DemFGen Manual

Bryan Chadwick

September 8, 2009

Chapter 1

Introduction

The `DemeterF`class generator (*demfgen* for short) is a Java and C#[1], class/parser generator and data binding tool (similar to `DemeterJ`) that merges data structure and behavior descriptions into clean code, specifically suited for use with `DemeterF`. It supports modular files, parametrized class definitions, and parsing and printing of both non-generic and generic classes. The generator is part of the current `DemeterF` Library release, and no longer depends on the DemeterJ runtime[2]. All of the options we will discuss in this document are relavent for both Java and C# code generation, so we will freely switch between Java and C# when giving examples... be ready.

Running from the command line, the `DemeterF`class generator accepts three mandatory arguments: a class-dictionary (*CD*) file, a behavior (*BEH*) file, and an output directory. To simplify parametrized classes and interface definitions we use CD and BEH syntax modified from `DemeterJ` with a few additions that support generics and allow modular creation of CDs and libraries. Assuming the JDK and `javacc` (or MONO and `csjavacc`[3] for C#) are correctly installed, the generator can be called with a `--build` option to automatically compile the generated parser and classes. There are also a few other useful options: `--windows` supports running on Microsoft Windows Operating Systems, `--graph` supports generating a visual representation of the CD in the *DOT/Graphviz* language, `--noparse` suppresses parser generation completely, `--dgp:...` supports data-generic function generation providing various `toString` methods, `--pcdgp:...` supports per-class generation of methods like *getters* and *updaters*, and `--lib:...` automatically builds a JAR (or DLL) from the resulting generated code.

The rest of this chapter gives a quick introduction to the tool and its features, while the rest of the document provides a detailed description of file formats, options and code generation.

1.1 Quick Start

```
// test.cd: Sample CD File
package test;

IntList  = IntCons | IntEmpty.
IntCons  = <first> int <rest> IntList.
IntEmpty = .
```

Figure 1.1: CD File for simple `int` Lists

[1]The C# version is called `DemeterFCS`, or *demfgencs*

[2]DJ style traversal control is not built into the standard release.

[3]CSJavaCC is our port of JavaCC that generates C# parsers, available at `http://www.ccs.neu.edu/home/chadwick/demeterf/csharp/csjavacc.html`

Fig. 1.1 contains a simple CD file for structures representing Lisp-style lists of `int` in a package/namespace named `test`. In a CD file, abstract classes are defined using "`:`" providing variants separated by "`|`" (*e.g.*, `IntList`). Concrete classes are defined using "=" with field names contained in "`<>`" followed by their type (*e.g.*, `IntCons`). Concrete syntax can be placed between field definitions and before and after variant lists for parser generation. Fig. 1.2 is a simple Java BEH file that implements the method `length()` for `IntLists`. The `main` method calls the static parse method of `IntList` to parse an instance from a string.

```
// test.java.beh: Sample BEH File
IntList{{
  public static void main(String a[]) throws Exception{
      IntList lst = IntList.parse("1 2 3");
      System.out.println(lst);
      System.out.println(lst.length());
  }
  abstract int length();
}}
IntCons{{  int length(){ return rest.length()+1; }  }}
IntEmpty{{ int length(){ return 0; }  }}
```

Figure 1.2: Beh File for simple `int` Lists

Assuming the `DemeterF` package is correctly in the classpath and we have a directory `test` for generated files, we can generate the parser and classes for this example using the following command:

```
java demeterf test.cd test.java.beh ./test
```

After running this command, demfgen will generate four files for this CD in the directory `test`:

Java Files : `IntList.java`, `IntCons.java`, `IntEmpty.java`
JavaCC File : `theparser.jj`

The Java files contain the classes, field definitions, constructors, and any other text taken from the BEH file. In addition demfgen generates a `boolean equals(Object)` method and static `parse(...)` methods for all classes. Running `javacc` on `theparser.jj` will generate parser related files including `TheParser.java` that (in this case) supports the reading of our integer lists. We can then call `javac` on the Java files to compile them into `.class` files. Alternatively we can pass the `--build` option and demfgen will attempt to run `JavaC` and compile all the files automatically. There is also a corresponding `--lib:...` option that can be used to automatically build a JAR file from the compiled code. For example, the command:

```
java demeterf test.cd test.java.beh ./test --lib:test.jar
```

will generate a parser, Java files, and compile all of them into a JAR file named `test.jar`.

