TILE

Threaded • Interpretive • Language • Environment

St. Andrews University

Department of Computational Science

Senior Honours Project 1989-90

Implementation & Documentation by

Steven James

University of St. Andrews,

North Haugh,

St. Andrews,

Fife.

Preface

This manuscript is the outcome of a sequence of events that began in October 1989. It was originally intended that the honours project should take the form of a major programming assignment undertaken by a team of senior student working over a period of two terms. However due to unforeseen circumstances the original team broken up after only one term with little in the way of completed work. The remaining term was spent designing and implementing the threaded interpretive language outlined in this document.

The documentation contained herein is therefore not as extensive as could have been expected had the project run its normal course. It had originally been my intention in include a full project report and user manual, however the pressure of closing deadlines has regrettably forced me to omit these section leaving only those parts which I consider essential. Because of this decision I am now inclined to suggest that the documentation is aimed at the reader with some knowledge of a concepts and techniques underlying the design of a threaded interpretive language and not the novice.

Fortunately the deign and implementation is complete and stands as example of how much can be accomplished by one person in so short a time. In addition to the standard language I also though it necessary to incorporate the source code of some working examples of real applications. For this reason the source listing section contains a set of language extensions including such constructs as vectors, abstract data structures and a case statement as well as a complete and working basic compiler.

Table of Contents

1

Preliminary Specification

1.1 Introduction

The preliminary specification of a threaded interpretive language (hereafter refereed to as a **TIL**) is far less critical in terms of the influence it will have over the eventual shape of the language, than would be the case were it for a more conventional language such as **PASCAL** or **S-algol**. This is in part the result of a design decision that allows a **TIL** to be extended and retracted to such a degree that it no longer resembles or has any relation to its initial form. The specification is never the less an import and essential part of the design process, in that it provides an initial description of the project as a whole, and gives direction to the forth coming design effort.

1.2 Overview

The initial specification was to design and implement a **TIL** drawing upon the ideas and concepts from existing thread interpretive languages such as **FORTH** and **IPS**, while maintaining a high degree of independence from both, by incorporating many new ideas and implementation techniques.

1.3 Design Considerations

The specification of any large software project can seen to consist of two separate and distinct phases. The initial phase is concerned with the overall design considerations for the project as a whole, with little or no attention to specific design issues or fine detail. The design considerations for a **TIL** or indeed any computer language fall naturally into two categories.

1.3.1 Language Elements

A **TIL** can be described in terms of a collection of functionally dependent modules, each representing one unique aspect of the language.

1.3.1.1 The Interpreter

The interpreter is the primary interface between the user and the abstract machine. Broadly speaking its function can be described as the mapping of

lexemes for an input stream via the dictionary structure to fragments of abstract machine code. The source of such lexemes can be either the terminal keyboard or a file.

1.3.1.2 The Abstract Machine

The abstract machine is the underlying execution mechanism, and can be view as the very essence of a **TIL**. Its structure is more or less fixed, and is dependent upon the type of treaded execution implemented. The instruction set of the abstract machine is based around the functions provided by the **TIL**, each primitive within the language represents a single abstract machine operation.

1.3.1.3 The Dictionary

The dictionary provides the primary heap management system for a **TIL**. Its structure is hierarchically based around a system of linked lists organised into a tree structure, each leaf node representing a separate list, or vocabulary of secondary function definitions.

1.3.2 Language Constructs

Having described the elements of **TIL**, it now remains to describe the constructs supported by those elements.

1.3.2.1 Arithmetic

The arithmetic is based upon single length (16 bit), and double length (32 bit) signed integers. Elementary Arithmetic operations are provided to perform single and double length addition and subtraction, and single length multiplication and division.

1.3.2.2 Logic

The logic is based upon single length (16 bit) bitwise operations, and single length (16 bit) comparative operations. Elementary bitwise operations are provided to perform **and**, **or** , **xor**, and **not**. together with the equality operations (=) equal, (<) small than and (>) greater than.

1.3.2.3 Stack Manipulation

The stack manipulation is based upon the four elementary stack operations **swap**, **dup**, **over**, and **rot** .

1.3.2.4 I/O

The I/O is based around elementary stream manipulation operations, and provides a stream selection operation, and read and write character operations upon the selected stream.

1.3.2.5 Memory Manipulation

The memory manipulation is based around the storage and retrieval of byte quantities from 64K of addressable memory. single length (16 bit) operations may be implemented with two byte operations.

1.3.2.6 Dictionary Manipulation

The dictionary manipulation is based around the creation and subsequent location of dictionary headers and associated data.

1.3.2.7 Numeric Manipulation

The numeric manipulation is based around the **constant** and **literal** operations upon single (16 bit) integers.

1.3.2.8 String Manipulation

The string manipulation is based around the parsing of lexemes from the input stream, and the subsequent conversion of such lexemes into their integer equivalent.

1.3.2.9 Flow Control

The flow control is based around conditional and unconditional, forward and backward branch operations.

1.4 Implementation Considerations

The final phase of the specification process is concerned with the implementation considerations for the project as a whole, with little or not attention to to specific implantation issues or fine detail. The implementation considerations for a thread interpretive language fall naturally into two categories.

1.4.1 Hardware

The hardware considerations are primarily concerned with the selection of implementation hardware. The economics of departmental funding limit the choice to one of two machines.

1.4.1.1 Apple Macintosh Plus

The Apple Macintosh is an extremely successful microcomputer with an outstanding user interface, but a non standard operating system. The basic hardware comprises of a Motorola 68000 microprocessor and 1 Megabyte of ram. Peripheral are available via the Apple Talk™ network.

