FLPR

Release 0.1.0

Jun 11, 2020

Basics

flipper *noun*

1. A component of pinball machines, which were a staple of 1970s arcades but still have ardent fans.
2. One who renovates neglected properties for profit.

CHAPTER 1

Introduction

The *Fortran Language Program Remodeling* system, or FLPR, is a C++ library for manipulating Fortran source code. You could use FLPR to quickly develop small code maintenance utilities, or to build something as ambitious as a source-to-source translation tool to implement language extensions for your project.

CHAPTER 2

Distinguishing features of FLPR

- Layout and comment management:
  - Gives developers recognizable code when debugging transformed source.
  - Preserves important knowledge stored in comments.
- Proper extensible grammar:
  - Better than regexp at recognizing complex structures.
  - Allows implementation of application-specific language features.
- Compact API allows you to quickly write "disposable" utilities for doing specific code transformations.

Any questions? Please file an issue on GitHub, or email flpr-dev@lanl.gov

2.1 Getting Started

This page provides information on how to get FLPR installed, and gives some pointers on exploring sample applications.

Contents

2.1.1 Installing FLPR

Prerequisites

FLPR has a few external tool requirements. Their minimum versions are:

- CMake v3.12
- C++17 support. FLPR is regularly built with gcc8, gcc9, and clang8
- flex v2.6.4. Note that any compilation warnings/errors about C++17 not accepting the `register` keyword are due to using an old version of `flex`.
- gperf v3.1 (required only if changing one particular source file)

Obtain FLPR

The main home for FLPR is on GitHub, and it can cloned with:

```
$ git clone https://github.com/LANL/FLPR.git
```

Configure

FLPR uses CMake to manage the configuration process. Make sure that you have a `flex` and modern compiler available in the path and the configuration is as simple as:

```
$ mkdir build && cd build
$ cmake /path/to/flpr
```

There are two types of cmake build types supported: `Debug` or `Release`. These build configurations change the compiler flags being used. Development work should always be done in `Debug` mode (the default), while deployed builds should be done with `Release`. Specify a build type using *one* of these defines after the cmake command:

```
-DCMAKE_BUILD_TYPE=Debug
-DCMAKE_BUILD_TYPE=Release
```

Build

Once CMake has completed, FLPR can be built with make:

```
$ make
```

Note, by default, the FLPR test suite is built, but not executed. To run the tests, simply do:

```
$ make test
```

While the Sphinx documentation (what you are reading now) is not yet integrated into CMake, the code documentation is, and can be built with:

```
$ make doxygen
```

The code documentation is not installed, but is found in the `docs/html/index.html` file under the build directory.

Install

To install FLPR, just run:

```
$ make install
```

FLPR installs files to the `lib`, `include` and `bin` directories of the `CMAKE_INSTALL_PREFIX`. Additionally, FLPR installs a CMake configuration file that can help you use FLPR in other projects. By setting `FLPR_DIR` to point to the root of your FLPR installation, you can call `find_package(FLPR)` inside your CMake project and FLPR will be automatically detected and available for use.

2.1.2 Sample Applications

The FLPR library also contains some sample applications, which are intended to a feeling of what FLPR application development is like. The samples can be found in the `apps/` directory, and they are built and installed during the normal build process.

caliper.cc Demonstration program to insert fictitious performance caliper calls in each external subprogram and module subprogram (not internal subprograms). The caliper calls include the subprogram name as an actual parameter, and mark the beginning and end of each executable section. The executable section is scanned for conditional return statements: if they exist, a labeled continue statement is introduced about the end caliper, and the return is replaced with a branch to that continue.

ext_demo.cc Demonstration of how to extend an *action-stmt* parser. This example introduces a rule that accepts a comma after the *io-control-spec-list* of a *write-stmt*, which, while not allowed by the standard, is accepted by many Fortran compiler front-ends.

flpr-format.cc A weak attempt at an analogue to `clang-format`. It can do fixed-form to free-form conversion, reindent code using specified rules, remove empty compound statements (e.g. convert `foo; ; bar` to `foo; bar`), or split compound statements. See *Extending sample applications with FLPRApp* for details on why the structure of this file is unusual.

flpr_show_cst.cc Shows the concrete syntax trees (Stmt_Tree) generated by all statement parsers that match statements entered on standard input. This can be used to verify the shape of the CST for some statement. Note that you can get multiple parsers that accept an input. For example, a string accepted by *assignment-stmt* is also accepted by *action-stmt* and *forall-assignment-stmt*.

module.cc Demonstrating how to selectively insert a *use-stmt* into subprograms that contain a *call-stmt* to a particular name. You may want functionality like this when moving old code into modules.

parse_files.cc Just has FLPR build parse trees for a list of inputs. You can use this to see if FLPR would run into any problems on your code base.

