Polyanna: A Sorcerer-based Symbolic Manipulation of Polynomials

In this paper, we describe Polyanna--a simple symbolic calculator which can add a multiply polynomials. In addition to being a useful tool, polyanna serves as an illustration of some of Sorcerer's capabilities.

1.0 Introduction.

In this paper we illustrate the how the powerful tree walking and manipulating features of SORCERER can be used to simply and almost automatically create applications which would require tedious and intricate programming if done by hand. Even top-flight prorammers can find themselves logging many hours with their favorite debuggers when attempting to write the kind of recursive, pattern-matching code which crops up in applications like optimizing compilers and Artificial Intelligence programs.

Using SORCERER has much the same flavor as using a compiler compiler, and programmers who are already proficient with ANTLR will, in a few days time, find themselves tackling with ease projects which they might never have done otherwise.

2.0 Features of Polyanna

Before we dive into the details of interfacing ANTLR and SORCERER, let's get a feel for the kind of tool we would like to create. We'll call it Polyanna, because it simple polynomial manipulator. We'd like it to be able to perform operations like the following:

```
> expand((x^2 + 2*x + 6) * (3*x + 7));
   3      2
3 x  + 13 x  + 32 x + 42
> int(5 * x^2 + 6, x);
     3
5/3  X + 6 x
> diff((x^3 + 6*x^2+17), x);


  2
3 x  + 12 x
```

There are 3 steps which must be done to execute the kind of commands which are illustrated above:

1. Use ANTLR to read in a command, and create an abstract syntax tree for it.

2. Use SORCERER to walk the tree in order to determine the proper actions to do.
3. Use SORCERER to manipulate the resulting polynomials in such a was as to execute the command.

3.0 A Step-by-Step Description of the Creation of Polyanna.

In this section, we describe in a rather exhaustive fashion the steps required to create an application which uses sorcerer. We hope that the detail does not frighten away potential users of Sorcerer--in fact this much detail should make it easier to use. Conversely, we hope we do not bore the more advanced users of PCCTS with excessive detail.

With these caveats in mind, we proceed to describe in detail the creation of polyanna:

3.1 Create directory for the project.

In our case, we use a directory named “poly” to hold all of the files which are assoticated with this project.

3.2 Determine what tokens are necessary.

Looking over the example interaction above, we see that polynomials contain + and *, as well as variable names and numbers. An invaluable source of token definitions comes for free in the example ansi C grammar available from most ftp sites which carry PCCTS. The token definitions for handling floating-point numbers was lifted directly. The other tokens are self-explanatory.

```
#token INT “[1-9][0-9]*”
#token OPENP “\(“
#token CLOSP “\)”
#token VAR “[a-zA-Z][a-zA-Z]*”
#token FLOATONE “([0-9]+.[0-9]* | [0-9]*.[0-9]+) {[eE]{[\-\+]}[0-9]+} {[fFlL]}”
#token FLOATTWO “[0-9]+ [eE]{[\-\+]}[0-9]+ {[fFlL]}”
#token PLUS “\+”
#token MINUS “\-”
#token MULT “\*”
#token EXPN “\^”
```

3.3 Write the AST.h File

polyanna uses the built-in abstract syntax tree creation capabilities of PCCTS. Therefore, we must create an AST.h which contains the things which ANTLR is looking for. Useful templates for building your own AST.h files can be found in the testcpp/ directory of the PCCTS distribution directory. The exact contents of our AST.h reflects the fact that we

want not only ANTLR to automatically create our abstract syntax trees, but also that we wish SORCERER to manipulate them.

We define the class AST as inheriting from the supplied ASTBase class. This automatically gives us the fields which are necessary in order to have PCCTS automatically create the ASTs for us:

```
class AST: public ASTBase {
```

Next we must put in a type() function in for purposes of communicating with SORERER. The type() function returns an in which is used to determine the token which this AST node is associated with:

```
protected:
  int _type;
public
 int type(){
    return _type;
}
```

Now we must write the constructor for the AST class. The purpose of the constructor is to tell PCCTS how to create an AST node from an ANTLRToken. It must look at the token which was scanned, and set the _type field accordingly:

```
 AST(ANTLRTokenBase *t){
    _type=t->getType();
     switch(_type){
    case INT:
          numb=atof(t->getText());
          _type=FLOATONE;
           break;
    case FLOATONE:
           numb=atof(t->getText());
            break;
    case FLOATTWO:
           numb=atof(t->getText());
           _type=FLOATONE;
            break;
    default:
           text = new char[strlen(t->getText())+1];
            strcpy(text, t->getText());
             _type=NOT_AT_TOKEN;
             break;
   }
}
```

And as a final touch to our AST class, we include a preorder_action() function for purposes of printing out our ASTs:

```
void preorder_action() {
    switch(_type){
    case FLOATONE:
 printf("%f ", numb);
 break;
    default:
 printf("%s ", text);
 break;
  }
}
```

3.4 Write the ANTLR Grammar for Recognizing Polynomials

Polynomials have a simple structure--which means that fortunately a simple grammar will suffice to parse them:

```
class poly_parser {
polynomial:
term (MULT^ term)*
;
term:
VAR {EXPN^ constant}
   | constant {PLUS^ term}
;
constant: INT
 | FLOATONE
 | FLOATWO
;
}
```

3.5 Create the Makefile Using genmk

genmk is an extremely useful program which takes care of much of the hassles of using PCCTS. However, it is not yet SORCERER-aware, so we must post-edit its output. However, by using it we automate as much of our job as possible. Here is the command line which I used to create the polyanna Makefile:

genmk -CC -class poly_parser -project polyman poly_grammar.g > Makefile genmk -CC -project polyman -class poly_parser poly_grammar.g -trees >Makefile

The first patch which needs to be made to the output is to tell the Makefile where to find the PCCTS header and library files. You'll have to change the PCCTS macro to point to the PCCTS home directory.

