECL, TRANSLAMBDA, and TRANSEQEXP eliminate declarations, sequence parameter forms, and sequence expressions.Their effect is essentially equivalent to repeated application of the following equivalences

p is e; b=>(λ(p)(b))(e)

λ(p_1,...,p_n)b (when n=1)

=>λi(p_1 is i 1;...;$p_n$is i n; b)

e_1,...,e_n (when n=1)

=>(i_1 is e_1;...;i_n is e_n; λi(CASE i OF i_1,...,i_n))

where....i,i_1,...,i_n are unique internal identifiers".

The working of these three functions is not immediately

obvious, and might be clarified somewhat by stating that their ultimate effect is that expressions of the form:

p IS e; b => (λ(p)(b)(e) (as stated above)

$\lambda(p_1,p_2,\ldots,p_n)b$

$\Rightarrow \lambda I(\lambda p_1(\lambda p_2,\ldots(\lambda p_n\ b)I\ 1)\ldots.)I\ n-1)I_n$

$(e_1,e_2,\ldots,e_n)$

$\Rightarrow \lambda I\ 1(\lambda I\ 2(\ldots.\lambda I\ n(\lambda I\ n+1\ CASE\ I\ n+1\ OF\ (I\ 1,I\ 2,\ldots$

$I\ n))e_n)\ldots.e_2)e_1$

the components of each expression being similarly transformed as appropriate. Thus thses expression forms are transformed into nested λ-expressions, each of which takes a single argument.

Expressions involving =, AND, OR, or:= are also eliminated during the translation from concrete to abstract Gedanken, and explicit calls of COERCE are inserted for the implicit coercion performed by certain functions.

The Abstract Machine

The machine defining the interpretation of abstract Gedanken data-structures is fundamentally similar to the sharing machine, but diverges in some respects, and boasts a number of elaborations.

The construction of the machine is somewhat different, this being apparent in the abstract syntax for states. Each state of the Gedanken machine consists of an SECD state, together with two extra fields, a memory and an atomcount.

The memory is a sequence of value denotations, each of which specifies the value of an explicit reference. Each element may be accessed by the appropriate references by means of the integer-valued address-field contained in each explicit reference, giving the number of the memory component which specifies its value.

The atomcount is an imteger value giving the number

of atoms which have been created during program execution. Each atom has an integer name field, i being the name of the ith atom created, where i =atomcount+1 at the time of that atom's creation.

Another divergence from the sharing machine is that each dump-element consists not of an entire state, but merely of a control, stack and environment. The memory and atomcount are added to progressively throughout execution and are used in such a way that they do not need to be saved and reinstalled. The dump component may be omitted because a Gedanken machine dump consists of asequence of dump elements rather than a single element. When the transition rule is applied, the entire sequence may be passed to the new state, the sequence being used according to a stack discipline.

The transition rule used is based on that of the sharing machine. If P is the abstract form of the program, and a state takes the form (control, stack, environment, dump, memory, atomcount), then starting from the initial state

(unitseq p,(),(),(),(),0)

transition is applied repeatedly until a terminal state is reached, i.e. one in which the control and dump are both empty. The value of the program is then the first (and only) stack element.

As in the sharing machine, the first control instruction is considered for each application of transition. If the control is empty, the control, stack and environment of the first dump component are installed, otherwise a branch is made on the first control instruction.

The range of values that a control instruction may take greatly exceeds that of the sharing machine. The instructions may be broadly divided into three groups:-

(a) The classes of expressions which may be included in an

abstract program. In Gedanken these include the three basic AE's, and also constants, conditionals, case expressions, blocks, recursive declarations and label declarations.

(b) Control instructions created during execution to enable complex instructions to be broken down into a sequence of simpler instructions.

(c) Basic function instructions which effect the evaluation of the basic functions of Gedanken.

(a) Expressions

The evaluation of each expression-type is described in turn. However it should be mentioned that in some cases this evaluation is not completed in a single application of transition. In such cases one application of the transition function will break down the expression into a sequence of simpler instructions as mentioned in (b) above.

Constants

The value of the constant is put on the stack.

Identifiers

The value of the identifier is found using the GETVAL function, and put on the stack.

Function Designators (AE combinations)

The finction and argument parts are evaluated, and the value of the former is applied to the value of the latter.

Lambda Expressions

When the expression is evaluated, a closure comprising the λ-body and the current environment is loaded into the stack. Also included in the closure is an instruction to bind the parameter of the λ-expression to the value on top of the stack. Thus when this closure is subsequently applied, the body and control of the closure are installed as the current control and

environment, and the environment is extended by binding the parameter to its argument. The closure-body is then executed in this extended environment.

Conditional Expressions

In Gedanken the conditional expression is invariably of the form IF ⟨e_1⟩ THEN ⟨e_2⟩ ELSE ⟨e_3⟩. The premiss (e_1) is evaluated first, producing a Boolean result on the stack. Then either the conclusion (e_2) or alternative (e_3) is installed as the control and evaluated, depending on the Boolean result being true or false respectively.

This method of evaluation is exactly equivalent to the functional if e_1 ($\lambda()e_2, \lambda()e_3$)() expression suggested for conditionals in the AE/SECD system, with only the correct branch being evaluated each time.

Case Expressions

The index part is evaluated, and a check is made that its value is an integer not less than one, and not greater than the number of component expressions in the body. Then for an index of value n, the nth expression in the body is evaluated.

Blocks

A new dump element (comprising the current control, stack and environment) is put on the dump, and the stack is set to empty. This is very similar to the treatment of nested sub-expressions in the sharing machine. The control is then set to the following sequence

⟨recursive declaration⟩,....,⟨label declarations⟩,....,
⟨environment mk⟩,⟨exec (⟨block body⟩)⟩

Thus subsequently the recursive declarations, and then the label declarations will be evaluated, and the environment will be marked. The block body will then be evaluated in the resulting environment.

Recursive Declarations

It is in connection with recursive declarations and also label declarations that the environment is marked during block execution. The environment mark then delimits the environment available to the recursive functions and labels declared in that block.

When a recursive declaration of the form ⟨id⟩ ISR ⟨exp⟩ is evaluated, the environment is extended by an element binding the identifier part to a recursive function denotation for the λ-expression. A recursive function denotation has a control, but no environment field. When a recursive denotation is later yielded in a search of the environment, in order that it may be applied an environment field will be added, being that portion of the environment beginning at the last encountered mark. In this manner a function may refer to itself, without needing to use the fixed-point function suggested by Landin for the SECD and sharing machines.

