Prop
Language Reference Manual

(for version 2.3.4)

Allen Leung
E-mail: leunga@cs.nyu.edu
http://valis.cs.nyu.edu:8888/~leunga
Courant Institute of Mathematical Sciences
251 Mercer Street
New York, NY 10012

October 2, 2005

Abstract

This reference manual provides an introduction to **Prop**, release 2.3.4. **Prop** is a multiparadigm extension of C++, and is designed for building high performance compiler and language transformation systems, using pattern matching and rewriting. This guide describes the syntax and semantics of the **Prop** language and describes how to develop programs using the **Prop** to C++ translator.

Contents

1 Introduction **4**
- 1.1 Availability . . . 4
- 1.2 Organization of this Reference Manual . . . 4

2 General Concepts **5**
- 2.1 Syntactic Conventions . . . 5
- 2.2 Basic syntactic classes . . . 5
- 2.3 Literals . . . 5
- 2.4 The **Prop** Language . . . 5

3 Lexer and Parser Generation **7**
- 3.1 Lexer Specification . . . 7
  - 3.1.1 Regular expressions . . . 7
  - 3.1.2 Lexeme . . . 8
  - 3.1.3 Lexeme class . . . 8
  - 3.1.4 Tokens . . . 9
  - 3.1.5 The `matchscan` statement . . . 9
  - 3.1.6 Class `LexerBuffer` . . . 11
  - 3.1.7 Class `IOLexerBuffer` . . . 12
  - 3.1.8 Class `IOLexerStack` . . . 12
- 3.2 Parser Specification . . . 13
  - 3.2.1 Syntax class . . . 13
  - 3.2.2 Syntax declaration . . . 13
  - 3.2.3 Precedence . . . 14
  - 3.2.4 `expect:` n . . . 14
  - 3.2.5 Production rules . . . 15
  - 3.2.6 Parser report . . . 16
  - 3.2.7 Interfacing with the generated lexer . . . 16
- 3.3 Debugging Parsers . . . 16

4 Algebraic Datatypes and Pattern Matching **17**
- 4.1 A quick tour of pattern matching . . . 17
  - 4.1.1 Pattern matching versus object-oriented style . . . 18
  - 4.1.2 More examples . . . 19
  - 4.1.3 Variants of match . . . 22
- 4.2 Algebraic Datatypes . . . 23
  - 4.2.1 Instantiating a datatype . . . 25
- 4.3 Pattern Matching . . . 25
- 4.4 Refining a datatype . . . 27
- 4.5 Memory management . . . 29
  - 4.5.1 Garbage collection . . . 29
  - 4.5.2 Persistence . . . 30

5 Inference **32**
5.1 An Example . . . 32
5.1.1 Another example . . . 33
5.2 Inference Class . . . 33
5.3 Inference Rules . . . 34

6 Tree Rewriting **35**
6.1 A rewriting example . . . 35
6.1.1 Conditional rewriting and actions . . . 36
6.2 Rewrite class . . . 37
6.3 Rewriting rules . . . 37
6.4 Rewriting modifiers . . . 38
6.4.1 Rewriting modifier example . . . 38
6.5 The `rewrite` statement . . . 40
6.5.1 Rewriting modes . . . 41
6.5.2 State caching optimization . . . 41
6.5.3 Specifying secondary indices . . . 42
6.5.4 Using an internal index . . . 42
6.5.5 Using an external index . . . 43
6.6 Short circuiting rewrite paths with `cutrewrite` . . . 45
6.7 Conditional failure with `failrewrite` . . . 46
6.8 Confluence and termination . . . 46
6.9 Debugging Tree Rewriting . . . 47
6.10 Optimizing Tree Rewriting . . . 47

7 User defined datatypes: Views **49**
7.1 A first example . . . 49
7.2 Another view example . . . 51
7.3 Syntax of view definitions . . . 52

8 Graph Types and Graph Rewriting **54**
8.1 Graph Types . . . 54
8.2 The Graph Interface . . . 54
8.3 Set Formalisms . . . 54
8.4 Graph Rewriting . . . 55

9 Running the Translator **56**
9.1 Options . . . 56
9.2 Error Messages . . . 57

A Garbage Collection in the Prop Library **63**
A.1 Introduction . . . 63
A.2 The Framework . . . 63
A.3 Our Framework . . . 64

A.4 The Implementation 64
A.5 Architecture 64
A.6 The Programmatic Interface 64
A.7 Memory Allocation 65
A.8 The GC Protocol 66
A.9 Messages and Statistics 67
A.10 The Bartlett style mostly copying collector 68
A.11 The Mark Sweep collector 68
A.12 Finalization 69
A.12.1 Weak Pointers 70
A.12.2 The Heap Walker 70

1 Introduction

This is the language reference manual for **Prop**, a compiler generator. **Prop** can be used to generate lexers, parsers, analysis and transformation tools using pattern matching and rewriting. The **Prop** language is a *multi-paradigm* extension of C++[Str91] and C++ programs generated from **Prop** specifications can be compiled into efficient code.

Version 2.3.4 of **Prop** includes the following major features:

1. Algebraic datatypes and pattern matching, and automatic mapping of algebraic datatypes into C++ class hierarchies.
2. A LALR(1) parser generator in the tradition of *yacc* and *bison*. Each grammar is encapsulated into a parser class and it is possible to have multiple instances of a parser.
3. Regular expression string matching and lexer generation.
4. Transformation of algebraic datatypes using conditional rewriting.
5. In addition, using the view mechanism, it is possible to treat externally defined C or C++ data structures as if they were **Prop** defined datatypes. The full range of pattern matching and rewriting capability can be used on these externally defined data..

In addition, we intend to implement/fix-up the following features in the near future:

1. Persistence.
2. Pretty printing/reading.
3. Visualization of data structures using *vcg*[San94, San95].
4. Automatic mapping of graph structure specifications into C++ classes.
5. Graph rewriting system generation.

1.1 Availability

The latest release of **Prop** 2.3.4 is available from

```
http://valis.cs.nyu.edu:8888/~leunga/prop.html
```

New updates of the source code and documentation will be available from this site.

1.2 Organization of this Reference Manual

This reference manual is organized as follows. Section 2 overviews the general features of **Prop**. Section 3 describes the lexer and parser specification formalisms. Section 4 describes the algebraic datatypes and pattern matching features. Section 6 describes the tree rewriting mechanism. Finally, section 9 describes the process of running the **Prop** translator.

We'll use the following extended BNF notation to specify the syntax of the language. Terminals are typeset in `typewriter` font, while non-terminals are typeset in *itallics* font. Given a syntactic class S, we use S **,** ... **,** S to denote one or more occurrences of S separated by commas. Similarly, S**;** ... **;** S denotes one or more occurrences of S separated by semi-colons. Slanted brackets are used to denote an optional occurrence of a syntactic class. We'll combined the two forms when we want to denote zero or more occurrences of some term. For example, *[* S **,** ... **,** S *]* denotes zero or more occurrences of S.

2 General Concepts

2.1 Syntactic Conventions

The syntax of **Prop** is an extension of C++. Most new constructs of **Prop** starts with new keywords, listed below.

```
applicative   arb           as           bagof        begin      bitfield     card
category      classof       collectable  constraint   dataflow   datatype     declare
dequeof       dom           domain       edges:       elsif      end          equiv:
exists        expect:       feature      finalizable  forall     fun          function
functor       graphrewrite  graphtype    implies:     inference  inherited    inline
instantiate   is            law          left:        less       lexeme       listof
loop          mapof         match        matchall     matchscan  module       multimapof
mutable       nodes:        of           persistent   prec:      printable    priqueueof
procedure     queueof       ran          refine       relation   rewrite      right:
setof         sharing       signature    space:       syntax     synthesized  then
time:         topdown       traced       treeparser   tupleof    unifiable    unique
view          where         with         xor:         bottomup:  topdown:     before:
preorder:     postorder:    cutrewrite   failrewrite  index:
```

In addition, the following new multi-character symbols are added:

```
<->  <=>  :=  :-  .(   .[
#[   #(   #{  [|  |]   (|  |)  {|  |}
```

2.2 Basic syntactic classes

We'll use *Integer*, *Char*, *Real*, *String* and *Id* to denote the syntactic classes of integers, characters, real numbers, strings and identifiers, respectively. As in C++, characters are quoted with the single quotes `'`, while strings are quoted with double quotes `"`. An identifier is any sequence of alphanumeric of characters that begins with a letter. In addition, we'll use the syntactic class *Stmt* to denote any valid combination of C++ and **Prop** statements.

2.3 Literals

In addition to the usual C++ literals, **Prop** introduces two new type of literals: *quark literals* and *regular expression literals*. Quark literals are of type `Quark` and are written as strings prefixed by the `#` symbol.

Quark **::=** `#`*String*

Quarks act like atoms in Lisp, in which name equality implies pointer equality. In contrast, in C++ two string literals that are identical in value do not necessarily reside in the same address. Given strings `s1` and `s2`, `strcmp(s1,s2) == 0` does not imply `s1 == s2`. Quarks are defined in the library include file `<AD/strings/quark.h>`.

Regular expression literals are similar to string literals except that they are quoted by slashes `/`. We'll discuss the regular expression in section 3.1.1.

2.4 The Prop Language

The basic **Prop** language is the same as C++ with the following syntactic extensions:

	Keywords	Functions
1	`datatype refine instantiate`	Algebraic datatype definitions
2	`match matchall`	Pattern matching
3	`rewrite`	Rewriting
4	`lexeme`	Lexical category definition
5	`matchscan`	Lexical scanning
6	`syntax`	Parser specification
7	`graphtype`	Graph data structure specification
8	`graphrewrite`	Graph rewriting

Like C++, a **Prop** is typically divided into the *specification*, which defines the data structure and its interface, and the *implementation* parts. **Prop** specifications should be placed in a file with a `.ph` suffix, while an implementation should be placed in a file with a `.pC`[1]. The translator will convert each `.p`*xxx* into a file with the suffix *.xxx*. The translated output can then be fed to the C++ compiler for compilation.

[1] The suffixes `.pcc`, `.pCpp` etc. can also be used.

3 Lexer and Parser Generation

In this section we'll describe the lexer and parser generation mechanism of **Prop**. Note that the use of these features is not mandatory: the user can substitute any other lexer/parser generation tools or use hand written lexers and parsers. On the other hand, higher level of integration is possible when using **Prop**'s lexer and parser mechanisms.

3.1 Lexer Specification

Lexical analyzers are specified using the `matchscan` statement. This construct is responsible for generating the actual string matching DFA. The actual buffering mechanisms are provided by the classes `LexerBuffer`, `IOLexerBuffer` and `IOLexerStack`. These classes are part of the support library distributed with **Prop**.

We assume that the user is familiar with lexer generator tools like *lex*[Les75] or *flex*[Pax90].

3.1.1 Regular expressions

Figure 1 describes the set of current supported regular expressions in **Prop**. The syntax is similar to what is found in *lex*, or *egrep*.

c	matches character c if it is not a meta character
e_1e_2	matches e_1 then e_2
`.`	matches any character except `\n`
`\`c	matches escape sequence c
`^`e	matches e at the start of the line
e_1`|`e_2	matches e_1 or e_2
e`*`	matches zero or more e
e`+`	matches one or more e
e`?`	matches zero or one e
`(`e`)`	grouping
`<<`$C_1, C_2 \ldots C_n$`>>`e	matches e only if we are in a context from one of $C_1, C_2, \ldots C_n$
`{`*lexeme*`}`	matches lexeme

Figure 1: Regular expressions.

