

Grace - Daughter of Gray

1 Introduction

Gray is a parser generator, written by Anton Ertl, that is distributed with GForth. Grace provides two extensions to Gray, a BNF front end and the ability to generate a standalone parser for the target language. This document describes how to use Grace, it does not describe the design of Grace.

1.1 BNF front end

Gray requires a grammar with productions specified in a format specific to Gray, this format is similar to an extended BNF with special notation. Disadvantages of this are that a user has to learn the Gray format and that any grammars already written in BNF have to be manually converted into Gray format. The first extension is to the front end to Gray so that it will recognise a conventional extended BNF source file. Why do this? There are basically two reasons, firstly more people are familiar with standard BNF, secondly when developing a parser for a language with a published grammar, use of Grace avoids the need for manual translation of the grammar, which is, admittedly, not difficult but tedious and therefore error prone.

1.2 Standalone parser generation

Gray compiles the parser into memory and is, therefore, an on-line tool. Using the compiled parser requires the user to compile Gray, then the parser prior to running the parser. The second extension to Gray is to generate the parser as Forth source code, thus making Grace an off-line tool - generate the parser once then use it without the overheads of Gray. While these overheads are not large, particularly in the context of today's desktop computers, this gives the user the choice of an off-line or on-line parser.

1.3 Modes of operation

The code for Grace has been factored into files so that Grace can be used in any of four modes by using conditional compilation:

- a. with both extensions i.e. BNF input file and generating a standalone parser.
- b. with Gray format input generating a standalone parser
- c. with BNF input compiling an in-memory parser
- d. with neither extension i.e. emulates Gray with Gray input compiling an in-memory parser

1.4 ANS Forth compliancy

Grace is written in ANS Forth and should be compatible with any ANS Forth compliant system e.g. GForth. Descriptions in this document on how to run the examples will assume the use of GForth.

1.5 Copyright

The changes and additions to Gray are copyright G W Jackson, the copyright of the original code remains with Anton Ertl and is issued under the GNU public licence. Both tools may be freely used for any purpose as long as the copyright notices in the files are preserved if they are modified and/or distributed further. They are made available in the hope that they will be of use and no guarantee or warranty is provided.

2 BNF input format - the first extension

This extension is basically a bolt-on front end to Gray that recognises a BNF format and calls the appropriate interface words in Gray. To use this the user simply prepares a source file specifying the grammar in the BNF format described below (see section 2.6 for an example) and, having included the appropriate files, to call a word such as `grace` in file `grace.fth` with the name of the BNF file. The BNF format recognised by Grace is similar to that used in specification of the XML grammar with two extensions. This section uses terms that are common in the description of programming languages and compilers. An explanation of these is given in any book on compilers e.g. the Red Dragon book ('Compilers, Principles Techniques and Tools' by Aho, Sethi and Ullman, 1986).

2.1 Productions

The grammar for a given language expressed in BNF format consists of a series of productions or rules that define non-terminals in terms of a sequence of terminals and non-terminals. A production has the form:

```
non-terminal ::= sequence of terminals and/or non-terminals;
```

where items on the right hand side may be separated by operators and qualified by other operators. Note the terminating semi-colon used to separate productions – this is an extension to the XML style BNF notation.

2.2 Non-terminals and terminals

A non-terminal is a string of alphanumeric characters including an underscore, it must start with a letter or underscore. Consecutive non-terminals must be separated by white space.

The symbol `::=` separates the non-terminal being specified from its definition.

There are two types of terminal:

- Those specified by enclosing a character string in quotes i.e.
`'string-a'` or `"string-b"`
where `string-a` and `string-b` may contain any non-control character except `'` and `"` respectively. This type of terminal will typically be keywords and operators of the language to be parsed by Gray.
- Those such as identifiers and numbers that are specified by a regular expression, e.g. in the input to LexGen. These must have the same form as a non-terminal i.e. a sequence of alphanumeric characters starting with a letter or underscore. These need not be enclosed in quotes but should be used in the form which the lexical scanner will recognise. For example see the use of `ident` in the production for `MetaIdentifier` in section below.

