JOINT ROYAL MELBOURNE INSTITUTE OF TECHNOLOGY AND
COMMONWEALTH SCIENTIFIC AND INDUSTRIAL RESEARCH ORGANISATION
PARALLEL SYSTEMS ARCHITECTURE PROJECT

IDA
A Dataflow Programming Language

TR 112 075R

P. G. Whiting

Division of Information Technology
C.S.I.R.O.
c/o Department of Communication and Electronic Engineering
Royal Melbourne Institute of Technology
124 Latrobe St
Melbourne 3000
Australia

Version 1.0 October 1988

Abstract

This paper describes the IDA programming language - a language designed to make efficient use of the RMIT/CSIRO dataflow computer. It is a functional language, based upon M.I.T's Id Nouveau which makes use of parallel architectures implicitly - i.e., the programmer does not have to use special constructs when coding algorithms. The compiler determines which parts can be executed concurrently and generates machine code to take full advantage of the underlying architecture.

Table of Contents

List of Examples

List of Tables

Chapter 1
Introduction

This report describes the IDA programming language, designed primarily for use on the RMIT/CSIRO Dataflow Machine [1, 2]. This language is *functional* since all expressions in a program return a value and are free of *side-effects*[1], which are the inadvertent or intentional changing of the global environment through the execution of a expression statement. These properties allow IDA to make use of the unique environment offered by a dataflow computer which provides fine-grained parallelism.

The Joint RMIT / CSIRO Parallel Systems Architecture project officially commenced in May 1986 as a collaborative project between the Royal Melbourne Institute of Technology and the Commonwealth Scientific Industrial Research Organisation. The purpose of the project is to investigate parallel algorithms, methodologies, languages and machine architectures. Given the resources, the project has concentrated on the dataflow model of computation [2].

The variant of dataflow being used is based on an architecture designed in 1976 by Egan at Manchester University, UK [3]. A multiprocessor emulation facility is available for high speed emulation of dataflow programs as well as a conventional discrete event simulation of the architecture. As well as IDA (a version of Id Nouveau [4]), compilers are under development for GHC (Guarded Horn Clauses, a logic programming variant providing parallelism & committed choice non-determinism[5]), and SISAL (Streams and Iteration in a Single Assignment Language [6]), an applicative language for exploiting parallelism In addition, DL1 (Dataflow Language 1) [7] was completed in the early stages of the project. Work is currently progressing on the design of high speed processing elements which for use in a high speed multiprocessor computer.

IDA (Id Nouveau Australia) is a subset of MIT's Id Nouveau [4] which was designed for their Tagged-Token Dataflow Architecture [8]. Id Nouveau removes from the programmer the burden of specifying which parts of the source code are to be executed in parallel by using information implicit in the data dependencies in the program. This also means that the programmer does not need to change his code if the configuration of the target computer is altered.

To take advantage of some of the unique features of the RMIT/CSIRO Dataflow Machine, it was decided to produce a compiler for Id Nouveau, and this would allow the designers of the compiler to include some of their own ideas for improving a dataflow language. Therefore, a subset of Id Nouveau was designed as this provided source code compatibility and yet still allowed us the freedom to include our own design choices.

When the writing of the IDA compiler began, Id Nouveau did not have any form of data-structure typing[2], however it was felt necessary to include a typing scheme as this allows better code for the underlying architecture to be produced and assists with the coding or debugging of IDA programs. The typing scheme used is based on Pascal's and allows user-defined types.

Besides the type-checking we have also introduced a new function into the Id Nouveau syntax - *sequence*. This function allows us to provide input / output routines

---

[1]Later it will be shown that this is not entirely true.

[2]Type checking is now included in the subsequent releases of Id Nouveau.

which the original Id Nouveau and most other dataflow and functional languages omit.

Finally, one of the features of the underlying architecture is that it has instructions to work with the stream data structure [9]. The IDA compiler includes streams as a pre-defined type along with high level operators for manipulation.

1.1 Differences between IDA and Id Nouveau

Beside the differences already mentioned: the data-typing scheme, streams, the sequence function and input/output, there are a number of differences between the two languages. As this is the first release of the IDA compiler, there are some features which have not been included in this release. Here is a list of these differences and omissions:

- Prefix operators.
- Curried functions
  In this release of the compiler curried functions have not been implemented, but the type-checking has been coded to generate an error only when too many arguments have been supplied to a function.
- Implicit and Polymorphic typing -
  A subsequent release of MIT's Id Nouveau development system Id World, has described their typing scheme based on these two methods. Therefore to maintain source-code compatibility with Id Nouveau, later releases of the IDA compiler will also include these typing schemes, which will be user selectable.
- Stream and Union data types and operators -
  not implemented in this release,
- Sequence function and Input/Output -
  not implemented in this release, and
- Standard procedures (chapter 10) -
  not implemented in this release.

1.2 The IDA programming language overview

As with an Id Nouveau program, an IDA program is made up of
a) a collection of definitions (or declarations), and
b) an expression (also called the "query").
A definition is the mechanism whereby an expression or value is named. If a definition includes parameters then it signifies a procedure definition[1]. The query initialises program execution and produces the final result. The general structure of an IDA program is illustrated in the following example :-

```
% program to calculate the volume of a sphere
% definitions

def pi = 3.142;

def power b:number e:number returns number =
if (e = 0) then
   1
else
   b * (power b (e - 1));

def spherevol r:number returns number =
4/3 * pi * (power r 3);

% program body and query

let
   var radius : number;{ declaration }
   assign
   radius = 10
in
   spherevol radius;
```

Example 1.1 - Sphere Volume

In this example it can be seen that the definitions occur outside the query *Let-Block* and include both the binding of a value to a name as well as function definitions (`power` and `spherevol`). A Let-Block consists of two parts - the *Let-Assignments* and *Let-Return* sections. The first section is used for binding names to expressions or values used either in subsequent statements within the Let-Assignments section or in the Let-Return section. The second section is an expression which defines the value returned by the Let-Block. In this case, we see the declaration of the identifier `radius` and its binding to the value `10` within the Let-Assignments section and the return expression `spherevol radius` in the Let-Return section. It is permissible for type declarations to occur outside the Let-Block, and for nested definitions to appear within the Let-Assignments section of the Let-Block.

[1]It is possible to define a procedure without parameters.

Chapter 2
Tokens and Constants

Tokens are the smallest meaningful units in a IDA program and structurally correspond to the words and punctuation of an English sentence. The tokens of IDA are classified into *special symbols, identifiers, numbers,* and *character strings.*

The text of an IDA program consists of tokens and separators, where a separator is either a blank or a comment. Two adjacent tokens must be separated by one or more separators if each token is an identifier, number or word symbol (keyword or reserved word).

No separators can be embedded within tokens, except in character-strings.

2.1 Character Set and Special Symbols

The IDA programming language uses the ASCII character set, with letters, digits and blanks being subsets of this character set:

- *letters* are those in the English alphabet [A .. Z and a .. z],
- *digits* are based on Arabic numerals [0 .. 9], and
- *blanks* consist of the space, tab and end-of-line (CR) characters.

Special symbols and *word symbols* are tokens having one or more fixed meanings. The following characters and character pairs are special symbols :-

+ - / * ^	arithmetic operators
=	assignment and equality operator
< >	comparison operators
[ ]	array selection
,	tuple and parameter separator
( )	parameter closure
:	type allocator
;	statement separator
<>	not equal
<=	less than or equal operator
>=	greater than or equal operator
..	subrange symbol
%	original comment (Id Nouveau)
(* *)	block comment (IDA)

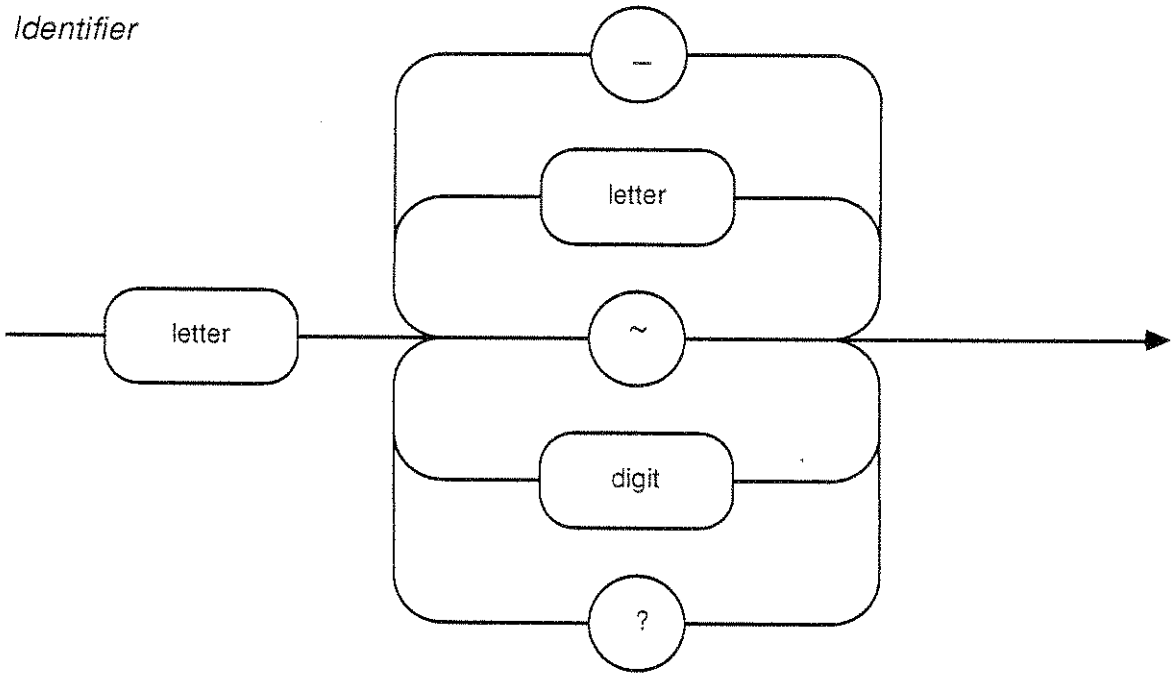
The following are the reserved words :-

```
and        array     boolean   by       char
const      def       do        downto   else
for        from      if        in       let
new/next   nil       not       number   or
returns    string    then      to       tuple
type       union     var       while
```

The case of the letters in word symbols is ignored.

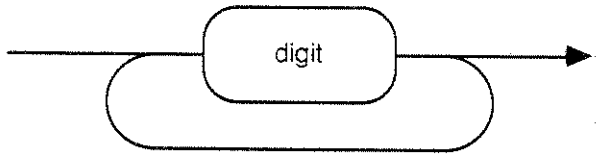
2.2 Identifiers

Identifier

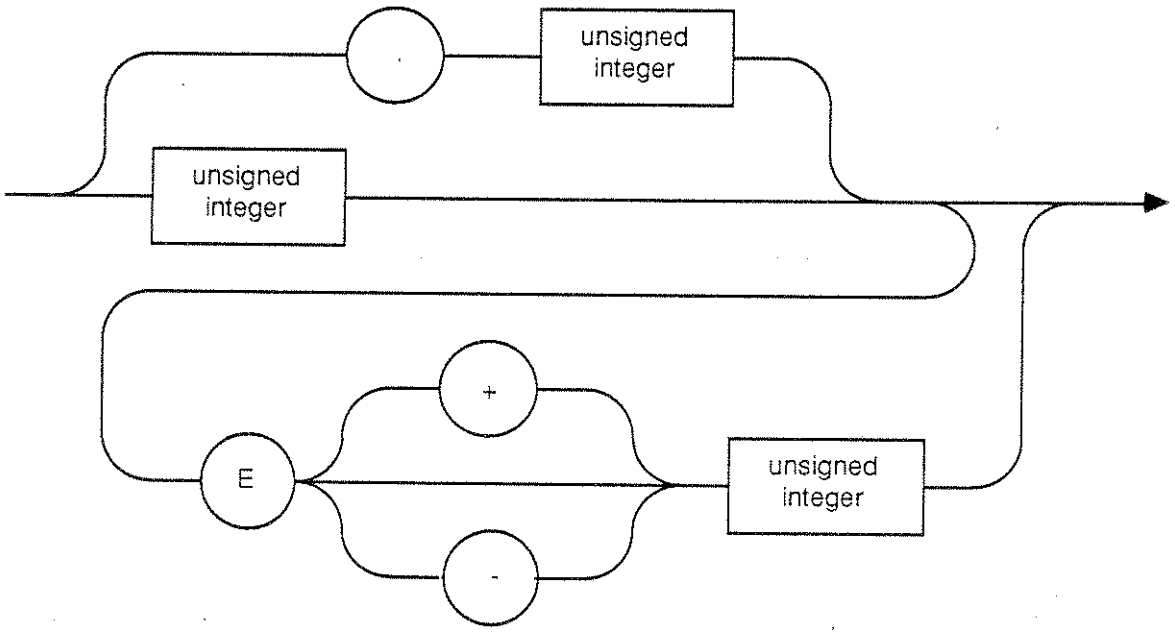


2.3 Numbers

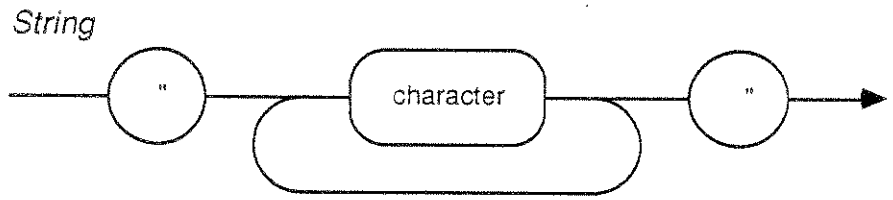
Unsigned Integer



Unsigned Number



2.4 Character Strings



2.5 Comments

When programming, it can be useful to place notes explaining your code by using comments. Such comments are ignored by the compiler and treated as blanks and therefore do not effect your program in any way.

Traditionally, Id Nouveau signifies a comment by using '%' at the start of the line, and all text up to the next end of line is ignored, for example :-

```
% this is an example of the traditional Id Nouveau comment
% used to explain a complex piece of programming
```

This is satisfactory to describe the finer details of some code, but what if you do not want to delete a section of code and do not want the compiler to check it. The answer is to comment the entire block of code and it is unreasonable to expect the programmer - to place '%' in front of every line, Therefore IDA also includes a block comment facility (which may be nested), achieved by surrounding a block of code with the special symbols '(*' and '*)' as in the following example.

```
(*
if flag then
   val = min v1 v2
else
   val = max v1 v2;
*)
```

2.6 Compiler Files

The source code of an IDA program must be in a text file and comply with the file naming conventions for UNIX with the suffix `.ida` so that the IDA compiler can identify it as being an IDA source file. The compiler produces an output file which has the `.ida` extension stripped off and replaced with `.ifl`.

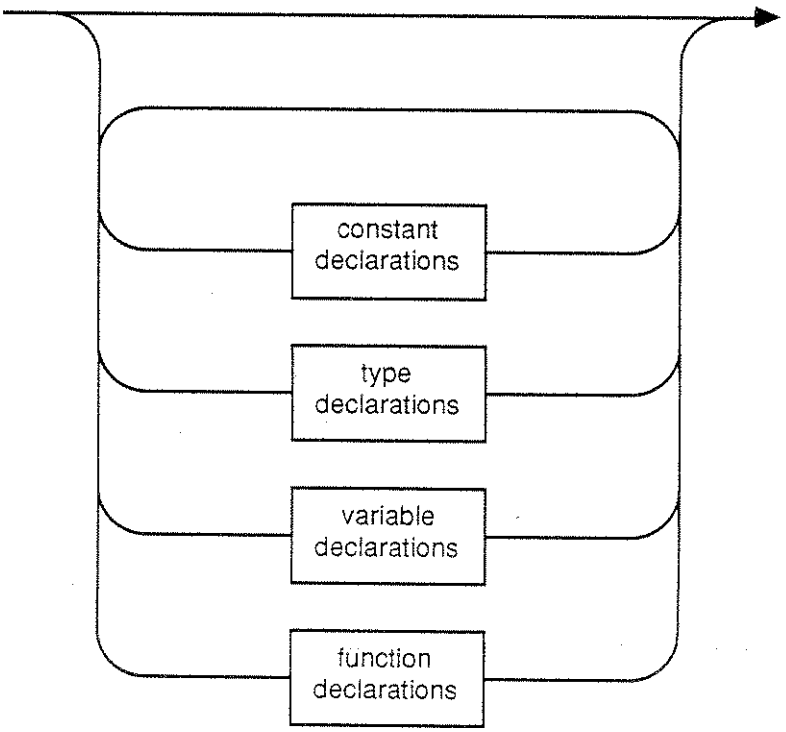
Chapter 3
Blocks, Scope and Activations

3.1 DEFINITION OF A BLOCK

A *block* consists of a *declaration-part* and a *tuple-expression-part*. Every block is part of a function-declaration or a Let-block (the body of most IDA/Id Nouveau programs). All identifiers that are declared in the declaration-part of a block are *local* to that block.

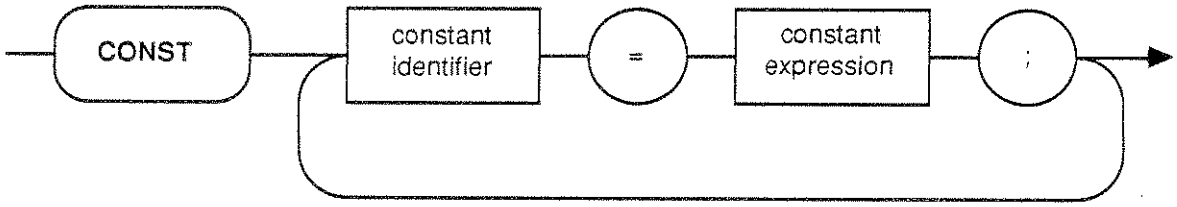
Block

Declarations



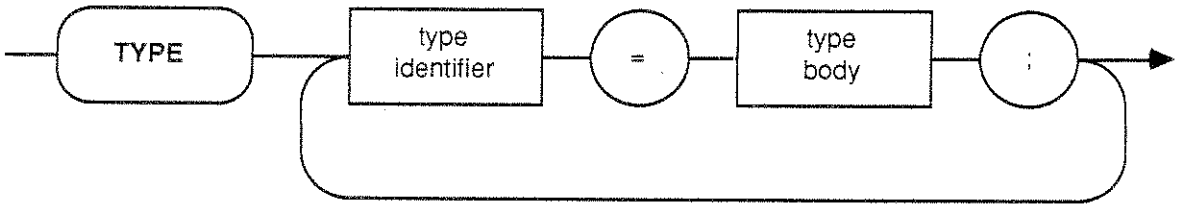
A *constant-declaration-part* contains declarations which bind identifiers to constant names for the duration of the current block.

Constant Declarations



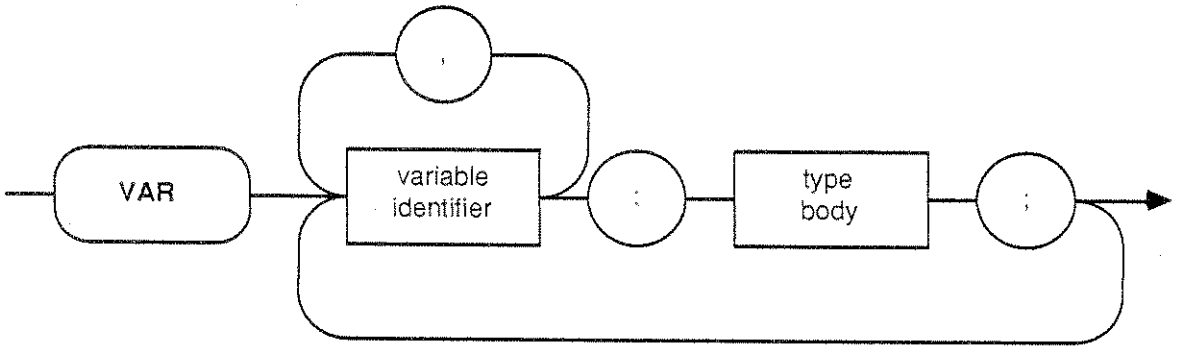
A *type-declaration-part* contains *type-declarations* (see Chapter 4) which allow the programmer to create his own types to be used when designating the attributes of identifiers. The type-declaration exists for all child blocks enclosed within the defining parent block .

