An Overview of ETAC

1 February 2019

Copyright © **Victor Vella** (2018, 2019)
All rights reserved.

Other Related ETAC Documents

ETAC_Preliminaries.pdf	Preliminaries before using ETAC
ETACProgLang(Official).pdf	The Official ETAC Programming Language
RunETAC.chm	Run ETAC Scripts Help
ETACWithCPP.pdf	ETAC: Interacting with C++
ETACCompiler.pdf	The ETAC Compiler
ETACCompiler.chm	ETAC Compiler Help
ETACErrorCodes.pdf	ETAC Compilation and Run-time Error Codes

Legal Information

ETAC™ and the **ETAC logo**  are unregistered trademarks of Victor Vella for *computer software incorporating an implementation of a computer programming language*. There may be other owners of the "ETAC" trademark used for other purposes.

MS-DOS® and **Windows**® are registered or unregistered trademarks of Microsoft Corporation.

PostScript® is a registered trademark of Adobe Systems Incorporated.

This document is copyright © by Victor Vella (2019). All rights reserved. Permission is hereby granted to make any number of exact electronic copies of this document without any remuneration whatsoever. Permission is also granted to make annotated electronic copies of this document for personal use only. Except for the permissions granted, and apart from any fair dealing as permitted under the relevant Copyright Act, no part of this document may be reproduced or transmitted in any form or by any means without the express permission of the author. The copyright of this document shall remain entirely with the original copyright holder.

The author of this document shall not be liable for any direct or indirect consequences arising with respect to the use of all or any part of the information in this document, even if such information is inaccurate or in error. The information in this document is subject to change without notice.

Contents

Document Conventions

The following symbolic conventions are used in this document.

Symbol	Meaning
‹*x*›	separates *x* as a unit of information from the surrounding text.
x···	means zero, one, or more of the same kind as *x*.
[*x*]	means that *x* optional.
(*x*)	groups *x* as a unit.
x{*y*}	means that if *x* is not present, then *y* is the default.
x|*y*	means that only *x* or *y* applies, but not both (could have more than two options).
...	represents omitted text (as usual).
$_{WS}$	represents a whitespace character.
$_{EL}$	represents the character or characters indicating the end of a text line.
XX$_{H}$	*XX* represents a two digit number in hexadecimal base.
text	maroon coloured italic text is a link to the text's definition.
♦	indicates the end of a block of text.

An Overview of ETAC

This document is for version **1** of the **ETAC Programming Language** implemented in program RunETAC.exe version **2-0-6-ena**.

(Australian English)

1. Introduction

ETAC™ (pronounced: E-tack) is a dictionary and stack based interpreted script programming language. ETAC is designed as a programming tool for use by programmers, not as an ornamental work of art for computer academics to marvel at. As a consequence, ETAC is an easy and intuitive language to use with no gimmickry or ornamental features. However, because ETAC is extreme versatile, and being a dictionary and stack based language, it is designed for experienced professional programmers (meaning programmers having a professional attitude). The language is not designed with the restrictions to cater for amateur programmers. The ETAC programming language is not based on any other programming language.

The ETAC programming language is a high-level syntactic enhancement of the TAC (pronounced: tack) programming language (which was not publicly released). TAC operates by sequentially activating text tokens from a file in conjunction with three object stacks one of which contains dictionaries of defined token words. Such a language that sequentially activates text tokens is called a "token activated language" or "TAL" for short. TAC is an acronym for **T**oken **A**ctivated **C**ode. Instead of merely activating tokens from left to right, as is the case for other TALs, TAC activates programmer-determined groups of tokens sequentially from left to right, but the tokens in each group are activated from right to left ("reverse-flow"). Such a system is called "sequential reverse-flow activation" (explained in more detail in this document). A TAC program, therefore, is not strictly written in postfix notation (unless the programmer wants to), but in groups of tokens with prefix notation activated in reverse. Such a system can be enhanced to incorporate a high-level syntactic structure without sacrificing the versatility of a TAL. The result of such an enhancement is that a TAC program could be written using any combinations of high-level syntax (as is typical of traditional high-level block structured languages) and TAL type syntax (as is typical of TALs such as PostScript® and FORTH) in a seamless manner. ETAC incorporates the said high-level syntactic enhancement of TAC, and is an acronym for **E**nhanced **T**oken **A**ctivated **C**ode.

An ETAC program is written in a single-byte text file (only Western European **Windows**® code page 1252 is supported) which is interpreted directly by the **Run ETAC Scripts** program (RunETAC.exe). Since ETAC is a stack-based language, the ETAC programmer is responsible for maintaining the *TAC stacks* (especially the *object stack*).

RunETAC.exe can be executed either from the **MS-DOS**® or **Windows**® environment via a command line. From within **Windows**®, the command line is typically entered in a shortcut file. Note that ETAC is not released for other platforms or operating systems — it is only released for the **Windows**® operating system using the x86 (32-bit) architecture (can also run on the x64 (64-bit) architecture) beginning with **Windows**® **XP**. Also, ETAC is released only in the (Australian) English language. The help file (RunETAC.chm) for **Run ETAC Scripts** contains full details on how to run an ETAC program.

This document presents an overview of the ETAC programming language for those who want to gain a general idea of the nature of the language. This overview is NOT THE SPECIFICATION of the ETAC programming language, and does not contain all the features and details of the language. To understand this document properly, the reader needs to have a general knowledge of computer programming and, in particular, an understanding of the basic concepts of dictionary and stack based programming languages.

2. ETAC Code Sample

For those who cannot wait to see what *ETAC text script* looks like, an ETAC code sample (without explanation) for illustrative purposes is presented below. Notice that the code is in the form of traditional block structured style syntax, but keep in mind that ETAC is a dictionary and stack based language (note the **pop** and **swap** *commands* on the tenth line, which are typical commands of a stack-based programming language). Comments are shown in green, and ignored by the *ETAC interpreter*. Incidentally, the **bold blue** text represents some of the syntactic enhancements of ETAC over TAC.

```
[* Function Definitions *]
[* Determines if a sub-string exists in a string sequence. *]
FindString :- fnt:(pStrSeq[*str-seq*] pStr[*str*]) [* => bool *]
{
RtnVal :- false; [*rtn*]
SrcStr :- ?;

   do with SrcStr of pStrSeq while not RtnVal
   {
      RtnVal := ( pop swap find_str pStr SrcStr != -1 ); [* Tenth line. *]
   };

   RtnVal; [*RETURN*]
};

[* Splits a string sequence at a line number and character offset. *]
SplitLines :- fnt:(pTextLines[*str-seq*] pLineNum[*int*] pCharOff[*int*])
{
ExtrLine :- ?;

   ExtrLine := pTextLines%[pLineNum] := extr_str 0 pCharOff pTextLines%[pLineNum];

   if ( |pTextLines| = pLineNum ) then
      {pTextLines +:= ExtrLine;}
   else
      {pTextLines<%[(pLineNum + 1)] := ExtrLine;}
   endif;
};

[* Function Calls *]
TextLines :- ["A string here", "Another one", "The final string"];
Found := FindString(TextLines "one");
SplitLines(TextLines 2 7); ♦
```

The function `SplitLines` (shown in the example above) can be written as an ETAC *procedure* in typical stack-based syntax without using variables, as shown below.

```
[* Splits a string sequence at a line number and character offset. *]
SplitLines :- [* str-seq line-num-int char-off-int => - *]
{
   copy 2; swap; get_elm; copy_any 4; extr_str 0; copy_any 3; swap; copy_any 5; put_elm;

   copy_any 3; copy_any 3; size; swap; pop; eq;
   if_else {mk_seq 1; roll -1 3; add2 1; insert;} {swap; add2; roll -1 3; pop;};

   pop; pop;
};

[* Function Call *]
TextLines :- ["A string here", "Another one", "The final string"];
SplitLines TextLines 2 7; ♦
```

This second example of `SplitLines` is significantly more difficult to read and write than the first example of `SplitLines`. In fact, writing and debugging stack-based style code without using variables could take up to ten times longer than writing and debugging the same code in high-level block structured style code using variables. And using stack-based syntax is not necessarily more efficient to execute than using the equivalent high-level block structured syntax. For this reason,

ETAC text script is typically written in high-level block structured syntax. Note, however, that *ETAC text script* can be written seamlessly as a mixture of both styles of coding.

To further illustrate the syntactic versatility of the ETAC programming language, consider a function, `Factorial`, to calculate the factorial of an integer. The function is defined in two ways. The first way uses the recursive method in a traditional block structured style syntax. The code uses a parameter and a local variable. A programmer not familiar with the ETAC programming language would not be able to tell that the code is written in a dictionary and stack based language.

```
[* Traditional block structured language form of the factorial function. *]
Factorial :- fnt:(pNum[*int*]) [* => factorial-int *]
{
Num :- pNum;

   if ( pNum = 0 ) then
      {pNum := 1;}
   endif;

   if ( pNum > 1 ) then
      {Num := Factorial((Num - 1));}
   else
      {Num := 1;}
   endif;

   (pNum * Num); [*RETURN*]
}; ♦
```

The second way to define the `Factorial` function uses a *procedure* definition without using any parameters, local variables, or the *dictionary stack* (other than for storing the actual *procedure* itself). It uses only the *object stack*. This code runs the most efficiently; it does not use the recursive method. The factorial factors (1, 2, **…**, *n*) are all pushed onto the *object stack* first, then multiplied together.

```
[* Stack-based language form of the factorial function using the object stack only. *]
Factorial :- [* int => factorial-int *]
{
   (* 1 1 do_for {} 1 1 swap); [*RETURN*]
}; ♦
```

As can be seen from the illustrations above, the ETAC programming language is, arguably, the first of an evolutionary step of dictionary and stack based token activated programming languages, capable of full traditional high-level block structured syntax with the versatility and efficiency of a token activated stack-based language.

3. Features of ETAC

Some of the noteworthy features of ETAC are:

1. Uses *sequential reverse-flow activation*.
2. An *operator* can be placed in any position within the parenthesis of an *operator expression* (‘non-fix’ notation).
3. ‘if’, ‘choice’, and ‘iterative’ *command* structures can be modified at run-time. The structure of accessing *sequences* of multiple depth can also be modified at run-time.
4. Any variable can have any type of value at any time, including another variable as value.
5. The size of *sequences* can change dynamically, and the *sequences* can have any mixture of elements including other *sequences*.
6. Capable of full traditional high-level block structured language style syntax, as well as traditional stack-based language style, or any mixture of both.
7. Can construct *sequences* and *procedures* at call-time (of a *procedure*).
8. Can internally create *ETAC text script* and execute it in a new *ETAC session* which shares *command* and *operator* definitions with the *main ETAC session*.

9. Can compile *ETAC script* into binary form for convenience (but is not required) via the **ETAC Compiler** program.
10. Contains an internal interactive debugger for visually tracing the *activation* of *script tokens* in *ETAC scripts* when running in debug mode.