For example five legal terminals are: `'begin'` `"IF"` `'a"name'` `''` `fred`

Two illegal terminals are: `'abc'ef'` `""`

2.3 Production definitions

If we represent a general non-terminal or terminal as A and B the following may be used on the right hand side of productions:

A B	matches A followed by B
A B	matches A or B but not both
A*	matches zero or more occurrences of A

A+	matches one or more occurrence of A
A?	matches zero or one occurrence of A (i.e. A is optional)
(A B)	parentheses group the contents into a unit so that, for example (A B)+ means one or more occurrences of A followed by B

These may be combined and extended indefinitely e.g. A B (C | D)*

A production may be spread over several lines of text. Productions may be in any order except that the first production is taken as the parser name (see below). White space is needed between non-terminals but is not necessary between other symbols.

2.4 Extensions to BNF

BNF2Gray has two rules in addition to those of extended BNF:

- The non-terminal on the left hand side of the first production is taken as the name of the parser and must not be used in other productions.
- Each production must be terminated with a semicolon.

2.5 Comments and actions

The set of productions can also hold comments and actions.

2.5.1 Comments

These look like a C comment and are ignored by Grace e.g.

```
/* This is a comment */
```

Comments cannot be nested and can contain any character or string except */. Comments can be spread over more than one line.

2.5.2 Actions

Actions are surrounded by braces, { and } and can cover more than one line. They contain Forth code that is executed when the terminal for that part of the production has been recognised. For example in section there is a production for `SyntaxRule`, in which `{startRule}` is an action. The word `startRule` is a Forth word which is called when a `MetaIdentifier` has been recognised by the parser generated by Gray. Actions may be a sequence of Forth words.

Actions are copied through to the generated parser code inside the colon definition for the production containing the action e.g. see the example in section .

Actions being copied through to the output can lead to different behaviour between the parsers generated by Gray and Grace (see the restrictions in section 7).

2.6 Example grammar

The BNF parser used in Grace was generated by Grace itself from the following grammar complete with comments and actions - beware, depending on the font used for display or print, it may be difficult to distinguish between the parentheses and braces characters:

```

/* BNF grammar for the Grace BNF parser */

BNF ::= {startGrammar} Comment* (SyntaxRule Comment*)+ {endGrammar};
SyntaxRule ::= MetaIdentifier {startRule} ' ::= ' DefinitionsList
              {endRule} ';' ;
DefinitionsList ::= SingleDefinition ('|' {||} SingleDefinition)*;
SingleDefinition ::= Factor+;
Factor ::= ( MetaIdentifier
            | {getTerminal} terminal_string
            | GroupedSequence      ) Qualifier?
            | ActionSequence
            | Comment;
MetaIdentifier ::= {getNonTerminal} ident;
GroupedSequence ::= '(' {({} DefinitionsList ')'} {)});
Qualifier ::= {??} '?' | {++} '+' | {**} '*';
Comment ::= {skipComment} '/*'
            /* Any character string except <end comment> */
            '*/';
ActionSequence ::= {BNF-action} '{' /* User defined Forth code */ '}'
                 {endAction};

```

2.7 Running the BNF parser

More detailed information on this uses examples and follows later in this document (see section 4).

3 Parser generation - the second extension

This extension to Gray is an alternative back-end that writes a Forth source code parser to a file instead of compiling the equivalent directly into memory. Therefore this part of Grace replaces the code generation part of Gray entirely.

3.1 Forth code generation

The approach taken to code generation is:

- Each production becomes a colon definition or `:noname` definition (if a production is used before it is defined). The name of the colon definition is the production name surrounded by angle brackets.
- References to other productions are converted to a call to that production's colon definition name.
- Terminals become either a test on a constant value or a test of membership of a set where there are more than one possible valid terminals.
- BNF operators are converted to appropriate Forth control structures.
- Actions are copied in-line to colon definitions without any changes.
- Comments are ignored.

For examples compare the BNF code in section 2.6 and the generated parser in file `graceparser.fth`.

4 Using Grace

4.1 Overview

As described before Grace is based on the Gray parser generator (reference 1) and the documentation available in that package should be read and preferably understood before using Grace - see the examples provided with Gray. This section will describe the use of Grace in each of the 4 modes available and a common example will be used. This example is based on the Mini language provided as an example in the Gray distribution.

