The RNPL Reference Manual

Robert Marsa
Matthew Choptuik
Center for Relativity
The University of Texas at Austin
Austin, TX, 78712-1081
marsa@hoffmann.ph.utexas.edu
matt@infeld.ph.utexas.edu

March 1995

Introduction

The acronym `RNPL` stands for Rapid Numerical Prototyping Language. It is a language for expressing time-dependent systems of partial differential equations and the information necessary for solving them using finite-difference techniques. It has advantages over traditional programming languages such as C and FORTRAN because it only requires the user to enter the essential structure of the program while it fills in the details.

`RNPL` can produce complete working programs, or a "skeleton" that the user can complete. The programs include facilities for reading parameters from a file, interactive control over output times and functions, memory management, and state dumping for calculation interruption and restart. Throughout this manual, program examples will be given in Courier , while the rest of the text will be in Times.

Chapter 1

Program Structure

An RNPL program is completely declarative. There are no loop constructs, branch instructions, or sub-functions–just a series of declarations. These declarations can occur in any order.

The input stream is (as usual) broken up into tokens. Tokens are collections of non-white-space characters and are separated by white-space. A white-space character is a space, tab, or new-line. White-space is ignored except as a token separator, so programs are free-form in the sense of C.

1.1 Comments

There are two kinds of comments in RNPL programs. The first kind must start with a `#` at the beginning of a line. It continues till the end of the line. The second kind starts with `//` and ends at the end of the line (just like a C++ comment). The following example illustrates both kinds of comments.

```
# This is the first kind of comment
float A on grid1  // This is the second kind of comment
// So is this
```

1.2 Parameters

A parameter can be declared in several ways depending on whether it is a scalar or a vector, whether or not it has a default value, and whether or not it is a "constant." Some example declarations are:

```
parameter int fred
constant parameter float jim := 5
parameter float george[10]
parameter float ted[3] = [2.0 1.7 11]
parameter string name := "file_name"
constant parameter string comments[2] := ["comment 1" "comment 2"]
```

As the examples show, the parameter declaration begins with the reserved word `parameter` (optionally preceded by the reserved word constant). This is followed by a type which can be int , float , or string. Next comes a name with a size specification if the parameter is a vector. Finally, there is an optional assignment statement consisting of either = or := and a value.

Parameters provide a means of getting information to the program at run time. The use of parameters is discussed in section 4.1.

1.2.1 System Parameters

Along with the parameters discussed above, is a special type of parameter known as the system parameter. These parameters are program dependent, but must be known at compile time. Therefore, system parameters must have default values and are not read from parameter files. Currently system parameters must be scalars. There is one predefined system parameter which specifies the memory size for FORTRAN programs in doubles. Its default value is 2000000. An example statement follows:

```
system parameter int memsiz := 100000
```

System parameters are placed in a file called `sys_param.inc`.

1.3 Coordinate Systems

The `RNPL` compiler must know which names are coordinates. It further groups these names into systems. Some example coordinate declarations are:

```
rect coordinates t,x,y,z
sph coordinates t,r,theta,phi
```

First comes a coordinate system name, then the reserved word `coordinates`, and then a comma-separated list of names. Order is important, with the first coordinate in the list taken to be time. Any time coordinate can be repeated in other coordinate systems, but each spatial coordinate can be used only once.

1.4 Grids

Grids define the spatial regions over which the grid functions will be defined as well as their storage. A grid declaration can take one of several forms, the longest of which would be something like:

```
uniform rect[x,z] grid g1 [1:Nx][1:Nz] {xmin:xmax}{zmin:zmax}
```