The symbols `\ [ ] ( ) { } << >> * + . - ? |` are meta characters and are interpreted non-literally. The escape character `\` can be prepended to the meta characters if they occur as literals in context.

Precedence-wise, meta characters `*`, `+` and `?` bind tighter than juxtaposition. Thus the regular expression `ab*` means `a(b*)`. Parenthesis can be used to override the default precedence.

Character classes are of the form as found in *lex*: (i) c_1`-`c_2 denotes the range of characters from c_1 to c_2; (ii) single (non-meta) characters denote themselves; (iii) the meta character `^` can be used to negate the set. For example, the regular expression `[a-zA-Z][a-zA-Z0-9]*` specifies an alphanumeric identifier that must starts with a letter. Similarly, the regular expression `[^ \t\n]` matches any character except a space, a tab or a newline.

Lexemes are simply abbreviated names given to a regular expression pattern. They act like macros in *lex*.

While a lexer is scanning, it may be in one of many *contexts*. Contexts can be used to group a set of related lexical rules; such rules are only applicable when the contexts are active. This makes the lexer behave like a set of DFAs, with the ability to switch between DFAs under programmer control.

3.1.2 Lexeme

Lexemes are defined using the `lexeme` declaration in **Prop**. Its syntax is

Lexeme_Decl ::= `lexeme` *Lexeme_Eq* `|` ... `|` *Lexeme_Eq* `;`
Lexeme_Eq ::= *Id* `=` *Regexp*

For example, the following lexeme definition is used in the **Prop** translator to define the lexical structure of common lexical items.

```
lexeme digits      = /[0-9]+/
     | sign        = /[\+\-]/
     | integer     = /{digits}/
     | exponent    = /[eE]{sign}?{digits}/
     | mantissa    = /({digits}\.{digits}?|\.{digits})/
     | real        = /{mantissa}{exponent}?/
     | string      = /"([^\\"\n]|\\.)*"/
     | character   = /'([^\\'\n]|\\.[0-9a-f]*)'/
     | regexp      = /\/([^\/\n*]|\\.)([^\/\n]|\\.)*\//
     ;
```

Note that regular expression literals are written between slashes: */re/*.

3.1.3 Lexeme class

Often we wish to group a set of lexemes together into *lexeme classes* if they logically behave in some uniform manner; for example, if they act uniformly in a lexical rule. By grouping related lexemes together into a class we can refer to them succinctly by their class name.

The syntax of a lexeme class declaration is

Lexeme_Class_Decl ::= `lexeme class` *Lexeme_Class_Eq* `and` ... `and` *Lexeme_Class_Eq* `;`
Lexeme_Class_Eq ::= *Id* `=` *Lexeme_Spec* `|` ... `|` *Lexeme_Spec*
Lexeme_Spec ::= *String*
| *Id Regexp*

The following example, the lexeme classes `MainKeywords`, `Symbols`, and `Literals` are defined.

```
lexeme class MainKeywords =
    "rewrite" | "inference" | "match" | "matchall" | "matchscan"
  | "refine"  | "classof" | "type" | "datatype" | "instantiate"
  | "lexeme"  | "bitfield" | "begin" | "syntax"
  | "dataflow" | "module" | "signature" | "constraint" | "declare"
  | "procedure" | "fun" | "function" | "domain"
  | "graphtype" | "graphrewrite"

and  Symbols =
    ".." | "..." | "<->" | "::" | "&&" | "||" | "++" | "--" | "->"
  | "<<" | ">>" | ">=" | "<=" | "+=" | "-=" | "*=" | "/=" | "%=" | "=="
  | "!=" | "<<=" | ">>=" | "&=" | "|=" | "^=" | "=>" | "<-" | "<=>"
  | ":=" | ":-" | LONG_BAR /\-\-\-\-\-+/

and Literals =
    INT_TOK     /{integer}/
```

```
  | REAL_TOK    /{real}/
  | CHAR_TOK    /{character}/
  | STRING_TOK  /{string}/
  ;
```

3.1.4 Tokens

Lexical tokens are defined using the `datatype` declaration. The syntax is as follows.

Tokens_Decl	::=	`datatype` *Id* `::` `lexeme` `=` *Token_Spec* `|` ... `|` *Token_Spec* `;`	
Token_Spec	::=	`lexeme class` *Id*	Include a lexeme class
	|	*Id [Regexp]*	Single token spec

A token datatype is defined by including one of more lexeme classes, and by defining stand-alone tokens. If a lexeme class is included, all the tokens defined within the lexeme class are included. A C++ `enum` type of the same name as the token datatype is generated. If a token is given an identifier name, then the same name is used as the `enum` literal. On the other hand, if a string is used to denote a token, then it can be referred to by prefixing the string with a dot `.` For example, the token `"=>"` can be referenced as `."=>"` within a program.

As an example, the following token datatype definition is used within the **Prop** translator. Here, the keywords are first partitioned into 6 different lexeme classes. In additional, the tokens `ID_TOK`, `REGEXP_TOK`, etc. are defined.

```
datatype PropToken :: lexeme =
    lexeme class MainKeywords
  | lexeme class Keywords
  | lexeme class SepKeywords
  | lexeme class Symbols
  | lexeme class Special
  | lexeme class Literals
  | ID_TOK       /{patvar}/
  | REGEXP_TOK   /{regexp}/
  | QUARK_TOK    /#{string}/
  | BIGINT_TOK   /#{sign}{integer}/
  | PUNCTUATIONS /[\<\>\,\.\;\&\|\^\!\~\+\-\*\/\%\?\=\:\\]/
  ;
```

3.1.5 The `matchscan` statement

The `matchscan` statement is used to perform tokenization. The user can specify a set of string pattern matching rules within a `matchscan` construct. Given a object of class `LexerBuffer`, the `matchscan` statement looks for the rule that matches the longest prefix from the input stream and executes the action associated with the rule. Ties are broken by the lexical ordering of the rules.

The general syntax is as follows:

Matchscan	::=	*Matchscan_Mode [Context_Spec]* `(` *Exp* `)`	variant 1
		`{ case` *Matchscan_Rule* ... `case` *Matchscan_Rule* `}`	
	|	*Matchscan_Mode [Context_Spec]* `(` *Exp* `)`	variant 2
		`{` *Matchscan_Rule* `|` ... `|` *Matchscan_Rule* `}`	
	|	*Matchscan_Mode [Context_Spec]* `(` *Exp* `) of`	variant 3
		Matchscan_Rule `|` ... `|` *Matchscan_Rule*	
		`end matchscan ;`	
Matchscan_Mode	::=	`matchscan` *[* `while` *]*	case sensitive
	|	`matchscan*` *[* `while` *]*	case insensitive
Context_Spec	::=	`[` *Id*, ... , *Id* `]`	
Matchscan_Rule	::=	*[* `<<` *Context* , ... , *Context* `>>` *]*	
		`lexeme class` *Id* `:` *Matchscan_Action*	
	|	*[* `<<` *Context* , ... , *Context* `>>` *]*	
		Regexp `:` *Matchscan_Action*	
Matchscan_Action	::=	`{` *Code* `}`	
	|	*Code*	for variant 1 only

The two different modes of operation are `matchscan` and `matchscan*`, which respectively match strings case sensitively and insensitively. The modifier `where` may optionally specify that the matching process should repeat until no rules apply, or the end of stream condition is reached.

By default, if no rules apply and if the input stream is non-empty, then an error has occurred. The `matchscan` statement will invoke the method `error()` of the `LexerBuffer` object by default.

For example, the following is part of the **Prop** lexer specification.

```
   datatype LexicalContext = NONE | C | PROP | COMMENT | ...;

int PropParser::get_token()
{
   matchscan[LexicalContext] while (lexbuf)
   {
      ...
   |  <<C>> /[ \t\\\014]/:            { emit(); }
   |  <<C>> /(\/\/.*)?\n/:            { emit(); line++; }
   |  <<C>> /^#.*/:                  { emit(); }
   |  <<PROP>> lexeme class MainKeywords: { return ?lexeme; }
   |  <<PROP>> lexeme class SepKeywords: { return ?lexeme; }
   |  <<PROP>> QUARK_TOK:            { return QUARK_TOK; }
   |  <<PROP>> BIGINT_TOK:           { return BIGINT_TOK; }
   |  <<PROP>> REGEXP_TOK:           { return REGEXP_TOK; }
   |  <<PROP>> PUNCTUATIONS:         { return lexbuf[0]; }
   |  <<PROP>> /[ \t\014]/:          { /* skip */ }
   |  <<PROP>> /(\/\/.*)?\n/:        { line++; }
   |  /\/\*/:                        { emit(); set_context(COMMENT); }
   |  <<COMMENT>> /\*\//:            { emit(); set_context(PROP); }
   |  <<COMMENT>> /\n/:              { emit(); line++; }
   |  <<COMMENT>> /./:               { emit(); }
   |  /./: { error("%Lillegal character %c\n", lexbuf[0]); }
   }
}
```

Here, the lexer is partitioned in multiple lexical contexts: context `C` deals with C++ code while context `PROP` deals with **Prop** extensions. The special context `COMMENT` is used to parse `/* */` delimited comments. Contexts are changed using the `set_context` method defined in class `LexerBuffer`.

The special variable `?lexeme` can be used within a rule that matches a lexeme class. For example, within the rule

```
|  <<PROP>> lexeme class MainKeywords:    { return ?lexeme; }
```

the variable `?lexeme` is bound to the token `."rewrite"` if the string "rewrite" is matched; it is bound to the token `."inference"` if the string "inference" is matched and so on.

3.1.6 Class `LexerBuffer`

We'll next describe the class `LexerBuffer` and its subclasses.

Class `LexerBuffer` is the base class in the lexical buffer hierarchy. It is defined in the library include file `<AD/automata/lexerbuf.h>`. This class is responsible for implementing a string buffer for use during lexical analysis.

As it stands, it can be used directly if the lexer input is directly from a string. Memory management of the buffer is assumed to be handled by the user.

