Programming Languages.

PROGRAMMING LANGUAGES.

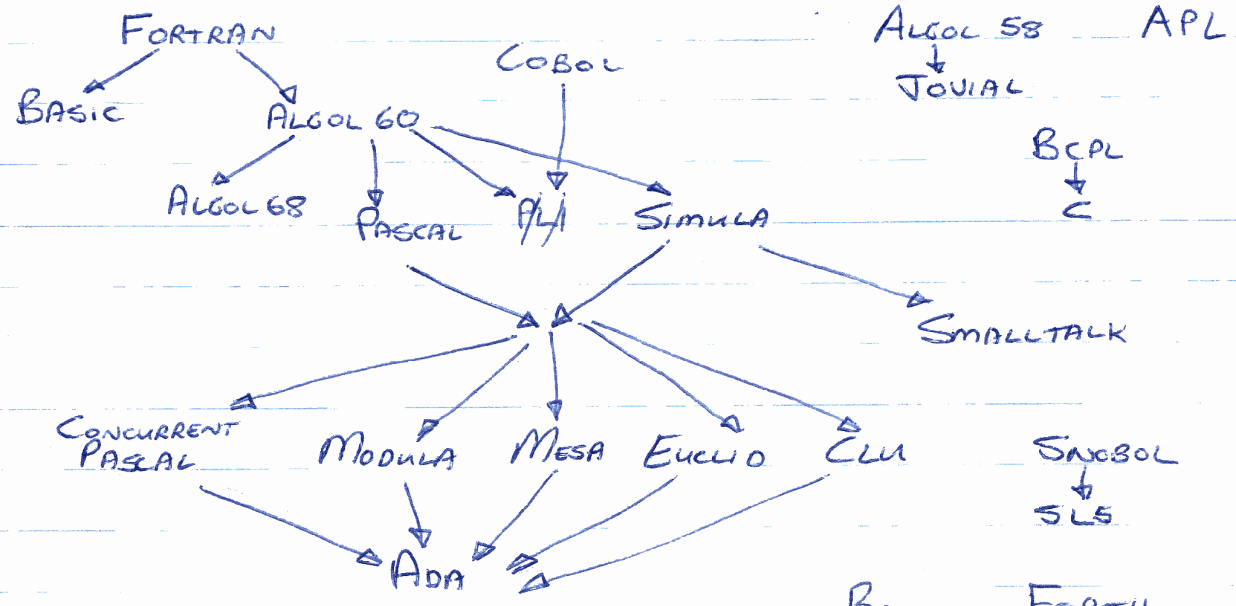
The same computational process or algorithm can be expressed in or mapped onto different symbolic structures or programs.

One important difference between symbolic structures is that between <u>imperative</u> & <u>applicative</u> programs. This is important enough to characterise two different classes of programming languages.

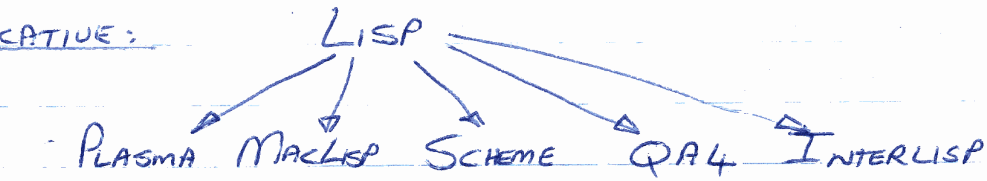
We will study & compare different ways of mapping computational structures onto symbolic structures.

1 LANGUAGE PEDIGREES

IMPERATIVE:



APPLICATIVE:



2 IMPERATIVE VS APPLICATIVE LANGUAGES

An imperative (highly sequential) language is one which achieves its primary effect by changing the state of variables by assignment. Collectively, the variables describe the state of the computation at any point in time. This concept is so widely accepted that many people cannot imagine doing it any other way.

An applicative language is one which achieves its primary effect by applying functions either recursively or through composition, the result of which are (hopefully) the answers we are expecting.

As an example, consider the Euclidean GCD algorithm. In FORTRAN, this could be:

NB FORTRAN is almost purely imperative.

LISP is purely functional (applicative)

PASCAL is imperative with some applicative features

```
1   IF (N.EQ.0) GOTO 2
    R = REM(M,N)
    M = N
    N = R
    GOTO 1
2   PRINT(M)
    STOP.
```

To make this applicative, we first rewrite it (SNOBOL structure)

```
1   IF (N.EQ.0) GOTO 2,3
3   R ← REM(M,N)    GOTO 4
4   M ← N           GOTO 5
5   N ← R           GOTO 1
2   PRINT(M)
```

We now convert it to functional transformations on all the variables:

$F_1(M,N,R) = (M,N,R)$ $\quad (N \text{ EQ } 0) \rightarrow F_2, F_3$

$F_3(M,N,R) = (M,N,\text{REM}(M,N)) \rightarrow F_4$

$F_4(M,N,R) = (N,N,R) \rightarrow F_5$

$F_5(M,N,R) = (M,R,R) \rightarrow F_1$

$F_2(M,N,R) = (M,N,R)$

And thus:

$F_1(M,N,R) = (N=0 \Rightarrow F_2(M,N,R),\ F_3(M,N,R))$

$F_3(M,N,R) = F_4(M,N,\text{REM}(M,N))$

$F_4(M,N,R) = F_5(N,N,R)$

$F_5(M,N,R) = F_1(M,R,R)$

$F_2(M,N,R) = M$

Finally, we get:

$F_1(M,N,R) = (N=0 \rightarrow M,\ F_1(N,\text{REM}(M,N),\text{REM}(M,N)))$

simplifying by removing the temporary variable R:

$F(M,N) = (N=0 \rightarrow M,\ F(N,\text{REM}(M,N)))$ ⟶

3 WHAT IS AN IDENTIFIER? (God knows what this is all about!)

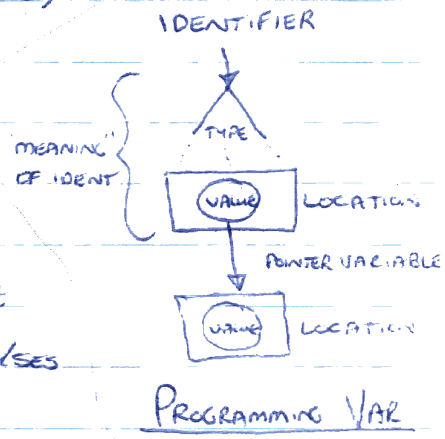
An expression has a structure (eg operator & operands, function & arguments). An identifier has no (significant) structure: it is used

- to hide or abbreviate structure (value identifiers eg 1, 2, 3)
- to abstract structure from structure (variable identifiers)

Note: 14 and 123 are expressions $14 = 1*10+4$ etc.
Also, in some systems (eg lambda calculus) operators such as * are also identifiers

Identifiers can (must?) be:

- introduced
- defined (or bound to a meaning)
- initialised (or bound to a value)
- used.



PROGRAM (SUBPROGRAM)

declaration part — Introduces & defines identifiers

statement part — INITIALISES & USES identifiers

TOKENS are identifiers with insignificant structure that are table-bound for their meaning. These include: standard identifiers, reserved words, special symbols, and literals. They can be programmer created (variables, procedure names, literals etc) or language created (reserved words & special symbols). Note that the structure of literals (constants of some data type whose value is given by the sequence of symbols - ie significant structure eg [], 3.1416, etc) is significant and their meaning is thus RULE BOUND, not table bound.

CONSTRUCTS contain parts with seperate significance

eg expressions (evaluations)
functions (evaluations, rules/abstractions)
definitions (bindings; environment) (declarations)
commands (transitions: state)
procedures (transition rules)
blocks (scopes)

Unlike constructs, we cannot work out the significance of a token by examining the individual symbols of which it is made.

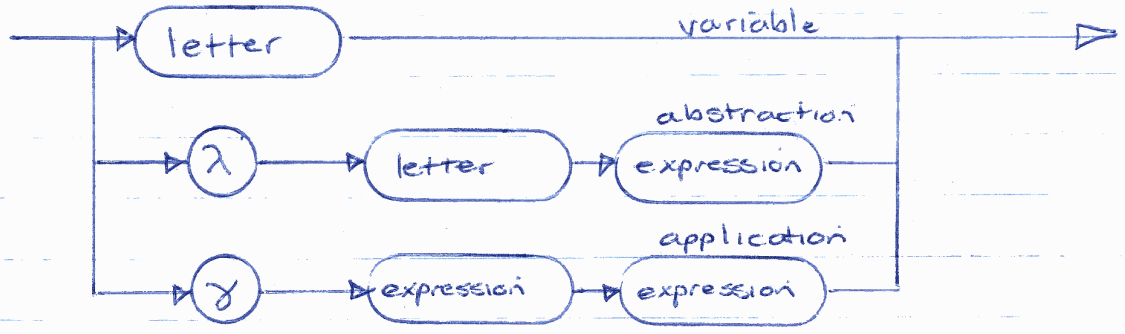
4 CHURCH'S LAMBDA CALCULUS.

The lambda calc. is a language for "pure" function evaluation. It has a minimum number of primitives and no explicit facilities for execution of a series of instructions. However, the idea of instruction execution is nevertheless firmly embedded in the language.

In the l.c., instruction execution corresponds essentially to substitution of values for bound variables. The l.c. is remarkably insensitive to the order in which instructions are executed (ie substitutions are performed), and serves to exhibit the basic structural characteristics underlying many programming languages.

The basic elements used in building expressions of the lambda calculus are called variables, and will be represented by lower case letters. The domain of variables of the lambda calculus is the set of all functions.

SYNTAX OF LAMBDA EXPRESSIONS.



Semantics of Lambda Calculus.

We classify variables as: binding, bound, and free

Binding variables immediately follow λ

A bound variable x is bound in an expression M if it is a binding variable, or if there is an expression M' of the form $\lambda x M''$ in M, where M'' includes x.

A free variable x is free in an expression M if it is not bound in M

A variable x is said to occur bound (free) in an expression M if M contains a bound (free) instance of x.

Expressions of the form $\lambda x M$ represent one-argument functions which determine a rule of correspondence between arguments A and values obtained by substituting A for all free occurrences of x in M.

Expressions of the form $\gamma F A$ are called operator-operand combinations and specify the application of the operator F to the operand A. If F is of the form $\lambda x M$, then we have the basic evaluation/reduction rule of l.c, ie:

$\gamma \lambda x M A$ is an operator-operand combination with an operator part $\lambda x M$ and an operand part A. Its value is obtained by substitution of A for all free occurrences of x in M

---

eg $\gamma\lambda x x, \gamma y z \rightarrow \gamma y z$

$\gamma\lambda x x \lambda x x \rightarrow \lambda x x$

$\gamma\lambda x \gamma x y \lambda z z \rightarrow \gamma\lambda z z y \rightarrow y$

$\gamma\lambda x \gamma x x \lambda x \gamma x x \rightarrow \gamma\lambda x \gamma x x \lambda x \gamma x x$, yielding a non-terminating reduction.