The first word can be `uniform` or `nonuniform`, though only the former is currently defined. Next comes the name of the coordinate system followed by a list of coordinates on which the grid is defined. The above grid is two dimensional with coordinates x and z. After the coordinate system comes the reserved word `grid` followed by the grid name. Next comes the index region. In this example, the first index starts at 1 and goes to `Nx`, while the second starts at 1 and goes to `Nz`. `Nx` and `Nz` must be defined elsewhere. The index regions can contain arbitrary expressions such as `[A*B+C-2:4*Nx-5/a]`, however, as discussed in section 2.2, it is best to keep to forms like `[1:Nx]` and `[0:Nx-1]`, where `Nx` has been declared as a parameter. Finally, comes the coordinate region which gives the actual spatial ranges of the coordinates. In the example, we have $xmin \leq x \leq xmax$ and $zmin \leq z \leq zmax$. Coordinate regions must be of the form `{name1:name2}`, where `name1` and `name2` have been declared as parameters.

Other forms of the grid declaration leave out one or more of the above parts. The minimum allowable declaration is:

```
uniform rect grid g2
```

This declaration (along with the example coordinate declaration in section 1.3) declares `g2` to be a three dimensional grid with coordinates x, y and z. The index region will be `[0:Nx-1][0:Ny-1][0:Nz-1]` for C output and `[1:Nx][1:Ny][1:Nz]` for FORTRAN output. The coordinate region will be `{xmin:xmax}{ymin:ymax}{zmin:zmax}`

1.5 Grid Functions

A grid function is a function defined on a grid at one or more times. Some examples of grid function declarations are:

```
float A on g1 at -1,0,1
int B on g2 at 0,1
float C on g1
float D on g2 at -1,0,1 alias
float E on g3 at 0,1 "Electric Field"
```

First comes the grid function type, either `float` or `int`. Next comes the name followed by the reserved word `on` and the grid name on which the function is defined. If the declaration stopped here (such as that for `C` above), we get a single time level. Adding the reserved word `at` followed by a list of offsets (positive or negative integers) gives a function defined on one time level for each offset. For instance the definition for `A` would give a three time level function defined at times $n-1, n$, and $n+1$. Next comes the optional reserved word `alias` which declares common storage for the first and last time levels. Following any of these declarations can be a string which is used as a "print name" for the grid function. Uses for the print name will be explained in chapter 2.

1.6 Attributes

An attribute is a flag array associated with the grid functions. For instance, an attribute may tell which grid functions are to be output and which are not. Attributes are defined in a similar manner to vector parameters, except the size is replaced by an encoding. The encoding is either `encodeone` or `encodeall`, with `encodeone` giving one value per grid function and `encodeall` giving one value per time level per grid function. For instance, if five grid functions are defined, three of which have three time levels each while the remaining two have two levels each, then an attribute marked as `encodeone` would have a length of five, while an attribute marked as `encodeall` would have a length of thirteen. In this case an output flag array could be defined as either

```
attribute int out_gf encodeone
```

or

```
attribute int out_gf encodeone := [0 0 1 1 0]
```

1.7 Derivative Operators

Derivative operators are operators which act on grid functions. They are used for turning differential equations into finite difference equations. Here is a declaration for a forward difference operator:

```
operator D_FW(f,r) := (<0>f[1] - <0>f[0])/dr
```

Whenever an operator is used in an expression (see section 1.12), the operator is replaced by its definition. The name `f` is arbitrary. It simply shows where the expression goes in the definition. For instance, if `D_FW(3*A+B,r)` appeared in an expression, the `f`'s in the right hand side would be replaced by `3*A+B`. The notation `<0>f[1]` is interpreted as f^n_{i+1}, that is, f at the nth time level and i+1st grid position. The `<>[]` is really an operator which acts on expressions as follows:

`<a>f[b]` $\rightarrow$ f if f is a number or parameter or time coordinate
`<a>f[b]` $\rightarrow$ f_{i+b} if f is a spatial coordinate
`<a>f[b]` $\rightarrow$ f_{i+b}^{n+a} if f is a grid function

Three dimensional forward difference operators would look like this:

```
operator D_FW(f,x) := (<0>f[1][0][0] - <0>f[0][0][0])/dx
operator D_FW(f,y) := (<0>f[0][1][0] - <0>f[0][0][0])/dy
operator D_FW(f,z) := (<0>f[0][0][1] - <0>f[0][0][0])/dz
```