If it makes sense to develop CD/BEH files separately we can add to the original `test` package by using an `include` statement. If we also need the behavior of the defined classes then we can also include it in the top level BEH file. Fig. 1.3 shows an addition to the `test` package that defines a `Main` class containing two `IntLists`, separated by a colon.

The generated parser includes the parsing code for the included library, and the Java/C# files are generated with the specified behavior. If we had previously packaged this code into an external library, or had other hand written definitions, then we can use `nogen` in front of definitions to signify that Java/C# code should not be generated. Likewise to eliminate the generation of parse code for a specific class/interface we can prefix the definition with `noparse`, or both `nogen` and `noparse` can be signified by using the `extern` prefix. Note that only one of those prefixes may be used.

That wraps up the quick intro; the next chapter describes the syntax of CD and BEH files; chapter 3 discusses parametrized classes and the generation of generic classes. Chapter 4 details the generation of classes, canonical methods, parsers, and command line options. Chapter 6 finishes with a longer CD/BEH example.

```
// use.cd: Uses test.cd
include "test.cd";

package test;

MainC = <left> IntList ":"
        <right> IntList EOF.
```

```
// use.beh: Uses test.java.beh
include "test.java.beh";

MainC{{
  public static void main(String a[]) throws Exception{
      Main m = Main.parse("1 2 3:4 5");
      System.out.println(m.length());
  }
  int length(){ return left.length()+right.length(); }
}}
```

Figure 1.3: CD and BEH Files that use/include `IntList`

Chapter 2

File Formats

`DemeterF` uses CD and BEH input file formats that are simpler, but quite like to those of `DemeterJ`. Notably, the format allows the inclusion of other CD and BEH files that can define other packages, classes, and behavior, and prarmetrized definitions generate Java/C# generic classes and support unlimited nesting. Each file format will be described in detail in the following sections, while parametrized definitions and generated code will be discussed in later chapters.

2.1 *CD* Files

Below (Fig. 2.1) is an *EBNF* description of the class dictionary syntax used by `DemeterF`. CHAR stands for any (possibly escaped) non double quote character. Double quotes and others (newline, tab, backslash, etc.) can be escaped as usual. DIGIT is any character `'0'` through `'9'` and IDENT is a normal Java/C# identifier, which may also include underscores (_) or dollar signs ($)[1].

```
CDFile  ::= Include* [ PackageDef ] [ LookAhead ] Import* TypeDef*
String  ::= "\"" Char* "\""
Integer ::= Digit Digit*
Include ::= "include" String ";"
PackageDef ::= "package" Ident ("." Ident )* ";"
LookAhead ::= "lookahead" "=" Integer ";"
Import  ::= "import" Ident ("." Ident)* [ ".*" ] ";"

TypeDef ::= [ Modify ] (Class | Intfc)
Modify  ::= "nogen" | "noparse" | "extern"
Syntax  ::= String | "EOF"
Class   ::= Decl "=" [ Use ("|" Use)* ] (Field | Syntax)* [ Impl ] "."
Intfc   ::= "interface" Decl ":" Use ("|" Use)* "."

Field   ::= "<" Ident ">" Use
Decl    ::= Ident ["(" ParDef ("," ParDef)* ")"]
ParDef  ::= Ident [":" Ident ]
Use     ::= Ident ["(" Use ("," Use)* ")"]
Impl    ::= "implements" Use ("," Use)*
```

Figure 2.1: CD File Syntax

[1]The dollar sign is not part of a valid C# identifier, though the Java specification allows it.

A CDFile begins with optional Include, PackageDef, LookAhead, and Import statements (in that order). Package and import statements declare the package and imports to be applied to all generated classes defined directly in the current file. The lookahead declaration is carried over to the parser, setting the global lookahead option for `JavaCC` (or `CSJavaCC`). Class and interface definitions follow; defining *abstract* classes (or *sum/variant* types) and/or `concrete` classes (or *product/record*) types). Abstract classes are defined using a colon ("`:`"), with subclasses separated with a bar ("`|`"). Concrete classes are defined using equals ("`=`"), with field names in brackets (`< >`) followed by their type. Note that we eventually generate Java or C# code, so field names must be valid identifiers in the language, which excludes keywords such as `for`, `public`, and `return`.