In addition, a C++ computer programmer can extend the native set of *commands* and *operators* via *external TAC libraries* (implemented as dynamic linked libraries). The *ETAC interpreter* contains an internally implemented *standard TAC library*. An ETAC program can also execute C++ application program functions, which allows an application program to use ETAC as a macro language via a special ETAC dynamic linked library (AppETAC.dll).

Details of the features listed above (and other features) are presented in the rest of this document. However, not all the features of the ETAC programming language are presented. The official definition of the ETAC programming language is presented in the document ETACProgLang(Official).pdf.

4. Sequential Reverse-flow Activation

To understand how an ETAC program operates, it is essential to understand the concept of *sequential reverse-flow activation*.

In a traditional TAL (a **t**oken **a**ctivated **l**anguage like the PostScript® and FORTH programming languages), the tokens of a script written in such a language are activated from left to right. For example, to perform the calculation of a mathematical expression like ‹2 + 3 × 5›, a (hypothetical) TAL script might contain something like

```
3 5 mult 2 add
```

This pushes the numbers `3` and `5` onto the stack then multiplies those top two numbers (`3` and `5`) and adds the result to the pushed third number (`2`), leaving the result (`17`) on the stack. The order of the operations are strictly from left to right using postfix notation. With *sequential reverse-flow activation*, the order of operation is determined by the programmer, as illustrated by the equivalent script fragments below.

(1) `mult 5 3; add 2;`

(2) `3; mult 5; add 2;`

(3) `3; add 2 mult 5;`

(4) `add 2 mult 5 3;`

(5) `3; 5; mult; 2; add;`

Each of the five examples above is an alternative way of performing the same calculation. The tokens in each example are separated into groups each of which ends with a ‘marker token’ (the semicolon). Each group is called a *token statement*. The list of *token statements* are performed from left to right sequentially, but within each *token statement*, the tokens are activated from right to left (“reverse-flow”). So, in example **(1)**, the number `3` is pushed onto the stack, followed by the number `5`, then the two numbers are multiplied (`mult`) together leaving the result (`15`) on the stack. That ends the activation of the first *token statement*. Next, the number `2` is pushed onto the stack, then that `2` and the previous result (`15`) are added (`add`) together, leaving the number `17` on the stack. That ends the activation of the second *token statement*. The other examples operate in a similar manner; each example effectively operates in the same way. Notice that the script in example **(4)** can be regarded as using strictly prefix notation, and the script in example **(5)** can be regarded as using strictly postfix notation.

A TAL using *sequential reverse-flow activation* is neither necessarily strictly written in postfix notation nor strictly written in prefix notation. It is said to be written in ‘non-fix notation’. One advantage in using *sequential reverse-flow activation* in a TAL is that the tokens can be arranged to be more easily understood as groups of *token statements* using prefix notation. Example (4), above, is easier to understand than example (5) or the original example. Another advantage is that a suitably designed interpreter can internally convert high-level block structured syntax to *token statements* in

prefix notation. For illustration purposes only, the high-level data member selection statement ‹MyData.Member;› would be internally converted to the *token statement* ‹**SELECT** MyData Member;›, and the high-level member function call statement ‹MyData.Fnt(10, "hello");› would be internally converted to the *token statement* ‹**DATA** MyData {**CALL** Fnt;} 10 "hello";›. SELECT, DATA, and CALL would be internal *commands* known only to the interpreter. The conversions can be more complicated than shown in those illustrations. The *ETAC interpreter* uses a similar scheme to internally convert high-level syntax to *token statements* in prefix notation.

It is important to note that the tokens within a *token statement* are processed from right to left by the *ETAC interpreter*. This implies that, for example, in a *token statement* such as ‹Cmd 21 "string" 5.6;›, the topmost *stack object* on the *object stack* before Cmd is *activated* is 21 not 5.6. In other words, the order of *stack objects* from top to bottom on a *TAC stack* is represented by the order of the tokens from left to right in a *token statement*. For example, a *command* that concatenates three strings and leaves the result on the *object stack* would have its input arguments and output result documented as ‹[* str1 str2 str3 => str *]›, where str1, str2, and str3 represent the input arguments with str1 representing the topmost string *stack object*, and str representing the output result (again, with the topmost string *stack object* presented on the left in case there is more than one output object).

5. ETAC – Operational Overview

Readable code conforming to the ETAC programming language typically exists in one or more text files written by an ETAC programmer. The *ETAC text script* in a file is in the form of *script tokens*, most of which represent *stack objects*. There are *script tokens* that represent integer and decimal numbers, strings, *commands* and *operators*, memory blocks, markers, *sequences* of *stack objects*, and null *stack objects*. The *ETAC interpreter* consists of a *script interpreter*, a *binary interpreter*, and a *TAC processor*. The *script interpreter* reads, pre-processes, and parses all the *script tokens* in the *ETAC text script*, checks for correct syntax, then passes each *script token* (with a few exceptions) to the *TAC processor*. The *TAC processor* creates a *stack object* from each *script token* passed to it then *activates* that *stack object* according to its type. A *stack object* carries out one of a number of predefined actions when it gets *activated*.

The *TAC processor* contains three *TAC stacks*: the *object stack* (where most of the action occurs), the *operator stack*, and the *dictionary stack*. An action can cause a *stack object* to be pushed onto one of the *TAC stacks*, *activate* other *stack objects*, or create other *stack objects* based on existing *stack objects*. A *sequence* of *stack objects* can be created, and that *sequence* is itself a *stack object*. Thus, *subsequences* of *stack objects* can be created. The elements of a *sequence* of *stack objects* can be *activated*.

A certain class of *stack objects* (*comops*) can contain a name. The names of such *stack objects* are held in a *dictionary* on the *dictionary stack*, and each name is associated with a *stack object* (which can contain a *sequence* of *stack objects*). When such a named *stack object* gets *activated*, its name is found in a *dictionary item* on the *dictionary stack*, and the associated *stack object* or *sequence* gets *activated*. In this way, more complex *stack objects* can be built up from less complex ones.

Variable allocation and type checking is done at run-time, and variables are not declared as having specific types. All data and processes defined in ETAC by a programmer are encapsulated in *stack objects* that contain values. *ETAC text script* can be pre-processed via pre-processor directives.

6. Stack Objects

A *stack object* is an entity that includes the following two properties: a type and a corresponding value. All programmer data in ETAC exists in *stack objects*. A *stack object* normally resides on a *TAC stack*, but can exist temporarily in the *ETAC interpreter*.

The following table lists the different types of *stack objects* and their values.

Type	Value
Integer	An integer from –2,147,483,648 to 2,147,483,647.
Decimal	A decimal number from $\pm 2.2250738585072014 \times 10^{-308}$ to $\pm 1.7976931348623158 \times 10^{308}$ and 0.0 (zero).
String	A consecutive sequence of single-byte characters possibly containing escape codes.
Sequence	An internal reference to any number (including zero) of indexed *stack objects* understood as a unit.
Procedure	An internal reference to a *sequence* whose elements get *activated*.
Command	A name having the syntax of a *comop identifier*.
Operator	A name having the syntax of a *comop identifier*.
Memory	An internal reference to a block of memory.
Dictionary	An internal reference to a *dictionary*.
Mark	A special internal integer from `0` to `7`.
Null	The internal integer zero.
EXE	An integer representing a *custom comop number*. ♦

In addition, there are emulated *stack objects* internally implemented by the *ETAC interpreter* in terms of combinations of certain *stack objects* described above. The emulated *stack objects* are listed below.

Type	Value
Function	An internal emulation containing a *procedure* (*ETAC function*).
Data object	An internal emulation containing a *dictionary* (*data object*). ♦

Each *stack object* is associated with a set of possible actions determined by the *stack object's* type. When the *ETAC interpreter activates* a *stack object*, an appropriate one of those actions is performed by default. For example, when a command *stack object* is *activated*, its name is searched for in a *dictionary* on the *dictionary stack*, and the *stack object* associated with that name in the *dictionary* is *activated*. When an integer *stack object* is *activated*, it gets pushed onto the *object stack*. Also, some *stack objects* perform a different action on their second *activation*. An ETAC programmer can alter the action of some *stack objects* (limited to the set of possible actions).

7. The TAC stacks

In general, a 'stack' is a last-in-first-out list of items (LIFO). The 'top' of the stack is the most recent item put onto the stack. There are three stacks (*TAC stacks*) used in ETAC: **(1)** the *object stack* (the main one), **(2)** the *dictionary stack* (for *dictionaries*), and **(3)** the *operator stack* (for operator *TAC objects*). The *TAC stacks* have no defined size (their size is said to be "infinite").

The *stack objects* on all three *TAC stacks* can be manipulated in various ways: The order of the *stack objects* can be changed, *stack objects* can be *copied* and *duplicated*, *stack objects* can be created and deleted, the number of *stack objects* on a *TAC stack* can be obtained.

The ETAC programmer is responsible for maintaining the *TAC stacks*. Any *stack object* put onto a *TAC stack* will remain there until it is removed via programmer code.

7.1 The Object Stack

The *object stack* is the main working *TAC stack*, and can contain any type of *stack object*. The *object stack* is used mainly for passing data for processing among *comop stack objects*. It can also be used for temporarily storing *stack objects*.

7.2 The Dictionary Stack

The *dictionary stack* can contain only *dictionaries*. A *dictionary item* in any *dictionary* on the *dictionary stack* can be searched for by name from the top of the *dictionary stack* to the bottom. When a *dictionary item* is found, its associated *stack object* can be automatically *activated* or pushed onto the *object stack*.

7.3 The Operator Stack

The *operator stack* can contain only operator *stack objects*, and is used to temporarily store the operator *stack object* existing in an *operator expression* until the last *operator* argument has been put onto the *object stack*. Then, the topmost operator *stack object* on the *operator stack* is automatically *activated* to carry out its intended purpose.

8. Comments

Comment text exists between a pair of matching comment delimiters. The opening (left) delimiter of the pair is a square bracket followed immediately by a contiguous series of one or more asterisks. The series is delimited by a non-asterisk. It has the form ‹`[*···*`*c*› (where *c* is a non-asterisk). The matching closing (right) delimiter is a series of contiguous asterisks (with a non-asterisk start delimiter) of the same length as the opening delimiter, followed immediately by a closing square bracket. It has the form ‹*c*`*···*]`› (where *c* is a non-asterisk). The first sequence of characters after the opening delimiter that form the closing delimiter, outside of any comments that have the same matching delimiters, will be regarded as the matching delimiter. Between these two comment delimiters, any characters can exist (except characters forming a matching closing delimiter outside of any comments that have the same matching delimiters).