The class `LexerBuffer` has three constructors. The default constructor initializes the string buffer to NULL. The two other constructors initialize the string buffer to a string given by the user. In the case when the length is not supplied, the buffer is assumed to be `'\0'`-terminated. The two `set_buffer` methods can be used to set the current string buffer. Notice that all lexical analysis operations are done in place. The user should not alter the string buffer directly, but should use the interface provided by this class instead.

```
class LexerBuffer {
public:
   LexerBuffer();
   LexerBuffer(char *);
   LexerBuffer(char *, size_t);
   virtual ~LexerBuffer();
   virtual void set_buffer (char *, size_t);
           void set_buffer (char *);
};
```

The following methods are used access the string buffer. Method `capacity` returns the size of the buffer. Method `length` returns the length of the current matched token. Methods `text` can be used to obtain a point to location of the current matched token. The string returned is guaranteed to be `'\0'`-terminated. Methods `operator []` return the ith character of the token. Finally, method `lookahead` returns the character code of the next character to be matched.

```
   int capacity () const;
   int length   () const;
   const char * text () const;
         char * text ();
   char  operator [] (int i) const;
   char& operator [] (int i);
   int lookahead () const;
   void push_back (int n)
```

In addition to the string buffer, the class `LexerBuffer` keeps track of two additional types of information: the current context of the DFA, and whether the next token starts at the beginning of the line, or in our terminology, whether it is *anchored*. These are manipulated with the following methods:

```
   int context    () const;
```

```
   void set_context (int c = 0);
   Bool is_anchored() const;
   void set_anchored(Bool a = true);
```

Finally, the following methods should be redefined by subclasses to alter the behavior of this class. By default, the class `LexerBuffer` calls `fill_buffer()` when it reaches the end of the string; subclasses can use this method to refill the buffer and return the number of characters read. Currently, `fill_buffer` is defined to do nothing and return 0. When it reaches the end of the file (i.e. when `fill_buffer()` fails to refill the buffer and the scanning process finishes), method `end_of_file` is called. Currently, this is a no-op. Finally, the error handling routine `error()` is called with the position of the beginning and the end of the buffer in which the error occurs. By default, this routine prints out the buffer.

```
protected:
   virtual size_t fill_buffer();
   virtual void   end_of_file();
   virtual void   error(const char * start, const char * stop);
```

3.1.7 Class `IOLexerBuffer`

Class `IOLexerBuffer` is a derived class of `LexerBuffer`. It automatically manages an internal buffer and receives input from an `istream`. It is defined in the file `<AD/automata/iolexerbuf.h>`

This class provides the following additional features: Constructor `IOLexerBuffer(istream&)` ties the input to a stream. If the default constructor is used, then it automatically ties the input to the stream `cin`. The method `set_stream` can be used to set the stream to some other input.

```
class IOLexerBuffer : public LexerBuffer {
   size_t    buffer_size;  // size of the buffer
   istream * input;        // input stream
public:
   IOLexerBuffer();
   IOLexerBuffer(istream&);
   virtual ~IOLexerBuffer();
   virtual void set_stream (istream&);
};
```

By default, class `IOLexerBuffer` reads new data from the input stream using a line buffering discipline. This mechanism is suitable for interactive, but for other applications it may be more efficient to use block buffered input. The protected method `read_buffer` controls this reading behavior; it is called whenever the data in the string buffer has been consumed and new input is needed from the stream. It is passed the position of the buffer and its remaining capacity, and it returns the number of new characters that are read. Subclasses can redefine this method.

```
protected:
   virtual size_t read_buffer(char *, size_t);
```

3.1.8 Class `IOLexerStack`

Class `IOLexerStack` is a derived class of `LexerStack`. It provides a mechanism of reading from a stack of `istream`'s. Streams can be pushed and popped from the stack. The next token is obtained from the stream from the top of the stack. The class allows allows easy implementation of constructs such as the `#include` file mechanism of the C preprocessor.

The interface of this class is listed below:

```
class IOLexerStack : public IOLexerBuffer {
public:
   IOLexerStack();
   IOLexerStack(istream&);
   virtual ~IOLexerStack();

   virtual void     push_stream (istream&);
   virtual istream& pop_stream  ();
};
```

3.2 Parser Specification

Parsers are specified as a two phase process:

1. First a *syntax class* is defined. A syntax class declaration is like a normal C++ class declaration and has a similar syntax, except that **Prop** will also generate the interface of the parser. All parsers that **Prop** generates are encapsulated within a class. This makes it easy for the programmer to add additional data for parsing use, and to have multiple instances of a parser.
2. Secondly, the grammar of the language is defined in a *syntax* declaration as a set of productions, in a syntax similar to that of *yacc*.

We'll describe these two phases.

3.2.1 Syntax class

A syntax class definition specifies an object class that encapsulates a parser. Its syntax is the same as the usual C++ class definition, except that the prefix `syntax` is used:

Syntax_Class_Decl ::= `syntax class` *Id* [: *Inherit_List*] { *Class_Body* } ;

This specification automatically generate a class with the following interface (assuming that the parser class in question is called `ParserClass`)

```
public:
   ParserClass();  // constructor
public:
   virtual void parse();
```

The constructor and the method `parse()` will be automatically generated later.

A parser class expects the following method to be defined:

```
int get_token();
```

The function of `get_token` is to return the next token from the lexer stream whenever it is called. It should return `EOF` if the stream is empty.

3.2.2 Syntax declaration

The grammar of a language is specified in the `syntax` declaration. Its syntax is listed as follows:

Syntax_Decl ::= `syntax` *Id* {
[Precedence_Decl ... Precedence_Decl]
[Expect_Decl]
[Production_Rule ... Production_Rule]
}

The name of a `syntax` declaration should match that of a `syntax class`. A `syntax` declaration is divided into three parts: (i) the operator precedence section, (ii) the `expect:` *n* section, and (iii) the grammar section.

3.2.3 Precedence

Operator precedences are specified using precedence declarations. The syntax is:

Precedence_Decl	::=	`left:` *Integer Operator ... Operator* `;`	left associative operators
	|	`right:` *Integer Operator ... Operator* `;`	right associative operators
Operator	::=	*Char*	single character operator
	|	*String*	multi-character operator
	|	*Cons*	constructor

The integer associated with a precedence declaration is the precedence level of an operator. The higher the level of an operator, the *lower* its precedence.

For example, the following set of precedences are used in the **Prop** language parser:

```
left: 23 "as";
left: 22 "::";
left: 21 "||";
left: 20 "equiv:";
left: 19 "xor:";
left: 18 "implies:";
left: 17 "&&" "and";
right: 16 "|=" "&=" "^=" "<<=" ">>=";
right: 15 '=' ":=" "+=" "-=" "*=" "/=" "%=";
left: 14 '|';
left: 13 ':';
left: 12 ';';
left: 11 '^';
left: 10 '&';
left: 9 "==" "!=";
left: 8 '<' '>' ">=" "<=";
left: 7 "<<" ">>";
left: 6 '+' '-' "with" "less";
left: 5 '*' '/' '%';
left: 4 "++" "--";
left: 3 '!' '~' "arb" "card" "dom" "ran";
left: 2 '[' ']' '{' '}' '.' "->" ;
```

3.2.4 `expect:` *n*

The declaration `expect` *n* specifies that there should be *n* shift/reduce errors in the grammar. If the actual grammar deviates from this number, a warning will be generated. The alternative form `expect: _` suppresses all warnings. The syntax is:

Expect_Decl	::=	`expect:` *Integer* `;`	expects *n* shift/reduce errors
	|	`expect: _ ;`	expects many shift/reduce errors

This declaration is optional. If omitted, warnings will be printed if parser conflicts are found during parser generation.

3.2.5 Production rules

The syntax of the production rules is as follows:

Production_Rule	::=	*Id [Type_Exp] :*	
		[One_Alt | ... | One_Alt] ;	
One_Alt	::=	*[Symbol ... Symbol]*	one alternative
Symbol	::=	*Id*	non-terminal
	|	*Cons*	terminal (a datatype constructor)
	|	*String*	terminal (a datatype constructor)
	|	*Char*	terminal (from the ASCII character set)
	|	`?`	the error terminal
	|	`$`	the end of file terminal
	|	{ *Code* }	embedded actions

Each rule specifies a set of productions of the form

$$\begin{array}{lcl} lhs & : & A_{11}A_{12}\ldots A_{1n_1} \\ & | & A_{21}A_{22}\ldots A_{2n_2} \\ & \vdots & \\ & | & A_{m1}A_{m2}\ldots A_{mn_m} \end{array}$$

where lhs is the non-terminal and A_{ij}'s are a mixture of terminals, non-terminals and embedded actions. If synthesized attributes are used, then the type of the s-attribute should be annotated next to the lhs non-terminal.

Terminals can be specified in a few ways: (i) a character literal is a predefined terminal of the same name, (ii) a string literal is matched with a `datatype` or a `lexeme class` constructor of the same name, (iii) an identifier is a terminal if there is a `datatype` constructor of the same name. Any other identifiers are assumed to be non-terminals.

For example, the following are two sets of production rules used in the **Prop** translator itself:

```
ty Ty:  simple_ty               { $$ = $1; }
|       ty '=' exp              { $$ = DEFVALty($1,$3); }
|       ty '*'                  { $$ = mkptrty($1); }
|       ty '&'                  { $$ = mkrefty($1); }
|       ty '[' ']'              { $$ = mkptrty($1); }
|       ty '[' exp ']'          { $$ = mkarrayty($1,$3); }
;
exp Exp:
        app_exp                 { $$ = $1; }
|       exp '+' exp             { $$ = BINOPexp("+",$1,$3); }
|       exp '-' exp             { $$ = BINOPexp("-",$1,$3); }
|       exp '*' exp             { $$ = BINOPexp("*",$1,$3); }
|       exp '/' exp             { $$ = BINOPexp("/",$1,$3); }
|       exp '%' exp             { $$ = BINOPexp("%",$1,$3); }
|       exp '^' exp             { $$ = BINOPexp("^",$1,$3); }
|       exp "+=" exp            { $$ = BINOPexp("+=",$1,$3); }
|       exp "-=" exp            { $$ = BINOPexp("-=",$1,$3); }
|       exp "*=" exp            { $$ = BINOPexp("*=",$1,$3); }
```

```
|       exp "/=" exp            { $$ = BINOPexp("/=",$1,$3); }
|       exp "%=" exp            { $$ = BINOPexp("%=",$1,$3); }
```

etc ...

Here, `ty` is the non-terminal of this production. The type of the synthesized attribute of this rule is `Ty`, and it is written next to the non-terminal. Like *yacc*, the synthesized attribute of a rule can be referenced using the `$` variables: variable `$$` denote the synthesized attribute of the current rule, while `$`n where n is 1, 2, 3, ... denote the synthesized attribute of the nth rhs symbol of the current rule.

3.2.6 Parser report

If the command line option `-r` is used, then in addition to generating the code for the grammar, the translator will also generate a comprehensive report of the grammar and the resulting LALR(1) automaton. The amount of details in this report can be varied using the the verbose option `-v`:

`-v1` Print full information for all LR(0) states.

`-v2` Also print out the lookahead set of a reduction rule. This may be useful for tracking down shift/reduce and reduce/reduce conflicts.

`-v3` Do both `-v1` and `-v2`.

3.2.7 Interfacing with the generated lexer

The easiest way of interfacing with a lexer is to embed an object of class `LexerBuffer`, `IOLexerBuffer`, or `IOLexerStack` within a syntax class[2]. We can then define the tokenization method `get_token` using the `matchscan` statement described in section 3.1.5.

3.3 Debugging Parsers

Currently, only the most rudimentary debugging facilities have been implemented for parser. The user can define the macro `DEBUG_`C, where C is the name of a syntax class before the `syntax` definition. This will activate the parser debugging code. During parsing, the reductions that are taken will be printed to `cerr`.

[2] Alternatively, inheritance may be used.

4 Algebraic Datatypes and Pattern Matching

Prop implements algebraic datatypes and pattern matching in the style of Standard ML[HMM86]. Tree, DAG and even graph structures can be specified as a set of datatype equations in algebraic datatype specifications. **Prop** then proceeds to translate these into concrete C++ classes. This makes it very easy to implement complex data structures.

Algebraic datatypes can be manipulated and transformed using **Prop** pattern matching constructs, which decompose a datatype value into its constituents. Complex patterns involving multiple objects can be specified. These are translated into efficient testing and selection code in C++.

In the following we shall give a brief overview of the pattern matching features of **Prop**. For most users of modern declarative languages many of these features are already familiar constructs.

4.1 A quick tour of pattern matching

Algebraic datatypes are specified using `datatype` definitions, which define the inductive structure of one of more types using a tree-grammar like syntax. When a datatype is declared, the following operations are implicitly defined by the datatype compiler: (1) the constructors for all the variants of a type; (2) the identity test operator `==`, and the assignment operator `=` for this type; and (3) the member functions needed to decompose a datatype value during pattern matching.

