CHAPTER 3

Basic *e* Concepts

In this chapter we discuss the basic constructs and conventions in ***e***. These conventions and constructs are used throughout the later chapters. These conventions provide the necessary framework for understanding ***e***. This chapter may seem dry, but understanding these concepts is a necessary foundation for the successive chapters.

Chapter Objectives

- Understand conventions for code segments, comments, white space, numbers, constants, and macros.
- Describe how to import other ***e*** files.
- Define the data types such as scalar type and subtypes, enumerated scalar type, list type, and string type.
- Understand syntax hierarchy of statements, struct members, actions, and expressions.
- Explain the use of simulator variables in ***e***.

3.1 Conventions

e contains a stream of tokens. Tokens can be comments, delimiters, numbers, constants, identifiers, and keywords. ***e*** is a case-sensitive language. All keywords are in lowercase.

3.1.1 Code Segments

A code segment is enclosed with a begin-code marker **<'** and an end-code marker **'>**. Both the begin-code and the end-code markers must be placed at the beginning of a line (leftmost), with

no other text on that same line (no code and no comments). The example below shows three lines of code that form a code segment.

```
<'
    import cpu_test_env;
'>
```

Several code segments can appear in one file. Each code segment consists of one or more statements.

3.1.2 Comments and White Space

e files begin as a comment which ends when the first begin-code marker **<'** is encountered.

Comments within code segments can be marked with double dashes (**--**) or double slashes (**//**).

```
a = 5;          -- This is an inline comment
b = 7;          // This is also an inline comment
```

The end-code **'>** and the begin-code **<'** markers can be used in the middle of code sections, to write several consecutive lines of comment.

```
Import the basic test environment for the CPU...
This is a comment

<'
    import cpu_test_env;
'>

This particular test requires the code that bypasses bug#72 as well as
the constraints that focus on the immediate instructions. This is a
comment

<'
    import bypass_bug72;
    import cpu_test0012;
'>
```

3.1.3 Numbers

There are two types of numbers, sized and unsized.

3.1.3.1 Unsized Numbers

Unsized numbers are always positive and zero-extended unless preceded by a hyphen. Decimal constants are treated as signed integers and have a default size of 32 bits. Binary, hex, and octal constants are treated as unsigned integers, unless preceded by a hyphen to indicate a negative number, and have a default size of 32 bits.

The notations shown in Table 3-1 can be used to represent unsized numbers.

Table 3-1 Representing Unsized Numbers in Expressions

Notation	Legal Characters	Examples
Decimal integer	Any combination of 0-9 possibly preceded by a hyphen - for negative numbers. An underscore (_) can be added anywhere in the number for readability.	12, 55_32, -764
Binary integer	Any combination of 0-1 preceded by *0b*. An underscore (_) can be added anywhere in the number for readability.	0b100111, 0b1100_0101
Hexadecimal integer	Any combination of 0-9 and a-f preceded by *0x*. An underscore (_) can be added anywhere in the number for readability.	0xff, 0x99_aa_bb_cc
Octal integer	Any combination of 0-7 preceded by 0o. An underscore (_) can be added anywhere in the number for readability.	0o66_123
K (kilo: multiply by 1024)	A decimal integer followed by a K or k. For example, 32K = 32768.	32K, 32k, 128k
M (mega: multiply by 1024*1024)	A decimal integer followed by an M or m. For example, 2m = 2097152.	1m, 4m, 4M

3.1.3.2 Sized Numbers

A sized number is a notation that defines a literal with a specific size in bits. The syntax is:

width-number ' (**b|o|d|h|x**) *value-number***;**

The width number is a decimal integer specifying the width of the literal in bits. The value number is the value of the literal and it can be specified in one of four radixes, as shown in Table 3-2.

> **NOTE** If the value number is more than the specified size in bits, its most significant bits are ignored. If the value number is less that the specified size, it is padded by zeros.

Table 3-2 Radix Specification Characters

Radix	Represented By	Example
Binary	A leading 'b or 'B	8'b11001010
Octal	A leading 'o or 'O	6'o45
Decimal	A leading 'd or 'D	16'd63453
Hexadecimal	A leading 'h or 'H or 'x or 'X	32'h12ffab04

3.1.4 Predefined Constants

A set of constants is predefined in ***e***, as shown in Table 3-3.