Label Declarations

When the declaration is evaluated the current environment is extended by an element binding the label name to a recursive label denotation. This comprises the body of the innermost block containing the label and also the current dump. When this denotation is subsequently yielded in a search of the environment (i.e. when control is transferred to this label) then the current control, stack, environment and dump are discarded. The stack is emptied, the denotation body is installed as control, and the environment, as with recursive functions, becomes that starting at the last encountered environment mark. Thus even if control is transferred from outside the block containing a label, any declarations preceding the label still hold, and the entire state of the computation, apart from reference bindings and

atomcount, returns to that at the time of declaration of the label. This is effectively the 'program-closure' described by Landin for the sharing machine.

(b) The Control Instructions

These are created during program execution, and are used to instruct the machine in what manner instructions or instruction sequences are to be manipulated. Their effects are as follows:-

"EXEC: The sequence of expressions in the instruction body is evaluated, and the value of the last expression is added to the stack.

BRANCH: If the first stack component is TRUE (or FALSE) then the CONCLUSION (or ALTERNATIVE) is evaluated and its value replaces the first stack component.

SELECT: If the first stack component is an integer i, then the ith component of the instruction body is evaluated and its value replaces the first stack component. If the first stack component is the atom LL (or UL), then it is replaced by 1 (or the length of the instruction body).

BIND: The environment is extended by binding the identifier in the instruction body to the first stack component, which is deleted from the stack.

APPLY:The second stack component, which must be a function denotation, is applied to the first stack component, and the result of this application replaces the first two stack components.

MARKENV: A mark is added to the environment.

DELETE: The first stack component is deleted."

(c) Basic Function Instructions

These denote the basic functions and values of Gedanken. Their evaluation is effected with the use of two predefined sequences. One of these consists of all the

predefined strings of the language, while the other consists of the value denotations necessary to effect the meanings of these strings.

In the case of the strings "TRUE", "FALSE" and "QUOTECHAR" the value denotations are merely the appropriate values, TRUE, FALSE and ".

"LL", "UL" and "ERROR" are denoted by the fieldless records for lldenotation, uldenotation and errordenotation.

The remaining strings denote functions of zero, one or two arguments, some with implicit coercions and some without. The value denotation for each of these is a function denotation with an empty environment field, and whose sequence of control instructions consists of the appropriate basic instruction preceded by an instruction sequence termed a prelude. When the value denotation comes to be evaluated, this prelude is evaluated first and ensures that the correct argument(s), coerced as appropriate, and in the required order (last argument first) are on top of the stack when the basic function is executed.

Thus when an identifier cannot be found in any search of the environment, it is assumed that it must be a predefined string. It is searched for in the sequence of predefined strings, and if found the corresponding component from the sequence of value denotations is returned as its value.

The effect of evaluating each of the basic function instructions is described below. It is assumed that the arguments have already been spread and coerced by the prelude. In all cases an error condition will result if the arguments are not of the required type.

REF or NCREF

The argument is removed from the stack and added to

memory, and a new explicit reference is put on the stack, its integer valued address being the component number of the new memory element.

SET or NCSET

The first and second elements are taken from the stack, and must be an expression (e) and reference denotation (r) respectively. The execution depends on whether the reference is explicit or implicit.

(a) For an explicit reference, the memory component indicated by its address field is changed to the value of e, which is itself placed on top of the stack.

(b) For an implicit reference the following sequence is concatenated onto the top of the stack:-

(e, setf function of r, e)

Apply and delete instructions are added to the control so that subsequently the setf function will be applied to e and the result deleted.

Thus e is left on the stack top as the result for either type of reference.

Thus assignment is treated as just another basic function, and not as a fundamental expression type, which it is in the sharing machine system.

VAL

The first stack element, which must be a reference, r, is removed.

(a) If r is explicit, with address field n, then the value of the nth memory component is put on the stack.

(b) If r is implicit, its valf function is executed, and the result put on the stack.

COERCE If the stacktop is a reference, then VAL and COERCE instructions are put on the control as the first and second elements respectively. Otherwise nothing at all is done, i.e. the stack top is returned as a result. The effect is that VAL will be applied to the stack top and to subsequent results until a non-reference value is produced, and this value will then be the result of the coercion.

GOTO

The stack top must be a programmer-defined label denotation. The control, environment and dump fields of the label are installed, and the stack is set to empty. Execution then continues using these new values.

Predicate Functions ISINTEGER, ISBOOLEAN, ISCHAR, ISATOM, ISFUNCTION, ISREF, ISLABEL

The stack top is removed, and tested for membership of the appropriate class, and the appropriate Boolean result is put on the stack.

IMPREF

The first and second stack elements, which must both be functions, (f_1 and f_2) are removed. A new implicit reference denotation, using f_2 and f_1 as its SETF and VALF fields respectively, is created and put on the stack as a result.

EQUAL and NCEQUAL

The first and second stack elements are removed and the value TRUE is put on the stack if:-

(a) They are both members of the same one class from integer, Boolean, and character, and both have the same value, or

(b) They both denote LL or both denote UL, or

(c) They are both program atoms with the same name fields, or

(d) They are both explicit references with the same address

field - i.e. they both denote the same reference.

Otherwise the value FALSE is put on the stack.

GREATER(CHARGREATER)

The top two stack elements, which must both denote integers (characters) are removed and a Boolean result is put on the stack, being TRUE if the second element is the greater and FALSE otherwise. (N.B. an ordering is placed on the characters).

INC (DEC) The stack top, which must be an integer, is removed, and an integer result 1 greater (less) is returned on the stack.

READCHAR

A single character is read and the denotation placed on the stack.

WRITECHAR

The stack top must be a character denotation. The character that it denotes is written, and the denotation is left on the stack as a result.

CHAPTER 4 - AN IMPLEMENTATION OF GEDANKEN

In Reynolds' definition of Gedanken he gave an abstract syntax for the language, showed how to translate concrete Gedanken programs into abstract objects satisfying this syntax, and defined the semantics by means of an interpreter for the abstract programs.

