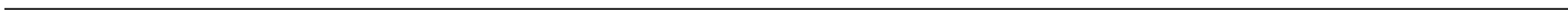
VILLANOVA UNIVERSITY
DEPT. OF ASTRONOMY AND ASTROPHYSICS

PHOEBE Scripter API

PHOEBE version 0.30

Andrej Prša
andrej.prsa@villanova.edu

January 2008



Contents

1 Introduction 1
1.1 The scripter API . . . 2
1.2 What to expect from this document? . . . 2
1.3 Things you need to know . . . 3

2 General properties 4
2.1 PHOEBE invocation . . . 5
2.2 Input files . . . 5
2.3 Formatting rules . . . 5
2.3.1 External scripts . . . 6
2.3.2 Interactive mode . . . 6
2.4 Output redirection . . . 8
2.4.1 Shell redirection . . . 8
2.4.2 Internal redirection . . . 9

3 Working with identifiers 10
3.1 Parameters . . . 10
3.2 Variables . . . 10
3.3 Types of variables . . . 10
3.4 Identifier scope validity . . . 12

4 Arithmetics 13
4.1 Type propagation . . . 17

5 Loops and conditionals 18

6 Directives 22
6.1 General scripter directives . . . 22
6.2 Online help . . . 22
6.3 Interfacing the file system . . . 23
6.4 Arithmetic operations . . . 24
6.5 Working with variables . . . 25
6.6 Working with parameters . . . 26
6.7 User-defined functions and macros . . . 28
6.8 Executing external scripts . . . 30
6.9 Outputting results . . . 31
6.10 Quitting the scripter . . . 31

7 Commands 32
7.1 Input and output files . . . 32
7.2 Getting and setting parameter values . . . 33
7.3 Getting user input . . . 36
7.4 Data transformations . . . 36

7.5 Computing data curves 37
7.6 Handling spectra 38
7.7 Minimization 43

8 Usage examples **49**

A PHOEBE scripter grammar **51**
A.1 Computer language basics 51
A.1.1 Computer language syntax 52
A.1.2 Computer language semantics 53
A.1.3 Abstract syntax trees (ASTs) 54
A.2 Lexing LALR(1) grammars 57
A.3 Parsing LALR(1) grammars 57
A.4 Constructing ASTs in PHOEBE 57
A.5 Symbol tables 57

B Qualifiers **58**
B.1 Model parameters 60
B.2 Data parameters 60
B.3 Physical parameters 62
B.4 Minimization-related parameters 67
B.5 Scripter-related parameters 67

1 Introduction

PHOEBE stands for PHysics Of Eclipsing BinariEs. It is a facility to model eclipsing binary stars as efficiently and as accurately as possible. PHOEBE is divided in two parts:

PHOEBE library:

PHOEBE library is the modeling engine. It consists of many functions that are written to accurately and reliably do tasks related to the modeling of eclipsing binaries and are described by the PHOEBE library specifications. It is *not* an executable, only a library: you need a driver to access those functions. If you want to write your own driver, then the library is all you need. Otherwise you may choose among existing drivers. For modeling back-end PHOEBE uses Wilson-Devinney's WD code.

PHOEBE driver:

To access the functions in PHOEBE library, you need a driver. The driver's task is to make use of library's functions, be it as a self-standing program or an interactive interface. Interfaces are usually referred to as the front-ends. There are currently two front-ends available for PHOEBE: the scripter and the graphical user interface. Both interfaces are being developed simultaneously with the library.

The idea behind PHOEBE is to relieve scientists from spending unreasonable amounts of time learning technical details of the given code implementation and rather use that time for the actual science.

PHOEBE is in the stabilizing phase. This means that the code is expected to run smoothly, it has been tested to some extent and as soon as sufficient confidence is gained that everything works as expected, the stable release will come out. It is thus of utmost importance that you provide feedback to the developers and let us know your opinions and suggestions. The best way to get in touch is via PHOEBE discussion mailing list, where both developers and users share ideas and information on PHOEBE and eclipsing binaries in general.

1.1 The scripter API

The PHOEBE scripter Application Programming Interface (API) book introduces a scripting front-end to PHOEBE library. It features special functions, procedures, macro definitions, full support for arithmetics, loops, conditionals and other constructs to interact with the library. These constructs give the user many advantages over the traditional graphical user interface. Furthermore, it is *fully ANSI-C compliant*, so it is portable and it compiles by virtually every compiler and runs under any platform around.

1.2 What to expect from this document?

PHOEBE scripting language is a new programming language developed specifically for modeling eclipsing binary stars. The scripter is an interactive interface that "speaks" this language. The user communicates with the program by passing *statements* that perform particular actions. A set of statements is called a *script*. Script statements are made of smaller building blocks: *directives* and *commands*. This document thoroughly describes all such building blocks, along with their synopsis and lots of examples. This section introduces the rationale behind PHOEBE and lists some prerequisites for running the scripter comfortably. In Section 2 we shall concentrate on general properties of the scripting language, followed by the discussion on parameter identifiers, user-defined variables and supported types in Section 3. Section 4 is dedicated to arithmetic rules and type propagation. Loops and conditionals will be covered in Section 5, and basic building blocks that make statements will be thoroughly overviewed in Sections 6 and 7. To round up the overview, several usage examples will be given in Section 8. In Appendix A we elaborate on grammatical concepts of the scripting language itself.

Along with description and synopsis, a working example is given for each scripter directive or command. When examples depend on file locations, we shall presume that PHOEBE is run from the its base directory (usually `phoebe-0.30/`). If you run PHOEBE from a different directory, simply replace file paths with the ones that correspond to your directory layout.

1.3 Things you need to know

The aim of this document is to introduce PHOEBE scripting language as thoroughly as possible. For practical reasons we do not cover the details that are not directly connected with the scripter. Throughout the document we thus assume that:

- The reader is familiar with basic concepts of eclipsing binary modeling. Although experience using PHOEBE in the GUI mode is recommended, it is not required to follow this document.
- The reader is comfortable with handling ASCII files (being able to create, read, write and modify them).
- The reader is acquainted with the operating system and knows how to use the command line.

2 General properties

PHOEBE scripting language is an *interpreted* language. This means that scripts need not be compiled before their execution, they are evaluated on the fly. Built-in PHOEBE commands, on the other hand, are pre-compiled, which assures maximum computational speed.

We start the overview of the scripter with few general remarks and term definitions of the scripting language itself.

- A *statement* is the smallest self-contained interpretable piece of input. It begins with a *directive* or a *command* and is followed by zero, one or more arguments. Arguments to directives are passed without any delimeters, while arguments to commands are enclosed in parentheses '(', ')'.

- A *block* of statements is a piece of input that is delimited by curly braces '{', '}'. It is evaluated as a whole rather than on statement-by-statement basis. Blocks may contain sub-blocks, they may be parts of loops, macro definitions or conditional statements. They may also be stand-alone, to achieve segmented evaluation.

- A *script* is a set of statements. Script flow is linear (there are no unconditional jumps commonly known as `goto` sentences) and is interpreted left-to-right, top-to-bottom.

- A *script unit* is a self-contained interpretable block with a void or non-void result. There are 2 script units in PHOEBE: *macros* which don't return any value and *functions* which return a value.

- An *input* is a script or a set of scripts that is read in either interactively from the terminal or from an external file. If the input is interactive, statements are evaluated on statement-by-statement basis. If it is read from an external file, the input is evaluated as a whole.

- A *comment* in the input is denoted by the # character. Everything beyond it until the end of line is discarded.

- A *shell command* is an external command that is executed by the system shell. Shell commands are invoked with the escape character '!'.

Switch:	Mode of operation:
no switch	runs PHOEBE in interactive scripter mode
-c	runs PHOEBE configurator
-h	displays a concise help screen and exits
-v	displays PHOEBE version and exits
-e	executes the script name.script and exits

Table 1: A list of all PHOEBE command-line switches and arguments.

2.1 PHOEBE invocation

Once PHOEBE is properly compiled and installed, it is ready to be run. Mode of operation is determined by command-line switches and arguments. All switches and arguments are optional and order-insensitive. Table 1 gives them in tabular form. PHOEBE invocation synopsis is:

```
phoebe_scripter [-chv] [-e name.script] [paramfile.phoebe]
```

If PHOEBE is run without any arguments, interactive scripter mode will be initiated. That is the main focus of this document.

2.2 Input files

The input consists of alpha-numerical characters (’a’ through ’z’, ’A’ through ’Z’, ’0’ through ’9’), punctuations (’.’, ’,’, ’_’), operators (’+’, ’-’, ’*’, ’/’, ’^’, ’<’, ’>’, ’=’), braces (’{’, ’}’), parentheses (’(’, ’)’, ’[’, ’]’), comments (’#’) and quotes (’'’, ’"’). It may contain an arbitrary number of whitespaces (spaces, tabs and newlines). All other ASCII characters may be used only in literals (enclosed in quotes) and comments (preceeded by ’#’).

2.3 Formatting rules

PHOEBE scripter may be used in two ways: interactively or by interpreting external scripts. In general, script formatting rules are very flexible and they do not require any position-dependent formatting. However, since interactive mode evaluates statements on line-by-line basis, several

context-dependent formatting rules apply. External scripts are evaluated as a whole and there are thus no formatting restrictions of any kind.

2.3.1 External scripts

External scripts are evaluated as a whole, so no formatting restrictions apply. Input file format is very flexible – it is up to the user to choose the layout of a script. Since PHOEBE is insensitive to whitespaces, it does not matter whether a single line corresponds to a single statement or to more statements. The following snippet:

```
print "Entering a loop."
for (i = 1; i <= 5; i++) {
  print "Iteration no. ", i
}
```

is equivalent to:

```
print "Entering a loop." for (i = 1; i <= 5; i++) {
print "Iteration no. ", i }
```

or to:

```
print "Entering a loop." for (i = 1;
      i <= 5;
   i++)
                 {
print "Iteration no. ",
i }
```

Obviously some layouts are more practical than others, but this is completely up to the user to decide which suits him best.

2.3.2 Interactive mode

Formatting rules become context-dependent when line-by-line processing is done. If installed, PHOEBE uses the GNU readline library for line input.

All line editing capabilities and history management are supported by this library. If the library is not installed, the scripter will support only simplest line editing features.

Since the parser is insensitive to whitespaces (including newlines), the scripter must not evaluate the line until a statement is complete. This is done automatically by the lexer: whenever a block start is encountered, the scripter scope increases by 1; whenever a block end is encountered, it decreases by 1. When it reaches 0, all input lines are concatenated and evaluated as a single statement. Consider the following example:

```
 1   > for (i=1; i<=2; i++)
 2   PHOEBE scripter: syntax error
 3   > for (i=1; i<=2; i++) {
 4    1> for (j=1; j<=2; j++) {
 5    2> print "This is scope 2"
 6    2> }
 7    1> print "This is scope 1"
 8    1> }
 9     This is scope 2
10     This is scope 2
11     This is scope 1
12     This is scope 2
13     This is scope 2
14     This is scope 1
15   > # Back in prompt
```

We defer the explanation of the used directives to later sections. Numbers 1–15 on the left represent line numbers and are added for reference only, they are not a part of the scripter output.