A comment can be nested within an outer comment (to any level) having the same or different delimiters as the delimiters of the outer comment, for example, ‹`[*I am a comment [* and I am nested*] I am the rest of the comment*]`› is a nested comment with the inner comment having the same delimiters (shaded) as the outer comment. The following is an example of comments nested with different delimiters: `[**I am a comment [* I am nested*] I am still a comment**]`. The comment delimiters of the inner comment are regarded as text by the outer comment, not as comment delimiters. Therefore, the following is also valid: `[*I am a comment [** I am not nested, I am part of the comment*]`. The shaded text is not regarded as a comment delimiter because it is different from the opening and closing delimiters of the outer comment. However, the following example is <u>not</u> valid: `[*I am what ? [* I am not nested, I am not a comment*]`. The last comment delimiter is the closing delimiter of the shaded text; the outer ‘comment’ has no closing delimiter.

Care must be taken when using comments. For example, ‹`[****]`› indicates only an opening comment delimiter with four asterisks, not both an opening and a closing delimiter with two asterisks each. A way to achieve the desired effect (an empty comment) is: `[** **]`.

9. Variables

In ETAC, a “variable” is a *dictionary item*, consisting of a *dictionary keyword* (the variable’s “name”) and an associated *stack object* (the variable’s “value”) intended to be different at various times during an *ETAC session*. The variable’s value is modified by replacing that *stack object*. A variable name and its value is initially <u>allocated</u> to a *dictionary* and initialised at program run-time, not at program design-time. The value of a variable is typically assigned (replaced) by a *stack object* existing on the *object stack*. The same variable can have any type of value at any time within *ETAC code* as desired by the ETAC programmer. More than one variable of the same name can exist in a *dictionary*. The first-found variable of a particular name on the *dictionary stack* masks other variables of the same name.

In ETAC, the concept of a variable is merely for the convenience of the programmer and is not a component of the language (the ETAC programming language can be understood without that concept). A variable is merely a *dictionary item*, but a variable’s name is expressed as a *command* or *operator*, or sometimes, a string in *ETAC text script*.

9.1 Allocating Variables

Variables must be allocated to a *dictionary* before they can be used. A variable can be allocated only to the topmost *dictionary* on the *dictionary stack*, either directly or indirectly. When a variable is allocated, it is also initialised with a value. The following examples show how to allocate a variable directly to the topmost *dictionary* whether or not that variable already exists in any *dictionary* on the *dictionary stack*.

```
(1) MyVar :- 10; [* Allocates an integer. *]
(2) MyVar :- "I am all string"; [* Allocates a string. *]
(3) MyVar :- (5 + 3.7); [* Allocates the result of an expression. *]
(4) MyVar :- MyVar2; [* Allocates the value of another variable. *]
(5) 99.9; MyVar :-; [* Yes, you can do this (creates and initialises MyVar with 99.9). *]
(6) 99.9; :-; MyVar; [* !!!But you cannot do this. *]
(7) MyFnt :- fnt:(...){...}; [* Allocates a function. *]
(8) MyProc :- {...}; [* Allocates a procedure. *] ♦
```

If the variable name exists in a string, then the variable is allocated as in the following examples.

```
(1) new_dict_item "MyVar" 10;
(2) new_dict_item "MyVar" "I am all string";
(3) new_dict_item ("My" + "Var") (5 + 3.7);
(4) new_dict_item MyStrVar MyVar2; [* Assume that MyStrVar contains a string. *]
(5) 99.9; "MyVar"; new_dict_item;
(6) 99.9; new_dict_item "MyVar";
(7) new_dict_item "MyFnt" fnt:(...){...};
(8) new_dict_item "MyProc" {...}; ♦
```

As can be seen in the illustrations above, the ‹:-› symbol and the **new_dict_item** *command* accomplish the same task — that of allocating a new variable to the topmost *dictionary*. However, the left argument of ‹:-› must be an explicit variable name; it cannot be a quoted string.

A variable is indirectly (automatically) allocated if an attempt is made to assign (rather than allocate) to a non-existing variable.

9.2 Assigning Variables

The value of an existing variable can be replaced by assigning a *stack object* to that variable. If the variable does not exist anywhere in a *dictionary* on the *dictionary stack*, then the variable is automatically allocated to the topmost *dictionary* on the *dictionary stack* before being assigned. The following examples illustrate how to assign a variable.

```
(1) MyVar := 10; [* Assigns an integer. *]
(2) MyVar := "I am all string"; [* Assigns a string. *]
(3) MyVar := (5 + 3.7); [* Assigns the result of an expression. *]
(4) MyVar := MyVar2; [* Assigns the value of another variable. *]
(5) 99.9; MyVar :=; [* Yes, you can do this (assigns 99.9 to MyVar). *]
(6) 99.9; :=; MyVar; [* !!!But you cannot do this. *]
(7) MyFnt := fnt:(...){...}; [* Assigns a function (not usually done). *]
(8) MyProc := {...}; [* Assigns a procedure (not usually done). *] ♦
```

If the variable name exists in a string, then the variable is assigned as in the following examples.

```
(1) asn_dict_item "MyVar" 10;
(2) asn_dict_item "MyVar" "I am all string";
```

```
(3) asn_dict_item ("My" + "Var") (5 + 3.7);
(4) asn_dict_item MyStrVar MyVar2; [* Assume that MyStrVar contains a string. *]
(5) 99.9; "MyVar"; asn_dict_item;
(6) 99.9; asn_dict_item "MyVar";
(7) asn_dict_item "MyFnt" fnt:(...){...};
(8) asn_dict_item "MyProc" {...}; ♦
```

As can be seen in the illustrations above, the ‹:=› symbol and the **asn_dict_item** *command* accomplish the same task — that of assigning a value to an existing variable (or allocating a new variable to the topmost *dictionary* if the variable is nonexistent). However, the left argument of ‹:=› must be an explicit variable name; it cannot be a quoted string.

9.3 Retrieving the Value of a Variable

A *command* effectively contains the name of a variable. If a variable contains an integer, decimal, string, *sequence*, memory object, mark object, *ETAC function*, *data object*, or null, then the presence of the variable name (ie: the *command*) in *ETAC text script* pushes the value of the variable onto the *object stack*. If the value is a *dictionary*, then that value is pushed onto the *dictionary stack*. For other values, an appropriate action is performed. The following example illustrates the idea.

```
Var1 := "Tin "; Var2 := 10; [* Assignments. *]
Var := (Var1 + Var2); [* Combines the retrieved values of Var1 and Var2. *] ♦
```

In the last line, the value of `Var2` is pushed onto the *object stack*, followed by the value of `Var1`. The two values are combined via the '+' *operator* leaving a string (`"Tin 10"`) on the *object stack*, which is then assigned to `Var`.

9.4 Global and Local Variables

In ETAC, all variables are global because they are merely *dictionary* items existing in a *dictionary* on the *dictionary stack*. However, a *dictionary* can be made to exist temporarily only during the time that an *ETAC function* or *procedure* is executing. Such a *dictionary* is regarded as a *local dictionary*, and the variables allocated in a *local dictionary* will be destroyed along with the destruction of that *local dictionary*. The effect is that the variables in a *local dictionary* simulate (temporary) local variables.

10. Object Types

Stack objects are created by the presence of appropriate tokens in *ETAC text script*. When the *ETAC interpreter* encounters a token, it internally creates a *stack object* corresponding to that token, then performs a predetermined action with respect to that *stack object* depending on its type.

The following sections describe some of the common data types in the ETAC programming language.

10.1 Numbers and Booleans

There are two forms of numbers: (**1**) integers and (**2**) decimal numbers. There is no distinct boolean type, but integer values can be used as logical or binary *boolean values*. **true** and **false** are intrinsic boolean constants.

```
(1) 2145 [* Integer. *]
(2) -57 [* Integer. *]
(3) 23.0 [* Decimal. *]
(4) 58e-3 [* Decimal. *]
(5) 0x35A2C49F [* Binary boolean. *]
(5) false [* Logical boolean (same as 0). *]
(6) true [* Logical boolean (same as –1). *] ♦
```

Each token presented above represents an integer *stack object*.

10.2 Characters and Strings

There is no distinction between a string of characters and a single character in ETAC. A single character is a string of one character. Strings can be modified. The same string cannot be shared among different variables. A string token is delimited by double quote characters; single quote characters are not recognised as string delimiters. The first character of a string is at offset zero.

```
(1) ""  [* Empty string. *]
(2) "Hello string"  [* String. *]
(3) "c"  [* A character (same as a string). *] ♦
```

Each token presented above represents a string *stack object*.

Note that a string can contain single-byte characters only from the Western European **Windows**® code page 1252. Unicode strings are not supported.

A string can be modified in various ways, and its length (number of characters) can be obtained. This is illustrated in the following examples.

```
(1) | "Hello-string" |  [* String length (12). Whitespaces are necessary here. *]
(2) StrLen := |(Str + "my string")|;  [* String length. *]
(3) void StrLen := str_len "string";  [* String length (6). *]
(4) Str#7  [* Accessing a string character (at offset 7). *]
(5) Off := 3; Char := Str#Off;  [* Accessing a string character (at offset 3). *]
(6) Str#(1 + 3) := "string";  [* Replacing a string character (at offset 4) with a string. *]
(7) Str#* := "Me last";  [* Replacing the last string character with a string. *]
(8) Str := put_str 9 3 ReplStr Str;  [* Replacing 3 characters of a string (at offset 9). *]
(9) Str := ins_str 9 "C" Str;  [* Inserting one character (at offset 9). *]
(10) Str#2 := "";  [* Deleting one character (at offset 2). *]
(11) Str += "string";  [* Appending to a string. *] ♦
```

Examples **(1)** , **(2)**, and **(3)** illustrate how to obtain the length of a string. Example **(4)** illustrates the typical way to index a string. Example **(5)** illustrates that the index position can be an expression that returns an offset. Examples **(6)** to **(8)** illustrate how string characters in a string (`Str`) are replaced. Example **(9)** illustrates how a string is inserted into another string (`Str`). Example **(10)** illustrates how a string character can be deleted. Example **(11)** illustrates how a string can be appended to another.

10.3 Memory Objects

Memory *stack objects* represent arbitrary binary data, typically the contents of a file. A memory *stack object* can be created pre-initialised with data. The data can be accessed and modified directly, and text data can be extracted into a string. The size (the number of bytes of usable data) of a given memory *stack object* is variable.

The following examples illustrate how to create a memory *stack object* and modify and extract the usable data within it.

```
(1) &0h3F049A25Bd804CfD58  [* Initialised with nine bytes. *]
(2) &0  [* Memory block size is 50kB, usable data size is zero bytes. *]
(3) &20  [* Memory block size is 20 bytes, usable data size is zero bytes. *]
(4) Var := &0h7A203B4899; void Val := peekb 4 Var;  [* Data accessed. *]
(5) Var := &0h7A203B4899; void pokes 2 30 Var;  [* Data modified. *]
(6) Var := &0h737472696E67; Str := mem_to_str Var;  [* Extraction to string. *] ♦
```

Example **(1)** illustrates a memory *stack object* being created and initialised with the specified bytes of usable data, then pushed onto the *object stack*. Examples **(2)** and **(3)** illustrate a default memory *stack object* being created with 50,000 bytes and with 20 bytes, respectively, but with zero usable data size.

