PCRC Fortran 90 and HPF Syntax Test Suite

D.B. Carpenter

Northeast Parallel Architectures Center,
Syracuse University,
111 College Place,
Syracuse, New York, 13244-4100

DRAFT

Abstract

The Fortran syntax test suite is designed as a test of Fortran 90 and High Performance Fortran (HPF) compilers. It targets the parsing, or syntax-checking, phase of the compilers.

The suite currently consists of 8 standard Fortran 90 programs and one HPF program. The programs were constructed synthetically from the formal syntax definition of the languages. They exercise every syntax rule at least once. The tables of syntax rules used to generate the codes are also supplied with the test suite.

The test programs are annotated with comments flagging the first use (within the file) of each syntax rule. When a parser erroneously flags a syntax error on a particular line, these annotations should enable rapid isolation of the unrecognised syntax rule.

1 The Codes

The codes are located in the directory `src/synthetic`.

The first eight codes

```
test1.f90
test2.f90
test3.f90
test4.f90
test5.f90
test6.f90
test7.f90
test8.f90
```

are standard Fortran 90. They are designed to exercise every BNF syntax rule in appendix B of the *Fortran 90 Handbook* [1].

By far the largest code is `test1.f90`. The remaining codes are only provided to test variants in the syntax rule deriving the main program. Since, by convention, a single file contains at most one main program, these variants were placed in separate files.

The final code

```
testh.f90
```

is an HPF program. It is designed to exercise every BNF rule in appendix B of the *High Performance Fortran Language Specification*, version 1.0 [2].

Presently the codes are provided only in Fortran 90 free format. Fixed format versions may be provided later.

The test programs were constructed specifically to test Fortran parsers, and by construction they are syntactically legal programs. They also respect the non-syntactic constraints of Fortran: variables are declared, expressions are properly typed, explicit procedure interfaces are provided, referenced modules are defined, referenced statement labels exist, etc, wherever these things are required by the rules of Fortran. Additionally all referenced procedures are defined. It should therefore be possible to compile and link all test programs. The results of *running* the programs is undefined—`test1.f90` in particular exercises all the file IO syntax of Fortran. Given the aims of the suite, it did not appear worthwhile to devise a program that would behave sensibly in an unknown file-system environment.

2 Trouble-shooting

The test programs are annotated with comments flagging the *first use* within the current file of each syntax rule. These comments will often help in isolation of a compiler bug. For example when one Fortran 90 compiler was used to compile the program `test1.f90` it produced the error message

```
f90: Error: test1.f90, line 2017: Syntax error, found ','
      when expecting one of: <END-OF-STATEMENT> ; <IDENTIFIER>
            <INTEGER_KIND_CON> <INTEGER_CONSTANT> (
  doc3 : DO, var123 = 1, 2
-----------^
```

Inspecting `test1.f90` we find

```
                                                          2012
! do-stmt                      R818(b)                    2013
! nonlabel-do-stmt             R820(a)                    2014
! loop-control                 R821(c)                    2015
```

```
                                                                    2016
  doc3 : DO, var123 = 1, 2                                          2017
    CONTINUE                                                        2018
  END DO doc3                                                       2019
                                                                    2020
```

The three comments on lines 2013-2015 specify the syntax rules which are used for the first time in the following statement, line 2017. These rule are given in full in the auxilliary file `rsymbols` (for Fortran 90 rules and `hsymbols` for HPF rules. For more discussion of the format of these files see section 3). Searching this file for the rules `R818`, `R820` and `R821` we find that the optional comma really is justified by the rule `R821(c)` for *loop-control*:

```
loop-control                    DONE
  R821(a)       , do-variable = scalar-numeric-expression, scalar-
numeric-expression, scalar-numeric-expression
  R821(b)       do-variable = scalar-numeric-expression, scalar-nu
meric-expression, scalar-numeric-expression
  R821(c)       , do-variable = scalar-numeric-expression, scalar-
numeric-expression
  R821(d)       do-variable = scalar-numeric-expression, scalar-nu
meric-expression
  R821(e)       , WHILE (scalar-logical-expr)
  R821(f)       WHILE (scalar-logical-expr)
```

For confirmation we can check rule `R821` in appendix B of the Fortran 90 Handbook, and verify that there are no additional constraints forbidding the comma. Apparently this particular compiler does not handle this syntax rule correctly (at least in this context).