Operator definitions can be nested as in:

```
operator D_FW(f,r) := (<0>f[1] - <0>f[0])/dr
operator D_BW(f,r) := (<0>f[0] - <0>f[-1])/dr
operator D_CN1(f,r,r) := D_BW(D_FW(<0>f[0],r),r)
operator D_CN2(f,r,r) := D_BW(D_FW(<1>f[0],r),r)
```

As you can predict, the definition of `D_CN1` will result in the usual centered second derivative, namely $(f_{i+1}^n - 2f_i^n + f_{i-1}^n)/dr^2$, while the definition of `D_CN2` will result in the same thing applied at the advanced time level, that is $(f_{i+1}^{n+1} - 2f_i^{n+1} + f_{i-1}^{n+1})/dr^2$. The list of coordinate names after the `f` signifies with respect to which coordinate(s) the derivative is taken.

Although operators are defined like derivatives and act as derivatives under certain circumstances (see section 1.12), they can be defined to perform other functions, such as the following definition which performs spatial averaging.

```
operator AVG(f,r) := (<0>f[1] + <0>f[0])/2
```

Because operators are internally treated as derivatives, even definitions such as this need the coordinate list.

1.8 Residuals

Residuals define the system of equations, typically by using derivative operators. Consider the following residual definition:

```
residual phi { [0:0]       := D_LF(phi,t) ;
               [1:Nx-2]    := D_LF(phi,t,t) - D_LF(phi,x,x) ;
               [Nx-1:Nx-1] := D_LF(phi,t) }
```

Assuming the proper definitions of the derivative operators, this residual will encodes the linear wave equation on a string with the end points fixed. An equivalent declaration would be:

```
residual phi { [0:0]       := D_LF(phi,t) = 0 ;
               [1:Nx-2]    := D_LF(phi,t,t) = D_LF(phi,x,x) ;
               [Nx-1:Nx-1] := D_LF(phi,t) = 0 }
```

First comes the reserved word `residual` followed by the name of the grid function whose residual is being defined. Next comes a bracket-enclosed set of index regions and expressions. The index region shows over what range the expression is a valid description of the behavior of the system.

The union of the index regions should equal the index region of the grid on which the function is defined, but this is not required. Note that each region-expression pair is separated by a semi-colon.

The residual tells how to determine the advanced value of a grid function. Thus, the residual must contain `<a>f...` where `a` is the offset to the most advanced time level defined for grid function `f`.

Part of a residual for a three dimensional grid function would look like:

```
residual A { [1:Nx][1:1][1:1] := D_LF(A,x) + D_FW(B,y) ;
          [Nx:Nx][1:Ny][1:Nz] := <1>A[0][0][0] = 5.0*C }
```

The reserved word `residual` can be preceded by the reserved word `evaluate` which tells the compiler to produce code which will evaluate the residual.

In addition, the word `residual` can be followed by a global offset for example:

```
residual <1>[0] A { [1:Nx] := ... }
```

This offset is applied globally to each expression appearing in the residual.

See section 1.12 for more information on expressions.

1.9 Initializations

An initialization defines the initial data for a grid function. Its form is identical to the residual declaration with `residual` replaced by `initialize`. However the expression is interpreted differently. Consider the following initialization declaration:

```
initialize phi { [1:Nr] := amp*exp(-((r-c)/delta)^2) }
```

Unlike the residual declaration, the expression in the initialization must not contain its grid function. The retarded time level of the grid function is set to the expression. In the case above, `phi` will be set to a Gaussian.

1.10 Loop Driver

The loop driver declaration has the form `looper name`, where `name` is any identifier. When the compiler processes the driver statement, it looks in its library directory for a file called name.drv_lang, where lang is c, f77, or f90. It then simply includes this file into the generated source.