Right now we want to focus our attention to scoping. As you see, the first `for` entry (line 1) fails because the parser expects a block to supersede the read input. Since a line ends before the '`{`' delimeter is encountered, syntax error is reported (line 2).

A second `for` entry (line 3) is accepted since a block delimeter '`{`' is encountered and the scope level is increased by 1. To facilitate the user's task of counting scopes, current scope level is printed in the prompt – note '`1>`' and '`2>`' in lines 4–8.

A third `for` entry (line 4) opens a new block and thus increases the

scope level to 2. After the first `print` statement (line 5) is read, the inner block is closed (line 6) and the scope level is reduced to 1 (line 7).

Finally, by line 8 the scope level is reduced to 0. At that point, all statements from lines 3 to 8 will be concatenated into a single statement which will be parsed and evaluated.

Note that system calls mentioned earlier (the `!` escape character) may be used only in interacive mode and that `!` *must* be the first character in the line.

2.4 Output redirection

Any output from PHOEBE scripter may be redirected in two ways: by shell redirection (`sh`, `bash`, `csh` ...) or by internal PHOEBE redirection.

2.4.1 Shell redirection

By default, PHOEBE scripter uses `stdout` stream for output. `stdout` points to the terminal from which PHOEBE was started, but it may be changed with redirection operators '>' and '>>' for writing and appending to an external file, respectively. Consider the following three examples:

```
phoebe_scripter -e example.script
```

This command runs PHOEBE scripter in automatic mode, where the script `example.script` will be executed and all output PHOEBE makes will be directed to the terminal.

```
phoebe_scripter -e example.script > result.data
```

The redirection operator '>' is used to send all output PHOEBE makes to external file `result.data`. If the file exists, it will be overwritten without confirmation.

```
phoebe_scripter -e example.script >> result.data
```

The redirection operator '`>>`' is also used to send all output to external file `result.data`, but this time *appending* it to the contents of that file. If the file doesn't exist, it is created.

2.4.2 Internal redirection

Although redirecting the output using a shell is easy, it is often not quite what is needed: sometimes only parts of the output are relevant and should be output to external files; sometimes it is desireable to redirect different parts to multiple (distinct) external files. This is where internal redirection comes to play. It is similar to shell redirection, only the operators are slightly different: `PHOEBE` uses '`->`' and '`->>`' for writing and appending, respectively. These operators are followed by a literal or a string identifier and may be used after statements, statements in blocks or blocks themselves. In example,

```
calc 1+2 -> "result"
```

will create a new external file `result` or overwrite it if it already exists. Next, a statement

```
calc 3+4 ->> "result"
```

will append the result of the $(3 + 4)$ evaluation to `result`, creating it if it doesn't exist. Both '`->`' and '`->>`' may be grouped, i.e.

```
{
calc 1+2 ->> "result"
calc 3+4 ->> "result"
}
```

would produce the same result as:

```
{
calc 1+2
calc 3+4
} ->> "result"
```

3 Working with identifiers

Very often (if not always) working with actual numbers and literals is not enough to get the job done. Typical scripts are not written to be run only once, so their dependence on actual numbers is not desired. Computing `5!`, for example, could indeed be done by multiplying `5 · 4 · 3 · 2 · 1`, but it would be more practical to create a small script that does the computation for any given value. This is where *identifiers* come to play. Identifiers are sequences of one or more characters that uniquely identify a *parameter* or a *variable.*

3.1 Parameters

Parameters are a special type of identifiers that determine all aspects of the eclipsing binary model. These are, for example, the semi-major axis, the inclination, a set of morphological constraints for the given binary etc. `PHOEBE` uses *qualifiers* to uniquely identify all such parameters. All qualifiers are *predefined*, the complete list along with exact naming specification is given in Appendix B, page 58.

3.2 Variables

Variables are user-defined identifiers that stand for a value. For example, in the expression `a = 3 + b`, `a` and `b` are variables. The opposite of a variable is a *constant.* In the expression above, number `3` is a constant. Every variable has a name, called the *variable name*, and a data *type.* A variable's data type indicates what sort of value the variable represents, such as whether it is an integer, a floating-point number, a string etc.

3.3 Types of variables

`PHOEBE` scripting language defines and uses the following *basic* types of variables:

Integers. This numerical type is composed of one or more digits (`0` through `9`). Any leading zeroes are discarded by the parser. The accuracy of arithmetic operations on integers is infinite.

Booleans. Expressions that may be true or false. `PHOEBE` defines two reserved keywords `TRUE` and `FALSE` (with aliases `YES` and `NO`) that are used for boolean arithmetics.

Boolean arithmetics is conditional and as such doesn't have any accuracy limitations. It should be stressed, however, that comparing real values should never be done by boolean equality testing. For example, testing (`1.23456 == 1.23456`) is very bad, avoid it at all costs. Rather, test (`1.23456 - 1.23456 <` ε) for some reasonable value of ε.

Floating point numbers. This numeric type is composed of one or more leading digits followed by a decimal point ('`.`') and zero or more trailing digits, i.e. `12.5`, `2.`, `0.21`. The leading digit is always necessary; `.2` is thus illegal in `PHOEBE` grammar.

Floating point numbers may also be given in exponential form (the powers of `10`), i.e. `1.3E-3` stands for $1.3 \cdot 10^{-3}$, `0.3E+2` for $0.3 \cdot 10^{2}$ etc.

The accuracy of arithmetic operations on floating point numbers conforms to double `IEEE` standard, which means precision is of the order of 10^{-12} for a single operation.

Strings. This non-numeric type is composed of the arbitrary number of alpha-numeric characters (`a` through `z`, `A` through `Z`, `0` through `9`), a dash (`-`), an underscore (`_`) or a white-space enclosed in quotes (`"`). Strings are always given by their value, so they are also referred to as **literals**. For example, `"This is a literal"`.

Arrays. This type is an arbitrary-dimensional array of floating point numbers. The accuracy of array manipulation is the same as for floating point numbers, conforming to double `IEEE` standard (10^{-12}).

In addition to these basic data types, variables may also have *structured* types. Structured types are *read-only* constructs with predefined fields of any basic data type. Structures cannot be constructed or modified by the user; they have a fixed layout and content that is set by a structure-returning command. For example, minimization algorithms return a structure containing a list of χ^2 values for each passband, a weighted cost function of the whole fit, original and corrected parameter values etc. We shall introduce these structured types when discussing the respective commands that return them.

3.4 Identifier scope validity

There are two practical alternatives how to handle variables in interpreted languages: to have automatic variables that are valid throughout the script unit or to have declared variables and their validity governed by local scopes. PHOEBE implements the former alternative: the type of the variable is determined from the assignment context and its validity pertains to the script unit (a function or a macro) in which it was defined.

Example:

```
set i=1
set d=5.2
set a={1,2,3}
```

The `set` directive is used to assign the value to the variable; in the example above we set an integer value `1` to the variable `i`, a real value `5.2` to the variable `d` and a 3-dimensional array `{1,2,3}` to the variable `a`. This means that the type of the variable is automatically determined from the context. All variables `i`, `d` and `a` will remain valid throughout the script unit or until the `unset` directive is used to undefine them.

Script unit names are stored in a global symbol table, which means that they are valid throughout the input. This is desired, since this way macros and functions are globally defined and may be used in any other script unit. Parameters are identified by the corresponding qualifiers, which are also stored in a global symbol table. The last globally valid identifiers are mathematical and physical constants given in Table 5.

4 Arithmetics

PHOEBE arithmetics closely follows standard programming notation and operator associativity/precedence rules. Table 2 gives a complete list of all supported operators sorted by precedence from lowest to highest.

Operation:	Symbol:	Associativity:	Example:
assignment	=	right	i = 2
addition	+	left	1 + 2
subtraction	-	left	1 - 2
multiplication	*	left	1 * 2
division	/	left	1 / 2
unary + sign	+	left	+ 2
unary − sign	-	left	- 2
exponentiation	^	right	1 ^ 2

Table 2: A list of all PHOEBE scripter binary and unary operators. The table is sorted by the operators' precedence from lowest to highest.

Associativity determines the way PHOEBE scripter reduces given expressions. Consider the following example:

$$a\ op\ b\ op\ c.$$

If the operator *op* is left-associative, this means that the scripter will reduce the above expression like this:

$$(a\ op\ b)\ op\ c.$$

In contrast, if the operator *op* is right-associative, the scripter will reduce it to:

$$a\ op\ (b\ op\ c).$$

The associativity importance could be easily demonstrated on the assignment operator:

$$a = b = c.$$

If the '=' operator was left-associative, the above statement would reduce to:

$$(a = b) = c,$$

which would imply that a should get the value of b and then the value of c, rendering b useless. But since the = operator is right-associative, we have:

$$a = (b = c),$$

which means that b will get the value of c and then a will get the value of b.

The precedence of operators is determined by the "strength" of the operator; the multiplication operator `*` should preceed the addition operator `+`, but it should be preceeded by the exponentiation operator `^` (see Table 2). The presedence may be enforced by using parentheses, e.g. $(a + b) * c$. Parentheses always reduce the expression within prior to anything that follows and/or preceeds them.

For evaluation, all standard boolean operators are supported. All boolean operators are left-associative; see Table 3 for a detailed list. Boolean operators return `TRUE` if the condition is met and `FALSE` if the condition is not met.

Since `PHOEBE` scripter supports arithmetics on variables, symbolic operators are also available. Unary symbolic operators are left-associative, while binary symbolic operators are right-associative. See Table 4 for details.

Most important constants are added for convenience and uniformity of calculations. Declared as variables, their name starts with prefix '`CONST_`'. Constants are listed in Table 5.

`PHOEBE` scripter also features most basic mathematical functions, which are listed in Table 6. If you need other functions, you may build them yourself by using the `define` directive (see page 28).

The basic principle `PHOEBE` is built on is *generality*. Thus same arithmetic operators are used for different types of variables. For example, a '`+`' operator will add two integers to an integer, two floating point numbers to a floating point number, two arrays to a new array and concatenate two strings into a new string. Table 7 lists the results of the binary operators' evaluation on different data types.

Operation:	Symbol:	Associativity:	Example:
equal-to	`==`	left	`1 == 1`
not-equal-to	`!=`	left	`1 != 2`
less-or-equal-to	`<=`	left	`1 <= 2`
less-than	`<`	left	`1 <  2`
greater-than	`>`	left	`2 >  1`
greater-or-equal-to	`>=`	left	`2 >= 1`
logical and	`&&`	left	`a && b`
logical or	`||`	left	`a || b`
logical negation	`!`	left	`!(a && b)`

Table 3: The complete list of all `PHOEBE` scripter boolean operators.

Operation:	Symbol:	Associativity:	Example:
unary increment	`++`	left	i++
unary decrement	`--`	left	i--
increment-by	`+=`	right	i += 5
decrement-by	`-=`	right	i -= 3
multiply-by	`*=`	right	i *= 2
divide-by	`/=`	right	i /= 3

Table 4: The complete list of all `PHOEBE` scripter symbolic operators.