Definitions can be modified with "`nogen`", stating that code should not be generated for them, with "`noparse`", stating that it should not be included in the generated parser, or with "`extern`", stating that the definition should be ignored for both code and parser generation[2].

```
// simple.cd
package mypkg;

import java.io.PrintStream;

Pair(X) = <left> X <right> X.
FltPair = "(f" <p> Pair(Float) ")".
IntQuad = "(i" <q> Pair(Pair(Integer)) ")".

interface Concrete : IntQuad | FltPair.
```

Figure 2.2: Example CD File with parametrized classes

Fig. 2.2 contains a CD that describes the package `mypkg` with three classes (`Pair`, `FltPair`, and `IntQuad`), one of which is *generic*. It also defines an interface (`Concrete`) that is implemented by the two non-generic classes. Each of the generated files will import `java.util.PrintStream` since it will be used in the example BEH file. The quoted strings in the definitions of `FltPair` and `IntQuad` define the *concrete syntax* of the data structures, while the generated java files make up there *abstract syntax*. The generated parser will parse a string like `"(i 2 4 6 8)"` as an `IntQuad`, resulting in an object similar to the construction expression:

```
new IntQuad(
  new Pair<Pair<Integer>>(
        new Pair<Integer>(2,4),
        new Pair<Integer>(6,8)))
```

Java/C# style single/multi-line comments can be used within the CD file to document classes, comment them out, or anything else that is useful. Once we have structural definitions we can add behavior (methods, etc.) to classes using BEH files.

2.2 *BEH* Files

Behavior files allow methods and other syntax to be placed in classes separate from their structural definitions. Fig. 2.3 defines the simple behavior file syntax in *EBNF*, reusing the Include definition from before. Text is any string that does not include "}}".

Any number of BEH files may be included, supporting the corresponding inclusion of related CD files. The Text for a given class name is placed within the body of the class or interface definition during code generation. An example behavior file for the earlier CD is given in Fig. 2.4 . Note that the *type parameter* of `Pair`, `X`, is the same as in the CD definition, ßsimple.cd. We don't include any other files, but if needed we could split this into multiple files or include other library behavior.

[2]But not for DGP, which will be explained later

BEHFile ::= Include* Beh*
Beh ::= Ident "{{" Text "}}"

Figure 2.3: BEH File Syntax

```
// simple.beh
Pair{{
   X getLeft(){ return left; }
   X getRight(){ return right; }

   void print(PrintStream p){
       p.println("("+left+", "+right+")");
   }
}}
FltPair{{
   float product(){
       return (p.getLeft()*p.getRight());
   }
}}
```

Figure 2.4: Example BEH File with parametrized classes

```
package mypkg;
import java.io.PrintStream;

public class Pair<X>{
  public X left;
  public X right;

  /* ... */

  X getLeft(){ return left; }
  X getRight(){ return right; }

  void print(PrintStream p){
     p.println("("+left+", "+right+")");
  }
}
```

Figure 2.5: Java code for `Pair`

The resulting class definition for `Pair` after inserting behavior is shown in Fig. 2.5 . The generated constructor and *canonical* methods (`toString`,`equals`, *etc.*) are left out to save a little space. The added methods, actually, any text between "{{" and "}}", are placed in the generated class after any generated code. More than just methods may be added to class bodies. Within the BEH file you can insert any legal Java/C# code, including comments, static/private fields, and alternate constructors. Because the BEH declarations are placed directly within the class definitions, care must be taken to provide syntactically and type correct code.

Using the `--build` option (or likewise `--lib:...`) causes `DemeterF`to build the generated code using `javacc/csjavacc` and `javac/gmcs`. On Microsoft Windows the `--windows` option can be used, which will compile using Microsoft's `csc`, rather than searching for `Mono`. This functionality has been tested on Linux and Unix (Solaris), and with Windows using the `--windows` opotion or the Cygwin shell. Any reports of experience with other systems (Mac OSX?) would be appreciated.