Examples **(4)**, **(5)**, and **(6)** illustrate a memory *stack object* being created and initialised with the specified bytes, then assigned to the variable `Var`. For example **(4)**, the byte at offset 4 is extracted from the memory *stack object* (in `Var`) and assigned to `Val`. The value of `Val` will be 99_H. For example **(5)**, the two bytes (short integer) at offset 2 in the memory *stack object* are replaced with the number 30 ($1E00_H$). For example **(6)**, the contents of the memory *stack object* is converted to a string then assigned to `Str` (the contents is: `string`).

Two or more memory *stack objects* can be concatenated. A string can be concatenated to, or inserted at the front of, a memory *stack object*. The following examples illustrate the idea.

```
(1) Mem := (&0h7A203B4899 + &0h737472696E67); [* Memory concatenation. *]
(2) void add2 Mem1 &0h737472696E67; [* Memory concatenated to Mem1. *]
(3) Mem += "string here"; [* String concatenated to a memory stack object Mem. *]
(4) Mem := add2 "string here" &0h737472696E67; [* String insertion at front. *]
(5) (+ Mem1 "string here" Mem2) [* String and memory concatenation to Mem1. *]
(6) Mem := (+ &0 Mem1 Mem2 Mem3 Mem4); [* Multiple memory concatenation. *] ♦
```

Examples **(1)**, **(2)**, and **(6)** illustrate how the usable data in memory *stack objects* can be concatenated into the first memory *stack object*. Examples **(3)** and **(4)** illustrate, respectively, how a string can be concatenated to the end, and inserted at the front, of a memory *stack object*. Example **(5)** illustrates how a string then a memory *stack object* can be concatenated to the end of the first memory *stack object* (`Mem1`).

A memory *stack object* can be expanded with zero-initialised ("null") bytes of usable data, or the usable data contracted. In general, adding an integer to a memory *stack object* expands the usable data by the specified number of null bytes; subtracting an integer from a memory *stack object* truncates the usable data by the specified number of bytes. The integer is interpreted as a positive number. The position of the integer relative to the memory *stack object* determines whether the said operation is performed at the front or the end of the usable data.

The following examples illustrate how the size of usable data in a memory *stack object* can be modified.

```
(1) Mem := (&0h7A203B4899 + 20); [* Memory expanded at end. *]
(2) void add2 Mem1 20; [* Memory Mem1 expanded at end. *]
(3) Mem := (Mem - 54); [* Memory truncated at end. *]
(4) Mem := add2 33 &0h737472696E67; [* Memory expanded at front. *]
(5) (10 - Mem) [* Memory truncated at front. *]
(6) Mem := sub2 29 Mem; [* Memory truncated at front. *] ♦
```

Examples **(1)** and **(2)** illustrate a memory *stack object* expanded with 20 null bytes of usable data at the end of the existing usable data. Example **(3)** illustrates the usable data of a memory *stack object* being truncated at the end by 54 bytes. Example **(4)** illustrates a memory *stack object* expanded with 33 null bytes at the front of the usable data. Examples **(5)** and **(6)** illustrate a memory *stack object* being truncated by the specified number bytes at the front of the usable data.

The size of the usable data in a memory *stack object* can be obtained.

```
(1) void MSize := size Mem; [* Size of a memory stack object. *] ♦
```

Example **(1)** obtains the size (`MSize`) of the usable data in a memory *stack object* (`Mem`).

10.4 Sequences

The ETAC programming language defines *sequences* of *stack objects* of any type (including *sequences*) rather than arrays. *Sequences* are not primary objects and must be built via *sequence expressions*

before they can be used. The index number of the first element of a *sequence* is one. The number of elements in a given *sequence* is variable and can be changed by *ETAC code* at any time.

A *sequence* can be built at the build-time of its parent *procedure*, or at the call-time of its parent *procedure*. A *sequence expression* builds the *sequence* elements on the *object stack* before those element are converted into an actual *sequence*.

The following examples illustrate how a *sequence* can be built.

```
(1) [3, "Hello string", 4.5, {...}] [* Regular sequence. *]
(2) [(3 + 4), [21, true], MyVal] [* Sequence containing a subsequence. *]
(3) [1, 2, 3, 4, 5, 6] [* Regular sequence containing numbers representing indices. *]
(4) [3 2 1, 5 4, 6] [* Funny way to create the sequence at (3). *]
(5) end_seq 6 5 4 3 2 1 start_seq; [* Another way to create the sequence at (3). *]
(6) start_seq; 1; 2; 3; 4; 5; 6; end_seq; [* ditto *]
(7) [] [* Empty sequence. *] ♦
```

Each group of tokens presented above represents a *sequence*. In example **(2)**, the *sequence* is built each time its parent *procedure* or *ETAC function* is called because the variable `MyVal` needs to be *evaluated* for each call. Likewise, for examples **(5)** and **(6)**, the *sequence* is created using *commands*, and so it is created each time the parent *procedure* or *ETAC function* is called. The examples **(3)**, **(4)**, **(5)**, and **(6)** create the same *sequence* where each element corresponds to its index. Example **(4)** utilises *sequential reverse-flow activation* using a comma as the *marker token*. (Actually, all *sequence expressions* using square brackets utilise *sequential reverse-flow activation*.)

A *sequence* can be modified in various ways, and its size (number of elements) can be obtained. This is illustrated in the following examples.

```
(1) |[3, "Hello string", 4.5, {...}]| [* Sequence size (4). *]
(2) SeqSize := |(Seq + [5, 8, 20])|; [* Sequence size. *]
(3) void SeqSize := size [1, 2, 3]; [* Sequence size (3). *]
(4) Seq%[3, 7] [* Accessing a subsequence element (at index 3,7). *]
(5) Idx := [3]; Elm := Seq%Idx; [* Accessing a sequence element (at index 3). *]
(6) Seq%[4] := 21.876; [* Replacing a sequence element (at index 4). *]
(7) Seq%[] := "Me last"; [* Replacing the last sequence element. *]
(8) Seq<%[9] := [5, 8, 20]; [* Inserting one sequence (at index 9). *]
(9) Seq<%%[9] := [5, 8, 20]; [* Inserting the (3) elements of a sequence (at index 9). *]
(10) Seq +:= [5, 8.3, 20]; [* Appending one sequence. *]
(11) Seq ++:= [5, [8, "string"], 20]; [* Appending the (3) elements of a sequence. *] ♦
```

Examples **(1)** , **(2)**, and **(3)** illustrate how to obtain the size of a *sequence*. Example **(4)** illustrates the typical way to index a *sequence* (in this case a *subsequence* is accessed). Example **(5)** illustrates that the index position can be an expression that returns an index *sequence*. Examples **(6)** and **(7)** illustrate how a *sequence* element is replaced. Examples **(8)** and **(9)** illustrate how elements are inserted into a *sequence*. Examples **(10)** and **(11)** illustrate how elements are appended to a *sequence*.

10.5 Dictionaries

Dictionaries, as such, typically exist on the *dictionary stack*, and are created at the top of the *dictionary stack* with a capacity to contain a specified number of *dictionary items*. The number of *dictionary items* in a *dictionary* is not limited to the initial capacity. Any number of additional *dictionary items* can be added to the *dictionary* at any time. *Data dictionaries* are automatically pushed onto and popped off the *dictionary stack* whenever the *members* of a *data object* are accessed.

```
(1) new_dict "My Dictionary" 0; [* Allocation of initial default size dictionary. *]
(2) new_dict "" 30; [* Allocation of initial specified size (30 items) dictionary. *] ♦
```

The two examples above illustrate how to create an empty *dictionary* with the specified name and initial size.

Dictionaries on the *dictionary stack* are globally accessible, but can be created to exist temporally as *local dictionaries* (typically for use with *procedures* and *ETAC functions*). While a *local dictionary* exists temporarily, it can be accessed globally. An *ETAC function* automatically creates a *local dictionary* before the function's body is executed, and automatically deletes that *local dictionary* after the function body's execution has ended.

The following examples illustrate how a *local dictionary* can be created and destroyed explicitly and implicitly.

```
(1) MyProc :- [* Allocation of local dictionary in a procedure (if required). *]
    {
    start_local; [* Creates local dictionary. *]
       ...
    end_local; [* Destroys local dictionary. *]
    };
(2) MyFnt :- fnt:(...) [* ETAC function automatically creates temporary local dictionary. *]
    {
    start_local; [* Automatically creates local dictionary here. *]
       ...
    end_local; [* Automatically destroys local dictionary here. *]
    }; ♦
```

Example **(1)** illustrates how to explicitly create and destroy a *local dictionary* in a *procedure*. Example **(2)** illustrates that a *local dictionary* is temporarily created automatically for an *ETAC function* (**start_local** and **end_local** are not inserted by the programmer in an *ETAC function*).

The topmost *dictionary* on the *dictionary stack* can be assigned to a variable as illustrated below. The **defr** *command* causes the *dictionary* to be pushed onto the *object stack* when the variable is later presented, rather than being pushed onto the *dictionary stack* by default.

```
MyDict := defr pull_dict; ♦
```

Two or more *dictionaries* can be combined. Note that combining *dictionaries* is rarely done.

```
(1) pull_dict ++:= Dict1; [* Conjoining dictionary items to the topmost dictionary. *]
(2) (++ Dict1 Dict2 Dict3); [* Conjoining dictionary items of many dictionaries. *] ♦
```

Example **(1)** illustrates how the topmost *dictionary* on the *dictionary stack* can have *dictionary items* added at the top from another *dictionary* (`Dict1`). Example **(2)** illustrates how the *dictionary items* of a number of *dictionaries* can be combined to the first *dictionary* (`Dict1`).

10.6 Data Objects

A *data object* is logically a set of named *stack objects*, and is emulated by the *ETAC interpreter* as a *data dictionary*. The *dictionary items* of the *data dictionary* constitute the *members* of the *data object*. A *data object* has no intrinsic identification (it is "anonymous") but is typically allocated or assigned to an *ETAC variable*, which is used as the *data object's* identification. However, a *data object* can be given a name stored in a *member variable* of that *data object*. Such named *data objects* can be used stand-alone, as variant records, or as object-oriented classes and their instances. *Data objects* are created by the **data:** *ETAC statement* as illustrated in the examples below.

```
(1) MyData :- data: [* Data object creation. *]
    {
       mdMember1 :- 10;
       mdMember2 :- [3, "string", 21.9];
       mdProc :- {...};
       mdFnt :- fnt:(...) {...};
    };
(2) [* Named data object used for object-oriented programming. *]
    @DefData("MyDataObj") ["BaseObj1", "BaseObj2", "BaseObj3"]
    {
       @Name :- "MyDataObj"; [* Automatically allocated. *]
       [* Member allocations and assignments. *]
    };
(3) DataObj := @Data("MyDataObj"); [* Named data object access. *]
(4) MyData1 :- @NewData("MyDataObj"); [* An instance of a named data object. *] ♦
```

Example **(1)** illustrates how a regular *data object* can be created. The *members* are allocated to the *data object's data dictionary*. Example **(2)** illustrates creating a named *data object* derived from three base *data objects* (which are also named *data objects* in this case). Example **(3)** illustrates how to access a named *data object*. Example **(4)** illustrates how to create a new instance of a named *data object*.

