1 INTRODUCTION: Types Basic, Structured, Dynamic
Uses Of Typing: Layout and Allocation, Error Checking

2 Layout and Allocation Of Basic and Simple Structured Types Integers, Records

3 Layout and Allocation Of Stuctured Types- Arrays

4 Layout and Allocation Of Static Structures, Basic Pointers and Array

5 Layout and Allocation Of Dynamic Types Pointers and The Heap

6 Dynamic Types-Possibility Of Higher Level Data Structure Specification

7 Defining Composite Types Grammar For Specification-Modula Example

8 Parsing and Layout Of Composite Data Structures

9 Typing, Error Checking, Type

10 Type Equivalence Structural and Name

**CONTENTS**

© M C Paull 2008

## Data, Data Structures, And Typing Overview

**Basic Types**: There are a number of reasons for a programming language to incorporate data-structures and for assigning types to such structures. There are a large number of issues associated with type assignment. These issues are introduced by a quick, rough **history** of their occurence in the development of programming languages. Initially **data was a binary number in a location, and the locations were arranged in a linear array.** The **operations were arithmetic on a single argument type, a binary number, so there was no need to consider a variety of types.** But **numbers are of different types**, ex., integer, float with mantissa and exponent. These each necessitate different interpretation of the contents and perhaps the size of a location. Thus the memory **layout** of a location now depends on the type of that number. Variables receive the type of the number they represent. Also an **operation may make sense for a number of types. For integers and reals the functions** addition, multipllication, division,all apply There are difference in what must be done to implement these operations. for different number types.(ex. multiplication applied to floats requiring the addition of exponents, integer division requiring the rounding of remainders). An operation which applies to a number of types is **polymorphic.**

Furthermore **operation might have different types as arguments**, say an integer and float. The question then arrises as to the **type of the result of the operation.** which will typically be assigned to a variable of a given type.

Initially these operations were largely built directly into the hardware. Nevertheless in order to know which of the operations (which machine language instruction) would be applied for these instructions the **type** (nature) of the number is needed Once assemblers and compilers were available these types were necessarily made explicit in the program specification. That is both **variables and constants had to be typed**. This was done mostly by **representation for the constants and by simple declaration for variables.**

Programs **originally dealt with numbers or more generally mathematics exclusively.** To make input and output legible one had to deal with characters. Someone recognized that those binary numbers could be interpreted as **characters**. and a sequence of such characters (strings) could be seen as a a word and a sequence of words as etc So another **type , with its applicable operations,** was introduced.

The variable and constants considered thus far are basic types. Many builtin operations apply to them. Also one can define procedures which take such types as arguments.

**Structured Types:** Still associated with mathematics a **more structured data type,** the array, was introduced. **Arrays** are ordered collections of quantities of a more basic type. Declaring an array, A, in most imperative languages gives **rise to a number of variables A[1], A[2],** one for each element of the array. These are each of one of basic types. The built in operations allowed on them (**A[1], A[2],..)** are those allowed on the basic type. However many **allow the passing of arrays** (the value of all their elements) as arguments of programmer defined procedures. (This will usually involve the **use of a pointe**r (considered later), since copying entire arrays into procedures-which will later leave the Runtime stack is very inefficient). **Records** are another popular data structure. In these again what becomes available to operations is access to the elements of the record. These elements are each named and unlike the array may be of different type. **Array elements** are **homogeneous** in type while a **record elements** are **heterogeneous.**

**Dynamic Types**: So far all the data types and structures we have considered can be **layed out and assigned memory (allocated) at compile time.** The introduction of the **pointer type** a allows indirect access to complex types. The pointer can be passed giving indirect access to an arrays elements so copies of the array arenot necessary. The pointer also allows data structures which can grow and shrink during run time. It can string records together by making one or more elements of the record pointer types. With the pointer we get **dynamic data structures-lists, trees, t**hat are created and destroyed at runtime. This iimportant extension of data types introduces some problems. Keeping track of data still in use and no longer needed as well as well as acquiring new memory for a program requires additional runtime involvement of the system.