1.4.1.2 Sun 3/60

The Sun 3/60 is an integrated development environment with an outstanding user interface, and standard unix™ operating system. The basic hardware comprises of a Motorola 68020 microprocessor and 8 Megabytes of internal ram, and an indefinite amount of virtual memory. Peripherals are available via the Sun Eithernet™ network.

The **Sun 3/60** promises both better performance, and a greater degree of portability, with the additional advantage of the support of a major operating system.

1.4.2 Software

The software considerations are primarily concerned with the selection of an implementation language. The possible alternatives can be short listed to a choice of just two, **ANSI 'C'** and **PS-Algol**.

1.4.2.1 ANSI 'C'

ANSI 'C' is rapidly establishing itself as the standard systems programming language. It offers both efficient execution, and a high degree of portability between machines and operating systems.

1.4.2.2 PS-Algol

PS-Algol is relatively new programming language little known outside the university environment. It offers many outstanding feature such as persistent database management and first class procedures, but suffers in terms of execution and portability because of its relative obscurity.

ANSI 'C' promises both better performance and a greater degree of portability than **PS-Algol**. There also exist a large number software tools to aid in the development and debugging of **'C'** programs.

1.5 Summary

To recap, the project is to design and implement a **TIL** based around an interpreter to parse lexemes from the input stream, a dictionary structure to map lexemes to abstract machine code fragments, and an abstract machine to execute the code fragments. The project will be implemented upon a **Sun 3/60** using **ANSI 'C'**.

2

The TILE Dictionary

2.1 Introduction

The dictionary constitutes the largest single data structure used within a **TIL**. It not only provides the memory management mechanism, but is also an essential and integral part of the interpreter. Its function with respect to the interpreter, can be broadly described as the mapping of lexemes from the input stream, to machine code fragments and data, stored within the dictionary structure.

2.2 Dictionary Structure

The dictionary is based around a linked list, with each dictionary entry as a node within the list structure. Each dictionary entry consists of two parts, the head and the body. The head of the entry contains some information about the name of the word (**the name field**), a pointer to the previous word in the dictionary (**the link field**) and a pointer to the machine code associated with the word (**the code field**). Figure 2.2.1 show two typical dictionary entries, a code primitive (left) and a secondary (right). The name field address is prefixed with a header byte, containing information related to the nature of the word. The individual bits within the byte are used for different purposes by the interpreter. Bits 1 to 6 inclusive contain the length in bytes of the word name (up to a maximum size of 63 bytes), bit 7 indicates whether or not the word can be found via a dictionary search (used to hide words during compilation) and bit 8 indicates if the word is immediate (executable during compilation). Figure 2.2.2 shows the bit-by-bit format of the header byte.

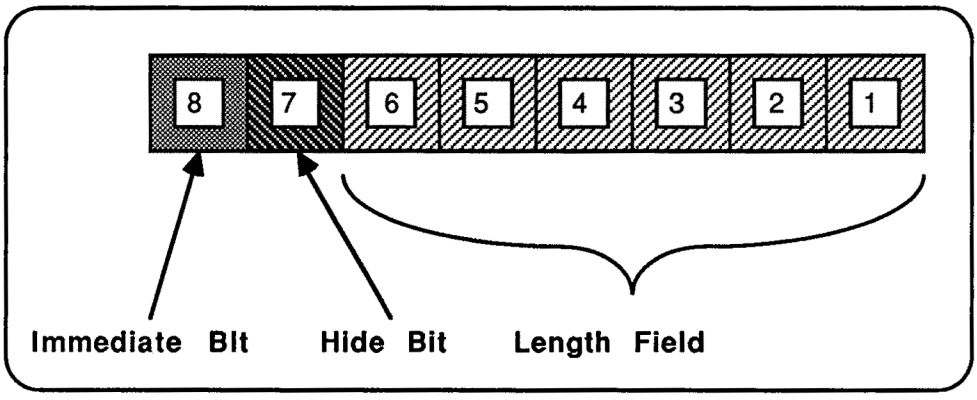

Figure 2.2.2 - Diagram of the header byte.

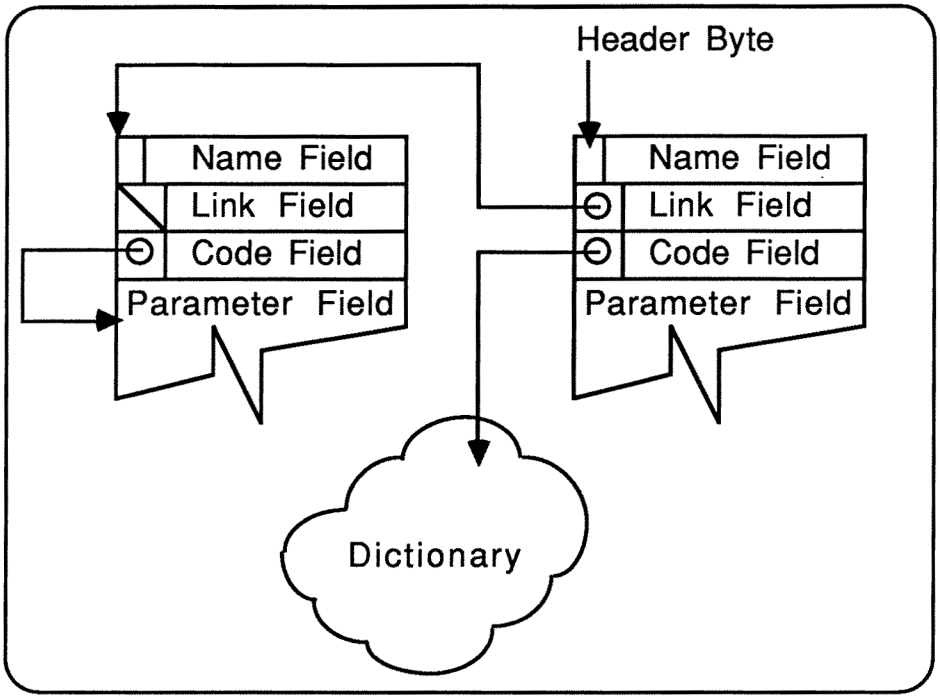

Figure 2.2.1 - Diagram of two typical dictionary entries.