3 Construction of the Test Suite

The test programs were constructed systematically to use all rules of Fortran 90 and HPF syntax, starting from the formal syntax references in the appendices of the language definitions.

The guiding strategy in constructing the programs was to approximate a depth-first, backtracking exploration of the syntax rule set. In practise we were more flexible about the order of visiting the rules. A strict depth-first approach would sometimes yield very unrealistic program sections, and would sometimes generate programs which did not satisfy the non-syntactic (i.e., the non-context-free) constraints of the language.

Wherever practical we follow the precise grammar given in the references. Occasionally some normalisation was required to make the rules more suitable for the our purposes (of course, a given language can be generated by many

different BNF grammars, and our normalisations do not change the set of legal programs)

The construction proceeds in three phases. For the sake of definiteness we call them *syntactic*, *lexical* and *semantic*.

3.1 Syntactic phase

The starting rule for a Fortran program is given in the Fortran Handbook as

R201 *executable-program* **is** *program-unit*
[*program-unit*]...

We copy this rule into a file called `rsymbols`, where it is written in the form

```
executable-program              IN PROGRESS
  //R201(a)     program-unit
                program-unit
                ...
  //R201(b)     program-unit
```

The `rsymbols` file plays a central röle in the construction. It is used to record syntax rules for grammar symbols as they are encountered. Before proceeding any further with the construction, we will describe the format of this file. The format is different to the *Handbook* format, because we wish to eliminate optional parts within single syntax rules, labelling every variant of a rule individually.

Wherever the Handbook notation for a rule allows optional parts, we split the rule into several sub-rules labelled a, b, c, ... (we do not regard this subdivision as a normalisation of the grammar, *per se*—just as a different notation for the original grammar). As another example, a rule with two optional parts, such as

R611 *substring-range* **is** [*scalar-int-expr*] : [*scalar-int-expr*]

will, when it is encountered, be split into *four* parts enumerating all combinations of options:

```
substring-range                 IN PROGRESS
  //R611(a)     scalar-int-expr : scalar-int-expr
  //R611(b)     : scalar-int-expr
  //R611(c)     scalar-int-expr :
  //R611(d)     :
```

Two conventions for repeated terms are used in the `rsymbols` file: `{X}...` means *one* or more repeats of `X` (on the current line), and `...` (on a line alone) means zero or more repeats of the previous line. The second convention is illustrated in rule `R201(a)` above. There are a few rules in the Handbook where optional parts appear inside repeats. For example

R529 *data-stmt* **is** DATA *data-stmt-set* [[,] *data-stmt-set*]...

In these cases sub-dividing rules and using the above repeat conventions are insufficient to eliminate optional parts in individual rules. In these cases we normalise the grammar, introducing new symbols. In the example we introduce a new symbol *data-stmt'* as follows

```
data-stmt                       IN PROGRESS
  //R529(a)     DATA data-stmt-set {data-stmt'}...
  //R529(b)     DATA data-stmt-set

data-stmt'                      IN PROGRESS
  //R529(c)     , data-stmt-set
  //R529(d)     data-stmt-set
```

Using these conventions we can label every variant of every rule in the syntax.

Returning to the start rule `R201`, we first select the first variant `R201(a)`. The right-hand side of the rule is copied to an initially empty file `test1.f90` to give

```
program-unit
program-unit
...
```

Meanwhile, in the `rsymbols` file we mark the first variant as *visited* by "uncommenting" it—removing the leading slashes:

```
executable-program              IN PROGRESS
  R201(a)       program-unit
                program-unit
                ...
  //R201(b)     program-unit
```

In the emergent test program the first symbol requiring expansion is `program-unit`. We copy rule `R202` to the `rsymbols` file as

```
program-unit                    IN PROGRESS
  //R202(a)     main-program
  //R202(b)     external-subprogram
  //R202(c)     module
  //R202(d)     block-data
```

uncomment the first variant in that file, and replace the first instance `program-unit` in the test program by the RHS of the rule to get

```
main-program
program-unit
...
```

Much later, when the expansion of the first `program-unit` in the test program is completed, the second `program-unit` symbol is encountered. Since the first rule for *program-unit* is already marked as visited (it was uncommented when it was used), we use the next rule R202(b), and uncomment that. Finally, when all rules for *program-unit* have been visited, the `IN PROGRESS` label for the symbol in `rsymbols` is replaced by `DONE`. Once a symbol is *done*, it is no-longer expanded if it is encountered in the test file (during this phase of the program generation). Any remaining occurrences of the symbol are left as non-terminals.