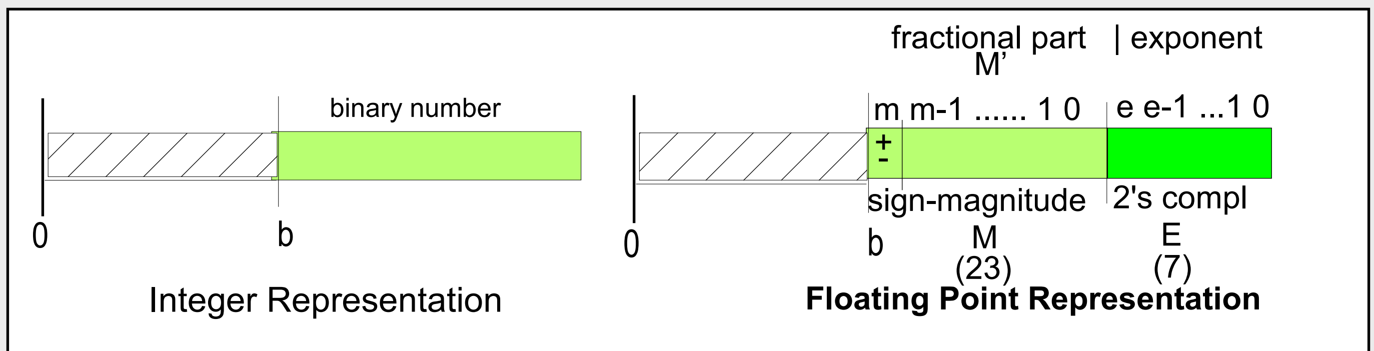
© M C Paull 2008

Data structures such as stacks, lists, and trees can be constructed, grown and shrunk at runtime by explicit use of pointers and appropriately designed records with one or more of its elements of pointer type. All these structures can be constructed in similar ways. In fact we know that a language like Scheme **Lisp builds lists more or less automatically** without detailed aid from the programmer using the equivalent of such records and pointers. Realizing this possiblility of generalized construction **programming language facilities have been introduced which allow the programmer to define lists, trees, etc. directly** without dealing with the details of records and pointers. (ML)

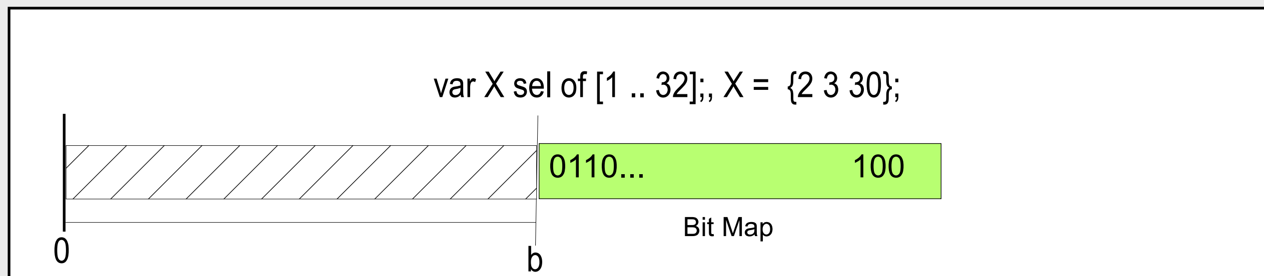
Today a programmer can, building on the data structures native to the programming language, construct their own data structures and define the operations to be performed on them through the Class construct. Using the Class construct as a **type** the connection between functions and data becomes explicit. Even prior to the aid of an explicit Class structure this idea of grouping a desired data structure, with appropriate functions like a Stack with *push( x )*, *pop(y)*, or a graph with *adjacent_vertex(i,v)* called an Abstract Data structure was available. It is a great aid. fo algorithm development.

But these higher level construct, the stack or graph, must be built from data types built into a language-these must be identified, often by declaration so the compiler can allocate storage for them and otherwise determine that they are used legitimately, namely with the appropriate operations. So the distinction of integers, fixedpoint, and floating numbers may on the surface be of little interest to the programmer, but distiguishing them allows targeted allocation for efficient use computer resources as well as chances for error checking.