Specified *members* of a *data object* can be made private (or "exclusive") to specified *function members* of that *data object* as illustrated in the following example.

```
(1) MyData :- data:
    {
       mdFnt1 :- fnt:(...) {...}; [* mdFnt1 can access exclusive members. *]
       mdFnt2 :- fnt:(...) {...}; [* mdFnt2 can access exclusive members. *]
       mdFnt :- fnt:(...) {...}; [* mdFnt cannot access exclusive members. *]

    exclusive: ["mdFnt1", "mdFnt2"]
    {
       [* Exclusive members allocated here.
          These members can only be accessed by mdFnt1 and mdFnt2. *]
    };
    }; ♦
```

Example **(1)** illustrates how *function members* can be made to access *members* exclusively.

Two *data objects* can be combined as illustrated in the following examples (`MyData` and `MyOwnData` contain a *data object*).

```
(1) MyData ++:= data:{...}; [* Conjoining data object members. *]
(2) MyData ++:= MyOwnData; [* Conjoining data object members. *] ♦
```

A *member* of a *data object* can be selected and its value modified as illustrated in the following examples (`MyData` contains a *data object*).

```
(1) MyData.mdMember [* Accessing a data object member. *]
(2) MyData.mdMember := 10; [* Modifying a data object member. *]
```

```
(3) MyData.
    {
       mdMemb1 :- 25.9; [* Allocates a new member. *]
       mdMemb2 := ("this " + "string"); [* Modifies an existing member. *]
       write_con mdMemb2; [* Does something with a member. *]
    }; ♦
```

Example **(1)** access a *member* of a *data object*. Examples **(2)** and **(3)** modify *data object members*. For example (3), the topmost *dictionary* on the *dictionary stack* is the *data dictionary* of the *data object* being accessed, but only while the code within the braces is *active*.

10.7 Procedures

Procedures (including the body of *ETAC functions*) are not primary objects and must be built via *procedure expressions* before they can be used. A *procedure* has no intrinsic identification (it is "anonymous") but is typically allocated or assigned to an *ETAC variable*, which is used as the *procedure's* identification. A *procedure* can be modified in various ways, but such modifications are rarely done. A *procedure* is effectively a *sequence* that, by default, gets executed when *activated*. Operations that can be done to a *sequence* can also be done to a *procedure* after the *procedure* is pushed onto the *object stack*.

A *procedure* can be built at the build-time of its parent *procedure*, or at the call-time of its parent *procedure*. A *procedure expression* builds the *procedure* elements onto the *object stack* before those element are converted into an actual *procedure*.

The following examples illustrate how a *procedure* can be built.

```
(1) Proc :- {A := 3; B := 4; C := (A + B);}; [* Procedure creation at build-time. *]
(2) start_proc; [* Procedure creation at call-time. *]
       A `:= 3;  B `:= 4;  C `:= `( `A `+ `B `);
    end_proc; Proc :-;
(3) {; [* Procedure creation at call-time. *]
       A `:= 3;  B `:= 4;  C `:= `( `A `+ `B `);
    }; Proc :-;
(4) A := ...; B := ...;
    {; [* Unique procedure creation at call-time. *]
       C `:= `( A `+ B `); [* A and B get evaluated before procedure creation. *]
    }; Proc :-; ♦
```

Example **(1)** illustrates a typically created *procedure*. The braces are actual *commands* that begin ({) and end (}) the construction of the *procedure* on the *object stack*. Once built, the *procedure* is typically allocated to a variable by the programmer. This *procedure* is built when its parent *procedure* or *ETAC function* is built. Examples **(2)**, **(3)**, and **(4)** illustrate how a *procedure* can be built anew each time its parent *procedure* or *ETAC function* is called. Note the use of the acute accent character (`). This is required so that the *comops* within the *procedure* body get pushed onto the *object stack* as elements rather than get executed. When the *procedure* gets executed later via Proc, then those *comops* will be executed as required. Example **(3)** is an alternative way of creating the *procedure* of example (2). In example **(4)**, the variables A and B get *evaluated* each time before the *procedure* gets created anew (note the absence of the acute accent character before those two variables).

10.8 Functions

ETAC functions are emulated, not primary objects, and must be defined via function definitions before they can be used. All *ETAC functions* are "anonymous", that is to say that they have no intrinsic identification. *ETAC functions* can be *copied*, *replicated*, or *duplicated*, and allocated or assigned to any number of other variables. Mostly, however, they are only allocated to one variable by which they are called.

When called, an *ETAC function* will first create a *local dictionary*, and allocate the parameters (if any) to that *local dictionary*. The parameters are initialised with the *stack objects* existing on the *object stack* in the same order that the parameters are specified; the first parameter specified on the left will be initialised with the topmost *stack object*, the second parameter will be initialised with the second-top *stack object*, and so on. The function body is then *activated*. Finally, the *local dictionary* is destroyed.

The example below illustrates an *ETAC function* being defined with three parameters, and allocated to a variable.

```
MyFnt :- fnt:(pPar1 pPar2 pPar3)
{
    ... [* Programmer defined function body here. *]
}; ♦
```

The *ETAC function* above is logically equivalent to the following (the pale blue text is created internally by the *ETAC interpreter*).

```
MyFnt :- fnt:(pPar1 pPar2 pPar3)
{
start_local; [* Automatically creates local dictionary here. *]
pPar1 :-; pPar2 :-; pPar3 :-; [* Automatically allocates parameters here. *]

    ... [* Programmer defined function body here. *]

end_local; [* Automatically destroys local dictionary here. *]
}; ♦
```

A *function command* must be called using a pair of parentheses. There must be no whitespace between the left parenthesis and the *function command*. An *ETAC function* can return as many *stack objects* on any *TAC stack* (but typically the *object stack*) as desired. The first four examples below illustrate a number of ways that the *ETAC function* defined above can be called.

```
(1) MyFnt(21.7 "text" 99); [* Typical call. *]
(2) MyFnt(21.7 "text") 99;
(3) MyFnt() 21.7 "text" 99;
(4) 99; MyFnt(21.7) "text"; [* Exotic call. *]
(5) F(do I to 3 {I;}); [* Equivalent to: F(3 2 1). *]
(6) A := B := C := F() [* F() returns three stack objects, which are assigned right to left. *]♦
```

Note that the arguments of a function call need not all be within the parentheses, but they do need to exist in the appropriate order on the *object stack*.

10.9 Other Object Types

A mark *stack object* is a unique type of *stack object* that does not contain useful data. There are eight mark *stack objects* (numbered 0 to 7) for use by the ETAC programmer for their own purposes. The *ETAC interpreter* uses the mark 0 *stack object*.

A null *stack object* is a unique type of *stack object* that does not contain any useful data. There is only one type of null *stack object*. A null *stack object* is typically used to initialise a newly allocated *ETAC variable* to indicate that it does not have a legitimate value.

The following examples illustrate the use of the null and mark *stack objects*.

```
(1) MyVar :- ?; [* Allocates a null stack object (?) to a variable. *]
(2) MyVar := !1; [* Assigns a mark 1 stack object (!1) to a variable. *]
(3) Myvar :- !7; [* Allocates a mark 7 stack object (!7) to a variable. *] ♦
```

An EXE *stack object* identifies code defined in another programming language (typically the C++ programming language).

11. Operator Expressions

In the ETAC programming language, *operator expressions* explicitly require parentheses. There are no implied parentheses around the *operators* and their arguments; specifically, multiplication and division do not have automatic precedence over addition and subtraction. The parentheses, *operator*, and operands together form an *operator expression*, which can be nested. The *operator* can be in any position within the parentheses (*operator expressions* therefore use "non-fix" notation). One or more whitespaces is required between an *operator* and its operands and among the operands themselves. The examples below illustrate some *operator expressions*.

```
(1) (3 + 2) [* Simple expression returns 5. *]
(2) (3 &add 2) [* Same as above. *]
(3) (* 5 8 2.6 9) [* Multiple arguments; operator at any position. Returns 936.0 . *]
(4) end_op 5 8 2.6 &mult 9 start_op; [* Same as above but unusual. *]
(5) start_op; 9; *; 2.6; 8; 5; end_op; [* Same as above but very unusual. *]
(6) (> Num1 Num2 Num3) [* Same as ((Num1 > Num2) &and (Num2 > Num3)). *]
(7) ((4 * 5) + 8) [* Nested expression (parentheses required). *]
(8) (+ do_for {} 1 1 10) [* Sums integers from 1 to 10. Returns 55. *] ♦
```

Note that *operator expressions* within parentheses are processed from right to left, and exactly one *operator* must exist within a pair of matching parentheses at the top level. Note also that the parentheses are actually *commands* not just punctuation.

The *operators* are: +, &add, -, &sub, *, &mult, /, &div, ++, &combine, ^, &power, =, &equal, !=, &n_equal, &and, &or, >, &great, >=, &great_eq, <, &less, <=, &less_eq. Some pairs of the *operators* are equivalent.

Operators can also be defined by the ETAC programmer via *procedures*. The easiest way to define an *operator* is to define a *procedure* that takes two arguments and returns a result on the *object stack*, then using the **mk_op** *command* to create the *operator*. The *operator* can then be used in an *operator expression* with two or more compatible arguments. The example below illustrates.

```
MyOpr :- mk_op {...}; [* The procedure takes two arguments returning a result. *]
Res := (&MyOpr Arg1 Arg2 Arg3 Arg4 Arg5); [* MyOpr used with many arguments. *] ♦
```

12. Flow Control

The ETAC programming language supports a number of iteration *commands* and a single universal iteration *ETAC statement*. There are also *commands* and *ETAC statements* that exit iteration and other *procedures*. There are two types of conditional *commands*, and two types of corresponding conditional *ETAC statements*. There are no "go to" statements in ETAC.

12.1 Conditionals

Conditional code controls the flow of *activation* based on a condition but without iteration.

12.1.1 If-then ETAC Statement

There is only one multi-conditional *ETAC statement* as illustrated in the following examples.

```
(1) if ( X > 3 ) then {[* true code *]} endif
(2) if Success then {[* true code *]} [else {[* false code *]}] endif
(3) if C1 then {...} [C2 then {...} ... Cn then {...}] [else {...}] endif ♦
```

Example **(3)** presents the full form of the conditional *ETAC statement*. `C1` to `Cn` (each of which can be a variable, *operator expression*, *procedure expression*, and more) must leave a *boolean value* on the *object stack*.