We'll select the internals of a compiler for a simplified imperative language as the running example in this section. Suppose that in this language an expression is composed of identifiers, integer constants and the four arithmetic operators. Then the structure of the abstract syntax tree can be specified as follows:

```
datatype Exp = INT (int)
             | ID  (const char *)
             | ADD (Exp, Exp)
             | SUB (Exp, Exp)
             | MUL (Exp, Exp)
             | DIV (Exp, Exp)
             ;
```

Abstract syntax of an expression such as *a * b - 17* can be constructed directly in a prefix syntax, directly mirroring that of the definition. The **Prop** datatype compiler will automatically generate a C++ class hierarchy to represent the variants of type `Exp`. Datatype constructor functions(not to be mistaken with C++'s class constructors) will also be automatically generated using the same names as the variants.

```
Exp formula = ADD(MUL(ID("a"),ID("b")),INT(17));
```

Datatype values can be decomposed using the `match` statement, which can be seen as a generalization of 's `switch` construct. Pattern matching is a combination of conditional branching and value binding. For example, a typical evaluation function for the type `Exp` can be written as in the following example. Notice that each arm of a `case` is in fact a pattern(with optional variables) mirroring the syntax of a datatype. The pattern variables(written with the prefix `?` in the sequel) of a matching arm is *bound* to the value of the matching value, which can be subsequently referenced and modified in the action of an arm.

```
int eval (Exp e, const map<const char *, int>& env)
{  match (e)
   {  case INT ?i:        return ?i;
      case ID  ?id:       return env[?id];
      case ADD (?e1,?e2): return eval(?e1,env) + eval(?e2,env);
      case SUB (?e1,?e2): return eval(?e1,env) - eval(?e2,env);
```

```
      case MUL (?e1,?e2): return eval(?e1,env) * eval(?e2,env);
      case DIV (?e1,?e2): return eval(?e1,env) / eval(?e2,env);
   }
}
```

4.1.1 Pattern matching versus object-oriented style

Although a comparable evaluation function can be written in object oriented style using late binding, as in below, in general pattern matching is much more powerful than late binding in C++, which only allows dispatching based on the type of one receiver.

```
// Class definitions
class Exp {
public:
   virtual int eval(const map<const char *, int>& env) const = 0;
};
class INT : Exp {
   int i;
public:
   int eval(const map<const char *, int>& env);
};
class ID : Exp {
   const char * id
public:
   int eval(const map<const char *, int>& env);
};
...

// Member functions
int INT::eval(const map<const char *, int>& env) const { return i; }
int ID ::eval(const map<const char *, int>& env) const { return id; }
int ADD::eval(const map<const char *, int>& env) const
   { return e1->eval(env) + e2->eval(env); }
int SUB::eval(const map<const char *, int>& env) const
   { return e1->eval(env) - e2->eval(env); }
int MUL::eval(const map<const char *, int>& env) const
   { return e1->eval(env) * e2->eval(env); }
int DIV::eval(const map<const char *, int>& env) const
   { return e1->eval(env) / e2->eval(env); }
```

For example, in the following function we use nested patterns, non-linear patterns (i.e. patterns with multiple occurrences of a pattern variable), and guards to perform algebraic simplification of an expression. Although the patterns are relatively simple in this example, in general arbitrarily complex patterns may be used.

```
Exp simplify (Exp redex)
{  // recursive traversal code omitted ...

   // match while repeats the matching process
   // until no more matches are found.
   match while (redex)
   {  ADD(INT 0,  ?x):     { redex = ?x; }
   |  ADD(INT ?x, INT ?y): { redex = INT(?x+?y); }
```

```
   |  ADD(?x,     INT 0)   { redex = ?x; }
   |  SUB(?x,     INT 0):  { redex = ?x; }
   |  SUB(?x,     ?x):     { redex = INT(0); }
   |  SUB(INT ?x, INT ?y): { redex = INT(?x-?y); }
   |  MUL(INT 0,  ?x):     { redex = INT(0); }
   |  MUL(?x,     INT 0):  { redex = INT(0); }
   |  DIV(?x,     ?x):     { redex = INT(1); }
        // don't divide by zero.
   |  DIV(INT ?x, INT ?y) | ?y != 0: { redex = INT(?x/?y); }
   |  ...
   }
   return redex;
}
```

Pattern matching in **Prop** is not restricted to one datatype at a time. In the following example, we use matching on multiple values to define equality on expressions inductively. For variety, we'll use the `fun` variant of `match`, which defines a function in rule form. Notice that the last case of the match set uses *wild cards* `_` to catch all the other non-equal combinations. Since C++ does not provide multiple dispatching, implementing binary (or n-ary) matching operations on variant datatypes are in general cumbersome and verbose in object-oriented style. In contrast, using an applicative pattern matching style many manipulations and transformations on variant datatypes with tree-like or graph-like structure can be expressed succinctly.

```
fun equal INT ?i,     INT ?j: bool: { return ?i == ?j; }
  | equal ID  ?a,     ID  ?b:       { return strcmp(a,b) == 0; }
  | equal ADD(?a,?b), ADD(?c,?d):   { return equal(?a,?c) && equal(?b,?d); }
  | equal SUB(?a,?b), SUB(?c,?d):   { return equal(?a,?c) && equal(?b,?d); }
  | equal MUL(?a,?b), MUL(?c,?d):   { return equal(?a,?c) && equal(?b,?d); }
  | equal DIV(?a,?b), DIV(?c,?d):   { return equal(?a,?c) && equal(?b,?d); }
  | equal _,          _:            { return false; }
  ;
```

4.1.2 More examples

As another example, we can specify the term structure of *well-formed formulas* in proposition calculus as follows. Notice that the constructors `F` and `T` are nullary.

```
   datatype Wff = F
                | T
                | Var     (const char *)
                | And     (Wff, Wff)
                | Or      (Wff, Wff)
                | Not     (Wff)
                | Implies (Wff, Wff)
                ;
```

Datatypes that are parametrically polymorphic, such as lists and trees, can be defined by parameterizing them with respect to one of more types. For example, both lists and tree below are parametric on one type argument `T`.

```
   datatype List<T> = nil
                    | cons(T, List<T>);
   datatype Tree<T> = empty
```

```
                | leaf(T)
                | node(Tree<T>, T, Tree<T>);

List<int> primes = cons(2,cons(3,cons(5,cons(7,nil))));
List<int> more_primes = cons(11,cons(13,primes));
Tree<char *> names = node(leaf("Church"),"Godel",empty);
```

As a programming convenience, **Prop** has a set of built-in list-like syntactic forms. Unlike languages such as ML, however, these forms are not predefined to be any specific list types. Instead, it is possible for the user to use these forms on any datatypes with a natural binary *cons* operator and a nullary *nil* constructor. For instance, the previous list datatype can be redefined as follows:

```
datatype List<T> = #[] | #[ T ... List<T> ];
List<int> primes = #[ 2, 3, 5, 7 ];
List<int> more_primes = #[ 11, 13 ... primes ];
List<char *> names = #[ "Church", "Godel", "Turing", "Curry" ];

template <class T>
   List<T> append (List<T> a, List<T> b)
   {  match (a)
      {  case #[]:          return b;
         case #[hd ... tl]: return #[hd ... append(tl,b)];
      }
   }
```

The empty list is written as `#[]`, while *cons(a,b)* is written as `#[ a ... b ]`. An expression of the special form `#[a, b, c]`, for instance, is simple syntactic sugar for repeated application of the *cons* operator, i.e.

```
 #[a, b, c] == #[a ... #[ b ... #[ c ... #[] ] ] ].
 #[a, b, c ... d ] == #[a ... #[ b ... #[ c ... d ] ] ].
```

List-like special forms are not limited to datatypes with only two variants. For example, we can define a datatype similar in structure to S-expressions in *Lisp* or *scheme*. Here's how such as datatype may be defined[3]:

```
datatype Sexpr = INT    (int)
               | REAL   (double)
               | STRING (char *)
               | ATOM   (const char *)
               | #()
               | #( Sexpr ... Sexpr )
where type Atom = Sexpr  // synonym for Sexpr
;
```

With this datatype specification in place, we can construct values of type `Sexpr` in a syntax close to that of *Lisp*. For example, we can define lambda expressions corresponding to the combinators I, K and S as follows:

```
Atom LAMBDA = ATOM("LAMBDA");
Atom f      = ATOM("f");
Atom x      = ATOM("x");
Atom y      = ATOM("y");
Atom NIL    = #();
```

[3]for simplicity, we'll use a string representation for atoms instead of a more efficient method.

```
Sexpr I = #(LAMBDA, #(x), x);
Sepxr K = #(LAMBDA, #(x), #(LAMBDA, #(y), x));
Sepxr S = #(LAMBDA, #(f),
             #(LAMBDA, #(x),
                #(LAMBDA, #(y), #(#(f,x), #(g,x)))));
```

Similar to list-like forms, vector-like forms are also available. This addresses one of the flaws of the C++ language, which lacks first class array constructors. Vectors are simply homogeneous arrays whose sizes are fixed and are determined at creation time[4]. Random access within vectors can be done in constant time. Unlike lists, however, the prepending operation is not supported. Vectors literals are delimited with the composite brackets `(| ... |)`, `[| ... |]`, or `{| ... |}`. In the following example the datatype `Exp` uses vectors to represent the cooefficients of the polynomials:

```
datatype Vector<T> = (| T |);
datatype Exp = Polynomial (Var, Vector<int>)
             | Functor (Exp, Vector<Exp>)
             | Atom (Var)
             | ...
where type Var = const char *;
Exp formula = Polynomial("X", (| 1, 2, 3 |));
```

Commonly used patterns can be given synonyms so that they can be readily reused without undue repetition. This can be accomplished by defining pseudo datatype constructors to stand for common patterns using *datatype law* definitions. For example, the following set of laws define some commonly used special forms for a *Lisp*-like language using the previously defined `Sexpr` datatype.

```
datatype law inline Lambda(x,e)  = #(ATOM "LAMBDA", #(x), e)
           | inline Quote(x)     = #(ATOM "QUOTE", x)
           | inline If(a,b,c)    = #(ATOM "IF", a, b, c)
           | inline Nil          = #()
           | inline ProgN(exprs) = #(ATOM "PROGN" ... exprs)
           | SpecialForm  = #(ATOM ("LAMBDA" || "IF" ||
                                    "PROGN" || "QUOTE") ... _)
           | Call(f,args) = ! SpecialForm && #(f ... args)
           ;
```

Note that the pattern `SpecialForm` is meant to match all special forms in our toy language: i.e. *lambdas*, *ifs*, *progn*'s and *quotes*. The pattern disjunction connective `||` is used to link these forms together. Since we'd like the `Call` pattern to match only if the S-expression is not a special form, we use the pattern negation and conjunction operators, `!` and `&&` are used to screen out special forms.

With these definitions in place, an interpreter for our language can be written thus:

```
Sexpr eval (Sexpr e)
{  match (e)
   {  Call(?f,?args):     { /* perform function call */ }
   |  Lambda(?x,?e):      { /* construct closure */ }
   |  If(?e,?then,?else): { /* branching */ }
   |  Quote(?x):          { return ?x; }
   |  ...:                { /* others */ }
   }
}
```

[4]For efficiency, bounds checking is *not* performed.

As an interesting note, the special form pattern can also be rewritten using regular expression string matching, as in the following:

```
datatype law SpecialForm = #(ATOM /LAMBDA|IF|PROGN|QUOTE/ ... _)
```

In addition, since the law constructors `Lambda`, `Quote`, `If`, `Nil` and `ProgN` are defined with the `inline` keyword, these constructors can be used to within expressions as abbreviates for their rhs definitions. For example, writing `Lambda(x,x)` is the same writing `#(ATOM("Lambda"),#(x),x)`.