Table 3-3 Predefined Constants

Constant	Description
TRUE	For boolean variables and expressions
FALSE	For boolean variables and expressions
NULL	For structs, specifies a NULL pointer; for character strings, specifies an empty string
UNDEF	UNDEF indicates NONE where an index is expected
MAX_INT	Represents the largest 32-bit **int** (2^{31} -1)
MIN_INT	Represents the largest negative 32-bit **int** (-2^{31})
MAX_UINT	Represents the largest 32-bit **uint** (2^{32}-1)

3.1.4.1 Literal String

A literal string is a sequence of zero or more ASCII characters enclosed by double quotes (“ “). The special escape sequences shown in Table 3-4 are allowed.

Table 3-4 Escape Sequences in Strings

Escape Sequence	Meaning
\n	New-line
\t	Tab
\f	Form-feed
\”	Quote
\\	Backslash
\r	Carriage-return

This example shows escape sequences used in strings. Although other constructs are introduced here only for the sake of completeness, please focus only on the string syntax.

```
<'
extend sys {

    m() is {
        var header: string = //Define a string variable
          "Name\tSize in Bytes\n----\t-------------\n";
        var p: packet = new;
        var pn: string = p.type().name;
        var ps: uint = p.type().size_in_bytes;
        outf("%s%s\t%d", header, pn, ps);
    };
};
'>
```

The result of running the example above is shown below.

```
Specman> sys.m()
Name    Size in Bytes
----    -------------
packet  20
```

3.1.5 Identifiers and Keywords

The following sections describe the legal syntax for identifiers and keywords.

3.1.5.1 Legal *e* Identifiers

User-defined identifiers in ***e*** code consist of a case-sensitive combination of any length of the characters A-Z, a-z, 0-9, and underscore. They must begin with a letter. Identifiers beginning with an underscore have a special meaning in ***e*** and are not recommended for general use. Identifiers beginning with a number are not allowed.

The syntax of an ***e*** module name (a file name) is the same as the syntax of UNIX file names, with the following exceptions.

- '@' and '~' are not allowed as the first character of a file name.
- '[', ']', '{', '}' are not allowed in file names.
- Only one '.' is allowed in a file name.

3.1.5.2 *e* Keywords

The keywords listed in Table 3-5 below are the reserved words of the ***e*** language. Some of the terms are keywords only when used together with other terms, such as "key" in "**list(key**:*key*)", "before" in "**keep gen** *x* **before** *y*", or "computed" in "**define** *def* **as computed**".

Table 3-5 List of Keywords

all of	**all_values**	**and**	**as a**	**as_a**
assert	**assume**	**async**	**attribute**	**before**
bit	**bits**	**bool**	**break**	**byte**
bytes	**c export**	**case**	**change**	**check that**
compute	**computed**	**consume**	**continue**	**cover**
cross	**cvl call**	**cvl callback**	**cvl method**	**cycle**
default	**define**	**delay**	**detach**	**do**
down to	**dut_error**	**each**	**edges**	**else**
emit	**event**	**exec**	**expect**	**extend**
fail	**fall**	**file**	**first of**	**for**
force	**from**	**gen**	**global**	**hdl pathname**
if	**#ifdef**	**#ifndef**	**in**	**index**
int	**is**	**is a**	**is also**	**is c routine**
is empty	**is first**	**is inline**	**is instance**	**is not a**
is not empty	**is only**	**is undefined**	**item**	**keep**
keeping	**key**	**like**	**line**	**list of**
matching	**me**	**nand**	**new**	**nor**
not	**not in**	**now**	**on**	**only**
or	**others**	**pass**	**prev_**	**print**

range	**ranges**	**release**	**repeat**	**return**
reverse	**rise**	**routine**	**select**	**session**
soft	**start**	**state machine**	**step**	**struct**
string	**sync**	**sys**	**that**	**then**
time	**to**	**transition**	**true**	**try**
type	**uint**	**unit**	**until**	**using**
var	**verilog code**	**verilog function**	**verilog import**	**verilog simulator**
verilog task	**verilog time**	**verilog timescale**	**verilog trace**	**verilog variable**
vhdl code	**vhdl driver**	**vhdl function**	**vhdl procedure**	**vhdl driver**
vhdl simulator	**vhdl time**	**when**	**while**	**with**
within				

3.1.6 Macros

The simplest way to define ***e*** macros is with the **define** statement. An ***e*** macro can be defined with or without an initial ` character.

```
<'
define WORD_WIDTH 16; //Definition of the WORD_WIDTH macro
struct t {
    f: uint (bits: WORD_WIDTH); //Usage of WORD_WIDTH macro
};
'>
```