The body (**parameter field**) of the entry contains either an abstract machine instruction in the case of a code primitive, a list of pointers to other dictionary entries in the case of a secondary or any other information that may need to be stored in association with a dictionary header.

2.3 Dictionary Classes

The various classes of word differ only in the contents of their code and parameter fields. The code field always contains a pointer to an abstract machine instruction and the parameter field will always contain data of some form.

2.3.1 Code Primitives

The parameter field of a code primitive contains the actual instruction to be executed by the abstract machine. The code field contains an address which points to the parameter field and subsequently the abstract machine instruction. The operation of **next** (see section 4) causes a jump to the address contained within the code field, and the execution by the abstract machine of the instruction therein.

2.3.2 Constants

The value of the constant is contained within a two byte parameter field. The code field contains the address of the abstract machine instruction **const_op** which

uses the address in **WA** (the word address register) to locate the parameter field and copy its contents to the parameter stack.

2.3.3 Colon-definitions

The parameter field of a colon-definition contains a list of the addresses of other **TILE** words. The code field contains the address of the abstract machine instruction **colon_op** to start interpretation of the parameter field list.

2.3.4 Words constructed using <builds and does>

The parameter field of such words may contain any combination of values and addresses, depending upon the **does>** part of the created word. On execution the address of the parameter area is left on the stack so that the contents can be used by the sequence of words following **dose>** in the creating word. The code field of such a word contains the address of the abstract machine instruction **does_op**.

2.4 Dictionary Vocabularies

A vocabulary is a subset of the dictionary defining a localised search order to those words defined within the vocabulary. The basic structure of a vocabulary is a tree with an initial vocabulary at the root, and each subsequent vocabulary definition as a branch. Figure 2.4.1 shows the basic dictionary structure containing the root vocabulary **tile** and two sub-vocabularies **extensions** and **basic**. The **basic** vocabulary also contains three sub-vocabularies **logic** , **in** and **out**.

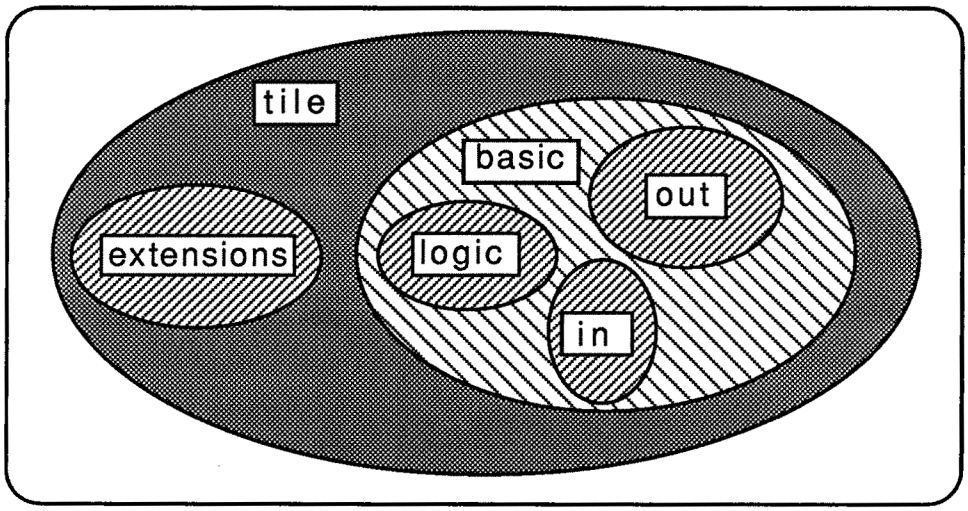

Figure 2.4.1 - Diagram of the basic dictionary structure.

2.4.1 Vocabulary Search

The search order of a vocabulary is defined by the user variable **CONTEXT** which contains the address of a variable containing the address of the most recently defined word within the **CONTEXT** vocabulary. The vocabulary search starts form the most recent definition and continues up the vocabulary tree until a terminator (**a word containing a null link field**) is encountered. The exact path the search follows is dependent upon the order in which the vocabularies were defined. The search initially starts with the **CONTEXT** vocabulary and then continues with the vocabulary in which it was defined (**the parent vocabulary**) starting from the vocabulary definition. Thus only those definitions present at the time a vocabulary was defined are search, and not those definitions added later.

2.4.2 Vocabulary Definitions

The structure of a vocabulary definition is based around a <build does> construct, with the address of the **CONTEXT** user variable together with its contents stored in the parameter field. The run-time action of a vocabulary definition is to store the address of the later part of the parameter field in the **CONTEXT** user variable, thus determining the order of all subsequent vocabulary searches. Figure 2.4.2.1 shows the structure of a vocabulary definition and gives the **TILE** code necessary to generate such a definition.