2.1.3 Extending sample applications with FLPRApp

The sample application `apps/flpr-format.cc` is a thin wrapper around calls to `apps/flpr_format_base.hh`. The idea behind this is to give you the opportunity to extend the input grammar, without having to replicate all of the flpr-format functionality. This structure applies to the `module` demonstration utility as well.

The flpr_format_base files are exported as a CMake package called FLPRApp, which contains application-related includes and libraries. To make an extended version of flpr-format, you need to add a `find_package(FLPRApp)` to your CMake configuration (in addition to `find_package(FLPR)`), and then build something that looks like `apps/flpr-format.cc` but registers FLPR statement extensions before looping across the files.

2.2 Coding Practices For Contributors

This page provides information on project expectations for C++ coding practices. Contributors should check that these practices are followed before submitting a pull request.

Contents

2.2.1 C++ Code Layout

FLPR uses `clang-format` to manage C++ code layout. There is a `.clang-format` file provided in the root directory, which is a copy of the LLVM style layout. Please run `clang-format` before commiting code: other developers should be able to run `git pull`, then `clang-format`, and see only their own changes.

2.2.2 C++ Language Standards and Portability

It is expected that the FLPR code base can be compiled with a C++ '17 compliant compiler, without compiler-specific extensions. If possible, please try both current GCC and LLVM compilers for portability.

2.2.3 Code Documentation

Adding Doxygen documentation blocks for files and functions is encouraged. Please use the `Qt` style of block comment without intermediate "*":

```
/*!
  \file somename.cc
  ... more doxygen text ...
*/
```

2.2.4 File Names

The following file name extensions are to be used:

.hh & .cc C++ language header and source files

.l Flex input

.md Markdown-style documentation

.rst Restructured Text-style documentation

File names should not depend on case-sensitivity for uniqueness.

2.2.5 Include Guards

The code section of a C++ header file should be wrapped with an include guard `#ifdef`. For the file `FLPR/src/flpr/Stmt_Tree.hh`, the guards should look like:

```
#ifndef FLPR_STMT_TREE_HH
#define FLPR_STMT_TREE_HH 1

... declarations ...

#endif
```

2.3 FLPR Application Development

This (very incomplete) section discusses designing and developing an application using FLPR.

2.3.1 Types of FLPR Applications

We tend to categorize FLPR-based applications into two modes of operation: one-shot vs. continuous transformations. By "one-shot" transformation, we mean that the application is used to transform the source code once, and the resultant code is used by developers from that point forward (i.e. committed to a repository). An example of a one-shot code transformations are code modernization tasks, such as transforming `real*8` to `real(kind=k)`. By "continuous" transformation, we mean that an application is called as a souce preprocessor before each and every compilation pass. Continuous transformation may be used to translate application-specific extensions to Fortran, or to inject compiler-specific optimization directives. FLPR is intended to make writing one-shot utilities quick and easy, while giving you the power to build production-level tools to include in your system.

2.4 Overview of FLPR Data Structures

This attempts to give a high-level overview of the various FLPR data structures and how they relate to one another. This hierarchy of data structures is built up from the low-level text representation up to the parsed files.

One note on design philosophy. FLPR is designed to manipulate source code in "gentle" ways, allowing a program to make minor changes without radically changing the appearance of the source code. To do this, it tracks a lot of physical layout data that is not present in compilers. However, it is also not intended to be a compiler front-end, so it may not currently provide the analysis and inquiry tools that you would expect in those.

2.4.1 Lexical Level

These are low-level entities related to the manipulation of Fortran input text. Considering that FLPR is designed to preserve or manipulate source code, including formatting, these classes are more complex than you may find in

a normal compiler. They must handle things like lexemes being split across physical lines and multiple statements compounded together on a line.

Line_Buf (Logical_File::Line_Buf) A buffer of raw strings representing Fortran source code.

File_Line A representation of the textual layout of a single raw source line. This separates the line into "fields", which describe parts of the line that Fortran treats specially (e.g. labels, continuations, trailing comments, etc). Line_Buf entries are converted to File_Line by `File_Line::analyze_fixed` or `File_Line::analyze_free`, depending on the input style. The File_Line is also used as a template for reformatting source lines.

Examples of fields in File_Line include `left_txt` (control columns for fixed form), `left_space` (whitespace between `left_txt` and `main_txt`) and `main_txt` (the body of the Fortran line). File_Line also provides line-level categorization flags, including `comment`, `blank`, `label`, etc.

Token_Text The representation of a single "atom" of input. Token_Text includes the lexeme (the actual text), its token value as determined by the lexer, and some textual layout information.

Logical_Line A contiguous set of File_Lines that belong together, such as (continued) statement(s), or a comment block. This structure maintains the format of the original input, so that it can be re-emitted in a similar form. NOTE: there may be more than one statement in a Logical_Line if they have been compounded with semicolons.

