

In this assignment you will review or learn how to build scanners and parsers using the tools flex and bison by building scanners and parsers for OpenAnalysis test input. You should work in groups of size two.

## 1 Motivation

OpenAnalysis is a program analysis toolkit that is part of an ongoing research project to determine how to separate program analysis from the specifics of the intermediate representations used by different compiler infrastructures. The main idea is that abstract analysis-specific IR interfaces are designed so that many different analysis algorithms can use the interface. To use the analysis algorithms within a specific compiler infrastructure the IR interfaces must be implemented for that infrastructure. Currently we can only test an analysis implemented within OpenAnalysis by using an IR Interface that has been implemented for a specific compiler infrastructure. This is cumbersome because often compiler infrastructures are significant pieces of software that have limited portability and whose IR Interface implementation might itself be buggy.

The goal of this assignment will be to build scanners and parsers for test input to the alias IR interface. Alias analysis is arguably the most important analysis because many other analyses depend on it. The test input only has information required by alias analysis and does not depend on any existing compiler infrastructure.

## 2 Alias Analysis

Alias and pointer analysis determines what program state each memory reference might access. In the C code shown below, the memory reference `*p` could access the location for variable `x` or the location for variable `y`. To perform conservatively correct pointer analysis, the alias analysis must indicate that `*p` could reference either (unless `gFlag` happens to be a constant at compile time).

```
int gFlag;
int main()
{
    int x, y, *p;

    if (gFlag) {
        p = &x;
    } else {
        p = &y;
    }

    *p = 4;
}
```

Alias analysis is very difficult to solve. There are a number of heuristics that have been developed that provide a broad spectrum of the tradeoff between accuracy and efficiency. In OpenAnalysis we believe we have developed an IR interface for alias analysis that can support all of the heuristics that do not use type information. The alias IR interface requires only a subset of the information that might be available within any IR. It needs to know what memory references are within each statement, a location description for each variable, and also if there are any pointer assignments within a statement (eg. `p = &x`).

For this assignment you will be implementing scanners and parsers that translate test data for the alias analysis in OpenAnalysis into the data structures required by the alias IR interface. Here is an example input file:

```
// int main() {
PROCEDURE = { < ProcHandle("main"), SymHandle("main") > }

// int x;
LOCATION = { < SymHandle("x"), local > }
// int *p;
LOCATION = { < SymHandle("p"), local > }

// all other symbols visible to this procedure
LOCATION = { < SymHandle("g"), not local > }

// x = g;
MEMREFEXPR = { StmtHandle("x = g;") =>
  [
    MemRefHandle("x_1") => NamedRef ( DEF, SymHandle("x") )
    MemRefHandle("g_1") => NamedRef ( USE, SymHandle("g") )
  ] }

// p = &x;
MEMREFEXPR = { StmtHandle("p = &x;") =>
  [
    MemRefHandle("p_1") => NamedRef( DEF, SymHandle("p") )
    MemRefHandle("&x_1") =>
      NamedRef( USE, SymHandle("x"), addressOf = T, accuracy = full )
  ] }
PTRASSIGNPAIRS = { StmtHandle("p = &x;") =>
  [
    < MemRefHandle("p_1"), MemRefHandle("&x_1") >
  ] }
```

The commented out statements are the C program statements that the input describes. Each type of information will have its own scanner and parser, because each analysis IR interface will use

some set of the available information types and the goal is to keep things as modular as possible. In other words, there will be a scanner and parser to parse the PROCEDURE information, LOCATION information MEMREFEXPR information and PTRASSIGNPAIRS information. The strings that you will be parsing lie within the curly braces. This is my second (and not last) round of designing these test input files. I would be happy to hear any ideas you might have for improving the overall software design and input syntax for the testing framework.

We will be covering alias analysis in more detail later in the course. For now the description I have provided here is should provide you sufficient background to do the assignment.

A more detailed description of OpenAnalysis is the paper "Representation-Independent Program Analysis" which can be found at <http://www.mcs.anl.gov/~mstrout/papers.html>.

### 3 Getting Started

1. Install a copy of OpenAnalysis, which is available via a CVS repository at Rice.
  - (a) Follow the directions at <http://www.hipersoft.rice.edu/cvs/index.html#anonymous> to get anonymous access to the CVS repository at Rice.
  - (b) Checkout the FIAlias branch of OpenAnalysis

```
// from your class account (or normal CS account)
% cvs co -r FIAlias OpenAnalysis
```

- (c) Build OpenAnalysis

```
% cd OpenAnalysis
% setenv CXXFLAGS '-g -O0'
% make -f Makefile.quick configure
% make -f Makefile.quick install
```

2. Get a copy of the initial project1 files

```
% cp ~cs553/project1/Project1.tar .
% tar xf Project1.tar
```

3. Build project1

- (a) Edit the Makefile to indicate the location of the OpenAnalysis directory. The platform is probably i686-Linux. Do an ls in the OpenAnalysis directory after building OA to check.
  - (b) Build it, you have to do make twice because of some bug in the make file. If you know the fix, please share it with me.

```
% make
% make
```

Building the project will result in two executables. The `testSubIR` executable tests the creation of the subsidiary IR that is then used by `TestAliasIRInterface` to implement the alias IR interface. The `driver` executable takes an input file on the command line. The input file needs to

be parsed to generate the same kinds of datastructures that are being created in testSubIR. The input file `Input/testSubIR.oa` corresponds to calls made in testSubIR. The driver executable calls the `top_parser` function which in turn should call the appropriate sub parser for each string within a curly brace. The sub parser for `PROCEDURE = { ... }` has already been implemented as an example. Your goal is to make calls to the `SubsidiaryIR` interface based on parsing input files.