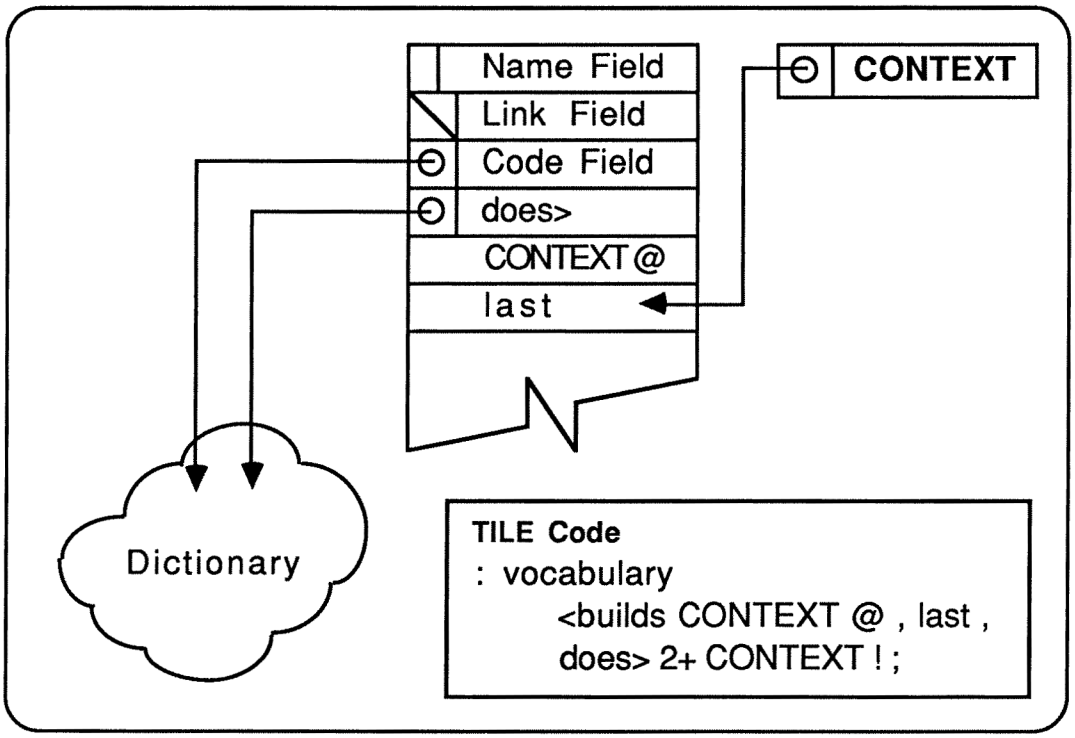

Figure 2.4.2.1 - Diagram of the structure of a vocabulary definition.

The second entry within the parameter field contains the address of the first entry within the parameter field of the previous vocabulary definition. This is used as part of the sealed vocabulary search mechanism described in section 2.4.3.

2.4.3 Sealed Vocabularies

The search mechanism described in section 2.4.1 can be extended to provide a more powerful and structured vocabulary search mechanism by the inclusion of the sealed vocabulary construct. A sealed vocabulary is a vocabulary containing a terminator that prevents the search from continuing further up the vocabulary tree. The effect of a terminator is to force the search mechanism to resume the vocabulary search starting from the most recent definition in the parent vocabulary.

2.5 Summary

The vocabulary construct although not the most complex of implementations is still never the less extremely powerful, enabling concepts such as context switching and operator overloading to fully exploited.

3

The TILE Interpreter

3.1 Introduction

The interpreter is essentially the man-machine interface of a **TIL** providing a link between the high-level constructs of the user and the low-level abstract machine. Although the interpreter is conceptually a single entity it is dependent for much of its operation upon other language components such as the dictionary and abstract machine. In this sense the interpreter can be seen as a mechanism for combining and manipulating the language environment.

3.2 Interpretation Mechanism

The mechanics of the interpreter are best portrayed in graphic form with the aid of a flowchart. Figure 3.2.1 shows just such a flowchart and outlines those essential operations required at each stage of the interpretation process and the sequence in which they occur.

3.2.1 Get next line

Accept the next character sequence from the input stream terminating upon a carriage return or end of file marker and store as a string in the text input buffer. This operation is supported as part of the abstract machine by the **inline** operation.

3.2.2 Get next word

Transfer the next space delimited lexeme from the text input buffer to the text scratch pad and store as a counted string. This operation is supported as part of the abstract machine by the **word** operation.

3.2.3 EOL ?

Check if the text input buffer has been exhausted.

3.2.4 Search the dictionary

Search the dictionary for the counted string in the text scratch pad starting with the most recently defined word in the **CONTEXT** vocabulary. This operation is