Constant identifier:	Symbol:	Value:
`CONST_PI`	π	3.14159265359
`CONST_E`	e	2.71828182846
`CONST_AU`	au	149 597 870.691 [km]
`CONST_RSUN`	$R_\odot$	696000 [km]
`CONST_MSUN`	$M_\odot$	$1.99 \cdot 10^{30}$ [kg]

Table 5: The complete list of all built-in mathematical and physical constants. If greater accuracy is required, simply redeclare these constants to a higher-precision number.

Function	$\mathcal{D}_f$:	$\mathcal{Z}_f$:	Description:	Example:
`sin(x)`	$\mathbb{R}$	$[-1,1]$	sine [rad]	`sin (0.5)`
`cos(x)`	$\mathbb{R}$	$[-1,1]$	cosine [rad]	`cos (0.5)`
`tan(x)`	$\mathbb{R}$	$\mathbb{R}$	tangent [rad]	`tan (0.5)`
`asin(x)`	$[-1,1]$	$[-\pi/2,\pi/2]$	arc sine	`asin (0.479)`
`acos(x)`	$[-1,1]$	$[0,\pi]$	arc cosine	`acos (0.877)`
`atan(x)`	$\mathbb{R}$	$[-\pi/2,\pi/2]$	arc tangent	`atan (0.464)`
`exp(x)`	$\mathbb{R}$	$\mathbb{R}^+$	e raised to x	`exp (1.0)`
`ln(x)`	$\mathbb{R}^+$	$\mathbb{R}$	natural logarithm	`ln (2.5)`
`log(x)`	$\mathbb{R}^+$	$\mathbb{R}$	decade logarithm	`log (2.5)`
`sqrt(x)`	$\mathbb{R}_0^+$	$\mathbb{R}_0^+$	square root	`sqrt (9)`
`rand(x)`	$\mathbb{R}$	$[0,x]$	random number	`rand (1)`
`trunc(x)`	$\mathbb{R}$	$\mathbb{Z}$	truncated number	`trunc (3.6)`
`round(x)`	$\mathbb{R}$	$\mathbb{Z}$	rounded number	`round (3.6)`
`int(x)`	$\mathbb{R}$	$\mathbb{R}$	integral part	`int (5.4)`
`frac(x)`	$\mathbb{R}$	$[0,1]$	fractional part	`frac (5.4)`
`abs(x)`	$\mathbb{R}$	$\mathbb{R}_0^+$	absolute value	`abs (-5.4)`
`norm(x)`	$\mathbb{R}^N$	$\mathbb{R}$	norm	`norm ({1,2})`
`dim(x)`	$\mathbb{R}^N$	$\mathbb{N}_0$	dimension	`dim ({1,2})`
`strlen(x)`	strings	$\mathbb{N}_0$	string length	`strlen ("blah")`
`isnan(x)`	$\mathbb{R}$	boolean	not-a-number test	`isnan(sqrt(-1))`

Table 6: The complete list of all basic mathematical operators `PHOEBE` scripter supports. $\mathcal{D}_f$ is the definition range (for which values the function may evaluate) and $\mathcal{Z}_f$ is the functional range (which values the function may assume).

4.1 Type propagation

Strict grammar rules demand that arguments of binary and assignment operators we have discussed so far must be of the same type. That would mean that adding an integer `2` to a floating point number `2.5` would not be possible. Since this strict formalism is too strict to be convenient, we allow for well-defined special cases where type propagation is performed. This means that, before adding `2` and `2.5`, the type of `2` is propagated from integer to a floating point type, `2` $\rightarrow$ `2.0`. Now the addition may be evaluated strictly, i.e. `2.0 + 2.5` to give `4.5`. Table 7 lists evaluated types for all binary and comparison operators.

+	**I**	**B**	**D**	**A**	**S**
I	I	–	D	A	S
B	–	–	–	–	–
D	D	–	D	A	S
A	A	–	A	A	–
S	S	–	S	–	S

-	**I**	**B**	**D**	**A**	**S**
I	I	–	D	A	–
B	–	–	–	–	–
D	D	–	D	A	–
A	A	–	A	A	–
S	–	–	–	–	–

*	**I**	**B**	**D**	**A**	**S**
I	I	–	D	A	–
B	–	–	–	–	–
D	D	–	D	A	–
A	A	–	A	A	–
S	–	–	–	–	–

/	**I**	**B**	**D**	**A**	**S**
I	D	–	D	A	–
B	–	–	–	–	–
D	D	–	D	A	–
A	A	–	A	A	–
S	–	–	–	–	–

%	**I**	**B**	**D**	**A**	**S**
I	I	–	–	–	–
B	–	–	–	–	–
D	–	–	–	–	–
A	–	–	–	–	–
S	–	–	–	–	–

^	**I**	**B**	**D**	**A**	**S**
I	I	–	D	A	–
B	–	–	–	–	–
D	D	–	D	A	–
A	A	–	A	A	–
S	–	–	–	–	–

==	**I**	**B**	**D**	**A**	**S**
I	B	–	B	–	–
B	–	–	–	–	–
D	B	–	B	–	–
A	–	–	–	B	–
S	–	–	–	–	B

!=	**I**	**B**	**D**	**A**	**S**
I	B	–	B	–	–
B	–	–	–	–	–
D	B	–	B	–	–
A	–	–	–	B	–
S	–	–	–	–	B

<=	**I**	**B**	**D**	**A**	**S**
I	B	–	B	–	–
B	–	B	–	–	–
D	B	–	B	–	–
A	–	–	–	–	–
S	–	–	–	–	–

>=	**I**	**B**	**D**	**A**	**S**
I	B	–	B	–	–
B	–	B	–	–	–
D	B	–	B	–	–
A	–	–	–	–	–
S	–	–	–	–	–

<	**I**	**B**	**D**	**A**	**S**
I	B	–	B	–	–
B	–	–	–	–	–
D	B	–	B	–	–
A	–	–	–	–	–
S	–	–	–	–	–

>	**I**	**B**	**D**	**A**	**S**
I	B	–	B	–	–
B	–	–	–	–	–
D	B	–	B	–	–
A	–	–	–	–	–
S	–	–	–	–	–

&&	**I**	**B**	**D**	**A**	**S**
I	–	–	–	–	–
B	–	B	–	–	–
D	–	–	–	–	–
A	–	–	–	–	–
S	–	–	–	–	–

||	**I**	**B**	**D**	**A**	**S**
I	–	–	–	–	–
B	–	B	–	–	–
D	–	–	–	–	–
A	–	–	–	–	–
S	–	–	–	–	–

Table 7: Results of the evaluation of binary operators on different types of variables. **I** stands for integers, **B** for booleans, **D** for double precision reals, **A** for arrays and **S** for strings. The first argument type of the binary operator is given vertically and the second argument type is given horizontally. For discussion on `PHOEBE` types please refer to Section 3.3, pg. 10.

5 Loops and conditionals

A scripting language without support for loops and conditionals would not be very useful. PHOEBE scripter features two standard loop functions – `while` and `for` statements – and two conditionals – `if` and `if-else` statements.

`while (condition) { statements }`

The `while` loop is the most basic loop that can be constructed. The condition is any boolean expression that is reduced to a simple `TRUE` or `FALSE`. A block of statements will be evaluated for as long as condition is `TRUE`. As soon as the condition evaluates to `FALSE`, the loop is finished and the program continues with a normal flow.

The `while` loop can be nested any number of times. Consider the following example:

```
set i=1
while (i<5) {
  set i=i+1
  set j=i
  while (j < 6) {
    calc sqrt (j+1)
    set j++
  }
}
```

The loop condition may be composed of any number of expressions, given that the reduced value is boolean. So conditions like:

```
while (a < 4)
while ( (a > 2) && (a < 6) )
while ( a > b && b > c && c > 5 )
```

are all legal.

```
for ( initialization; condition; action ) { statements }
```

The `for` loop has an expanded functionality with respect to the `while` loop. The synopsis above could be rewritten by using the `while` loop like this:

```
initialization
while (condition) {
  statements
  action
}
```

The `initialization` expression consists of an index variable initialization of the form:

```
var = value
```

The `condition` is any boolean expression. In case the condition is `TRUE`, the block of statements will be evaluated; if it is not, the loop will end and the program flow continues outside the loop.

The `action` is an expression that is evaluated after each successive loop. Usually it is something like `i++`, which increments the index variable `i` by one. Consider the following example:

```
for (i = 0; i <= 5; i++) {
  print "i = ", i
}
```

Just as a `while` loop, `for` loops may be nested to any depth. An example of nested `for` statements is:

```
for (i = 0; i <= 5; i++) {
  for (j = 0; j <= i; j++) {
    for (k = 0; k <= j; k++) {
      calc (i+j)^k
    }
  }
}
```

`if (condition) { statements }`

The `if` statement is the simplest form of conditional program flow branching. The `condition` is any expression that evaluates to a boolean value (either `TRUE` or `FALSE`). If the condition evaluates to `TRUE`, the block of statements will be evaluated, otherwise it won't.

Example:

```
set i=1
if (i < 2) {
  set j = i
  print "i is less than 2"
  if (j == i) {
    print "i is equal to j"
  }
}
```

`if (condition) { statements } else { statements }`

The `if-else` statement allows the user to specify actions for both outcomes of the condition evaluation. Much like the simple `if` statement, the `condition` gets evaluated to a boolean value. If the value is `TRUE`, the first block of statements will be executed. If it is `FALSE`, the second block of statements will be executed.

A well-known problem of a *dangling-else* structure is avoided by enforcing the usage of curly braces. Consider the following example:

```
if (a < b)
    if (b < c)
        print "Both are true"
  else
  print "To which 'if' does this apply?"
```

This code demonstrates the dangling-else problem - to which `if-else` structure does the `else` part belong – the first or the second? Since `PHOEBE` enforces the usage of braces, the above code would be written like this:

```
if (a < b) {
  if (b < c) {
    print "Both are true"
  } else {
    print "It applies to the 2nd statement"
  }
} else {
  print "It applies to the 1st statement"
}
```

This way there is no ambiguity on expense of writing curly braces for each `if` and `if-else` conditional statement.

6 Directives

Basic user interaction with the scripter is done by *directives*. Directives are simple instructions that form statements. They always appear at the beginning of a statement.

6.1 General scripter directives

Directives that are used to set or change the layout of the scripter are called general scripter directives.

`clear`

The `clear` directive clears the terminal. In order for this directive to work, you must use an ANSI-compliant terminal, since the scripter uses the ANSI escape sequence to clear the screen. Fortunately, most terminals of today are ANSI-compliant – e.g. `xterm`, DOS, ... This directive has influence only if the scripter is run in interactive mode (see Section 2.3.2 for details) and default screen output.

Example:

```
clear
```

6.2 Online help

This Section lists directives used to get online help. Online help is not meant to replace this document, only to give quick information on directive and command synopsis and usage.

`help [directive/command]`

The `help` directive is used whenever you want to see the syntax of a particular directive or command, the number of arguments it takes, ... You may use `?` as `help` alias. Pass a directive or a command to `help` to get information on its usage. The output is written in three fields:

Synopsis, which explains the usage format,
Summary, which gives a concise summary and a
Description, which elaborates on usage.

