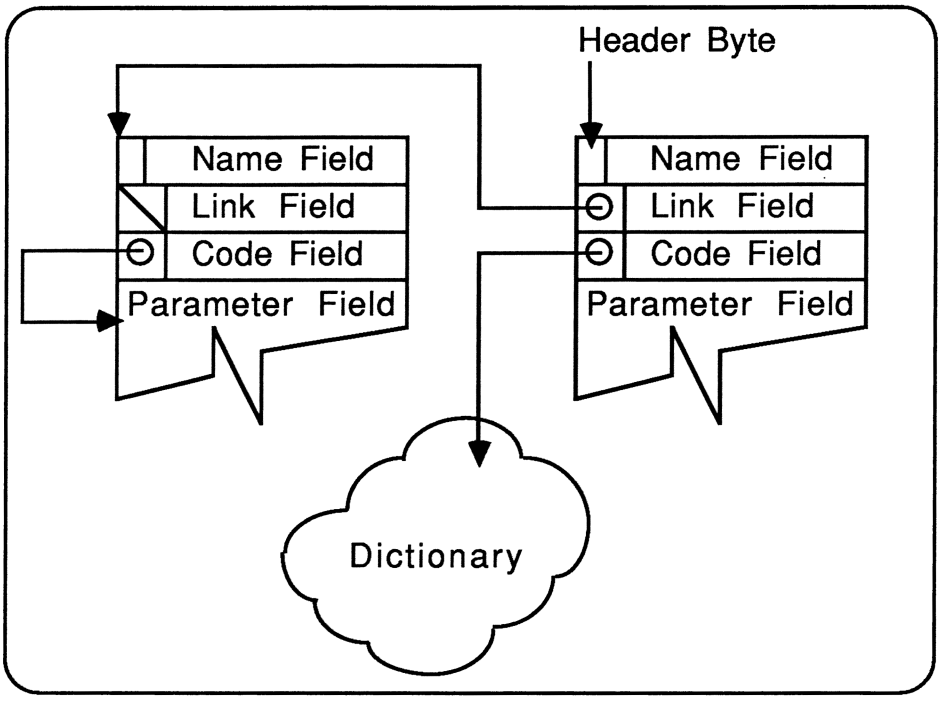

Figure 2.2.1 - Diagram of two typical dictionary entries.

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**.

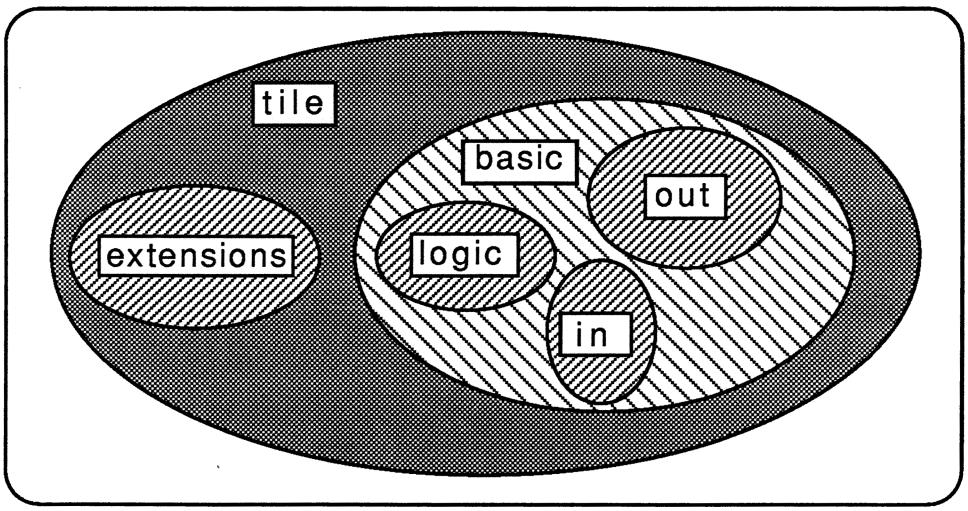

Figure 2.4.1 - Diagram of the basic dictionary structure.

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.