A version of the Mini parser is given for each mode of operation for illustrative purposes only, for application development, choose the mode required and develop the appropriate files for that mode (see the table in 4.3.1).

In the examples LexGen has been used to generate the lexical scanner, this is different to the Gray example. It is not necessary that LexGen be used with Grace, any other lexical scanner can be used as long as it returns numerical token values for the various symbols to be handled in the target grammar. For an example of an alternative see the file `mini.fs` in `gray5.zip` (reference 1) which uses Gray for the lexical scanner - it could be a useful learning exercise to use Grace to generate a lexical scanner for Mini. Also it is not necessary to understand LexGen at all to use Grace, but reference 2 about using LexGen with Gray may assist the understanding of how Gray works.

4.2 Files used for the Mini example

The set of files loaded for each of the four modes is given in the loader file `mini.fth`. Initially considering the files for mode 0 (Gray emulation) because the files for other modes are mostly variations of this, the set of Mini files is (ignoring library files):

```
ministt.fth
minitokens.fth
miniactions.fth
minitokens.fth
mini.gry
```

4.2.1 `ministt.fth` was generated by LexGen and with the library file `lexscanner2_0.fth` provide the lexical scanner which replaces the Gray equivalent in `mini.fs`. As stated before use of this scanner is not essential for Grace and so this file will not be discussed any further.

4.2.2 `minitokens.fth` defines the tokens used in a mini program. This file is included twice (see reference 2), firstly to define token values, secondly to define Gray terminals - the same thing is done in `mini.fs`. For the first use a word called `token` is defined in library file `lexscanner2_0.fth` to define constants, for the second use is redefined in `miniactions.fth`.

4.2.3 `miniactions.fth` is again mostly taken from `mini.fs` and the code is described in more detail below. For the other modes there are equivalents of this file that contain similar, if not identical, code. In the modes where a standalone parser is generated, the file is split into two, one for parser generation, the other for run-time.

4.2.4 `mini.gry` defines the grammar for a mini program - taken from `mini.fs`. This includes calls to actions that are defined in `miniactions.fth`. In modes where the input file is in BNF the file called `mini.bnf` is used instead.

4.3 Files for the different modes of operation

4.3.1 Each mode of operation requires the code to be split in various ways into different files. In the following descriptions the various files will be generically referred to as “target” files. The target files for each mode in the Mini example are given in this table.

Mode	Compile/Generate	Run-time
0 - Gray emulator	miniactions_gray.fth	
1 - BNF file input, compile parser into memory	miniactions.fth	
2 - Gray format input, generate a parser as Forth source code	minidefs_gray.fth	minirun.fth
3 - BNF file input, generate a parser as Forth source code	minidefs.fth	

4.4 Contents of the target files

The target files are given in the table in 4.3.1. The code in these files is divided into the types described in the next few paragraphs. Reference should be made to the appropriate files given in the table to see the actual code. Another example of a run-time file can be seen in the file `grace.fth`, which is the run-time file for parsing a BNF source file. The table in 4.4.8 below shows which code type is required for each mode of operation.

4.4.1 Interface to the scanner:

In the examples given these are calls to LexGen via the word `getNextToken` which returns the token value for the next symbol in the input file of the target language. If an alternative scanner is used it should provide the same functionality.

4.4.2 Source file handling

This simply opens and closes the target source code file. The Mini examples use a library file called `inputsources.fth` to do this so that the scanner can use ANS Forth words like `REFILL` on the input file. This code section depends on the needs of the scanner used.

4.4.3 Interface to Gray

Gray requires various things to be supplied (see reference 1):

- a call to `max-member` with the highest token value in use.
- a word to test whether a token (here held in variable `sym`) is a member of the supplied set. This word, `testsym?` has to have its execution token saved in Gray variable `test-vector`.
- a word to check that a result is true and, if so, to fetch the next token from the source file. Here this is called `?nextsym` and its execution token must be supplied to the Gray word `terminal` when a terminal is declared. This is done in the redefined word `token` below.