There are currently two predefined loopers: iterative and standard. The iterative looper makes an initial guess at the advanced grid function values, then calls the update routines and iterates until the norm of the residual is below a threshold. The standard driver just calls the update routines.

1.11 Updates

Update declarations can take many forms. Here are some examples:

```
auto update phi,pi,beta
```

```
stub evolver updates A,B,C
```

```
  header A, B[Bnp1,Bn,Bnm1], C[C], x,y,z,dt,auto work#0(5*Nx*Ny*Nz),
         static work#1(3*Nx*Ny-.5*Nz)
```

```
myroutine.inc myupdate update A,B header A,B,dt
```

The first form defines an automatic update. The declaration above would cause the compiler to produce a routine to update the grid functions phi, pi, and beta if residuals have been declared for them. Otherwise, the compiler has no idea how to update the grid functions and will produce an error message.

The second declaration will cause the compiler to produce the header for a routine called evolver which is expected to update grid functions A, B, and C. The body of the routine is left blank, to be filled in by the user. Following the reserved word `header`, comes a list of things to appear in the calling sequence for the function. A grid function name such as `A` above will cause all the time levels of A to be passed to the function. If A has three time levels (1,0,-1), then they will be named A_np1, A_n, and A_nm1 by default. The user can provide his own names to override the defaults as in the case of `B`. If only one name is provided (as for `C`), the time levels will be passed in as the elements of a single vector, the first component of which will be the advanced time level.

Other things that can appear in the header list are coordinates (such as x,y,z above) and coordinate differentials (such as dt). Parameters can also be included in the list. Work arrays are declared like the final two parameters. First comes `auto` or `static`. Static work arrays are declared at the start of the program and persist throughout. Auto work arrays are allocated before the call to the update routine and are destroyed afterwards. Next comes the word `work` followed by the `#` symbol and an integer. This integer is tacked onto the end of the word work to form the name of the array. Finally comes an expression for the size of the work array enclosed in parentheses.

1.12 Expressions

Expressions are made up mainly of identifiers separated by operators. `RNPL` defines the usual set of arithmetic operators (`+,-,*,/`) along with exponentiation (`^` or `**`). The operators obey the usual precedence rules.

Identifiers may be names of grid functions, coordinates, or parameters. In addition, grid functions may be supplied with temporal and spatial offsets (see section 1.7), and coordinates may be supplied with a spatial offset.

Expressions can also contain the well-known functions exp, log, tan, sin, cos, sinh, cosh, tanh, and sqrt as well as derivative operators.

Here is an example of a complicated `RNPL` expression:

```
a*b+(c*2.67/<0>phi[1] + d^2)/tan(theta) - r[-1]*D_(phi*3/a,r) +
  cos(3*eta)*D_(eta+D_(phi,r),r) + expand D_(a*b + c,r)
```

You'll notice the word `expand` before the last derivative operator. This tells `RNPL` to symbolically expand the derivative before making the operator substitution. Thus, the final term is equivalent to:

```
D_(a,r)*b + a*D_(b,r) + D_(c,r)
```

Chapter 2

Implementation

2.1 Case Sensitivity

Identifiers in RNPL are case sensitive if the target language is C and are case insensitive if the target language is FORTRAN. Regardless of the target language, reserved words may be given in all lower case or all upper case. Case combinations will produce syntax errors.

2.2 Coordinate Differentials

Coordinate differentials are assumed to be anything that is a coordinate name with a d in front of it, such as dt or dx if t and x are coordinates. A coordinate differential is defined by the index region and coordinate region of the first grid which uses its coordinate. For instance, if the following coordinate system and grids are defined:

```
rect coordinates t,x,y

uniform rect grid g1 [1:Nx][1:Ny] {xmin:xmax} {ymin:ymax}
uniform rect[x] grid g2 [1:N] {min:max}
```

then dx will be defined from g1 by $dx = (xmax - xmin)/(Nx - 1)$ even though g2 may have a different coordinate spacing.