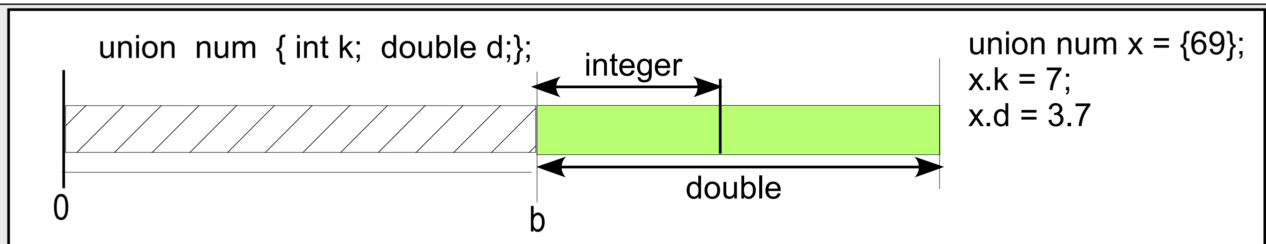
Layout Of Data Structures



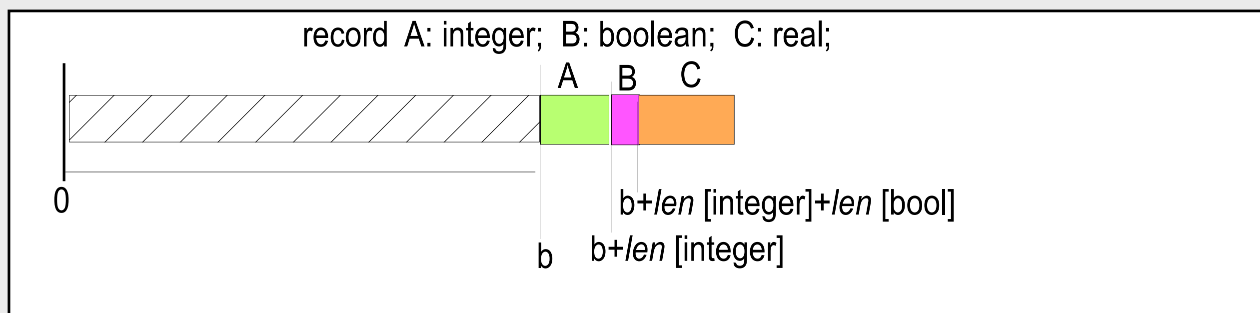

**BASIC TYPES**




**SET**




**UNION ( C )**




**RECORD**

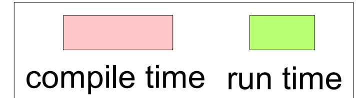
**LAYOUT @ COMPILE TIME, ALLOCATION @ COMPILE TIME**

© 2006 M C Paull

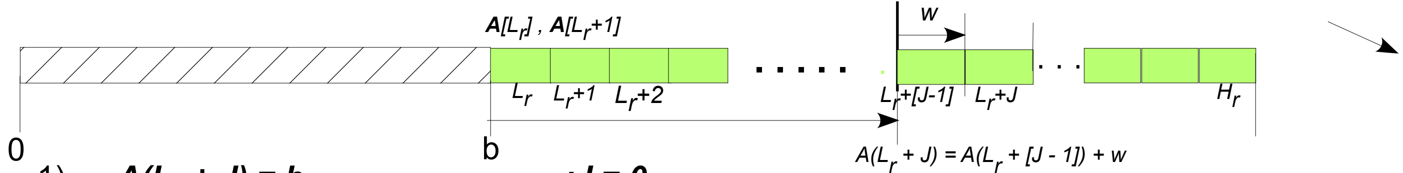
$L_r$ = low row index $H_r$ = high row index , $L_c$ = low column index, $H_c$ = high column index