4.1.3 Variants of match

Aside from the usual plain pattern matching constructs, a few variants of the `match` construct are offered. We'll briefly enumerate a few of these:

- The `matchall` construct is a variant of `match` that executes all matching rules(instead of just the first one) in sequence.
- Each rule of a `match` statement can have associated cost expressions. Instead of selecting the first matching rule to execute as in the default, all matching rules are considered and the rule with the least cost is executed. Ties are broken by choosing the rule that comes first lexically. For example:

```
match (ir)
{  ADD(LOAD(?r0),?r1) \ e1: { ... }
|  ADD(?r0,LOAD(?r0)) \ e2: { ... }
|  ADD(?r0, ?r1)      \ e3: { ... }
|  ...
}
```

- A `match` construct can be modified with the `while` modifier, as in the following example. A match modified thus is repeatedly matched until none of the patterns are applicable. For example, the following routine uses a `match while` statement to traverse to the leftmost leaf.

```
template <class T>
Tree<T> left_most_leaf(Tree<T> tree)
{  match while (tree)
   {  case node(t,_,_):  tree = t;
   }
   return tree;
}
```

- Finally, the `matchscan` variant of match can be used to perform string matching on a stream. For example, a simple lexical scanner can be written as follows.

```
int lexer (LexerBuffer& in)
{  matchscan while (in)
   {  /if/:                    { return IF; }
   |  /else/:                  { return ELSE; }
   |  /while/:                 { return WHILE; }
   |  /for/:                   { return FOR; }
   |  /break/:                 { return BREAK; }
   |  /[0-9]+/:                { return INTEGER; }
```

```
    |  /[a-zA-Z_][a-zA-Z_0-9]*/: { return IDENTIFIER; }
    |  /[ \t]+/:                 { /* skip spaces */ }
    |  ... // other rules
    }
}
```

4.2 Algebraic Datatypes

Algebraic datatypes are defined using the `datatype` construct. Its syntax is as follows.

Datatype_Decl ::= `datatype`
[Datatype_Spec `and` ... `and` *Datatype_Spec]*
[`where type` *Type_Spec* `and` ... `and` *Type_Spec]*
[`law` *Law_Spec* `and` ... `and` *Law_Spec]*
`;`

Each `datatype` declaration specifies a set of (potentially mutually recursive) types, a set of type abbreviations, and a set of pattern matching abbreviations called *laws*.

Datatype_Spec	::=	*Id [* `<` *Id*, ... , *Id* `>` *]*	
		[`:` *Inherit_List]*	
		[`::` *Datatype_Qualifiers]*	
		[`=` *Cons_Specs]*	
Cons_Specs	::=	*Cons_Spec* `|` ... `|` *Cons_Spec*	
		[`public:` *Datatype_Body]*	
Cons_Spec	::=	*Simple_Cons_Spec*	
		[`with` { *Class_Decl* } *]*	
Simple_Cons_Spec	::=	*Id [String]*	unit constructor
	|	*String*	unit constructor
	|	*Id [* `of` *] Type_Exp*	constructor with arguments
	|	`#[ ]`	unit list constructor
	|	`#{ }`	unit list constructor
	|	`#( )`	unit list constructor
	|	`#[` *Type_Exp* ... *Type_Exp* `]`	list constructor
	|	`#{` *Type_Exp* ... *Type_Exp* `}`	list constructor
	|	`#(` *Type_Exp* ... *Type_Exp* `)`	list constructor
	|	`[|` *Type_Exp* `|]`	vector constructor
	|	`(|` *Type_Exp* `|)`	vector constructor
	|	`{|` *Type_Exp* `|}`	vector constructor

Datatypes can be annotated with *qualifiers*. They tell the **Prop** translator to generate additional code for each datatype to provide additional functionality.

Datatype_Qualifiers	::=	*Datatype_Qualifier* ... *Datatype_Qualifier*	
Datatype_Qualifier	::=	`collectable`	garbage collectable
	|	`rewrite`	optimized for tree rewriting
	|	`persistent`	generate persistence interface
	|	`lexeme`	used for parser/lexer tokens
	|	`inline`	use inlined implementation
	|	`extern`	use non-inlined implementation

Type specifications assign abbreviations to commonly used type expressions. They act like `typedef`'s in C++, except that **Prop** can make use of the type information provided by these specifications.

Type_Spec ::= *Id* = *Type_Exp*

Law specifications are used to define abbreviations to datatype patterns[5]. They behave like macros in some sense, except that unlike macros, they are properly type checked by **Prop**. In addition, if the keyword `inline` is used, then **Prop** will treat the lhs as an expression and generate a function of the same name that can be used to construct the datatype term.

Law_Spec ::= *[* `inline` *] Id [Law_Arg]* = *Pat*
Law_Arg ::= *Id*
| (*Id*, ... , *Id*)

Prop recognizes the following set of type expressions.

Type_Exp	::=	*Type_Id*	type name
	|	*Type_Exp* *	pointer type
	|	*Type_Exp* &	reference type
	|	*Type_Qualifier Type_Id*	qualified type
	|	*Datatype_Qualifier Type_Exp*	annotation
	|	(*Type_Exp*)	grouping
	|	(*Type_Exp* , *TypeExp*, ... , *TypeExp*)	tuple type
	|	(*Type_Exp* , *TypeExp*, ... , *TypeExp*)	tuple class
	|	{ *Lab_Type_Exp*, ... , *Lab_Type_Exp* }	record type
	|	*Type_Exp* = *Exp*	default value
Lab_Type_Exp	::=	*Id* : *Type_Exp*	
Type_Qualifier	::=	`class`	a class type
	|	`const`	constant type
	|	`rewrite`	rewritable
	|	`collectable`	garbage collected
	|	`persistent`	persistent object

In general, four different types of datatype constructors can be defined:

nullary constructors These are constructors without an argument. **Prop** maps these into small integers and no heap space will be consumed.

tuple argument constructors These are constructors with one or more unamed arguments.

record argument constructors These are constructors with one or more named arguments written within braces. For example, in the following datatype definition a record argument constructor called `Add` is defined:

```
datatype Exp = Add { left : Exp, right : Exp }
     | ...
```

To construct an `Add` expression, we can use any one of the following forms:

```
Exp e1 = Add(x,y);
Exp e2 = Add'{ left = x, right = y };
Exp e3 = Add'{ right = y, left = x };
```

Note that we are allowed to rearrange the labels in any order if the record expression form is used.

[5]Patterns are discussed in section 4.3.

empty constructors These are constructors written with an empty argument `()`. For example, in the following definition an empty constructor called `NONE` is defined:

```
    datatype Exp : public AST
= NONE ()
| NONE2
        | ...
```

Here, the constructor `NONE` will inherit from the class `AST`, while the nullay constructor `NONE2` will not.

4.2.1 Instantiating a datatype

Each declared datatype should be *instantiated* in some implementation file using the `instantiate datatype` statement. The translator will generate additional code to implement the datatype. The general syntax is:

Instantiate_Decl	::=	`instantiate datatype` *Type_Spec*`,` ... `,` *Type_Spec*
	|	`instantiate extern datatype` *Type_Spec*`,` ... `,` *Type_Spec*
Type_Spec	::=	*Id* [`<` *TypeExp*`,` ... `,` *TypeExp* `>`]

Tips: When a datatype is defined with the `inline` qualifier, **Prop** will try to generate inline methods whenever possible. On the other hand, if the `extern` qualifier is used, **Prop** will try to generate an non-inline implementation, and only one copy are generated at where the `instantiate datatype` statement occurs.

Since most **Prop** datatypes are light-weight objects, `inline` is taken as the default. The user can override this default using `-save_space` option of the translator (see page 57).

Tips: When the option `-fno-implicit-templates` is used under G++, the user should `#define` the symbol before an `instantiate datatype` statement. This will enable the code for explicit template instantiation.

4.3 Pattern Matching

The basic pattern matching construct is the `match` statement. A match statement can be seen as a generalization of C++'s `switch` statement, in which the expression that follows a `case` can be a general complex pattern.

Its basic syntax is as follows:

Match	::=	*Match_Mode* `(` *Exp* `)` `and` ... `and` `(` *Exp* `)`	
		{ `case` *Match_Rule* ... `case` *Match_Rule* }	variant 1
	|	*Match_Mode* `(` *Exp* `)` `and` ... `and` `(` *Exp* `)`	
		{ *Match_Rule* `|` ... `|` *Match_Rule* }	variant 2
	|	*Match_Mode* `(` *Exp* `)` `and` ... `and` `(` *Exp* `)` `of`	variant 3
		Match_Rule `|` ... `|` *Match_Rule*	
		`end match ;`	
Match_Mode	::=	`match` [`while`]	
	|	`matchall` [`while`]	
Match_Rule	::=	*Pat* [*Guard*] [*Cost*] `:` *Match_Action*	
Guard	::=	`if` *Exp*	
	|	`where` *Exp*	
	|	`|` *Exp*	
Cost	::=	`\` *Exp*	cost expression
Match_Action	::=	{ *Code* }	
	|	*Code*	for variant 1 only

The 3 syntactic variants of the match statement are equivalent in meaning. A `match` statement takes a list of match objects and a set of pattern matching rules. The first rule that matches the pattern completely will be executed. Notice that guards can be placed on each rule, and a rule is only applicable if the guards evaluate to true.

Cost minimization In addition, if cost expressions are specified, then the matching process will proceed as follows:

1. Locate all applicable rules. If no such rule exists then exit.
2. Compute the cost expressions of all applicable rules.
3. Choose the action that has the lowest cost to execute. If a tie occurs, resolve by the textual order of the rules.

Match all rules The `matchall` statement is a variant of `match`. In the matchall mode, the actions of *all* applicable rules are executed in the textual order. Caution: since the matching variable binding process occurs *before* the rules are executed, the rhs actions should not alter the matched value or the pattern variables may be bound to the wrong value.

Repetition In addition, both `match` and `matchall` can be annotated with the `while` modifier. It this case the semantics of the matching construct is to repeat until no more rule is applicable. For example, the following code fragment returns the last element of a list

```
datatype List = #[] | #[ int ... List ];

int last (List l)
{  match while (l)
   { case #[]:         assert(0); // an error
     case #[i]:        return i;
     case #[h ... t]:  l = t;
   }
}
```

Pattern syntax The basic form of a pattern is a literal, which that matches only a single integer, real number, string, etc. Complex patterns can be constructed inductively using datatype constructors, tuples, lists, and *logical pattern connectives*.

The syntax of the patterns is as follows:

Pat	::=	*Integer*	matches integer literal
	|	*Real*	matches real literal
	|	*Char*	matches character literal
	|	*Quark*	matches quark literal
	|	*String*	matches string literal
	|	*Regexp*	matches any string that matches *regexp*
	|	*Pat_Var*	a pattern variable; matches anything
	|	_	wild card; matches anything
	|	*Cons*	matches a datatype constructor
	|	*Cons PatArg*	a constructor application pattern
	|	*Id* `as` *Pat*	binds a variable to the sub-pattern
	|	*Pat* `:` *Type_Exp*	typed pattern
	|	*Pat* `||` *Pat*	matches either pattern
	|	*Pat* `&&` *Pat*	matches both patterns
	|	`!` *Pat*	matches anything but the pattern
	|	`(` *Pat* `)`	Grouping
	|	`#[` *Pat*, ... , *Pat* `]`	list pattern; exact length match
	|	`#[` *Pat*, ... , *Pat* `...` `]`	list pattern; non-exact length match
	|	`[|` *Pat*, ... , *Pat* `|]`	vector pattern; exact length match
	|	`[|` *Pat*, ... , *Pat* `...` `|]`	vector pattern; matches to the left
	|	`[|` `...` *Pat*, ... , *Pat* `|]`	vector pattern; matches to the right
PatArg	::=	*Pat*	
	|	`(` *Pat*, ... , *Pat* `)`	
	|	{ *Lab_Pat*, ... , *Lab_Pat* [...] }	
Lab_Pat	::=	*Id* `=` *Pat*	
Pat_Var	::=	*Id*	
	|	`?`*Id*	

4.4 Refining a datatype

A datatype definition can be refined in a few ways:

1. **inheritance** A datatype can be declared to be inherited from one or more classes $C_1, C_2, \ldots, C_n$. The effect is that all variants of a datatype will be inherited from the classes $C_1, C_2, \ldots C_2$.