It was decided to implement this formal definition on the IBM 360/44 computer at St. Andrews University, using BCPL as the defining language. The original definition was translated as faithfully as possible from Gedanken into BCPL, in order to preserve the correct semantic interpretation.

The Gedanken programs are interpreted in three passes:

(a) Lexical Analysis

A fairly standard lexical analyser is used, which accepts Gedanken program text, and translates each symbol into an internal integer representation. The output from this pass is a symbolic representation of the input text.

(b) Syntax Analysis and Translation into Abstract Form

The output of lexical analysis is parsed according to the concrete grammar of Gedanken, to build a syntax derivation tree, making use of a top-down 'fast-back' parsing algorithm. Concurrently with syntax analysis the derivation tree is translated into an abstract Gedanken data-structure.

(c) Interpretation of the Abstract Gedanken Program

The abstract program produced in the second pass is interpreted, using a translation of Reynolds' abstract machine, an elaboration of the Sharing Machine.

A more detailed description of each stage of the processing now follows.

(A) Lexical Analysis

The text of the Gedanken program is transformed during lexical analysis to the following effect:

1. Each complete textual symbol is replaced by an integer, the internal representation of the symbol. These integer representations, together with semantic information giving the values of constants and the names of identifiers, form the output string.
2. Blanks and quotation-marks are eliminated.
3. If an illegal character is encountered, a message to this effect is output. Thereafter the character is ignored, and the analysis proceeds with the next character.

The method of analysis is based on an algorithm described by Gries (ref 8). The scanner makes use of two tables:

1. The Character Class Table (Table C)

This assigns an integer class to all the characters which are permissible in the text of a Gedanken program. This class is used to decide on the interpretation to be put on each new symbol of the program, as it is encountered. The table is direct-access and is keyed by the EBCDIC representation of the character. Illegal characters are assigned a class of zero.

2. Table of Symbol Representations (Table Symdef)

This gives an internal integer representation for each of the 21 possible types of symbol in a Gedanken program. There is an entry for each reserved word or delimiter, and one each

for identifiers, quoted constants and digital constants. A symbol is located by a sequential search, its index in the table being its internal representation.

The procedure for the recognition of each symbol may be summarised into an algorithm:

1. Read up to the next non-blank character.
2. Obtain its class from Table C.
3. If the class is zero, print an error message and return to step 1. Otherwise:
   (a) If the character is a digit, continue to read characters until a non-digit is reached. Output the symbol for a digital constant and the values of the digits just read.
   (b) If the character is alphabetic, continue to read characters until something nonalphanumeric is encountered. Search Table Symdef to find out if the characters just read form a reserved word.
      (i) If so, output its internal representation.
      (ii) Otherwise output the symbol for an identifier and the characters which denote that identifier.
   (c) If the character is a quotation mark, continue to read until another quotation mark is encountered. Output the integer representation for a constant, and the characters composing it.
   (d) If the character is a colon, read the next character to check for the double symbol ":=". Output the appropriate integer representation, according to the result.
   (e) If the character is any other punctuation symbol, locate it in Table Symdef, and output its integer representation.

The procedure is repeated until the character "?" is encountered, which indicates the end of the input text.

Note In this implementation the character "£" is used instead of "λ", since the latter is not available.

(B) Syntax Analysis and translation to Abstract Form

Syntax Analysis

The syntax analyser accepts the output string of lexical analysis, analyses it according to the grammar of Gedanken, and produces from it a derivation tree representing the structure of the Gedanken program.

The parser is of the top-down, 'fast-back' variety, and like the lexical analyser is based on an algorithm described by Gries (ref 9). Gries' own terminology has been used in the following brief description of his algorithm.

The Parsing Algorithm

The algorithm may be described in terms of men whose job it is to build the derivation tree. At any time during the parse, there is a man standing on each node of the partial tree formed so far. The man at each node is responsible for the men on his sub-nodes. The man standing on the root node will be assigned the task of building the entire tree.

The man at the root begins by looking at the first derivation for the distinguished symbol of the grammar. If he is unable to build the tree from the input string using the first rule, he will try the second and so on. If none of the derivation rules work, then the input text does not form a correct program of the grammar.

To find out whether a derivation is correct, he will go through it sequentially, adopting for each component a son, whose job it is to try to find his allotted component. These sons in turn will adopt sons to find their own derivations.

Ultimately, the assigned goals will be actual program tokens, which must be matched with the symbols of the input string.

If a man receives a report from any of his sons that the son has tried all his derivations, but is unable to find his assigned goal, then the man will disown all his current sons and start adopting a new set according to the next alternative derivation.

A stack is used to build the derivation tree. Each stack element represents one node of the tree, and has the form

(goal, gind, fat, son, iind)

The components have the following meanings:

goal - the symbol for which the man is searching, assigned to him by his father

gind - the location in the grammar of the symbol in the derivation for his goal that the man is currently working on

fat - the location in the stack of the man's father, or zero for the root node

son - the location in the stack of the man's most recently adopted son

iind - the location of the symbol in the input string currently awaiting recognition.

In this implementation the algorithm has been extended to permit two additional features in the rules of the grammar:

(a) A string which may occur in a derivation any number (including zero) of times.

(b) A string which may occur in a derivation, but which may legitimately be absent.

The algorithm requires that the grammar be arranged in a certain way, so that the correct rules are tried first. For

example, a rule of the form

E::=T + E

must be tried before a rule

E::=T

Also it is necessary to devise a notation to indicate which parts of derivations are the 'repeatable' and 'optional' strings. The grammar of Gedanken, rearranged to suit these requirements, is read in immediately prior to parsing, and stored in core in the form of a tree. Each node has the form:

(name,def,alt,pred,need)

where the fields denote the following:

name - the internal name for the grammar symbol, S; represented by that node. For non-terminal symbols the name is a three-character abbreviation of the grammar element, packed into one machine-word. For terminals the integer internal representation of the symbol is used.

def - this field is zero if S is a terminal; otherwise it indicates the node for the first symbol in the first derivation for S.

alt - if S is the first symbol in a derivation, this field points to the node for the first symbol in the next alternative derivation. If no alternative exists, or S is not the first symbol in a rule then alt is set to zero

pred - set to zero if S is the first symbol of a derivation. Otherwise it points to the node for the preceding symbol in the rule.

need - set to 1 for all nodes in 'optional' parts of derivations, to 2 for nodes in 'repeatable' portions, and to zero for all other nodes.