Type Declarations



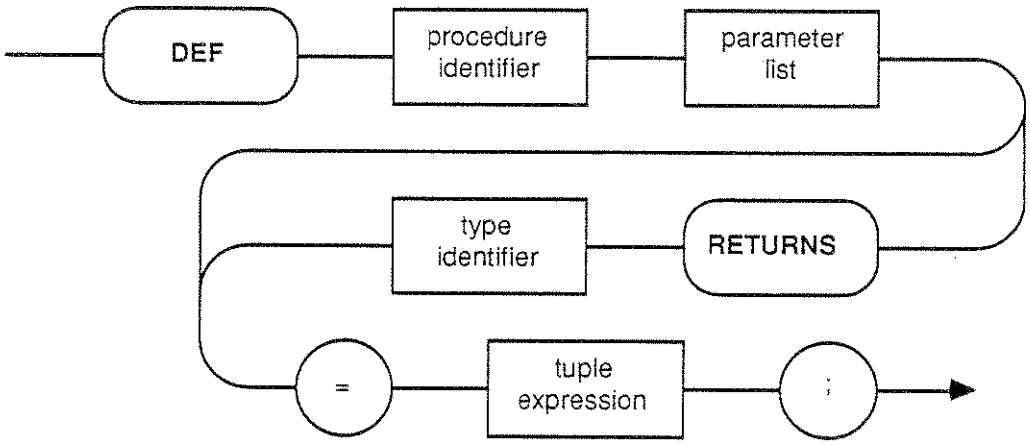
A *variable-declaration-part* contains *variable-declarations* (see Chapter 5) which are the means by which the programmer associates a type (or attribute) to an identifier. The variable-declaration exists for all child blocks enclosed within the defining parent block. Variables occuring on the left-hand side (LHS) of an assignment must be declared within the block that the assignment occurs in - while variables on the right-hand side (RHS) can be defined in any preceding block.

Variable Declarations



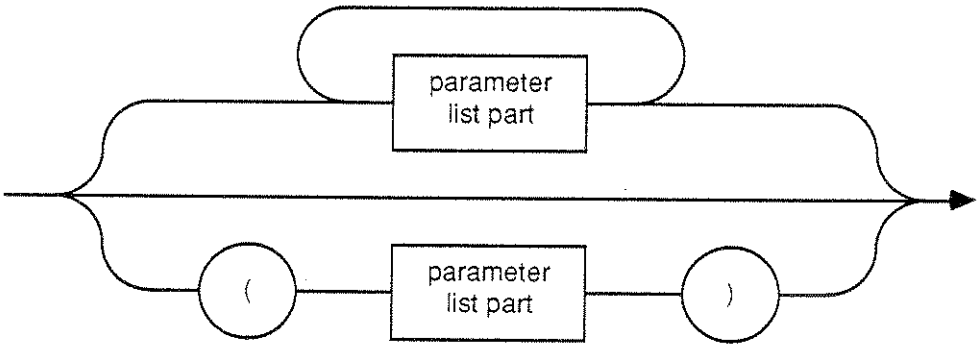
A *procedure-declaration-part* contains *procedure-declarations* (see Chapter 8) which are used to define segments of code which are used repeatedly from different sections of a program. The procedure-declaration contains 3 parts - the input interface (parameters into the procedure), the body of the procedure (segment of code), and the output interface (the attribute of the value returned by the procedure). The procedure declaration only exists for the current block of the program.

Procedure Declarations

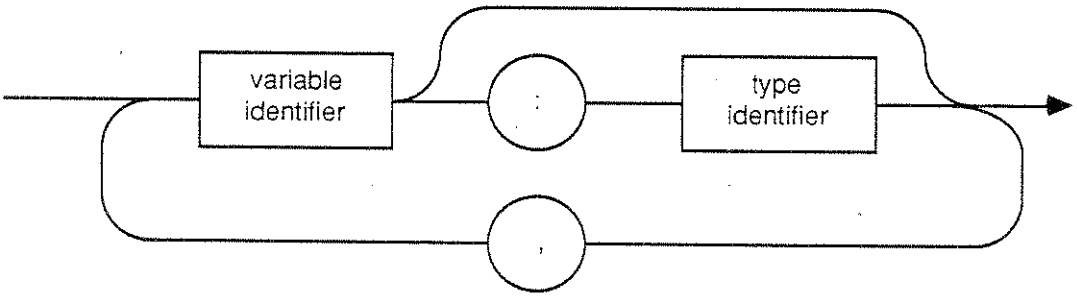


A *tuple-expression-part* specifies the algorithmic actions (see Chapters 6) to be carried out in executing the current block.

Parameter List



Parameter List Part



3.2 RULES OF SCOPE

3.2.1 Scope of a Declaration

When an identifier is declared in a declaration-part of a program, the programmer is *announcing* its existence and type to the compiler. This allows the compiler to check that it is being used correctly, each time it discovers the identifier in a portion of code. Each occurrence of an identifier must appear within the scope of its declaration.

The scope of a declaration is the block or segment of code which contains the declaration and all subsequent blocks *nested within* that block (see Sections 3.2.2 & 3.2.3).

3.2.2 Redeclaration in an Enclosed Block

```
let   (* first block *)
   var
      alpha, delta, gamma : number;

   assign
      alpha = 11;
      delta = 13;
      gamma =
         let (* second block *)
            var alpha, beta : number;

            assign

            alpha = 0.1;
            delta = 0.2
         in
            alpha + beta;
      (* alpha is now equal to '11' *)
in
   (alpha + beta) / gamma;
```

In this code segment there are two blocks of code, one contained within the other. Now if an identifier `alpha` is declared in the first block and then `alpha` is declared again in the second, there is two instances of the same variable. The scope of an identifier extends to the end of the current block and any blocks contained within it - unless, within an inner block there is another declaration of the same variable to replace the first. On leaving the scope of the second declaration, the attributes of the first declaration are returned to the identifier. This is the case with the identifier `alpha` - it starts with the attributes of type `number` and the value `11` in the first block and on entering second block these attributes are replaced with type `number` and value `0.1`. On leaving the scope of the second block, and entering the scope of the first block, the attributes of `number` type and value `11` are returned to `alpha`.

3.2.3 Position of Declaration within its Block

The declaration of an identifier must be placed at the start of the block so that it precedes all instances of the identifier in the following code - i.e., identifiers cannot be used until they are declared. It is not permissible to declare an identifier more than once per block.

3.2.4 Identifiers of Standard Objects

IDA provides a set of standard (predeclared) constants, types and procedures which can be used anywhere throughout an IDA program. The constants comprise of :-

- `nil`, `true`, and `false`

The types comprise of :-

- `boolean` and `number`.

while the procedures are listed in Chapter 10.

3.3 ACTIVATIONS

The activation of a block can be described as the execution of that block. Normally a block is either *inactive* (it is not being executed) or *active* (currently executing), but it is possible to have multiple activations of the same block if it is *recursive*[1] or *mutually recursive*[2].

The execution of the procedure `power 4 4` (as defined below), would lead to four activations of the procedure `power`

```
def power b : number n : number returns number =
if (n = 1) then
   b
else
   b * (power b (n - 1));
```

Example 3.1 - Procedure Power.

```
power 4 4
      = 4 * (power 4 3)
      = 4 * 4 * (power 4 2)
      = 4 * 4 * 4 * (power 4 1)
      = 4 * 4 * 4 * 4          = 256
```

Example 3.2 - Activations of Power 4 4.

`power` repeatedly calls itself, creating new activations, until the condition `(n = 1)` is met. When this occurs instead of calling itself again, the procedure returns the value of `b` to be used in a previous activation of the procedure. This continues until execution returns to the initial activation of the procedure which then returns the final value.

Every activation of the procedure (block) has a separate copy of its parameters and variables available, therefore these copies do not in any way affect the attributes of identifiers in previous or later activations.

It is important to remember that identifiers may only have values assigned to them once. This is referred to as *single assignment*[3] due to the nature of the model, and before this assignment, the value is *undefined*. Unlike traditional programming languages, the programmer cannot *initialise* variables because when a variable is assigned its actual value, the compiler will generate a multiple-assignment error.

---

[1]A recursive block (normally occuring within a procedure) is one which calls itself.

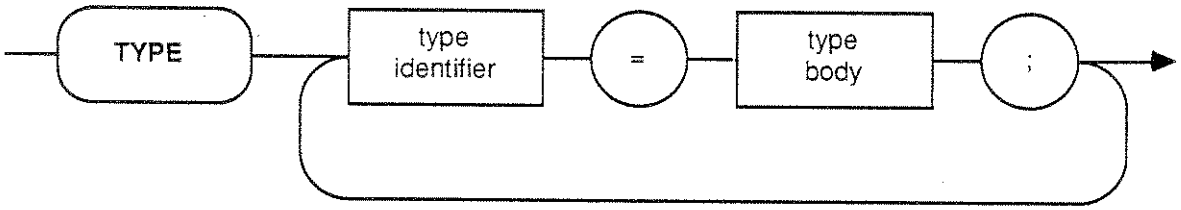
[2]A mutually recursive block (requires two or more procedures which call each other) invokes another block which in turn invokes the first block again.

[3]Identifiers in different activations of a loop body are logically distinct even if they have the same name.

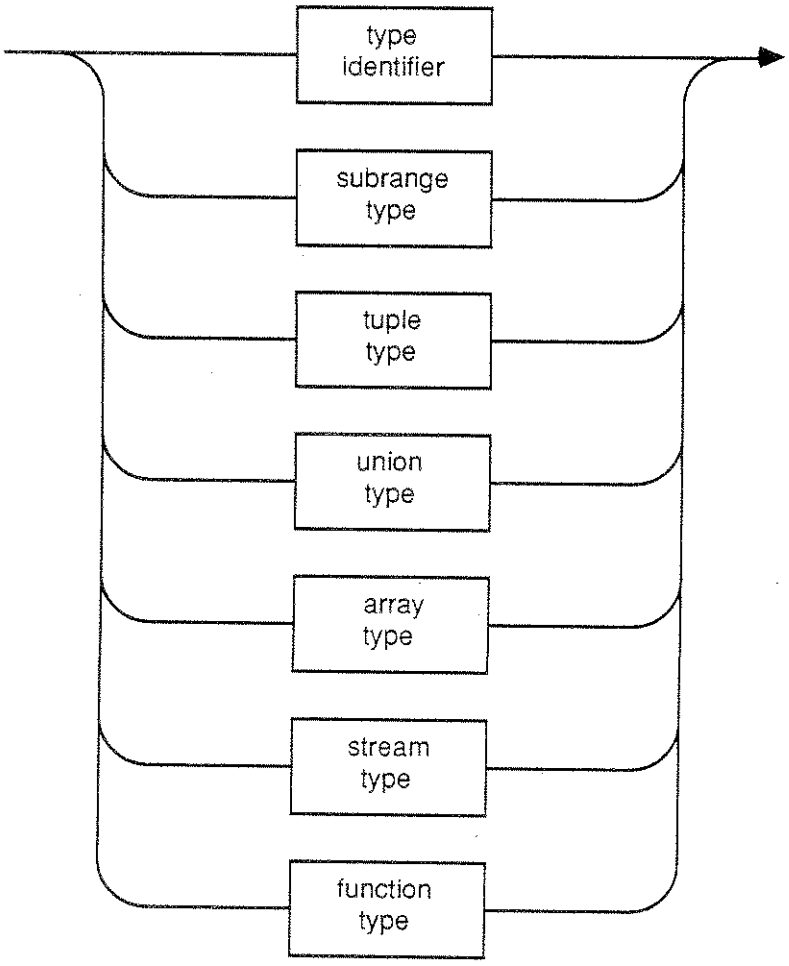
Chapter 4
Types

A *type* is used by a variable to associate the set of values and the range of operators that the variable can correctly use. A programmer can create new types to tailor a program to a particular application. A *type-declaration* associates an identifier with a type.

Type Declarations



Type Body



When an identifier appears on the left-hand side of a type-declaration, it is bound to the attributes on the right-hand side. A type-identifier can be used anywhere within the block in which its type-declaration appears. In IDA, a type-identifier may not appear in its own declaration unless it is a data structure type such as a tuple (see Section 4.3.1) or union (see Section 4.3.2).

4.1 BASIC TYPES

4.1.1 Numbers, Strings, Booleans and nil

IDA numbers can be only of one type : `number` which consist of a sequence of digits with an optional decimal point or can be expressed using *scientific notation* with a *mantissa* and an *exponent*, e.g. :-

```
15    15.0    0.0045    9.768E8    -6.765E-2
```

Strings are made up of any series of characters enclosed within double-quotes ("). Nested strings are permitted, but must be signified by pairs of double-quotes ("").

```
"This is an example of a string"
"this is an example of a ""nested"" string"
```

There are two boolean constants (after all you can't have anymore than two - can you?), written as

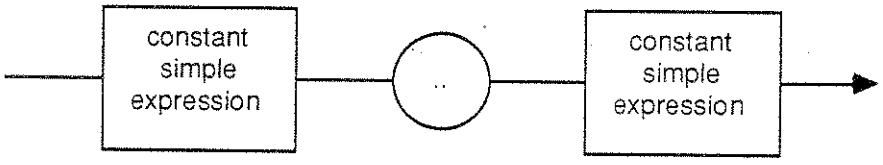
```
true                false
```

Finally there is the special value - `nil`. It is used to terminate identifiers and fields when using data-structures such as unions and streams and building recursive structures such as lists and trees.

4.2 SUBRANGE TYPES

A *subrange-type* is used to define a subset of values that lie within a certain range.

Subrange Type



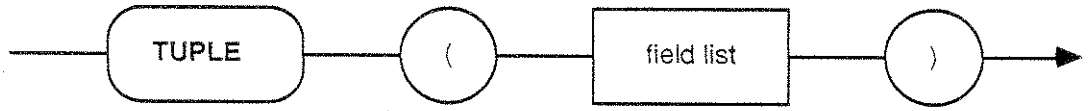
Both constants used to define a subrange must be of type integer, with the first constant (referred to as the *lower-bound*) being less than the second constant (*upper-bound*). Subranges are most commonly used to declare the dimension of arrays (see Section 4.3.3), and an error is given if an attempt is made to access a value outside the bounds.

4.3 STRUCTURED TYPES

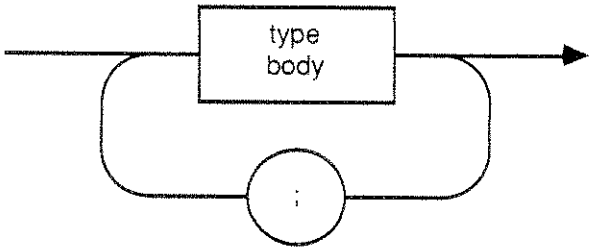
4.3.1 Tuple Types

A *tuple-type* is a structure which consists of a fixed number of components called *fields*, each of which may (but doesn't have to) be a different type. One of the fields can even be of the same type as that being defined.

Tuple Type



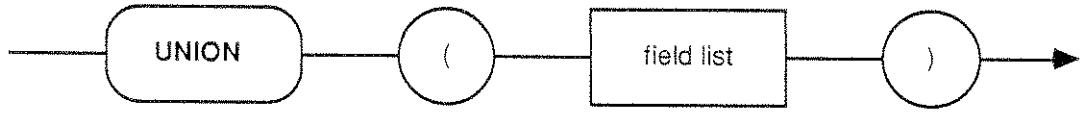
Field List



4.3.2 Union Types

A *union-type* is a structure which consists of a fixed number of components called *fields*. The fields can even be of the same type as that which allows recursive data-structures (section 5.4 explains the difference between tuples and unions).

Union Type

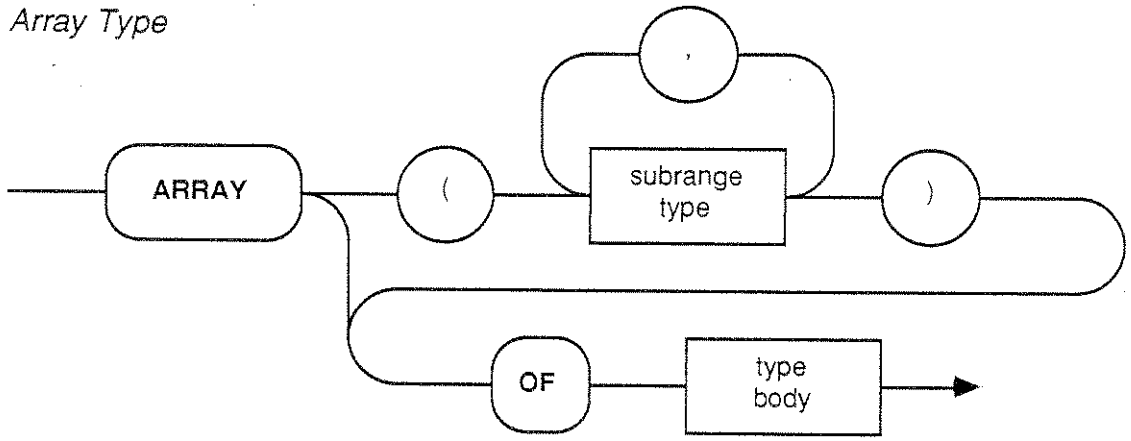


4.3.3 Array Types

An *array-type* can be likened to a one-dimensional table of components that are all of the same type (called the *component-type* of the array) and addressable by an index.

The component-type of the array is determined by the type following the reserved word `of`. The size of the array is determined by the subrange and this means that IDA arrays must be indexed by integers. Although the user can only work with variables of type `number`, the compiler converts these into integers so that the arrays can be indexed.

Array Type



If the subrange field between the parenthesis '(..')' contains more than one subrange, then this denotes a multi-dimensional array with the number of subranges representing the number of dimensions. In the example below, there are 3 subranges, so `hospital` is a 3-dimensional array of type boolean used to determine if a bed is vacant.

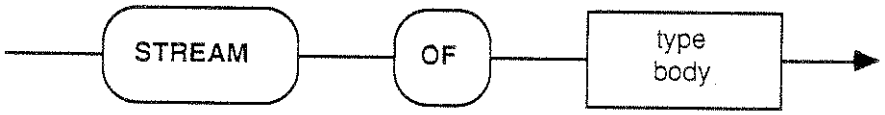
```
levels = 1..10;
rooms = 1..8;
beds = 1..4;

hospital = array (levels, rooms, beds) of boolean;
```

Example 4.1 - Array Example

4.3.4 Stream types

A *stream-type* [9] is a theoretically infinite list of heterogeneous data items used for file input/output and producer/consumer applications.

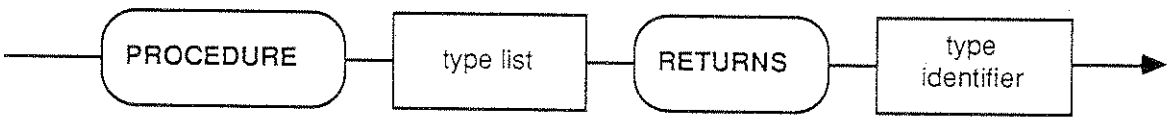
Stream Type



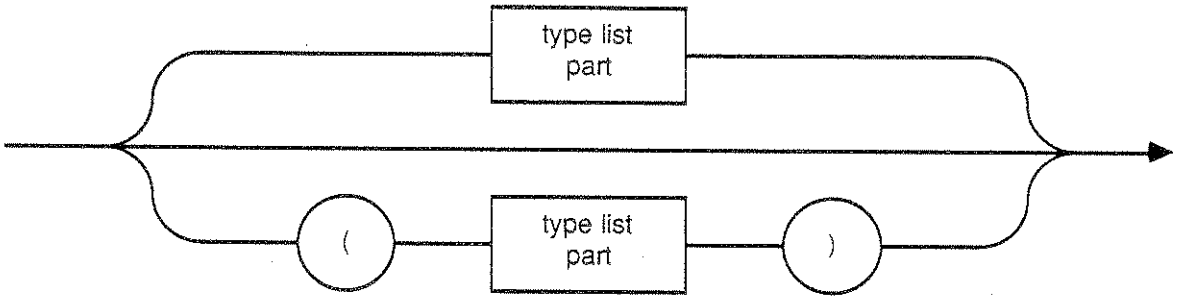
4.4 PROCEDURE TYPES

A *procedure-type* is used to declare a procedure to be used as an object capable of being passed to a procedure as a parameter or returned as the result of a procedure (see Chapter 8).