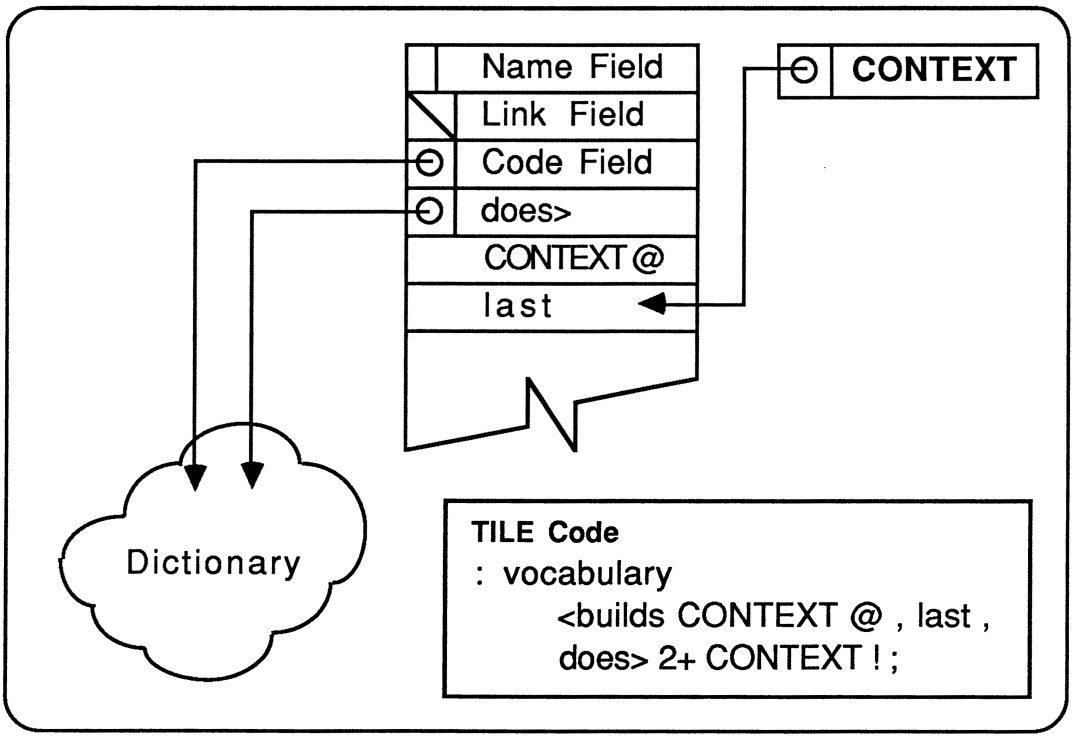

Figure 2.4.2.1 - Diagram of the structure of a vocabulary definition.

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

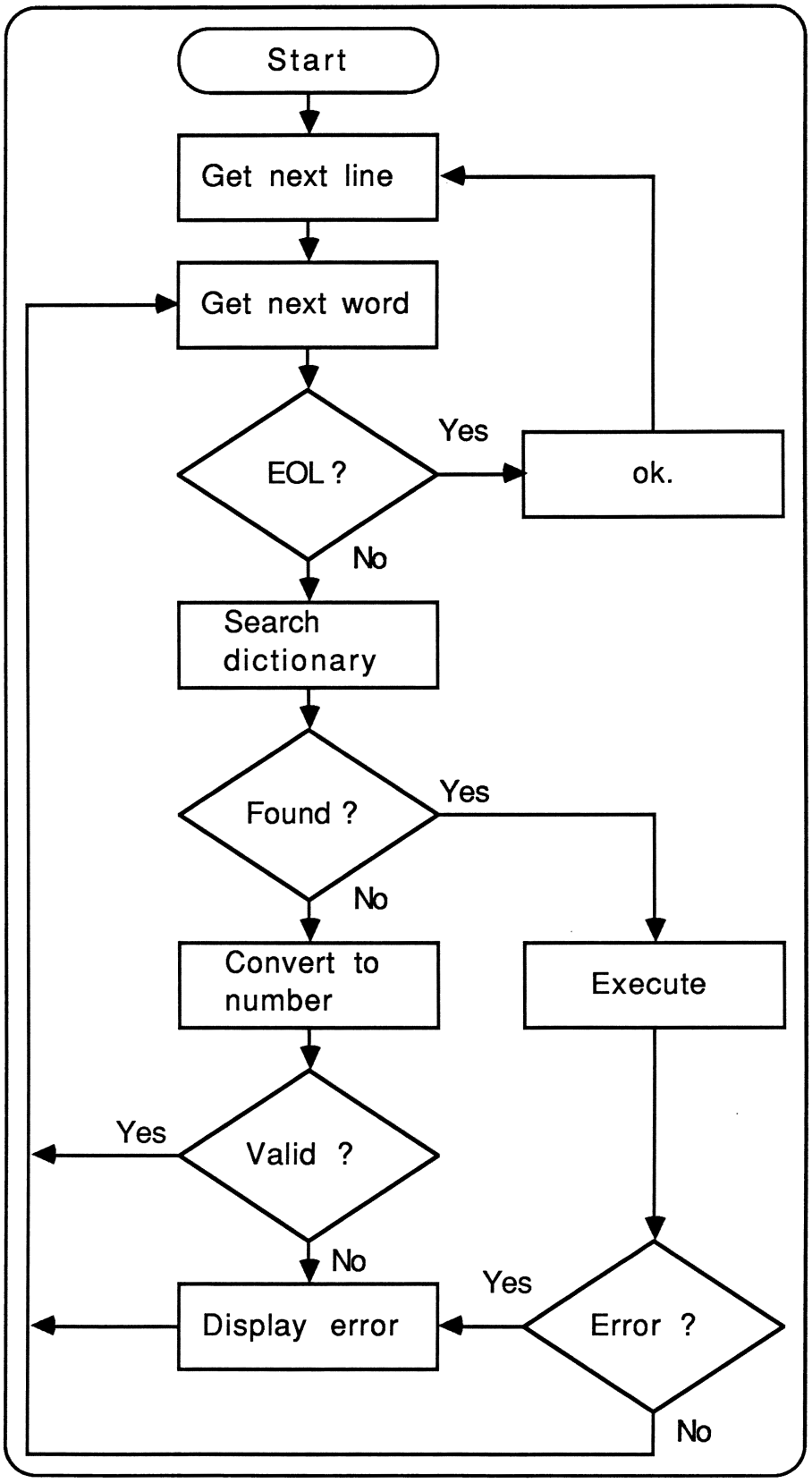

Figure 3.2.1 - Flowchart of the TILE interpreter.

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