Parsing proceeds as in the basic algorithm with the addition of certain rules made necessary by the extensions mentioned above:

1. If a man receives a report of failure from one of his sons, he must look at that son's need field, to see if he was an essential part of the derivation. If so, he proceeds according to the basic algorithm. If the son was in an 'optional' or 'repeatable' string, however, he disowns all sons in that part of of the derivation, and reports success to his own father.
2. When a man has received a report of success from every component of a derivation, he must check the need field of his youngest son to see if the end part of the rule forms a 'repeatable' string (these only occur at the ends of derivations). If this is so, he must return his attention to the start of this string andattempt to find another occurrence; otherwise he reports success to his father, as in the basic algorithm.

Another stack, TOPVALS, has been introduced to deal with these extensions. Every time the beginning of an optional or repeatable part of a derivation is encountered a new element is placed on this stack.

Each stack element has two components:

1. The level of the derivation tree stack before the start of that portion of the derivation.
2. The current man's youngest son before the optional or repeatable string, i.e. his last 'essential' son.

This stack operates as follows:

1. When a man adopts the first son of an optional or repeatable

string, an element is added to the stack.

2. If a son in an optional or repeatable string reports failure, the derivation tree stack returns to the level indicated in the top element of TOPVALS. All the man's sons younger than the one specified in the top TOPVALS element are disowned. This top element is now removed from the stack.

3. If an entire optional string is recognised, the top element of the stack will not be required, and is therefore deleted.

4. If the end of a repeatable string is recognised, so that attention must return to the start of that string, the top stack element is replaced by a new one giving the current information.

Modified thus, the basic parser may be used for the syntax analysis of a Gedanken program.

Gedanken Syntax Errors

Syntax errors are recognised in five ways:

(a) Each of the reserved words and delimiters may only appear in certain constructs of the grammar. Their recognition in effect identifies the construct which is being used. If a reserved word or delimiter is not followed by the expected element, then the syntax rules have been violated. A complete list of what must follow each symbol is given in Table II of the Appendix.

(b) The reserved words THEN and ELSE may be used only in conjunction with IF. Similarly OF may appear only after CASE.

(c) Conversely the word IF must always be followed by a construct of the form ..... THEN ..... ELSE, and CASE by ..... OF .....

(d) The parser might find a complete program structure before reaching the end of the input string. This is usually the result of the mismatch of parentheses.

(e) The converse of (d) may occur, i.e. the input string may be exhausted before a complete program structure has been built up. This is also likely to be due to a parentheses mismatch.

Types, of course, are not checked at this stage, and diagnostics for type errors are not produced until the program is interpreted.

On the discovery of a syntax error, the following actions are taken:

(1)A message is output to the programmer, stating what the error is, and the context in which it occurred. The context is given by writingout the last ten symbols of the program text prior to the point where the error occurred.

(2) The derivation tree stack is returned to the elvel of the last element whose goal was a block-2, the highest level of block. A block-2 thereby becomes the next element to be sought.

(3) The grammar index (gind) is set to point to a block-2.

(4) The pointer to the input string (iind) is moved to the symbol immediately following the next semi-colon, since these act as block seperators.

(5) Two flags are set:

(i) ERRFLAG is set to 'true', to indicate to the parser that a syntax error has been found, and that the normal parsing procedure is to be temporarily abandoned. This flag is reset to 'false' whenever a new component of a derivation is sought.

(ii) GOFLAG is set to 'false' once and for all when the first syntax error is encountered. It indicates that although parsing will continue, there will be no attempt to interpret the program, and the translation to abstract form should be discontinued.

In the case of an error of type (e), actions (2), (3) and (4) are unnecessary, and are omitted. This recovery procedure may give rise to parenthesis mismatches, and messages indicating these after the occurrence of other syntax errors are quite likely to be spurious.

Briefly then, if a syntax error is identified, the parser fails up to block level, and continues from there, while the translation to abstract form and program interpretation do not take place.

The Organisation of Space

The use of space is not organised according to a stack discipline, so a method is required for controlling the special storage area reserved for the abstract Gedanken data-structures. The correct amount of space must be allocated when it is needed, and released for re-use when a given data-structure is no longer required. The problem is handled in a fairly conventional manner.

All the free space in the record area is held in the form of a chain, the location of the first item of the chain being held in a variable, FREELIST. Subsequent items of the chain contain two items of information:

(1) The first word holds the number of contiguous machine words of free space starting at that location.

(2) The second word contains the location of the next link in the chain.

If n words of storage are required to create a record then the FINDSPACE function chains through the free-space until it finds an item n or more words long. The chaining is adjusted to exclude these words, and their location is returned as the result of FINDSPACE. If there is no space sufficiently large then garbage collection is necessary.

Garbage Collection

The second word of every record is a 'mark-word' used by the garbage collector. If set, it indicates that the data-structure in question is still in use, while if it is not set, that structure is no longer accessible and the space may be set free ready for re-use.

Garbage collection involves three main stages:

(1) The record space is searched sequentially and every mark-word is unset.

(2) All the records which are currently accessible are rooted in the current state. Any other data structures are inaccessible anyway, and the space they occupy is wasted. A complete traverse is made of the current state resetting all the mark-words, taking precautions to ensure that no part of the state is traversed more than once. This, however, does not protect quite all the records which must be saved. If garbage collection takes place in the middle of an execution of the transition function, those records and sequences created during that call will be vulnerable to garbage collection.

The problem is solved by using another stack, RECSTACK, upon which is placed the location of every record or sequence as it is created. The data-structures on this stack are also traversed, and their mark-words reset. At the end of each call of the transition function RECSTACK is emptied ready for re-use.

(3) The space is again searched sequentially, and the chain of free-space is re-composed, chaining around all records and sequences for which the mark-word is set, and linking up all areas with three or more contiguous free words.

If FINDSPACE looks again but is still unable to find a space sufficiently large, then program execution is terminated because of shortage of space.

Representation of the Abstract Gedanken Program

Before it can be evaluated, the parsed Gedanken program must be translated into an abstract form, the equivalent of the abstract ISWIM mentioned in Chapter 3. The abstract program forms an information-structure, amenable to evaluation by the interpreter.

An abstract Gedanken program is made up of sequences and records, where:

a sequence is a one-dimensional array

and

a record is a finite collection of fields, each identified by a field-name. Each record belongs to a particular class, such that all records in the same class have the same set of fields.