Note that the reduction rule can be applied to $\gamma\lambda x M A$ only if M contains no (unnes) bound occurrences of x, and provided A does not contain any free occurrences of variables bound in M. The following renaming rule can help resolve these naming conflicts:

Let M be any well-formed part of a lambda expression other than a variable following λ. Then if x is bound in M, M can be replaced by the substitution provided M contains no free occurrences of x and A does not occur in M

An expression to which no further reductions can be applied is said to be in <u>reduced form</u>. An expression for which there is no finite sequence of reductions resulting in a reduced-form expression is said to be irreducible (eg the last example above) It can be shown that reducibility is undecidable.

The l.c. can be interpreted as a function calculus.

Letters ≡ variables over the domain of functions

Abstractions ≡ definitions of functions

Applications ≡ applications of defined functions to argument functions to yield result functions.

ie, in $\gamma \lambda x M N$, a function defined as $\lambda x M$ where x is a formal parameter and M is the body of the function, is applied to an actual parameter N.

eg $* : \lambda a \lambda b \lambda c \; \gamma a \gamma b c$
$2 : \lambda a \lambda b \; \gamma a \gamma a b$
$3 : \lambda a \lambda b \; \gamma a \gamma a \gamma a b$

Then $3 * 2$ is evaluated with:
$\gamma\gamma \; \lambda a \lambda b \lambda c \; \gamma a \gamma b c \;\; \lambda a \lambda b \gamma a \gamma a \gamma a b \;\; \lambda a \lambda b \gamma a \gamma a b$
$\rightarrow \lambda a \lambda b \; \gamma a \gamma a \gamma a \gamma a \gamma a \gamma a b$

Notes:

① Renaming is necessary to prevent clashes (confusion), eg:
$\lambda b \gamma \lambda a \lambda b \gamma a b b \rightarrow \lambda b \lambda b \gamma b b$
If we had just used b, we would not know the ordering after reducing.

② ORDER OF EVALUATION:
We can use different orders of evaluation, and provided these terminate*, the results will fall in the same value class (equivalence class under renaming).
Left-to-right evaluation ≡ call by name
Right-to-left evaluation ≡ call by value

The CHURCH-ROSSER THEOREM states that all terminating evaluation strategies are equivalent.

* it is possible for one evaluation to terminate and another not.

③ $\gamma\lambda a\,\gamma\lambda b\,\gamma bb\,\lambda c\,\gamma a\,\gamma cc\;F$

$\rightarrow \gamma\,\lambda b.\gamma bb\,\lambda c\,\gamma F\,\gamma cc$

$\rightarrow \gamma\,\lambda c.\gamma F\,\gamma cc\,,\lambda c\,\gamma F\,\gamma cc$ (say x)

$\rightarrow \gamma F\,\gamma\,\lambda c.\gamma F\,\gamma cc\,,\lambda c\,\gamma F\,\gamma cc$ $(= \gamma F x)$

$\rightarrow \gamma F\,\gamma F\,\gamma\;\lambda c\,\gamma F\,\gamma cc\;\lambda c\,\gamma F\,\gamma cc$ etc. $(= \gamma F\,\gamma F x)$

So we have: $x \rightarrow \gamma F x \rightarrow \gamma F\,\gamma F x \rightarrow \ldots\ldots$

x is a FIXED POINT of F.

This gives us <u>recursion</u>

For $F = \ldots F \ldots$ or $F = \gamma\,\lambda g(\ldots g \ldots)F$?

write $F = \gamma\,Y\,\lambda g\,(\ldots g \ldots)$

where Y is a FIXED POINT FINDER.

NB Note composition / recursion / non-termination.

5. GENERAL PURPOSE MACROGENERATOR.

GPM is a symbol-stream processor which scans sequences of symbols residing in its input stream from left to right, normally copying them to an output stream but performing special action when certain symbols are encountered. The special actions in which we are particularly interested are macro-calling and macro-definition.

Macros in GPM are essentially arbitrary functions. A macro call results in the use of a previously defined macro defn.

The notation used is explained overleaf.

- [ ] delineate macro calls
- [, , ,] delineate parameters. Parameter 0 is the macro name
- [DEF, x, y] calls the MACRO DEFINITION FACILITY (ie a system defined macro DEF), which stacks a new macro with macroname x and macrobody y.

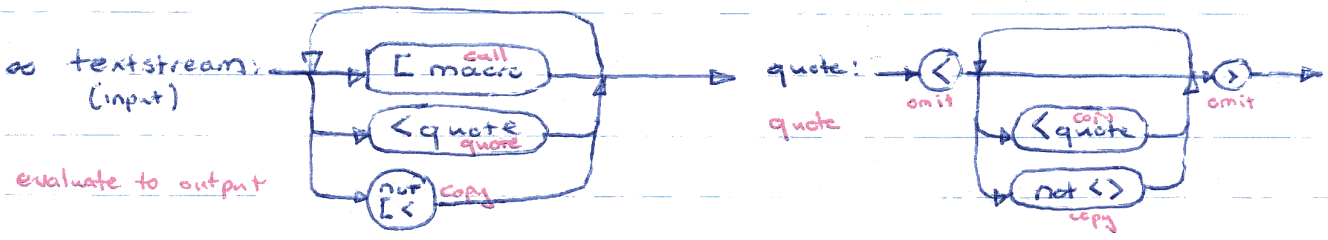
- A macrocall is an instruction to be replaced by the macrobody most recently bound to the macroname. This is called macroexpansion. In a macrobody the symbols #n act as formal parameters and are replaced by the n^{th} actual parameter of the macrocall.

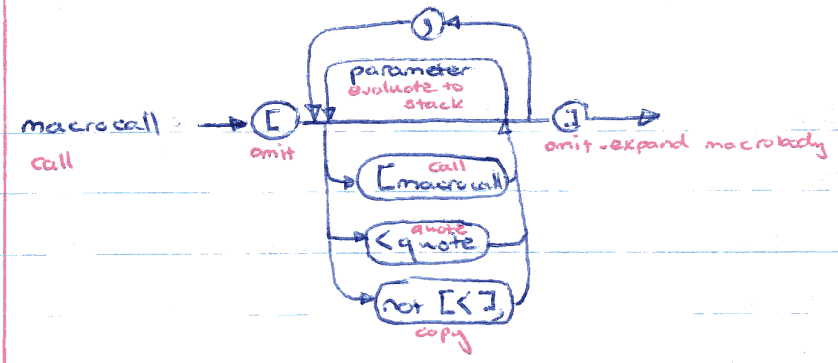
- < > delineate quotations, the function of which is to distinguish symbols which are merely mentioned (ie data) from those which are used (ie programs)

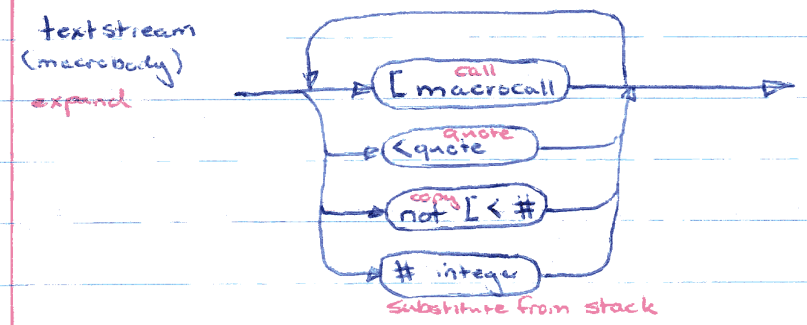
SYNTAX

{ } represents 0 or more repetitions. break(-) signifies anything other than -

textstream → { break(<[) | quote | macrocall }
quote → < { break(<>) | quote }
macrocall → [parameter { , parameter }]
parameter → { break([,]<) | quote | macrocall }
formalparameter → # digit { digit }







SEMANTICS.

Treat each syntactic unit of a textstream as an expression to be evaluated.

① A textstream is evaluated by simply copying.

② A quote is evaluated by omitting the opening <, copying the enclosed text without further evaluating it, and omitting the closing >

③ A macrocall is evaluated by:

(a) evaluating each parameter in turn to a temporary stack of parameters numbered from ① upwards

(b) applying an environment or definition table to parameter text ① (the macroname), and deriving a replacement textstream (the macrobody).

(c) evaluating the macrobody, adding this rule: replace every formal parameter #n with a copy of parameter text n from the temporary stack.

The output stream is the evaluation of the input stream.

Macrocalls can occur textually within parameters, and thus nested within other macrocalls. Entry into any level of static textual nesting of this sort is deemed to cause, at evaluation time, entry into a further level of dynamic evaluation nesting. Thus there must be provision for a stack of temporary stacks of parameter texts. Exit from the textual nesting causes exit from the evaluation level, passing the value obtained (ie the textstream produced) to the enclosing level, and loss of all sub-evaluations and sub-values derived in the course of the evaluation.

The GPM environment or definition table contains 6 initial system definitions:

DEF, UPDATE yield as macrobody the null string, but have a side-effect on the definition table for the current and all nested levels of evaluation. DEF extends & UPDATE overwrites the defn-table so that when it is applied to parameter text 1 of DEF (the macroname), it yields parameter text 2 of the call of DEF (the macrobody).

[VAL, X] yields the unevaluated macrobody currently named X

BIN, DEC, BAR - used for arithmetic.

Macro-evaluation may be thought of as a process of scanning symbols in a source stream and moving them to a target stream. The identity of the source and target streams and the processes in between are determined by the scanning mode

① Copy mode (from input to output streams)

② Macro-expansion mode (when] encountered - source stream is macro-body in defn table, target stream could be output stream, run-time stack or defn table).

③ Macro-definition mode: (when DEF is encountered - source stream is input stream / run-time stack / defn table, target stream the defn table)

④ Parameter-evaluation mode (when [encountered in source stream - source stream may be input stream / run-time stack / defn table, target stream is macro-expansion stack).

⑤ Parameter substitution mode (when # encountered during macro-expansion, causes switching of the source stream from the macro-defn table to an actual-parameter stream in the run-time stack. The target stream remains the same)

⑥ Quote mode - (when < is encountered while in one of the other modes. Copies source → target, ignoring all control symbols other than matching quotes. Exit from quote mode occurs when a > matching the opening quote is encountered.)

Simple Examples.

[DEF, N, <A #1 #2 B>] [N, C, D] → ACDB

[DEF, P, <A [N, C, D] #1>] [P, F] → AACBDBF

[DEF, X, <A [DEF, Y, #1] >] [X, B] → A and defines the parameterless macro Y with body B.

[DEF, X, A [DEF, Y, B]] defines macro X whose body is a value A and a macrodefn Y. The macrodefn Y is temporary and is deleted when the definition of X is completed. However, temporary macros defined in parameter strings permit local values to be associated with symbols during a macro expansion followed by a reinstatement of a previous value when macroexpansion is complete.

GPM evaluation

[DEF, PRED, <[?, Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9, [DEF, ?, <#>#1]]>]

MACROCALL evaluate & stack parameters • evaluate macrobody (= ENV(paramØ)) to calling text (substituting for #n from parameterstack) • pop parameters and local macro definitions

PARAMSTACK : Ø DEF 1 PRED 2 [?,Ø,1,2,3,4,5,6,7,8,9,[DEF,?,<#>#1]] ••
(DEF: text, PRED: text, 2: quotation)

DEF system macro with special effect : place new macro on macro-stack

MACROSTACK : ... PRED = [?,Ø,1,2,3,4,5,6,7,8,9,[DEF,?,<#>#1]]

outputs nullstring

PARAMSTACK :

[PRED, 6]

MACROCALL

PARAMSTACK : Ø PRED 1 6 ••
(PRED: text, 6: text)

ENV(PRED) = [?,Ø,1,2,3,4,5,6,7,8,9,[DEF,?,<#>#1]]

MACROCALL

PARAMSTACK: Ø PRED ... •• Ø ? 1 Ø 2 1 3 2 ... 10 9 11 •
(? ... : text ...)