2. **member functions** Member functions and addition attributes can be attached to the datatype variants using the keyword `with`.

For example, the following datatype `Exp` is inherited from some user defined classes `AST` and `Object`. Furthermore, a virtual member function called print is defined.

```
datatype Exp : public AST, virtual public Object
   = INT int
         with { ostream& print(ostream&); }
   | PLUS  Exp, Exp
         with { ostream& print(ostream&); }
   | MINUS Exp, Exp
         with { ostream& print(ostream&); }
   | MULT  Exp, Exp
         with { ostream& print(ostream&); }
   | DIV   Exp, Exp
         with { ostream& print(ostream&); }
   | VAR { name : const char *, typ : Type }
```

```
        with { ostream& print(ostream&); }
   public:
   {  virtual ostream& print (ostream&) = 0;
      void * operator new (size_t);
   }
;
```

The allocator and printing methods can be defined as follows:

```
void * classof Exp::operator new(size_t) { ... }
ostream& classof INT::print(ostream& s)   { return s << INT; }
ostream& classof PLUS::print(ostream& s)
   { return s << '(' << this->#1 << '+' << this->#2 <<')'; }
ostream& classof MINUS::print(ostream& s)
   { return s << '(' << this->#1 << '-' << this->#2 <<')'; }
ostream& classof MULT::print(ostream& s)
   { return s << '(' << this->#1 << '*' << this->#2 <<')'; }
ostream& classof DIV::print(ostream& s)
   { return s << '(' << this->#1 << '/' << this->#2 <<')'; }
ostream& classof VAR::print(ostream& s)   { return s << name; }
```

The special type form `classof` *con* is used to reference the type of a constructor. Arguments of a constructor uses the following naming convention: (i) if the constructor C takes only one argument, then the name of the argument is C; (ii) if C takes two or more unnamed arguments, then these are named `#1`, `#2`, etc; (iii) finally, labeled arguments are given the same name as the label.

For readability reasons, it is often useful to separate a datatype definition from the member functions and inheritance definitions. The `refine` declaration can be used in this situation. For example, the above example can be restated more succinctly as follows:

```
//
// Datatype definition section
//
datatype Exp = INT int
             | PLUS  Exp, Exp
             | MINUS Exp, Exp
             | MULT  Exp, Exp
             | DIV   Exp, Exp
             | VAR { name : const char *, typ : Type }
;

//
// Refinement section
//
refine Exp : public AST, virtual public Object
       { virtual ostream& print (ostream&) = 0;
         void * operator new (size_t);
       }
and    INT, PLUS, MINUS, MULT, DIV, VAR
       { ostream& print(ostream&);
       }
;
```

The general syntax of the `refine` declaration is as follows:

Refine_Decl ::= `refine` *Refine_Spec* `and` ... `and` *Refine_Spec* `;`
Refine_Spec ::= *Id*, ... , *Id*
[`:` *Inherit_List]*
[`::` *Datatype_Qualifiers]*
[`{` *Datatype_Body* `}` *]*

Here, *Id* refers to either a constructor name or a datatype name.

4.5 Memory management

By default, a datatype constructor calls the `operator new` to allocated memory. There are a few ways to change this behavior: (i) redefine the `operator new` within the datatype using refinement; or (ii) inherit from a class that redefines this method.

If a *placement* `operator new` is defined in datatype, the placement constructor syntax can be used to invoke this operator:

Placement_Cons ::= *Cons* `'(` *Exp*, ... , *Exp* `)` `(` *[Exp*, ... , *Exp]* `)`
| *Cons* `'(` *Exp*, ... , *Exp* `)` `{` *[Id* = *Exp*, ... , *Id* = *Exp]* `}`

For example, in the following the extra parameter `mem` will be passed to placement `new` operator.

```
Exp e1 = INT'(mem)(1);
Exp e2 = VAR'(mem){ name = "foo", typ = t };
```

4.5.1 Garbage collection

The support library contains an implementation[6] of a conservative garbage collector using two algorithms: one using a mostly copying scheme[Bar88, Bar89] and one using mark-sweep.

A datatype can be defined to be garbage collectable with the qualifier `collectable`. In the following example, the datatype `Exp` is will be allocated from the a garbage collected heap instead of the default heap. Since it references another user defined garbage collectable object `SymbolTable`, the pointer to that symbol table is also marked as `collectable`.

```
// SymbolTable is collectable
class SymbolTable: public GCObject
{
public:
   class Id : public GCObject { ... };
   ...
};

datatype Exp :: collectable
   = INT int
   | VAR (collectable SymbolTable::Id, collectable SymbolTable *)
   | ADD Exp, Exp
   | SUB Exp, Exp
   | MUL Exp, Exp
   | DIV Exp, Exp
   ;
```

[6] The implementation has only been tested on Linux, SunOS and Solaris. Please contact the author for details if you'd like to port the collector to your particular platform.

The corresponding `instantiate datatype` statement for `Exp` will generate the appropriate tracing methods for garbage collection.

Please see appendix A for more details concerning garbage collection.

4.5.2 Persistence

Datatypes can be interfaced with the persistent object library supplied with the translator using the `persistent` qualifier. A persistent object defines the protocol for serializing its representation into an architecture independent format on a persistence stream.

For example, the following datatype declaration defines a persistent type `Exp`. Note that a pointer to the symbol table is annotated with the `persisent` qualifier to signify that the object associated with the pointer should be traversed during the serialization phase.

```
// SymbolTable is persistent
class SymbolTable: public PObject
{
public:
   class Id : public PObject { ... };
   ...
};

datatype Exp :: persistent
   = INT int
   | VAR (persistent SymbolTable::Id, persistent SymbolTable *)
   | ADD Exp, Exp
   | SUB Exp, Exp
   | MUL Exp, Exp
   | DIV Exp, Exp
   ;
```

A persistent object id *must* be provided with each distinct datatype. Object ids are used to identify the classes of persistent objects. Thus they must be unique within an application. Each persistent type tag is simply a string, and can be defined using the `refine persistent` statement.

For example, the following statement declares the persistent object id for type `Exp` above to be `"Simple expressions"`.

```
refine persistent Exp => "Simple expressions";
```

The corresponding `instantiate datatype` statement will generate the appropriate methods to communicate with a persistent stream.

```
instantiate datatype Exp;
```

To write an object e to the datafile `"foo"`, we can say:

```
#include <AD/persist/postream.h>
ofstream out("foo");
Postream pout(out);
pout << e;
out.close();
```

To read the object back from the file, we can say:

```
#include <AD/persist/pistream.h>
```

```
EXP e;
ifstream in("foo");
Pistream pin(in);
e = (Exp)read_object(pin);
in.close();
```

For more details concerning persistent streams and persist objects, please see directory `<AD/persist>` for details.

5 Inference

Semantic processing in compilers and other language processors, such as data flow analysis, can frequently be specified as in a rule-based, logical deductive style. In **Prop**, deductive inference using forward chaining[7] is provided as a built-in mechanism, orthogonal to pattern matching and rewriting, for manipulating user-defined algebraic datatypes.

Similar to syntax classes, **inference classes** may be used for data encapsulation. An inference class is a combination of a C++ class, a database of inference relations, and a collection of inference rules of the form *lhs* `->` *rhs*. The lhs of an inference rule is a set of patterns in conjunctive form. During the inference process, a rule is fired when its lhs condition is satisfied. A fired rule then executes the corresponding rhs action, which may assert or retract additional relations from the database. Using multiple inheritance, it is possible to combine a rewriting class with an inference class such that the rewriting process generates new relations to drive the inference process, or vice versa.

5.1 An Example

Datatype relations are not a distinct kind of data structure but are in fact simply algebraic datatypes declared to have a `relation` attribute. For example, in the following definition three relation types `Person`, `Parent` and `Gen` are defined.

```
datatype Person :: relation = person (const char *)
    and  Parent :: relation = parent (const char *, const char *)
    and  Gen    :: relation = same_generation (const char *, const char *);

instantiate datatype Person, Parent, Gen;
```

Using these relations we can define an inference class that computes whether two persons are in the same generation. Nine axioms are defined (i.e. those whose lhs are empty) in the following. The two inference rules state that (1) a person is in the same generation as him/herself, and (2) two persons are in the same generation if their parents are in the same generation.

```
inference class SameGeneration {};

inference SameGeneration
{  -> person("p1") and person("p2") and
      person("p3") and person("p4") and
      person("p5");

   -> parent("p1", "p2") and
      parent("p1", "p3") and
      parent("p2", "p4") and
      parent("p3", "P5");

   person ?p -> same_generation (?p, ?p);

   parent (?x, ?y) and parent (?z, ?w) and same_generation (?x, ?z)
   -> same_generation(?y, ?w);
};
```

In general, datatypes qualified as `relation`s will inherit from the base class `Fact`, while a rewrite class definition implicitly defines two member functions used to assert and retract facts in the internal database.

[7] The current implementation of Prop translates inference into very naive code. This feature will be phased out and replaced by the more general and much faster graph rewriting mechanism.

For example, in the above example, the following protocol will be automatically generated by the inference compiler.

```
class SameGeneration : ...
{
public:
    virtual Rete&    infer       ();       // start the inference process
    virtual ReteNet& operator += (Fact *); // assert fact
    virtual ReteNet& operator -= (Fact *); // retract fact
};
```

Using these methods, an application can insert or remove relations from an inference class. This will in turn trigger any attached inference rules of the class.

5.1.1 Another example

Consider the following example, which is used to compute Pythagorean triangles. Only one axiom and two rules are used. The axiom and the first rule are used to assert the relations `num(1)` to `num(n)` into the database, where `n` is limited by the term `limit(n)`. The second inference rule is responsible for printing out only the appropriate combinations of numbers.

```
datatype Number :: relation = num int | limit int;

inference class Triangle {};

inference Triangle
{  ->  num 1;

       num m
   and limit n | n > m
   ->  num (m+1);

       num a
   and num b
   and num c | a < b && b < c && a*a + b*b == c*c
   ->  { cout << a << " * " << a << " + "
              << b << " * " << b << " = "
              << c << " * " << c << "\n";
       };
};
```

Now, to print all the triangle identities lying in range of 1 to 100, we only have to create an instance of the inference class, insert the limit, and start the inference process, as in below:

```
Triangle triangle;
triangle += limit(100);
triangle.infer();
```

5.2 Inference Class

This section is incomplete.

5.3 Inference Rules

This section is incomplete.

6 Tree Rewriting

Prop's tree rewriting mechanism let us transform a tree in algebraic datatype form into another tree according to a set of *rewriting rules*. Unlike plain pattern matching described in section 4.3, which only apply to the root of a tree, rewriting rules are applicable to all parts of the tree. This allows the user to concentrate on developing the set of transformations; the actual traversal of the tree object is handled implicitly by **Prop**. When used properly, this mechanism has an advantage over plain pattern matching since rewriting rules remain valid even when a new variant to a datatype is introduced.