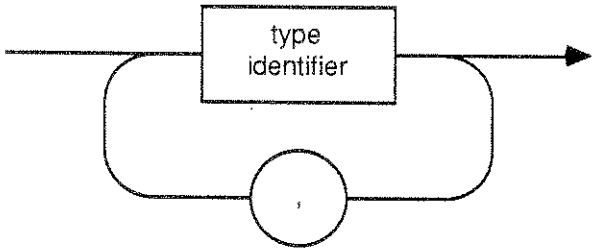
Procedure Type



Type List



Type List Part



4.5 IDENTICAL AND COMPATIBLE TYPES

When writing IDA programs there are three possible conditions concerning the types of identifiers and expressions. These are that the types are:

1) *identical*,
2) *compatible*, and
3) *assignment-compatible*.

Sometimes it is necessary to meet all three conditions and other times it is necessary to meet only one.

4.5.1 Type Identity

```
types
   scale = number;
   subscale = scale;
   ...
var
   average, present : number;
   bottom, top : scale;
   start, finish : subscale;
   ...
```

Example 4.2 - Type Identity Example

If we take two identifiers - A and B, we can refer to them as being of *identical* type if :-

- A and B use the same type-identifier. In the above code segment, `average` and `present` use the same type-identifier.
- A and B are declared with equivalent types. In the above code segment, `start` and `top` have identical types because `subscale`, `scale` and `number` are all equivalent.

In the following circumstances, it is imperative to have identical types :-

- The parameters used in a call to a procedure must be the same type as those in the declaration of the procedure (see Chapter 8).
- The result returned by a procedure must be the same type as that declared in the procedure definition (see Chapter 8).

4.5.2 Type Compatibility

As mentioned previously, it is sometimes necessary for types to be *compatible*. This is often a requirement of assignment - i.e., that the type of the right-hand side of the assignment is compatible with the type of the identifier on the left-hand side.

Two types can be described as *compatible*, if:

- Both types are identical (see Section 4.5.1),
- Both types are of type `number`, and
- One type is a subrange and the other is of type `number`.

4.5.3 Assignment Compatibility

Assignment compatibility is necessary whenever an identifier is given a value either directly using an assignment statement, indirectly when passing parameters to procedures.

If we imagine an assigment statement `(id = exp)`, where id is of type `A` and exp is of type `B`, then we can say that exp is assignment-compatible when:

- `A` and `B` are of identical types,
- `A` is the type number, and `B` is a subrange-type, and
- `A` is a basic-type, and `exp` is the accessing of a structure-type with base-type identical to `A`.

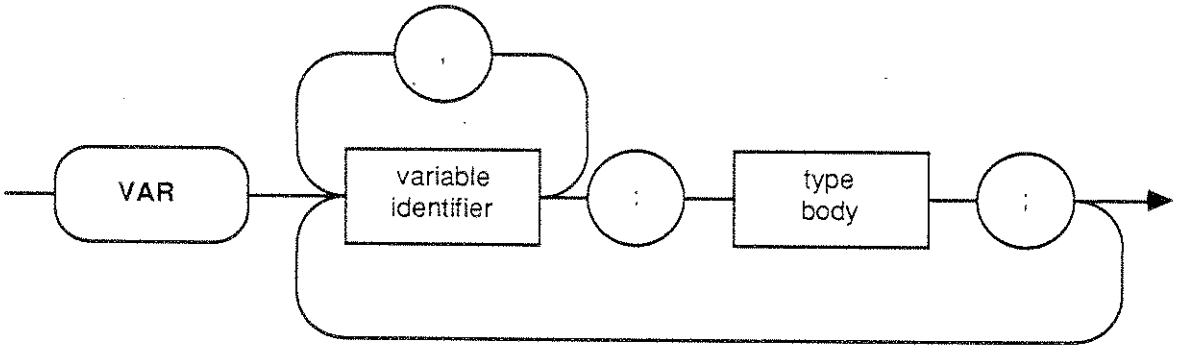
An error will be generated if assignment-compatibility is required, and none of the above rules are met.

Chapter 5
Variables

5.1 Variable Declarations

A *variable* is a name used to identify an expression, value or data-structure. A *variable-declaration* is a list of identifiers, followed by their types, which informs the compiler of what type can be bound to the identifiers.

Variable Declarations



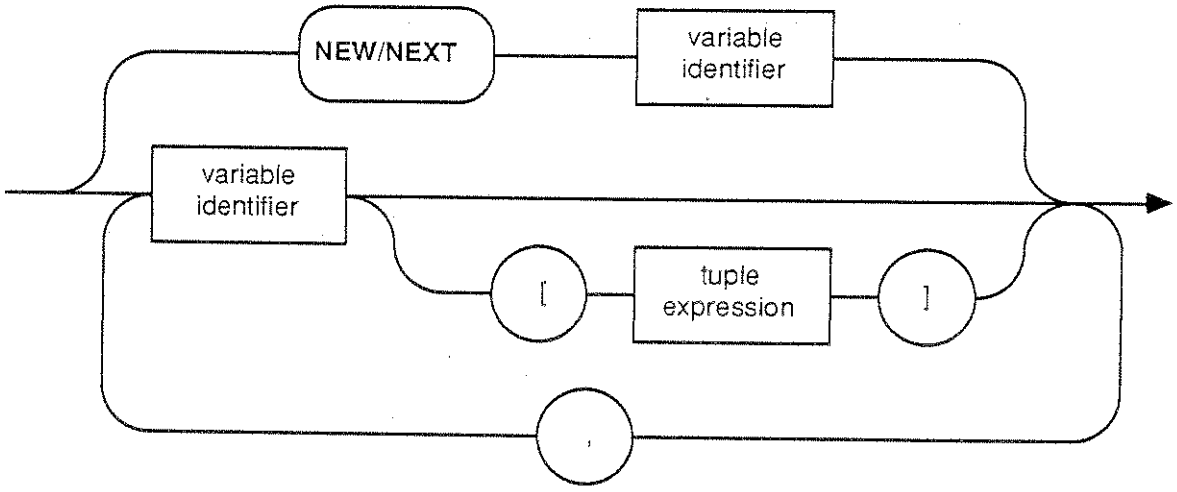
A *variable-identifier* can be used anywhere within the block in which its declaration appears. In IDA, there are no restrictions placed on the length of variable-names.

Example of variable-declarations :-

```
balance, deposit, repayment : number;
cancel, continue : boolean;
greeting : string;
matrix : array[1..5,1..5] of number;
tuple_x : tuple(number;number;boolean);
blf : union(boolean;number);
```

5.2 Variable References

Variable



The use of a variable signifies either a reference to :-

- the value bound to the identifier if it is of a simple-type,
- the component of the identifier if it is of a structured-type, or
- the data-structure if the identifier is of a structured-type .

5.3 Qualifiers

As the above segment from the IDA syntax charts show, a variable-reference consists of a variable-identifier followed by zero or more qualifiers. Each qualifier changes the meaning of the variable-reference.

In IDA, there is only need for one type of qualifier which is used with arrays. Using `matrix` from the above variable-declarations as an example, the use of the identifier by itself is a reference to the entire array-variable :-

```
matrix
```

While the use of the identifier followed by an array index is a reference to a particular component of the array-variable :-

```
matrix[row,column]
```

5.4 Accessing components of Tuples and Unions

Although these types of data-structures do not have qualifiers, their components can be extracted using a form of pattern-matching. Using the variable-declarations from above :-

```
class_size, balance, cancel = tuple_x
```

This is an example of accessing the components of a tuple. The order of the identifiers on the LHS must match so that their types are compatible with the types of the components within the tuple. Therefore, `class_size` is assigned the value of the number field within `tuple_x`, balance is assigned the value of the number field and `cancel` is assigned the value of the boolean field.

Unions use the same format to access their fields. Consider the following segment of code :-

```
types
   element = union(number;element);

var
   list0, list1, list2, list3 : element;
   val1, val2, val3 : number;

% code to assign values to variable "list0"
   ....

% accessing first 3 components of variable "list0"
   val1,list1 = list0;
   val2,list2 = list1;
   val3,list3 = list2;
```

Example 5.1 - Union Access

This example shows how unions can be used to define a recursive data-structure, and in this case a list. The last three statements are examples of how to access the components of a union. The order of the identifiers on the LHS must match so that their types are compatible with the types of the components within the union. The first statement, takes the value of the first number field and assigns it to `val1`, while the value of the `element` field is assigned to `list1`. This is in effect a form destructive head and tail operator. The second number component of the original `list0`, is obtained by extracting the value of the number field from `list1`, the tail of the original `list0`. Finally the third number component of the original `list0`, is extracted from `list2`, the tail of `list1`.

Chapter 6
Expressions

In IDA, expressions normally consist of *operators* and *operands*. An operator is usually an arithmetic function such as '+', and an operand is a value/argument used by the operator. As in most programming languages most of the operators are diadic - that is they require two operands though there are some monadic operators such as '-' and `not`, which require only one operand.

When an expression consists of two or more operators, certain rules are used to determine the order in which the operators execute. Taking the example below :-

```
3 * 3 + 1 / 4                 = 9.25
(3 * 3 + 1) / 4               = 2.5
(3 * (3 + 1)) / 4             = 4
3 * ((3 + 1) / 4)             = 3
3 * (3 + 1 / 4)               = 9.7
```

it can be seen the value that is represented by an expression can vary quite considerably depending on the order in which the operators are used and parentheses ( ) can be added to make the code more readable for the programmer and to override the default execution order of the operators[1].

The rules used to determine how an expression is evaluated are referred to as *precedence rules* :-

- An operand appearing between two operators of differing precedence is associated with the operator of higher precedence.
- An operand appearing between two operators of the same precedence is associated with the left-most operator.
- An parenthesised expression is always evaluated before is associated with an operator.

Operators	Precedence	Categories
not	highest	unary operators
*, /, mod, and	second	"multiplying" operators
+, - , or	third	"adding" operators & signs
=, <>, <, > <=, >=	lowest	relational operators

Table 6.1 - Operator Precedence

The precedence rules are represented by the syntax charts for expressions which are made up from simple-expressions, terms and factors (see Table 6.1).

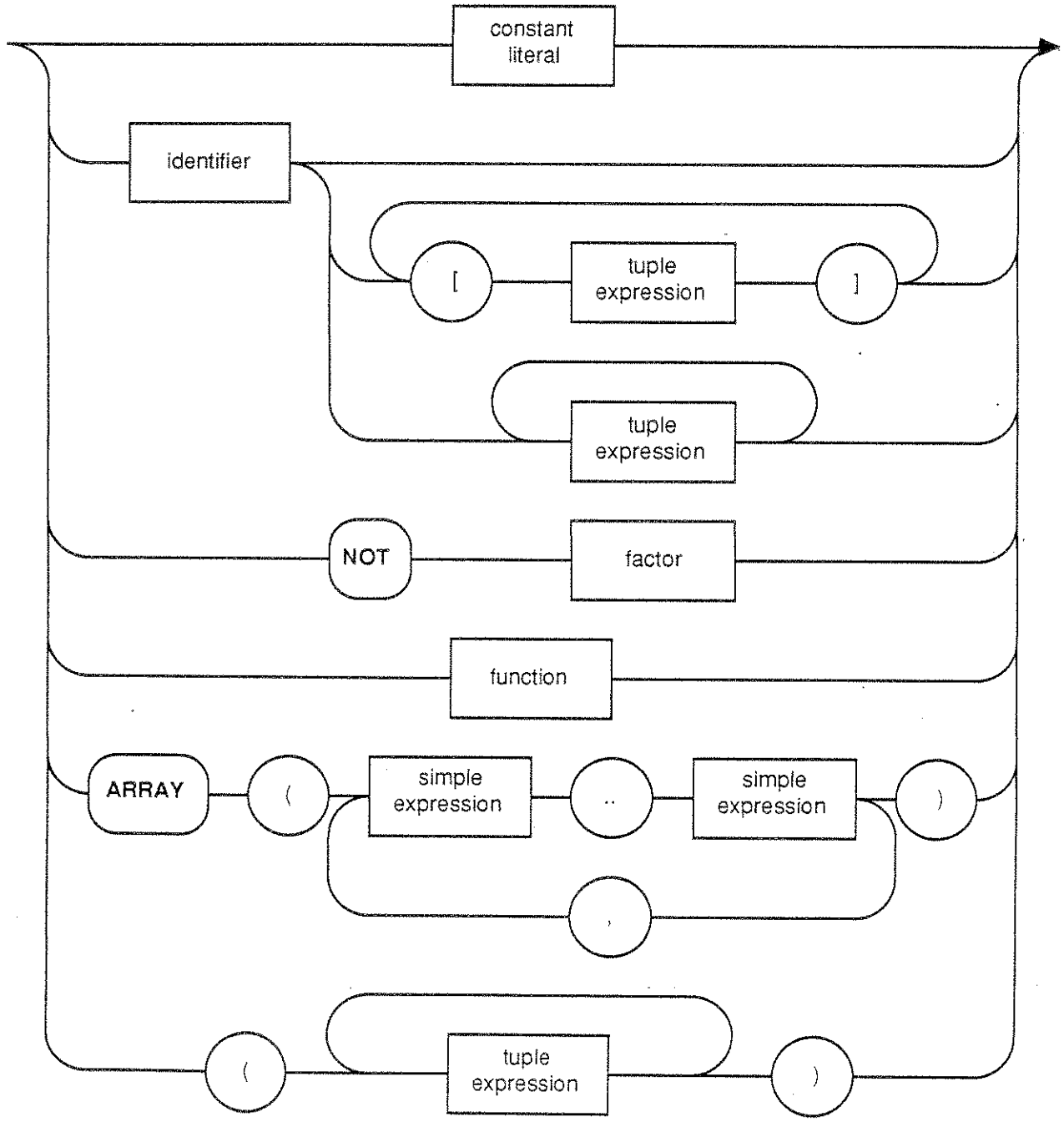
---

[1]When creating conditionsl expressions for use in IF and WHILE statements (Chapter 7), all subexpressions must be parenthesised :-

```
if ((count<=0) or (count>=9)) and (not quit)) then
    ....
```

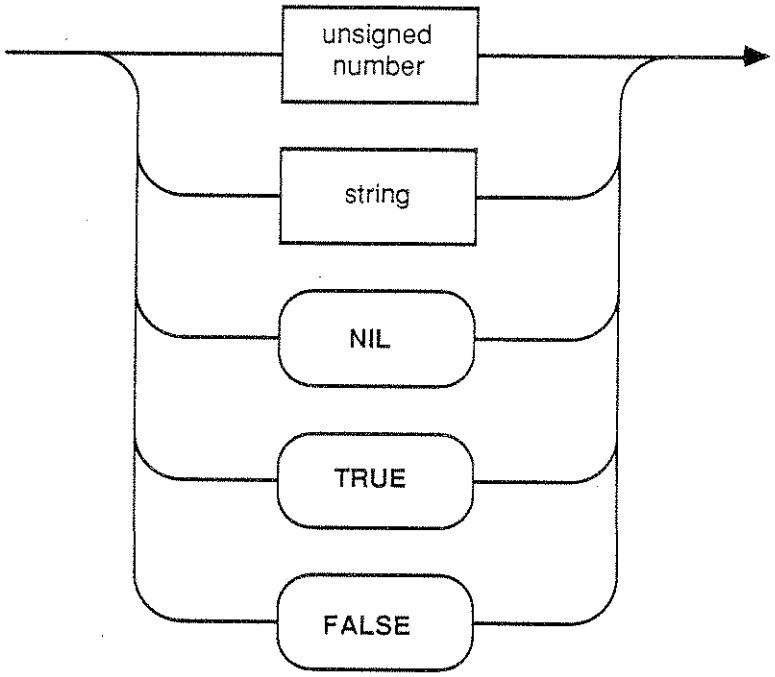
The syntax chart for a *factor* is shown below:

Factor



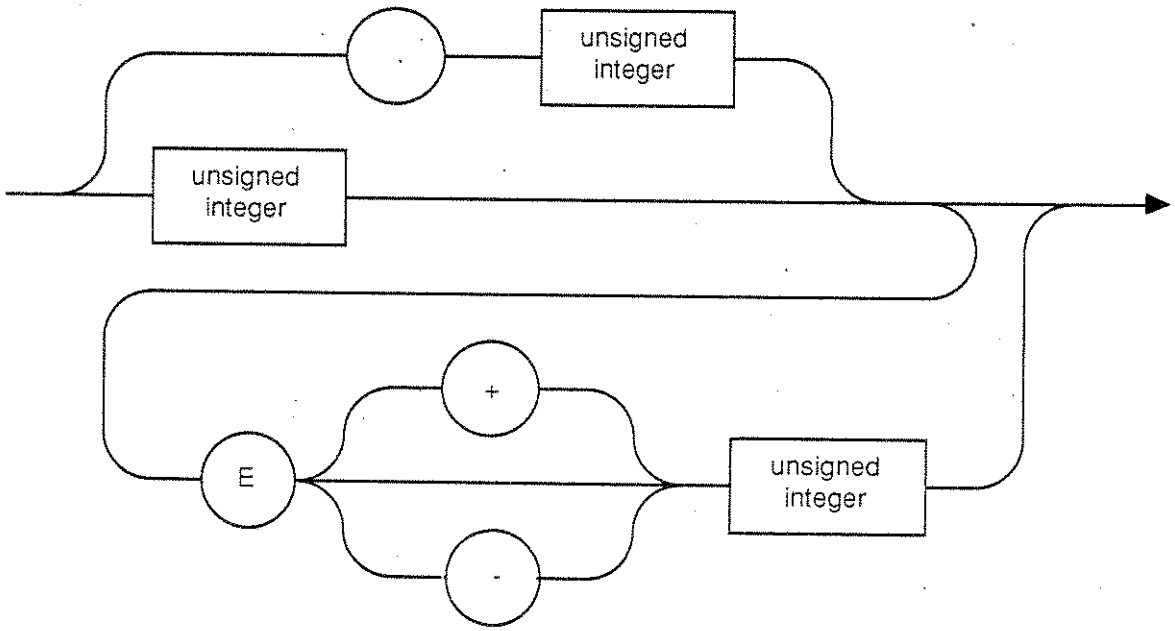
A *constant literal* can be any one of the following values :-

Constant Literal

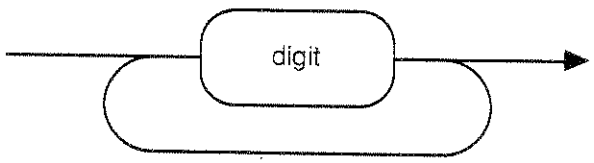


A *unsigned number* represents a numeric value with the following syntax :-

Unsigned Number

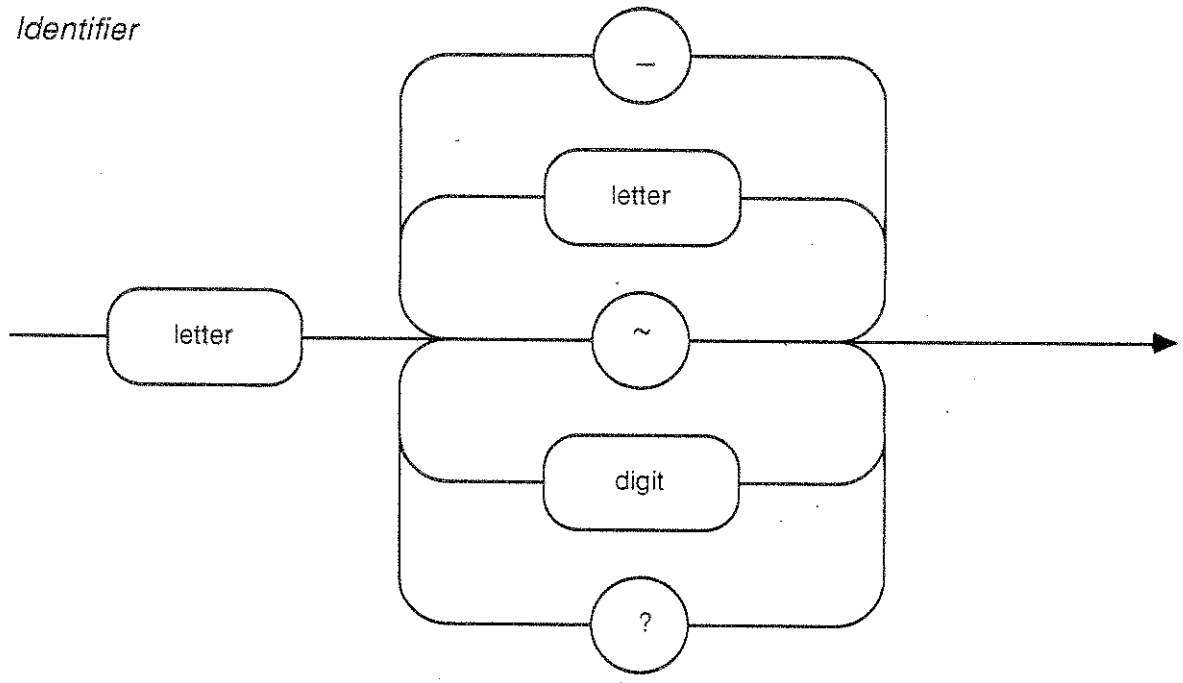


Unsigned Integer



The *identifier* branch of the factor syntax represents 3 IDA constructs - an identifier for a named value, an array and a procedure call where the parameters are represented by a tuple-expression. Identifiers are not case-sensitive or restrained by size. Here are a few examples :-

```
a
y2
equal_to_100?
have_a_beer?
av_class_size
```



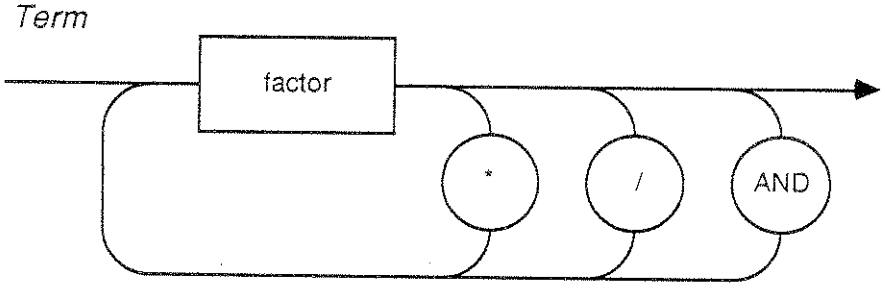
The *function* branch represents the major statements available in IDA - `if`, `while`, `for`, `let`, and `sequence` (see Chapter 7).