[DEF, ?, <#>#1]

MACROCALL

PARAMS: Ø ... •• Ø ... • Ø DEF 1 ? 2 # 6 •
(DEF: text, 1: text, 2: quotation + substitution, 6)

DEF

MACROSTACK : ... PRED = ...

outputs nullstring ? = #6

PARAMSTACK : Ø ... •• Ø ? ... 11 •

ENV(?) = #6

outputs 5

PARAMSTACK : Ø PRED 1 6

MACROSTACK : ... PRED = [?,Ø,1,2 ... 9,[DEF...]]

outputs 5

PARAMSTACK :

MACROSTACK : ... PRED = [?,Ø,1,2,3,4,5,6,7,8,9,[DEF,?,<#>#1]]

GENERAL.

GPM is a function language which allows the user to write in abbreviated form anything which involves considerable repetition of certain patterns. A repeated pattern can be given a name viz

[DEF, name, < pattern >]

The quotes <> may be omitted if they serve no useful function, but are necessary if, for example, the pattern contains a].

From the function language point of view, the structure [X, A, B, C] is equivalent to the structure X(A,B,C), and except when X is DEF can be read as 'function X of A, B and C'.

eg [A, B, [DEF, A, < Q#1 >]] "A(B) where A(x) is Q#1."

Local procedures can be effected by quoting - including the definition within the call - no effective difference, but more elegant.

In "One Man went to Mow", [REST, n] is defined as

[VERSE, n] [n]

The call of n contains the definition of n, again more elegant.

[REST, 2, [def, REST, < VERSE, #1 > [#1, [def, #1, < . [REST, [S, >#1 <]]>] [def, 9, <...>]]>]]

[n] is defined in [REST, n] as [REST, succ(n)]

This is a parameterised definition, where the actual parameter is the first parameter of REST. By defining [n] in [REST, n] instead of [REST, n] := [VERSE, n] [REST, succ(n)] we allow alternative definitions each time n is called.

Quoting is used to delay evaluation of the quoted expression till calltime. Functional composition can thus be achieved by quoting inner calls in the definition of outer macros.

Recursion by inner calls (direct or indirect) of outer macros - exit must be provided by conditional calls. These are done by including alternative definitions of inner macros in the definition of an outer macro ~~[0] [1] [2] [REST, succ[n]] for 2≤n≤9~~ together with a quoted call of a macro whose name is a parameter of the outer call and can be made to vary over the set of alternatives provided.

In IMNTM, [n] = [REST, succ[n]] for 2≤n≤9
[9] = [...] so succ[9] is undefined and termination is provided.

6 Markov Normal Algorithms.

Syntax: <algorithm> → { <production> }

<production> → <string>^{t} → <string>^{r}

<string> → { <symbol> }.

Semantics.

A production specifies a transition on a datastring: replace the leftmost occurrence of string t in the data by string r.

An algorithm specifies alternative productions to be applied to the datastring in order until one of them succeeds.

Apply the algorithm to the datastring repeatedly until no alternative production applies. Halt on halt symbol.

Examples

- Remove zeroes $\Phi \rightarrow$
- Prefix single zero $\rightarrow \Phi$.
- Remove leading zeroes $*\Phi \rightarrow \mid * \rightarrow . \mid \rightarrow *$
- Postfix single zero $*x \rightarrow x \mid * \rightarrow \Phi . \mid \rightarrow *$?
- Reverse a string $**x* \rightarrow x** \mid ** \rightarrow . \mid *xy \rightarrow y*x \mid \rightarrow *$
- Duplicate a string

<COPYRIGHT> $*xy \rightarrow y*x \mid *x,y \rightarrow ,y*x \mid * \rightarrow , \mid -x \rightarrow x-*x \mid x, \rightarrow x \mid -, \rightarrow . \mid \rightarrow -$

<COPYLEFT> $*xy \rightarrow y*x \mid ,x \rightarrow x*x, \mid * \rightarrow \mid , \rightarrow . \mid \rightarrow ,$

- Identity function $\rightarrow .$ 'Undefined' function $\rightarrow$
- Concatenate $, \rightarrow$
- Behead $x \rightarrow .$ Retail $xy \rightarrow$

7. EVALUATION, CONTROL & ACCESS STRUCTURES

"The meaning of a sentence must remain unchanged when a part of the sentence is replaced by an expression having the same meaning" FREGE 1892 <referential transparency>

EVALUATION is modelled by "gradual replacement" ie step by step. We can have a general rule of replacement laid down on all strings, eg λ-calculus modelling the function 'evaluate', with particular rules of replacement fitted to particular domains of strings (eg Markov A's). The latter is an abstraction of programs from data, allowing 'types' of data and operations.

CONTROL structures are developed from program parts. (eg loop constructs are a middle way between MA's and GOTO programming (SNOBOL)).

ADDRESSING or ACCESS structures are developed from data parts (as imposed by the program), showing increasing independence of EVALUATION structure

- by PATTER MATCHING (MA's, SNOBOL, GPM)
- by LIST SELECTOR functions (LISP)
- by INDEXING functions (cf machine addressing)

8 STORAGE MANAGEMENT.

Environment : program / data available to current module.

Contrast: IDENTIFIERS = program variables = Syntactic entities

LOCATIONS = Storage variables = Semantic entities

IDENTIFIERS	LOCATIONS.
• appear in a program, and are statically bound to their denotation within the scope of their declaration or binding construction	• are in general computed (array indexing, pointers) and may even be <de> allocated dynamically
• identifier denotations "nest" so that on leaving an inner scope the outer environment reverts.	• the contents of a location may be irreversibly updated at any point.
• the class of values denotable by identifiers is (typically) wide: locations, arrays, procedures, labels, switches, strings, call-by-name parameters (ALGOL 60)	• the class of values storable in locations is (typically) narrow: booleans, integers, floating point (ALGOL 60).
• The environment 'of' a procedure is the environment of its definition (?) (extended by parameter bindings)	• the store 'of' a procedure is the store current at call time, and the store resulting is passed back to the point of call.

8.1 # PARAMETER PASSING, IDENTIFIER BINDING & SCOPE.

The binding time of an identifier is the time at which the association is made between the identifier and its allocated storage. Thus FORTRAN variables are bound at compile time, and PASCAL variables during execution at procedure entry.

Each time a routine with parameters is called, the formal parameter identifiers of the routine are bound to the actual parameter values given by the program that called the routine. Various binding rules are possible - in general, the called procedure specifies the binding rule, so it would be more accurate to say "pass-by-<rule>" than "call-by-<rule>".

① CALL-BY-TEXT : ?

② CALL-BY-NAME : This allows the parameter to be used for both input and output. The expression in the actual parameter position is reevaluated each time the formal parameter is used.

③ CALL-BY-REFERENCE : This also allows both input & output. The address of the actual parameter is determined when the routine is called. If the actual parameter is an expression, the value of the expression is stored in a temporary local variable and the address of that variable is used. Whenever the routine accesses a formal parameter, the variable passed as an actual parameter is accessed directly (eg PASCAL var)

④ CALL-BY-VALUE : Input only, not output. The compiler creates a local variable for each parameter, identified by the formal parameter name. When the routine is executed, the actual parameter supplied by the caller is evaluated, and its value is assigned to the local variable. Whenever the routine references the formal parameter, the local variable is used: the routine may not alter the actual parameter (in some

languages, even the local copy of the parameter may not be altered).

⑤ CALL-BY-RESULT: This allows the parameter to be used for output, but no value for the parameter may be assumed at the beginning of the procedure. The final value of the local variable is copied into the actual parameter at the end of the procedure.

⑥ CALL-BY-VALUE-RESULT: Allows parameters to be used for both input and output. This is much like call-by-value, except that the local temporary value in the procedure is modifiable, and its final value is copied back into the actual parameter at the end of the routine.

Early binding is more efficient, late binding more flexible. Change in a variable's value changes its attributes, eg. size.

static storage (eg FORTRAN)
↓ dynamic arrays

stack-based storage (eg PASCAL) NB RECURSION
↓ recursive data structures

heap based storage - explicit (eg PASCAL)
heap based storage - implicit (eg LISP)

Complete program-data equivalence (eg 'structure' in labels, identifiers eg SNOBOL) involves inability to distinguish 'compile'-time from 'interpret'-time absolutely.

We can divide programming languages into those in which binding can be done by the compiler (static binding) and those for which binding must be done while the program is running (dynamic binding)

FORTRAN uses a simple static binding scheme. FORTRAN has no blocks, so the only subprograms are the main program and its subroutines and functions. With the exception of COMMON data and subroutine formal parameters, all identifiers in a subprogram are local, and the FORTRAN compiler will allocate storage for each identifier at the end of the subprogram, as well as generating direct references to storage for the identifiers. Parameters in FORTRAN are passed by reference; identifiers in COMMON are declared in a COMMON statement which also determines their offset relative to the beginning of a block of COMMON data. After processing by the link editor, direct references to these can be made.

ALGOL also uses static binding, but here the rule used to resolve identifier uses is more complicated than in FORTRAN. In an ALGOL program, both blocks (beg.n ... end) and procedure definitions can occur, but any two such subprograms must either have no overlap or be nested, one wholly inside the other. Thus for each subprogram S we may define an environment consisting of a sequence of subprograms $S_1, S_2, \ldots, S_n$ where $S_1 = S$, S_2 is the subprogram most closely surrounding S_1, etc. A subprogram may use identifiers declared by any of the subprograms in its environment. When searching for the identifiers, S_1 identifiers are checked first, then S_2, etc. This is the "MOST CLOSELY NESTED" rule: An identifier d mentioned at some point in the program is bound to the declaration of d in that subprogram which declares d and most closely surrounds the point in the program.

Languages such as SNOBOL, APL and LISP cannot have static binding. In dynamic binding, the rule is the "MOST RECENTLY INITIATED" rule - A use of identifier d refers to the declaration of d in that subprogram which was most recently initiated (prior

to this use of a) and which has a declaration of a'.

In ALGOL60 we need a statically allocate region for own variables, plus a stack of activation records for other variables.

In PASCAL, storage is similar to ALGOL60, except records can be allocated dynamically by the creation of pointer variables. A heap is used to implement them.