In **Prop**, a rewriting system is developed in a similar manner as a parser. The first phase requires the user to defines a *rewrite class* to encapsulate the rewriting rules.

Frequently, the rewriting mechanism is used to collect information about a tree structure; furthermore, more transformation rules are *conditional*: i.e. we want them to be applicable only if certain conditions are satisfied. We can accomplish both of these tasks by defining data members and methods in the rewriting class. These are accessible during the rewriting process. Information collected during rewriting can be stored within the data members.

In the rewriting formalism, equational rules of the form *lhs* `->` *rhs* are specified by the user. During processing, each instance of the lhs in a complex tree is replaced by an instance of the rhs, until no such replacement is possible. Equational rules can often be used to specify semantics based simplification (e.g. constant folding and simplification based on simple algebraic identities) or transformation(e.g. code selection in a compiler backend[AGT89]).

Unlike plain pattern matching, however, the structural traversal process in rewriting is implicitly inferred from the type structure of an algebraic datatype, as specified in its definition.

There are two main forms of rewriting modes available:

- The first is **normalization** mode: a given tree is reduced using the matching rules until no more redexes are available. There are two modes of operations available:
  - in **replacement** mode, the redex of a tree will be physically overwritten.
  - in **applicative** mode, on the other hand, a new tree corresponding to the replacement value will be constructed.

  Replacement mode is used as the default since it is usually the more efficient of the two.
- The second form is **reduction** and **transformation**. In this mode a tree parse of the input term is computed. If cost functions are attached to the rules, then they will also be used to determine a minimal cost reduction sequence. During this process attached actions of a rule may be invoked to synthesize new data.

Each independent set of rewriting rules in **Prop** is encapsulated in its own **rewrite class**. A rewrite class is basically a normal C++ class with a set of rewriting rules attached. During rewriting, the data members and the member functions are visible according to the normal C++ scoping rules. This makes it is easy to encapsulate additional data computed as a side effect during the rewriting process.

6.1 A rewriting example

Consider an abbreviated simplifier for the well-formed formula datatype `Wff` defined in the section 4.1.2. The rewrite class for this can be defined as follows. Since there is no encapsulated data in this example, only the default constructor for the class needs to be defined. A rewrite class definition requires the traversal list to be specified. This is simply a list of datatypes involved in the rewriting traversal process. In this instance only `Wff` is needed.

```
rewrite class Simplify (Wff)
```

```
{
public:
   Simplify() {}
};
```

The rewrite rules for the simplifier can then be specified succinctly as follows. Like the `match` statement, in general the rhs of a rewrite rule can be any statement. A special statement `rewrite(e)` can be used to rewrite the current redex into another form. If the rhs is of the form `rewrite(e)`, then it can be abbreviated to `e`, as in below:

```
rewrite Simplify
{  And(F,  _):      F
|  And(_,  F):      F
|  And(T,  ?X):     ?X
|  And(?X, T):      ?X
|  Or (T,  _):      T
|  Or (_,  T):      T
|  Or (F,  ?X):     ?X
|  Or (?X, F):      ?X
|  Not(Not(?X)):    ?X
|  Not(And(?X,?Y)): Or(Not(?X), Not(?Y))
|  Not(Or(?X,?Y)):  And(Not(?X), Not(?Y))
|  Implies(?X,?Y):  Or(Not(?X), ?Y)
|  And (?X, ?X):     ?X
|  Or (?X, ?X):      ?X
// etc ...
};
```

The rewrite class definition creates a new class of the same name. This new class defines an implicit `operator ()` with the protocol below. This member function can be invoked to perform the rewriting in a functional syntax.

```
class Simplify : ... {
{  ...
public:
   void operator () (Wff);
   // Wff operator () (Wff); // if rewrite class is applicative
};

Wff wff = ...;
Simplify simplify; // create a new instance of the rewrite class
simplify(wff);    // rewrite the term wff
```

6.1.1 Conditional rewriting and actions

Rewriting rules may be guarded with predicates to limit their applicability. In addition, the *rhs* of a rewrite rule is not limited to only a replacement expression: in general, any arbitrarily complex sequence of code may be used. For example, in the following set of rewriting rules we use guards to prevent undesirable replacements to be made during expression constant folding:

```
rewrite ConstantFolding
{  ADD (INT a, INT b):  INT(a+b)
|  SUB (INT a, INT b):  INT(a-b)
```

```
|  MUL (INT a, INT b):
   {  int c = a * b;                      // silent overflow
      if (a == 0 || b == 0 || c / b == a) // no overflow?
      {  rewrite(INT(c)); }
      else
      {  cerr << "Overflow in multiply\n"; }
   }
|  DIV (INT a, INT b) | b == 0:  { cerr << "Division by zero\n"; }
|  DIV (INT a, INT b):  INT(a/b)
|  // etc...
};
```

6.2 Rewrite class

The syntax of a rewrite class declaration is as follows:

Rewrite_Class_Decl ::= `rewrite class` *Id* `(` *TypeExp*, ... , *TypeExp* `)`
[`:` *Inherit_List] [* `::` *Rewrite_Mode* ... *Rewrite_Mode]*
{ *Class_Body* } ;

Rewrite_Mode ::= `treeparser`
| `applicative`
| `topdown`

This is simply the usual C++ class declaration syntax extended to the following information:

a traversal list enclosed in parenthesis. The traversal list of a rewrite class defines the set of types that rewriting should traverse. If a datatype object contains an argument of a type not listed in the traversal list, its value will not be altered.

rewrite mode this defines the rewriting mode of the rewrite class.

6.3 Rewriting rules

Rewriting rules are specified in a rewrite declaration. The syntax of a rewriting specification is as follows:

Rewrite_Decl	::=	`rewrite` *Id*	variant 1
		{ *[Index_Decl]* `case` *Rewrite_Rules* ... `case` *Rewrite_Rules* }	
	|	`rewrite` *Id*	variant 2
		{ *[Index_Decl] Rewrite_Rules* `|` ... `|` *Rewrite_Rules* }	
Rewrite_Rule	::=	*[Rewrite_Modifier] [Id* `->` *]*	
		Pat [Guard] [Cost] : *Rewrite_Action*	
Rewrite_Modifier	::=	`bottomup:`	bottom up mode
	|	`topdown:`	top down mode
	|	`before:`	before actions
	|	`preorder:`	preorder actions
	|	`postorder:`	postorder actions
Rewrite_Action	::=	{ *Code* }	
	|	*Exp*	

The name of a rewrite declaration should match the name of a rewrite class. The two syntactic forms of `rewrite` have equivalent semantics.

The special statement `rewrite`(e) may be used inside the rhs action to rewrite the current redex into e. Note that `rewrite`(e) is a branching statement; statements after `rewrite` are not reachable.

6.4 Rewriting modifiers

In version 2.3.3 onward, rewrite rules can be modified by the keywords: `bottomup:`, `topdown:`, `before:`, `preorder:` and `postorder:`. These modifiers alter the order in which a set of rewriting rules is applied.

These modifiers act like delimiters, similar to the way scoping keywords like `public:`, `protected:` and `private:` are used to delimit declarations in C++. For instance, a set of rules prefixed by the modifier `topdown:` will utilize the topdown strategy for reduction. If no modifier are specified, then bottom-up will be the default.

Note that all five modes of rules can be used together in a rewriting system under tree rewriting mode (see 6.5.1) and their executions are intermixed together.

The semantics of these modifiers are as follows:

`bottomup:` This specifies the modified rewriting rules should be applied in the default bottom-up mode. In this mode, the innermost/leftmost redex is chosen as the next redex. So reduction occurs in a bottomup manner. This means that if a term t is a redex and if the term t contains a subterm t' which is also a redex, t' will be reduced before t.

`topdown:` This specifies the modified rewriting rules should be applied in a topdown mode. In this mode, the rewriter will first try to locate the outermost/leftmost redex. Reduction will occur in a (mostly) topdown manner.

`before:` This specifies that the modified rewriting rules should be tried before the topdown phase. In addition, unlike topdown mode, if state-caching if used (see 6.5.2), the rules are only tried once. Otherwise, these act just like the preorder modifier, described below.

`preorder:` This specifies that the modified rewriting rules should be tried only at the preorder traversal of a term.

`postorder:` This specifies the rewriting rules should be tried only at the postorder traversal of a term.

6.4.1 Rewriting modifier example

We'll use the following example, extracted from a query optimizer for a subset of the relational calculus, to demonstrate the use of the rewriting modifiers.

The query optimizer transforms the following abstract syntax, defined as a set of **Prop**'s datatypes.

```
datatype List<T> = #[] | #[ T ... List<T> ];
datatype Literal = INT int
                 | STRING const char *
                 | BOOL Bool

and      Exp     = OP Id, Exps
                 | APP Id, Exps
                 | LIT Literal
                 | ID Id
                 | TUPLE Exps                  // [ E1, E2, ... En ]
                 | FORALL Id, Exp, Exp         // forall x in E1 . E2
                 | EXISTS Id, Exp, Exp         // exists x in E1 . E2
                 | GUARD (Exp, Exp)            //
                 | GENERATOR (Ids, Exps, Exp)  // [x_1,...,x_n] in X_1*...*X_n
                 | LET  (Id, Exp, Exp)

law inline Int  i    = LIT(INT i)
```

```
|   inline Boolean b = LIT(BOOL b)
|          True      = Boolean true
|          False     = Boolean false
|   inline And a,b   = OP("and",#[a,b])
|   inline Or  a,b   = OP("or", #[a,b])
|   inline Not a     = OP("not",#[a])
|   inline Eq a,b    = OP("=",  #[a,b])
|   inline Ne a,b    = OP("/=", #[a,b])
|   inline Gt a,b    = OP(">",  #[a,b])
|   inline Ge a,b    = OP(">=", #[a,b])
|   inline Lt a,b    = OP("<",  #[a,b])
|   inline Le a,b    = OP("<=", #[a,b])
|   inline In a,b    = OP("in", #[a,b])
|   inline Union a,b = OP("union", #[a,b])
|   inline Count a   = OP("#", #[a])

where type Id         = const char *
and        Ids        = List<Id>
and        Exps       = List<Exp>
;
```

Note that existential and universal quantifiers are represented by the constructors `FORALL` and `EXISTS` respectively. For example, $\exists x \in A.p$ is presented as the term `EXISTS`(x, A, p). Here x is a *binding* occurrence for x; note that the variable x may occur within the predicate p. In addition, a list comprehension expression such as

$$\{e : [x_1, \ldots, x_n] \in X_1 \times \ldots \times X_n \mid p\}$$

is represented as the compound term

$$\texttt{GENERATOR}(\#[x_1, \ldots, x_n], \#[X_1, \ldots, X_n], \texttt{GUARD}(p, e))$$

Suppose we'll like to rename all variables in a query Q so that no two binding occurrences are given the same name. This can be easily accomplished using a combination of `preorder:` and `postorder:` rules in the following manner:

(i) Use preorder rules to check for new variable bindings, for example in `EXISTS`(x, A, p). For each new binding occurrence for x, create a new name for x. These will be performed before the subterms A and p are traversed.

(ii) Whenever a variable x is found during the sub-traversal of A and p, rename the variable by looking up its new name.

(iii) Finally, use postorder rules to remove the current variable substitution whenever we're exiting a binding occurrence.