Representation of the Abstract Syntax

Before syntax analysis and the concurrent translation into abstract form, a representation of the abstract syntax of Gedanken is read and stored. It is held in core as a BCPL structure which mimics the Gedanken definition of the abstract syntax.

In order to represent the syntax as a BCPL structure, each of the identifiers occurring in the Gedanken definition is abbreviated to a unique sequence of three letters, which may be packed into a single computer word. The syntax is represented by a BCPL vector with 45 elements, each of which is the address of a structure representing either a class definition or a union definition.

Each definition is represented by a BCPL vector having one element for each element in the appropriate Gedanken sequence. Thus each element will be either a three letter encoding for an identifier, or a pointer to a further BCPL vector if that element

is a sub-sequence.

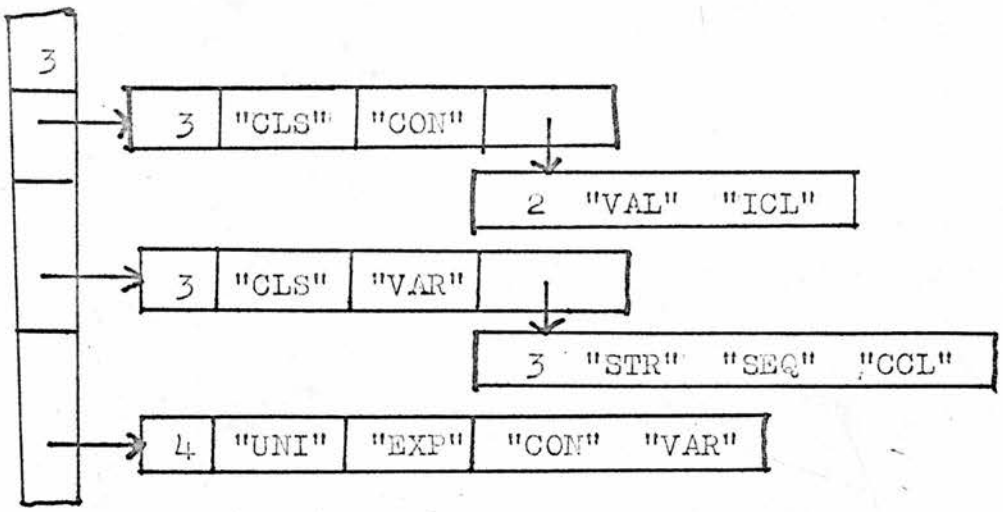
For example the grammar

(CLASS, CONSTANT, (VALUE, INTCLASS)),

(CLASS, VARIABLE, (STRING, SEQ, CHARCLASS)),

(UNION, EXP, CONSTANT, VARIABLE)

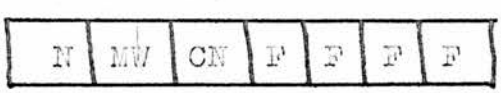
would be represented by the following BCPL structure:



The first word in each vector gives its length in words.

In Gedanken all data-structures are treated as functions, but this approach is not feasible in BCPL. The definition system has been copied as closely as possible, but records and sequences are held explicitly in BCPL vectors, in a special garbage-collected storage area. The functions for the manipulation of data-structures have been adapted accordingly.

A record is held in a BCPL vector as shown below:



N is an integer giving the number of computer words following it in that record.

MW is a mark-word, used for garbage collection.

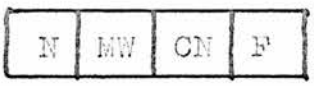
CN is the record class-name.

F is a field value.

Sequences are held in the same format as records,

but with a class-name of 'vec'.

Terminal values, i.e. Gedanken integers, Booleans and characters are held in four-word terminal records taking the form



The atoms LL and UL are simulated as follows:
LL always takes the value 1, and is held as a constant.
UL is found by deducting 2 from the value of N.

Records are created by a function M, which mimics the M in the Gedanken system. M accepts a class name, an integer n, and n-1 arguments; it reserves an appropriate amount of space in the free area, checks its field values using the set-membership functions, and constructs from them a new record of the given class.

The set-membership functions also mimic the T, TSEQ and TUNION functions of the Gedanken definition, searching through the abstract syntax to test that records are in a given class.

Translation into Abstract Gedanken

Associated with each production of the concrete grammar, there is an expression which specifies the translation of that phrase into abstract form, in terms of the translations of its subphrases. This makes it necessary that the nodes of the derivation tree be translated in such an order that no node is translated until after all of its sub-nodes have been translated. Such an ordering combines well with the parser used, and translation is performed concurrently with syntax analysis.

During parsing, whenever a node is successful in finding its assigned goal the appropriate translation is performed. A stack, PSTACK, is used to ensure that the correct

arguments are at hand, on the top of the stack, for the translation of each phrase. It is also used as a method of 'undoing' the relevant part of the translation when back-tracking occurs.

Each element of the stack has three components:

(1) The location of the node which will require the element as an argument of its translation function, i.e. the 'father' of the node producing the element.
(2) The name of the type of grammar element which produced the element.
(3) The location of the data-structure produced by the translation of the current node.

The order of recognition of the parser means that when a production is recognised, the translations of its sub-phrases will form the top elements of PSTACK. The arguments of the node are recognised by their first field, removed from the stack, and used by the translation function. The correct translation function is selected on the basis of the type of grammatical entity just recognised, and the number and type of arguments for it on the stack. The result of translation is then used to form a new element on top of the stack.

When back-tracking occurs it is merely necessary to remove any arguments for the failed node from the top of the stack. Any records or sequences created by the rejected portion of the derived tree are thereby cut loose, and made available for the next garbage collection.

The translation process has certain effects on the structure of the Gedanken program. Expressions involving =, AND, OR and :=, and also sequence expressions, sequence parameter forms and non-recursive declarations are eliminated by expressing

them in terms of other constructs. At this stage also, an additional class of internal identifiers is introduced for use during interpretation.

After translation labels are declared explicitly. They no longer appear in the block-body, but are included instead in a special label-declaration, which is paired with the list of unlabelled statements to be executed after a jump to the label.

Translation also inserts explicit calls of COERCE instead of the implicit coercion performed for certain grammatical forms.