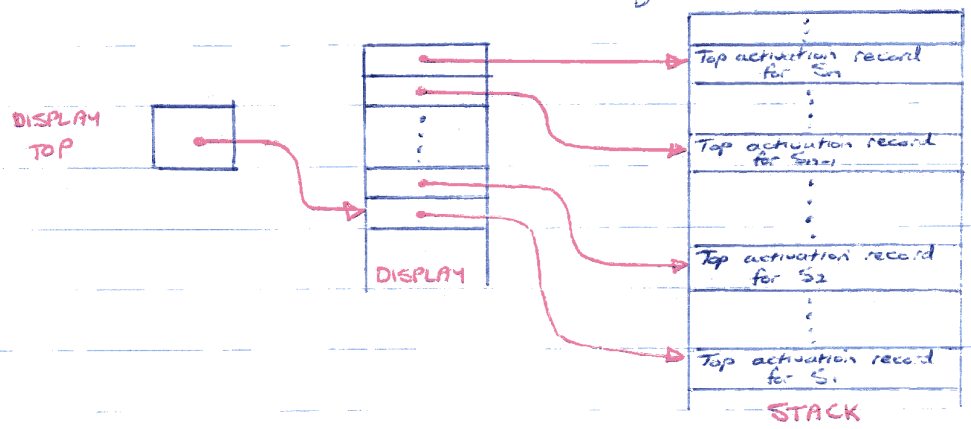
8.2. Recursion & Stack Allocation of Storage.

Suppose a programming language has the (usual) property that if subprogram S_1 is initiated, and subprogram S_2 initiates after S_1 begins, but before S_1 ends, then S_2 will end before S_1 does. Suppose that a block of memory will be used to hold all the data used by the entire program. The block will be treated as a stack of unequally sized records. Each time a subprogram S is initiated, a record, called the activation record, for the data of S is pushed onto the stack. When S terminates, its activation record is popped off the stack. Our first assumption assures us that the activation record for S will be on top of the stack when S ends, and further we shall not terminate any subprogram T whose activation record is below S's on the stack, since T must have initiated before S.

One advantage of stack allocation is that subprograms can share the same memory. It is exactly the environment of a subprogram S whose activation records appear on the stack when S is in execution.

Recall that in a programming language using static binding under the "most closely nested" rule, we can associate with each subprogram S_i an environment $S_1, S_2, \ldots, S_n$ consisting of all those subprograms to whose local variables statements of S_i could refer. When executing S_i, at least one activation record for each of these subprograms will appear on the stack. There may be more than one activation record for those S_i that are recursive, but the top activation record for S_i is above the top activation record for S_2, etc.

Any reference to a local identifier of S_i by S_i refers to the copy of that identifier in the top activation record for S_i. It is therefore only the top copies of the activation records for the S_i's whose data can be referenced and whose location must be known. We may therefore keep a stack of pointers to the top activation record for each subprogram in the environment. This stack is called a display, and can be represented schematically:



Accessing the needed data using displays is not hard. We must now consider how the display is to be modified when a new subprogram initiates and when an old one terminates. There are two cases to consider, depending on whether the subprogram is a block or a subroutine. Since the block cases is simpler, we consider it first. Upon entering a new block within the current block,

we push the block activation record onto the stack and a pointer to the A.R. onto the display. When the block terminates, we simply pop off the display.

If instead of entering a new block, S_i calls procedure P, the environment of P may not include all of $S_1, S_2, \ldots, S_n$. According to ALGOL rules P will be defined immediately within subprogram S_i for some i between 1 and n. The environment of P will be $P, S_i, S_{i-1}, \ldots, S_n$.

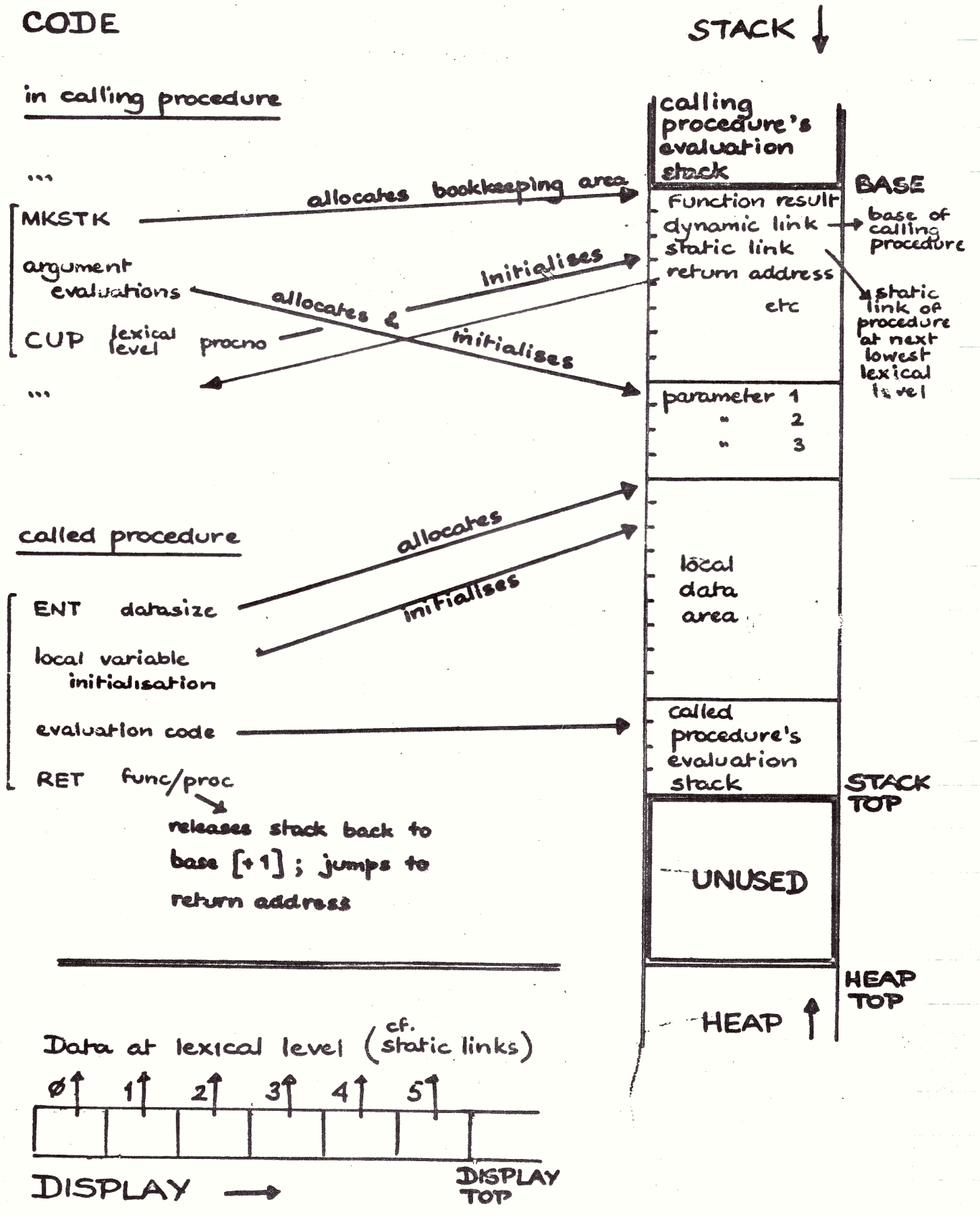
We may not remove the activation records for $S_1, S_2, \ldots, S_{i-1}$ since when P terminates and S_i resumes these records will once again be needed. We may simulate their removal by popping the top i-1 display pointers and storing them in the activation record for P. We then push a pointer to the activation record for P onto the display (cf Run-time Storage Management, Compiler Notes pp 30-3)

8.3 Stack Allocation with Dynamic Binding.

So far we have seen how a stack can be used to implement static binding under the "most closely nested" rule. A stack can also be used under the dynamic "most recently initiated" rule. Under the rule that a subprogram has its activation record pushed onto the stack when it begins and popped off when it terminates, we would at all times have a list of the active subprograms on the stack. The more recently the subprograms initiated, the higher on the stack it would be. The list on the stack is thus analagous to the environment of subprograms using static binding. To find the instance of X to which a use of identifier X refers, travel down the stack until the first activation record with a local

variable X is found. Alternatively we can eschew the stack of activation records, instead making a private stack for each identifier mentioned by the program. When a subprogram initiates, push down the stack for each local identifier belonging to the subprogram. When the program terminates, pop up the same stacks. While this method makes initiation and termination of subprograms relatively hard, it makes access of identifiers easy.

SNOBOL, which uses dynamic binding under the "most recently initiated" rule, permits dynamic creation of identifier names. However, the local variables of a subroutine (the only kind of subprogram in SNOBOL) must be declared in advance. Thus the second method described above can be used.


CODE
STACK
in calling procedure
...
MKSTK
allocates bookkeeping area
argument evaluations
Initialises
allocates &
CUP lexical level procno
initialises
...
calling procedure's evaluation stack
BASE
Function result
dynamic link
base of calling procedure
static link
return address
etc
static link of procedure at next lowest lexical level
parameter 1
" 2
" 3
called procedure
allocates
initialises
ENT datasize
local variable initialisation
evaluation code
RET func/proc
releases stack back to base [+1]; jumps to return address
local data area
called procedure's evaluation stack
STACK TOP
UNUSED
HEAP TOP
HEAP
Data at lexical level (cf. static links)
0 1 2 3 4 5
DISPLAY
DISPLAY TOP

9 Types & Data Abstraction

9.1 Type

The type of an identifier determines
- the set of values it may assume (domain)
- the set of operations to which it may be subjected.

Every programming language begins by supplying a set of data types. In LISP the major data type is the binary tree (called an S-expression) with the basic operations CAR, CDR and CONS. In modern imperative languages the usual built-in data types include integer, real, character and boolean. We will examine the typing system of languages, being the facility for defining new types and for declaring variables whose values are restricted to elements of a specific type with the expectation that type checking will be performed.

Advantages of typing systems include:
- common properties of variable are collected together in declarations, with the type name referring to these properties. To change the properties one need only change the type declaration, ie maintaining programs is made easier.
- specification is seperated from implementation. One can need not know how the data type is implemented, but only the properties, ie the declaration of the variable with its type gives an abstract description of its values, while the actual implementation details are reserved for the type definition.
- objects with distinct properties are clearly seperated and these restrictions can be enforced by the computer, improving both readability and reliability

Problems:

- should type information be conveyed at run-or compile-time?
  (strongly typed (ie compile time) languages have increased reliability, readability and maintainability, although they may be more cumbersome and less flexible)
- (how) should type checking be performed?
- should there be a way to parameterize types, and when should the evaluation of those parameters take place?

To understand a data type we must begin with its domain. When elts of a type are written in a program they are called Literals (constants whose value is given by their sequence of symbols). eg {true, false} are the literals of the type boolean.

A data type is scalar if its domain consists only of constant values (eg integer, real, boolean, char), and structured if its domain consists of members which are themselves composed of a set of types. Thus the elts of a structured type have fields, and those fields have types of their own. (eg array, record)

9.1.1 ENUMERATED DATA TYPES.

(An enumerated data type is a data type whose values (domain) are given in a list and whose only operations are equality & assignment (first cited PASCAL) The programming language must provide a mechanism for declaring and defining the new data type and for declaring variables whose values will come from the elts of the type. It is assumed that those literals are distinct (not necessarily true in ADA, where the name of the owner type may have to be specified if it includes a non-unique literal) and thus equality can be directly defined.

We can extend the usefulness of this type only permitting the domain to be ordered. The order in which the programmer lists the literals determines the order of the elts of the type. This allows the translator to define the relational operators such as >, <, >=, <= as well as ≠. We can also have (eg PASCAL) succ and pred operations.