The *array* branch of the factor syntax represents an Id Nouveau construct for the dynamic allocation of arrays. This has not been implemented in this release.

examples of factors :-

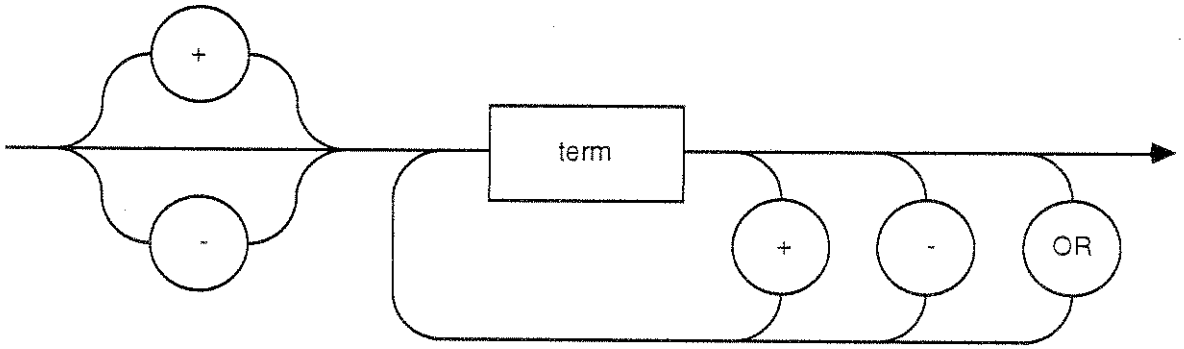
```
2.3                        { constant literal }
matrix[row,column]         { array reference }
copy lista,listb,lsize     { procedure call }
not cancel                 { negation of a boolean }
( 3 + 4, y2, cancel )      { bracketed tuple-expression }
```

The syntax for a *term* is shown below :-



The syntax for a *simple-expression* is shown below :-

Simple Expression



examples of terms :-

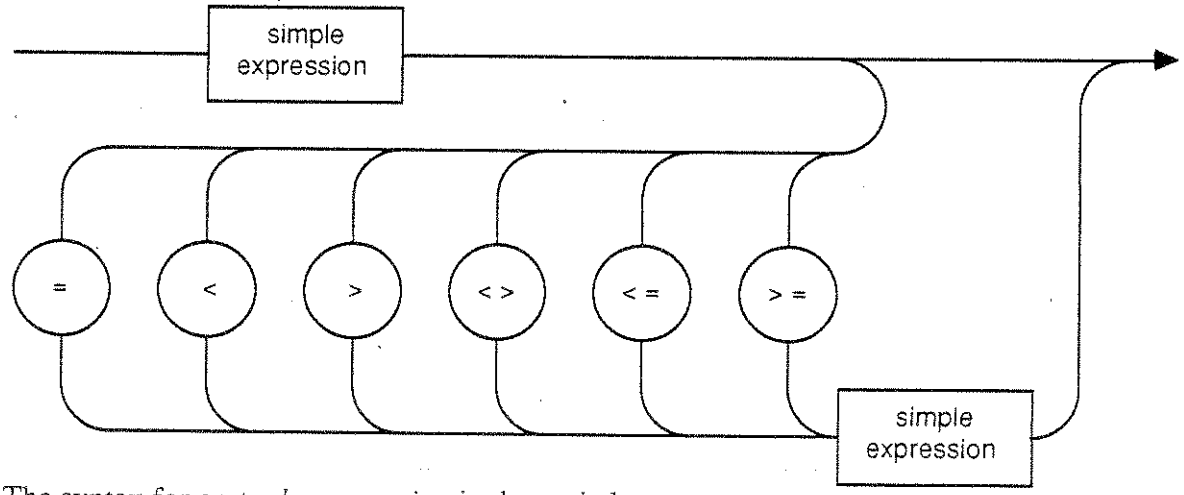
```
alpha * beta
42 / 7
cancel and error
```

examples of simple-expressions :-

```
- 6
nett - tax
result1 + result2
left or right
```

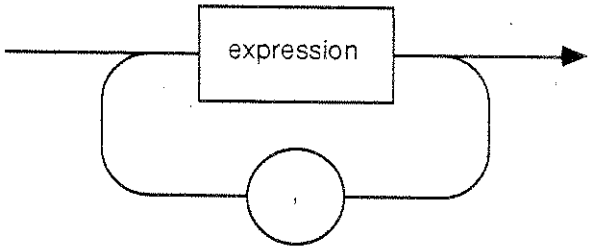
The syntax for an *expression* is shown below :-

Expression



The syntax for an *tuple-expression* is shown below :-

Tuple Expression



```
essions :-                    examples of tuple-expressions :-
:exp                             1, 2, 3
> stop_symbol                    fact 4, cancel, 9 * q + 1
top_score                        a < b, cramps?, num_cramps
l{ equality test & assignment operator }
```

ORS

on, the IDA operators are discussed and tables are presented which give the tion of operator, operand types, and type of result.

c operators

tors are used when writing mathematical expressions. As can be seen from the type of any operation is always the type number.

s	Operation	Operand 1 Type	Operand 2 Type	Result Type
	addition	number	number	number
	subtraction			
	multiplication			
	division			

Table 6.2 - Binary Arithmetic Operators

perators	Operation	Operand Types	Result Type
+	identity	number	number
-	sign negation		

Table 6.3 - Unary Arithmetic Operators

perators

rators take only operands which evaluate to be boolean and they return boolean unctions are those used in binary logic. They are normally used to control the ertain segments of code and are used as conditions within IF and WHILE

Operators	Operation	Operand Types	Result Type
or	disjunction	boolean	boolean
and	conjunction		
not	negation		

Figure 6.4 - Boolean Operators

6.1.3 Relational operators

Relational operators are also used to control the execution of IF and WHILE statements and are often used to produce operands for binary operators. They are used for comparing size of operands and testing for equality of operands.

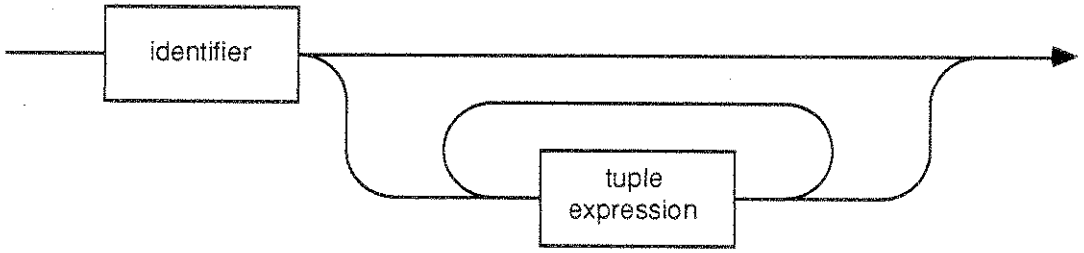
Operators	Operation	Operand Types	Result Type
=	equal	boolean number	boolean
<>	not equal		
<	less		
>	greater		
<=	less / equal		
>=	greater / equal		

Table 6.5 - Relational Operators

6.2 PROCEDURE CALLS

A procedure call is the IDA construct used to invoke the segment of code called a procedure. The declaration of the procedure outlines the interface into and out of the code segment and specifies the data required for its operation, the procedure of its body and the result the procedure returns. As the procedure always returns a value, its activation can be used as an expression. If the procedure-declaration includes a list of formal-parameters, then the procedure-call must have a matching list of actual-parameters.

Procedure Call



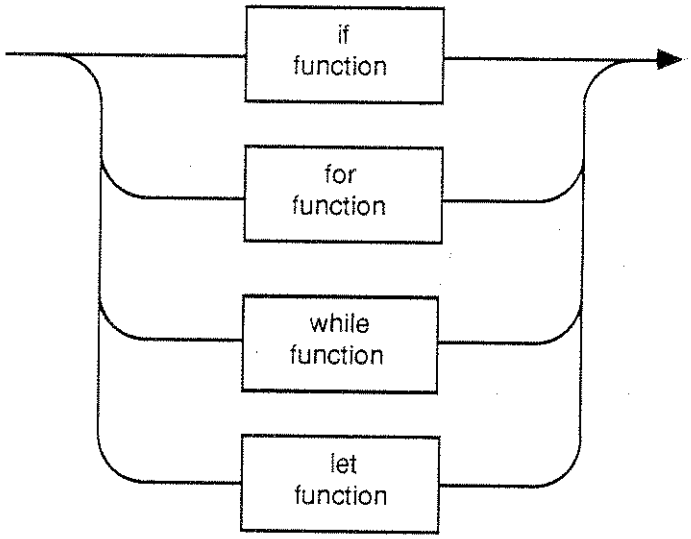
examples of procedure calls :-

```
matmult mat1 mat2 size
fact n
append n,list
```

Chapter 7
Functions

In IDA, functions must return values to comply with the functional nature of the language. The functions which make up the body of any IDA program, identify with the steps of the algorithm encoded by the program. As well as the *simple* functions already discussed (i.e. - expressions - see Chapter 6), there are five *structured* functions used to embody sections of code.

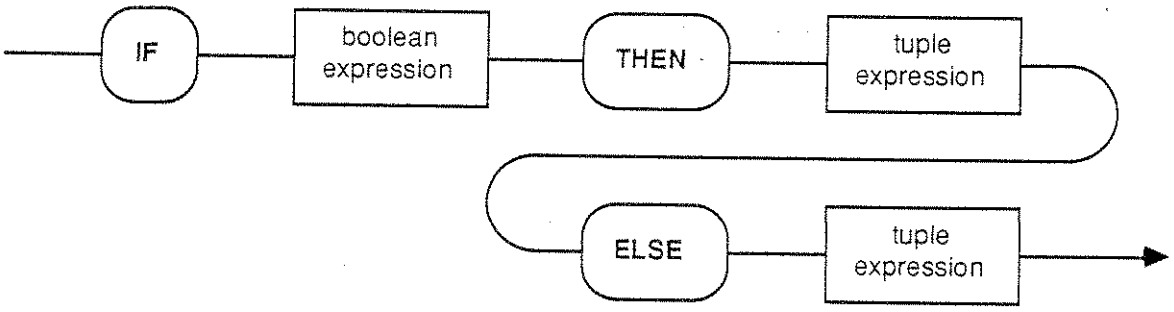
Function



Structured functions can consist of both simple and structured functions, that are to be executed conditionally (IF function), repeatedly (FOR and WHILE functions), as preparation for a complicated algorithmic step (LET function) or for input/output (SEQUENCE function).

7.1 The IF function

If Function



The IF statement is a conditional statement with two results. The result returned by the boolean expression is the *condition* which determines which of the two results is to be returned. If the condition is *true*, then the tuple-expression which follows the keyword `then` is evaluated. If the condition is *false*,then the tuple-expression which follows the keyword `else` is executed. To maintain the functional nature of the language both branches must be supplied so that the statement always returns a value.

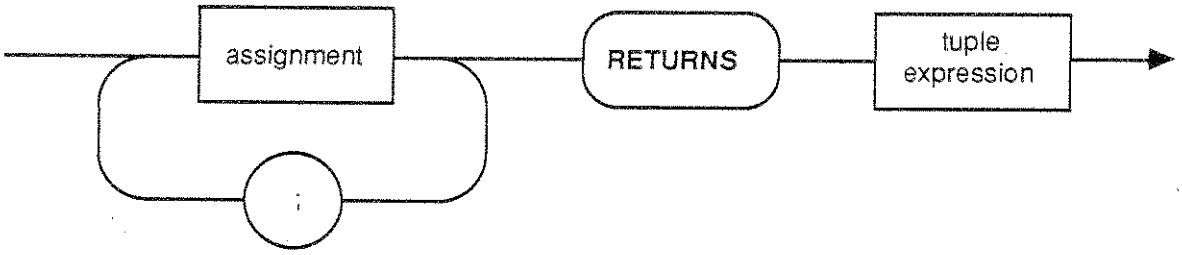
examples of IF functions :-

```
if val1<val2 then
   val1
else
   val2;

if do_cos then
   cos x
else
   sin x;
```

7.2 LOOPING FUNCTIONS

The looping functions in IDA are represented by two constructs reminiscent of many other programming languages - FOR and WHILE. They both consist of an expression which controls the repeated execution of the collection of functions making up the *loop body*.

Loop Body

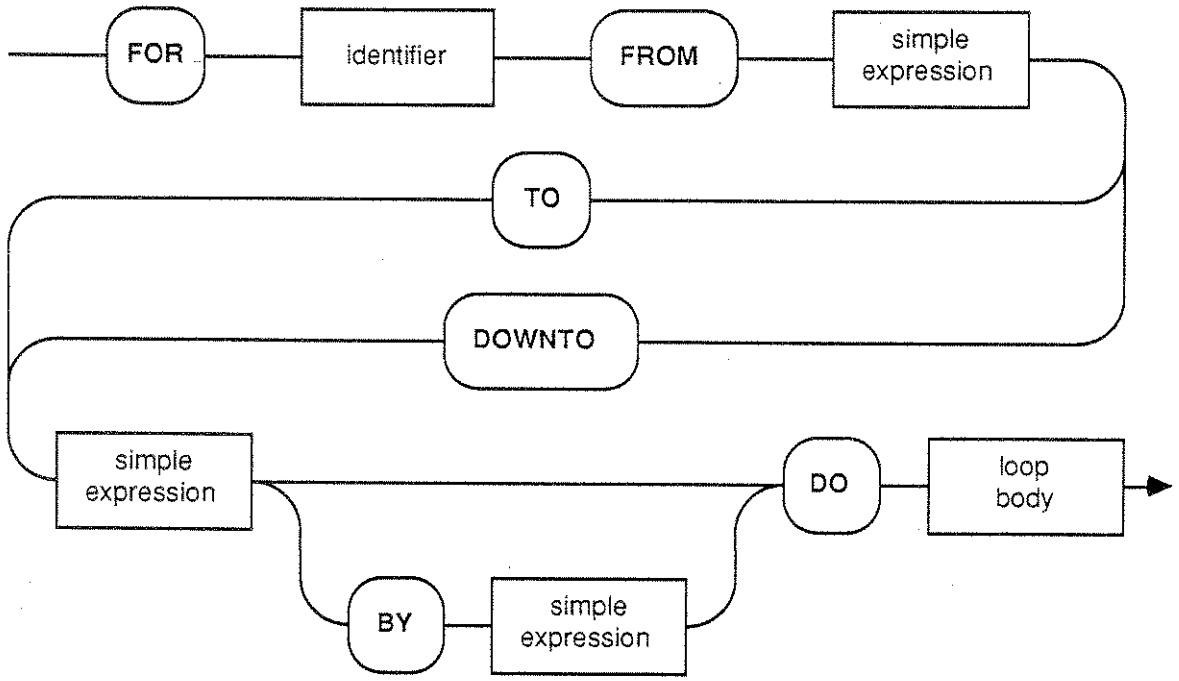


As can be seen from the syntax chart, the loop body contains assignments followed by a *return* tuple-expression. The return value maintains functionality and is the result of the loop execution. It is important that the user notes that identifiers used on the left-hand side (LHS) of assignments will be assigned a new value each time through the loop body. The reserved word `next` is introduced, to prefix a variable which is updated on each iteration of the loop and it creates data dependencies between these iterations (this will become clearer through examples).

7.2.1 The FOR function

The FOR function causes a single or a collection of functions to be executed a certain number of times. The number of repetitions is determined by a range of values which are expressed in the opening line. After the reserved word `for`, a variable identifier is supplied - this is referred to as the *control* variable and always contains a value within the defined range. The control variable is followed by the range which consists of an initial value (prefixed by `from`), a final value (prefixed by `to`/`downto`) and an optional third value (prefixed with `by`) which determines how large to make the increments through the range. As long as the value of the control identifier remains within the range, the loop body will be executed.

For Function



examples of FOR functions :-

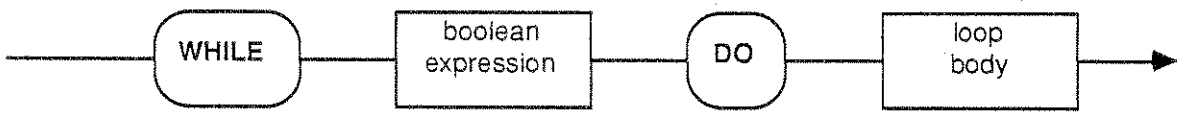
```
for i from lowbound to highbound do
   next total = total + i;
   part_total[i] = total
returns total;

for dir from 0 to 360 by 10 do
   axis       = dir mod 90;
   next answer  = if (axis = 0) then
                     axis
                  else
                     answer + 1;
returns answer;
```

From the above example it can be seen that the identifier `axis` is assigned a value through the first statement and as this value is not needed outside the current iteration it is not preceded by `next`. On the other hand, the identifier `answer` is sometimes incremented in the *else* branch of the IF statement and its value is used in the next and subsequent iterations of the loop body. Hence, it is preceded by `next`.

7.2.2 The WHILE function

The WHILE function depends on a conditional expression to determine the number of iterations of the loop body. As long as the value of the boolean expression is true, the loop body will be executed, and as soon as the expression returns false then the body is no longer executed and the result returned. Inside, the loop body therefore, is a function which will eventually result in the termination of the loop by causing the conditional expression to be false.

While Function



examples of WHILE functions :-

```
while epsilon > 0.001 do
   next sqrt = ((x / sqrt) + sqrt) / 2;
   next epsilon = abs (x - next sqrt * next sqrt)
returns sqrt;
```

In this example, as long as `epsilon` remains greater than a certain value the body will be executed. Note that in the second function the keyword `next` appears on both sides of the assignment operator. The importance of it on the LHS has already been discussed, but its use on the RHS is required to define which value of `sqrt` is to be used. In this case the value used is not that of the previous iteration but that calculated in the preceeding function.

```
while (not empty) and (month <= 12) do
   next empty = (rainfall[month] = 0)
   next month = month + 1
returns empty;
```

This example shows how control of the loop body is dependent on both statements to continue execution. If a month is found with no rainfall or the number of months extends outside of the present year then the loop is terminated.