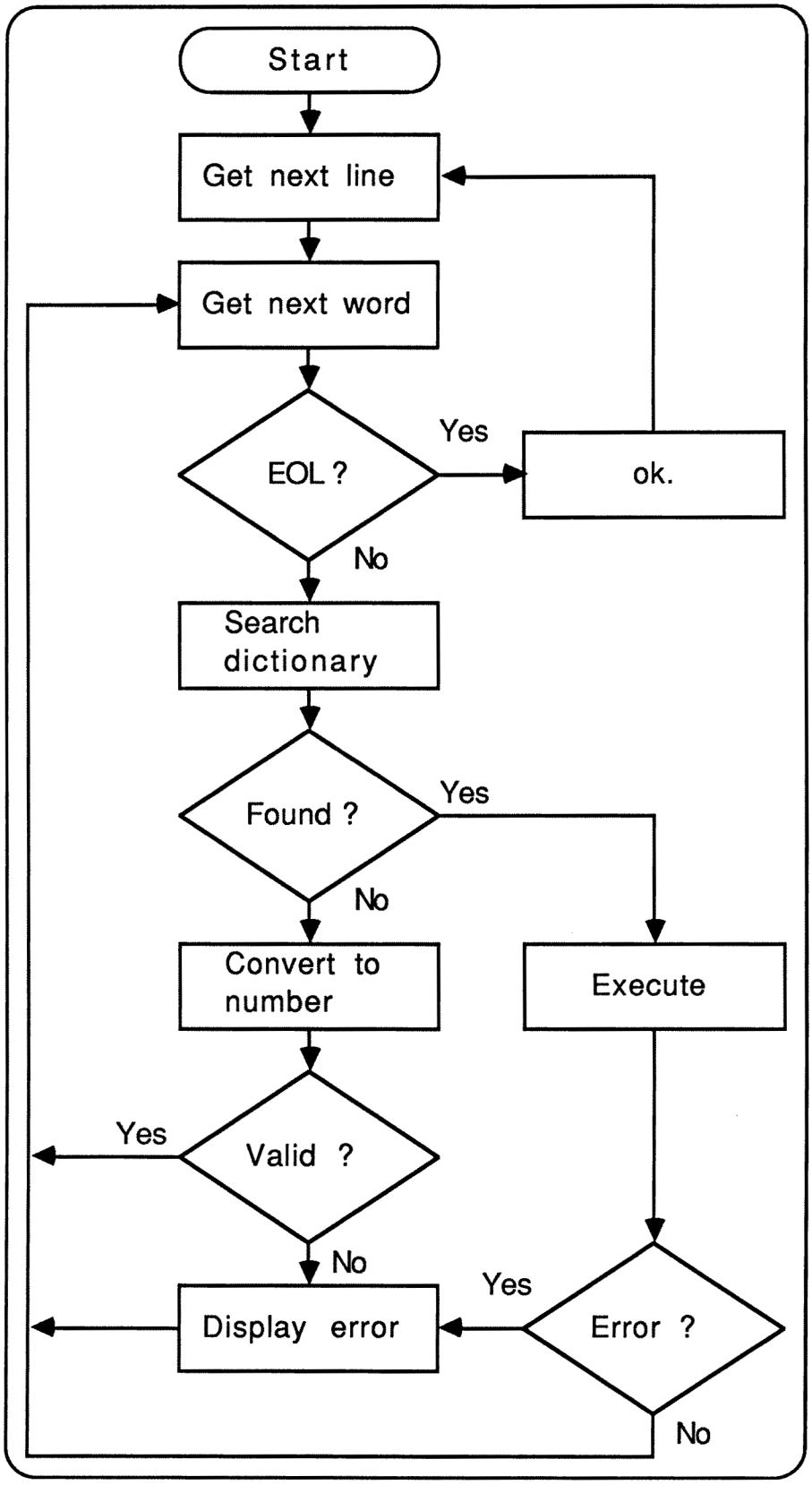

Figure 3.2.1 - Flowchart of the TILE interpreter.

supported as part of the abstract machine by the **find** operation. A more complete description of the dictionary search mechanism can be found in section 2.

3.2.5 Found ?

Check in the dictionary search described in section 3.2.4 has been successful.

3.2.6 Convert to number

Convert the counted string in the text scratch pad into its integer equivalent according to the current value of the **BASE** user variable and places the resulting value onto the parameter stack.This operation is supported as part of the abstract machine by the **number** operation.

3.2.7 Execute

Execute the word corresponding to the counted string in the text scratch pad using the code field address supplied by the dictionary search in section 3.2.4. This operation is supplied as part of the abstract machine by the **execute** operation.

3.2.8 Valid ?

Check if the conversion described in section 3.2.6 resulted in a valid number.

3.2.9 Error ?

Check if the execution described in section 3.2.7 resulted in any errors.

3.2.10 Display error

Display the appropriate error message and reset the abstract machine.

3.3 Interpretation and Compilation

The interpretation mechanism described in section 3.2 is limited in that all space delimited lexemes in the text input buffer are viewed in the context of execution that is there is no facility within the interpreter to compile rather than execute a word or value. This extension of the interpreter to encapsulate the idea of compilation may be considered to great a departure from the conventional concept of a interpreter to be justified. The modifications necessary to extend the existing interpreter are however minor enough to justify the their inclusion within the one mechanism.

3.3.1 Convert to number

The number conversion operation describe in section 3.2.6 need only be modified to the extent that the value produced by the conversion is compiled into the next available dictionary location as a literal rather than be placed on the parameter stack.

3.3.2 Execute

The execution operation described in section 3.2.7 need only be modified to the extent that the code field address supplied by the dictionary search is compiled into the next available dictionary location rather than be passed to the abstract machine for execution.

The exact action taken by the interpreter is dependent upon a user variable **STATE**. If the lexeme is to be interpreted the **STATE** variable must be false (0) if however the lexeme is to be compiled then the **STATE** variable must be true (-1). Modification of the interpreter state **STATE** variable is normally reserved for defining words such as (:) colon, (;) semi-colon, ([ ) left-bracket and (]) right-bracket.

3.4 Implementation

The **TILE** interpreter has been implemented around the structure indicated in the flowchart (see figure 3.2.1) using the abstract machine operations described in sections 3.2 and 3.3. There are however a number of procedures unique to the interpreter that are not indicated on the flowchart or supplied as part of the abstract machine that are never the less essential to the operation of the interpreter. The **is_immediate** procedure is used by the interpreter to interrogate the header byte of a word and ascertain if or not the word is immediate (executable at compile time). The **executable** procedure used by the interpreter to interrogate each word in tern to determine if or not the word is to be compiled or executed.

```
int is_immediate( nfa )
unsigned short nfa ;
{
        if( get_byte( nfa ) & IMMEDIATE_BIT )
                return( TRUE ) ;
        else
                return( FALSE ) ;
}
```

```
int executable( nfa )
unsigned short nfa ;
{
        if( get_word( STATE ) == FFALSE )
        {
                return( TRUE )
        }
        else
                return( is_immediate( nfa ) ) ;
}
```

3.5 Summary

The **TILE** interpreter is an elegant and simple mechanism for conversing with the underlying abstract machine. Unlike conventional interpreters the distinction between the **TILE** language and the interpreter is not quite so distinct. The interpreter derives much of its beauty form the structures and constructs of the languages itself and as these are extend so to is the interpreter.