You can also import a file with Verilog **‘define** macros using the keywords **verilog import**.

```
macros.v (Verilog defines file)
`define BASIC_DELAY   2
`define TRANS_DELAY   `BASIC_DELAY+3
`define WORD_WIDTH 8

-----------------------------------------------------
dut_driver.e (e file)
<'
verilog import macros.v; //Imports all definitions from
                         //macros.v file
//Macros imported from Verilog must be used
//with a preceding ‘.
struct dut_driver {
    ld: list of int(bits: `WORD_WIDTH); //use verilog macro
    keep ld.size() in [1..‘TRANS_DELAY];//use verilog macro
};
'>
```

3.1.7 Importing *e* Files

e files are called modules. An ***e*** file can import another ***e*** file using the **import** keyword. The **import** statement loads additional ***e*** modules before continuing to load the current file. If no extension is given for the imported file name, a ".e" extension is assumed. The modules are loaded in the order they are imported. The **import** statements must be before any other statements in the file.

```
//File Name: pci_transaction_definition.e
<'
type PCICommandType: [ IO_READ=0x2, IO_WRITE=0x3,
                       MEM_READ=0x6, MEM_WRITE=0x7 ];
struct pci_transaction {
    address: uint;
    command: PCICommandType;
    bus_id: uint;
};
'>
//End File: pci_transaction_definition.e
------------------------------------------------------------
//File Name: pci_transaction_extension.e
<'
//Import the file defined above. Note that the .e
//extension is assumed in an import statement
import pci_transaction_definition; //.e extension is the default

extend pci_transaction {
    data: list of uint;
};
'>
//End File: pci_transaction_extension.e
```

If a specified module has already been loaded or compiled, the statement is ignored. For modules not already loaded or compiled, the search sequence is:

1. The current directory
2. Directories specified by the SPECMAN_PATH[1] environment variable
3. The directory in which the importing module resides

3.2 Data Types

This section discusses the basic data types in ***e***.

1. This is an environment variable used by Specman Elite for setting up search paths.

3.2.1 Scalar Types

Scalar types in ***e*** are one of the following:

- Numeric
- Boolean
- Enumerated

3.2.1.1 Numeric and Boolean Scalar Types

Table 3-6 shows predefined numeric and boolean types in ***e***.

Table 3-6 Scalar Types

Type Name	Function	Usage Example
int	Represents numeric data, both negative and non-negative integers. Default Size = 32 bits	length: int; addr: int (bits:24); // 24-bit vector
uint	Represents unsigned numeric data, non-negative integers only. Default Size = 32 bits	delay: uint; addr: uint (bits:16); // 8-bit vector
bit	An unsigned integer in the range 0–1. Size = 1 bit	valid: bit; // 1-bit field
byte	An unsigned integer in the range 0–255. Size = 8 bits	data: byte; // 8-bit field data: uint (bits:8); // Equivalent //definition
time	An integer in the range 0–2^{63}-1. Default Size = 64 bits	delay: time; //64-bit time variable
bool	Represents truth (logical) values, TRUE (1) and FALSE (0). Default Size = 1 bit	frame_valid: bool; //TRUE or //FALSE

3.2.1.2 Enumerated Scalar Types

Enumerated types define the valid values for a variable or field as a list of symbolic constants. For example, the following declaration defines the variable *instr_kind* as having two legal values.

```
<'
//Implicit enumerated type. immediate=0, register=1
type instr_kind: [immediate, register];
'>
```

These symbolic constants have associated unsigned integer values. By default, the first name in the list is assigned the value zero. Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items + 1.

It is also possible to assign explicit unsigned integer values to the symbolic constants. This method is used when the enumerated types may not be defined in a particular order.

```
<'
//Explicit enumerated type. immediate=4, register=8
type instr_kind: [immediate=4, register=8];
'>
```

It is sometimes convenient to introduce a named enumerated type as an empty type.

```
<'
type packet_protocol: []; //Define empty type
'>
```

Once the protocols that are meaningful in the program are identified the definition of the type can be extended.

```
<'
//Extend this type in a separate file.
//No need to touch the original file.
extend packet_protocol : [Ethernet, IEEE, foreign];

//Define a struct that uses a field of the above type.
struct packet {
kind: packet_protocol; //Define a field of type
                       //packet_protocol. Three possible values.
};
'>
```