$J+L_r$ = the index = $r$  $K+L_c$ = the index = c

$w$ = word size,  $R$ = row size = $wN_{columns/ow}$




**var A : array** [$L_R$ .. $H_R$] of **integer**




1) $A(L_r + J) = b$ $\quad :J = 0$

2) $A(L_r + J) = A(L_r + [J - 1]) + w$ $\quad :J > 0, L_r+J <= H_r$

**Difference Equations**

: $A(L_r + J) =$ $\boxed{b + Jw}$ $\quad :J >= 0, L_r+J <= H_r$

**Solution**

1) $b + Jw = b + 0w$ $\quad :J = 0$ OK

2) $b + Jw = b + [J - 1]w + w$ $\quad :J > 0$

$b + Jw$ ~~$- w + w$~~ OK

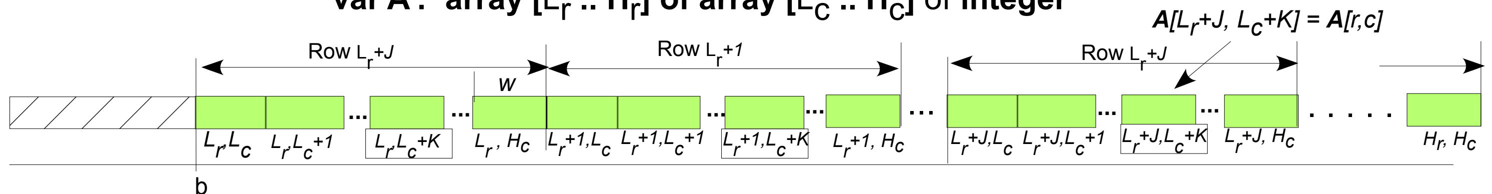
**Proof**

Let $r = L_r + J$, so $J = r - L_r$ and

Solution: $A(r) = b + (r - L_r) w =$ **$b - L_r w$** + **$rw$** : $\quad r-L_r >= 0, r <= H_r$ **Final Solution**

Significance: *$b - L_r w$ is an Array constant computed at Declaration, $rw$ is computed at each reference [A(r)]*

**var A : array** [$L_r$ .. $H_r$] **of array** [$L_C$ .. $H_C$] of **integer**




1) $A(L_r + J, L_c + K) = b$ $\quad : J = 0, K=0$

2) $A(L_r + J, L_c + K) = A( L_r + [J - 1], L_c + K ) + R$ $\quad : J > 0, K >= 0$

3) $A(L_r + J, L_c + K) = A( L_r + J, L_c + [K - 1] ) + w$ $\quad : J >= 0, K > 0$

**Difference Equations**

Solution $A(L_r + J, L_c + K]) =$ $\boxed{b + JR + Kw}$ $= A(r, c)$

**Solution**

Proof

1) $b + JR + Kw = b + 0R + 0w = b$ $\quad : J = 0, K = 0$ OK

2) $b + JR + Kw = b + [J - 1]R + Kw + R$ $\quad : J > 0, K >= 0$

$b + JR + Kw = b + JR - R + Kw + R$ OK

3) $b + JR + Kw = b + JR + [K - 1]w + w$ $\quad : J >= 0, K > 0$

$b + JR + Kw = b + JR + K - w + w$ OK

**Proof**

Let $r = L_r + J$, $c = L_c + K$ so $J = r - L_r$, $K = c - L_c$

Solution $A(r, c) = b + (r - L_r) R + (c - L_c)w =$ **$b - L_r R - L_c w$** + **$rR + cw$** **Final Solution**

**ARRAYS- Derivation of Layout Parameters**

**LAYOUT @ COMPILE TIME , ALLOCATION @ COMPILE TIME**

© 2006 M C Paull

**Layout and Allocation**

Layout = The relation of the Names to Locations of the elements of a Data Type
Allocation = The actual assigning of the Layout.

In C all Layout is done at statically at compile time, However like many languages actual allocation of local variables in a Procedure is done on Procedure entry But there are special variables, declared as **static**, allowed whose values though delcared in a procedure are not destroyed between procedure calls. These are allocated statically.

```
Int  make()
      {static char buffer[100];         Allocated at Compile time
              char temp[50];            Allocated in Runtime Stack  when make() is called (and
               ....                           lost on return from make().
               ....
      }
```

**STATIC STORAGE**

Global arrays are layed out and allocated at Compile time, but only pointers to arrays are passed into Procedures. C enforces this by creating a pointer to an array when the array is declared-the array name is a constant pointer-it will always point to thet array.