The Logical_Line is a fundamental data structure that: groups File_Lines; presents a sequence of Token_Text representing the Fortran tokens; and organizes Token_Text into ranges for categorization.

LL_TT_Range This is more of an intermediary than anything, but it is the base of other classes. The "Logical Line Token Text Range" is a class that allows a range of Token_Text in a given Logical_Line to be handled as a stand-alone entity, storing references to both the TT_Range and to the source Logical_Line.

LL_Stmt The "Logical Line Statement" is meant to represent one Fortran statement. Derived from LL_TT_Range, it provides all the physical layout associated with a range of tokens in a Logical_Line, and acts as anchor for (optionally) holding a concrete syntax tree for the statement (see next section). An LL_Stmt also holds any comment block found above the Fortran statement. This is to allow a comment to stay anchored to a statement as text moves around.

Logical_File A sequence of Logical_Lines and LL_Stmts that make up a file, plus other identifying information. This is the level at which Logical_Lines are inserted, modified, deleted or split.

2.4.2 Grammar Level

The results of parsing sets of input lines are stored as trees, generically defined in `Tree.hh`. Each node in the tree can store data, and can have an arbitrary number of branches (children).

To make tree traversal a little more convenient, the idea of a "cursor" is employed. This is effectively an iterator that can move in four directions: "up" (go up a level in the tree), "prev" (go to previous node at this level sharing up), "next" (go to next node at this level), and "down" (go down to the first branch off of this node). Dereferencing a cursor with $\star$ or $->$ returns a reference to the user data stored at the current node.

Stmt::Stmt_Tree A Concrete Syntax Tree (CST) representing the structure of one complete Fortran statement. These encode the productions of the statement grammar, and are produced by the rules in `parse_stmt.cc`. Dereferencing a Stmt_Tree cursor returns an ST_Node_Data struct (defined in `Stmt_Tree.hh`), which contains the root syntax tag for the statement, and the LL_TT_Range that covers the statement.

Prgm::Prgm_Tree A tree of Stmt_Trees, organizing Fortran statements into larger constructs, parts, and procedures. These are produced by rules in `Prgm_Parsers_impl.hh`. Dereferencing a Prgm_Tree cursor returns a Prgm_Node_Data struct (defined in `Prgm_Tree.hh`). If the cursor is an interior node of the tree (`!is_stmt()`), the Prgm_Node_Data can be used to obtain the range of LL_Stmts covered by this node. Otherwise, either the `ll_stmt()` or the `stmt_tree()` can be accessed direcly from the Prgm_Node_Data.

Procedure This adapter class takes a Prgm_Tree cursor to a procedure (function-subprogram, subroutine-subprogram, main-program, or separate-module-subprogram) and creates (possibly-empty) LL_Stmt ranges across important regions of the procedure. This form is a lot easier to manipulate than the Prgm_Tree. For example, this can return an iterator to just the USE statements, avoiding a lot of condition testing in client code. This is the easiest mechanism to use for changing statements in a particular section.

2.4.3 Aggregate Objects

These are the high-level entities that should be used by most client applications to interrogate or manipulate Fortran source code.

Parsed_File A lazy-evaluation container for a Logical_File and associated Prgm_Tree. This class also provides indentation functionality.

Procedure_Visitor Given a Parsed_File, call a provided function on each procedure.

2.4.4 Missing Things

Unfortunately, concrete syntax trees (CSTs) are a real pain to work with, as they represent the organization of tokens, rather than that of language concepts. This is why compilers generally skip this step and go right to abstract syntax trees (ASTs). However, for gently manipulating the text in a line of source code, a CST is the way to go. FLPR needs to add AST support, or the functional equivalent thereof, for easier interrogation (e.g. "what subroutine name is being specified in this call-stmt?")

As of yet, there are also no symbol tables ("symtabs") in FLPR. These are being worked on, and also drive the need for AST functionality.

Finally (for now), it would also be useful to have some stmt-label management, to easily identify unused labels or unreachable code. This requires better parsing of things like write-stmt and computed-goto-stmt.

2.5 FLPR Library Development

Notes on extending the FLPR library itself.

coming soon...

2.6 Standards Conformance

This document discusses the language that FLPR recognizes, and where there are deficiencies.

2.6.1 Standards Basis

This implementation is based on the 28-DEC-2017 Draft N2146 of the ISO/IEC JTC1/SC22/WG5 Fortran Language Standard, available from the Fortran Working Group. As such, it targets the language of the Fortran 2018 standard.