The second change which is necessary for this project is to tell ANTLR that we need a lookahead of two and we want interactive input:

Change: **AFLAGS = -CC -gt** to : **AFLAGS = -CC -gt -k 2 -g**k

3.6 Write the Sorcerer file

Because we want to keep the ANTLR grammar and the SORCERER manipulator in sync, we have to make sure the same tokens are used by both ANTLR and SORCERER. This can be accomplished by including the tokens in the #header directive. We also want to use the same ASTs; therefore we include the AST.h which we wrote above as well:

```
#header
<<
#include <stdio.h>
#include "tokens.h"

/* Interface with the ANTLR ASTs */
#include "AST.h"
typedef AST SORAST;
>>
```

The rest of our SORCERER file will contain the various actions which let us manipulate the polynomials.

3.7 Patch the Makefile to Include Actions to Generate SORCERER Stuff

Because genmk doesn't do this for us automatically, we have to by hand add in the various actions to the Makefile to generate the sorcerer stuff. Remember, the name of the class which you define in the sorcerer file is the name of the.C file which sorcerer spits out. I've called my sorcerer file "poly_treewalk.sor" and it defines a class called poly_manipulator. Therefore, I'll get as output a poly_manipulator.C file which needs to be compiled and linked in with the rest of polyanna. To make this work I need to make the following changes to the Makefile:

1. The SOR macro needs to be set to the directory where SORCERER is installed.

2. The SOR_H macro needs to be set to the directory where the SORCERER include files are found.

3. The CFLAGS macro needs to include the switches -I$(ANTLR_H) -I$(SOR_H)

4. The SOR_SPAWN macro is set to the two files which SORCERER creates: poly_manipulator.C poly_manipulator.h

5. The SRC needs to have the following source files added to its list: poly_grammar.C, poly_parser.C, poly_manipulator.C.

6. The OBJ macro needs to have the object files corresponding to the above added as well.

7. Actions must be added to compile the.C files which are output by SORCERER:

poly_manipulator.C: poly_manipulator.sor $(SOR) -transform -CPP poly_manipulator.sor

poly_manipulator.o: poly_manipulator.C $(CCC) -c $(CFLAGS) $(SORCERER_INCLUDE) poly_manipulator.c

This concludes the steps necessary to integrate a SORCERER manipulator into your PCCTS-based projects. Hopefully most of this will be automated by a future version of genmk (hint hint Ter).

3.8 How Polyanna Manipulates Polynomials.

Because of space limitations, we give only a high-level overview of how Polyanna manipulates polynomials. A fuller description will appear in an extended version of this paper, perhaps incorporated as a chapter in a future edition of the PCCTS/SORCERER reference manual.

3.9 The Workhorse Step: Adding or Multiplying two Monomials.

When manipulating polynomials, the fundamental operations are adding or multiplying two monomials. We consider first monomial addition. If we are adding two monomials which have the same variable raised to the same exponent, then we simply add their coefficients. For example, “3 * X^2” plus “4 * X^2” yields “7 * X^2”. If the monomials have different exponents, then the result is a polynomial with two terms.

Multiplying two monomials is even easier: the result is always a monomial whose coefficient is the product of the coefficients of the input monomials, and whose exponent of its variable is the sum of the input exponents. Using SORCERER, it is trivial to walk the constructed ASTs and output a new AST which represents the results of these operations.

3.10 Arbitrary Polynomial Multiplication/Addition

Addition of two arbitrary polynomials is built up in two steps. The first step is to create SORCERER rules which add a monomial to an arbitrary polynomial. Second, when adding two arbitrary polynomials, the first rule is then recursively called for each of the terms of the summand.

Similarly, arbitrary polynomial multiplication is built up in a similar two-step way. Routines are created which multiply a polynomial by a monomial. To multiply two polynomials, we then call this routine many times and sum up the intermediate steps.

3.11 Polynomial Differentiation

Differentiation a monomial is trivial except for one caveat: it isn't closed over the polynomials, i.e. if the exponent of the variable is -1, then the result is not a polynomial. In this case we report an error, otherwise, we multiply the coefficient by the exponent and then subtract one from the exponent, i.e. "5 * X ^2" yields "10 * X".

For differentiation of a full polynomial, it suffices to simply traverse it differentiating each term.

4.0 Conclusion

The use of SORCERER should be much more widespread than it currently is. A programmer who already knows ANTLR has already mastered 90% of the skills necessary to use SORCERER, and after taking the plunge, will find a world of new possibilities open.