A facility for defining enumerated types is a simple and efficient way of raising the expressive power of a language. It can be further extended by allowing subrange types (creating new types from a subset of an existing enumerated or integer type selected by a range constraint, giving upper and lower bounds) The new type ~~contains~~ includes all values of the original type which satisfy the range constraint, and all of the operations of the old type are inherited by the new type. Whether variables can be mixed with variables whose types are subranges of their types is language dependent. eg colour = (blue, red, green, white) How about circular ~~subranges~~ enumerated types circular; allowing succ(white) = blue, etc plus subset = green... blue (green, blue)?

9.1.2 ELEMENTARY DATA TYPES.

NUMBERS.

Dependent upon machine implementation of arithmetic.
Advantage: speed ; disadvantage: machine dependent, unportable.

ADA attempts to overcome this by allowing the user to specify the number of digits, resolution, etc.
A further problem is presented by polymorphic operators - those which have multiple meanings depending upon the types of its arguments (eg +). One method of dealing with this is

coerce the result to a type depending on the types of the operands; an alternative is to coerce the operands to the result type first

eg FORTRAN P = Q + I / J

Result coercion computes P = Q + REAL (I / J)

Operand coercion computes P = Q + REAL(I) / REAL(J)

In PL/1 this is complicated further by the fact that numbers can be BINARY or DECIMAL and can have attributes such as FIXED, FLOAT and various precisions. The introduction of precision forces the language to supply (usually complex) rules for the precision of operation results.

BOOLEANS.

Domain = {true, false}. Operations and or not xor, implies, equiv

Axiomatically:

x and y = if x then y else false

x or y = if x then true else y

x xor y = if x then not y else y

not x = if x then false else true

x imp y = if x then y else true

x equiv y = if x then y else not y = not (x xor y)

The constants true, false can be represented differently, eg

ALGOL : true, false, cannot be mixed with numeric types

PASCAL, Ada: pre-defined (redefinable) enumerated type (true, false).

PL/1 true = any bit-string with at least one non-zero bit.

CHARACTERS

FORTRAN - ~~originally~~ only allowed Hollerith strings and A-format.
ALGOL 60 - character strings enclosed in quotes, but no string variables or operations.

Mid 1960's character(string) data type introduced, and relational operators generalised to work on character strings (assume ordering of character set is well-defined). PL/1 was the first language to really provide string features, eg variable length character strings, concatenation operator |, LENGTH function, SUBSTR(ing) function (SUBSTR(string, start, length)), INDEX (BASIC 'INSTR' string search) (INDEX(string1, string2)), VERIFY (matching strings) and TRANSLATE (character replacement) However, this made implementation difficult and run-time operation inefficient. PASCAL moved away from providing such a general set of capabilities. SNOBOL, on the other hand, has very powerful features tailored to string processing, as well as an implementation which is largely machine independent.

POINTERS

A pointer data type is a type whose domain consists of pointers to other variables - introduced in PL/1, then in PASCAL and ADA. The pointer type is a very powerful feature, as it allows creation of complicated dynamic data structures - however it can make a program much more difficult to understand and can create subtle errors.

One aspect which is considered a difficulty of pointers in PL/1 is that a pointer can point to anything at all. In an attempt to control pointers more tightly, Pascal introduced

the idea of declaring pointer types so that they are restricted to point only to objects of a single type. In Pascal the only defined operations are creation, assignment, equality and dereferencing. There is also the special constant nil which belongs to every data type and points to no element at all. Garbage collection must be done by the programmer in Pascal, using dispose. An alternative is automatic garbage collection (eg LISP)

Variables in a language can be either static or dynamic. Static variables have a life which is tied to the program unit in which they are declared (so it is static program text)
Dynamic variables have a life which is independant of the program unit in which they are created. Their beginning is usually caused by the execution of a so-called allocator function (Run-time error from accessing uninitialised pointers)

Problems: • we don't know beforehand how many variables there will be
• complex structures ⇒ several pointers may point to the same variable
• it is possible for a dynamic object to be lost so that no pointer variable can ever reach it.

Pros: • pointers permit the life of storage to exceed block boundaries and instead to depend on the desires of the programmer
• pointers permit the sharing of storage amongst several variables
• they permit complex data structures which can be selectively updated.

One other problem is the dangling reference problem - an attempt to update a location which has already been disposed of can create havoc as the storage may now be used for other purposes.

9.1.3 # Structured Data Types.

In modern programming languages there are two ways of aggregating data so that it can be treated as a unit - the array and record (or structure)

An array is an aggregate of homogeneous data elts which are identified by their position within the aggregate.
A record is an aggregate of (possibly) heterogeneous data elts which are identified by name.

Arrays

An array is characterised by a name, a list of dimensions, a type for its elements and a type and range for its index set. Typically, but not necessarily, the index set is a consecutive set of integers.

One important design issue is whether the bounds of each dimension must be constants or can be expressions. In a static language such as FORTRAN constants must be used (In FORTRAN the lower bound is always 1, FORTRAN 77 excepted) In ALGOL60 the lower & upper bound can be any arithmetic expression which evaluates to an integer. The Pascal report only allows constants, as Wirth felt the loss of efficiency of dynamic array bounds was not worth the gain in flexibility. The real problem in Pascal though is that array size is considered to be part of the array's type definition, so arrays of different sizes are type incompatible. Some Pascal implementations have countered this serious restriction by allowing dynamic array bounds. Array sizes may also vary during execution in Ada, PL/1, ALGOL60 and SIMULA (Also - array descriptors row/col major order, mapping functions, etc of course!!)

In some languages an array need not be indexed only by integers. Eg Pascal we can use any (ordered) enumerated type.

In ALGOL68 nonrectangular arrays can be defined by using, for eg [1:3] ref [] real w which declare w as an array of three items, each of which references an array of reals. So for eg:

```
[1:a] real x
[1:b] real y
[1:c] real z
w[1] := x
w[2] := y
w[3] := z
```

gives a 2d array whose rows are of length a, b and c respectively.

Selecting a portion of an array is nicely done in PL/1 where an asterisk in any position implies all possible indices of that dimension are included. So, for eg: if we have W[1:3, 1:5] then:

PL/1:	W(3,*)	W(*,5)
ALGOL68	W[3,]	W[,5]
APL	: W[3;]	W[;5]

would be the third row and fifth column respectively.

In ALGOL68 a subarray can itself be indexed, so for eg W[4,5] = W[,5][4] = W[4,][5]
PL/1 does not allow this. Further in ALGOL68 we can use trimming, so for example W[3, 2:4] would represent the three elts W[3,2], W[3,3], W[3,4]. In ALGOL the index must be an integer.

Thus there are many ways in which arrays vary from language to language, including:

- syntax for naming arrays & array elts
- what types can be assumed by the value components
- what index types are allowed
- run-time or compile-time bounds determination
- complexity of array addressing
- array sharing, if any
- initialisation statements
- built-in operations

RECORDS.

A declaration of a record type describes the various fields and the types of their values. The issue of how to reference the components of a record has not been resolved in the same way by all languages. Knuth advocated functional notation. If ibm is a variable of type company, then he suggests we access the fields by writing director(ibm) department(ibm) [13], etc, using field names as function selectors. An alternative is the qualified name form, eg ibm.director, ibm.department[25] (used by Pascal & Ada) ALGOL is functional - we would write director of ibm, etc.

Records were introduced with COBOL structures. PL/1 followed in this tradition. More recent languages such as Pascal use the term record, and have a field list containing a fixed part followed by a variant part. We can avoid the use of qualified names (in Pascal) using the with ibm-do statement.

A record type with a variant specifies several alternative variants of the type. A variant part depends on a special ~~set of~~ component of the record, called its discriminant. Each variant defines the components that are legitimate for a specific value of the discriminant. The form for expressing the variant is often taken as the case statement of the language.

The variant record is essentially a mechanism for creating a union of types. In Pascal no record or array constants are allowed, so programmers must resort to element-by-element assignment, guaranteeing that the record passes through an inconsistent state. These inconsistencies can be problematic if the tag field is changed without changing the variant part to the appropriate new type. This problem was corrected in Euclid, which allows parameterised types so the tag field can be treated as a formal parameter to the declaration, with variant records being declared with a tag parameter any. In ALGOL68 the discriminated union is not part of the record mechanism but is treated as a seperate type, a union record which is a union of various other record types (also c)

ALGOL68 and Ada also provide record initialisation mechanisms

9.1.4 TYPE COERCION.

Can the type of all variables be determined at compile-time (static type checking) or will run-time routines be needed (dynamic type checking). Static type checking allows the compiler to do a great deal of consistency checking and in addition it can produce more efficient object code. But if the type is not known until run-time, all that the compiler can do is generate a call to a subroutine which will check the types and perform the correct operation. With dynamic type checking variables names may be declared, but their type not stated. The gain here is greater flexibility and simplicity in the writing of programs (eg LISP, APL). This flexibility seems especially desirable for interactive languages. The penalty is that each variable must carry with it a property list which is queried at run-time to determine its type.

Coercion - process whereby a value of one type is converted into a value of another type. This conversion process may be implicit (eg during evaluation of expressions, or assignments). Implicit conversions already occurred in early languages eg FORTRAN. PL/I took it to the extreme - type coercion between all built-in types.

Widening: coerce set to superset eg $\mathbb{Z} \rightarrow \mathbb{R}$ no loss of info

narrowing: coerce set to subset eg $\mathbb{R} \rightarrow \mathbb{Z}$ possible loss of info (PASCAL rounding, PL/I truncation, illegal?)

Coercion typically only amongst scalar, not structured types. PASCAL only (implicitly) provides $\mathbb{Z} \rightarrow \mathbb{R}$ and widening and narrowing of subranges.

A record type with a variant specifies several alternative variants of the type. A variant part depends on a special ~~set of~~ component of the record, called its discriminant. Each variant defines the components that are legitimate for a specific value of the discriminant. The form for expressing the variant is often taken as the case statement of the language.

The variant record is essentially a mechanism for creating a union of types. In Pascal no record or array constants are allowed, so programmers must resort to element-by-element assignment, guaranteeing that the record passes through an inconsistent state. These inconsistencies can be problematic if the tag field is changed without changing the variant part to the appropriate new type. This problem was corrected in Euclid, which allows parameterised types so the tag field can be treated as a formal parameter to the declaration, with variant records being declared with a tag parameter any. In ALGOL68 the discriminated union is not part of the record mechanism but is treated as a seperate type, a union record which is a union of various other record types (also C)

ALGOL68 and Ada also provide record initialisation mechanisms

9.1.4 TYPE COERCION.

Can the type of all variables be determined at compile-time (static type checking) or will run-time routines be needed (dynamic type checking). Static type checking allows the compiler to do a great deal of consistency checking and in addition it can produce more efficient object code. But if the type is not known until run-time, all that the compiler can do is generate a call to a subroutine which will check the types and perform the correct operation. With dynamic type checking variables names may be declared, but their type not stated. The gain here is greater flexibility and simplicity in the writing of programs (eg LISP, APL). This flexibility seems especially desirable for interactive languages. The penalty is that each variable must carry with it a property list which is queried at run-time to determine its type.