3.2.1.3 Scalar Subtypes

A subtype can be created from one of the following:

- A predefined numeric or boolean type (**int**, **uint**, **bool**, **bit**, **byte**, **time**)
- A previously defined enumerated type
- A previously defined scalar subtype

Creation of subtypes is shown in the example below.

```
<'
//Define an enumerated type opcode
type opcode: [ADD, SUB, OR, AND];

//Define a subtype using the previously defined type
//This subtype includes opcodes for logical operations
//OR, AND
type logical_opcode: opcode [OR, AND];

//Define a subtype of a predefined scalar type, 4 bit
//unsigned vector
type small: uint(bits:4);

//Define a struct that uses the above types
struct instruction {
op1: opcode; //Field of type opcode
op2: logical_opcode; //Field of type logical_opcode
length: small; //4 bit unsigned vector
};

'>
```

3.2.2 List Types

List types hold ordered collections of data elements. Items in a list can be indexed with the subscript operator [ ], by placement of a non-negative integer expression in the brackets. List indexes start at zero. You can select an item from a list by specifying its index. For example, my_list[0] refers to the first item in the list named my_list. Lists can be of any type. However, a list of lists is not allowed. All items in a list must be of the same type. Lists are dynamically resizable and they come with many predefined methods.

Lists are defined by using the **list of** keyword in a variable or a field definition.

```
<'
struct packet {
addr: uint(bits:8); // 8-bit vector
data1: list of byte; //List of 8-bit values
};
```

```
(continued...)
struct sys {
packets[10]: list of packet; //List of 10 packet structures
values: list of uint(bits:128); //List of 128-bit vectors
};
'>
```

e does not support multidimensional lists (lists of lists). To create a list with sublists in it, you can create a **struct** to contain the sublists, and then create a list of such structs as the main list. In the example above, the packet struct contains a list of bytes. In **sys** struct, there is a list of 10 packets. Thus, **sys** contains a list of lists.

3.2.2.1 Keyed Lists

A keyed list data type is similar to hash tables or association lists found in other programming languages. Keyed lists are defined with the keyword **key**. The declaration below specifies that *packets* is a list of packets, and that the *protocol* field in the *packet* type is used as the hash key. Keyed lists are very useful for searching through a list with a key.

```
<'
type packet_protocol : [Ethernet, IEEE, foreign];
struct packet {
    protocol: packet_protocol;
};
struct dtypes {
    m() is { //Method definition explained later in book
        // Define local variable, keyed list
        var packets : list (key: protocol) of packet;
    };
};
'>
```

3.2.3 The String Type

A string data type contains a series of ASCII characters enclosed by quotes (“ ”). An example of string declaration and initialization is shown below.

```
<'
struct dtypes {
    m() is { //Define a method (procedure)
      var message: string; //Define a variable of type string
      message = "Beginning initialization sequence…";
                          //String value
      print message; //Print string
    };
};
'>
```

3.3 Simulator Variables

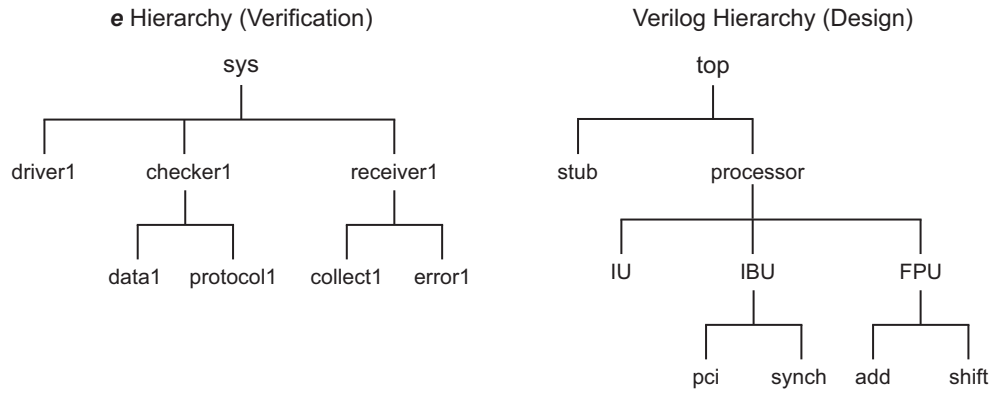
In Chapter 2, we discussed two hierarchies in an ***e***-based environment.