4

The TILE Abstract Machine

4.1 Introduction

The design of an abstract machine for any high level language will invariably be a compromise between the variously conflicting design aims. Portability, code efficiency, and the type and number of operations provided, must all be considered in-order to produce the combination that meet the language requirements as closely as possible. Ideally the abstract machine should map almost exactly onto the language it supports, with a one-to-one correspondence between an instruction in the high level language and an abstract machine instruction. An exact mapping is not however possible because of structural differences between the conceptual representation of semantic concepts on the one hand, and mechanisms required to achieve those semantic concepts on the other. **PASCAL** and **S-algol** have abstract machines based around this concept of high-level correspondence, and as a result both abstract machines are large and complex. Threaded interpretive languages on the other hand are far more fundamental in both concept and design than other high level languages, and subsequently this is reflected in the relative simplicity of the underlying abstract machine.

4.2 The Elements

The **TILE** abstract machine can be viewed as two interdependent modules, an underlying architecture to provide an environment through which **TILE** computations can be expressed. and the instruction set to manipulate the environment and provide the basic building blocks of the language, from which all **TILE** programmes will ultimately be composed.

4.3 The Architecture

The **TILE** abstract machine has been kept to an absolute minimum and consists of three 16 bit address registers and a 16 bit return stack, which are used exclusively by the abstract machine for the purpose of executing threaded instructions. A 16 bit parameter stack for the general data manipulation, and 64K bytes of addressable memory.

The three address registers can be further described in-terms of their relative meaning in the context of thread execution. The instruction register (**I**) contains the address of the

next instruction in the threaded list of the current keyword. The word address register (**WA**) containing the address of the current keyword or the address of the first code body location of the current keyword. The code address register (**CA**) contains the address of the next abstract machine instruction to be executed.

The two stacks are perhaps the most important part of the abstract machine, only by exploiting their unique properties can **TILE**'s reverse Polish notation and thread execution be implemented. The exact sizes of each of the stacks is dependent upon the type of computation required.

4.4 The Instruction Set

The very nature of a **TIL** leads to one of the most unusual design decisions regarding the instruction set of the underlying abstract machine. Because the entire language is based around a small number of mutually exclusive function it is perfectly feasible to implement the entire language as abstract machine instructions. That is for each function within the language there is an equivalent abstract machine instruction.The advantage in both speed and size of this approach is obvious, and indeed there are many commercially available microprocessors that provide just this type of implementation. The structure of a **TIL** also provides yet another strategy when designing the instruction set for the abstract machine. Because of the extendible nature of the language there is a minimum instruction set from which all other function can be defined, thus the instruction set need only contain these essential atomic instruction.

The **TILE** abstract machine can be regarded as a compromise between both of the design philosophies described, providing enough instruction to ensure efficient execution, and few enough to keep the abstract machine to sensible proportions.

4.5 Functional Composition Of The Instruction Set

The instruction set of the **TILE** abstract machine can be divided into ten broad functional categories, each representing one particular aspect of the computational environment.

4.5.1 Arithmetic

All arithmetic operations can be approximated to addition in one form or another. Subtraction, multiplication and division can all be achieved using just the addition operation. For completeness and efficient execution however, it is sensible to implement as many arithmetic functions as possible at a low-level to remove any unnecessary bottle-necks. Arithmetic operations using both signed

and unsigned quantities in single and double precision formats are also a feature which must be taken into consideration when deciding the extent to which arithmetic operations will be incorporated at the abstract machine level.

The **TILE** abstract machine is based around the minimum arithmetic requirements, and provides single and double length addition operations and the corresponding negation operations to convert between integer and 2's complement format, in order to implement subtraction. Unsigned single-length multiplication and division are the only other operations provided by the abstract machine, all other more complex arithmetic functions are defined at a higher level interns of theses lower level primitives.

4.5.2 Logic

In the same way that all arithmetic operations can be approximated to addition and negation. Logic operations can be approximated to a combination of **not** and **and**. However in the interests of efficient execution it is sensible to implement all logic operations separately at the abstract machine level. Conventional languages such as 'C' provide two types of logical operators, those concerned with Boolean truth values, and those concerned with bitwise or binary values. The overloading of logic operators in this way is entirely the result of a decision to adopt (1) and (0) as true and false truth vales respectively. Such a representation requires a complementary set of Boolean logic operations that perform essentially the same function as their bitwise counterparts. This ambiguity can however be eliminated by adopting (-1) and (0) as true and false truth values respectively, thus removing the need for a second set of logical operators.

The **TILE** abstract machine is based around bitwise logic operators and provides **AND** , **OR** , **XOR** and **NOT** as primitive logic operations, and (=) equal, (<) smaller than and (>) greater than as primitive comparative logic operations. The truth values of (-1) and (0) are conventionally adopted to represent Boolean truth values and prevent the ambiguity described above.

4.5.3 Stack Manipulation

The stack is the major parameter passing mechanism utilised by the abstract machine. It is therefore important to ensure that the stack manipulating instructions are as efficient as possible. For this reason all stack intensive operations are implemented at the abstract machine level.