2.3 Special Parameters and Attributes

There are several "special" parameters defined by RNPL. These are declared by the program (if not by the user) and given default values. They will be read from the parameter file if it contains them. These parameters are shown in the following list along with their definitions and default values.

```
constant parameter float start_t := 0 // start time
constant parameter int iter := 100 // number of iterations
constant parameter float epsiter := 1e-5 // iteration threshold
constant parameter int fout := 0  // file output (0 no, 1 yes)
constant parameter int ser := 0 // fs output (0 no, 1 yes) if appropriate
constant parameter float lambda := .5 // dt/dr, dr=sqrt((dx^2+dy^2+dz^2)/3)
constant parameter int rmod := 1 // output every rmodth time step
```

```
constant parameter string in_file // name of file from which initial data
                                  // will be read
constant parameter string out_file // name of file to which data will
                                   // be written
constant parameter int level := 0 // refinement level
constant parameter int s_step := 0 // starting iteration number
constant parameter string tag := "" // prepend symbol for grid function names
constant parameter int N<c>0 := 2 // base number of grid points for the
                                // coordinate c (there is one for each spatial
                                // coordinate)
```

The names `N<c>`, where `<c>` is a coordinate, are declared internally to `RNPL`. They are NOT parameters, nor should they be declared by the user. `N<c>` is defined at run time by `N<c>` = `N<c>0` $2^{level}+1$. Thus, if a grid is defined with an index region of length `N<c>`, it will be automatically scaled simply by changing the value of level in the parameter file. Grids defined in this way will always have an odd number of points in each dimension.

There is one pre-defined attribute: `out_gf`. This is an integer array with one element for each grid function, that is:

```
attribute int out_gf encodeone
```

Each element is assigned the default value of 0. This element tells whether output is enabled for that grid function or not. The values of this attribute can be changed during program execution.

Chapter 3

Compiler Usage

3.1 Code Generation

The RNPL compiler is called from the command line with the following command:

```
rnpl -lang [program_file]
```

The switch -lang tells RNPL what the target output language is, c, f77, or f90.

The compiler generates code for two programs, the solver and the initial data generator. If program_file is specified, rnpl will output the solver code to a file. The extensions .rnpl or _rnpl will be removed from the end of program_file if they exist. The appropriate extension for the output language is then appended (.c or .f). The initial data generator is written to program_file_init plus the appropriate extension (.c or .f) If no program_file is specified, rnpl reads from stdin and writes to stdout. In this case, the initial data generator goes to r_out_init (.c or .f). Errors are directed to stderr.

RNPL sends the code for the update routines to a file named updates(.h or .f). Also, a default attribute file is produced, named .rnpl.attributes.

In the case of FORTRAN output, two include files named globals.inc and other_glbs.inc are also produced.

The compiler needs certain support files which it looks for in the current directory or in the directory defined by the environment variable RNPL_PATH.

3.2 Target Language Compilation

Once the RNPL compiler has been executed, the C or FORTRAN sources must be built. In the case of C output, simply compile the two .c files since updates.h is automatically included. For FORTRAN code, compile the solver and updates.f and link them together.

Chapter 4

Generated Programs

4.1 Parameter Files

A parameter file is an ASCII file containing arbitrary text along with lines of the form name := value, where name is the name of a parameter, and value is its new value. At run time, RNPL generated programs will read this file and use any values found to initialize parameters.

Here is an example parameter file:

```
This is a parameter file
tag := "a_"
level := 1
Nx0 := 100
xmin := -10
in_file := "init_data.hdf"
out_file := "dump.hdf"
```

4.2 Solver

If RNPL is run on wave.rnpl and the resulting source files are then compiled into executables named wave and wave_init, then the solver (wave) can be executed by typing "wave param_file" or "wave" on the command line. Param_file is an ASCII file of parameter values as discussed above. If no parameter file is given on the command line, the program will prompt for one. If there is no initial data file (in_file), wave will execute wave_init to generate one. If wave_init doesn't exist, wave will print a warning and continue without reading the initial data. If there is an initial data file, wave will read it and continue. Every rmod time steps (rmod is a parameter), wave will generate output as specified by out_gf, ser, and fout. It will print the current step and value of time to stdout. During execution, the user can type Ĉ or Ẑ to stop wave. A menu will then be presented which allows the user to change the output frequency (rmod) and which grid functions will be output (out_gf). The user can also choose to quit. Upon termination (either forced or after "iter" time steps), wave will dump state to out_file. Execution can later be resumed by copying out_file to in_file and rerunning wave.