Semi-Basic Functions

Certain of the functions used in Gedanken are not basic in a theoretic sense, but are used for convenience of programming. In Reynolds' system these functions are not accepted by the interpreter, but are defined in terms of basic functions. Concrete Gedanken programs are assumed to be enclosed in parentheses and preceded by the declarations of these functions, namely UNITSEQ, NOT, INTTODIGIT, DIGITTOINT, VECTOR, NEG, ADD, SUBTRACT, MULTIPLY, DIVIDE and REMAINDER.

However, in practice, this approach was found to be unworkable, since the time required to execute arithmetic operations defined entirely in terms of incrementing and decrementing by 1, was prohibitive. The arithmetic and type conversion functions were therefore incorporated into the interpreter as basic functions, and only UNITSEQ and VECTOR are declared before each program.

(C) Interpretation of the Abstract Program

The BCPL interpreter is modelled very closely on the Gedanken abstract machine. The different natures of the two

languages, however, make certain changes necessary.

The record creation and testing functions are translated as directly as possible into BCPL, and these translations search through the abstract syntax as do the Gedanken versions. However, since BCPLrecords are strictly data-structures and are not functional, it is not possible to apply them to field-names, and thereby select field values. This is done by means of a BCPL function, FIELD, which accepts a record R and a field-name F, and returns the value of field F in record R. It does so by taking the class of R, and then consulting the abstract syntax to find the position of field F in a record of that class.

The difference between functional and non-functional data-structures is also reflected in the translation into BCPL of the sequence manipulation functions, i.e. TAIL, CONS, AUG, REPLACE, CONC and SUBSEQ. The translated functions all create a new sequence of appropriate length, and copy into it the values of the appropriate components of the original sequence(s). Also there is no BCPL equivalent of the Gedanken sequence expression form. Where these occur in the definitional machine, a vector is constructed corresponding to the sequence expression form, and this is used as an argument to the sequence manipulation functions.

Transition

The transition function proceeds much as in the Gedanken machine, examining the first control element (henceforth referred to as X) of the current state, and replenishing it from the first dump element if the control is empty.

The class of X is determined by successively using function T to test it for each of the possible classes of control values. When its class is established the appropriate action is

taken.

The fields of the current state are stored as global variables at the start of each execution of the transition function. Certain of these will be changed during TRANSITION, and they will all be used as the fields of the new current state created at the end of the function.

CHAPTER 5 - A CRITICAL REVIEW OF THE IMPLEMENTATION

When the BCPL implementation was run on the IBM 360/44 computer at St. Andrews University it appeared to interpret Gedanken programs correctly. However it was too slow to be of use as a teaching tool. Even when full core (i.e. 256K bytes) was used, very simple Gedanken programs took a long time to run. For example it took about 4 minutes to r n a program to find factorial 3, and examples A and B included in the Appendix took 14 and 16 minutes respectively. Garbage collection was found to be necessary even for very short simple programs.

A major reason for this extreme slowness is an over-strict adherence to Reynolds' definitional system, which resulted in great inefficiency in the implementation. The definitional system is just that, and was not intended as the basis of an implementation. The situation is worsened in translation to BCPL by certain devices made necessary by the lack of functional data-structures in that language.

A useable implementation might be achieved by means of modifications which, while involving certain changes to Reynolds' system, would still preserve his intended interpretation.

1. A great deal of time is spent in searching the abstract syntax in order to test record class, create records and select fields. In fact this is unnecessary, since in the interpreter the M and T functions are always used with a known class argument and FIELD is always applied to a known field-name.

The general M function for record construction should be discarded, and the records should be constructed directly, both in the translation process and in the interpreter. The class of record required, and therefore the classes of its

component fields are always known, so that any type checks considered necessary could also be made directly.

Similarly there is no need for the general type-checking functions T, TSEQ and TUNION since the record type being tested for is specified throughout the interpreter. A direct test on the class-name is all that is necessary to find out whether a record is of a given class.

The FIELD function could also be dispensed with, since the record type and class name are always known. The appropriate component from the BCPL data-structure modelling the record could be selected immediately.

If these changes were made there would be no need to consult the abstract syntax at all. There would therefore be no point in storing it in core, so that a saving of 630 words of space in the free area would be made in addition to the saving of time.

2. Another major inefficiency lies in the modelling of all Gedanken sequences as fixed-length BCPL vectors. This is satisfactory for a sequence which will always be handled as a unit (such as the sequence of characters forming a program identifier) or on which the only operation is selection of a single component (such as the body of a case-expression). However it means that most sequence manipulation operations require the creation of a new vector, and the copying across of components from the old vector(s) to the new. This is clearly very inefficient for sequences of instructions, stacks and sequences of dump elements, since the principal operations on these are the addition or deletion of the first one or two elements. It would be much better to model such sequences as linked lists, and thereby avoid much copying and duplication of structures.

Thus, while identifier names, basic function instructions and the bodies of case expressions and sequence pforms could still be modelled as BCPL vectors, all other sequences specified in the abstract syntax would be better modelled as linked lists.

The head of each list should specify:

(1) The address of the first link in the list.
(2) The address of the last link.
(3) The number of links in the list.

All these fields are zero in the case of the empty list.

Each list element should specify:

(1) The address of the data-structure forming the body of that particular element.
(2) A pointer to the next element of the list.

The manipulation of such lists would require merely the creation of new list heads or the changing of pointers, rather than the time- and space-consuming copying of sequences. The amount of garbage-collection necessary would therefore be drastically reduced.

These two modifications are fundamental to the interpreter, and would greatly improve its efficiency. Further improvements could be made by a number of more minor changes.

3. Basic function instructions are carried around as sequences of characters. Considerable space could be saved by giving these integer names. Not only would such a representation be more compact, but also it would allow the interpreter to select the appropriate instruction using the BCPL equivalent to a case-statement, instead of matching each character sequence in

turn.

4. In the abstract form of Gedanken, an entire computer word is used to mark records for garbage-collection, which is wantonly extravagant of space. The first word of each record contains an integer giving the record length, for which two bytes would be more than adequate. One of the remaining two bytes could then be used as a mark-byte, at a saving of one word per record, sequence or list element.

5. The parser used involves back-tracking, albeit of a limited amount. This is undesirable, especially since the translation of the program into abstract form takes place concurrently with parsing. A parser which did not back-track, perhaps a precedence parser, would be more satisfactory. However the amount of time used for parsing is very small compared to that used for interpretation and the problem is of fairly low priority.