12.1.2 If-then ETAC Function

The conditional *ETAC function* is typically used when a *stack object* is required to be pushed onto the *object stack* based on a condition as illustrated in the following example.

```
(1) @IfElse(C1 {[* true code *]} {[* false code *]}) [* Typically returns a stack object. *]
♦
```

12.1.3 If-then Commands

There are three conditional *commands* as illustrated in the following examples.

```
(1) if_do {[* true code *]} ( X > 3 )
(2) Success; if_do {[* true code *]};
(3) if_else {[* false code *]} {[* true code *]} Success
(4) if_then [C1, {...}[, C2, {...}, ..., Cn {...}][, {...}]]
(5) X := [C1, {...}, C2, {...}]; X ++:= [C3, {...}, {...}]; if_then X; ♦
```

Example **(4)** presents the full form of the **`if_then`** multi-conditional *command*. `C1` to `Cn` (each of which can be a variable, *operator expression*, *procedure expression*, and more) must each leave a *boolean value* on the *object stack*. The *command* requires a *sequence* as its only argument. The *sequence* contains condition and code pairs; the optional last element is the "else" code. Since the argument is a *sequence*, that *sequence* can be altered at run-time, as shown in example **(5)**, allowing the structure of the **`if_then`** *command* to be dynamic.

12.1.4 Choice ETAC Statement

There is only one choice *ETAC statement*. The choice statement compares (indicated by the *operator* after the first argument) the value of a *stack object* (via the argument after "`when`") with a number of possible values (via the arguments before "`then`") and *activates* the *procedure* associated with the possible value (the *procedure* after "`then`") when the first one of the comparisons *evaluates* to `true`. If no comparison *evaluates* to `true` then an alternative *procedure* (after "`else`") is optionally *activated*.

The following examples illustrate the choice *ETAC statement*.

```
(1) when X = 3 then {...} [Val then {...} ... (A + B) then {...}] endwhen
(2) when (X - Y) >= {...} then {...} [... 5 then {...} ][else {...}] endwhen
(3) when V op X1 then {...} [X2 then {...} ... Xn then {...}] [else {...}] endwhen ♦
```

Example **(3)** presents the full form of the choice *ETAC statement*. *op* is one of the following: ‹=› and ‹is› (these two are the same), ‹!=›, ‹<›, ‹>›, ‹<=›, ‹>=›.

12.1.5 Choice Commands

There are three choice *commands*. These operate in a similar fashion as the choice *ETAC statement* but without the "`then`", "`else`", and "`endwhen`" keywords.

The following examples illustrate the choice *commands*.

```
(1) switch X `= [3, {...}[, Val, {...}, ..., (A + B), {...}][, {...}]]
(2) switch (X - Y) `ge [{...}, {...}[, ... 5, {...}][, {...}]]
(3) switch ... `&MyOp [...]
(4) switch V `op|cmd [X1, {...}[, X2, {...}, ... Xn, {...}][, {...}]]
(5) S := [X1, {...}[, X2, {...}]]; S ++:= [X3, {...}]; C := {...}; switch V C X;
```

```
(6) switch_eq [3, {...}[, Val, {...}, ..., (A + B), {...}][, {...}]] X
(7) select [:#TAC_INT:, {...}, :#TAC_MEM:, {...}, ..., [, {...}]] Var ♦
```

Example **(4)** presents the full form of the **switch** *command*. X1 to X*n* must each leave a *stack object* on the *object stack* for comparison with the *evaluation* of V. *op* is a comparison *operator* that leaves a *boolean value* on the *object stack*, and *cmd* is a *command* that takes two arguments and leaves a *boolean value* on the *object stack*. The *command* requires a *sequence* as its third argument. The *sequence* contains value and code pairs; the optional last element is the "else" code. Since the third argument is a *sequence*, that *sequence* can be altered at run-time, as shown in example **(5)**, allowing the structure of the **switch** *command* to be dynamic. Example **(6)** illustrates the **switch_eq** *command*, which is a **switch** *command* with an inbuilt "equals" (=) *operator* (this example operates the same manner as does example (1)). Example **(7)** illustrates the **select** *command*, which requires a *sequence* as the first argument. The *sequence* consists of a series of pairs of elements. The first element of a pair *evaluates* to a *stack object* type (or *sequence* of such) and the second element of a pair is typically a *procedure* requiring the value of Var as its argument. The type of *stack object* pushed onto the *object stack* by Var is compared with the *stack object* types indicated in the *sequence*. If that type is equal to the first found type in the *sequence*, then the *procedure* in the pair is *activated* with the argument indicated by Var. If no type is found in the *sequence*, then the optional last element in the *sequence* is *activated* with the said argument. Note that the structure of all three choice *commands* can be altered at run-time.

12.2 Iterations

Iteration code repeats the flow of *activation* during or until an explicit or implicit condition is satisfied.

12.2.1 ETAC Iteration Statement

There is only one universal iteration *ETAC statement*. The iteration statement *activates* a *procedure* a specified number of times based on a counter or one or more specified conditions. Iterations terminate when the first one of the conditions satisfy its termination criterion, or they can be terminated explicitly.

The following examples illustrate the iteration *ETAC statement*.

```
(1) do {...}  [* Exactly one iteration. *]
(2) do repeat [Count] {...}  [* Unlimited or fixed number of iterations. *]
(3) do Idx [from Start] [to End] [step Step] {...}  [* Indexed iterations. *]
(4) do with Elm of Seq {...}  [* Iterations with a sequence element. *]
(5) do while Cond {...}  [* Iterations while a condition is satisfied. *]
(6) do Idx to |Seq| while (Seq%[Idx] > 10) {...}
(7) do Idx with Elm of Seq while not (Elm <= 10) {...}
(8) do [Idx [from Start] [to End] [step Step]] [with Elm of Seq] [while Cond]
{...}♦
```

Examples **(1)**, **(2)**, and **(8)** present the full form of the iteration *ETAC statement* (note that example (8) incorporates seven possible modes of iteration). Idx and Elm must be variables. Start, End, and Step can each be an explicit integer, variable, *operator expression*, *procedure expression*, and more. Seq typically *evaluates* to a *sequence*. Cond *evaluates* to a *boolean value* each time it is *activated*.

12.2.2 Command Iterations

There are six iteration *commands*. These operate in a similar fashion to the iteration *ETAC statement*, but the different iteration modes cannot be combined. There are also other *commands* that perform iterations implicitly.

The following examples illustrate the iteration *commands*.

```
(1) do_repeat {...} [* Unlimited number of iterations. *]
(2) do_loops {...} Count [* Fixed number of iterations. *]
(3) do_for {...} Step Start End [* Indexed iterations. *]
(4) do_with {...} Seq [* Iterations with a sequence element. *]
(5) do_while {...} Cond [* Iterations while a condition is satisfied. *]
(6) do_until {...} Cond [* Iterations until a condition is satisfied. *] ♦
```

`Count` must leave a non-negative integer *stack object* on the *object stack*. `Step`, `Start`, and `End` must leave an integer *stack object* on the *object stack*. `Seq` must leave a *sequence* or *procedure* on the *object stack*. `Cond` must leave a *procedure* or command *stack object* on the *object stack* that *evaluates* to a *boolean value* each time it is *activated*. In example **(5)**, `Cond` is *activated* prior to each iteration. In example **(6)**, the *procedure* is *activated* at least once; `Cond` is *activated* after each iteration.

12.3 Exiting Code Blocks

A 'code block' is a *procedure* whether or not it is part of a *command* or an *ETAC statement*. The flow of *activation* can 'break' (exit without completing) out of a *code block* either absolutely or conditionally.

The *break ETAC statements* are:

exit_do	Unconditionally breaks out of the logically immediate iteration *ETAC statement*.
donext	Unconditionally causes control to go to the end of the logically immediate iteration *ETAC statement* and continues with the next iteration (if there is one).
exitdo_if	Conditionally breaks out of the logically immediate iteration *ETAC statement*.
donext_if	Conditionally causes control to go to the end of the logically immediate iteration *ETAC statement* and continues with the next iteration (if there is one). ♦

The *break* ETAC *function commands* are:

@Error	Calls the handler of the most recent call to **@TrapError**.
@Exit	Breaks out of the logically immediate **@BreakOnExit** *command* unconditionally.
@Return	Breaks out of the logically immediate *ETAC function procedure* unconditionally.
@ExitIf	Breaks out of the logically immediate **@BreakOnExit** *command* conditionally.
@ReturnIf	Breaks out of the logically immediate *ETAC function procedure* conditionally. ♦

The *break commands* are:

break	Breaks out of the logically immediate **do_for**, **do_loops**, **do_repeat**, **do_until**, **do_while**, **do_with**, or **switch** *commands*. Also breaks out of the logically immediate iteration or choice *ETAC statement*.
end	Immediately ends the current *ETAC session*.
go_end	Ends the logically immediate *procedure* depending on a specified condition.
exit_err	Exits the current *ETAC session* with a *TAC error code* unless the *TAC error code* is trapped by another *command* before the *ETAC session* ends. ♦

13. Object-oriented Programming

Whilst the ETAC programming language is not intrinsically an object-oriented language, nevertheless, object-oriented programming can be emulated entirely in *ETAC code* by 'named' *data objects*.

Object-oriented programming, as emulated in the ETAC programming language, requires:

1. an 'archetype' *data object*,
2. the ability to derive additional *archetype data objects* from existing ones (*data object* 'derived' from 'base' *data object*),
3. data that is specific to each *data object derived* from the same *base data object* ('particular' *members*),
4. data that is common (shared) among all *data objects derived* from the same *base data object* ('common' *members*),
5. the ability for specified *ETAC functions* to have exclusive access to specified data,
6. the ability of *members* of a *derived data object* to mask the *members* of its *base data object*,
7. the ability of *function members* of a *base data object* to access data in its *derived data object*,
8. the ability of *function members* of a *derived data object* to access data in its *base data object*,
9. any number of instances (as *data objects*) to be created from any *archetype data object* ('instance' *data object*).

Named *data objects* emulate object-oriented capabilities via the **@Data**, **@DefData**, and **@NewData** *ETAC functions*. "Class inheritance" is achieved via **@DefData**, which can use named *data objects* or regular *data objects* as "base classes". **@DefData** is intended to create archetypes (which act as "class definitions") of named *data objects* for the *main ETAC session* (which includes all other *ETAC sessions*). A new instance of a named *data object* based on such an archetype is created via **@NewData**.

14. Data Input and Output

An ETAC program can receive input from the command line, and can also read from and write to files, the console, and dialog boxes. Dialog boxes containing controls can be created by the ETAC programmer via resource files to receive user input data and to show data. Native ETAC does not create graphics windows nor can it create graphics images of any kind. However a full interactive graphics system is intended to be included in AppETAC.dll in the future.

15. External Code Execution

ETAC code existing in a separate file or in a memory *stack object* can be called via the **exec_tac** *command*. External executable programs ('.exe') can be executed via the **exec_prog** and **exec_prog_sync** *commands*. Functions in an *external TAC library* can be called via the **@ImportLib**, or for more control, via the **load_lib** *commands*.