Examples:

```
help
help help
help open_parameter_file
? open_parameter_file
```

6.3 Interfacing the file system

When run interactively, PHOEBE scripter takes control over the file system. This Section introduces directives that are used for interfacing the file system and for executing shell commands from within the scripter.

```
! shell_command
```

The '!' character is used to escape to the underlaying shell and to execute the shell command. It may be any type of command or external program that the file system has access to.

Examples:

```
!ls
!gcalc &
```

```
ls  [dirname]
dir [dirname]
```

These two directives list all files within the current directory. Please note that the current implementation is Linux-specific, use the system call `!dir` under other operating systems.

Examples:

```
ls
ls /
dir ../data
```

`cd [dirname]`

The `cd` directive changes a current working directory to the directory passed as argument. If no argument is given, `cd` will change to user's home directory defined by the shell variable `$HOME`If `GNU readline` library is found, name completion using the `[TAB]` key will be available.

Examples:

```
cd
cd /
cd phoebe-0.30
```

`pwd`

This directive prints the location of the current working directory.

Examples:

```
pwd
```

6.4 Arithmetic operations

`calc expression`

The `calc` directive is used to perform all kinds of arithmetic operations: addition, multiplication, functional evaluation, . . . Refer to Section 4, page 13 for details on arithmetics.

Examples:

```
calc 1+2
calc 1.2e2-(2.1-4*0.3)^2 / 12.3
calc sin(3.1415926) - cos(2-1.3/2.0)
```

6.5 Working with variables

```
set var = expression
set var = array (dimension)
set var = phoebe_command (command_args)
```

This directive assigns a value to the given variable. Variables may assume numeric values (integers and double precision real numbers), boolean values (`TRUE` or `FALSE`), strings or arrays. Once the variable is assigned, it may be used in place of its corresponding value.

Before variable assignment takes place, the `expression` is reduced to its simplest form. Thus the expression `set i=1+2` is first reduced to `set i=3`, then the assignment is performed.

Arrays of any type (integer, boolean, real or string) may be defined either by their contents or by their dimension. Since there is no limit on array dimension, initializing it to an empty array and then filling in the values with a loop is much more convenient than having to type all elements by hand. That is why arrays may also be defined by the `array()` function.

Examples:

```
set i=1
set j = i
set a = (i+j)^3 - sqrt (12.3e-1)
set name = "The value" + " of j is " + j
set arr = array(3)
set arr[1] = 1
set arr2 = {2, 3, 4}
set chi2 = minimize_using_nms (1e-3, 0)
```

`unset var`

This directive releases the variable contents and removes it from the symbol table. It can be called from any scope, the variable that **first** occurs in the symbol table will be freed.

Examples:

```
set i=1
stdump       # Stack dump
unset i=1
stdump       # Stack dump
```

6.6 Working with parameters

PHOEBE refers to its parameters by the corresponding qualifiers. The scripter features two directives that enable the user to inspect parameters without having to declare any variables. To see how values of parameters may be assigned to variables, please refer to Section 7.2, page 33.

`show qualifier`

This directive shows the value of the parameter, which is referenced by the corresponding qualifier. Appendix B gives a complete listing of all PHOEBE parameters.

Example:

```
show phoebe_lcno
show phoebe_hla
```

`info qualifier`

This directive displays all relevant information about the passed parameter: its description, qualifier, type and value. If the parameter is adjustable, all minimization parameters are also outputted: whether it is marked TBA (To Be Adjusted), which are its lower and upper limits and what is its step size. All information is given in tabular form, as an example for `phoebe_hla` qualifier is shown in Table 8.

```
> info HLA

  Description:    LC primary star flux leveler
  Qualifier:      phoebe_hla
  Type:           double array
  Value:          {9.095000, 15.630000}
  Adjustable:     yes
  | Marked TBA:   no
  | Step size:    0.010000
  | Lower limit:  0.000000
  | Upper limit:  10000000000.000000
```

Table 8: An example of PHOEBE scripter directive `info HLA` output for UV Leonis data.

Examples:

```
info phoebe_lcno
open_parameter_file ("../data/UVLeo.phoebe")
info phoebe_lcno
info phoebe_hla
```

`list reqlist`

The `list` directive prints the requested list on screen. The following

requests are supported:

`parameters`	all qualifiers defined in PHOEBE
`qualifiers`	alias for `parameters`
`tba`	parameters marked for adjustment

Examples:

```
list parameters
```

6.7 User-defined functions and macros

This Section explains how to define and use functions and macros in PHOEBE. The only difference between the two is that functions return a numerical value and macros don't.

```
define userfunc () { statements }
define userfunc (arg1, arg2, ..., argN) { statements }
```

Use the `define` directive to form your own numerical functions. Passed arguments are enclosed in parentheses; even if the function takes no arguments, empty parentheses are still required.

The value of user-defined functions is set by the `return` statement. If the value is not returned or is invalid (non-numeric), void will be returned, which will abort further computation.

User-defined functions are evaluated as any other expression, e.g. by using the `calc` or `print` directive. They may be nested any number of times.

Examples:

```
set PI=3.1415926
define deg (angle) { return angle * 180 / PI }
define rad (angle) { return angle / 180 * PI }
calc deg (rad (45))
calc sin (rad (30))

define r (x, y, z) {
  set r_squared = x^2 + y^2 + z^2
  set r = sqrt (r_squared)
  return r
}
calc r (1, 2, 3)
```

`macro macroname (arg1, arg2, ..., argN) { statements }`

The `macro` directive is used to define a logical block of code. A *macro* is an arbitrary set of statements that doesn't return any numeric value. If you need the function to return a value (so as to be used in expressions), take a look at the `define` directive above.

Macro names may be constructed with all alpha-numerical characters (`a..z`, `A..Z`, `0..9`) and the underscore character ('_'). All other characters are not valid. The number of passed arguments is arbitrary; if the macro doesn't depend on any argument, empty parentheses are still necessary.

Macros are executed by their names, without any leading directive. If the macro's name is `runme ()`, then it would simply be run by issuing `runme ()` at the prompt.

Macros may be nested any number of times. They may call other macros and functions, they may be built with loops and conditionals and they may interact with the user.

Examples:

```
macro sumpot (a, b, c) {
  print "a = ", a, ", b = ", b, ", c = ", c
  for (i=1; i<=5; i++) {
    print "(a+b+c)^i = ", (a+b+c)^i
  }
}
sumpot (1, 2, 3)

macro nms (tol, maxiter) {
  minimize_using_nms (tol, maxiter)
}
execute nms (1e-3, 0)
```

6.8 Executing external scripts

Usually scripts are written in external files and then executed interactively from the scripter. This allows easy script manipulation and testing. Directives in this Section are used for executing external scripts.

```
execute "scriptname"
```

The `execute` directive is used to execute an external script. The passed argument is either a literal filename or a string variable containing a filename.

Example:

```
open_parameter_file ("../data/UVLeo.phoebe")
!ls ../scripts
execute "../scripts/find_preliminary_solution.script"
```

6.9 Outputting results

This Section presents directives for outputting text and/or results to the terminal or the output file.

```
print arg1 [, arg2, ..., argN]
```

The `print` directive is used for printing out messages to standard output (e.g. the terminal, the redirected filename, ...). The directive accepts one or more arguments, which must be either literal strings (enclosed in quotes) or expressions returning a numerical type. Several arguments are separated with the ',' delimeter. As explained in Section 2.4.2 on page 9, `print` directive may be used with internal redirection operators '->' and '->>'.

Examples:

```
print "This is", " ", "PHOEBE", " example"
set i=3.5
print "i = ", i, " and 3*(i+2) = ", 3*(i+2)
print "An example of output redirection" -> test.out
```

6.10 Quitting the scripter

```
quit
```

This directive closes the interactive scripter. Prior to quitting, you are asked to confirm this. If you are calling the scripter from a file, you don't have to explicitly call quit, the end-of-file marker will do it for you. Also, you may use `CTRL+D` to quit without confirmation.

Example:

```
quit
```

7 Commands

Scripter commands are built-in functions to manage, process and evaluate specific actions. Arguments to scripter commands are passed by value or by variable name in parentheses.

7.1 Input and output files

PHOEBE stores the values of parameters in *parameter* files. PHOEBE parameter files are special input files consisting of qualifier = value statements. They are insensitive to white-spaces, newlines and comments (lines beginning with the # character). Parameter files are simple ASCII files which PHOEBE may generate for you, but you may wish to write or edit them yourself. The consecutive order of parameter entries within a parameter file is arbitrary. To each parameter there is a corresponding qualifier. A full list of all qualifiers is given in Appendix B.

open_parameter_file (filename)

This script command opens a PHOEBE parameter file. The argument must be either a literal (a string enclosed in quotes) or a variable with a literal value. You may pass both relative and absolute pathnames.

Example:

```
open_parameter_file ("UVLeo.phoebe")
# The statement above is equivalent to
# the following usage of a variable:
set file="UVLeo.phoebe"
open_parameter_file (file)
```

save_parameter_file (filename)

This script command saves current parameters to a PHOEBE parameter file. If the file doesn't exist, it will be created and if it does exist, it will be overwritten. The passed argument may be either a literal or a variable with a literal value.

Examples:

```
open_parameter_file ("UVLeo.phoebe")
mark_for_adjustment (phoebe_incl, 1)
save_parameter_file ("UVLeo-new.phoebe")
```

`create_wd_lci_file (filename, mpage, curve)`

Sometimes you may want to call Wilson–Devinney program directly, e.g. testing different WD versions, checking on consistency etc. `PHOEBE` scripter offers this command to create a raw LCI file. The first passed argument is the lci filename, which has to be enclosed in quotes. The second argument is WD's `MPAGE` parameter, which tells WD what to calculate. Please note that only `MPAGE=1` (light curves) and `MPAGE=2` (radial velocities) are currently supported. The last, third argument is the curve you wish to create the LCI file for.

Example:

```
open_parameter_file ("UVLeo.phoebe")
create_wd_lci_file ("phoebe_defaults.lci", 1, 1)
```

7.2 Getting and setting parameter values

`PHOEBE` defines an exhaustive set of physical parameters related to eclipsing binaries. Each parameter is identified by its *qualifier*. A qualifier is a reserved identifier, which is used to reference values of parameters. A full list of all `PHOEBE` qualifiers is given in Appendix B. For discussion on `PHOEBE` parameters behind qualifiers please refer to `PHOEBE` User manual.

`set var = get_parameter_value (qualifier)`

This command defines the variable `var` to contain the value of the given `PHOEBE` qualifier. The type is always inherited from the type of the queried

parameter. If a qualifier is passband-dependent, `ident` will become an array. Array elements are referenced by '[' and ']' delimeters, e.g. `ident[1]`, `ident[2]`, ...

The command adheres to all arithmetic rules for expressions, listed in Section 4.