Coercion - process whereby a value of one type is converted into a value of another type. This conversion process may be implicit (eg during evaluation of expressions or assignments). Implicit conversions already occurred in early languages eg FORTRAN. PL/1 took it to the extreme - type coercion between all built-in types.

widening: coerce set to superset eg $\mathbb{Z} \rightarrow \mathbb{R}$ no loss of info

narrowing: coerce set to subset eg $\mathbb{R} \rightarrow \mathbb{Z}$ possible loss of info (PASCAL rounding, PL/1 truncation, illegal?)

Coercion typically only amongst scalar, not structured types. PASCAL only (implicitly) provides $\mathbb{Z} \rightarrow \mathbb{R}$ and widening and narrowing of subranges.

ALGOL68 provides 6 types of conversions
(no narrowing)

- widening
- dereferencing
- deproceduring
- rowing (extend single length to a row of that argument)
- uniting (metaconversion of a type into a union of which that type is a member)
- voiding

9.5 TYPE EQUIVALENCE.

The problem of type compatibility can be solved in at least 2 ways

Rule 1: NAME EQUIVALENCE
Two variables are of the same type iff (they are declared (together or declared) using the same type identifier name

Rule 2: STRUCTURAL EQUIVALENCE
t v. a. of. s. t. iff the components of their type are the same in all respects.

Eg suppose in PASCAL we have

```
type  A = 1..10;
      B = A;
```

then by rule 1, A and B are different, by rule 2 they are the same.

Name equivalence is simple to determine, but forces the programmer to define extra type names. Structural equivalence also has problems. eg:

```
var  A: record x, y: real end;
     B: record u, v: real end;
```

Here the structures are the same, but the fieldnames are different.
Once pointers are introduced determining structural equivalence becomes very complex.

ISO Pascal uses a third method: declaration equivalence. Two types are equivalent in this sense if their structures were described by the same actual type specification, i.e., either the type names are equal or they lead back to the same array, record or file spec. EUCLID uses a more sophisticated method: two type names are the same, if, after all the synonyms have been removed, the resulting definitions are equivalent. Algorithm (to check =):

- start with each type
- replace each type name by its definition, unless it is a module or exported from a module. Substitute any actual parameters.
- replace each variable by its type name
- repeat until there's nothing to do.

Ada uses rule 1 (NAME)

9.2 DATA ABSTRACTION:

An abstraction is a way of representing a group of related things by a single thing which expresses their similarities and suppresses their differences. A data abstraction in a programming language is a mechanism which encapsulates (shields) the representation of a datatype and the implementation of the operations with which the datatype is equipped. The basic types are usually abstract in this sense (eg integer in Pascal, string in SNOBOL) but not always (eg pointer in Pascal). The structured types are often not abstract (eg record) especially if user-defined.

Some languages offer the user the data abstraction facility by introducing the idea of a closed scope: identifiers must be explicitly declared within the scope or explicitly mentioned in a import list to be 'visible' within the scope. To be visible outside their declared scope, identifiers must be explicitly mentioned in an export list.

Such a closed scope has the following components:

- import and export lists
- a representation part
- an implementation part
- initialisation and finalisation code
- a name

Abstract specification of data types

á la GUTTAG

LIFO

structure STACK ;

newstack ( ) → stack
push (stack , item) → stack
pop (stack) → stack
top (stack) → item
isnew (stack) → boolean

LIFO

declare stk : stack ; i : item ;

pop (push (stk, i)) = stk
top (push (stk, i)) = i
isnew (newstack) = true
isnew (push (stk, i)) = false

restrictions pop (newstack) = error
top (newstack) = error

---

structure QUEUE ;

FIFO

newq ( ) → queue
add (queue, item) → queue
delete (queue) → queue
front (queue) → item
isnew (queue) → boolean

formally identical with above

FIFO

declare q : queue ; i : item ;

isnew (newq) = true
isnew (add (q, i)) = false

recursive definitions

new

delete (add (q, i)) = if isnew (q) then newq else add (delete (q), i)
front (add (q, i)) = if isnew (q) then i else front (q)

restrictions front (newq) = error
delete (newq) = error

Examples of DATA ABSTRACTION:

modules	MODULA, Euclid	structure
packages	Ada	Horowitz pp 241ff
classes	SIMULA	newtype, (definition procedure)
clusters	CLU	
forms	ALPHARD	Tennent pp 196ff

MODULA

```
module SMALLSETMODULE;                          <- name
    use ...                                     <- import list
    define SMALLSET, INSERT ...                 <- export list
    const LIMIT = 100;
    type SMALLSET = record INDEX : Integer;     <- representation
                           ITEMS : array 1: LIMIT of integer;
                    end; ...
    procedure INSERT (const I: integer; var S: SMALLSET);   <- implementation
    ... end INSERT; ...
    procedure INIT (var S:SMALLSET); begin S.INDEX := 0 end INIT;
end SMALLSETMODULE;                             <- initialization
...
var P : SMALLSET; ...                           <- declaration (with representation hidden)
...                                             (calls initialization)
INSERT (K, P); ...
```

operation (with implementation hidden)

NB Modules (& their local objects) are local to their containing procedure
Modules may be nested
Exported variables are read-only
Export is OBSCURE : i.e. fieldnames of exported records invisible.
(contrast MODULA-2)

Ada

parameterization of package

```
generic   SIZE : integer; type ELEM is private;
package STACKS is
        type STACK is limited private;
        procedure PUSH (S : in out STACK; E : in ELEM);
        procedure POP  (S : in out STACK; E : out ELEM);
        OVERFLOW, UNDERFLOW : exception;
    private
        type STACK is record
                SPACE : array (1 .. SIZE ) of ELEM;
                INDEX : integer range 0 .. SIZE := 0;
        end record;
end;

package body STACKS is
        procedure PUSH (S : in out STACK; E : in ELEM) is
        begin  if S.INDEX = S.SIZE then raise OVERFLOW; endif;
           S.INDEX := S.INDEX + 1;  S.SPACE (S.INDEX) := E;
        end PUSH;
        procedure POP (...
        ...
end STACKS;
...

package INTEGERSTACK is new STACKS (SIZE => 100, ELEM => integer);
...
declare    use INTEGERSTACK;      X, Y : STACK;    ...
begin
   ... PUSH (X, 175); ...
end.
```

assignment, equality not automatically available

Specification part

visible (export list)

invisible representation

initialization

invisible implementation

import list

10 LANGUAGES.

10.1 ADA

Ada is intended to support the construction of large programs by teams of programmers. An Ada program is usually designed as a collection of packages. A package may represent an abstract data type or a set of data objects shared between subprograms, but more generally a package contains an integrated set of type definitions, data objects, and subprograms for manipulating these data objects. Such packages can be made available through a program library.

STORAGE MANAGEMENT

Heap for pointer types and dynamic arrays
Stack for statically bound ids (block structure)

TYPES

Strong, diverse, powerful typing. Built in string type.
Type equivalence: name equivalence only.

- Enumerated data types:
  - if 2 different enumerations have a common elt, interpretation is determined by context. Programmer can qualify which.
  - range constraints eg SUMMER: MONTHS range DEC.. FEB
  - subtyping eg SUBTYPE winter IS MONTHS range JAN.. AUG
  - booleans and characters.

- Elementary data types
  - numbers - some feature explicit
    eg DIGITS 10 RANGE -1.0 TO 1.0
  - characters, booleans
- Pointer types
- Structured data types
  - arrays : variable dimensions
    eg operations =, <>, assignment, indexing, slicing, initialisation
  - records : similar to Pascal, has qualified name form
    eg operations = <> component selection, initialisation
    variant records - discriminant change entire record,
    not selective updating as in Pascal.
    subtype records are allowed.
- Derived types
  - types defined on existing types
  - eg TYPE newcol IS NEW colour
    newcol & colour are now regarded as different types.
- Private types
  - enclosed within a package
  - implementation hidden within the package.

DATA ABSTRACTION

The abstract data typing facility in Ada consists of syntactic (functional), semantic (possible results) and restriction specifications. This gives an implementation (representation) independent definition of the data type - it is up to the implementor to supply a sufficiently complete set of rules.

- Representation may be termed:
  - 'limited' : :=, =, <> are not automatically available
  - 'limited private' : programmers can provide procedures to implement the above 3 operations.

10.2 C

A high level assembler with some block structured features.

STORAGE MANAGEMENT.

Stack-based (run-time) due to block structure.

SCOPE RULES

Block structure allows static scope rules - efficient, but bad for portability
- local variables
- static (permanent) variables (eg STATIC CHAR c)
- register variables for heavy use (eg REGISTER INT I).

SUBROUTINES

- all procedures are functions
- all parameter calls are by value - to call by reference, the address of the variable must be passed.
- multiple return points are permitted
- recursion

DATA TYPES

- Not strongly typed, much coercion and type mixing allowed. No user-defined typing (ie not TYPE declaration)
- Elementary types (char, int, float, double, constants (#define))
- Only structured type is array - dimensions always start at 0.
- pointers

10.3 SNOBOL 4

String / language manipulating language.

STORAGE MANAGEMENT.

Most operations on variable-length strings ⇒ heap-based storage with variable size elements.

Each element has associated with it:
- a garbage collection bit
- a block length indicator
- a compaction pointer field (2 pointers are used for full compaction)

So that garbage collection with full compaction can be done.

DATA CONTROL FEATURES.

GENERAL

- OPSYN feature allows the programmer redefinition of primitive operation symbols
- distinct naming and creating features ie ARRAY(10) creates an array and returns a pointer to it without naming it.
- variables can be used multiply, eg

```
x = x(2) : x
↑   ↑        ↑
variable  subroutine call  label
```

the unique variable meaning is determined through its context.

LOCAL ENVIRONMENTS

- destroy on return and recreate on entry
- a central stack of local environment tables is used to facilitate recursion (can still use 'base addr + offset'; just change base addr.)

NON-LOCAL ENVIRONMENTS

- all variables are global, unless specified as local
- this facilitates 'most recent association' rule.
- no extra burden, as SNOBOL performs run-time type checking anyway.

SUBROUTINES

- no distinction between subroutines & functions; subroutine calls can be used as both.
- recursion is allowed
- parameters passed by reference
- defining function DEFINE ('V(A,B),F,X,Y')
  - V: name
  - A,B: formal params
  - F,X,Y: local vars

TYPES

- elementary types are strings & integers - otherwise there is no typing.
- indirection allowed: eg $A is regarded as the variable whose name is stored in A (can have $$A, etc)
  ⇒ all variable names must be held in memory at run-time
- arrays: not really necessary. Write Y=X(I) as Y=$('X'∇I)
- pointers: the DATA statement can be used to create pointer data structures
- because of structure of SNOBOL4, there is a large amount of coercion but lots of flexibility. The language is thus poorly protected against semantic errors.

FORTRAN

Scientifically orientated with poor string manipulation
Card / column orientated format. No declarations,
statement seperators or reserved words. Poor looping
structure ⇒ lots of GOTO's.

STORAGE MANAGEMENT.

- Static - no run-time storage management.
- allocation during translation remains fixed
- each subprogram compiled seperately
- ⇒ no recursion or dynamic data structures, but
  efficient code.
- late version provide a run-time stack for variable arrays.

DATA CONTROL

- Scope rules & referencing environments are determined statically
- local environments - each reference replaced by BA+offset
  computation, eliminating need to hold identifier
  data at run-time.
- non-local environments explicitly specified by common - only
  a single level of non-local referencing. Provides good
  error-detection etc but not very flexible.
- subroutine parameters passed by reference only.
- functions are allowed - many supplied maths functions.