4.3 Initial Data Generator

The initial data generator can be run from the command line as well as being called by the solver. It reads the same parameter file as the solver and writes the initial data to in_file.

4.4 Output

One of the design goals of `RNPL` is to provide uniform and automatic access to I/O facilities to aid the user in examining the results of computations. As discussed in section 2.3, there is currently one pre-defined attribute: `out_gf`. If the special parameter `fout` is set, then every `rmod` time steps, output is generated for every grid function for which output has been enabled via `out_gf`. Output is to `.hdf` files: one file for each selected grid function is created, and each file consists of a sequence of dumps labelled with the actual output time. These files can then be post-processed with a variety of software, including the Explorer^{TM} module, `ReadHDF_GFTO` available via anonymous ftp from `helmholtz.ph.utexas.edu` in `/pub/explorer/modules`.

The interface to the lower level `.hdf` routines is also directly accessible from `C` or `Fortran` codes which have *not* been generated using `RNPL` and is described in *The RNPL User's Guide*.

Chapter 5

RNPL Grammar in BNR Format

dec_list	$\rightarrow$	
	$\mid$	dec_list declaration
declaration	$\rightarrow$	param_dec
	$\mid$	coord_dec
	$\mid$	grid_dec
	$\mid$	gfunc_dec
	$\mid$	attrib_dec
	$\mid$	d_operator
	$\mid$	residual
	$\mid$	initialization
	$\mid$	looper
	$\mid$	update
param_dec	$\rightarrow$	**param** p_type name
	$\mid$	**param** p_type name **assignop** scalar
	$\mid$	**param** p_type name v_size
	$\mid$	**param** p_type name v_size **assignop** vector
	$\mid$	**const param** p_type name
	$\mid$	**const param** p_type name **assignop** scalar
	$\mid$	**const param** p_type name v_size
	$\mid$	**const param** p_type name v_size **assignop** vector
coord_dec	$\rightarrow$	name **coordinates** coord_list
grid_dec	$\rightarrow$	g_type name **grid** name i_region c_region
	$\mid$	g_type name **grid** name
	$\mid$	g_type name **obrack** coord_list **cbrack** **grid** name i_region c_region
	$\mid$	g_type name **obrack** coord_list **cbrack** **grid** name
gfunc_dec	$\rightarrow$	type name **on** name
	$\mid$	type name **on** name str
	$\mid$	type name **on** name **at** o_list

	|	type name **on** name **at** o_list **alias**
	|	type name **on** name **at** o_list str
	|	type name **on** name **at** o_list **alias** str
attrib_dec	→	**attrib** p_type name encoding
	|	**attrib** p_type name encoding **assignop** vector
d_operator	→	**operator** d_op **assignop** expr
residual	→	**resid** name **obrace** res_list **cbrace**
	|	**resid** time index name **obrace** res_list **cbrace**
	|	**evaluate resid** name **obrace** res_list **cbrace**
	|	**evaluate resid** time index name **obrace** res_list **cbrace**
initialization	→	**initialize** name **obrace** res_list **cbrace**
looper	→	**looper** name
update	→	name name **update** coord_list **header** ref_list
	|	**stub** name **update** coord_list **header** ref_list
	|	**auto update** coord_list
p_type	→	**int**
	|	**float**
	|	**string**
name	→	**iden**
scalar	→	**inum**
	|	**minus inum**
	|	**num**
	|	**minus num**
	|	**str**
v_size	→	**obrack inum cbrack**
vector	→	**obrack** scalar_list **cbrack**
coord_list	→	name
	|	coord_list **comma** name
g_type	→	**uniform**
	|	**nonuniform**
i_region	→	**obrack** expr **colon** expr **cbrack**
	|	**obrack** expr **colon** expr **cbrack** i_region
c_region	→	**obrace** name **colon** name **cbrace**