There are also certain changes which would have only a small effect on efficiency, but which would make the modified interpreter more elegant and more readable.

6. It would be possible to alter the abstract syntax slightly, without affecting the semantics of the language, so that a field with a given name is always in a given location within a record. This could be effected by rearranging the order of the fields in certain records, and renaming the fields in others. For example, many different record types in the abstract syntax have a field called 'body'. In a 'bind' or 'exec' class record this is the first field, in a 'λ-expression' record it is the second, etc. If these fields were renamed, for example 'body1', 'body2' etc. according to their position within a record, then they could be used directly as field selectors. In BCPL this

could be done by binding the field names to their component position using a MANIFEST declaration.

7. The major branch of the interpreter on the type of the first control instruction is effected by repeated testing against each possibility until a successful comparison is made. A direct branch could be made using the BCPL equivalent of a case-statement, by assigning integer class-names instead.

Each record class should be represented by an integer code rather than the present three-character string. These codes like the field names in 6 above, could be bound to the appropriate class-names for greater readability. Similarly an integer code could be used to identify sequences, and the heads and links of linked-lists.

Thereby one word could be saved on each of the above structures if the integer codes used are all less than 128. The code could be stored in a single byte, namely the remaining byte in the first word of each BCPL vector, as mentioned in 4 above. It is true that this suggestion and 4 would involve masking, since BCPL is a word orientated language. However, in the case of the class-names the extra time required for the masking would probably be little more than that used in the present string-handling. Also a considerable proportion of the time used is spent garbage-collecting, and the saving of two words per record should reduce this considerably.

REFERENCES

1. Reynolds, J. GEDANKEN: A Simple Typeless Language which Permits Functional Data Structures and Co-routines. Argonne National Laboratory.

2. Reynolds, J. GEDANKEN-a simple typeless language based on the principle of completeness and the reference concept. CACM 13, 5, 308-319 (1970).

3. Strachey, C. Varieties of Programming Language. High Level Languages. Infotech State of the Art Report 7.

4. Church, A., The Calculi of Lambda-Conversion, Princeton University Press, Princeton 1941.

5. Landin, P.J. A Formal Description of Algol 60. Formal Language Description of ALGOL 60. Formal Language Description Languages for Computer Programming, ed. T.B. Steel, Amsterdam. North Holland Publishing Company 1966 (266 - 294).

6. Landin, P.J. Correspondence between ALGOL 60 and Church's Lambda Notation. Parts I and II. Comm. ACM. Vol. 8 No's 2 and 3, 89 - 101, 158 - 165.

7. Landin, P.J. The Next 700 Programming Languages. Comm. ACM Vol. 9. March 1966. 157 - 166.

8. Gries, D., Compiler Construction for Digital Computers, John Wiley and Sons, 1971,64 - 71.

9. ibid., 91-2.

APPENDIX

Productions	Associated Translation Functions
<identifier> ::= <identifier token>	λX M(PROGIDENT, X)
<exp$_0$> ::= <integer token>	λX M(CONSTANT, CONVERTINT X)
| <quoted string token>	λX TRANSTRING HEAD TAIL X
| <identifier>	λX X
| (<block$_2$>)	λX X
<exp$_1$> ::= <exp$_0$>	λX X
| <exp$_0$><exp$_1$>	λ(X, Y) M(FUNCTDES, M(FUNCTDES, COERCECON, X), Y)
<exp$_2$> ::= <exp$_1$>	λX X
| <exp$_1$> = <exp$_2$>	λ(X, Y) M(FUNCTDES, EQUALCON, TRANSEQEXP(X, Y))
<exp$_3$> ::= <exp$_2$>	λX X
| <exp$_2$> AND <exp$_3$>	λ(X, Y) M(CONDEXP, M(FUNCTDES, COERCECON, X), M(FUNCTDES, COERCECON, Y), M(CONSTANT, FALSE))
<exp$_4$> ::= <exp$_3$>	λX X
| <exp$_3$> OR <exp$_4$>	λ(X, Y) M(CONDEXP, M(FUNCTDES, COERCEON, X), M(CONSTANT, TRUE), M(FUNCTDES, COERCECON, Y))
<exp$_5$> ::= <exp$_4$>	λX X
| IF <exp$_6$> THEN <exp$_6$> ELSE <exp$_5$>	λ(X, Y, Z) M(CONDEXP, M(FUNCTDES, COERCECON, X), Y, Z)
| λ <pform$_0$><exp$_5$>	TRANSLAMBDA
| <exp$_4$> := <exp$_5$>	λ(X, Y) M(FUNCTDES, SETCON, TRANSEQEXP(X, Y))

TABLE I.

Productions	Associated Translation Functions
<exp$_6$> ::= <exp$_5$>	λX X
| <empty>	TRANSEQEXP
| <exp$_5$> , <exp$_5$> {, <exp$_5$>}*	TRANSEQEXP
| CASE <exp$_6$> OF <exp$_5$> {, <exp$_5$>}*	λX M(CASEEXP, M(FUNCTDES, COERCECON, X 1), TAIL X)
<pform$_0$> ::= <identifier>	λX X
| (<pform$_1$>)	λX X
<pform$_1$> ::= <pform$_0$>	λX X
| <empty>	λX M(SEQPFORM, X)
| <pform$_0$> , <pform$_0$> {, <pform$_0$>}*	λX M(SEQPFORM, X)
<block$_0$> ::= <exp$_6$>	λX M(BLOCK, (), (), UNITSEQ X)
| <exp$_6$> ; <block$_0$>	λ(X, Y) M(BLOCK, (), Y LDECLPART, CONS(X, Y BODY))
| <identifier> : <block$_0$>	λ(X, Y) M(BLOCK, (), CONS(M(LDECL, X, Y BODY), Y LDECLPART), Y BODY)
<block$_1$> ::= <block$_0$>	λX X
| <identifier> ISR λ <pform$_0$><exp$_5$>; <block$_1$>	λ(X, Y, Z, W) M(BLOCK, CONS(M(RDECL, X, TRANSLAMBDA(Y, Z)), W RDECLPART), W LDECLPART, W BODY)
<block$_2$> ::= <block$_1$>	λX X
| <pform$_1$> IS <exp$_6$>; <block$_2$>	TRANSDECL
<program> ::= <block$_2$>	λX X