7.3 The LET function

The LET function prepares a group of identifiers for use in a complicated expression, and is divided into two parts - the *Let-Assignments* where expressions are bound to identifiers, and the *Let-Return* where the result of this function is calculated. The Let statement is unusual in that it is the only construct where it is permissible not to return a value and this is done with a pair of empty parenthesis in the Let-Return.

examples of LET functions :-

This example simply shows a value being bound to an identifier for use in in expression.

```
let
   x = 2
in
   x + x;
```

This example demonstrates how a Let-statement is used to prepare identifiers for use in a more complicated expression (here it is a WHILE statement). The two variables in the Let-Assignment section are initialised, so that they can be used to control the WHILE statement.

```
let
   month = 1;
   empty = false
in
   while (not empty) and (month <= 12) do
   empty = if (rainfall[month] = 0) then
              true
           else
              false;
   next month = month + 1
return empty;
```

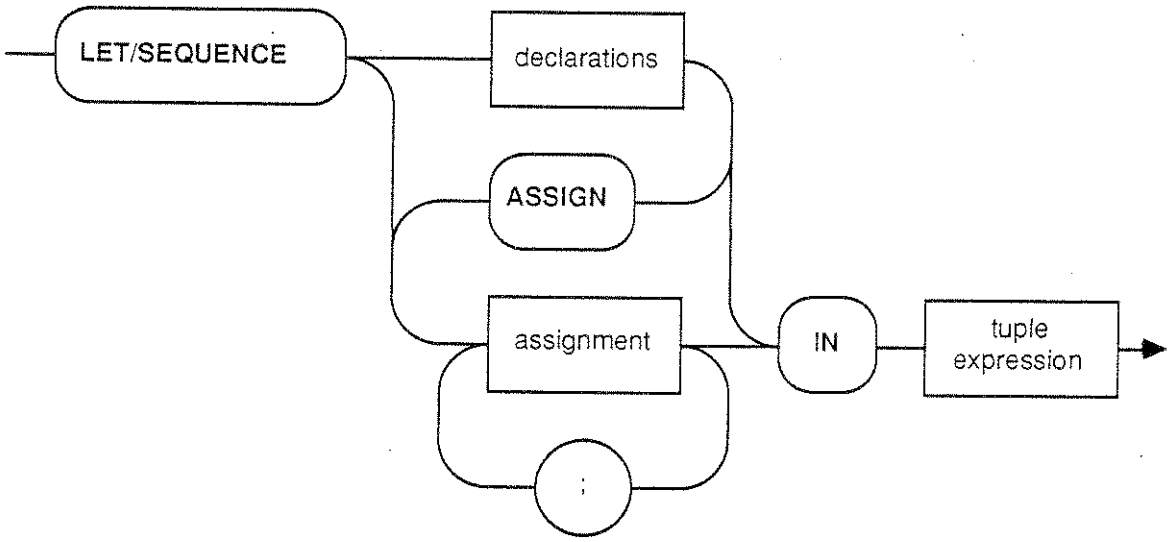
This examples shows the diversity of places the let-statement can be used. It can be used in the Let-Assignments section of a Let-statement to create one of the bindings for the Let-Return. It can also be used in the Let-Return section. The introduction of a nested Let-statement is used to create a new scope, so that a new instance of a variable can be used and assigned a different value (in this case x = 4 in one Let-statement and x = 7 in another).

```
let
   x2 = let
        x = 4
      in
        x * x;
   x3 = 3 * 3 * 3
in
   let
      a = x2 + x3;
      b = 10;
      x = 7
   in
      (a + b) / x;
```

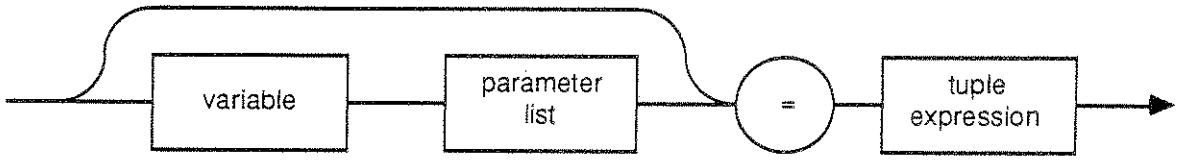
7.4 The SEQUENCE function

This function closely resembles the structure of the LET function but is used with input and output. (see Chapter 9).

Let/Sequence Function



Assignment



7.5 Commands

The above syntax for an assignment, shows that it is permissible not to provide an identifier on the left-hand side to bind to the resulting expression. This is referred to as a *command*, which are extremely useful when working with arrays.

```
def array_expr returns number =
let
   const
      n = 10;
   var
      x : array(1..n) of number;
      i, sum : number;

   def f i:number returns number = i * 8;

   assign
      = for i from 1 to n do
         x[i] = f i;
      returns ();
      sum = 0
in
   for i from 1 to n do
      next sum = sum + x[i]
   returns sum;
```

Example 7.1 - Command Example

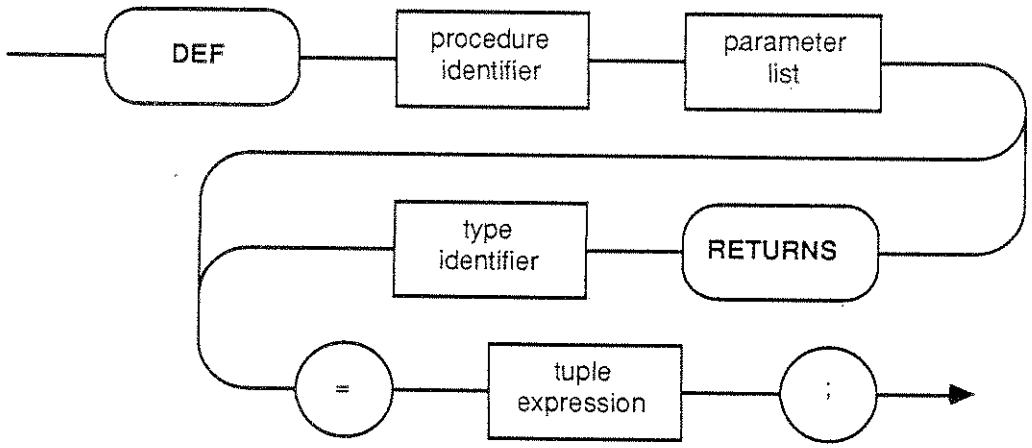
In this example above it can seen that a command is used in the Let-Assignment. It is making use of a `for` function and the procedure `f` to fill the elements of the array `x`. This is an example of a side-effect expression which alters an identifier without explicitly showing it. The other unusual construct is the object returned by the same `for` function - the *null* tuple - represented by `()`. These two features combined, form a powerful means to build an array.

Chapter 8
Procedures

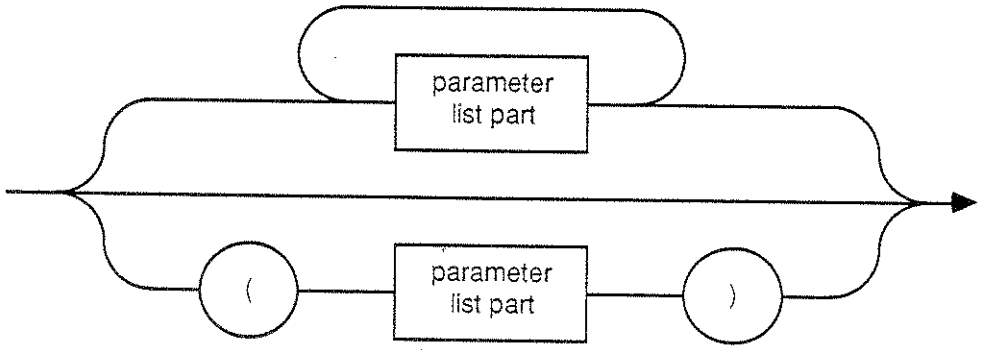
8.1 Procedure Declarations

A *procedure-declaration* creates the input and output interface needed to use the body of code associated with the procedure identifier (see Section 6.2).

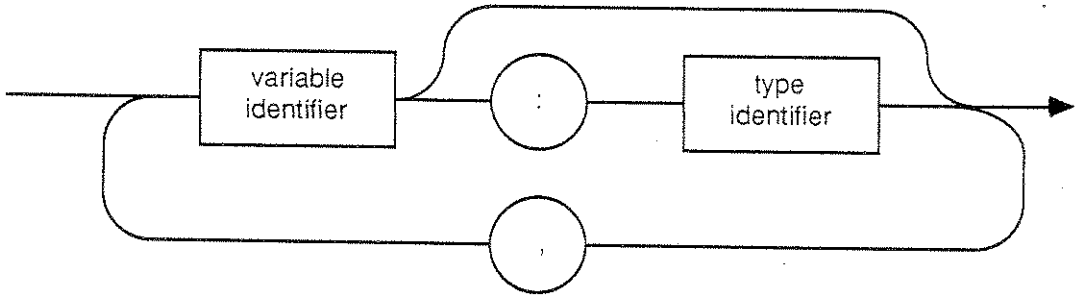
Procedure Declarations



Parameter List



Parameter List Part



The procedure identifier specifies the identifier for the procedure and the parameter list defines the input interface. The parameter list represents the values that need to be supplied by the programmer when the procedure is to be invoked. The type identifier after the keyword

`returns` determines the type of result that will be produced by the procedure, and also sets up the output interface. The tuple-expression represents the body of the procedure.

A procedure is activated by a procedure call (see Section 6.2), which supplies the procedure's identifier and the actual parameter list (if needed).The tuple-expression which constitutes the procedure body is evaluated using the parameters as operands to internal expressions as required. If the procedure identifier appears within the body of the procedure then it is said to be recursive (see Section 3.3) as it calls itself.

Example of a procedure-declaration :-

```
def fact n:number returns number =
if (n = 0) then
   1
else
   n * (fact (n - 1));
```

8.2 Parameters

The formal parameter list declared as part of a procedure declaration declares the parameters of the procedure. Each declared parameter is local to the procedure being declared and can be referenced by its identifier in the procedure body.

In IDA, each parameter supplied as an argument to a procedure-call must be an expression, and its type must be the same as that of the formal parameter. The current value of the expression is assigned to the formal parameter of the function.

8.3 Parameter List Compatibility

Parameter list compatibility is required between the parameter lists of the procedure declaration (formal parameters) and those supplied in the procedure call (actual parameters). Two parameter lists are said to be compatible if they contain the same number of parameters and the types of parameters in the corresponding positions match. Two parameters match if they are of *identical* type.

Chapter 9
Input / Output

This chapter describes the standard ("built-in") procedures which are included in IDA, from interfacing with the user of an IDA program. This allows the programmer to access standard input and output as well as dynamic files and is achieved through the sequential execution SEQUENCE statement (essential for effective Input/Output).

In IDA, Input/Output (I/O) is achieved through the use of *files*. Although most other languages have their own special file-type, IDA uses one of existing types to represent files - *streams*, and to declare a file-identifier, the programmer declares its type as being *stream of char*.

A file variable may be bound to an external file on the host. This file will most likely be a named collection of data stored on a peripheral device or, depending on the device it may be the actual peripheral device. A file variable not associated with an external file or device, is described as being *anonymous*. A file variable associated with an external device will normally be associated with one of the three predefined file identifiers which are opened automatically as the execution of the program begins:

- `stdin` (the input device - a read-only file normally associated with the keyboard),
- `stdout` (the output device - a write-only file normally associated with the terminal screen), and
- `stderr` (the error report file/device - a write-only file normally associated with the terminal screen).

For a file variable to be used it must first be opened. An existing file must be opened via the `open` procedure, and a new file must be created and opened using the `create` procedure. Both procedures open the file and move the file pointer to the start of the file.

To access the contents of the files, the `read` and `readln` procedures are used, while to update a file or add information to the file the `write` and `writeln` procedures are used. There are two final procedures used during I/O - `exists`, for checking if a file already exists and `close`, for closing a file.

9.1 SEQUENTIAL EXECUTION

To provide a sensible mechanism for I/O, it is necessary to permit sequential operations, however, providing sequentiality by explicit data dependencies at the source level is not a satisfactory solution, particularly in a single assignment language. IDA provides a special construct that causes top-level sequentiality. The reduction in concurrency can be reduced by only enforcing the sequencing for each line or function. Any operations inside these will run in parallel as will the evaluation of the arguments of each sequenced line. For example :-

```
= writeln stdout a * b + c * d, a - b;
= writeln stdout "hello world!";
```

will correctly output the result of the arithmetic expression first, followed by the string "hello world!". Of note is the fact that the expression will still be evaluated in parallel, i.e., both the multiplications and the subtraction simultaneously, followed by the addition. Whilst this may seem to be of marginal improvement over totally sequential machines, it is worth noting that those expressions can be arbitrarily complex. Using this approach, it is quite an easy matter for the compiler to generate the required sequencing instructions rather than the programmer explicitly (and tediously) doing so. The actual syntax duplicates the LET statement but replaces LET with the keyword SEQUENCE. So the complete program to perform the previous example will look something like this:

```
sequence
   const
      a = 1;
      b = 2;
      c = 3;
      d = 4;
   assign
   = writeln stdout a * b + c * d, a - b;
   = writeln stdout "hello world!"
in
   ();
```

Naturally, the use of the SEQUENCE statement can occur at any level of scoping and its correct operation is up to the programmer. Also, any operation can be sequenced in this manner, not just I/O.

9.2 STANDARD I/O PROCEDURES

This following terms are used throughout this section:

<file>	- stream of char (including stdin, stdout and stderr). Only ASCII files are supported.
<filename>	- file identifier (must be a literal string).
<error>	- an integer returned by the procedures to signify success or failure. A value of 0 is success, while any other value signifies failure.
<arguments>	- blank separated list of variables.
[.....]	- contents of brackets may be optionally supplied by programmer.

9.2.1 The OPEN Procedure

```
<error> = open <file> [<filename>]
```

The OPEN procedure opens an existing file to be accessed via <file> and using the physical file <filename>. <file> is a unique file identifier generated by OPEN. It is an error to open a non-existent file. A temporary file is opened if no <filename> is supplied.

9.2.2 The CREATE Procedure

```
<error> = create <file> [<filename>]
```

The CREATE procedure creates a new file, where <file> and <filename> are the same as for OPEN. It is an error to create an existing file. A temporary file is opened if no <filename> is supplied.

9.2.3 The CLOSE Procedure

```
<error> = close <file>
```

The CLOSE procedure closes <file> which has been previously opened or created. It is an error to attempt to close <file> when it has not previously been opened or created.

9.2.4 The EXISTS Procedure

```
<boolean> = exists [<filename>]
```

The EXISTS procedure returns true if the specified <filename> exists; false if the file specified does not exist. This permits the error free usage of OPEN and CREATE.

9.2.5 The READ Procedure

```
<error> = read <file> <arguments>
```

The READ procedure reads the specified <arguments> from the specified <file>. It is an error to read from a file <file> that has not been opened or created.

9.2.6 The READLN Procedure

```
<error> = readln <file> [<arguments>]
```

As for READ with the exception that after all <arguments> have been read, all remaining values on the <file> are skipped until a newline or end of file

9.2.7 The WRITE Procedure

```
<error> = write <file> <arguments>
```

The WRITE procedure writes the specified <arguments> to the output <file> after evaluation of the <arguments>. Output field widths can be specified as in Pascal with the ':' operator. After evaluation of the <arguments>, the values being output must be scalar. It is error to write to a <file> that has not been opened or created.

9.2.8 The WRITELN Procedure

```
<error> = writeln <file> [<arguments>]
```

As for WRITE with the exception that a newline is written directly after the <arguments> have been written to the output <file>.

A simple I/O example :-

```
(*
This simple program opens a file and writes out the result of a
very simple expression together with a string. It is not
possible to read strings as they are non-scalar, but characters
may be read. The result of the I/O operations are not tested for
since the EXISTS function checks for files being overwritten.
*)

const
   a = 10;
   b = -10;

def writeout returns () =   (* file I/O sequence *)
   sequence
      var
         c : number;
         infile, outfile : stream of char;
      assign
         = open outfile "filename";
         = writeln outfile a+b, "hello world!";
         = close outfile;
         = open infile "filename";
         = readln infile c;
         = writeln stdout "c = " c;
         = close infile
   in
      ();

def error =                 (* error routine *)
   let
      = writeln stderr "filename already exists ..."
   in
      ();

if exists "filename" then   (* main program *)
   writeout
else
   error;
```

Example 9.1 - Input/Output Example

Chapter 10
Standard Procedures

This section describes all the standard ("built-in") procedures in the IDA programming language except for the I/O procedures already discussed in Chapter 9.

10.1 ARITHMETIC PROCEDURES

As well as the simple operators given in the syntax chart of expression, IDA provides others for some of the more complicated mathematical functions.

10.1.1 The INR operator

Returns the inner-products of two arrays.

10.1.2 The EXP operator

Returns the exponential of a numeric value.

result type: `number`
parameter list: **`exp`** `arg`

This call returns the value of e^{arg}, where e is the base of the natural logarithms.

10.1.3.The PWR operator

Returns the value of a numeric value being raised to the power of a second numeric value.

result type: number
parameter list: **`pwr`** `arg1 arg2`

This call returns the value of $arg1^{arg2}$.

10.1.4 The ABS operator

Returns the absolute value of a numeric value.

result type: `number`
parameter list: **`abs`** `arg`
This call returns the absolute value of `arg` - i.e., if `arg` is negative then `-arg` is returned, otherwise `arg` is returned.

10.1.5 The LNE operator

Returns the natural logarithm to base e of a numeric value.

result type: `number`
parameter list: **`lne`** `arg`

This call returns the natural logarithm of arg - i.e., $\log_e$ `arg`.

10.1.6 The LN2 operator

Returns the natural logarithm to the base 2 of a numeric value.

result type: `number`
parameter list: **`ln2`** `arg`

This call returns the natural logarithm of arg - i.e., $\log_2$ `arg`.

10.1.7 The LOG operator

Returns the natural logarithm to the base 10 of a numeric value.

result type: `number`
parameter list: **`log`** `arg`

This call returns the natural logarithm of arg - i.e., $\log_{10}$ `arg`.

10.1.8 The SQT operator

Returns the square-root of a numeric value.

result type: `number`
parameter list: **`sqt`** `arg`

This call returns the square-root of arg - i.e., $\sqrt{\texttt{arg}}$.

10.1.9 The SQR operator

Returns the square of a numeric value.

result type: `number`
parameter list: **`sqr`** `arg`

This call returns the square of arg - i.e., `arg * arg`, or $\texttt{arg}^2$.

10.1.10 The SIN operator

Returns the sine of a numeric value.

result type: `number`
parameter list: **`sin`** `arg`

This call returns the trigonometric sine of `arg`.

10.1.11 The COS operator

Returns the cosine of a numeric value.

result type: `number`
parameter list: **`cos`** `arg`

This call returns the trigonometric cosine of `arg`.

10.1.12 The TAN operator

Returns the tangent of a numeric value.

result type: `number`
parameter list: **`tan`** `arg`

This call returns the trigonometric tangent of `arg`.

10.1.13 The ATN operator

Returns the arc_tangent of a numeric value representing a radian.

result type: `number`
parameter list: **`atn`** `arg`

This call returns the trigonometric arc_tangent of `arg`.

10.1.14 The ASN operator

Returns the arc_sine of a numeric value representing a radian.

result type: `numeric`
parameter list: **`asn`** `arg`

This call returns the trigonometric arc_sine of `arg`.

10.1.15 The ACS operator

Returns the arc_cosine of a numeric value representing a radian

result type: `number`
parameter list: **`acs`** `arg`

This call returns the trigonometric arc_cosine of `arg`.

10.1.16 The RND operator

Converts a real constant literal into an integer constant literal.

result type: `integer`
parameter list: **`rnd`** `arg`

This call returns an integer result which is the value of `arg` rounded to the nearest whole number. If `arg` is exactly halfway between two whole numbers or larger than the halfway value, the result is the larger absolute number. If `arg` is smaller than the halfway value, the result it the smaller absolute number.