Chapter 3

Generics

Parametrization can help a great deal when abstracting data structures for algorithms. In the CDFile syntax (Fig. 2.1) you can see that a Decl may include a comma separated list of ParDefs, *type parameters*, in parenthesis. The existence of type parameters defines a generic class, while a *use* of a parametrized type allows us to generate a `parse`/`print` method for each specific use, or instantiation, of a generic class.

Fig. 3.1 is an example of a generic CD file that demonstrates a simple use of type parameters. It defines a single class (`Triple`) in the package `lib` that represents a generic triple of the given type. A portion of the generated Java file (`Triple.java`) will look like that in Fig. 3.2 .

```
// lib.cd
package lib;

Triple(T) = <l> T <c> T <r> T.
```

Figure 3.1: CD File with a parametrized class

```
public class Triple<T>{
    public T l;
    public T c;
    public T r;

    public Triple(T l, T c, T r){ /*...*/ }

    /* ... parse/equals methods ... */
}
```

Figure 3.2: Generated Java for `Triple`

Because generic classes have type parameters, we can substitute different types, write more generic algorithms, and abstract more structure, while retaining our ability to parse, print, and compare objects. In Java and C#, class definitions we can place *bounds* on type parameters that specifiy a type (or interface) that any class used as a parameter must extend (or implement). The default bound in both cases is `Object`. In `DemeterF` we can specify a bound on a type parameter by adding it to the parameters of a Decl. For ParDef we use a colon to signify a bound on a type parameter; an example is shown in Fig. 3.3 .

In order to have size for a generic `Pair`, we introduce the interface `Size` that contains a single `size()` method. The `Pair` implementation of `size()` can then use `size()` on `a` and `b`, while retaining the seperate types. The Java compiler will also check that any use of `Pair<...>` is given type parameters that implement the `Size` interface.

```
// bound.cd
package bound;

interface Size: Apple
              | Orange
              | Pair.

Pair(A:Size, B:Size) = "(" <a> A <b> B ")".

Apple = "apple" <c> String <s> int.
Orange = "orange" <s> int.
```

```
// bound.beh
Size{{ int size(); }}
Apple{{
  public int size(){ return s; }
}}
Orange{{
  public int size(){ return s; }
}}
Pair{{
  public int size(){
    return a.size()+b.size();
  }
}}
```

Figure 3.3: CD/BEH File with parametric bounds

Parameterized definitions in `DemeterF` are actually more usefull than Java/C# generic classes because of the mix of structure and syntax in the definitions. We can actually create definitions that are parametrized by syntax, which will be expanded by `DemeterF` automatically when needed (*e.g.*, for parser generation).

```
// syntax.cd
CommaList(X) = <lst> SepList(Comma,X).
BarList(X) = <lst> SepList(Bar,X).

// Open interface for various seperators
interface Syntax : Comma | Bar.
Comma = ",".
Bar = "|".

// Generic Seperated List
interface SepList(S:Syntax,X) : NeSList(S,X) | SEmpty(S,X).

NeSList(S:Syntax,X) = <first> X <rest> SList(S,X).
SList(S:Syntax,X) : SCons(S,X) | SEmpty(S,X).
SCons(S:Syntax,X) = <s> S <first> X <rest> SList(S,X).
SEmpty(S:Syntax,X) = .
```

Figure 3.4: CD File with parametric bounds

Fig. 3.4 shows a generic implementation of seperated lists. The syntax that separates elements of the list is given as the first parameter of the class, while the second is the elements to be stored. The two definitions

Chapter 4

Class/Parser Details

In earlier chapters you've seen a few examples of CD files and generated classes. In this chapter we discuss generated canonical and perser methods, including details (and possible pitfalls) of class and parser generation.

4.1 Generated Methods

```
public X(Y y, ..., Z z)
boolean equals(Object o)
String toString()
```

4.2 Field Classes

4.3 Parsing

```
X parse(String s) throws ParseException
X parse(java.io.InputStream in) throws ParseException
```

4.4 Pitfalls

Type parameter mismatches. Parsing with type parameters. Interfaces/sum types and parsing (order, empty string, etc...)

Chapter 5

Data-Generic Programming (DGP)

Chapter 6

Larger Example

Larger Grammar/Program Example(s)...