TYPING

- no abstract data typing, records, pointers, sets or enumerations
- elementary data types - integer, real, char, boolean, double precisio
- arrays - integer subscripts from 1... Late implementations
  allow variable arrays through subroutine calls.

10.5 LISP.

A list-processing language based solely on dispa.

STORAGE MANAGEMENT

Uses stack, but also requires a heap for dynamic addition of elements to a list.

STACK-BASED

- each activation record contains a return pointer and temporaries for expression evaluation and parameter transmission
- local referencing environments (A-list entries) are normally stored in a seperate stack for efficiency (stack is called the A-list). The stack containing return pointers & temporaries may then be hidden from the user & allocated sequentially.

HEAP-BASED

- garbage collection is done, ⇒:
  - any active elt must be reachable from outside the heap
  - every pointer outside the heap, which points inside the heap, must be identifiable
  - pointers within active elts pointing to other heap elts must be identifiable
- these conditions are satisfied by:
  - each elt is formatted identically with 2 pointer fields
  - there are a small set of pointers which may contain pointers from outside the heap (A-list, OB-list, pushdown-list, etc).

Scope Rules

These are determined dynamically

Local Environments

- are not retained

Non-Local Environments

- "most-recent association in the calling chain" is used
- a run-time stack of local environments is used for this purpose (A-list)
- above technique ⇒ run-time type checking necessary.

THE C PROGRAMMING LANGUAGE.

<program> → { <include> #include <filename> }
{ <define> #define <identifier> <expression> }
{ extern <type> <name_list> }
[<type>] main ([<name-list>]) — argc, argv optional
[<declarations>]
<function_body>
{ [<type>] <name> ([<name-list>])
[<declarations>]
<function_body> }

<filename> → <device> : <name>
<device> → mdv1 | mdv2 | etc
<identifier> → uppercase name conforming to C naming conventions.
<type> → int | double | float | char | long | unsigned | short
<name_list> → <name> { , <name> }
<name> → <letter> { <letter> | <digit> | _ }
<letter> → 'A' | ... | 'Z' | 'a' | ... | 'z'.
<digit> → '0' | .. | '9'
<arguments>
<declarations> → { <type> <name> ; }
<function_body> → { <declarations> <statements> return [expression] ; }
<statements> → <statement> ; { <statement> ; }
<statement> → { <statements> } | <directive> | break |
<switch case> | continue | default: <statement> | <do> | <for> |
goto <name> | <if> | <while> | <call> | <expr> | ; ?
<case> | default: <statement>

{ } 0 or more

[ ] 0 or 1

<directive> → <asm> | <define> | <iff> | <include> | <undef>

<asm> → #asm <assembly source code> #endasm

<define> → #define <identifier> <expression>

<iff> → #if <constant_expr> <statements> [#else <statements>]
#endif

<include> → #include <filename>

<undef> → #undef <identifier>.

<switch> → switch(<expression>) statement ;

<do> → do <statement> while (<expression>) ;

<for> → for (<init> ; <test> ; <incr>) <statement> ;

<if> → if (<expression>) <statement> [; else <statement>] ;

<while> → while (<expression>) <statement> ;

<call> → <name> (<expression_list>)

<expression_list> → <expression> [, <expression_list>] | ε

<case> → case <constant_expr> : <statement> ;

C COMPILER

1 REGISTER USAGE

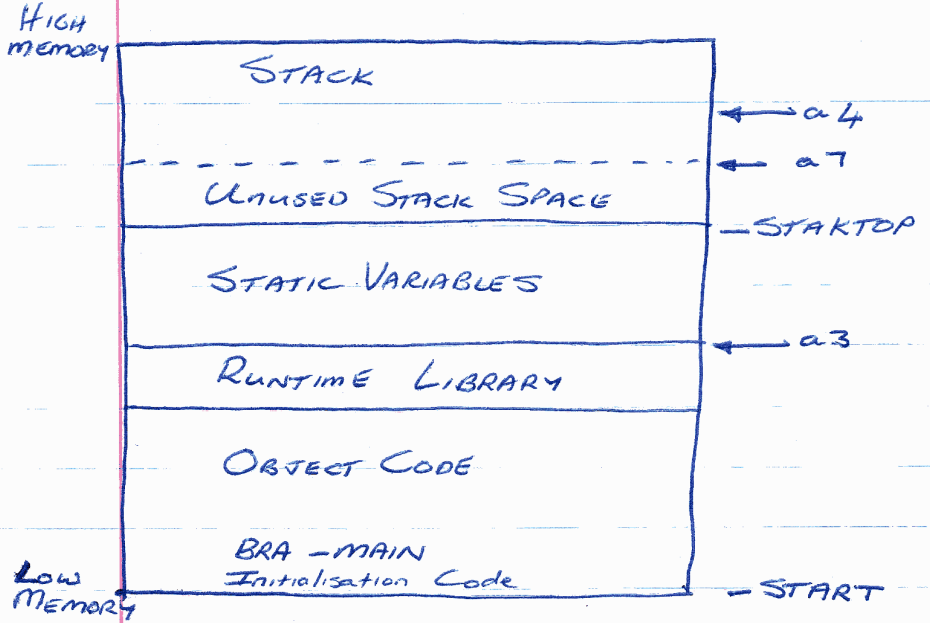
ADDRESS REGISTERS

a0 - a2 - used for QDOS
a3 - static (global variable) base pointer
a4 - local variable frame pointer
a5 - primary register (also function value returns)
a6 - secondary register
a7 - stack pointer SP

DATA REGISTERS

d0 - d1 - reserved for QDOS
d2 - # arguments passed to last function
d3 - } expression temporaries
d4 -
d5 - d7 - not used.

2. MEMORY MAP



3. Code Overlays

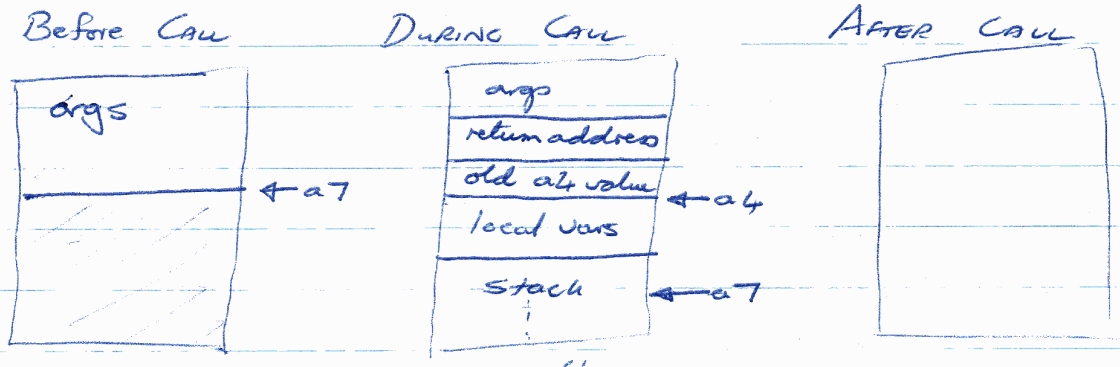
3.1 Function Calls

Loop: { GetArg (value or address – 32 bits each)
PUSH ARG ONTO STACK }

SAVE #ARGS IN d2.

CALL FUNCTION

ADD.L #ARGS*4, SP



3.2 Function Entry (Actually done in function)

Loop: { Get local decl
Calculate & Save offset
Increment space-reqd counter }

LINK a4, # –SPACE

3.3 During Function

Declarations
Loop as 3.2. SUB.L #SPACE, SP (or equivalent)

Variable Accesses use –ve offset from a4
Parameter Accesses use +ve offset from a4

3.4 End of Function

UNLK a4
RTS.

CONTROL CONSTRUCTS.

While (cond) statement ;

-STARTLABEL :

Generate & print startlabel, generate endlabel, save on while stack.

Generate code for cond

beq -endwhile

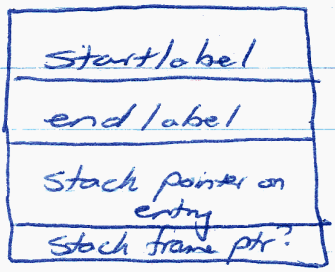
Generate code for statement

bra -STARTLABEL

-ENDWHILE :

For break, generate bra -endwhile

WHILE STACK RECORD :



The ponters are saved so that they can be restored on exit in case they are not the same as on entry.

do {statement} while (expr);

Generate, stack & print -dostart:, generate & save -doend
Generate code for statement
Generate code for expr (result in a5)
tst a5 (may be unnecessary)
bne -dostart
Unstack do (labels, sp, frame↑)

If break, generate bra -doend

for (expr1 ; expr2; expr3) statement ;

Each expr is an optional expression list

* Check semantics of this for expr 2 as a list. See below
* Isn't expr 1 actually a statement? and expr 3? No.
* Also, which code comes first, expr 2 or expr 3? expr 2.
~~Generate code for expr1~~ Missing expr 2 defaults to 1.

SEMANTICS OF COMMA OPERATOR
Expressions are evaluated left to right, with results being discarded except for last expr.

SEMANTICS OF FOR:

```
expr1 ;
  while (expr2)
        { statement};
          expr3;
        }.
```

Generate code for expr 1
Generate & stack loop record

_startloop:

Generate code for expr 2
tst a5
beq _endloop

Generate code for statement
Generate code for expr 3

bra _startloop

_endloop:

Unstack loop record.

SUMMARY OF LOOP OVERLAYS.

If continue / break at any point, generate bra _startloop / bra _endloop

1 • Generate & stack loop record

2 FOR LOOP: Generate code for expr 1

3 _startloop:

4 FOR LOOP: Generate code for expr 2.
DO LOOP
WHILE LOOP: Generate code for expr

5 FOR, WHILE LOOPS: tst a5 / beq _endloop

6 Generate code for statement

7 FOR LOOP: Generate code for expr 3
DO LOOP: Generate code for expr

8. FOR, WHILE LOOPS: bra _startloop
DO LOOP: tst a5 / bne _startloop

9 _endloop:

10 Unstack loop record

Loop Record (16 bytes)

start loop
end loop
SP on entry
Frame ptr a4 on entry

IF (expr) statement1; ⟨ else statement2; ⟩

Generate & save labels
Generate code for expr
tst a5
beq _flabel
Generate code for statement 1
[If else, bra _elabel]
_flabel:
[If else, generate code for statement 2
_elabel:]

Untack record. (could go on loop stack, but maybe dangerous if there are dangling else's although I don't think this should be a problem.)

SWITCH expr { case list default }

case_list → case-stmt *
case-stmt → CASE expr : stmt-list
opt default → { default : stmt-list

Generate & stack labels
~~_switchlabel:~~
Generate code for expr
~~tst a5~~ movl a5, -(SP)
~~beq~~
Case-stmt: Generate code for expr
movl ~~d6~~, (SP), a6
subl a5, a6
bne _next label

Generate code for stmt-list

bra _outlabel

_nextlabel:

Create a new nextlabel

Default: Generate code for stmt-list

_outlabel:

addq #4, SP | pop off expr value

Pop off labels record.

OPERATOR PRECEDENCE PARSING

Precedence relns:

$a <\cdot b$	a yields precedence to b
$a \doteq b$	a, b have same precedence
$a \cdot> b$	a takes precedence over b.