Examples:

```
set filename = "UVLeo.phoebe"
open_parameter_file (filename)
set lcno = get_parameter_value (phoebe_lcno)
print "RV number in ", filename, " is ",
                         get_parameter_value (phoebe_rvno)
```

```
set_parameter_value (qualifier, value)
set_parameter_value (qualifier, value, curve)
```

To set a value of the given parameter, call this command with parameter qualifier as the first argument and its value as the second parameter. If the parameter is passband-dependent (e.g. each curve has its own value), pass the curve number as the third argument. Instead of both numeric value and curve index, a variable may be passed.

Examples:

```
set_parameter_value (phoebe_lcno, 1)
set_parameter_value (phoebe_hla, 8.0, 1)
set phsv=5.0
set_parameter_value (phoebe_phsv, phsv)
```

```
set_parameter_limits (qualifier, valmin, valmax)
```

Adjustable parameters may be fitted either by the unconstrained, semi-constrained or fully constrained minimizer. To set the allowed boundaries of a particular parameter, use this command. Values `valmin` and `valmax`

may be any expressions of real type. Note that `valmax` should be larger than `valmin`.

Example:

```
info phoebe_sma
set_parameter_limits (phoebe_sma, 5.0, 10.0)
info phoebe_sma
```

```
set_parameter_step (qualifier, step)
```

Use this function to declare a step size for the given adjustable parameter that will be used by the minimizer. The value of `step` may be any expression of real type, positive or negative.

Example:

```
info phoebe_sma
set_parameter_step (phoebe_sma, 1e-3)
info phoebe_sma
```

```
set levarray = compute_light_levels ()
set level = compute_light_levels (curve_index)
```

This command computes light levels by minimizing the sum of squares of the residuals. It is meant to replace the light level parameter and, by providing analytic approach to computing it, make its adjustment obsolete. The command takes one optional argument, curve index; if it is omitted, light levels are computed for all defined light curves, otherwise it is computed for the passed light curve.

Example:

```
set hlas = compute_light_levels ()
set lcno = get_parameter_value (phoebe_lcno)
for (i = 1; i <= lcno; i++)
  set_parameter_value (phoebe_hla, i, hlas[i])
```

7.3 Getting user input

It sometimes proves convenient in PHOEBE functions and macros to prompt the user for the given argument instead of insisting that all arguments must be passed by the calling statement. For this reason PHOEBE implements a special command that achieves this.

```
set var = prompt (prompt_string [, expected_type])
```

This command writes out the `prompt_string` and prompts the user for input. The input is then parsed and its value is then stored in a variable `var`.

Currently, `prompt` can parse four basic types of input: integers, booleans, real values and literals. The type is automatically inherited from the parser. If only a given type is allowed, an optional second argument `expected_type` may be used. It can be one of the following: `"integer"`, `"boolean"`, `"double"`, `"string"`. The command will loop until proper type of input is supplied or until `CTRL+D` (empty line) is encountered.

Examples:

```
set answer = prompt ("Is this correct? [y/n] ", "boolean")
if (answer == TRUE) { print "This is correct!" }
else                { print "This is not correct." }

set i = prompt ("Enter a number: ", "double")
set j = prompt ("Enter another number: ", "double")
print "Their sum is ", i+j
```

7.4 Data transformations

Commands within this specialized set perform different operations on arrays exclusively. Their primary purpose is to manipulate PHOEBE *array* variables. They may be regarded as extensions of basic arithmetic operations covered in 4, page 13.

The most notable exception of these commands is the synopsis: they are to be used with the `set` directive (see page 25 for details). The typical synopsis thus looks something like this:

```
set new = command_name (old, params [, optional params])
```

The `new` and `old` identifiers are arrays (data structures) and parameters are any types of expressions.

```
transform_hjd_to_phase (in [, hjd0, period, dpdt, pshift])
```

This command acts on independent variable - it transforms heliocentric Julian date (HJD) to ephemeris phase. Optional arguments are ephemeris arguments: the origin of HJD (`hjd0`), binary star period in days (`period`), first time derivative of period dP/dt (`dpdt`) and an arbitrary phase shift (`pshift`). If any of optional arguments is missing, `PHOEBE` will take parameter table values.

Examples:

```
open_parameter_file ("../data/UVLeo.phoebe")
set hjd = column ("../data/UVLeo.B", 1)
set ph1 = transform_hjd_to_phase (hjd)
get_parameter_value (hjd0, phoebe_hjd0)
set ph2 = transform_hjd_to_phase (hjd, hjd0 + hjd0/100)
```

7.5 Computing data curves

The following commands generate output files with computed (synthetic) light and radial velocity curves.

```
set deps = compute_lc (indeps, curve)
```

Use this command to compute synthetic light curve in phase points passed as elements of the array `indeps`. Argument `curve` is the index of the light curve that will be computed. The return value is an array of the same dimension as the array `indeps`.

Examples:

```
open_parameter_file ("UVLeo.phoebe")
set indeps = {-0.2, -0.2, -0.1, 0.0, 0.1, 0.2, 0.3}
set deps = compute_lc (indeps, 1)
for (i = 1; i <= dim(indeps); i++) {
  print indeps[i], " ", deps[i]
}
```

`set deps = compute_rv (indeps, curve)`

Use this command to compute synthetic radial velocity curve in phase points passed as elements of the array `indeps`. Argument `curve` is the index of the radial velocity curve that will be computed. The return value is an array of the same dimension as the array `indeps`.

Examples:

```
open_parameter_file ("UVLeo.phoebe")
set indeps = array (100)
for (i = 1; i <= 100; i++) {
  set indeps[i] = -0.5 + i/100
}
plot_using_gnuplot (indeps, compute_rv (indeps, 1))
```

7.6 Handling spectra

`PHOEBE` features a basic environment for handling spectral energy distribution (SED) data. Support is still rudimentary, but the basic functionality is already implemented.

A spectrum is characterized by the following parameters:

Spectral range: the wavelength interval $[\lambda_l, \lambda_u]$, where λ_l and λ_u are lower and upper wavelength limits, respectively.

FWHM: the full-width-half-maximum of the instrument response to a monochromatic source. It is related to the standard deviation by the following relation: FWHM$= 2\sigma\sqrt{2\log 2}$.

True resolution: the ratio $\mathcal{R}$ between the wavelength λ and the FWHM of the spectrum.

Sampling: wavelength spacing $\Delta\lambda$ between sample points in the spectrum.

Sampling resolution: the ratio R between the wavelength λ and the sampling step $\Delta\lambda$.

Sampling is independent of the true resolution and is usually something like 2–5 times the FWHM. This parameter determines the number of sampling points on the wavelength interval. `PHOEBE` preserves the sampling resolution throughout the wavelength range, so the size of $\Delta\lambda$ depends on the wavelength: $\Delta\lambda = \lambda/R$.

We proceed by introducing the commands implemented in `PHOEBE` for handling spectra.

```
set_spectra_repository (dir_name)
```

This command sets the directory `dir_name` as a synthetic spectra repository. The repository must contain precomputed spectra with (at least) two columns: wavelength and spectral energy distribution (SED). The spectra are identified by filenames, which must be in the following format:

$$\texttt{F}\underbrace{\texttt{2500}}_{\lambda_{\rm min}}\underbrace{\texttt{10500}}_{\lambda_{\rm max}}\underbrace{\texttt{V050}}_{v_{\rm rot}}-\underbrace{\texttt{R20000}}_{R}\underbrace{\texttt{P00}}_{[\mathrm{M/H}]}\underbrace{\texttt{T6000}}_{T_{\rm eff}}\underbrace{\texttt{G45}}_{\log g/g_0}\texttt{K2NOVER.ASC}$$

Here $\lambda_{\rm min}$ and $\lambda_{\rm max}$ are the lower and upper wavelength interval limits in angstroms, respectively, $v_{\rm rot}$ is the rotational velocity in $\mathrm{km\,s^{-1}}$, $R = \lambda/\Delta\lambda$ is the resolution, [M/H] is the metallicity in Solar abundances, $T_{\rm eff}$ is effective temperature in K and $\log g/g_0$ is the gravitational acceleration at the star's surface. Keyword `K2NOVER` means that the overshooting effect is not taken into account. There must be no whitespace characters in filenames.

Examples:

```
set_spectra_repository ("/media/cdrom")
```

```
set var = get_spectrum_from_repository (T, log g/g0, [M/H], vrot)
```

This command queries the repository set by the `set_spectra_repository` command and obtains the spectrum that corresponds to the passed parameters by linear interpolation. The arguments are effective temperature `T`, gravity acceleration `log g/g0`, metallicity `[M/H]` and rotational velocity projection `vrot`.

Examples:

```
set s1 = get_spectrum_from_repository (5800, 4.3, 0.0, 10)
```

```
set var = get_spectrum_from_file (filename)
```

This command reads in the spectrum contained in an external file `filename`. The file must contain at least two columns: the first one is the wavelength and the second one is the flux. Empty lines, commented lines and lines with invalid format will be silently discarded.

Examples:

```
set s1 = get_spectrum_from_file ("spectrum.data")
```

```
set var = broaden_spectrum (spectrum, R)
```

This command broadens the spectrum to the true resolution `R`. It keeps the original sampling of `spectrum`. The broadening preserves the true resolution over the whole wavelength range, so for low-resolution spectra ($R \leq 10$) the deviation from pure Gaussian correction is noticeable. This command applies *only* the broadening due to the change of true resolution, it does *not* apply any physical line-broadening corrections like Doppler or Stark broadening.

Examples:

```
# Get R=50000, ll=2500, ul=10500 spectrum from repository:
set s1 = get_spectrum_from_repository (5800, 4.3, 0.0, 10)
# Broaden it to R=20000:
set s2 = broaden_spectrum (s1, 20000)
```

```
set var = resample_spectrum (spectrum, ll, ul, R)
```

This command resamples the spectrum variable `spectrum` to the wavelength interval [`ll`, `ul`] and resolution `R`. Resampling is done on the sub-bin level so that the contents of original bins are fairly distributed between the newly sampled bins. The command may thus be issued to resample the spectrum to a lower resolution or higher resolution, or to retain the original resolution, but to modify the wavelength range.

Examples:

```
# Get R=50000, ll=2500, ul=10500 spectrum from repository:
set s1 = get_spectrum_from_repository (5800, 4.3, 0.0, 10)
# Broaden it to R=20000 (with the same sampling):
set s2 = broaden_spectrum (s1, 20000)
# Crop and resample it to R=20000 (change the sampling):
set s3 = resample_spectrum (s1, 6000, 7000, 20000)
```

```
set var = apply_doppler_shift (spectrum, vr)
```

Use this command to apply a Doppler shift to the spectrum variable `spectrum` that corresponds to the radial velocity `vr` passed in $\mathrm{km\,s^{-1}}$. The value of `vr` is positive for a redward Doppler shift (receding sources) and negative for a blueward Doppler shift (approaching sources). This command *changes* the wavelengths of bins; if you wish to preserve original bins and redistribute Doppler-shifted SED according to the shift, simply resample the Doppler-shifted spectrum to the original wavelength interval and resolution by using the `resample_spectrum` command.