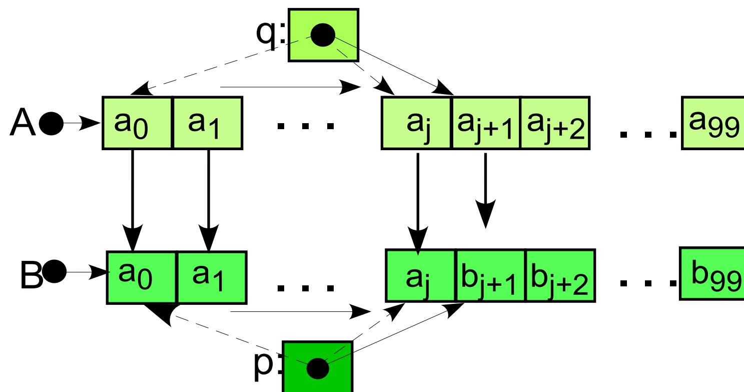
**Arrays and Pointers in C**

Declaration of an array in C

int A[100];

Results in A being a (constant) pointer to array A, with A + i being a pointer to A[i]. Associating pointers with arrays allows them to be passed more efficiently as (effectively reference) actual parameters.

Since A is effectively a pointer it can be assigned to another variable which is a declared pointer




```
char  A[100];
char  B[100];
int i=0;
char *p, *q  (p and q point to chars)
p = B[i];
q = A;
for(;;)
   {*p = *q;
     if (*p = EOS) break;
     p++;
     q++; }
```

**POINTERS AND ARRAYS**

**Dynamic Allocation Variable Sized Data Structures**

The size and layout of storage for each type is known statically i.e;at compiler time: Pointers too are layed out and allocated at compile time but the object to which they point (other than an array, generally for large data structures static allocation is much more efficient ), though layed out at compile time, is often allocated at runtime. So dynamic data-structures are made up of **units whose size and layout are determined statically, though the number of such units may be determined at runtime** The structures are built by linking these units.

1. In Pascal a pointer points only to dynamic objects. If p is declared to be a pointer to an object of type P then **new**(p) causes p to point to a newly created object of type P. The storage for dynamic storage items is kept in a global area, independent of the run-time stack, called the **heap.**
2. The de-referencing symbol is " p↑" is the object pointed to by pointer p
3. **Assignments are allowed** between pointers of the same type.
4. The equality(non equality) test between pointers of the same type are allowed.
5. **dispose**(p) releases the storage to which p pointed. it leaves the pointer p **dangling.**
6. There is a special type **nil** to which any pointer can point.

Assume that the program has performed a series of instructions on the path P, from its start to its the latest instruction, c. Assume at c that p points to some object, then the last assignment of the pointer p was (excluding **dispose**) either by **new**(p) on P or by an assignment of the form p **:=** $p_1$ In general assume at c that $p_1$ points to some object, then the last assignment of the pointer $p_1$ was either by **new**($p_1$) on P or by an assignment of the form $p_1$ **:=** $p_{2,}$ and so on. So if dispose is never used (or ignored) there can be no dangling pointers in other words no **memory leaks** no creation of **garbage**

```
cell = record;
          INFO: integer
          NEXT:link
       end;
type link =↑cell;
var p:link;
var f: link;
new(p);
p↑.INFO = 16;
f = p;
new(p);
p↑.INFO = 32;
p↑.NEXT = f;
f↑.NEXT = nil;
```

p, f, 16, 16, 16, 32, 32, 16, nil, 32

An infinity of names can be given types:

*T*(p) is link

*T*(p↑) is cell

*T*(p↑.INFO ) is integer

*T*(p↑.NEXT) is link

*T*(p↑.NEXT↑.NEXT ) is *nil*

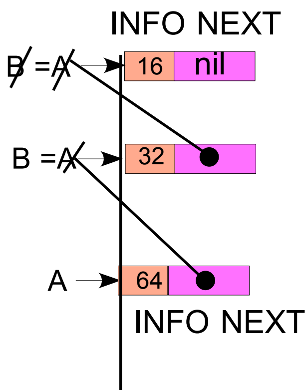
LINKED LIST (Built from Last to first cell )

© 2006 M C Paull

**This restriction-disallowing pointers to access any storage assigned to variables** means that the only data pointed to in Pascal is on the Heap-a global storage area. Therefore, when the top of run- timeStack is popped no pointer will be left dangling. This is called the Alias Avoidance principle.