## 4 The Assignment

You should create the following files: `LOCATION.y`, `LOCATION.l`, `MEMREFEXPR.y`, `MEMREFEXPR.l`, `PTRASSIGNPARIS.y`, and `PTRASSIGNPAIRS.l`. Change the Makefile so that these files are properly passed to bison and flex. Their implementation should result in the routines `LOCATION_parse()`, `MEMREFEXPR_parse()`, and `PTRASSIGNPAIRS_parse()`. The input file `Input/testSubIR.oa` should provide enough examples for you to determine how to specify the regular expressions for the tokens and how to specify the context free grammer for parsing. The parse routines should be called from the `top_parser()` routine as appropriate. Note that you will have to uncomment out parts of the `testSubIR.oa` file as you get the appropriate sub parsers to work.

### 4.1 Syntax and Semantics for Input

```
PROCEDURE = { < ProcHandle("main"), SymHandle("main") > }
```

The `PROCEDURE` syntax involves only a single tuple that specifies a `ProcHandle` and a `SymHandle` for the procedure.

```
LOCATION = { < SymHandle("x"), local > }
```

The `LOCATION` syntax involves describing a variable with the location abstraction. The only information needed is whether the symbol is strictly "local" or "not local".

```
MEMREFEXPR = { StmtHandle("p = &x;") =>
  [
    // NamedRef with default values no addressOf and full accuracy
    MemRefHandle("p_1") => NamedRef( DEF, SymHandle("p") )
    // NamedRef with all information specified
    MemRefHandle("&x_1") =>
      NamedRef( USE, SymHandle("x"), addressOf = T, accuracy = full )
  ]
}
```

The notation for `MEMREFEXPRs` is described at <http://www-unix.mcs.anl.gov/OpenAnalysisWiki/moin.cgi/LocationAbstraction> in Section 2.4 (Thanks to Brian White who wrote up some reasonable notation conventions). That same page also has more examples. Note that there is a long form and a more succinct form. You should be able to parse either. Also note that there are default values. For example, with a `NamedRef` the default value for `addressOf` is false and the default value for `accuracy` is full.

```
PTRASSIGNPAIRS = { StmtHandle("p = &x;") =>
  [
    < MemRefHandle("p_1"), MemRefHandle("&x_1") >
  ] }
}
```

The PTRASSIGNPAIRS is the StmtHandle mapped to a list of MemRefHandle pairs indicating the target and the source for pointer assignments, < target, source >.

## 4.2 Extra

- You can change the syntax however you would like as long as the same information is conveyed and you briefly describe why.
- For each bug you find in the ManagerFIAlias implementation, you will receive one extra credit point as long as you are the first one in the class to find it.

## 5 Hints for doing the assignment

In your test .oa files, the strings for each MemRefHandle must be unique. In other words, if you access the variable x in three different locations then you need MemRefHandle("x\_1"), MemRefHandle("x\_2"), and MemRefHandle("x\_3").

You can pass a pointer to anything up the parse tree as long as you put the pointer type in the union.

Information about programming with OpenAnalysis can be found at <http://mcs.anl.gov/OpenAnalysisWiki/moin.cgi>.

**You should start this assignment early;** depending on your familiarity with bison, flex, C, C++ and STL, there may be some surprises in store for you. I will be available during regular office hours and by email. I can look at your code and help point you in the right direction, but the amount of help I can give may be inversely proportional to the amount of time until the due date. By no means should you spend several hours trying to figure out a weird bug; consult me for help. When e-mailing me about the project, send a copy of the relevant section of your code (not as an attachment; send it as text appended to your message). Give a good description of what is going wrong, which should include information about the stack in a debugger.

## 6 Your Report

The report is an essential part of your completed assignment. Use it to describe your solution, assumptions, difficulties, insights, and results. Organize and present your document as if it were the only basis for your assignment's grade. The format of your writeup is up to you, but it should minimally answer the following questions:

- How did you design the context free grammar (CFG) for each of the specialized input languages? What assumptions are you making?
- The mini-languages in this project can actually be expressed with regular expressions; therefore, using bison is overkill. Why can the mini-languages be expressed with regular expressions instead of needing CFGs?

- What feature in C/C++ can the alias analysis not handle yet? Think about how you would express all the features in C/C++ to the alias IR interface.
- What problems did you encounter while developing your program? If you knew someone who was just about to start work on this assignment, what advice would you give them?
- What, if any, outside sources did you use (e.g., articles, books, other students)? This is particularly important. It's OK to look at books and articles and speak with your professor and fellow students (although sharing code and working together is strictly forbidden), but as with any scientific document, you should always cite your references and collaborations. You can either cite collaborations in footnotes or in a separate Acknowledgment section.
- How did you test your program? Does your program work on the examples provided?
- What other examples have you tried your program on? For this assignment, you should give at least two non-trivial examples not in this document that show off your program's functionality.

There's no exact number of pages you should write, but if you've got between two and four then you're probably in the right ballpark.

## 7 What to turn in

You will turn in your work using WebCT. Put your paper in your copy of the `Project1` directory, tar up a clean version of `Project1` (ie. do a `make clean`), and submit the tar ball.

You should provide enough code so that the `driver` executable generates the same output as the `testSubIR` executable when run on `Input/testSubIR.oa`. *Do not* submit your executable or object files; these can be very large and might overflow the disk if many students were to submit them. Your writeup should be in one of the following formats: ASCII text,  $\text{\LaTeX}$ , or PDF.

Also, turn in a hard copy of your report at the **beginning** of class on the due date.

## 8 Due date

This assignment is due Friday September 9th, at **2:00pm**. Late assignments will be penalized 10% per day.