Example:

```
# Get R=50000, ll=2500, ul=10500 spectrum from repository:
set s1 = get_spectrum_from_repository (5800, 4.3, 0.0, 10)
# Apply the blueward Doppler shift of 120 km/s:
set s2 = apply_doppler_shift (s1, -120)
# Resample the spectrum to the original bin layout:
set s3 = resample_spectrum (s2, 2500, 10500, 50000)
```

```
set var = add_spectra (s1, s2 [, w1, w2])
```

This command adds the two spectra `s1` and `s2`. If the corresponding

weight factors `w1` and `w2` are present, they are used to weight individual spectra. If they are omitted, `w1` = `w2` = 0.5 is assumed.

Example:

```
# Get two spectra from the repository:
set s1 = get_spectrum_from_repository (5800, 4.3, 0, 20)
set s2 = get_spectrum_from_repository (6200, 4.5, 0, 30)
# Add them with two different weights:
set s3 = add_spectra (s1, s2)
set s4 = add_spectra (s1, s2, 0.3, 0.7)
```

`set var = convolve_spectra (s1, s2 [, ll, ul, R])`

This command convolves the two spectra `s1` and `s2`. If the optional wavelength interval limits `ll` and `ul`, and the resolution `R` are passed, the convolved spectrum will be resampled to that wavelength interval and that resolution, otherwise it will retain the default values of the repository spectra, namely `ll` = 2500, `ul` = 10500 and `R` = 50000.

Example:

```
# Get the spectrum from the repository:
set s1 = get_spectrum_from_repository (5800, 4.3, 0, 20)
# Get the filter transmission function from a file:
set s2 = get_spectrum_from_file ("johnsonB.data")
# Convolve the two:
set s = convolve_spectra (s1, s2)
```

`set var = integrate_spectrum (spectrum [, ll, ul])`

This command integrates the spectrum variable `spectrum`. If the optional arguments `ll` and `ul` are passed, the integration is done on the wavelength interval [`ll`, `ul`], otherwise the whole wavelength range is used.

Example:

```
# Get the spectrum from the repository:
```

```
set s1 = get_spectrum_from_repository (5800, 4.3, 0, 20)
# Get the filter transmission function from a file:
set s2 = get_spectrum_from_file ("johnsonB.data")
# Convolve the two:
set s = convolve_spectra (s1, s2)
# Integrate it on the whole wavelength range:
set int = integrate_spectrum (s)
```

`plot_spectrum_using_gnuplot (spectrum [, ll, ul])`

This command uses GNUPlot to plot the spectrum variable `spectrum`. If the optional arguments `ll` and `ul` are passed, the plotting is done on the wavelength interval [`ll`, `ul`], otherwise the whole wavelength range is used.

Example:

```
# Get the spectrum from the repository:
set s = get_spectrum_from_repository (5800, 4.3, 0, 20)
# Plot it on the whole wavelength range:
plot_spectrum_using_gnuplot (s)
# Plot it around Halpha:
plot_spectrum_using_gnuplot (s, 6520, 6620)
```

7.7 Minimization

To find the best solution for the given data-set, PHOEBE uses several minimization algorithms to solve the inverse problem for eclipsing binaries. The cost function used for minimization is a passband-combined, weighted χ^2 value, where weighting is done by the passband weight.

All minimizers return a predefined *structured* type (c.f. Section 3.3, page 11) called a *minimizer feedback structure*. This generic structure has the following fields:

Field name:	Field description
algorithm	Name of the minimization algorithm
iters	Number of performed iterations
cputime	Computational time cost in seconds
pars	A list of adjusted parameters
initvals	A list of initial parameter values
newvals	A list of converged parameter values
ferrors	A list of formal parameter errors
chi2s	A list of passband χ^2 values
wchi2s	A list of weighted passband χ^2 values
cfval	Cost function value (master χ^2)

To access the fields of a structure, a '`.`' delimeter is used. For example, if `result` is the variable of this structured type and we wish to access its field `algorithm`, `result.algorithm` would be used.

Minimization commands never update the parameter values automatically; converged values should always be previewed (manually or automatically) and the values adopted by the `adopt_minimizer_results ()` command.

All minimization methods in PHOEBE can automatically compute passband luminosities (L_1^i or HLAs) for light curves and center-of-mass velocity (v_γ) for RV curves. Both L_1^i and v_γ are simple scale and shift parameters and as such may be easily computed instead of being minimized. This decreases the time-cost of the method while improving the quality of the fit. To enable or disable this automatic computation, use `phoebe_compute_hla_switch` and `phoebe_compute_vga_switch` qualifiers. For detailed discussion of this point please refer to (Prša & Zwitter 2005).

`mark_for_adjustment (qualifier, switch)`

To mark certain parameter for adjustment[1], use this command. If the parameter is to be adjusted, set `switch` to `TRUE` or simply `1`, otherwise set it to `FALSE` or `0`.

[1]Sometimes this is referred to as setting the TBA bit on or off, where TBA stands for To Be Adjusted.

Examples:

```
open_parameter_file ("UVLeo.phoebe")
mark_for_adjustment (phoebe_incl, TRUE)
mark_for_adjustment (phoebe_sma, 1)
mark_for_adjustment (phoebe_hla, NO)
```

`set var = compute_chi2 ([curve_index])`

This command computes the weighted χ^2 value of the passband data. If an optional argument `curve_index` is passed, χ^2 value of only that passband is computed, otherwise χ^2 values for all available passbands are computed and the array is returned.

Examples:

```
open_parameter_file ("UVLeo.phoebe")
set chi2s = compute_chi2 ()
print chi2s
print compute_chi2 (1)
```

`set result = minimize_using_nms (tolerance, max_iters)`

This command uses Nelder & Mead's downhill Simplex (NMS) minimizer. This is the multidimensional algorithm that converges steadily to the nearest minimum. With proper heuristics it can be very powerful. The passed argument `tolerance` determines the required accuracy for convergence. Argument `max_iters` is the number of maximum iterations to be allowed during minimization. Passing `max_iters = 0` will not limit the number of iterations (i.e. the number of iterations will be determined by the passed tolerance). The return value of this command is a minimizer feedback structure.

Since NMS is a derivativeless method, it can *never* diverge, but it is significantly slower than other algorithms. Using NMS is a good choice for seeking a good starting point for DC, which needs to be close to the minimum.

Examples:

```
open_parameter_file ("data/UVLeo.phoebe")
mark_for_adjustment (phoebe_incl, 1)
set result = minimize_using_nms (1e-3, 0)
print result
```

`set result = minimize_using_dc ()`

This command uses WD's Differential Corrections (DC) powered by the Levenberg-Marquardt scheme to perform the minimization. It is extremely fast, specifically adapted for eclipsing binaries. Since it is based on numerical derivatives, it may diverge in situations when the discrepancy between the model and the data is significant. In such cases, common when automatic handling is involved, NMS minimizer is preferred for initial steps. Note however that DC is significantly faster, so it will perform better than NMS for final-touch analysis.

DC minimizer doesn't take any arguments. It returns a minimizer feedback structure.

Examples:

```
open_parameter_file ("data/UVLeo.phoebe")
mark_for_adjustment (phoebe_incl, 1)
set result = minimize_using_dc ()
print result
```

`adopt_minimizer_results (result)`

This command takes a minimizer feedback structure `result` and applies the corrections to the parameter table. It does not in any way depend on the minimizer that was used to obtain the corrections.

Example:

```
open_parameter_file ("data/UVLeo.phoebe")
mark_for_adjustment (phoebe_incl, 1)
set initial_chi2s = compute_chi2 ()
set r = minimize_using_dc ()
if (r.chi2s < initial_chi2s) {
    adopt_minimizer_results (r)
}
else {
    print "Cost function value increased, aborting."
}
```

`kick_parameters (sigma)`

One way of getting true statistics from the minimizer is using heuristic scans, but that may be truly lengthy if more than a couple of parameters are fitted. For that reason PHOEBE implements another method that enables the minimizers to jump out of local minima: parameter kicking.

Whenever a minimum is reached by any minimizer to a given accuracy, it is useful to estimate the miminum's depth. Consider for a moment that standard deviations (σ_k) of observations are estimated properly so we decide to use them for χ^2 weighting:

$$\chi_k^2 = \sum_i w_k (x_i - \bar{x})^2 = \frac{1}{\sigma_k^2} \sum_i (x_i - \bar{x})^2. \tag{1}$$

Since the variance is given by:

$$s_k^2 = \frac{1}{N_k - 1} \sum_i (x_i - \bar{x})^2, \tag{2}$$

we may readily express χ^2 as:

$$\chi_k^2 = (N_k - 1) \frac{s_k^2}{\sigma_k^2}. \tag{3}$$

and the overall χ^2 value as:

$$\chi^2 = \sum_k (N_k - 1) \left(\frac{s_k}{\sigma_k} \right)^2. \tag{4}$$

If σ_k are fair, then the ratio s_k/σ_k is of the order unity and χ^2 of the order $N \cdot M$, where $k = 1 \dots N$ and $i = 1 \dots M$. This we use for quantizing χ^2 values:

$$\chi_0^2 \sim N \cdot M, \quad \left(\chi^2/\chi_0^2\right) \quad \text{used for quantization.} \tag{5}$$

This quantization measures the depth of the minimum. Parameter kicking is a way of punching the obtained parameter-set out of the minimum: using the Gaussian probability density function (PDF), the method randomly picks an offset for each parameter. How large should the offset be? That is determined by the kick's dispersion σ. Its value depends on the depth of the minima - if we are in a high minimum, then the kick should be strong, but if we are very low, i.e. below $(\chi^2/\chi_0^2) = 1$, then only subtle perturbations are admissable. The user constructs the rules of calculating σs for the kick himself; one suggested approach could be to use the following expression:

$$\sigma_{\text{kick}} = 0.5 \frac{(\chi^2/\chi_0^2)}{100}. \tag{6}$$

This causes σ_{kick} to assume a value of 0.5 for 10σ offsets and 0.005 for 1σ offsets, being linear in between. Please note that this σ_{kick} is *relative*, i.e. given by:

$$\sigma_{\text{kick}}^{\text{rel}} = \frac{\sigma_{\text{kick}}^{\text{abs}}}{\text{parvalue}}. \tag{7}$$

Example:

```
open_parameter_file ("data/UVLeo.phoebe")
mark_for_adjustment (phoebe_incl, 1)
set result1 = minimize_using_simplex (1e-2, 0)
# Two datasets, each having 300 vertices:
set sigma = 0.5 * result1.cfval / 600.0 / 100.0 * 0.5
kick_parameters (sigma)
set result2 = minimize_using_simplex (1e-3, 0)
```

8 Usage examples

This Section gives a couple of small PHOEBE scripts along with a line-by-line explanation to get you quickly started using the scripter. Each script has enumerated lines; these numbers are *not* a part of the script, they're added to more easily reference lines in the text.

```
1     # Calculate color index of the model. For UV Leonis,
2     # the two passbandes are Johnson B and V.
3     open_parameter_file ("UVLeo.phoebe")
4     hla = get_parameter_value (phoebe_hla)
5     define bvindex (hla1, hla2) { return -5/2 * log (hla1/hla2) }
6     calc bvindex (hla[1], hla[2])
```

Lines 1 and 2 are comments, since they start with the comment character `#`. Line 3 is the command that opens PHOEBE parameter file `"UVLeo.phoebe"`, which resides in the current working directory. Fourth line reads the flux level (HLA) values in an array `hla`. Since PHOEBE knows the type of `phoebe_hla` qualifier, the type gets assigned to `hla` automatically. The fifth line defines a new function called `bvindex`. It takes two arguments, `hla1` and `hla2`. Finally, the last line calls the `bvindex` function via the `calc` directive, which calculates the color index. The arguments to `bvindex` are the first and the second `hla` array element, denoted with an index enclosed between '`[`' and '`]`'.