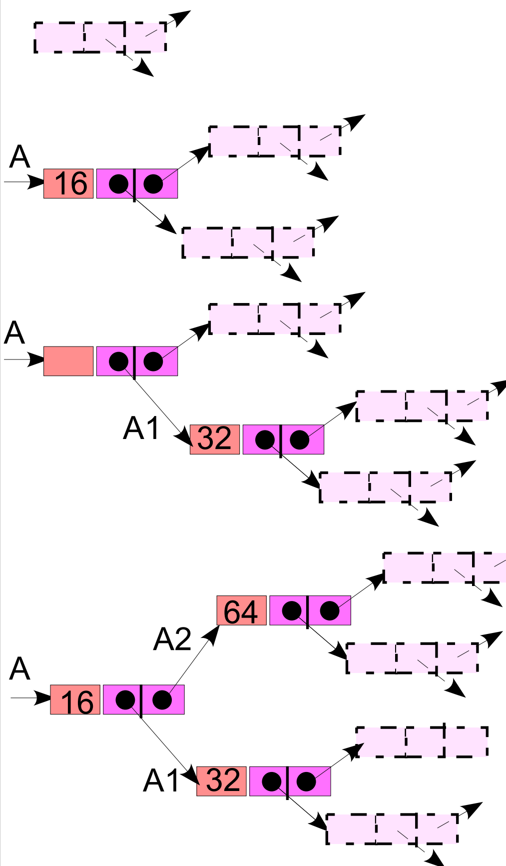
## Higher Level Language For Dynamic Data Structures

We have seen how Link List can be constructed out of the basic material of a record which has room for data and for pointers declared as pointing to a similar record. To do this construction one must detail the generation of new data of the given record type and juggle the pointers involved. The necessary work is formulaic. In fact one can generalize this dynamic construction technique and define a data structure which will generate a linked list and/or other uniform dynamic data-structures, and make them easier more natural to specify.




```
Type LinkList:    INFO[integer],  NEXT[LinkList];
var A, B [LinkList];
new(A);
INFO A = 16;
NEXT A = nil;
B = A;
new(A);
INFO (A) = 32;
NEXT (A) = B;
B = A;
new(A);
INFO(A) = 64;
NEXT (A) = B;
```

LINK LIST




```
Type Bi_tree INFO[integer] CHILD1[ Bi_tree ] CHILD2[ Bi_tree];
var A, A1,   [Bi_tree]:
new(A);
INFO (A) = 16;
new(A1);
CHILD1(A) = A1;
INFO A1  = 32;
new(A2);
CHILD2(A)= A2;
INFO A2  = 32;
CHILD2 A2 = A21
INFO A2  = 64;
B=A;
```

BINARY TREE


© 2006 M C Paull

T.Exp ----> T.Nm
----> Simp.T
----> **array** Simp.T **of** T.Exp [1]
----> **record** Nm **:** TExp **;** {Nm **:** TExp**;**}
----> **pointer to** TExp
----> **set o**f Simp.T
Simp.T ----> Basic.T | Enum | SbRng
Enum ----> **(** Nm { **,** Nm[2] } **)**
SbRng ----> [T.Nm] **[** Const .. Const **]**
Basic.T ----> **boolean** | **char** | **cardinal** | **integer** | **real**

T.Def ----> **type** T.Nm **=** T.Exp
T.Decl ----> **var** V.Nm **:** T.Exp

[1] if A is the array, Simp.T gives *range* in A[*range*] it can be a subrange [2 .. 7], A[5] for example, or boolean specifying a range of true A[true], or false , or char, specifying a range of ‘&’, A[‘&’], or “(” etc. Etc.

[2] Nm is an arbitrary nameT.Nm is a name assigned to a type in the expression **type** T.Nm **=** T.Exp

**Grammar For Subset Of Modula-2 Types**

const Lr = 1; Lc = 1; Hr = 100; Hc = 100;

**var A : array [**Lr .. Hr] of **integer**

**var A : array** [Lr .. Hr**] of array [**Lc .. Hc**]** of **integer**

**type** sam: **record** A: **integer;** B: **boolean** C: **real**

**var A : array** [Lr .. Hr**] of array [**Lc .. Hc**]** of **sam**

**type** Token **=** (exp, mult, div, sum, dif, eq)

**var** tok: **array char of** Token

tok[ ‘+‘ ] := sum

**var** S3 **: set of** [1 .. 5]

**var** y **: integer**

**type** complex=**record** re: **integer**; im: **integer**.

procedure init_complex( c1 **complex** )
{ var c1:**record** re:**integer**; im:**integer**;}