1. The design hierarchy represented in Verilog or VHDL.

2. The verification hierarchy represented in ***e***

An example of the two hierarchies is shown in Figure 3-1 below

Figure 3-1 Verification and Design Hierarchies



Any ***e*** structure should be able to access the simulator variables for reading or writing during simulation. The subsections below explain how to read and write simulator variables.

3.3.1 Driving and Sampling DUT Signals

In ***e***, one can access simulator variables by simply providing the hierarchical path to the variable within single quotes (‘’). The example below shows how to access the design hierarchy shown in Figure 3-1.

```
<'
struct driver{ //Struct in the e environment
r_value: uint(bits:4); //Define a 4 bit field to read

read_value() is { //Define a method to read simulator
                  //variable
   //Right hand side is the simulator variable
   //operand is a variable in module add in Verilog/VHDL
   //The "/" represents the traversal of hierarchy
   //Left hand side is an e field in struct driver
   r_value = '~/top/processor/FPU/add/operand';
};

write_value() is { //Define a method to write simulator
                   //variable
   //Left hand side is the simulator variable
   //Right hand side can be a constant, an e variable,
   //or a simulator variable.
   '~/top/processor/FPU/add/operand' = 7; //Write 7 to variable
};
'>
```

3.3.2 Computed Signal Names

While accessing simulator variables, one can compute all or part of the signal name at run time by using the current value of an ***e*** variable inside parentheses. For example, in Figure 3-1, if there are three processors, *processor_0*, *processor_1,* and *processor_2* instantiated inside top, it should be possible to pick out one of the three processor instances based on an ***e*** variable.

e allows the usage of many parentheses in the computation of a signal name. Thus, it is possible to dynamically choose the signal name at run time based on ***e*** variable values.

```
<'
struct driver{ //Struct in the e environment
id: uint(bits:2); //2 bit ID field determines processor 0,1,2
r_value: uint(bits:4); //Define a 4 bit field to read

read_value() is { //Define a method to read simulator
                  //variable
   r_value = `~/top/processor_(id)/FPU/add/operand';

   //If the id field == 1, then the above assignment
   //will be dynamically set to
   //r_value = `~/top/processor_1/FPU/add/operand';
};

'>
```

3.4 Syntax Hierarchy

Unlike Verilog or C, ***e*** enforces a very strict syntax hierarchy. This is very useful when one is writing or loading ***e*** code. Based on the error messages during loading, it is easy to determine the nature of the syntactical mistake. The strict hierarchy also makes it very difficult to make mistakes.

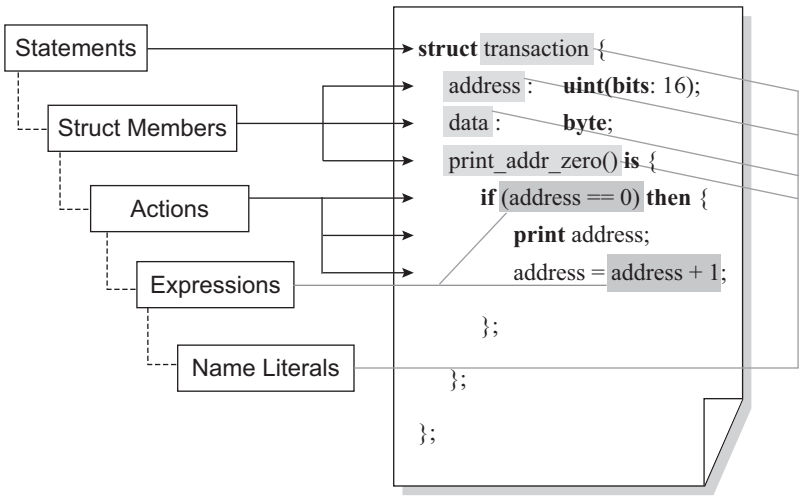
Every ***e*** construct belongs to a construct category that determines how the construct can be used. The four categories of ***e*** constructs are shown in Table 3-7 below.

Table 3-7 Constructs in the Syntax Hierarchy

Statements	Statements are top-level constructs and are valid within the begin-code **<'** and end-code **'>** markers.
Struct members	Struct members are second-level constructs and are valid only within a struct definition.
Actions	Actions are third-level constructs and are valid only when associated with a struct member, such as a method or an event.
Expressions	Expressions are lower-level constructs that can be used only within another ***e*** construct.