TABLE I (continued)

TABLE II

Syntactic Entities Required to Follow the Reserved Words and Punctuation Symbols

Reserved Word or Symbol	Required Successor
AND	expression 3
=	expression 2
OR	expression 4
ELSE	expression 5
IF	expression 6
THEN	expression 6
CASE	expression 6
IS	expression 6
OF	expression 5
:=	expression 5
ISR	λ
λ	parameter form
,	expression 5 or parameter
:	block 0
(	parameter form or block
)	any expression or block
;	block 2

```
UNITSEQ IS £X £I (CASE I OF X);
VECTOR ISR £(L,U,F) (L IS COERCE L; U IS COERCE U; F IS COERCE F;
    IF GREATER(L,U) THEN £I (I IS COERCE I;
    IF I = LL THEN L ELSE IF I = UL THEN DEC L ELSE GOTO ERROR)
    ELSE (V IS VECTOR(L, DEC U, F); T IS F U; £I (I IS COERCE I;
        IF I = UL THEN U ELSE IF I = U THEN T ELSE V I)));
  (
CONC IS £(X,Y) VECTOR(1, ADD(X UL, Y UL),
  £I IF GREATER(I,X UL) THEN Y SUBTRACT(I,X UL) ELSE X I);
CONS IS £(X, Y) CONC(UNITSEQ X, Y);
SUBSEQ IS £( L, U, X) VECTOR(1, INC SUBTRACT(U, L), £I X DEC ADD(I,L));
TAIL IS £X SUBSEQ(2, X UL, X);
FOR ISR £(A, B, F) IF GREATER(A, B) THEN ()  ELSE (F(A); FOR(INC A,B,F));
( Y IS REF 0;
V IS IMPREF(£X Y:=CONS(X,Y),£()(Z IS Y 1; Y:=TAIL Y; Z));
Y:="AB"; SET(V,"Z");
FOR(1, Y UL, £J WRITECHAR Y J); WRITECHAR VAL V;
FOR(1, Y UL, £J WRITECHAR Y J)))
??
 SIZE OF SSTRING=868
PARSE SUCCESSFULLY COMPLETED
Z A B Z A B END OF INTERPRETATION                                    00.35.24
/&
```

Example A

```
**********    ST.ANDREWS UNIVERSITY COMPUTING LABORATORY - 44MFT/SPOOLER END-OF-JOB RECORD
********** JOB NAME : MOSTEF     DATE : 74282      START TIME : 00.19.21   ELAPSED : 00.16.04
**********    CPU TIME : 00.14.18     WAIT TIME : 00.01.10     SYSTEM OVERHEAD : 00.00.20
**********    PAGES PRINTED :    56     CARDS PUNCHED :     0     CARDS READ :  2527
**********    A GRAND TOTAL OF          5715  INPUT/OUTPUT REQUESTS WERE HANDLED FOR THE JOB.
**********    THE AVAILABLE CORE WAS : 200 K     OF WHICH AT LEAST   61 K WAS USED
**********                    THE APPROXIMATE COST OF THIS JOB WAS  £  7.87
**********                  THIS JOB WAS EXECUTED IN THE BACKGROUND PARTITION
* THIS LINE PRINTED AT 00.40.03  ;
```

```
UNITSEQ IS £X £I (CASE I OF X);
VECTOR ISR £(L,U,F) (L IS COERCE L; U IS COERCE U; F IS COERCE F;
    IF GREATER(L,U) THEN £I (I IS COERCE I;
    IF I = LL THEN L ELSE IF I = UL THEN DEC L ELSE GOTO ERROR)
    ELSE (V IS VECTOR(L, DEC U, F); T IS F U; £I (I IS COERCE I;
        IF I = UL THEN L ELSE IF I = U THEN T ELSE V I)));
  (
ALIST IS ("CA","FS","PQ"); BLIST IS ("AF","SP","RH");
LOOKUP1 ISR £(Z,LIST,I)
      IF Z=(LIST I) 1 THEN (LIST I) 2
      ELSE IF I=LIST UL THEN NEG 1 ELSE LOOKUP1(Z,LIST,INC I);
( LOOKUP IS £(Z,LIST) LOOKUP1(Z,LIST,1);
      COUNT IS REF 1;
  FINDEND IS £Y
      ( X IS REF Y; LA IS REF 0; LB IS REF 0;
      A ISR £()(LVA IS VAL X; LA:=A1; X:=LOOKUP(LVA,ALIST);
      IF X= NEG 1 THEN (X:=LVA; GOTO A2)  ELSE ( IF LB=0 THEN B() ELSE GOTO LB);
      A1:  COUNT:=INC COUNT; A(); A2:);
B ISR £() (LVB IS VAL X; LB:=B1; X:=LOOKUP(LVB,BLIST);
      IF X=NEG 1 THEN (X:=LVB; GOTO B2) ELSE GOTO LA;
      B1: COUNT:=INC COUNT; B(); B2:);
A(); VAL X);
WRITECHAR FINDEND("C"); WRITECHAR INTTODIGIT COUNT))
??
 SIZE OF SSTRING=971
PARSE SUCCESSFULLY COMPLETED
Q 6 END OF INTERPRETATION
/&
```

01.01.09

Example B

```
**********************    ST.ANDREWS UNIVERSITY COMPUTING LABORATORY - 44MFT/SPOOLER END-OF-JOB RECORD
********************** JOB NAME : MOSTEF     DATE : 74282      START TIME : 00.42.45   ELAPSED : 00.18.24
**********************    CPU TIME : 00.16.38     WAIT TIME : 00.01.11     SYSTEM OVERHEAD : 00.00.19
**********************    PAGES PRINTED :    56     CARDS PUNCHED :     0     CARDS READ :  2532
**********************    A GRAND TOTAL OF         5725  INPUT/OUTPUT REQUESTS WERE HANDLED FOR THE JOB.
**********************    THE AVAILABLE CORE WAS : 200 K     OF WHICH AT LEAST   61 K WAS USED
**********************                    THE APPROXIMATE COST OF THIS JOB WAS  £  9.04
**********************                  THIS JOB WAS EXECUTED IN THE BACKGROUND PARTITION
* THIS LINE PRINTED AT 01.05.48  ;
```