16. Derived Syntax

There are two conceptual forms of syntax for the ETAC language. The 'primary syntax' is the defined syntax as presented in the document ETACProgLang(Official).pdf. 'Derived syntax' is syntax that is derived from *primary syntax* as a logical consequence of it. For example, `‹[1, 2, 3, 4]›` is a *sequence* expressed in *primary syntax* form; `‹[do I to 4 {I;}]›` is the same *sequence* expressed in *derived syntax* form. The latter syntax is a natural consequence of the definition of a *sequence expression*. Similarly, `‹B := A := 1 2;›` is a *derived syntax* form of `‹A := 1; B := 2;›`, and `‹B := A := B A;›` (swapping the values of two variables) is a *derived syntax* form of `‹X := A; A := B; B := X;›`. *Derived syntax* is possible as a direct consequence of ETAC being a dictionary and stack based token activated programming language — it is not additional syntax that has been added to ETAC. The *ETAC interpreter* does not explicitly syntax check code written in *derived syntax* form. Only the *primary syntax* forms are explicitly defined in the ETAC language and syntax checked by the *ETAC interpreter*.

As a result of the capability of *derived syntax*, exotic statement structures can be created. However, exotic statements are generally confusing to read, and should be avoided if possible (unless you want to give yourself airs).

The following is an illustration of exotic use of *derived syntax* (that is best to be avoided).

```
[* An illustration of exotic use of derived syntax. *]
MyFnt :- fnt:(A B S1 S2 E1 E2 V1 V2)
{
Idx :- ?;

   [
   do Idx
      from
         when A is
            21.4 then {S1;}
            if ( B > 55 ) then {V1;} else {V2;} endif then {S2;}
            else {3}
         endwhen
      to
         if ( if (A = 5) then {10} else {20} endif >= 15 )
            then
            {E1;}

            ( B = 5 ) then
            {100}

            else
            {E2;}
         endif
   {Idx;}
   ]; [*RETURN*]
}; ♦
```

The code above would be better understood if written as follows.

```
[* An illustration of preferred use of derived syntax. *]
MyFnt :- fnt:(A B S1 S2 E1 E2 V1 V2)
{
Idx :- ?; X :- ?; Y :- ?;

   X := [* Some comments here. *]
      when A is
         21.4 then {S1;}
         @IfElse(( B > 55 ) {V1;} {V2;}) then {S2;}
         else {3}
      endwhen;

   Y := [* Some comments here. *]
      if ( @IfElse(( A = 5 ) {10} {20}) >= 15 )
         then
         {E1;}

         ( B = 5 )
         then
         {100}

         else
         {E2;}
      endif;

   [do Idx from X to Y {Idx;}]; [*RETURN*]
}; ♦
```

Another two illustrations of exotic *derived syntax* that should usually be avoided are shown below.

```
[* Looks like appending to a conditional statement. *]
if ( A = 10 ) then {Seq1;} else {Seq2;} endif +:= "what?";

[* The conditional statement should be separated from the iteration statement. *]
do while if C then {(D >= E);} else {F;} endif {[* Some code. *]}; ♦
```

Bibliography

The Official ETAC Programming Language copyright © Victor Vella (2019).

Glossary

A

activate

a) When referring to a *script token* that creates a *stack object*, the *script token* is converted to a *stack object* by the *TAC processor* and then the object's *nominal action* is performed.

b) When referring to a *script token* that does not create a *stack object*, an appropriate action is performed depending on the type of *script token*.

c) When referring to a *stack object*, the *stack object* is temporarily *copied* by the *TAC processor* and then the *copied* object's *current action* is performed.

active

The period of time during which a *script token* or *stack object* is performing an action after having been *activated*.

B

binary interpreter

Part of an *ETAC interpreter* that processes *TAC binary instructions*.

boolean value

An integer interpreted as consisting of 32 binary flags, or a 'true' (`-1`) or 'false' (`0`) value. The *true* value is represented by having all the 32 binary flags set (achieved by the value `-1` based on a two's complement representation of integers). The *false* value is represented by having all the 32 binary flags unset. A *boolean value* is typically assigned by a hexadecimal number if used as binary flags, or by the `true` or `false` *intrinsic commands* if used as a logical condition.

C

command

A *script token* having the syntax of a *comop identifier*. A *command* can be in *script form* (eg: ‹`FilePath`›, ‹`tac.var`›, ‹`#abc%03?`›, ‹`sub:`›, ‹`.xyz-3`›) or *instruction form* (eg: ‹`CMD:FilePath`›, ‹`CMD:tac.var`›, ‹`CMD:#abc%03?`›, ‹`CMD:sub:`›, ‹`CMD:.xyz-3`›).

comop

A *command* or *operator* (**com**mand **op**erator), or a *stack object* created by such a *command* or *operator*.

comop identifier

A consecutive sequence of displayable characters with the following restrictions. The sequence must **not**:

- begin with a digit or colon character,
- begin with an uppercase character and have a colon in fourth character position (eg: ‹`Abc:d`› is invalid),
- be in the form of an integer or decimal number (eg: ‹`23`›, ‹`+23`›, ‹`2.3`›, ‹`-2.3`›, ‹`+2.3e5`›, ‹`.3E+2`›, ‹`0.3`› are invalid),
- be ‹`+`›, ‹`-`›, ‹`*`›, ‹`/`›, ‹`^`›, ‹`=`›, ‹`!=`›, ‹`<`›, ‹`>`›, ‹`<=`›, ‹`>=`›, ‹`++`›, ‹`?`›,
- contain whitespaces or the characters ‹`'`›, ‹`"`›, ‹`,`›, ‹`;`›, ‹`[`›, ‹`]`›, ‹`{`›, ‹`}`›, ‹`(`›, ‹`)`›.

A *comop identifier* may contain displayable characters above character code 127 ($7F_H$), but this is not recommended. *Comop identifiers* are case-sensitive.

Examples of *comop identifiers*: ‹FilePath›, ‹tac.var›, ‹#abc%03?›, ‹sub:›, ‹.xyz-3›.

copy (of a *stack object*)
To reproduce a *stack object* and its *embedded value* into another *stack object* (replacing that other *stack object*) such that the reproduced value and the original value are identical. The *embedded value* of a *stack object* that has a *resource value* is an internal reference to that *resource value*. Therefore, if such a *stack object* is *copied*, only its reference is reproduced not its *resource value*. Consequently, if a *stack object* that has a *resource value* is *copied* to another *stack object*, both objects will share the same *resource value*.

current action
A property of a *stack object* that indicates its current action when *activated*.

custom comop number
A positive integer identifying a particular custom module to execute for the *comop*. The module exists in the *standard TAC library* or an *external TAC library*, and is implemented in machine code not *ETAC code*.

D

data dictionary
The *dictionary* contained in a *data object*. That *data dictionary* is identified by the name defined by the private pre-processor definition ‹_DATA_DICT›.

data object
The container of a *dictionary* used as a programmer-defined data structure consisting of *stack objects* identified by name (see *dictionary keyword*). The *dictionary* itself is identified by the name defined by the private pre-processor definition ‹_DATA_DICT›.

dictionary
A *stack object* having a *resource value* consisting of a list of internally indexed *dictionary items*. The *dictionary item* having the highest index value in its *dictionary* is called the ‘topmost’ *dictionary item*.

dictionary item
An item in a *dictionary* consisting of a label having the syntax of a *comop identifier* and a *stack object*. A *dictionary item* need not be unique to any *dictionary*; a *dictionary* can contain more than one identical *dictionary item*, and any other *dictionary* can contain the same identical item. A *dictionary item* within a *dictionary* is uniquely identified by an internal index. When a *dictionary item* is added to a *dictionary*, the item gets the next index value in the *dictionary*. The *dictionary item* having the highest index value in its *dictionary* is called the ‘topmost’ *dictionary item*.

dictionary keyword
The label of a *dictionary item*. A dictionary keyword typically has the syntax of a *comop identifier*.

dictionary stack
One of the three stacks in the *ETAC interpreter* that can contain only *dictionaries*.

duplicate (of a *stack object*)
To entire reproduc a *stack object* and its value into another *stack object* (replacing that other *stack object*) such that the reproduced value and the original value share no resources. *Duplication* is recursive. If the *stack object* does not have a *resource value*, then the *embedded value* of that *stack object* is reproduced.

E

embedded value (of a *stack object*)
The value of a *stack object* that is exclusively associated with that object (eg: integer, decimal, and string *stack objects* have *embedded values*). An *embedded value* is not shared with other *stack objects*, and can therefore be changed independently of the value of those other objects.

error event
The situation that occurs when the action of an *active stack object* can no longer proceed. In such a case, the *ETAC interpreter* intercepts the action and takes appropriate action which typically consists of ending the *main ETAC session* unless the *error event* is trapped by appropriate *ETAC code*.

ETAC code
This is *ETAC script* or *TAC binary instructions*. A file containing *ETAC code* typically has an extension of `etac`, `tac`, `ptac`, or `btac`.

ETAC expression
A consecutive sequence of one or more *script tokens* as defined for *ETAC expression* in the document ETACProgLang(Official).pdf.

ETAC function
The container of a special ETAC created *procedure* that creates a *local dictionary* then assigns the *object stack* arguments to that *dictionary* before calling the programmer-defined *procedure*. An *ETAC function* is typically accessed via a *function command*.

ETAC interpreter
A computer program that processes *ETAC code*. An *ETAC interpreter* essentially consists of a *script interpreter*, a *binary interpreter*, and a *TAC processor*.

ETAC packed script
ETAC text script that has been pre-processed or expanded, and then compressed. A file containing *ETAC packed script* is a binary file, typically having an extension of `ptac`.

Note that the term "*ETAC packed script*" is used in the same sense as the word "code", as in "*ETAC packed script* code".

ETAC script
This is *ETAC text script* or *ETAC packed script*. A file containing *ETAC script* typically has an extension of `etac`, `tac`, or `ptac`.

Note that the term "*ETAC script*" is used in the same sense as the word "code", as in "*ETAC script* code".

ETAC session
The period devoted to the processing of *ETAC code* by the *TAC processor* after having been processed by the *script interpreter* or *binary interpreter* (whichever is appropriate). New *ETAC sessions* can exist among a given *ETAC session* for different *ETAC code*. Therefore, a given *ETAC session* can produce a new *ETAC session* (relating to different *ETAC code* from the given *ETAC session*) so that when the new *ETAC session* ends, the given *ETAC session* resumes.

ETAC statement
A consecutive sequence of one or more *script tokens* as defined in the document ETACProgLang(Official).pdf for *ETAC statement*.

ETAC text script
ETAC program code that is in human readable and writable text form. This includes *TAC text instructions*. *TAC text script* containing *comops* in the form of *variable identifiers* is also *ETAC text script*. A file containing *ETAC text script* typically has an extension of `etac` (or ‹`tac`› if the file contains only *TAC text script*).

Note that the term "*ETAC text script*" is used in the same sense as the word "code", as in "*ETAC text script* code".