**Examples**

In this Grammar the arguments of T.exp s for (a linear) **array, record** and **pointer** are recursive in T.exp. Therefore we can define X1 of X2 of X3 of .... Xn where Xi can be any of those three T.exps. A two dimensional array is simply a linear array of a linear array, One can establish a type or a pointer to an array of records or an arra of pointers for example. which is an array of records.The T.Def allows one to name a type, ex. an name an array , say A, and the define a record of with components which are As (similar to the **let** in scheme with fewer parenthesis.)

| | |
|---|---|
| <, =, > .... | [ integer/ real X integer/real] =boolean |
| + - * / | real X real -->real |
| + - * / mod, div | integer X integer ---> integer |
| and, or | boolean X boolean ---> boolean |
| not | boolean ---> boolean |
| := | type left == type right |

**Operations For Different Types And Expression Type Results**

**AN (ALMOST COMPLETE) TYPE SYSTEM.**

© 2006 M C Paull

In this Grammar the arguments of T.exp s for (a linear) **array, record** and **pointer** are recursive in T.exp. Therefore we can define X1 of X2 of X3 of .... Xn where Xi can be any of those three T.exps. A two dimensional array is simply a linear array of a linear array, One can establish a type or a pointer to an array of records or an array of pointers for example. which is an array of records. The T.Def allows one to name a type, ex. an name an array , say A, and the define a record of with components which are As (similar to the **let** in scheme with fewer parenthesis.)

**var A : array** char **of** char



```
T.Exp   ----> T.Nm
        ----> Simp.T
        ----> array Simp.T of T.Exp [1]
        ----> record Nm : TExp ;
                {Nm : TExp;}
        ----> pointer to TExp
        ----> set of Simp.T
Simp.T ---->  Basic.T | Enum | SbRng

Enum   ---->  ( Nm { , Nm [2] } )
SbRng  ----> [T.Nm] [ Const .. Const ]
Basic.T ----> boolean | char | cardinal |
                integer | real
T.Def  ---->  type T.Nm = T.Exp
T.Decl ----> var V.Nm : T.Exp
```

**var A : array** [Lr .. Hr**] of array [**Lc .. Hc**]** of **sam**



**Layout Of Nested Data Structures-Attribute Grammar Approach: Parse-Decorate**

© 2006 M C Paull

Typing and Error Checking

Agreement of operations and Operands-

For Basic Data Types:

Most Stringent operation takes 1 type, Result of operation on a type yields a type.
Least Stringent operation takes a number of Types (Poymorphism) ex. +: Reals and Integers.
Must specify the type of the result of an operation if polymorphic (usually the least restricted of types.)

**Strict**

| Operation | Types |
|---|---|
| <, =, > .... | integer [rea]l X integer / [real] ---> boolean |
| + - * / | real X real -->real |
| + - * / mod, div | integer X integer ---> integer |
| and, or | boolean X boolean ---> boolean |
| not | boolean ---> boolean |
| % | integer X integer ----> integ |
| := | type of result (returns a value in C) ? |

**Alternatives**

| Operation | Types |
|---|---|
| + - * | real X real -->real |
| | integer X integer---> integer |
| | ireal X integer, real= integer X rea ---> real |
| % | real X integer ----> integer |
| | integerX integer ----> integer |
| | real X real ---> ?? |

For more Complex Structured ,Data Types-

What aspects of the Type (in particular structural parts, sizes, ranges, etc.) should be considered in deciding equivalent of types .for Assignments, and other operations A := B, type(A) must be compatible with type B.

**Type Expressions Type Equivalence**

For any operation on variables **A op B** to be legitimate it seems that A and B must be of equal (or close to equal types)

The examples above illustrate the reasonableness of having equal or closely related (equivalent) types as arguments of operations. This includes assignments of basic types x := y too . For a function f: $T_1$----> $T_2$ for any argument of f ought to be of the same type for which the function was defined-namely type $T_1$. For simple types, **integer** or **char** equivalence is easily determine,

For more complex data structures it becomes more difficult.