10.1.17 The TRC operator

Converts a real constant literal into an integer constant literal.

result type: `integer`
parameter list: **`trc`** `arg`

This call returns an integer result which is the value of `arg` rounded to the nearest whole number between 0 and `arg` inclusive.

10.1.18 The FLT operator

Returns a real constant literal which represents the result of the left-hand side of arg subtracted from the whole arg.

result type: `number`
parameter list: **`flt`** `arg`

example: `x = flt 5.657; where x now equals 0.657`

10.1.19 The SUCC operator

Returns the successor of an integer constant literal.

result type: `integer`
parameter list: **`succ`** `arg`

This call returns the successor of `arg`.

10.1.20 The PRED operator

Returns the predecessor of an integer constant literal.

result type: `integer`
parameter list: **`pred`** `arg`

This call returns the predecessor of `arg`.

10.2 SORTING OPERATORS

10.2.1 The GTS operator

Tests two literal values and swaps them if the first value is smaller than the second.

result type: `number` or `char`
parameter list: **`gts`** `arg1 arg2`

example:
```
x,y = gts 5 5.657;  where x now equals 5.657,
                          y now equals 5
```

10.2.2 The LTS operator

Tests two literal values and swaps them if the first value is greater than the second.

result type: `number` or `char`
parameter list: **`lts`** `arg1 arg2`

example:
```
x,y = lts m b; where x now equals b,
                     y now equals m.
(* iff m is not less than b *).
```

10.2.3 The MAX operator

Returns the greater of two literal values.

result type: `number` or `char`
parameter list: **`max`** `arg1 arg2`

example:
```
big = max 10 5;where big equals 10.
```

10.2.4 The MIN operator

Returns the smaller of two literal values.

result type: `number` or `char`
parameter list: **`min`** `arg1 arg2`

example:
```
tiny = min 10 5;     where tiny equals 5.
```

10.3 GENERAL PROCEDURES

10.3.1 The RNG operator

Determines if a literal value is in a range of values.

result type: number or char
parameter list: **rng** arg1 arg2 arg3

example: rng 5 10 20 result = false.

rng arg1 arg2 arg3 - returns *true* if arg1 is between arg2 and arg3, and *false* otherwise.

10.3.2 The WDW operator

Sets the upper limit of a range of values.

result type: number or char
parameter list: wdw arg1 arg2 arg3

example: wdw 5 10 20 result = 5.

wdw arg1 arg2 arg3 - returns
arg2 - if arg1 < arg2,
arg3 - if arg1 > arg3,
arg1 - otherwise.

10.3.3 The ORD operator

Returns the ordinal number of a character

result type: integer
parameter list: **ord** arg

example: ord 'a' result = 97 (Ascii value).

ord arg - returns the ordinality of arg, i.e., the Ascii value of arg.

10.3.4 The CHR operator

Returns the char value to an integer constant literal.

result type: char
parameter list: **chr** arg

example: chr 32 result = ' ' (blank/space).

chr arg - returns the char value whose ordinal number is arg.

For any char value arg, the following is always true:
chr (ord arg) = arg.

ACKNOWLEDGEMENTS

The author wishes to especially thank Mr. Neil Webb, who wrote the syntax-checker in the compiler and was responsible for the I/O extensions to IDA, Mr. Stephen Brobst (M.I.T.) who gave invaluable advice on the implementation of I-structures and the general semantics of Id Nouveau. I would also like to thank the other members of the Software Stream of the project - Mr. Mark Rawling, Mr. Simon Wail, our external consultant Dr. K. Ramamohanarao, and the Stream Leader - Mr. Bob Pascoe, for their assistance and advice.Thanks must also go to the Hardware stream, Mr. Alan Young and Mr. Ian Donaldson, and in particular, the Project Leaders - Dr. David Abramson and Dr. Greg Egan. A number of IDA examples have been converted from Id Nouveau source code obtained from M.I.T.'s Laboratory for Computer Science. The Parallel Systems Architecture Project at RMIT is being supported by the Commonwealth Scientific and Industrial Organisation (CSIRO) under an Information Technology joint research grant.

REFERENCES

1. D. Abramson & G.K. Egan, "The RMIT Data Flow Computer : A Hybrid Architecture", Royal Melbourne Institute of Technology Technical Report, TR-112-057R, 1987.
To be published in The Computer Journal.

2. D. Abramson and G.K. Egan, "An Overview of the RMIT/CSIRO Parallel Systems Architecture Project", Royal Melbourne Institute of Technology Technical Report, TR-112-065R, 1987.
Proc 11th Australian Computer Sciences Conference, Brisbane, 1988
Republished in Australian Computer Journal, August 1988.

3. G.K. Egan, "Dataflow : Its Application to Decentralised Control", Ph. D. Thesis, Department of Computer Science, University of Manchester, 1979

4. R. Nikhil, K. Pingali and Arvind, "Id Nouveau", Computation Structures Group Memo 265, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts

5. Kazanori Ueda, "Guarded Horn Clauses", Doctor of Engineering Thesis, University of Tokyo, Graduate School, 1986

6. McGraw, et al, "SISAL : Streams and Iteration in a Single Assignment Language, Language Reference Manual", Lawrence Livermore National Laboratories, M146

7. M. Rawling and C.P. Richardson, "The RMIT Data Flow Computer : DL1 User's Manual", Royal Melbourne Institute of Technology Technical Report, TR-112-058R, 1987

8. Arvind, D.E. Culler, R.A. Iannucci, V. Kathail, K. Pingali and R.E. Thomas, "The Tagged Token Dataflow Architecture", Laboratory for Computer Science, MIT, July 1983

9. K.S. Weng, "Stream Oriented Computation in Recursive Data Flow Schemes", Technical Memo 68, Laboratory for Computer Science, MIT, Oct 1975

A) J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Dataflow Processor", Proc 2nd Annual Symposium Computer Architecture, New York, May 1975

B) J.B.Dennis, G.A. Broughton, and C.K.C Leung, "Building Blocks for Dataflow Prototypes", Proc 7th Annual Symposium Computer Architecture, La Boil, France, May 1980

C) J.B. Dennis, G.R. Gao, and K.W. Todd, "Modelling the Weather with a Dataflow SuperComputer", IEEE Trans. Computers, Vol -33, No 78, July 1984, pp 592-603

D) Arvind and R.A. Iannucci, "A Critique of Multiprocessing von Neumman Style", Proc 10th Annual Int'l Symposium Computer Architecture, Stockholm, June 1983, pp 426-436

E) Arvind and K.P. Gostelow, "The U-Interpreter", Computer, Vol 15, No. 2, Feb 1982, pp 42-50

F) J. Gurd and I. Watson, "Data Driven Systems for High Speed Parallel Computing - part 2: Hardware Design", Computer Design, July 1980, pp97-106

G) Guy Lewis Steele, Jr. and Gerald Jay Sussman, "SCHEME : An Interpreter for Extended Lambda Calculus", Tech. Report 349, MIT Artificial Intelligence Lab, Dec 1975

H) Guy Lewis Steele, Jr. and Gerald Jay Sussman, "The Revised Report on SCHEME, a Dialect of LISP", MIT Artificial Intelligence Lab Memo 452, January 1978

I) S. Skedzieleski and J. Glauert, "IF1 - An Intermediate Form for Applicative Languages", Lawrence Livermore National Laboratories, July 1985

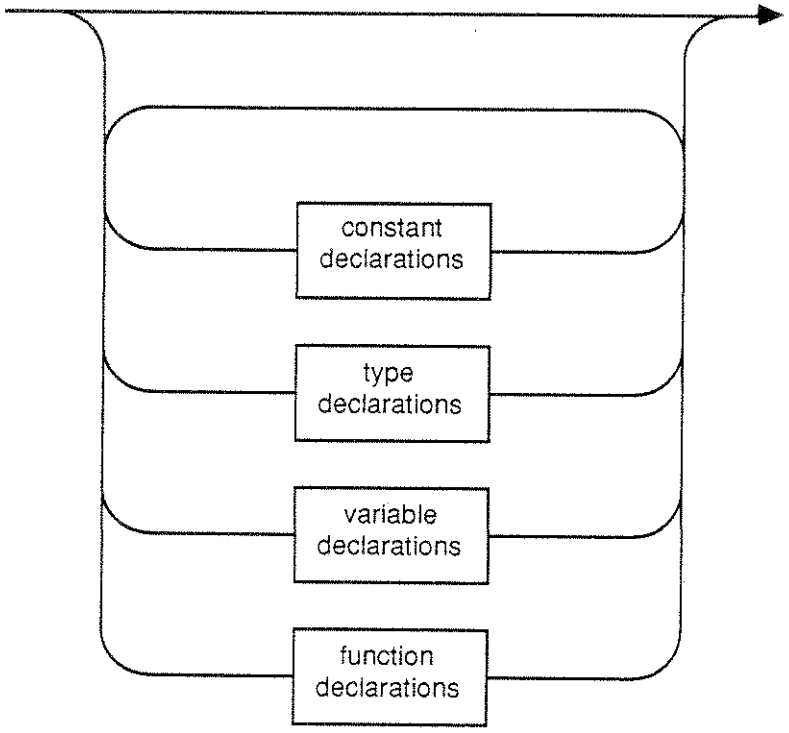
J) Arvind, R.S. Nikhil and K.K. Pingali, "I-Structures : Data Structures for Parallel Computing", Computation Structures Group Memo 269, MIT, Laboratory for Computer Science, Feb 1987

K).A.V.Aho, R. Sethi and J.D.Ullman, "Computers - Principles, Techniques and Tools", Addison Wesley, 1986