ETAC variable
A *dictionary item* whose *dictionary keyword* is in the form of a *variable identifier*, and whose *stack object* is intended to be different at various times during an *ETAC session*. The 'value' of an *ETAC variable* is the value of the said *stack object*. The '*variable object*' is the said *stack object* itself.

evaluate (of a *stack object*)
The value of a *stack object* after having been pushed onto a (particular) *TAC stack* as a result of the *activation* of a specified *stack object*. "*Stack object x evaluates* to *y* on *TAC stack z*" means that when *x* is *activated*, it pushes a *stack object* containing value *y* onto a *TAC stack z*. In some cases, *y* just represents the type of *stack object* pushed onto the *TAC stack*. For example, "*x evaluates* to an integer *stack object* on the *object stack*" means that when *x* is *activated*, it pushes an integer *stack object* onto the *object stack* (the value of the integer need not be specified). If the *TAC stack* is not specified, then the *object stack* is assumed.

external TAC library
A library of functions implemented by a programmer in the C++ programming language to extend the functionality of the ETAC programming language. The functions exist in a **Windows**® DLL (dynamic linked library), but are used as *comops* or *ETAC functions* by the ETAC programmer.

F

function command
A *command* associated with a *dictionary item* whose *stack object* is an *ETAC function*. When a *function command* is 'called', then its corresponding *ETAC function* is executed. When a *function command* is '*activated*', then its corresponding *ETAC function* is pushed onto the *object stack*.

function member
A *member* whose *stack object* is an *ETAC function*.

I

instruction form (of a *script token*)
A *script token* in the form of a *TAC text instruction*.

intrinsic command
A *command* that is associated with a function defined internally to the *ETAC interpreter*, or a *stack object* created by such a *command*. The *activation* of an *intrinsic command* does not involve the *dictionary stack* (an *intrinsic command* is *activated* directly).

L

lexical analyser
Part of the *script interpreter* that converts *lexical tokens* to *logical tokens* which are then syntax checked, modified, and rearranged as necessary.

lexical parser
Part of the *script interpreter* that parses *ETAC script* into *lexical tokens*.

lexical token
The smallest unit of information, in the form of text characters, that can be identified by the *lexical parser*.

local dictionary
A *dictionary*, typically existing temporarily, that is identified by the name defined by the private pre-processor definition ‹_LOCAL_DICT›. A *local dictionary* is typically used to contain the local variables of an *ETAC function*.

logical token
A combination of one or more *lexical tokens* and internal tokens regarded as a conceptual unit by the *lexical analyser* for the purpose of syntax checking and compiling a programming language.

M

main ETAC session
An *ETAC session* and all other new *ETAC sessions* produced directly or indirectly from that *ETAC session*, but not itself produced from any other *ETAC session*. A *main ETAC session* is typically begun via the RunETAC.exe and the AppETAC.dll computer programs.

member (of a *data object*)
A *dictionary item* of the *dictionary* contained in a *data object*.

member variable (of a *data object*)
A *member* of a *data object* that is an *ETAC variable* (or rarely a *TAC variable*).

N

nominal action (of a *TAC object*)
The default action of a *TAC object*.

O

object stack
One of the three stacks in the *ETAC interpreter* that can contain any type of *TAC object*. This is the main stack used by *ETAC code*.

operator
A *script token* containing the syntax of a *comop identifier*. An operator could be in *script form* qualified by a preceding ‹&› (eg: ‹&AddVect›, ‹&tac.var›, ‹&#abc%03?›, ‹&add:›, ‹&.xyz-3›) or *instruction form* (eg: ‹OPR:AddVect›, ‹OPR:tac.var›, ‹OPR:#abc%03?›, ‹OPR:add:›, ‹OPR:.xyz-3›). An operator is used in an *operator expression*.

operator expression
A consecutive sequence of *script tokens* involving an *operator* and its operands. There are two forms of *operator expressions*. One, where the operands are delimited by parentheses, and two, where the operands are delimited by the start_op and end_op *commands*. The *operator* of an *operator expression* can exist anywhere within its operand's delimiters.

Typically, when an *operator expression* is *activated*, its operands get *activated* first leaving the *operator* arguments on the *object stack*, then the *operator* gets *activated* and processes those arguments, returning the resultant *stack object* on the *object stack*. For example, the *operator expression* ‹(3 + 4 5)› will return 12 on the *object stack*. That *operator expression* can be written as: ‹(+ 3 4 5)›, ‹(3 4 5 +)›, ‹(3 4 + 5)›, ‹end_op 3 4 &add 5 start_op›, ‹start_op; 5; 4; &add; 3; end_op;›. Note that the *operator expressions* in all but the last example are *activated* from right to left; the *operator expression* of the last example is *activated* from left to right.

An *operator expression* can contain nested *operator expressions* as some or all of its operands, but each *operator expression* must contain exactly one *operator* at the top level.

operator stack
One of the three stacks in the *ETAC interpreter* that can contain only operator *stack objects*.

P

procedure
A special *sequence*, which, when *activated*, the elements of that *sequence* get *activated*. The elements of a *procedure* are typically command *stack objects*.

procedure expression
A group of *script tokens* that creates a *procedure*.

R

replicate (of a *stack object*)
This is the same as *duplicate*, except that *procedures* at all levels are *copied* rather than *duplicated*. The reproduced value and the original value share no resources except for *procedure resource values* which are shared.

resource value
The value of a *stack object* that can be shared with other *stack objects* of the same type — sequence, procedure, dictionary, and memory *stack objects* have sharable *resource values*. A *resource value* is internally referenced by the *stack object*; that reference itself is the object's *embedded value* (the reference itself is not available to the programmer, only the value being referenced, the *resource value*, is available).

S

script form (of a *script token*)
A *script token* not written in the form of a *TAC text instruction*. This is a more natural and intuitive style of expressing *script tokens*.

script interpreter
The part of the *ETAC interpreter* that processes *ETAC script*. The *script interpreter* consists of a *lexical parser*, a *script pre-processor*, and a *lexical analyser*.

script pre-processor
The script pre-processor is that part of the *script interpreter* that is responsible for pre-processing *ETAC text script*.

script token
A consecutive sequence of one or more *lexical tokens* regarded as a unit for the purpose of defining the syntax and semantics of the ETAC programming language.

sequence
A *stack object* having a *resource value* consisting of any number (including zero) of indexed *stack objects* understood as a unit. The indexed *stack objects* are the 'elements' of the *sequence*. The first *element* begins at index one, the second *element* is at index two, and so on. The number of *elements* in a given *sequence* is variable but limited by available memory. The *elements* of a *sequence* can be any type of *stack objects*, including *sequences*.

sequence expression
A group of *script tokens* that creates a *sequence* when *activated*.

sequential reverse-flow activation
The processing of *token statements* in *ETAC text script*, where the *script tokens* within each *token statement* are *activated* from right to left, but the *token statements* themselves within the *ETAC text script* are processed sequentially from left to right.

stack object
Any one of a number of certain groups of *TAC objects*.

standard TAC library
This is a library of custom *comops* that are implemented internally to the *ETAC interpreter*. Each *comop* has a unique *custom comop number*. Custom *comops* must be loaded via the execute custom *TAC text instruction* or the custom *command* before they can be *activated* (this is normally done automatically).

subsequence
A *sequence* that is a direct element of another *sequence*.

T

TAC binary instruction
A binary form of a *TAC text instruction*. *TAC binary instructions* exist in binary files. Any *ETAC code* can be compiled into *TAC binary instructions* by the **ETAC Compiler** program. A file containing *TAC binary instructions* typically has an extension of `btac`.

TAC error code
An error code produced as a result of an *error event* or produced explicitly by *ETAC code*. The file ETACErrorCodes.pdf contains a list the possible *TAC error codes*.

TAC object
An entity that has the capability of existing on a *TAC stack*, and consists of a type and corresponding value along with an indicator of some suitable action to perform.

TAC processor
Part of the *ETAC interpreter* that creates a *TAC object* from each *logical token* passed to it then *activates* the *TAC object* according to its type.

TAC stack
An *object stack*, *dictionary stack*, or *operator stack*.

TAC text instruction
A human readable text instruction of the form ‹*type*:*argument*› where *type* is any one of: `INT`, `DEC`, `STR`, `LBC`, `LBO`, `CMD`, `OPR`, `MRK`, `MEM`, `NUL`, or `EXE`, and *argument* is an appropriate argument for *type*. *TAC text instructions* may exist in *ETAC text script* files or in files containing only *TAC text instructions*. The **ETAC Compiler** program can compile *ETAC code* to *TAC text instructions*. A file containing *TAC text instructions* alone typically has an extension of `tac`.

TAC text script
TAC program code that is in human readable and writable text form. This includes *TAC text instructions*. *TAC text script* does not contain ETAC program code (*ETAC expressions* or *ETAC statements* other than assignment or allocation statements). A file containing *TAC text script* typically has an extension of `tac`.

Note that the term "*TAC text script*" is used in the same sense as the word "code", as in "*TAC text script* code".

TAC variable

A *dictionary item* whose *dictionary keyword* has the syntax of a *comop identifier*, and whose *stack object* is intended to be different at various times during an *ETAC session*. The 'value' of a *TAC variable* is the value of the said *stack object*. The '*variable object*' is the said *stack object* itself.

token statement

An arbitrary consecutive sequence of one or more *script tokens* where the final token in the sequence is followed by a comma ‹,›, semicolon ‹;›, right square bracket ‹]›, or right brace ‹}›. A *token statement* begins after a comma, semicolon, left square bracket ‹[›, or left brace ‹{›, or may begin at the start of *ETAC text script*.

V

variable identifier

A consecutive sequence of characters beginning with an alphabetic character ('a' to 'z' or 'A' to 'Z' or exotic Latin characters such as 'Ä'), an underscore (_), or an 'at' character (@). The subsequent characters are alphanumeric (alphabetic or '0' to '9') or underscore. Note that, by convention, *variable identifiers* beginning with an 'at' character, or an underscore followed by an alphabetic character or underscore, are reserved for system use. An ETAC programmer, therefore, is limited to defining *variable identifiers* containing alphanumeric characters and underscores, with the first character being an alphabetic character, or the first two characters being an underscore followed by a digit character. In addition, none of the strings "if", "then", "else", "endif", "when", "is", "endwhen", "do", "repeat", "from", "to", "step", "with", "of", "while", "exitdo", "exitdo_if", "donext", "donext_if", and "void" can be a *variable identifier*. *Variable identifiers* are case-sensitive.

The exotic Latin characters are: ª, ², ³, µ, ¹, °, À, Á, Â, Ã, Ä, Å, Æ, Ç, È, É, Ê, Ë, Ì, Í, Î, Ï, Ð, Ñ, Ò, Ó, Ô, Õ, Ö, Ø, Ù, Ú, Û, Ü, Ý, Þ, ß, à, á, â, ã, ä, å, æ, ç, è, é, ê, ë, ì, í, î, ï, ð, ñ, ò, ó, ô, õ, ö, ø, ù, ú, û ,ü ,ý, þ, ÿ. Those characters should be used only if necessary.

variable object

The *stack object* identified by a *TAC variable*, *ETAC variable*, or *member variable*.