The **TILE** abstract machine is based around a small number of stack operations and provides **SWAP** , **DUP** , **OVER** and **ROT** to manipulate the top three stack

elements, with a supplementary mechanism **>R** and **R>** to transfer data to and from the return stack to provide additional scratch work space.

4.5.4 I/O

All I/O can be regarded as either a read or write on a specific data stream. For the purposes of clarity, streams are usually divided into input streams (i.e. keyboard and file system) and out put streams (i.e. vdu and file system). All operations upon these streams can then be approximated to either a single or multiple character read or write. The more complex I/O functions can then be defined in terms of these low level primitives.

The **TILE** abstract machine supports two input streams (i.e. the keyboard and file system) and provides a universal line input operation **inline** for both, and a single character input operation **key** for the keyboard only. The file system can be selected as the primary input stream via the **load** operation, however when the input stream is terminated (i.e. an end of file marker is detected) the stream is reset to the keyboard. The only output stream supported is the vdu, with a single character output operation **emit**.

4.5.5 Memory Manipulation

All memory manipulation can be regarded as either a read or write. The exact size of the operation is dependent upon the amount of memory available, and the word length or size of the underlying abstract machine. Conventionally the minimum size of such operations is 8 bits or 1 byte, although there are architectures based around word lengths as large as 64 bits or 8 bytes, and as small as 4 bits or 1 nibble. Having established the word length, all subsequent memory operations can then be approximated to multiples of this fundamental unit.

The **TILE** abstract machine is based around a 16 bit or 2 byte word length, and provides the associated **@** (fetch) and **!** (store) operations to manipulate quantities of the basic length. In addition to the two 16 bit operations, there are also complementary 8 bit operations **C@** (c-fetch) **C!** (c-store) to manipulate byte quantities such as ASCII characters. Although the abstract machine manipulates 16 bit quantities, the memory system supports only 8 bit quantities. Therefore all 16 bit presentations are stored as two consecutive bytes with the most significant byte first.

4.5.6 Dictionary Manipulation

The dictionary constitutes over 90% of a **TIL**. It is therefore inevitable that the abstract machine will at least in part contain operations for manipulating and maintaining this essential component of the language.

The **TILE** abstract machine supports two dictionary oriented operations. The **find** operation to search the dictionary for a specific header, and the **create** operation to generate a new dictionary header. Other more complex dictionary operations such as localised vocabulary searching, constants, variables, and defining words can all implemented at a higher level in-terms of these dictionary primitives.

4.5.7 Numeric Manipulation

The manipulation of numeric information, as both constant and literal representations, is an essential part of the interpreter. Operations to evaluate in-line literals and constant functions must therefore be supported at the abstract machine level.

The **TILE** abstract machine supports two numeric oriented manipulation operations. The **constant** operation to evaluate constant functions and the **literal** operation to evaluate in-line literals.

4.5.8 String Manipulation

The manipulation of string information, as both ASCII representations of numeric values and delimited lexemes, is one of the central mechanisms of a **TIL**. Operations to converts ASCII character string to their numeric equivalent and scan the input stream for the next lexeme must be supported at the abstract machine level.

The **TILE** abstract machine supports two string oriented manipulation operations. The **number** operation to convert an ASCII character string into its numeric representation according the the current working **BASE**, and the **word** operation to transfer the next delimited lexeme from the input stream to the text scratch pad.

4.5.9 Flow Control

Conditional construct such as **if** ... **else** ... **then** , **begin** ... **while** ... **repeat** , and **begin** ... **until** are all variations upon essential the same flow control mechanism. This mechanism must therefore in provided as part of the abstract machine, to enable such higher level constructs to be implemented. All conditional structures can be approximated to two basic flow control mechanisms, a conditional branch

that transfers control depending upon the result of some Boolean expression, and an unconditional branch or jump, that will always transfer control regardless of the context in which it is executed.

The **TILE** abstract machine supports two flow control mechanisms. The conditional branch operation **branch** , and the unconditional branch operation **jump**.

4.5.10 Iteration

Iterative constructs such as **do** ... **loop**, and **do** ... **+loop** are essentially flow control mechanisms, and must therefore be implemented as part of the abstract machine. The flow control for both iterative constructs is depended upon an index incremented by one in the case of the **do** ... **loop**, and by a specified value in the cases of the **do** ... **+loop**.

The **TILE** abstract machine supports three iterative flow control mechanisms. The **do** operation to initiate the loop construct, the **loop** operation to maintain flow control based around an index increment of one, and the **+loop** operation to maintain flow control based around a specified index increment.

4.6 Execution Mechanism

The **TILE** abstract machine is based around a graph representation of function application. Each higher order function or secondaries, can seen as an n-tuple of function applications, which may themselves be n-tuples or abstract machine instructions. Thus the graph for a secondary is constructed from n-tuple function applications at each branch, and abstract machine instructions a each leaf node. Evaluating such a function is simply a case of performing and in-order traversal of the function graph, and executing the abstract machine instructions at each leaf node. The **TILE** abstract machine provides five basic mechanisms for evaluating or executing function graphs. The **colon** operation to mark the start of an n-tuple of function applications, the **semi** operation to denote the end of a n-tuple of function applications, the **next** operation to traverse the graph representing the next function application until a leaf node is encountered, the **run** operation to execute an abstract machine instruction at a leaf node, and the **does>** operations to transfer evaluation to a new point within the graph. Figure 4.6.1 shows the graph for a simple secondary function.