Figure 3-2 shows an example of the strict syntax hierarchy.

Figure 3-2 Syntax Hierarchy



The following sections describe each element of the syntax hierarchy in greater detail. Henceforth, any syntactical element in the book will be described as a statement, struct member, action, or expression.

3.4.1 Statements

Statements are the *top-level* syntactic constructs of the ***e*** language and perform the functions related to extending the ***e*** language and interfacing with the simulator.

Statements are valid within the begin-code **<'** and end-code **'>** markers. They can extend over several lines and are separated by semicolons. For example, the following code segment has two statements.

```
<'
    import bypass_bug72; //Statement to import bypass_bug72.e
    type opcode: [ADD, SUB]; //Statement to define
                             //enumerated type
'>
```

Table 3-8 shows the complete list of ***e*** statements.

Table 3-8 *e* Statements

struct	Defines a new data structure.
unit	Defines a new data structure with special properties.
type	Defines an enumerated data type or scalar subtype.
extend	Modifies a previously defined struct or type.
define	Extends the ***e*** language by defining new commands, actions, expressions, or any other syntactic element.
#ifdef, #ifndef	Is used with **define** statements to place conditions on the ***e*** parser.
routine ... is C routine	Declares a user-defined C routine that you want to call from ***e***.
C export	Exports an ***e*** declared type or method to C.
import	Reads in an ***e*** file.
verilog import	Reads in Verilog macro definitions from a file.
verilog code	Writes Verilog code to the stubs file, which is used to interface ***e*** programs with a Verilog simulator.
verilog time	Specifies Verilog simulator time resolution.
verilog variable reg | wire	Specifies a Verilog register or wire that you want to drive from ***e***.
verilog variable memory	Specifies a Verilog memory that you want to access from ***e***.
verilog function	Specifies a Verilog function that you want to call from ***e***.
verilog task	Specifies a Verilog task that you want to call from ***e***.

Table 3-8 *e* Statements (Continued)

vhdl code	Writes VHDL code to the stubs file, which is used to interface ***e*** programs with a VHDL simulator.
vhdl driver	Is used to drive a VHDL signal continuously via the resolution function.
vhdl function	Declares a VHDL function defined in a VHDL package.
vhdl procedure	Declares a VHDL procedure defined in a VHDL package.
vhdl time	Specifies VHDL simulator time resolution.

3.4.2 Struct Members

Struct member declarations are second-level syntactic constructs of the ***e*** language that associate the entities of various kinds with the enclosing **struct** or **unit**.

Struct members can only appear inside a **struct** definition statement. They can extend over several lines and are separated by semicolons. For example, the following struct "packet" has three struct members, *len*, *data,* and a method *m()*.

```
<'
struct packet{
    len: int; //Field of a struct
    data[len]: list of byte; //Field of a struct
    m() is { //Method(procedure) is struct member
    --
    --
    };
};
'>
```

A struct can contain multiple struct members of any type in any order. Table 3-9 shows a brief description of ***e*** struct members. This list is not comprehensive. See Appendix A for a description of all struct members.

Table 3-9 *e* Struct Members

Field declaration	Defines a data entity that is a member of the enclosing struct and has an explicit data type.
Method declaration	Defines an operational procedure that can manipulate the fields of the enclosing struct and access run time values in the DUT.
Subtype declaration	Defines an instance of the parent struct in which specific struct members have particular values or behavior.
Constraint declaration	Influences the distribution of values generated for data entities and the order in which values are generated.
Coverage declaration	Defines functional test goals and collects data on how well the testing is meeting those goals.
Temporal declaration	Defines ***e*** events and their associated actions.

3.4.3 Actions

e actions are lower-level procedural constructs that can be used in combination to manipulate the fields of a struct or exchange data with the DUT. Actions are associated with a struct member, specifically a method, an event, or an "on" struct member. Actions can also be issued interactively as commands at the command line.

Actions can extend over several lines and are separated by semicolons. An action block is a list of actions separated by semicolons and enclosed in curly braces, **{ }**.