References

[1] Kernighan, Pike, 1999: The Practice of Programming.

[2] Prša, A., & Zwitter, T. 2005, ApJ, 628, 426

A PHOEBE scripter grammar

This appendix covers the basics of language and compiler programming. Since the language formalism is not widely known in the astrophysical community, the basics as well as distinct features of PHOEBE scripter implementation are given here.

PHOEBE scripter language is a strict
context-free LALR(1) interpreted language.

This appendix aims to explain the above statement.

A.1 Computer language basics

A computer language is a construct much like spoken languages: it has a grammar which defines syntactic and semantic rules along with any exceptions that may be allowed.

A program written in any computer language has to be executed. There are two ways of achieving this: by *compiling* the source into a machine code or by *interpreting* the source statement-by-statement. A typical example for a compiler would be translating ANSI C source into Intel Pentium architecture machine code; an example for an interpreter would be the Unix shell interpreter (bash, csh, ...), which runs operating system commands interactively. Compilers are thus machine-dependent and faster, whereas interpreters are machine-independent and slower. Following this definition, PHOEBE scripter is an interpreter: it interprets statements and blocks one-by-one. That is why from now on we limit our discussion to interpreters.

The most important principle compilers and interpreters are based on is portability. The usual practice for interpreters is to perform their job in multiple phases:

1. **scanning**: a scanner maps input characters into tokens;
2. **parsing**: a parser recognizes sequences of tokens according to the language grammar and translates them into a universal, machine-independent *Intermediate representation* (IR). IRs are tree-like data structures

with nodes and branches defining actions. PHOEBE implements a particular example of an IR called the *Abstract syntax tree* (AST), which will be thoroughly explained in Section A.1.3;

3. **semantic analysis**: this phase performs type checking (whether the variables, functions and arguments in the source program are used in accordance with the language grammar);
4. **optimization**: the final front-end phase is optimizing the IR – removing redundant loops, unrolling `for` sentences etc.
5. **interpretation**: IR is executed by the interpreter.

Before we get into details about particular interpretation phases, let us define computer language building blocks that form the grammar.

A.1.1 Computer language syntax

A *syntax* of the spoken language gives the structure of *sentences*:

> "*The established rules of usage for arrangement of the words of sentences into their proper forms and relations.*"

The New Webster's Dictionary, 1989

Computer languages are indeed pretty similar: given the set of words, a *syntax* gives rules how these words may be arranged in *syntactic groupings*. A syntactic grouping is a bit broader term than a *sentence* in a way that even simpler constructs may be regarded as a grouping. Consider the following example of a syntactic rule for a grouping called `expression`:

```
expression : NUMBER
expression :            '-' expression
expression : expression '+' expression
```

We interpret this rule[2] as follows:

[2]Please note that the notation of the above example is not significant in any way, we use it to demonstrate how a syntactic rule could be composed. We defer the analysis of syntactic rule notation to A.3.

1. An `expression` may be constructed by using a `NUMBER`, e.g. a real value composed of digits and a floating point.
2. An `expression` may be constructed of a unary minus sign '`-`' and another `expression`.
3. An `expression` may be constructed of an expression, a binary plus sign '`+`' and another `expression`.

According to the 1st rule, a valid `expression` would be, for example, number `2.5`; so would a number `3.2`. Following the 2nd and 3rd rules, valid `expression`s are e.g. `- 2.5` or `2.5 + 3.2`. Since these rules are obviously recursive, valid `expression` is also `- 2.5 + 3.2 + - 2.5`. Note however that the numbers `2.5` or `3.2` or the operators `+` and `-` have no meaning whatsoever for syntactic rules – they are just symbols that are grammatically valid to form an `expression`, nothing else. Syntactic groupings don't even have to make sense in a language – it is up to *semantics* to take care of this.

A.1.2 Computer language semantics

Semantics of the spoken language gives meaning to *words*:

> "*The study of word meanings, especially as they develop and change. The study of the relationship between signs or symbols and that which they represent.*"

The New Webster's Dictionary, 1989

Again, we find close correspondence in computer languages: semantic analysis of a syntactic grouping defines its *meaning* and determines semantic *action* to be performed based on that meaning. To clarify this, let us write semantic rules for the syntactic grouping `expression` of the former example:

```
expression: NUMBER
```

$\longrightarrow$ read in the number to lhs expression

```
expression:            '-' expression
```

⟶ evaluate rhs expression and apply unary `-`

```
expression: expression '+' expression
```

⟶ evaluate both rhs expressions and apply binary `+`

(lhs and rhs stand for left-hand-side and right-hand-side, respectively).

Let us now analyse these semantic rules. If the grouping is recognised by the first rule, it means that a real value is read. As we mentioned earlier, the value itself bears no significance and it is up to semantic rule to *assign* its contents to `expression`. Returning to our former example, the number `2.5` is just a *numeric type* as it enters semantic analysis, but as it is analysed by the 1st rule, the *value* of `expression` now becomes `2.5`.

The 2nd rule instructs the right-hand-side `expression` to be *evaluated.* To evaluate means to perform the same semantic analysis recursively on that `expression` until the *simplest possible representation* is resolved. In our case, the simplest possible representation is `NUMBER`. The 2nd rule now *acts*: apply unary `-` to it. By this action the reduction of the right-hand-side is complete and a *new* value to the left-hand-side `expression` is assigned.

The 3rd rule is very similar to the 2nd: both right-hand-side `expressions` are evaluated to their simplest representation and a binary `+` operator is applied. This action reduces the right-hand-side to the simplest representation and assigns a new value to the left-hand-side `expression`.

So how are these actions executed? By keeping in mind that the execution must be independent of any underlaying programming language or any particular implementation, the execution lays on *abstract syntax trees*.

A.1.3 Abstract syntax trees (ASTs)

Ever been working with a Reverse Polish Notation (RPN) calculator? E.g. most Hewlett-Packard calculators are known to implement RPN. If you

had, you will find that the concept of abstract syntax tree (AST) is nothing else than the operational stack. But before we explain what is an AST and how is it constructed, let us briefly discuss the parsing flow.

Say we want to analyse the following statement syntactically and semantically by using rules of our previous example:

```
1.5 + - 2.5 + - 3.0
```

If we are to apply rules properly, we must consider *all* possible reduction channels to obtain the simplest possible representation for our statement. Figure 1 shows a diagram showing all reduction channels for a single recursion depth. To such a diagram we refer to as the *Direct Syntax Tree* (DST). Each square is a node containing an expression (denoted by `E`), a numeric value (denoted by `N`) or an operator (denoted by `-` and `+` for unary `-` and binary `+`, respectively). Each node may have one or more branches; inspecting Figure 1 shows that only simplest possible representation nodes don't have any branches, but form closed *leaf*s. By evaluating each node recursively, our goal is to reduce the flow to a single leaf.

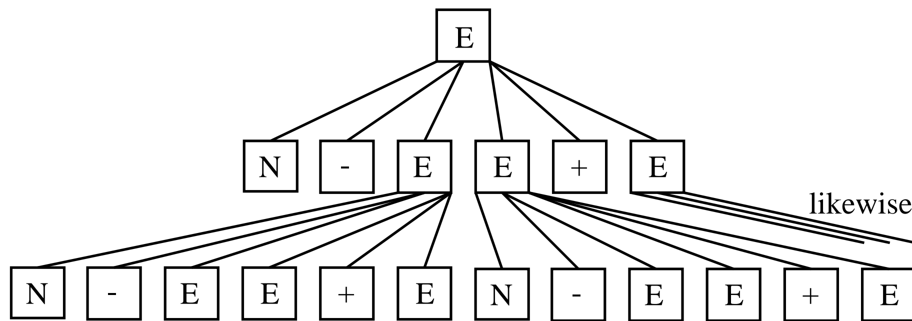


Figure 1: A Direct Syntax Tree (DST). The rightmost channel is omitted for figure brevity, but it should be easy to work out the tree's full form. Squares are called *tree nodes*. Letter `E` stands for `expression`, letter `N` for `NUMBER` and operator symbols for operations. The flow is bottom-up recursive.

It it obvious that the DST in Figure 1 doesn't suffice to reduce our statement: since it is composed of two additions and two unary negations, we would need *four* tree levels instead of just one. The gain of using DSTs is that we may interpret *linear* sets of statements in a single pass. The downside of DSTs is not only inconvenience for flow-charts; their evaluation gets more and more complex with new rules and deeper recursions used and the flow *must* be linear – hence no loops, conditionals, gotos etc.

Fortunately, there is a better way; instead of insisting on a single-pass interpreter, we build a two-pass interpreter. In the first pass we build a complete tree based on sytactic and semantic rules in an *abstract* way, covering only the reduction channels which are encountered. The second pass then calls an interpreter on this *abstract* tree (rather than on a set of statements) to execute it. To this modified tree we refer to as the *Abstract Syntax Tree* (AST[3]).

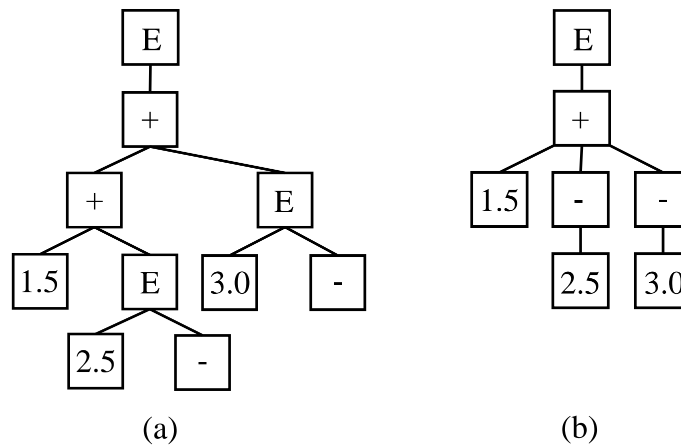


Figure 2: The Abstract Syntax Tree (AST). (a) The full tree without any implied evaluation or precedence. (b) The abridged tree with implied evaluation and precedence.

So how do we create ASTs? Figure 2 shows two diagrrams of the AST for our statement. You may recall our discussion on operator precedence and associativity from Section 4: unary '-' has precedence over the binary +' sign and is thus evaluated first. Furthermore, the unary '-' is right-associative, which means that it operates on its right argument first. Since in both cases of '-' usage the arguments are real values, this doesn't really influence the flow. However, since the '+' operator is left-associative, it means that the left two arguments are added first and you may recognise this in Figure 2a. Thus, if associativity and precedence are to be stressed, we use the expanded form of the AST: see Figure 2a. On the other hand, if both associativity and precedence are implied, an abridged form of AST is used: see Figure 2b.