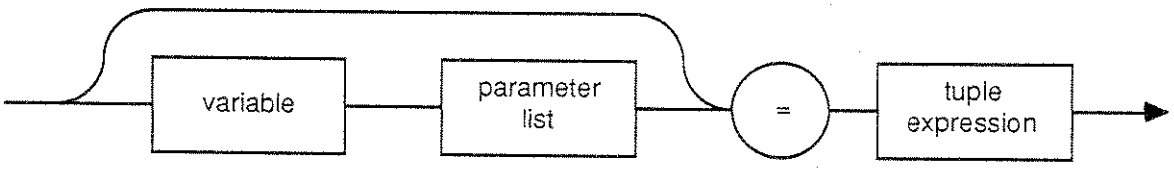
Appendix A - IDA Syntax

Block

Declarations

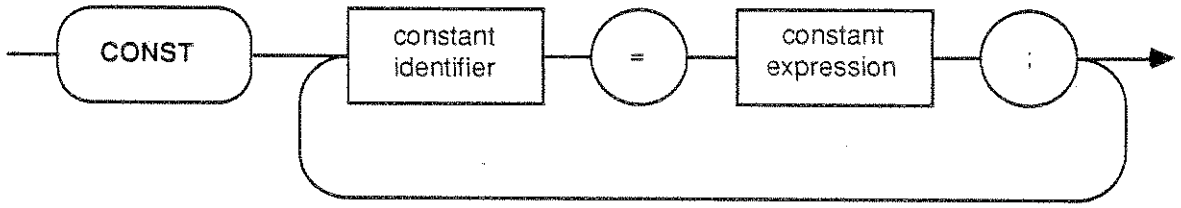


Assignment

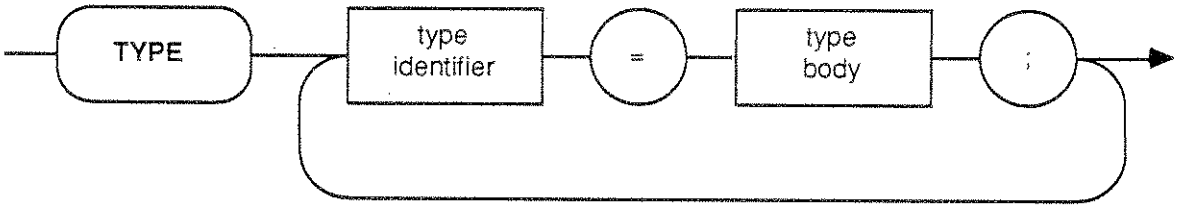


Appendix A - IDA Syntax continued

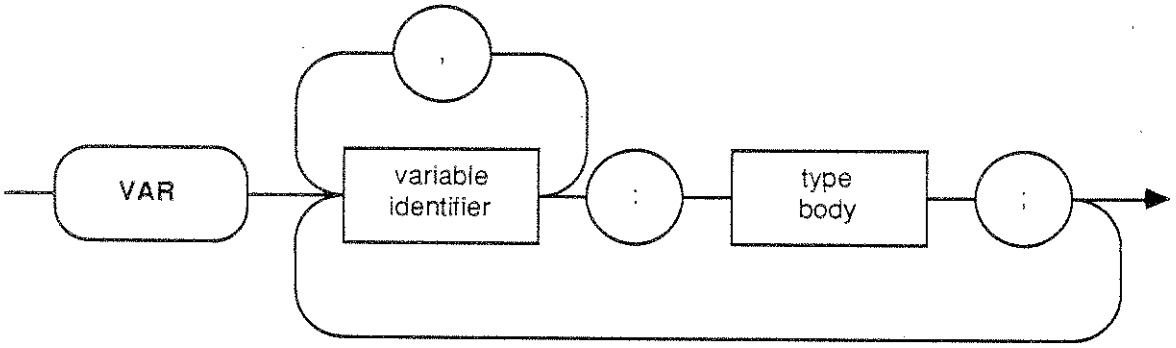
Constant Declarations



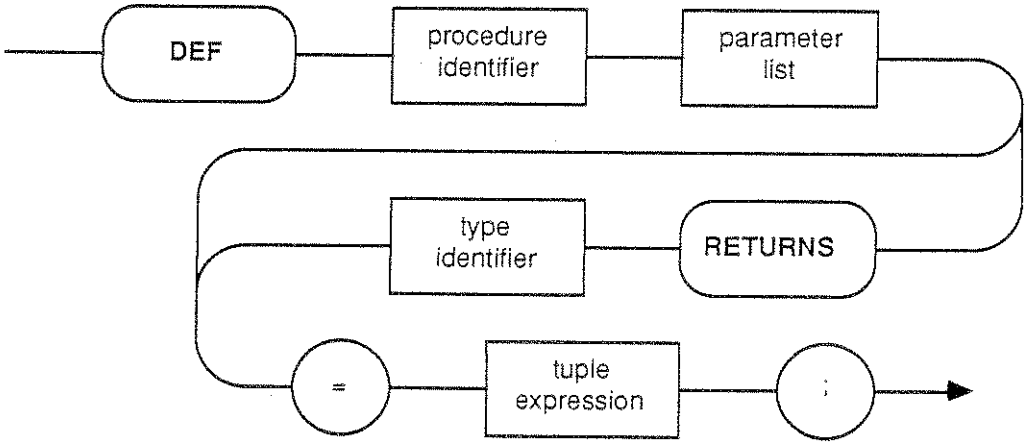
Type Declarations



Variable Declarations

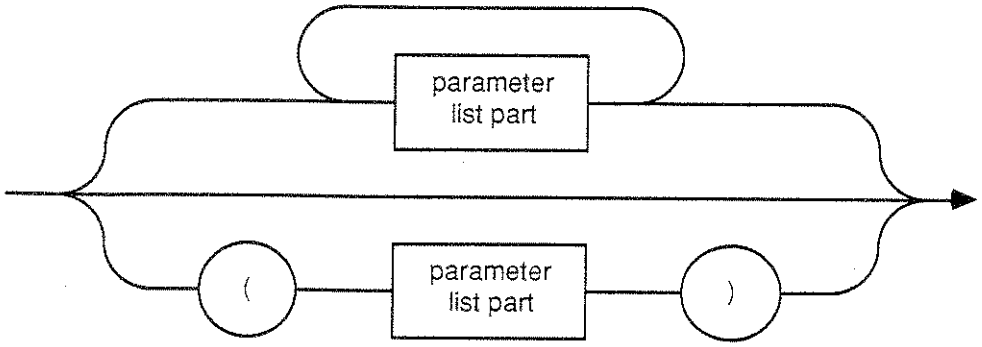


Procedure Declarations

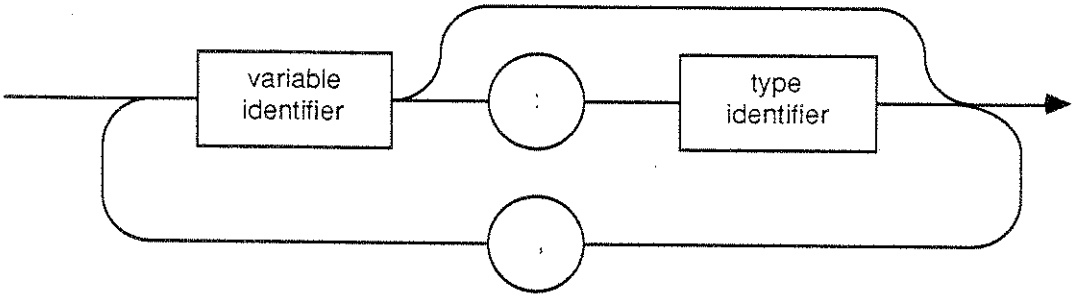


Appendix A - IDA Syntax continued

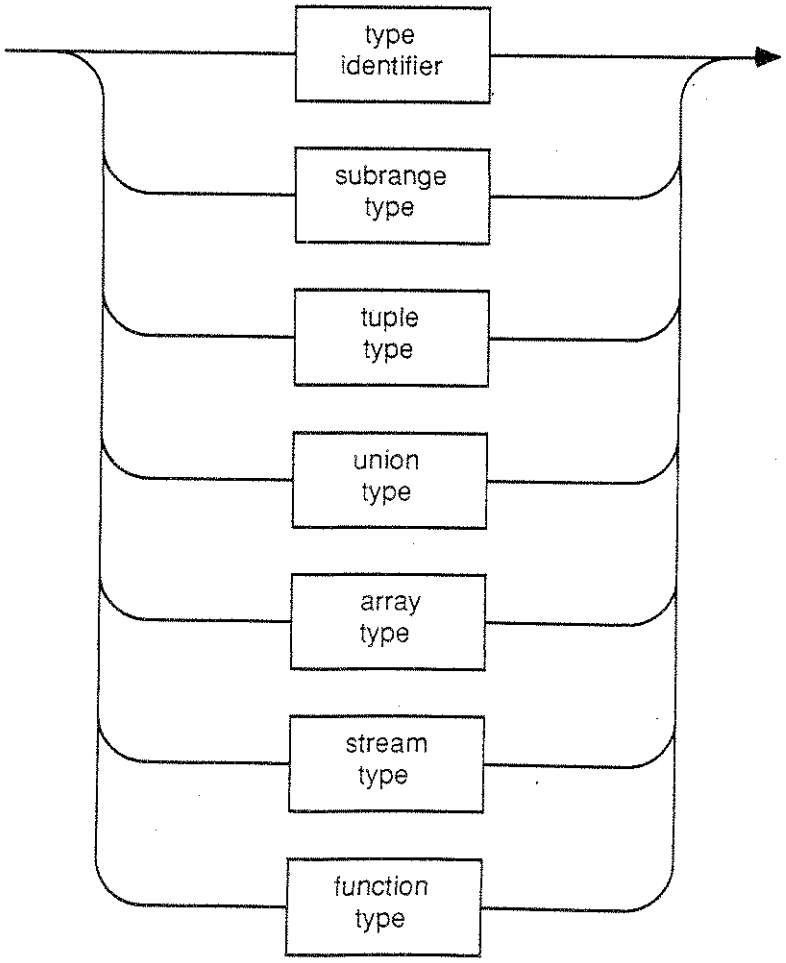
Parameter List



Parameter List Part

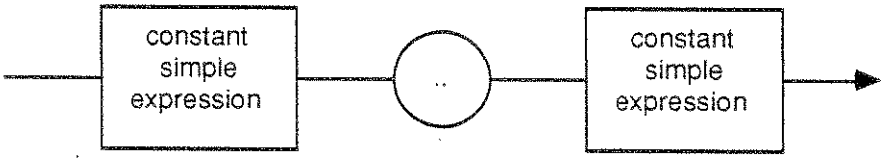


Type Body

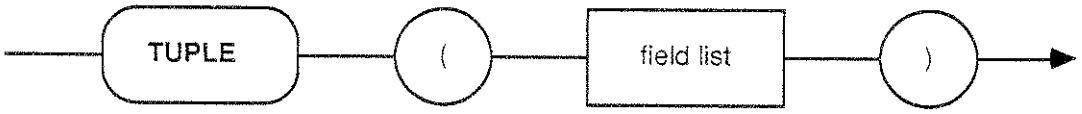


Appendix A - IDA Syntax continued

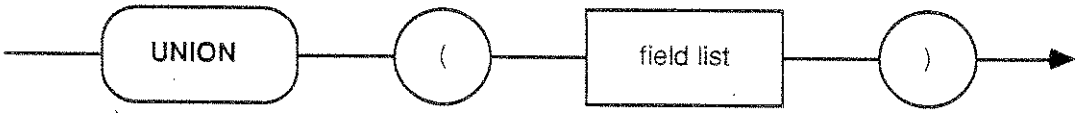
Subrange Type



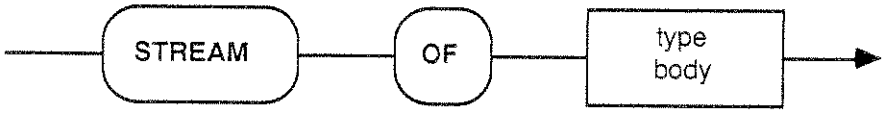
Tuple Type



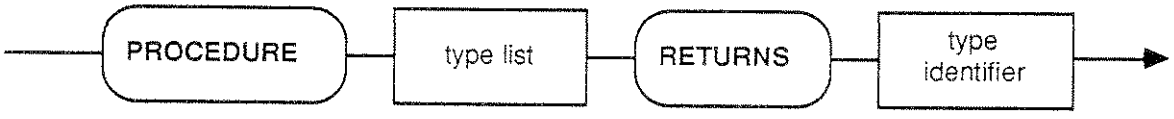
Union Type



Stream Type

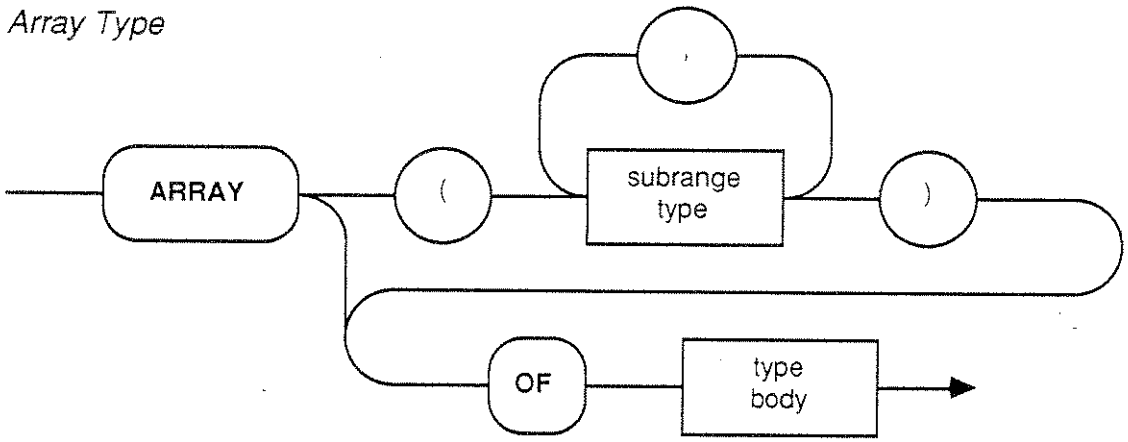


Procedure Type

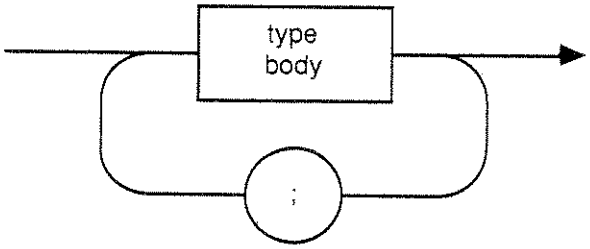


Appendix A - IDA Syntax continued

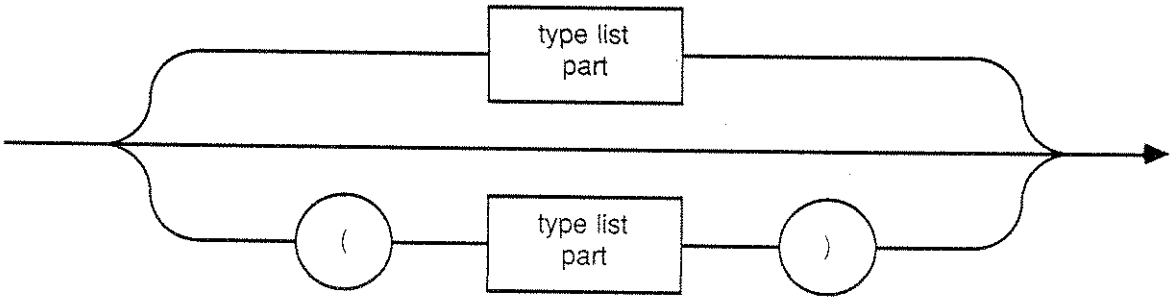
Array Type



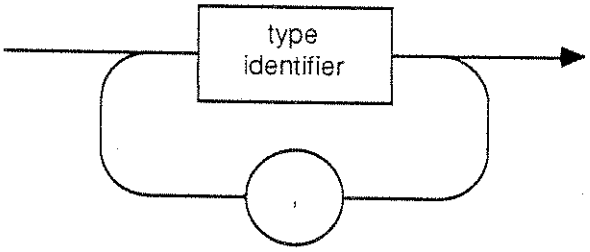
Field List



Type List

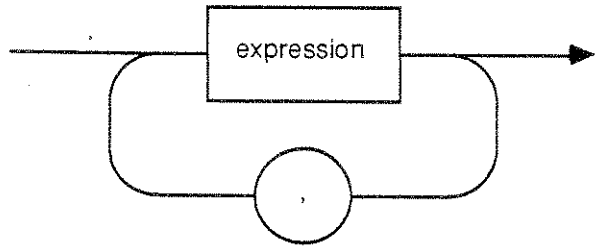


Type List Part

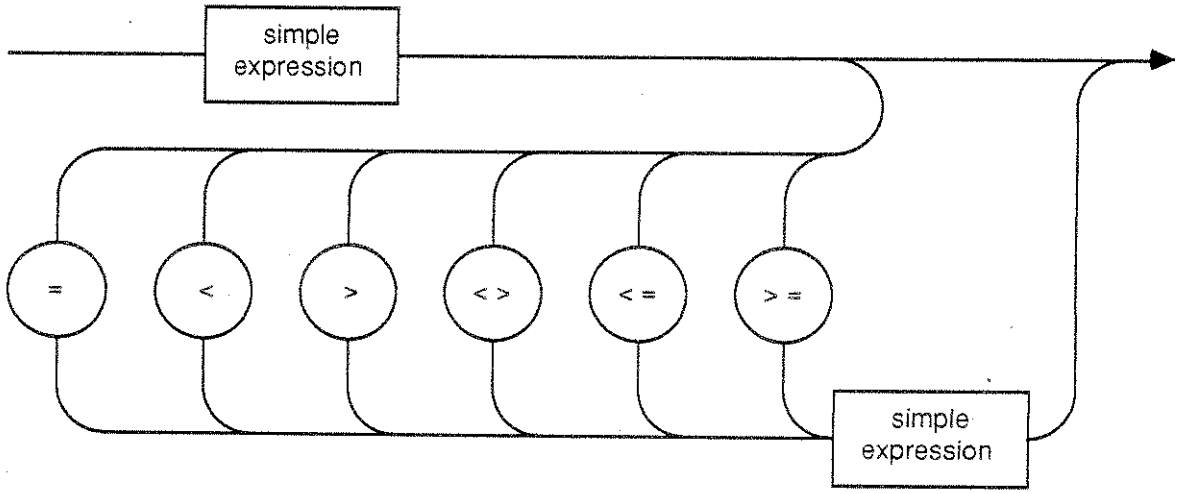


Appendix A - IDA Syntax continued

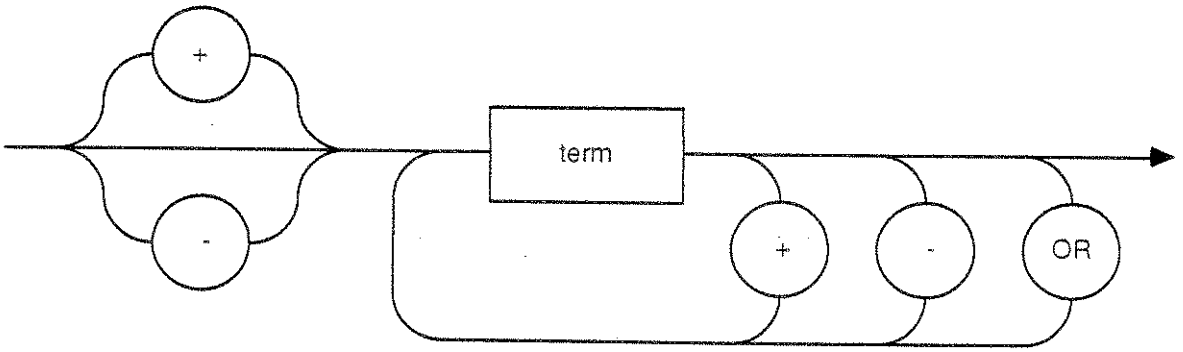
Tuple Expression



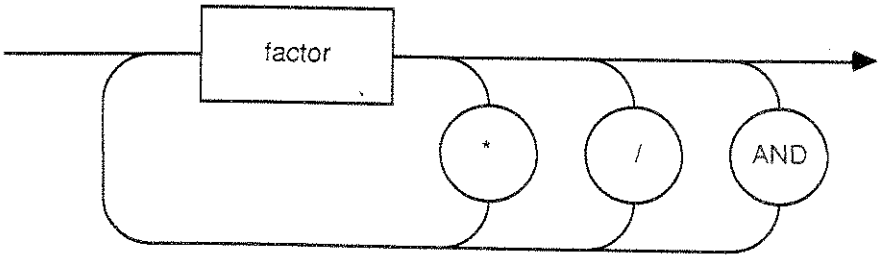
Expression



Simple Expression

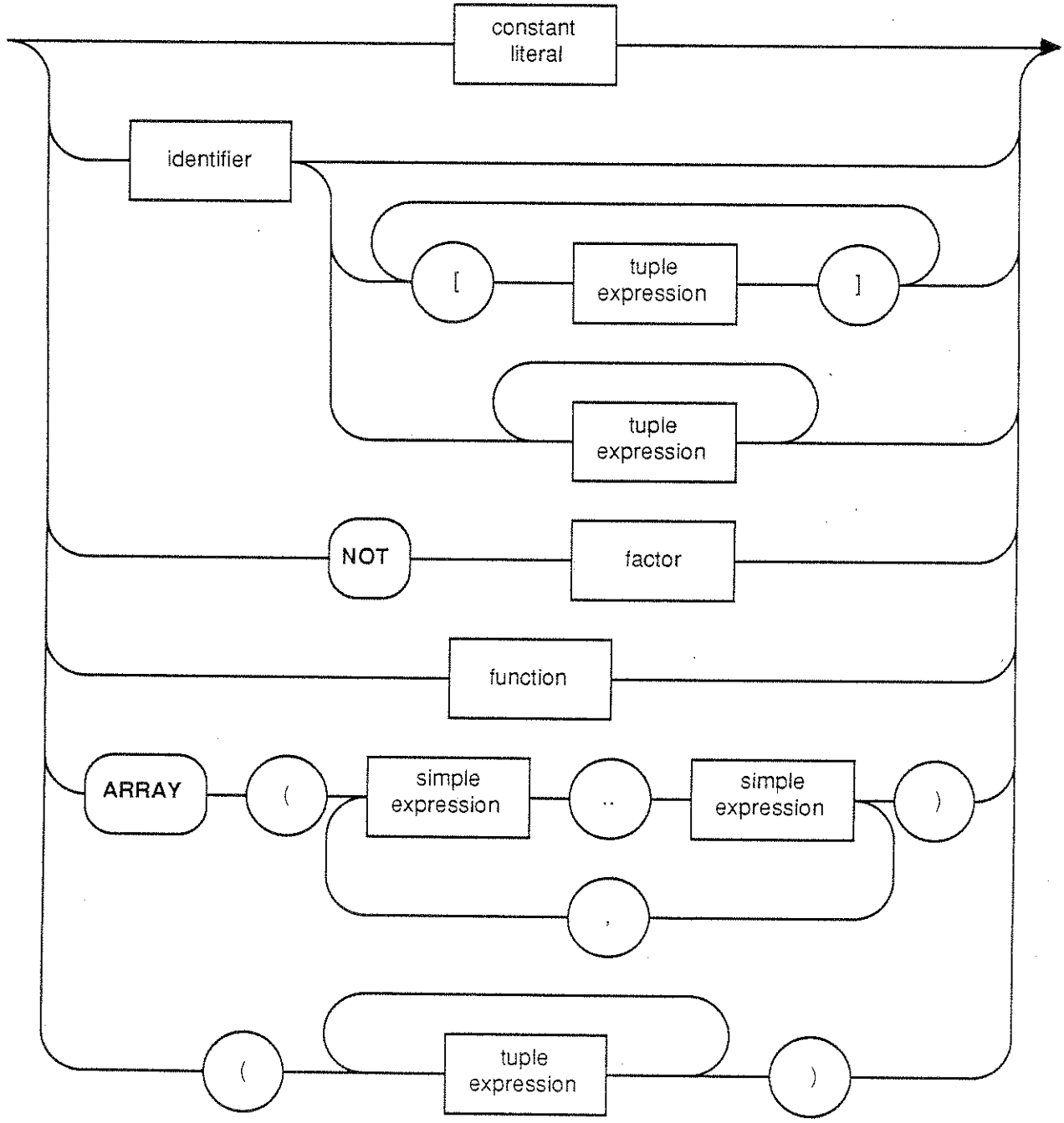


Term



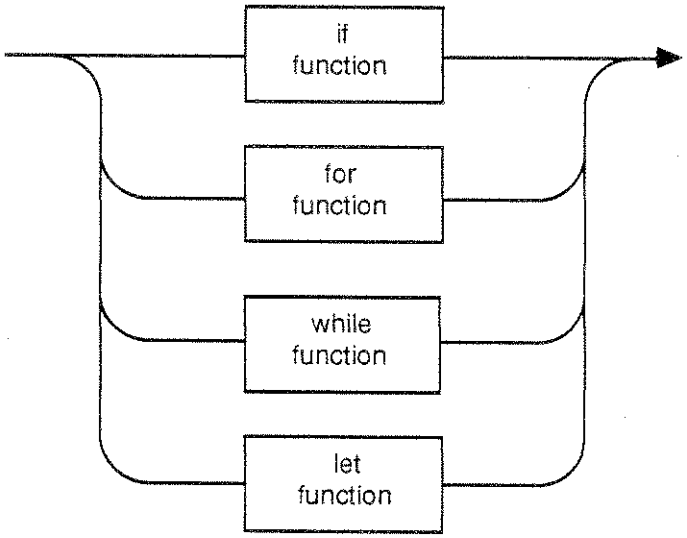
Appendix A - IDA Syntax continued

Factor

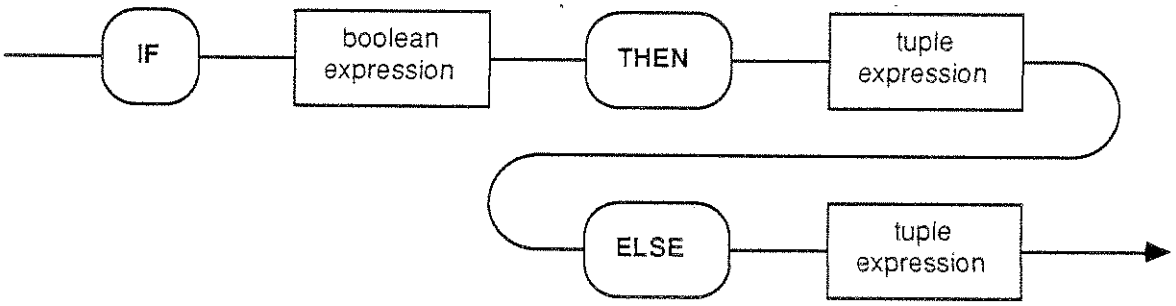


Appendix A - IDA Syntax continued

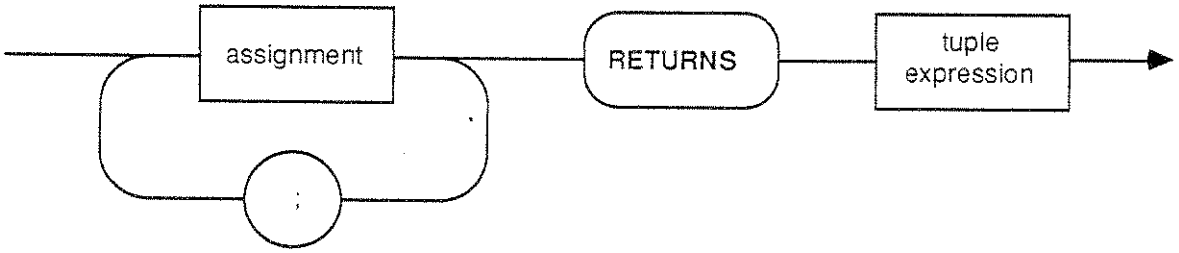
Function



If Function

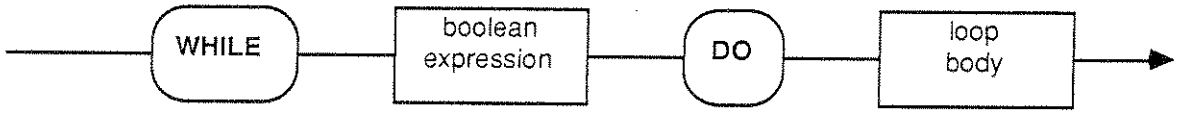


Loop Body

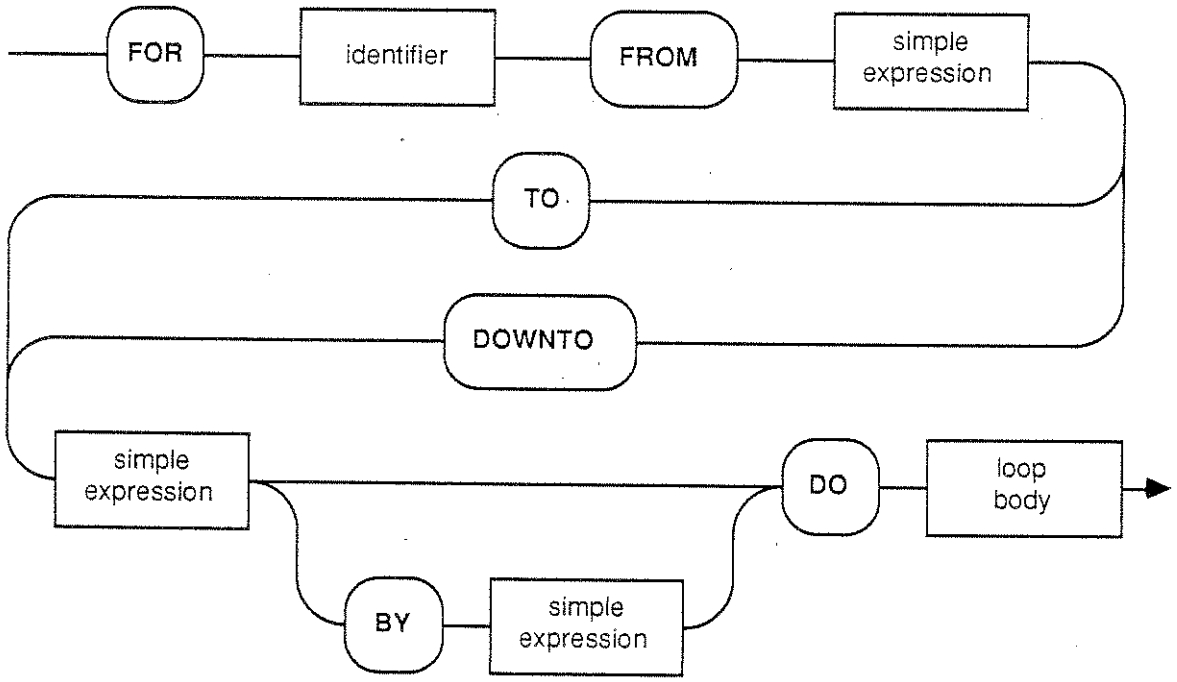


Appendix A - IDA Syntax continued

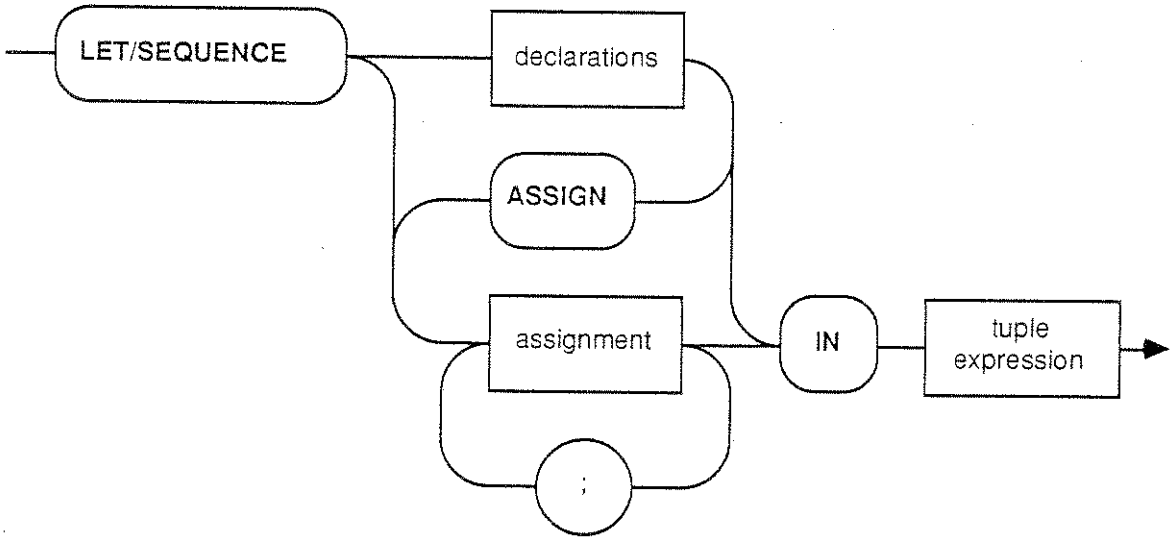
While Function



For Function

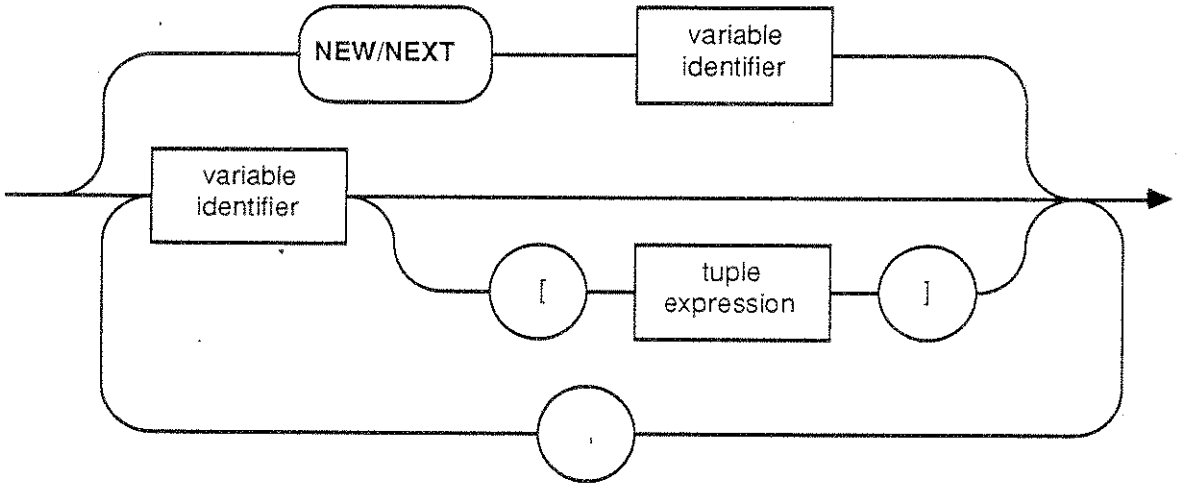


Let/Sequence Function

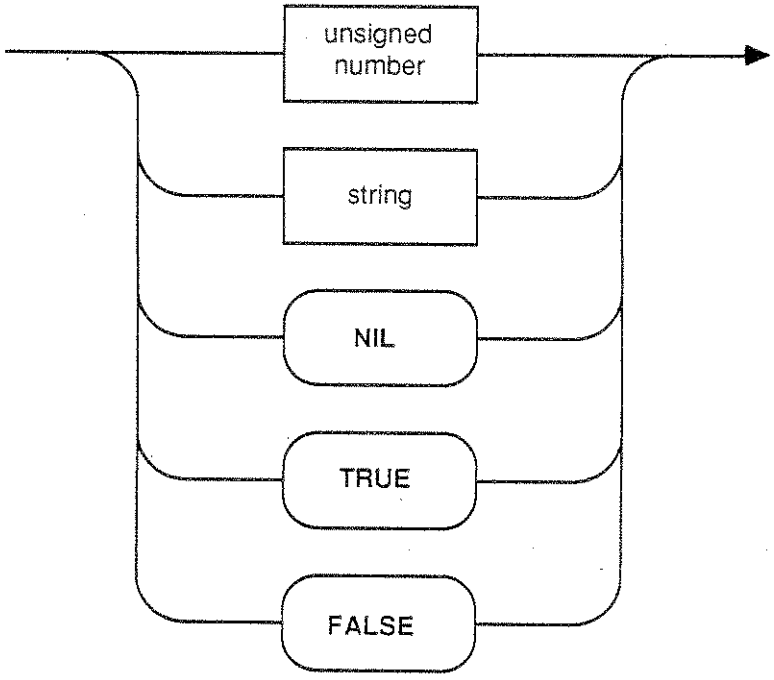


Appendix A - IDA Syntax continued

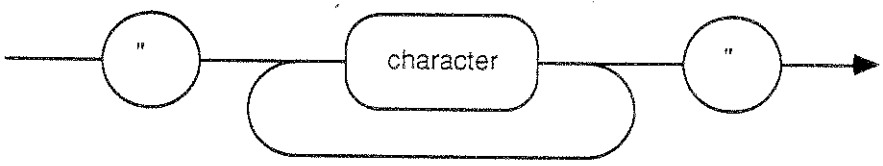
Variable



Constant Literal

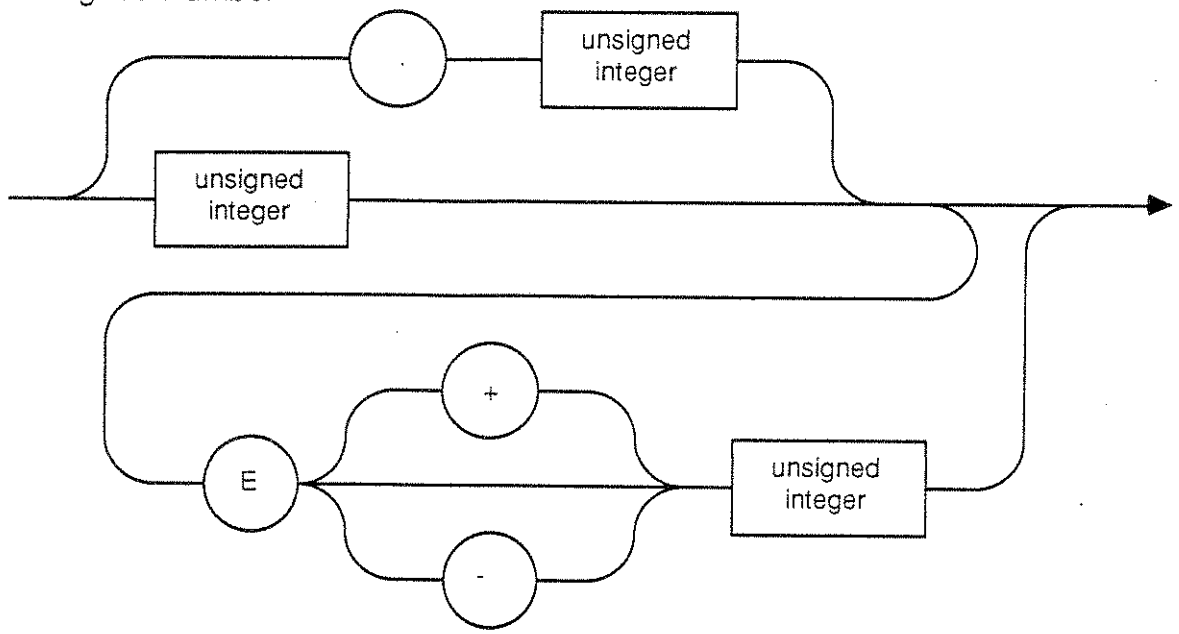


String

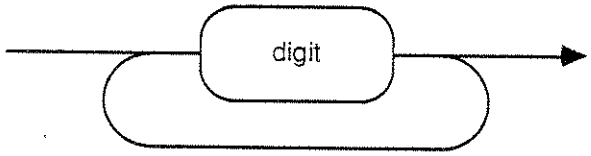


Appendix A - IDA Syntax continued

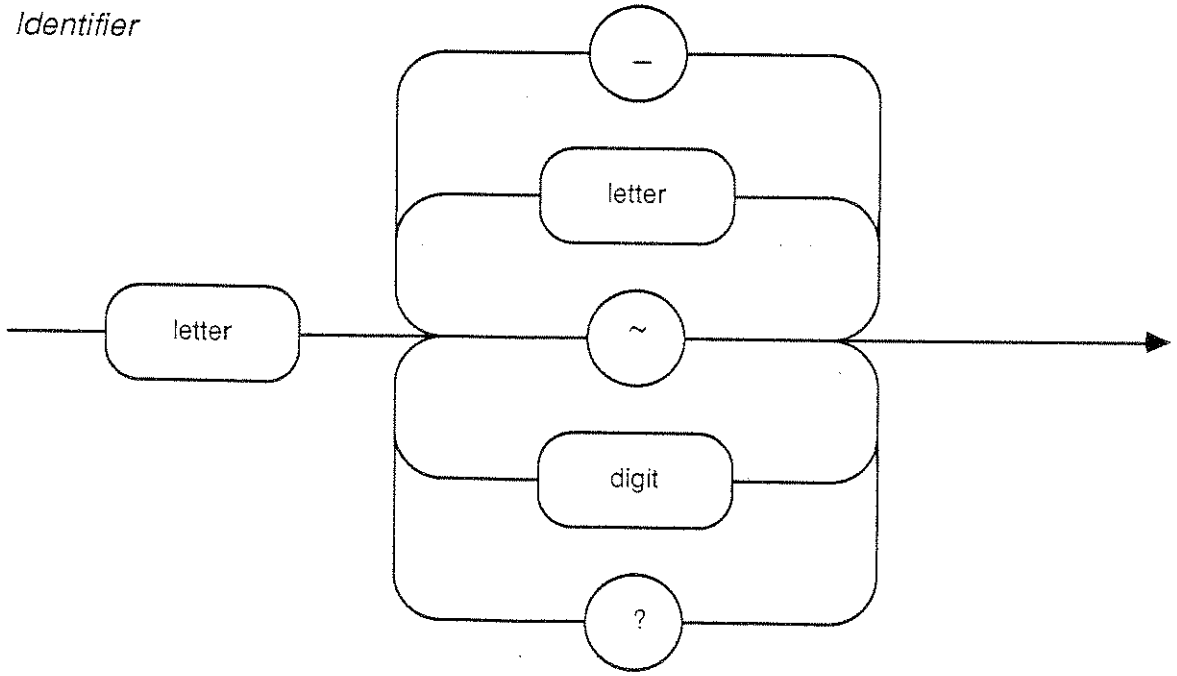
Unsigned Number



Unsigned Integer



Identifier



Appendix B - IDA EBNF

```
<block>                  ::=  <declaration> <tuple expression> ';'
<declaration>            ::=  <empty> |
                              <constant declaration> |
                              <type declaration> |
                              <variable declaration> |
                              <procedure declaration>
<constant declaration>   ::=  const <constant binding> { ';' <constant binding> } ';'
<constant binding>       ::=  <identifier> '=' <expression>
<type declaration>       ::=  type <type binding> { ';' <type binding> } ';'
<type binding>           ::=  <type identifier> '=' <type body>
<variable declaration >  ::=  var <variable binding> { ';' <variable binding> } ';'
<variable binding>       ::=  <identifier> { ',' <identifier> } ':' <type body>
<procedure declaration>  ::=  def <identifier> <parameter list>
                              [ returns <type identifier> ] '=' <tuple expression> ';'
<parameter list>         ::=  <empty>|
                              <parameter list part> |
                              '(' <parameter list part> ')'
<parameter list part>    ::=  <parameter binding> { ';' <parameter binding> }
<parameter binding>      ::=  <identifier> [ ':' <type identifier> ]
<type body>              ::=  <type identifier> |
                              <subrange type> |
                              <tuple type> |
                              <union type> |
                              <array type> |
                              <stream type> |
                              <procedure type>
<subrange type>          ::=  <constant simple expression> '..' <constant simple expression>
<tuple type>             ::=  tuple '(' <field list> ')'
<union type>             ::=  union '(' <field list> ')'
<stream type>            ::=  stream of <type body>
<procedure type>         ::=  procedure <type list> returns <type identifier>
<array type>             ::=  array [ '(' <subrange type> { ',' <subrange type> } ')' ]
                              of <type body>
<field list>             ::=  <type body> { ';' <type body> }
<type list>              ::=  <empty> |
                              <type list part> |
                              '(' <type list part> ')'
<type list part>         ::=  <type identifier> { ',' <type identifier> }
<tuple expression>       ::=  <expression> { ',' <expression> }
<expression>             ::=  <simple expression>
                              { <comparitor> <simple expression> }
<comparitor>             ::=  '=' | '<' | '>' | '<>' | '<=' | '>='
<simple expression>      ::=  [ <sign> ] <term> { <simple exp op> <term> }
<simple exp op>          ::=  '+' | '-' | 'or'
<term>                   ::=  <factor> { <term op> <factor> }
<term op>                ::=  '*' | '/' | 'mod' | 'and'
<factor>                 ::=  <constant literal> |
                              <ident exp> |
                              not <factor> |
                              <function> |
                              array '(' <array dimension> { ',' <array dimension> } ')' |
                              '(' <tuple expression> ')'
```

Appendix B - IDA EBNF continued

```
<ident exp>                 ::=  <identifier>