Operator grammar: No production rhs is ε or has two adjacent non-terminals

Eg: $E \to E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid id$

With usual precedences:

	+	−	*	/	↑	id	(	)	$
+	>		<				<	>	>
−	>		<				<	>	>
*	>		>				<	>	>
/	>		>				<	>	>
↑	>		>				<	>	>
id	>		>					>	>
(	<		<				<	=	
)	>		>					>	>
$	<		<				<		

	+ −	/ *	↑	id (	)	$	(Right)
+ −	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	
/ *	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	
↑	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	
id	$\cdot>$	$\cdot>$	$\cdot>$		$\cdot>$	$\cdot>$	
)	$\cdot>$	$\cdot>$	$\cdot>$		$=$	$\cdot>$	
$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$			
(	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$		
(left)							

Defn: An operator precedence grammar is an ε-free operator grammar in which the precedence relns are disjoint ie any a,b never more than one of $a \lessdot b$, $a \doteq b$ and $a \gtrdot b$ is true.

Algorithm

```
while (1)
    if (stack = $ and input = $ ) {accept; break;}
    else if (stack <· input or stack ≐ input)
            push (input)
         else if (stack ·> input)
                 repeat pop a
                 until stack <· a
              else error
```

Try an example

3 ↑ 4 + 5 * 2 / 3 − 6 * (2 − 1) $

Stack	Input	Action
$	3	push 3
$ 3	↑	pop 3
$	↑	push ↑
$ ↑	4	push 4
$ ↑ 4	+	pop 4
$ ↑	+	pop ↑
$	+	push +
$ +	5	push 5
$ + 5	*	pop 5
$ +	*	push *
$ + *	2	push 2
$ + * 2	/	pop 2

Stack	Input	Action
$ + *	/	pop *
$ +	/	push /
$ + /	3	push 3
$ + / 3	-	pop 3
$ + /	-	pop /
$ +	-	pop +
$	-	push -
$ -	6	push 6
$ - 6	*	pop 6
$ -	*	push *
$ - *	(	push (
$ - * (	2	push 2
$ - * (2	-	pop 2
$ - * (	-	push -
$ - * (-	1	push 1
$ - * (- 1	)	pop 1
$ - * (-	)	pop -
$ - * (	)	push)
$ - * ( )	$	pop)
$ - * (	$	? pop (
$ - *	$	pop *
$ -	$	pop -
$	$	

POP ORDER FOR 3 ↑ 4 + 5 * 2 / 3 - 6 * (2 - 1)

3 4 ↑ 5 2 * 3 / + 6 2 1 - * - ie RPN.

Precedence Functions

For symbols a, b

① $f(a) < g(b)$ if $a \lessdot b$
② $f(a) = g(b)$ if $a \doteq b$
③ $f(a) > g(b)$ if $a \gtrdot b$

Method

① Create symbols f_a and g_a for each a that is a terminal or $
② Partition the symbols into as many groups as possible, in such a way that if $a \doteq b$ then f_a and g_b are in the same group (use transitivity to help difficult cases)
③ Create a directed graph whose nodes are the groups of ②
For any a, b, if $a \lessdot b$ place an edge from g_b group to f_a group
and vice - versa
④ If the graph has a cycle, no precedence functions exist
If there are no cycles
$f(a) = \max \{$ length $\{$ paths from $f_a \}\}$
$g(a) = \max \{$ length $\{$ paths from $g_a \}\}$.

OPERATORS. (From highest precedence to lowest)

PRIMARY EXPRESSION OPERATORS (all l to r associative)

() .
[] →

-1 UNARY OPERATORS. (all r to l associative)

++	--	
*	&	[contents of, addr of
-	~	! [unary minus, 1's comp, log. NOT
(type)	sizeof	[cast, size

-2 ARITHMETIC (l to r) * / % [Mult, div, mod

-3 ARITHMETIC (l to r) + - [Add Subtract

-4 SHIFTS (l to r) >> << [Shift right | left

-5 RELATIONS (l to r) > >= < <= == !=

-6 BITWISE AND (l to r) &

-7 BITWISE XOR (l to r) ^

-8 BITWISE OR (l to r) |

-9 LOGICAL AND (l to r) &&

-10 LOGICAL OR (l to r) ||

-11 CONDITIONAL (r to l) ? :

-12 ASSIGNMENT OPS (r to l)

= += -= *= /= %=

>>= <<= &= ^= |=

-13 SEQUENCE (l to r) ,

GENERAL PURPOSE MACROGENERATOR (GPM) : Strachey,Computer Journal 8 3, October 19

Wegner, Programming Languages, Information Structures and Machine Organisation,
McGraw Hill Computer Science Series, paragraph 3.2, (pp 151ff)
Higman, A Comparative Study of Programming Languages, McDonald/Elsevier
Computer Monographs, chapter 8, (pp 55ff)

SYNTAX: ( ) signifies repetition any number of times, zero included.

```
  textstream          ::=   { break(<[) | quote | macrocall}
   quote              ::=   < { break(<>) | quote} >
   macrocall          ::=   [ parameter {, parameter} ]
    parameter         ::=   { break([,]<) | quote | macrocall}
   formalparameter    ::=   # digit {digit}

   e.g.  [DEF, _X, _<[>#1<]>] =>
         [DEF, _X, _<[> #digit <]>] => [DEF, _X, _<[> formalparameter <]>] =>
         [DEF, _X, _ quote #digit quote] =>
         [parameter, parameter, parameter] => macrocall => textstream
```

SEMANTICS:

Treat each syntactic unit of a textstream as an expression to be evaluated.
1. A textstring is evaluated by simply copying.
2. A quote is evaluated by omitting the opening <, copying the enclosed text without further evaluating it, and omitting the closing >.
3. A macrocall is evaluated by
   a. evaluating each parameter in turn to a temporary stack of parametertexts numbered from 0 upwards;
   b. applying an environment or definition-table to parameter text 0 (the macroname), and deriving a replacement textstream (the macrobody);
   c. evaluating the macrobody, adding this rule: replace every formalparameter #n with a copy of parametertext n from the temporary stack.

The output stream is the evaluation of the input stream.

NB. Macrocalls can occur textually within parameters, and thus nested within other macrocalls. Entry into any level of static textual nesting of this sort is deemed to cause, at evaluation time, entry into a further level of dynamic evaluation nesting. In other words, there must be provision for a stack of temporary stacks of parametertexts. Exit from the textual nesting causes exit from the evaluation level, passing of the value obtained (i.e. the textstream produced) to the enclosing level, and loss of all sub-evaluations and sub-values derived in the course of the evaluation.

The GPM environment or definition-table contains initially 6 system definitions:

DEF & UPDATE yield as macrobody the nullstring, but have a side-effect on the definition-table for the current and all nested levels of evaluation. DEF extends, and UPDATE overwrites, the environment or definition-table so that when it is applied to parametertext 1 of the call of DEF (as macroname) it yields parametertext 2 of the call of DEF (as macrobody).
e.g. [DEF,N,<A#1#2B>][DEF,P,<A[N,C,D]#1>][P,F] => AACDBF.
How does [DEF,X,<A[DEF,Y,#1]>] differ from [DEF,X,A[DEF,Y,#1]]?
What does [A,X,U,[DEF,A,<#1#2#1>]] yield?

VAL yields as replacement-string (as macrobody) the unevaluated macrobody currently corresponding to the macroname given by VAL's parametertext1.
e.g. [DEF,X,<aaa>][VAL,X] = [DEF,X,<<aaa>>][X]

BIN expects its parametertext1 to be a (possibly signed) decimal numeral, and yields an internal (externally inaccessible and unprintable) representation of the corresponding integer.
DEC expects its parametertext1 to be such an internal representation of an integer, and yields the corresponding signed decimal numeral.
BAR expects 3 parameters, the last two of which must evaluate to internal representations of integers, while the first must evaluate to +,-,*,/ or some other character. The call of BAR will then yield respectively the internal representation of the addition, subtraction, multiplication division or remainder-on-division of the two integers.

An implementation of GPM is available on the UNIVAC.

GPM "One man went to mow"

lines 1 - 9 irrelevant

10 - 22 FORTRAN-strategy definitions stacked for duration of program execution. NO OUTPUT

23 - 28 TITLE + VERSE 1

29 - 30 VERSES 2 - 9 by call [REST, 2]

- Why include definition of REST in call of REST ? PASCAL-strategy - definition stacked for duration of call only.

  IN THIS CASE, makes no effective difference. Merely more elegant.

- [REST, n] is defined as [VERSE, n] [n] , $2 \leq n \leq 9$

  Why is the definition of n in call of n ? Elegance again

  Why is the definition of n in call of REST ? To allow parameterizing of the definition of n by the immediately containing call of REST e.g. [REST, 5] → [5] → [REST, 6]

  But why not [DEF, REST, < [VERSE, #1] [REST, [5, #1]] >] without intermediate definition of n ?
  Intermediate definition of n allows of alternative definitions being offered every time n is called

---

N.B. Quotation as a device used in a definition to delay evaluation of the quoted expression till calltime

Functional composition achieved by quoting inner calls in the definition of outer macros.

Recursion if the inner calls are (directly or indirectly) calls of the outer macros, and an exit is provided by conditional calls

Conditional calls by including alternative definitions of inner macros in the definition of an outer macro + a quoted call of a macro whose name is a parameter of the outer call and can be made to vary over the set of alternatives provided.

GPM — ONE MAN WENT TO MOW
(SEE ANALYSIS IN PRINTED NOTES)

```
[def,?,LEFT][def,ALT,<[[?]]>,
[def,LEFT,<LEFT[update,?,RIGHT]>],
[def,RIGHT,<RIGHT[update,?,LEFT]>]]
[ALT] [ALT] [ALT] [ALT]

[AN UNDEFINED MACRO]

316-253=[dec,[bar,-,[bin,316],[bin,253]]]

[def,S,<[1,2,3,4,5,6,7,8,9,[def,1,<#>#1]]>]S defined
[def,P,<[?,0,1,2,3,4,5,6,7,8,[def,?,<#>#1]]>]P defined
[def,N,<[ ,ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,[def, ,<#>#1]]>]N defined
[def,HEAD,<
[N,#1] MEN WENT TO MOW,
       WENT TO MOW A MEADOW,
>]HEAD defined
[def,END,<
ONE MAN AND HIS DOG WENT TO MOW A MEADOW>]END defined
[def,TAIL,<[N,#1] MEN, [#1,[def,#1,<[TAIL,[P,>#1<]]>][def,6,<
[TAIL,5]>][def,2,<[END]>]]>]TAIL defined
[def,VERSE,<
[HEAD,#1][TAIL,#1]>]VERSE defined
```

succ
pred
letter number string

S - succ
P - pred
N - returns number string

[TAIL,[P,2]]

Typed in:

```
        ONE MAN WENT TO MOW
        ===================

ONE MAN WENT TO MOW,
        WENT TO MOW A MEADOW,[END]
[REST,2,[def,REST,<[VERSE,#1][#1],
[def,#1,<.[REST,[S,>#1<]]>][def,9,<...>]]>]]
 2                  2
```

Generates verses 1 - 9, then terminates by substituting ... for 9 (no succ).