To recall the introductory analogy with the RPN calculators: Abstract Syntax Trees are an *operational stack* in a sense that the AST execution

[3]Please note that AST is only one of many sound approaches to handle *intermetiate representations* (IR). For different grammars different IRs might prove better, but this is beyond the scope of this concise overview.

flows bottom-up. This notion in fact comes in naturally, for ASTs are used exactly for stack operations. We elaborate on this point in A.3.

A.2 Lexing LALR(1) grammars

A.3 Parsing LALR(1) grammars

A.4 Constructing ASTs in PHOEBE

A.5 Symbol tables

All identifiers and their values are stored in a symbol table. Whenever an identifier is used, the symbol table is queried for its value. To make this lookup as fast as possible, PHOEBE uses *hashed* symbol tables.

Having many identifiers with possibly long names in a symbol table would imply that the name lookup is the $\mathcal{O}(N)$ operation. If instead for each identifier a hash number is created by the given hash function which is not expensive to calculate, the lookup becomes the $\mathcal{O}(1)$ operation, given that the number of symbols in a symbol table isn't too large. PHOEBE implements a symbol table with size 97, which in practice means that the lookup will be optimal for as long as the number of all declared identifiers doesn't exceed $\sim$500. If it does, the lookup will still work, but the gain over conventional lookup methods won't be as significant.

For more information on hashing strings and using symbol tables please refer to [1].

B Qualifiers

This appendix lists all PHOEBE qualifiers and their corresponding types. Qualifiers are used as symbolic identifiers for parameters throughout PHOEBE.

Generic qualifier synopsis is: `prefix[_module]_contents[_type]`

Here the '[', ']' braces stress that the enclosed part is optionally used.

`prefix:`

Prefix is a keyword that denotes the origin of the qualifier. It is used to tell apart internal phoebe qualifiers from other qualifiers (i.e. the ones being introduced by the GUI and/or other plugins). In particular:

- if the qualifier is implemented by PHOEBE, the prefix is '`phoebe`';
- if the qualifier is implemented by the GUI, the prefix is '`gui`';
- if the qualifier is implemented by the plugin 'blah', the prefix is '`blah`'.

`module:`

Module is a keyword that denotes a group the qualifier belongs to. It is used to further classify qualifiers by parameters which they represent. In particular, if a qualifier identifies a light-curve related parameter, the module would be '`lc`'. If it identifies an RV curve related parameter, the module would be '`rv`'. It is optional in a sense that its usage is not always desired or even applicable.

`contents:`

Contents is a keyword that gives the qualifier's meaning. This part is the part that does unambigous identification. For the number of light curves, contents would be '`lcno`', for the semi-major axis, contents would be '`sma`'.

`type:`

Type is a keyword that determines the qualifier type. It is used to stress what is the native type of the parameter - is it a value, a switch or a list. Parameter values may be integers, reals, strings or arrays. Switches are always boolean. Lists contain predefined choices, of which only one a parameter may assume. Since most qualifiers will identify values, the 'value' type should be omitted for brevity. Other types must be given: 'switch', 'list' etc.

For further examples on qualifier construction please refer to Table 9.

prefix:	module:	contents:	type:	qualifier:
phoebe	n/a	sma	n/a	phoebe_sma
phoebe	lc	filename	n/a	phoebe_lc_filename
phoebe	n/a	ie	switch	phoebe_ie_switch
gui	gnuplot	device	list	gui_gnuplot_device_list

Table 9: Examples of forging PHOEBE qualifiers.

B.1 Model parameters

The following are parameters which determine morphological and conditional constraints of the model.

PHOEBE qualifier:	Type:	Short description:
phoebe_name	string	Star common name
phoebe_indep	string	Model independent variable
phoebe_model	string	Model morphological constraint
phoebe_lcno	integer	Number of observed light curves
phoebe_rvno	integer	Number of observed RV curves
phoebe_spno	integer	Number of flattened spectra
phoebe_asini_switch	boolean	Constraint $a \sin i = \text{const.}$
phoebe_asini_value	real	Value of the $a \sin i$ constant
phoebe_cindex_switch	boolean	Color-index constraint
phoebe_cindex	real	An array of color indices
phoebe_msc1_switch	boolean	Main-sequence constraint for star 1
phoebe_msc2_switch	boolean	Main-sequence constraint for star 2

B.2 Data parameters

The following are parameters which determine observational data (light and RV curves) and transformation factors.

PHOEBE qualifier:	Type:	Short description:
phoebe_lc_filename	string	LC data filename
phoebe_lc_sigma	real	LC data standard deviation
phoebe_lc_filter	string	LC data passband
phoebe_lc_indep	string	LC data independent variable type
phoebe_lc_dep	string	LC data dependent variable type
phoebe_lc_indweight	string	LC data individual weight type
phoebe_lc_levweight	string	LC data level weight type
phoebe_lc_active	boolean	Is LC data used for fitting
phoebe_rv_filename	string	RV data filename
phoebe_rv_sigma	real	RV data standard deviation
phoebe_rv_filter	string	RV data passband
phoebe_rv_indep	string	RV data independent variable type
phoebe_rv_dep	string	RV data dependent variable type
phoebe_rv_indweight	string	RV data individual weight type

Table continued on the next page...

PHOEBE qualifier:	Type:	Short description:
phoebe_rv_active	boolean	Is RV data used for fitting
phoebe_mnorm	real	Quarter-phase magnitude
phoebe_bins_switch	boolean	Should data be binned
phoebe_bins	integer	Number of bins
phoebe_ie_switch	boolean	Should data be de-reddened
phoebe_ie_factor	real	De-reddening factor
phoebe_ie_excess	real	De-reddening color excess
phoebe_proximity_rv1_switch	boolean	Rossiter effect for RV_1
phoebe_proximity_rv2_switch	boolean	Rossiter effect for RV_2

B.3 Physical parameters

The following are physical and geometrical parameters that determine shape and structure of the modeled binary.

PHOEBE qualifier:	Type:	Short description:
phoebe_hjd0	adjustable	Zero epoch time in HJD
phoebe_period	adjustable	Period value in days
phoebe_dpdt	adjustable	Period time derivative
phoebe_pshift	adjustable	Phase shift
phoebe_sma	adjustable	Semi-major axis in $R_\odot$
phoebe_rm	adjustable	Ratio of mass (M_2/M_1)
phoebe_incl	adjustable	System inclination in $^\circ$
phoebe_vga	adjustable	Systemic radial velocity
phoebe_teff1	adjustable	$T_{\rm eff}$ of star 1 in K
phoebe_teff2	adjustable	$T_{\rm eff}$ of star 2 in K
phoebe_pot1	adjustable	Surface potential of star 1
phoebe_pot2	adjustable	Surface potential of star 2
phoebe_logg1	adjustable	Surface gravity of star 1
phoebe_logg2	adjustable	Surface gravity of star 2
phoebe_met1	adjustable	Metallicity of star 1
phoebe_met2	adjustable	Metallicity of star 2
phoebe_f1	adjustable	Synchronicity of star 1
phoebe_f2	adjustable	Synchronicity of star 2
phoebe_alb1	adjustable	Albedo of star 1
phoebe_alb2	adjustable	Albedo of star 2
phoebe_grb1	adjustable	Gravity brightening of star 1
phoebe_grb2	adjustable	Gravity brightening of star 2
phoebe_ecc	adjustable	Orbital eccentricity
phoebe_perr0	adjustable	Argument of periastron

Table continued on the next page. . .

PHOEBE qualifier:	Type:	Short description:
phoebe_dperdt	adjustable	Periastron time derivative
phoebe_hla	adjustable	Passband luminosity of star 1
phoebe_cla	adjustable	Passband luminosity of star 2
phoebe_el3	adjustable	Third light
phoebe_opsf	adjustable	Opacity function
phoebe_atm1_switch	boolean	Model atmosphere for star 1
phoebe_atm2_switch	boolean	Model atmosphere for star 2
phoebe_reffect_switch	boolean	Detailed reflection effect
phoebe_reffect_reflections	integer	Number of reflections
phoebe_usecla_switch	boolean	Decouple L_2^i's from radiation law
phoebe_grid_finegrid1	integer	Star 1 fine grid size
phoebe_grid_finegrid2	integer	Star 1 fine grid size
phoebe_grid_coarsegrid1	integer	Star 1 coarse grid size
phoebe_grid_coarsegrid2	integer	Star 1 coarse grid size

Table continued on the next page…

PHOEBE qualifier:	Type:	Short description:
phoebe_ld_model	string	Empirical limb darkening model
phoebe_ld_xbol1	real	Bolometric LD coefficient x_1
phoebe_ld_ybol1	real	Bolometric LD coefficient y_1
phoebe_ld_xbol2	real	Bolometric LD coefficient x_2
phoebe_ld_ybol2	real	Bolometric LD coefficient y_2
phoebe_ld_lcx1	real	Passband LD coefficient x_1^{LC}
phoebe_ld_lcy1	real	Passband LD coefficient y_1^{LC}
phoebe_ld_lcx2	real	Passband LD coefficient x_2^{LC}
phoebe_ld_lcy2	real	Passband LD coefficient y_2^{LC}
phoebe_ld_rvx1	real	Passband LD coefficient x_1^{RV}
phoebe_ld_rvy1	real	Passband LD coefficient y_1^{RV}
phoebe_ld_rvx2	real	Passband LD coefficient x_2^{RV}
phoebe_ld_rvy2	real	Passband LD coefficient y_2^{RV}

Table continued on the next page…

PHOEBE qualifier:	Type:	Short description:
phoebe_spots_no1	integer	Number of spots on star 1
phoebe_spots_no2	integer	Number of spots on star 2
phoebe_spots_lat1	real	Spot latitude on star 1
phoebe_spots_long1	real	Spot longitude on star 1
phoebe_spots_rad1	real	Spot angular radius on star 1
phoebe_spots_temp1	real	Spot temperature factor on star 1
phoebe_spots_lat2	real	Spot latitude on star 2
phoebe_spots_long2	real	Spot longitude on star 2
phoebe_spots_rad2	real	Spot angular radius on star 2
phoebe_spots_temp2	real	Spot temperature factor on star 2
phoebe_spots_move1	boolean	Spots on star 1 move in longitude
phoebe_spots_move2	boolean	Spots on star 2 move in longitude

Table continued on the next page...

PHOEBE qualifier:	Type:	Short description:
phoebe_synscatter_switch	boolean	Add synthetic scatter to LCs
phoebe_synscatter_sigma	real	Synthetic scatter sigma
phoebe_synscatter_seed	real	Synthetic scatter RNG seed
phoebe_synscatter_levweight	string	Synthetic scatter level weighting scheme

B.4 Minimization-related parameters

PHOEBE qualifier:	Type:	Short description:
phoebe_compute_hla_switch	boolean	Compute passband levels automatically
phoebe_compute_vga_switch	boolean	Compute center-of-mass velocity automatically

B.5 Scripter-related parameters

PHOEBE qualifier:	Type:	Short description:
scripter_verbosity_level	integer	How verbose should scripter be