	$\mid$	**obrace** name **colon** name **cbrace** c_region
type	$\rightarrow$	**int**
	$\mid$	**float**
o_list	$\rightarrow$	**inum**
	$\mid$	**minus inum**
	$\mid$	o_list **comma inum**
	$\mid$	o_list **comma minus inum**
encoding	$\rightarrow$	**encodeone**
	$\mid$	**encodeall**
d_op	$\rightarrow$	name **oparen** expr **comma** coord_list **cparen**
	$\mid$	**expand** name **oparen** expr **comma** coord_list **cparen**
expr	$\rightarrow$	expr **plus** expr
	$\mid$	expr **minus** expr
	$\mid$	expr **equals** expr
	$\mid$	expr **times** expr
	$\mid$	expr **divide** expr
	$\mid$	expr **caret** expr
	$\mid$	**plus** expr
	$\mid$	**minus** expr
	$\mid$	**oparen** expr **cparen**
	$\mid$	d_op
	$\mid$	func
	$\mid$	gfunc
	$\mid$	coord
	$\mid$	name
	$\mid$	**num**
	$\mid$	**inum**
res_list	$\rightarrow$	i_region **assignop** expr
	$\mid$	res_list **scolon** i_region **assignop** expr
time	$\rightarrow$	**oabr inum cabr**
	$\mid$	**oabr minus inum cabr**
index	$\rightarrow$	**obrack inum cbrack**
	$\mid$	**obrack minus inum cbrack**
	$\mid$	**obrack inum cbrack** index
	$\mid$	**obrack minus inum cbrack** index
ref_list	$\rightarrow$	reference
	$\mid$	ref_list **comma** reference
scalar_list	$\rightarrow$	scalar

	|	scalar_list scalar
func	→	name **oparen** expr **cparen**
gfunc	→	time name index
coord	→	name **obrack inum cbrack**
reference	→	name
	|	name **obrack** coord_list **cbrack**
	|	**auto work pound inum oparen** expr **cparen**
	|	**static work pound inum oparen** expr **cparen**

Terminals

param	parameter or PARAMETER
assignop	:= or =
coordinates	coordinates or COORDINATES
grid	grid or GRID
obrack	[
cbrack	]
on	on or ON
at	at or AT
attrib	attribute or ATTRIBUTE
operator	operator or OPERATOR
resid	residual or RESIDUAL
obrace	{
cbrace	}
evaluate	evaluate or EVALUATE
initialize	initialize or INITIALIZE
looper	looper or LOOPER
update	update or updates or UPDATE or UPDATES
int	int or INT
float	float or FLOAT
string	string or STRING
iden	see below
inum	positive integer
minus	−
num	positive real number
str	"any characters"
comma	,
uniform	uniform or UNIFORM
nonuniform	nonuniform or NONUNIFORM
colon	:
encodeone	encodeone or ENCODEONE
encodeall	encodeall or ENCODEALL

oparen	(
cparen	)
expand	expand or EXPAND
plus	+
times	*
divide	/
caret	^ or **
scolon	;
oabr	<
cabr	>

An iden is string which starts with a letter or _ and contains letters, digits, and _. It can also be two or more iden's separated by . or ->. For example, the following are valid iden's:
frank
AlBerT0
_george123fred__
brad.charles
employer->name.first

Chapter 6

Acknowledgements

This work was supported by NSF PHY-9310083 to R.A. Matzner, and ASC9318152 (ARPA supplemented) to the Binary Black Holes Grand Challenge Alliance

Bibliography