The question then arrises as to what is to be considered to be of the same type-are arrays of the same size and type, but with different names, the same type? Each language must specify what are to be equivalent types and where there use is allowed. For assignments one might only allow assignments between a limited number of types--**arrays cannot be assigned to each other in C whether equivalent or not**.on the other hand one may allow arrays to be passed to procedures as long as they are equivalent to the ones for which the function was designed. So it is necessary to know when two arrays,or two records are to be considered of the same type. Obviously it depends on the type expressions which established the types when they were declared (but these can be quite complex nd non-obvious given that new types with new names can be established in most modern languages..

First recall that there are basic types, **integer,real,boolean**, etc. then there are first level types which are built by applying first level **type constructor:** (i.e. ,**array**[0..20 ]**of <type> end,** or **record <nm$_1$: type$_1$> ... <nm$_n$: type$_n$>end;** to a basic types to get data types **array**[0..20]**of integer end,** or **record <father: real> ... <uncle: boolean>end;**
and also we can apply first level type constructors to basic types *and* other first level constructors and their basic types.
i.e.,(i.e.,**array**[0..20] of **record:** father:**real**;mother:**integer end**. **pointer** to **real,**

More precise definitions of type equivalence, applicable to structured types, follows This requires a recursive specification of when each type constructor is to be considered equivalent. This may be done in a number rof different ways each of which is reasonable {though one will require a more complex selection algorithm than another.

**Name-Equivalence**(NE) and ,**Structural Equivalence**(SE) are two extremes of type equivalence. These are outlined togerther with some variations . NE is simple, and general and very safe though it ignores unnamed types.SE is more complex. It is outlined here for a simplified language.This language f has a variety of built-in types including: all basic types, integer, real etc., and type constructors limited to arrays, and records and the facility for naming new data types built from these All the equivalences defined here, both SE and NEs. assume a variable is given a type by an expression of the form: **var** X <type-expression>. Note **equivalences** are is reflexive, transitive and symmetric

**Name Equivalence:** Two types $T_1$ and $T_2$ are NE iff they have the same name

**Structural Equivalence for an abbreviated language:**

Two type expressions $T_1$ and $T_2$ are SE if and only if:

1 They are both identical basic types,ex both ( both integer or both real) or *same name* of a declared type

**2**. $T_1$ = **array**[$i_1..i_2$] **of** $S_1$ and $T_2$ = **array**[$j_1..j_2$] **of** $S_2$ , and $S_1$ and $S_2$ are SE. and $i_1 =_. j_1$ and $i_2 =_. j_2$

$T_1$ = **array**[X] **of** $S_1$ and $T_2$ = **array**[Y] **of** $S_2$ , and $S_1$ and $S_2$ are SE. and X and Y are SE

3. $T_1$ = **record** $a_1$:$T_1$, ...,$a_m$:$T_m$ **end** and $T_2$ = **record** $b_1$:$S_1$, ...,$b_p$:$S_p$ **end** and p = m, and for i = 1 to m, $a_i = b_i$ $S_i$ and $T_i$ are SE.

4. If a there is type declaration: **type** n =T then n and T are SE.

**Other possiblities:** involve using subsets of the set of rules of Structural Equivalence.

1. **type** A = **array** [0..10] **of integer**
2. **type** B = **array** [0..10] **of integer**

With declarations x:B; y:A; x and y are SE **(2)** But they are not NE.

---

3 **type** V = (a, e, i, o, u)

4a. **type** C =**record** z:**integer**; x: V **end**

, Fully expanded = **record** z:**integer**; x: (a, e, i, o, u) **end**

4b **type** C' =**record** x: V; z:**integer end**

C and C' are not SE. A record type includes the names of its parts in order,(Though the ordering is not important in use.

---

5a. **type** D=**record** z:**integer**; x: V **end**

Fully expanded =

**record** z:**integer**; x: (a, e, i, o, u) **end**

5b.. **type** E=**record** z:**integer**; x: (a,e,i,o,u) **end**

With declarations var: x:D; var: y:E, x, and y are SE (3),

---

6. type F = **array** V of C

7. **type** G = **array** (a, e, i, o, u) **of** D

Fully expanded

**array** (a, e, i, o, u) **of** record z:**integer**; x: (a, e, i, o, u) **end**

F and G are SE(2,3)

Generally these instances of such complex types are not normally assignedone to the other but they may be passed as parameters

C uses structural equivalence except for records

Modula Allows subranges of each other or both subranges of the same basic type

EXAMPLES

© 2006 M C Paull