Shown below is an example of an action (an invocation of a method, "transmit()") associated with an event, *xmit_ready*. Another action, **out()** is called in the *transmit()* method.

```
<'
struct packet{
    //Declare an event (struct member)
    event xmit_ready is rise('~/top/ready');
    //Declare fields (struct members)
    length: byte;
    delay: uint;
    //Declare an on construct (a struct member)
    //The transmit method is called in the on construct
    //See "On Struct Member" on page 184 for details.
    on xmit_ready {
        transmit(); //Call transmit method (Action)
    };
    //Declare a method (a struct member)
    transmit() is {
      length = 5; //Action that sets value of length
      delay = 10; //Action that sets value of delay
      out("transmitting packet..."); //Action to print
                                      //message
    };
};
'>
```

The following sections describe the various categories of ***e*** actions. These actions can be used only inside method declarations or the **on** struct member. Details on the usage of these actions are not provided in these sections but will be treated in later chapters.

3.4.3.1 Actions for Creating or Modifying Variables

The actions described in Table 3-10 are used to create or modify ***e*** variables.

Table 3-10 Actions for Creating or Modifying Variables

var	Defines a local variable.
=	Assigns or samples values of fields, local variables, or HDL objects.
op	Performs a complex assignment (such as add and assign, or shift and assign) of a field, local variable, or HDL object.

Table 3-10 Actions for Creating or Modifying Variables (Continued)

force	Forces a Verilog net or wire to a specified value, overriding the value driven from the DUT (rarely used).
release	Releases the Verilog net or wire that was previously forced.

3.4.3.2 Executing Actions Conditionally

The actions described in Table 3-11 allow conditional behavior to be specified in ***e***.

Table 3-11 Executing Actions Conditionally

if then else	Executes an action block if a condition is met and a different action block if it is not.
case *labeled-case-item*	Executes one action block out of multiple action blocks depending on the value of a single expression.
case *bool-case-item*	Evaluates a list of boolean expressions and executes the action block associated with the first expression that is true.

3.4.3.3 Executing Actions Iteratively

The actions described in Table 3-12 implement looping in ***e***.

Table 3-12 Executing Actions Iteratively

while	Executes an action block repeatedly until a boolean expression becomes FALSE.
repeat until	Executes an action block repeatedly until a boolean expression becomes TRUE.
for each in	For each item in a list that is a specified type, executes an action block.
for from to	Executes an action block for a specified number of times.
for	Executes an action block for a specified number of times.

Table 3-12 Executing Actions Iteratively (Continued)

for each line in file	Executes an action block for each line in a file.
for each file matching	Executes an action block for each file in the search path.

3.4.3.4 Actions for Controlling Loop Execution

The actions described in Table 3-13 control the execution of loops.

Table 3-13 Actions for Controlling Loop Execution

break	Breaks the execution of the enclosing loop.
continue	Stops execution of the enclosing loop and continues with the next iteration of the same loop.

3.4.3.5 Actions for Invoking Methods and Routines

The actions described in Table 3-14 illustrate the ways for invoking methods (***e*** procedures) and routines (C procedures).

Table 3-14 Actions for Invoking Methods and Routines

method()	Calls a regular method.
tcm()	Calls a TCM.
start *tcm()*	Launches a TCM as a new thread (a parallel process).
routine()	Calls an ***e*** predefined routine.
Calling C routines from ***e***	Describes how to call user-defined C routines.
compute_method()	Calls a value-returning method without using the value returned.
return	Returns immediately from the current method to the method that called it.

3.4.3.6 Time Consuming Actions

The actions described in Table 3-15 may cause simulation time to elapse before a callback is issued by the Simulator to Specman Elite.

Table 3-15 Time Consuming Actions

emit	Causes a named event to occur.
sync	Suspends execution of the current TCM until the temporal expression succeeds.
wait	Suspends execution of the current TCM until a given temporal expression succeeds.
all of	Executes multiple action blocks concurrently, as separate branches of a fork. The action following the **all of** action is reached only when all branches of the **all of** have been fully executed.
first of	Executes multiple action blocks concurrently, as separate branches of a fork. The action following the **first of** action is reached when any of the branches in the **first of** has been fully executed.
state machine	Defines a state machine.

3.4.3.7 Generating Data Items

The action described in Table 3-16 is useful for generating the fields in data items based on specified constraints.

Table 3-16 Generating Data Items

gen	Generates a value for an item, while considering all relevant constraints.

3.4.3.8 Detecting and Handling Errors

The actions described in Table 3-17 are used for detecting and handling errors.

3.4.3.9 General Actions

The actions described in Table 3-18 are used for printing and setting configuration options for various categories.