                                 { <array index> { <array index> } |
                                 <tuple expression> { <tuple expression> } }
<array index>               ::=  '[' <tuple expression> ']'
<array dimension>           ::=  <simple expression> '..' <simple expression>
<function>                  ::=  <if function> |
                                 <for function> |
                                 <while function> |
                                 <let function>
<while function>            ::=  while <boolean expression> do <loop body>
<for function>              ::=  for <identifier>
                                 from <simple expression>
                                 <direction> <simple expression>
                                 [ by <simple expression> ]
                                 do <loop body>
<direction>                 ::=  to | downto
<loop body>                 ::=  <assignment> { ';' <assignment> }
                                 returns <tuple expression>
<if function>               ::=  if <boolean expression> then <tuple expression>
                                 else <tuple expression>
<let/sequence function>     ::=  <start> <let assignment> in <tuple expression>
<start>                     ::=  let | sequence
<let assignment>            ::=  <assignment part> |
                                 <declaration> [ assign <assignment part> ]
<assignment part>           ::=  <assignment> { ';' <assignment> }
<assignment>                ::=  [ <varible> <parameter> ] '=' <tuple expression>
<variable>                  ::=  new <identifier> |
                                 next <identifier>|
                                 <variable part>
<variable part>             ::=  <identifier> <var qualifier> { ',' <variable part> }
<var qualifier>             ::=  <empty> |
                                 '[' <tuple expression> ']'
<constant literal>          ::=  <unsigned number> |
                                 <string> |
                                 nil | true | false
<unsigned number>           ::=  <unsigned integer> <exponent> |
                                 <unsigned integer> '.' <unsigned integer> <exponent> |
                                 '.' <unsigned integer> <exponent>
<exponent>                  ::=  <empty> |
                                 'E' <sign> <unsigned integer>
<sign>                      ::=  <empty> | '+' | '-'
<unsigned integer>          ::=  <digit> { <digit> }
<string>                    ::=  '"' <character> { <character> } '"'
<identifier>                ::=  <letter> { '_' | <letter> | '~' | <digit> | '?' }
<digit>                     ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter>                    ::=  'a'..'z' | 'A'..'Z'
```

Appendix C - IDA Example Programs

```
const
   n = 10;
type
   matrix = array(1..n,1..n) of number;

def mm a:matrix b:matrix returns matrix =
let
   var
      c : matrix;
      i, j, k, sum : number;

   assign
      = for i from 1 to n do
         = for j from 1 to n do
            c[i,j] =  let
                        sum = 0
                      in
                        for k from 1 to n do
                          next sum = sum + a[i,k] * b[k,j]
                        returns sum
         returns ()
      returns ()
in
   c;
```

Example D.1 - Matrix Multiply

Appendix C - IDA Example Programs

```
% definitions used in this 'wavefront.ida'
const
   high = 10;
type
   sea = array(1..high,1..high) of number

def wave returns sea =
let
   var
      ocean : sea;
      i, j : number;

   def min a:number b:number returns number =
   if a<b then a else b;

   assign
      = for i from 1 to high do
         ocean[i,1] = i
      returns ();
      = for j from 2 to high do
         ocean[1,j] = j
      returns ();
      = for i from 2 to high do
         = for j from 2 to high do
            ocean[i,j] = min ocean[i-1,k] ocean[i,j-1]
         returns ()
      returns ()
in
   ocean;

% query of 'wavefront.ida'
wave;
```

Example D.2 - wavefront.ida

Appendix D - IDA 'man' page

IDA(1) USER COMMANDS IDA(1)

NAME

ida – RMIT Id Nouveau compiler

SYNOPSIS

ida [**–d**] [**–f**] [**–if1**] [**–i2**] [**–l**] [**–p**] [**–w**] [**–dryrun**] [**–I***pathname*] [**–D***name*] [**–U***name*] [**–T**] [**–R**] *filename*

DESCRIPTION

ida will compile IdA source files into an intermediate form called *if1*. The resulting file (*filename.if1*) is in a form acceptable to the RMIT IF1 translator which produces an *i2* source file. The *i2* file is passed through the I2 assembler to create the *dfo* object file for execution on the RMIT dataflow machine, simulator and interpreter.

OPTIONS

–d	Enable code-generation debugging.
–f	Generate *i2* code (as opposed to *if1* code).
–if1	Force an *if1* file to be produced (as opposed to a *dfo* file).
–i2	Force an *i2* file to be produced (as opposed to a *dfo* file).
–l	Enable full listing to tty of source code.
–p	Enable parser debugging.
–w	Disable warning diagnostics.
–dryrun	Show but do not execute the commands constructed by the compiler driver.

The following options are passed on to the preprocessor (*cpp*(1)).

–I*pathname*	Look in *pathname* for **#include** files.
–D*name*	Defines a preprocessor name.
–U*name*	Undefines a preprocessor name.
–T	Use only the first eight characters for **#define** names.
–R	Allow recursive preprocessor macros.

FILES

file.ida	IdA source file
file.if1	object file

SEE ALSO

dfs(1), sdfs(1), tdfs(1), i2(1), if1(1), sisal(1).

IDA: A Dataflow Programming Language (TR-112-075R)

BUGS

I'm not silly enough to leave my contact address. I'm busy enough as it is!!