2.6.2 Explicit Deviations

There are several situations where we have elected not to conform with the Fortran specification:

1. The FLPR parser *is sensitive* to whitespace in the input text in *both* free and fixed source form. This is a deviation from Section 6.3.3.1.2. If you have very traditional fixed form source code which does not use spaces to delimit names and keywords, you will need to preprocess that source code with another tool before processing it with FLPR.
2. The FLPR parser does not recognize any nondefault character set.
3. FLPR does not enforce the constraints of the standard, and thus, is *not* a standard-conforming processor. See the *Potential "gotchas"* section for an example.

2.6.3 Retro-Extensions

Some features that have been deleted or made obsolescent in the current standard have been implemented to assist modernization efforts:

- FORTRAN 77 *kind-selector* (e.g. `real*8`)
- *computed-goto-stmt*
- *arithmetic-if-stmt*
- *forall-stmt*
- *label-do-stmt*
- *nonblock-do-construct*: FLPR introduces a non-standard *executable-construct*, called *nonblock-do-construct*, to handle parsing of F2008 *action-term-do-construct* and *outer-shared-do-construct*.
- The *concurrent-locality* rule for the CONCURRENT clause of *loop-control* has been made optional to allow *loop-control* to match F2008 *forall-header* CONCURRENT clauses.

Note that Hollerith constants are *not* implemented.

2.6.4 Incomplete Sub-statement Parsing

There are situations where FLPR does not build a completely elaborated concrete syntax tree for a statement. For example, the syntax rules:

- *expr*
- *io-control-spec-list*
- *position-spec-list*

are examples of incomplete parsers in FLPR (there are many). The expected behavior is that FLPR will indicate a range of tokens that would be matched by these rules, but present them as a simple sequence rather than as a tree.

FLPR is implemented in a top-down fashion, so the user should expect that FLPR can parse the general structure of a *program*, but the details of an individual statement may not be fully elaborated.

2.6.5 Known Deficiencies

There are some parsers that simply have not been implemented yet. The current list includes:

- *block-data*
- *change-team-construct*
- *critical-construct*
- *submodule*

These will be implemented soon, but if one in particular is holding you up, please file an issue on GitHub. Contributions are always welcome!

2.6.6 Potential "gotchas"

- FLPR ignores Fortran `include` directives and all preprocessor symbols, so it may get confused if you depend on macro expansion or file inclusion to produce valid Fortran. Note that there is an internal FLPR preprocessor, but it only works when directed to.
- FLPR parsers produce *concrete* syntax trees, which include all of the punctuation and layout information. These are challenging to extract information from. See the function `has_call_named()` in `apps/module.cc` for an example of how to traverse the concrete syntax tree. Going forward, more abstract representations will be introduced.
- Note that FLPR enforces the **rules** of the standard, but not the **constraints**. This will allow FLPR to accept input that will be rejected by a conforming compiler. For example, rule R1164 defines *sync-all-stmt* as "SYNC ALL [([*sync-stat-list*])]", and constraint C1171 specifies that a specifier may not appear in the *sync-stat-list* more than once. FLPR does not enforce that constraint.
- When looking for an *action-stmt* in the input to transform, remember to look "inside" *if-stmt*, which contain an *action-stmt*!

2.7 Why FLPR?

A fair question to ask is "why did you develop FLPR when there are several available open-source Fortran front-ends?" The main reason is that FLPR is designed for performing complex transformations on Fortran source code, whereas compiler front-ends are designed to produce intermediate representations of the code semantics. While both toolsets require an understanding the syntax of a Fortran source file, each toolset has a very different view of what is *important* about that file. For FLPR, the most important thing to capture is the layout of the text in the file, while a compiler front-end wants to discard that information as quickly as possible.

For simple tasks like converting `real*8` to `real(kind=k)`, generic text transformation tools like `sed` and `awk` can easily make changes across a large number of source files. However, consider the following one-shot modernization task. Suppose your project uses the convention that uppercased dummy arguments in subroutines implies that they are output arguments, and lowercase implies that they are input:

```
c     fixed-format file
      subroutine foo(n,m,S)
      integer n,m,S,locali
      ...
```

You can use FLPR to transform the type declarations into a modern form:

```
! free-format
subroutine foo(n,m,S)
  integer, intent(in) :: n,m
  integer, intent(out) :: S
  integer :: locali
  ...
```

Doing so requires syntactic information that would be extremely complicated to encode in terms of the regular expressions required by generic text transformation tools.

Another reason for using FLPR over most other tools is that it has an extensible grammar. You can easily register new *action-stmt* or new *other-specification-stmt* parsers (extending block-like structures is on the TO-DO list). With this capability, you can implement continuous transformation tools that translate application-specific statements into standard Fortran. Some examples where this might be useful are:

1. Replacing boilerplate code with simple-to-maintain statements;
2. Testing out potential new standard Fortran language features;
3. or adapting code to particular platform characteristics.