Table 3-17 Detecting and Handling Errors

check that	Checks the DUT for correct data values.
expect	Expects a certain temporal expression to succeed.
dut_error()	Defines a DUT error message string.
assert	Issues an error message if a specified boolean expression is not true.
warning()	Issues a warning message.
error()	Issues an error message when a user error is detected.
fatal()	Issues an error message, halts all activities, and exits immediately.
try()	Catches errors and exceptions.

Table 3-18 General Actions

print	Prints a list of expressions.
set_config()	Sets options for various categories, including printing.

3.4.4 Expressions

Expressions are constructs that combine operands and operators to represent a value. The resulting value is a function of the values of the operands and the semantic meaning of the operators. A few examples of operands are shown below.

```
address + 1
a + b
address == 0
'~/top/port_id' + 1
```

Expressions are combined to form actions. Each expression must contain at least one operand, which can be:

- A literal value (an identifier)
- A constant
- An ***e*** entity, such as a method, field, list, or struct
- An HDL entity, such as a signal

A compound expression applies one or more operators to one or more operands. Strict type checking is enforced in ***e***.

3.4.5 Name Literals (Identifiers)

Identifiers are names assigned to variables, fields, structs, units, etc. Thus identifiers are used at all levels of the syntax hierarchy, i.e., they are used in statements, struct members, actions, and expressions. Identifiers must follow the rules set in "Legal ***e*** Identifiers" on page 40. In the example below, identifiers are highlighted with comments.

```
<'
struct packet{ //"packet" is an identifier
    %address: uint(bits:2); // "address" is an identifier
    %len: uint(bits:6); //"len" is an identifier
    %data[len]: list of byte; //"data" is an identifier
    my_method() is { //"my_method" is an identifier
    result = address + len; //Identifiers are used in
                            //an expression
    --
    };
};
'>
```

3.5 Summary

We discussed the basic concepts of ***e*** in this chapter. These concepts lay the foundation for the material discussed in the further chapters.

- Conventions for code segments, comments, white space, numbers, constants, and macros were discussed.
- It is possible to import other ***e*** files in an ***e*** file. The **import** statements must always be the first statements in the file.
- ***e*** contains data types such as scalar type and subtypes, enumerated scalar type, list type, and string type.
- Simulator variables can be directly read or written from ***e*** code. One can access simulator variables by enclosing the hierarchical path to the simulator variable in single quotes.
- There is a strict syntax hierarchy of statements, struct members, actions, and expressions. A strict hierarchy enforces coding discipline and minimizes errors.

3.6 Exercises

1. Write a code segment that contains just one statement to import file *basic_types.e*.

2. Determine which comments in the following piece of ***e*** code are written correctly. Circle the comments that are written incorrectly.

```
<'
Import the basic test environment for the CPU...
    import cpu_test_env;
'>

This particular test requires the code that bypasses bug#72 as well as
the constraints that focus on the immediate instructions.

<'
    /*Import the bypass bug file*/
    import bypass_bug72;
    //Import the cpu test file_0012
    import cpu_test0012;
    --Import the cpu test file_0013
    import cpu_test0013;

'>
```

3. Practice writing the following numbers. Use _ for readability.

a. Decimal number 123 as a sized 8-bit number in binary
b. A 16-bit hexadecimal with decimal value 135
c. An unsized hex number 1234
d. 64K in decimal
e. 64K in ***e***

4. Name the predefined constants in ***e***.

5. Write the correct string to produce each of the following outputs.

a. “This is a string displaying the % sign”
b. “out = in1 + in2 \\”
c. “Please ring a bell \t”
d. “This is a backslash \ character\n”

6. Determine whether the following identifiers are legal:

a. system1
b. 1reg
c. ~latch
d. @latch
e. exec[
f. exec@

7. Define a macro *LENGTH* equal to 16.

8. Declare the following fields in ***e***.

a. An 8-bit unsigned vector called *a_in*
b. A 24-bit unsigned vector called *b_in*
c. A 24-bit signed vector called *c_in*
d. An integer called *count*
e. A time field called *snap_shot*
f. A string called *l_str*
g. A 1-bit field called *b*

9. Declare an enumerated type called *frame_type*. It can have values SMALL and LARGE. Declare a field called *frame* of this enumerated type.

10. Write the following actions with simulator variables.

a. Assign the value of simulator variable *top.x1.value* to *len* field in ***e***.
b. Write the value 7 to the simulator variable *top.x1.value*.

11. Describe the four levels of ***e*** syntax hierarchy. Identify three keywords used at each level of the hierarchy.