See file `miniactions.fth` for the actual code used. When generating a standalone parser only the call to `max-member` is required. However the other two are required when running the standalone parser.

4.4.4 Interface to the standalone Mini parser

This is only required for a standalone parser at run-time and must contain the definitions of `testsym?` and `?nextsym` as described in 4.4.3 and a word called `test-token`, all of which will be called in the generated parser e.g. see file `miniparser1.fth`. In `minirun.fth` the definition of `testsym?` is optimised as the number of tokens is less than the cell size. An alternative definition is given, but commented out, for multi-cell sets (i.e. with more than 32 members in a 32 bit system).

In addition the following values must be declared so that the parser can set their value:

```
0 value bytes/set
0 value bits/cell
```

```

0 value first-set
0 value parser-name

```

4.4.5 Definitions of actions

This contains definitions of any actions that are called in the grammar (both formats). These depend on the target language.

4.4.6 Redefinition of token

The tokens file is included for a second time so that a terminals can be created by Gray for each token. As the actions are likely to use token constants, this redefinition of `token` has to follow definition of the actions. There is a significant difference between the redefinition of `token` in files `miniactions_gray.fth` and `minidefs.fth`. The latter is used when generating a standalone parser.

4.4.7 A driver

A word is needed to make things happen i.e.

- compile a mini program (e.g. the word `mini` in file `miniactions_gray.fth`) or
- to generate a standalone parser (e.g. `generate-forth` in `minidefs.fth`) or
- to compile a program in the target language (e.g. `mini` in `minirun.fth`).

4.4.8 Code sections required for each mode of operation

Code section type	Mode 0 (Gray in) (Compile)	Mode 1 (BNF in) (Compile)	Mode 2 (Gray in) (Generate parser)		Mode 3 (BNF in) (Generate parser)	
			Generate	Run	Generate	Run
Scanner interface	✓	✓		✓		✓
Source file handling	✓	✓		✓		✓
Gray interface	✓	✓	✓		✓	
Mini parser interface				✓		✓
Definition of actions	✓	✓		✓		✓
Redefinition of <code>token</code>	✓	✓	✓		✓	
Compiler or parser driver	✓	✓	✓	✓	✓	✓

4.5 Running the examples

4.5.1 The examples can be run by setting some compilation constants at the start of the file `mini.fth`. These are `[forth-out]` and `[bnf-in]`, used as defined in this table:

Compilation constants		Mode of operation
[forth-out]	[bnf-in]	
1	1	Mode 3 - BNF file input, generate a parser as Forth source code
1	0	Mode 2 - Gray format input, generate a parser as Forth source code
0	1	Mode 1 - BNF file input, compile parser into memory
0	0	Mode 0 - Gray emulator

4.5.2 For each mode `mini.fth` includes the necessary library and Grace files before the appropriate mini files. As can be seen the files used do vary according to the mode of operation. When you develop your own parser cut and paste the set of files from `graceloader.fth` for the mode you wish to use into your own loader file replacing the mini files with your own application. Alternatively use `graceloader.fth` as it is with `[forth-out]` and `[bnf-in]` defined for the mode you want.

4.5.3 The loader files have been set up for use with GForth, use with another Forth system will probably require changes to the file paths. To run the examples with GForth:

- unzip the file `grace.zip` into a directory called `grace`
- edit the values of `[forth-out]` and `[bnf-in]` in file `mini.fth` for the required mode of operation
- type (note the forward slashes - back slashes won't work with GForth):

```
s" grace/mini/mini.fth" included <enter>
```

 which will compile or generate the Mini parser and run it with the example Mini program called `square.mini`. Note that in modes 2 and 3 the generated parser file will appear in the `mini` directory.
- To further test the mini program type (e.g.):

```
12 square <enter>
```

 which should then display the answer 144
- Repeat with another setting of the compilation constants

5 Installation and files provided

The following files are included in `grace.zip` which should be unzipped into a folder called `grace`:

To run Grace	
<code>graceloader.fth</code>	includes the grace files needed for parser generation.
<code>graycore.fth</code>	equivalent to <code>gray5.fs</code> less code generation and sets definitions
<code>gracegen.fth</code>	generates the parser as Forth source code
<code>graygen.fth</code>	compiles the parser in memory (<code>graycore</code> + <code>graygen</code> + sets is functionally equivalent to the original Gray)
<code>gracestt.fth</code>	State transition table for Grace (generated by LexGen)
<code>gracetokens.fth</code>	Declares the BNF symbols to be recognised by Grace in the grammar file
<code>grace.fth</code>	run-time action definitions for the Grace parser (holds the definitions described in)
<code>graceparser.fth</code>	the BNF parser (generated by Grace itself from the example in paragraph)
<code>ansify.fth</code>	library file with some common definitions
<code>xmini_oof.fth</code>	library file, extended Mini-OOF

sets.fth	library file for handling sets (originally extracted from Gray)
inputsources.fth	library file for using ANS words on text files
files.fth	library file to write to files
list.fth	library file for rudimentary list handling
sections.fth	library file for using ALLOCATED memory
lexscanner2_0.fth	a lexical scanner used with transition tables generated by LexGen
An example using the Mini example from the gray5.zip distribution	
mini.fth	includes necessary files and runs grace.fth to generate a Mini parser
minitokens.fth	defines the symbols and their tokens for the Mini language
ministt.fth	the state transition table for recognising symbols in a Mini language source file (generated by LexGen)
mini.bnf	BNF definition of the Mini language grammar (modes 1 and 3)
mini.gry	Gray format definition of the Mini language grammar (modes 0 and 2)
minidefs.fth	definitions to generate a forth source code parser for the Mini language from a BNF input file (mode 3)
minidefs_gray.fth	definitions to generate a forth source code parser for the Mini language from a Gray format input file (mode 2)
minirun.fth	run time definitions for the generated Mini language parser (modes 2 and 3)
miniactions.fth	the run-time file for the Mini language (mode 1)
miniactions_gray.fth	definitions to compile a parser into memory from a Gray format input file and then run it (mode 0)
square.mini	a Mini program
Documentation	
grace.pdf	This document

6 Errors

As with Gray error reporting is rather basic (see reference 1 for errors that can be reported in Gray and which are also reported in Grace) . The additional errors detected in Grace are:

1. Syntax errors when Gray has failed to recognise something in the input file, for example a missing semicolon will cause BNF2Gray to stop at the following ::= with a syntax error message.
2. When one of the memory buffers overflows when a message “Section overflow” will be displayed before the GForth trace messages. This can happen for the larger grammars and can be prevented by increasing the size of the section called `nonterm-buf` or `action-buf` in the file `gracegen.fth`.

7 Differences between Grace and Gray and restrictions

- 7.1 Actions are compiled in a different context in the two systems. In Gray actions are compiled as they are read in whereas the grammar is compiled later with calls to actions. In Grace actions are inserted into the

generated grammar. This could cause problems if the ANS Forth `BASE` or the compilation wordlist differs at the time of compilation

- 7.2 Actions within a production in Grace are compiled in-line with the parser, therefore they are in the same colon definition. In Gray they are compiled separately. This gives more flexibility in Grace e.g. use of the return stack between different action sequences.
- 7.3 Errors and warnings may be shown in a different position, a simple example is that an invalid word in an action will not show up until parser run-time in Grace, whereas Gray will report it immediately.
- 7.4 As Grace simply scans action text looking for a terminating `}` or `}}`, actions containing a string with embedded `}` characters will not be handled correctly whereas Gray will handle this correctly. This (unlikely?) problem can be avoided by defining the string in another word that is called in the action.
- 7.5 As the character `{` starts an action in BNF input, the Forth 200X syntax for locals cannot be used in actions. This can be handled by using them in another definition that is called in the action.
- 7.6 In Gray the user can choose the name of any production as the name of the grammar. In Grace the name of the first production is taken as the name of the grammar. For example in the Mini example the first production is:

```
Program ::= 'PROGRAM' ...
```

8 ANS compliance statement

(To be completed)

9 References

1. Gray by Anton Ertl from <http://www.complang.tuwien.ac.at/forth/gray5.zip>
2. LexGen from <http://www.qlikz.org/forth/lexgen>
3. Using LexGen with Gray, file `lgandgray.pdf` from the download in reference 2.