The default rule for the order of expansion of symbols in the test program is simply to expand the first non-terminal symbol in the current version of the program which does not have a "done" entry in `rsymbols`. This biases early occurrences of a non-terminal to expand to large expressions, because we usually put the variants of rules with all options selected first in the set of sub-rules for a symbol. If it appears that this ordering will lead to a very unbalanced, or semantically illegal program, the first candidate symbol is left unexpanded, and we look to the next candidate. The rules for the first symbol will eventually be used up by expanding later instances of it.

Where rules contain repeats (ellipses), these repeats are generally exploited where they will allow new sub-rules for *in progress* symbols to be used up. Otherwise, if they provide no such opportunity, the repeatable term is written just once in the test program. Sometimes a single sub-rule for a symbol will be used several times, to allow all sub-rules for some child symbol to be used up. For example `program-unit` is expanded several times by sub-rule `R202(b)` because there are two sub-rules for `external-subprogram` (function and subroutine definitions) and, in turn, several variations for functions and subroutines definitions.

Eventually, all symbols in `rsymbols` are marked "done". At this stage, all rules of the syntax have been exploited once, and the test program has grown into a very large syntactic term. It will still contain many non-terminal symbols.

As a useful by-product we have a file `rsymbols` containing all the syntax rules of the language.

3.2 Lexical phase

In this phase each remaining non-terminal symbol is replaced by a terminal or a string of terminals.

Each occurrence of an *xyz-name* symbol is replaced by a legal Fortran name. In general, every occurrence of such a symbol is given a *different* expansion. For example the first `variable-name` symbol encountered is replaced by `var1`, the second by `var2`, and so on. At this stage we do not worry about semantic requirements such as variables being declared before they are used. Every occurrence of a literal constant is given a value, usually a small value such as 0, 1 or 2 for an integer literal, or `'xx'` for a character literal. Other, higher-level

non-terminal symbols, are expanded, typically with the smallest fragment which is legal in the context.

As this phase proceeds, a table of all names (of program, variables, constants, procedures, dummy arguments, modules, local use aliases, common blocks, defined operators, derived types, derived type components, namelist groups, constructs, statement labels, block data subprograms, etc) introduced into the program is accumulated in a new file.

3.3 Semantic phase

In the final phase the program is "debugged". The static semantic constraints of the language are fulfilled. We ensure that variables are declared, expressions are properly typed, explicit procedure interfaces are provided, referenced modules are defined, referenced statement labels exist, and so on, wherever these things are required by the rules of Fortran. Definitions of all referenced procedures are added in the file which references them.

Mainly this phase just involves adding extra statements—usually declarations, or definitions of referenced program units. The procedure is guided by the "names" file accumulated in the lexical phase. Entries in that file are marked as the named entities are declared or defined. The procedure essentially terminates when all names are marked.

Occasionally (due to lack of foresight in the earlier phases) it may be necessary to change previously generated code fragments. One must then take care not to "lose" code which uniquely uses a particular syntax rule.

The whole procedure is repeated for the High Performance Fortran syntax, but in this case we only care about visiting the additional syntax rules of HPF. These rules are accumulated in the file `hsymbols`.

4 Testing the Test Suite

The Fortran 90 test programs have been tested with the IBM compiler, `xlf90`, the DEC compiler, `f90`, and the Parasoft compiler, `f90`. The IBM compiler fares best, only reporting one syntax error. The DEC compiler reports four syntax errors. The Parasoft compiler reports about two dozen. To the best of our knowledge all these reports are erroneous.

References

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, Jerrold L. Wagener, *Fortran 90 Handbook: Complete ANSI/ISO Reference*, McGraw Hill, New York, 1992.

[2] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 1.0*, 1993.