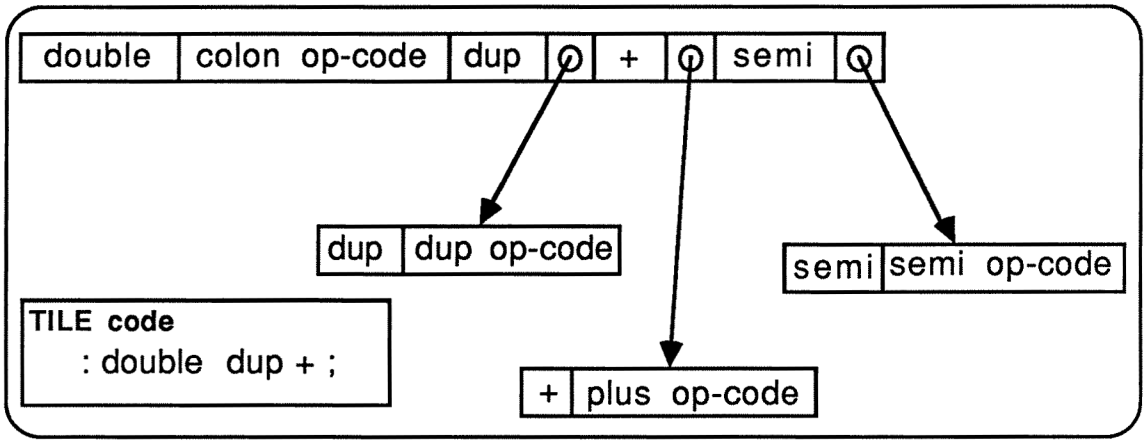

Figure 4.6.1 - Diagram of the graph for a secondary function.

4.6.1 The Colon Mechanism

The word containing the colon operation (**a colon-definition**) must have been called from some other word and initially **I** (**instruction register**) contains the address of a point within the calling word. The first action of the colon operation must therefore be to copy the contents of **I** to the return stack for later retrieval. The contents of **WA** (**word address register**) are then incremented by two to give the address of the start of the new word's parameter field and this address is stored in **I**. A call to **next** then starts the interpretation of the new word.

4.6.2 The Semi Mechanism

The word containing the semi operation must have been called as the last word from a colon-definition. The operation of semi is to transfer the top value from the return stack into **I** and call **next**. This value is the old contents of **I**, placed on the return stack by the **colon** operation and so the effect of the semi is to restore execution back to the point from which the colon-definition was called.

4.6.3 Next Mechanism

The operation of next is to execute the next word in the sequence. The contents of **I** Initially contains the address of the next word to be executed. The contents of the address contained within **I** are then therefore transferred to **WA** and **I** is incremented by two to point to the next word in the sequence. A call to **run** then starts execution of the new word.

4.6.4 Run Mechanism

The operation of run is to execute the abstract machine instruction in the sequence. The contents of the address contained within **WA** are then transferred

to **CA (code address register)** and **WA** in incremented by two to point to the next instruction in the sequence.

4.6.7 Does> Mechanism

The operation of does> is to execute the sequence of words following does> in the creating word. The action of does> is identical to that of **colon** save that before **next** is called the address of the parameter field of the word is copied onto the parameter stack.

4.7 Termination

Terminate either normally as the result of program completion, of abnormally as a result of some erroneous condition, is a facility that must be provided as part of the abstract machine.

The **TILE** abstract machine supports two termination operation. The **trap** operation which causes the abstract machine to trip back to the interpreter, and the **exit** operation which causes the abstract machine and the interpreter to terminate and return control to the operating system.

4.8 Implementation

The **TILE** abstract machine has been implemented as two separate modules, the architecture as described in section 4.3 and the instruction set as described in sections 4.5 though to 4.6. Both of theses modules have been further decomposed into their constituent components, each which is implemented separately as a **#include** library. The hierarchical structure of the abstract machine implementation, is a deliberate attempt to control complexity and improve the overall clarity of the source code.

4.8.1 Memory

The memory has been implemented as a 64K block of contiguous unsigned characters. The block is allocated with a **malloc** commend when **TILE** is first executed, and released with a **free** command upon termination.

```
char *base ;

void  claim_memory()
{
        base = (char *) malloc (BLOCKSIZE) ;
}
```

The memory manipulation is based around the get_byte and put_byte procedures. The **get_byte** procedure takes a base relative address and returns a pointer to the unsigned character that represents the contents of that address. The

put_byte procedure takes a base relative address and unsigned character, and stores the character at the position represented by the address.

```
unsigned short get_byte(addr)
unsigned short addr ;
{
        char *place ;
        place = base + addr ;
        return(*place) ;
}
```

4.8.2 Stacks

The two stacks have been implemented as two vectors of unsigned short's. This implementation was chosen over the dynamic equivalent (which is also implemented) simply because of the enormous performance overhead that would result from a linked list implantation.

The stack manipulation is based around four operations which are identical for both the return and parameter stacks. The **push** procedure push a an unsigned short onto the top the stack, and checks that overflow has not accrued. the **pop** procedure removes the top unsigned short form the stack, and checks that under flow has not accrued. The **reset** procedure resets the stack pointer to top of the stack, thus removing any data that may have been present. The **empty** procedure returns **TRUE** if then stack is empty, or **FALSE** if not.

```
unsigned short psstk[ PSTKSIZE ] ;

void push_ps(n)
        unsigned short n ;
{
        if( ! trapped())
                if( ps == PTSKSIZE )
                {
                        printf("Parameter stack overflow.\n") ;
                        forth_abort() ;
                }
                else
                pstk[ps++] = n ;
}
```

4.8.3 Arithmetic

The arithmetic described in section 4.5.1 is based around the unsigned short arithmetic operators provided as part of the basic **'C'** environment. The procedures **add_op** is a typical example of the structure of the single length arithmetic operations.