For example, the following set of rewriting rules accomplish this renaming task for our query language using exactly this method.

```
rewrite RemoveDuplicateNames
{
   // We use before and after rules to remove duplicates from variable names.
   // As each new binding occurrence is found, we enter a new binding
   // into the current environment.  Identifiers found within
   // the binding occurrence are then renamed.
```

```
preorder: // insert new bindings
   EXISTS(x,_,_):     { new_binding(x); }
|  FORALL(x,_,_):     { new_binding(x); }
|  GENERATOR(xs,_,_): { new_binding(xs); }
|  LET(x,_,_):        { new_binding(x); }

         // rename variables
|  ID x:             { rename(x); }

postorder: // removes the binding
|  EXISTS(x,A,_):      { old_binding(x); }
|  FORALL(x,A,_):      { old_binding(x); }
|  GENERATOR(xs,As,_): { old_binding(xs); }
|  LET(x,_,_):         { old_binding(x); }
};
```

Note that we have accomplished a depth first traversal using rewriting without writing any traversal code! As a side benefit, since the traversal is automatically determined by the structure of the datatypes, we do not have to rewrite this renaming code even if the abstract syntax of the query language is extended, as long as no additional binding operators are added.

6.5 The `rewrite` statement

While the rewrite class construct provides a very general abstraction for rewriting, in general its full power is unneeded. It is often convenient to be able to perform rewriting on a term without having to make a new name for a class just for the occasion, especially if member functions and member data are unneeded. To accommodate these situations, the `rewrite` statement is provided to perform a set rewriting transformations on a term without having to define a temporary rewrite class. It is simply syntactic sugar for the more general rewrite class and rewrite rules specifications. For example, a simplify routine for type `Exp` defined above can be specified as follows:

```
Exp simplify (Exp e)
{  // transformations on e before
   rewrite (e) type (Exp)
   {  ADD (INT a, INT b):  INT(a+b)
   |  SUB (INT a, INT b):  INT(a-b)
   |  MUL (INT a, INT b):  INT(a*b)
   |  ...
   }
   // transformations on e after
   return e;
}
```

The `rewrite` normally performs the replacement in-place. An applicative version of the same can be written as follows[8]:

```
Exp simplify (Exp e)
{  rewrite (e) => e type (Exp)
   {  ADD (INT a, INT b):  INT(a+b)
   |  SUB (INT a, INT b):  INT(a+b)
   |  MUL (INT a, INT b):  INT(a*b)
```

[8]The variable `e` is assigned the new copy.

```
   |  ...
   }
   return e;
}
```

The syntax of the rewrite statement is as follows. The traversal list of the set of rewrite rule is listed next to the keyword `type`.

Rewrite_Stmt ::= `rewrite (` *Exp* `) [ =>` *Exp* `]`
`type (` *TypeExp*`, ... ,` *TypeExp* `)`
`{` *[Index_Decl] Rewrite_Rules* `}`
| `rewrite (` *Exp* `) [ =>` *Exp* `]`
`type (` *TypeExp*`, ... ,` *TypeExp* `) of`
[Index_Decl] Rewrite_Rules
`end rewrite ;`

6.5.1 Rewriting modes

There are two basic modes of operations in a rewrite class: (i) *tree rewriting* and (ii) *tree parsing*. Tree rewriting repeatedly looks for redexes in a tree and transforms it into another tree. There are two sub-modes of operations: (a) *applicative* and (b) *in-place*. Applicative mode is specified using the `applicative` mode specifier when declaring a rewrite class. In this mode, a new tree is built from a bottom-up manner, and the subject tree is left unaltered. The default mode is "in-place." In this mode, the subject tree is overwritten with the transformed expression.

On the other hand, in tree parsing mode, the left hand side of a rewrite rules specification is treated as a tree grammar. The tree parser will look for a derivation of given tree object from the start non-terminal. A guarded rule will only be used if the guard evaluates to true.

If rules are annotated with cost expressions, then the tree parser will try to locate a derivation chain with a minimal total cost. After a derivation tree is found, the tree can be repeated traversed. The rhs actions will be invoked during this traversal process.

6.5.2 State caching optimization

The pattern matching process proceeds in a bottom up manner: for each subtree of the form $l(t_1, t_2, \ldots, t_n)$, $n \geq 0$, a state number is computed from the current tree label l and the state numbers of its subtrees $t_1, t_2, \ldots, t_n$. The set of matched rules can be directly computed from the current state number. This means that in the absence of redex replacements, pattern matching takes time proportional only $O(|t| + r)$, where $|t|$ is the size of the input tree t and r is time it takes to execute the rhs actions. That is, the runtime is insensitive to the number of rewrite rules or the complexity of the patterns.

However, if the input is a DAG rather than a tree[9], or if replacements are performed, then the above analysis may not hold. Replacements during rewriting often require state information of the replacement term to be recomputed to further the matching process. Since computation of state can involve a complete traversal of a term, replacement can become expensive if the replacement term is large. For instance, consider the following replacement rule, which replaces all expressions of the form *2*x* into *x+x*:

```
rewrite class StrengthReduction
{
   MUL (INT 2, ?x):  ADD(?x, ?x)
   ...
}
```

[9]Rewriting of a generic graph structure is not recommended with `rewrite`.

Since the subterm `?x` could be arbitrarily large, recomputing the state encoding for `ADD(?x,?x)` naively takes time in proportion to the size of `?x`. However, since the subtree `?x` occurs as part of the pattern, its state has already been computed, and it would be wasteful to recompute its state from scratch.

A similar observation can be made to the case when the input is a DAG, where many subcomponents are shared between nodes. In order to speed up the rewriting process, the state of each shared copy of the nodes should be computed only once.

In order to speedup the state computation process in these situations, can be enabled by specifying a *rewriting index*. This can be accomplished in two ways: (i) use the qualifier `rewrite` when defining a datatype. This specifies the *primary index*. Or alternatively, (ii) use index declarations to specify one or more *secondary indices*.

We'll first describe the first method. A `rewrite` qualifier can be used in the definition of a datatype. For instance:

```
datatype Exp :: rewrite
   = INT (int)
   | ID  (const char *)
   | ADD (Exp, Exp)
   | SUB (Exp, Exp)
   | MUL (Exp, Exp)
   | DIV (Exp, Exp)
   ;
```

The `rewrite` qualifier tells the translator to generate an extra state field in the implementation of `Exp`. This state field is implicitly updated during the rewriting process. When the rewriter encounters a node with a cached state it can quickly short-circuit all unnecessary state computation.

A word of caution: since each rewriting system computes its own encoding, rewriting systems should not be mixed if they shard the same index, i.e. actions of a rewriting system should not invoke another rewriting system on a term that is participating in the current rewriting system, if both systems use the same index. This limitation can be overcome by using secondary indices, which we'll discuss next.

6.5.3 Specifying secondary indices

From release 2.3.0 onward, it is possible to specify secondary indices within a rewriting system. This makes it possible to invoke other rewriting systems within the execution of another, while retaining the state caching optimizations throughout.

There are two forms of secondary index: *internal index* and *external index*. Internal indices are stored directly within a rewrite term and require the user to pre-allocate space for each term. In contrast, external indices are stored in a separate data structure such as a hash table.

The syntax of secondary index declarations is as follows:

Index_Decl	::=	`index:` *Index_Spec*, ... , *Index_Spec* ;	
Index_Spec	::=	*Type_Exp* `= none`	Disable indexing
	|	*Type_Exp* = *Id*	Internal index
	|	*Type_Exp* `= extern` *Id*	External index

6.5.4 Using an internal index

In order to make use of an index, certain member functions have to defined by the user. Given an internal index named *foo*, the rewriter invokes two functions named

```
int get_foo_state() const;
void set_foo_state(int);
```

within the generated code. The user should provide the implementations for these functions within the datatypes.

For example, reusing the well-formed formulas example (see section 4.1.2), an internal index on the datatype `Wff` can be implemented as follows:

```
#include <AD/rewrite/burs.h>
class WffIndex {
   int state;
public:
   WffIndex() : state(BURS::undefined_state) {}
   int get_wff_state() const { return state; }
   void set_wff_state(int s) { state = s; }
};

datatype Wff : public WffIndex
   = F
   | T
   | Var     (const char *)
   | And     (Wff, Wff)
   | Or      (Wff, Wff)
   | Not     (Wff)
   | Implies (Wff, Wff)
   ;

...

rewrite Simplify {
   index: Wff = wff;
   ...
};
```

Here, the class `WffIndex` provides the internal index interface functions `get_wff_state` and `set_wff_state` expected by the rewriter. Note that each term must be initialized with the state `BURS::undefined_state`. This constant is defined within the library include file `<AD/rewrite/burs.h>`.

To enable the index, in the rewriter we specify that datatype `Wff` should make use of the index named `wff`, using the `index:` declaration.

6.5.5 Using an external index

External indices are specified in a similar manner. Given a datatype *Foo* and an external index named *bar*, the rewriter invokes calls to the following functions:

```
int get_bar_state(Foo) const;
void set_bar_state(Foo, int);
```

Typically, these are implemented as member functions in the rewrite class.

The rewriter uses the function `get_bar_state` to lookup previously cached information set by the function `set_bar_state`. Typically, the implementation of the two functions can be implemented as hash tables, using the addresses of the terms as hash functions and pointer equality comparisons. Note that caching can be *lossy*; i.e. it is perfectly alright for the cache to eliminate cached information to keep its size under control. If no cache information is found, the function `get_bar_state` should return `BURS::undefined_state`.

Class `RewriteCache` To make it easy for the users to implement their own external indices, two sample external index implementation are provided. The first is the class `RewriteCache` defined in the library file `<AD/rewrite/cache.h>`.

The class `RewriteCache` implements a simple fixed capacity cache with the following member functions.

```
 RewriteCache();
 RewriteCache(int capacity);
~RewriteCache();

 void clear();                    // after rewriting
 void initialize();               // before rewriting
 void initialize(int capacity);   // before rewriting/also set capacity

 int capacity() const;

 void set_state(const void * t, int s);
 int get_state(const void * t) const;

 void invalidate(const void * t);
```

The user should call the function `initialize` before rewriting in order to setup the cache and/or clear out previously cached result. The functions `set_state` and `get_state` can be used to implement the necessary state caching functions expected by the rewriter.

Returning to our `Wff` example, we have the following possible implementation of the external index:

```
rewrite class Simplify (Wff)
{  RewriteCache cache;
public:
   int get_wff_state(Wff t) const { return cache.get_state(t); }
   void set_wff_state(Wff t, int s) { cache.set_state(t,s); }
};

rewrite Simplify
{  index: Wff = extern wff;
   ...
};
```

Tips: If manual memory management is used, a term should not be deallocated from memory if a cache contains a handle to it. The method `invalidate` can be used to make sure the entry associated with a term is removed from the cache.

Class `GCRewriteCache` *This class has not be completely tested.*

Since a garbage collectable object is not reclaimed by the collector whenever there exists a reference to it, using the class `RewriteCache` may create problems since objects are not reclaimed if it is referenced by the cache. The template class `GCRewriteCache` solves this problem by using *weak pointers* in its internal implementation.

The template class `GCRewriteCache<T>` expects `T` to be a class derived from `GCObject`. This class implements the following functions:

```
 GCRewriteCache();
 GCRewriteCache(int capacity);
~GCRewriteCache();
```

```
void clear();                    // after rewriting
void initialize();               // before rewriting
void initialize(int capacity);   // before rewriting/also set capacity

int capacity() const;

void set_state(T * t, int s);
int get_state(T * t) const;

void invalidate(T * t);
```

The usage of these functions are identical to that of class `RewriteCache`.

6.6 Short circuiting rewrite paths with `cutrewrite`

From version 2.3.0 onward, the `cutrewrite`(e) statement may be used wherever a `rewrite`(e) statement is expected. The `cutrewrite` statement can be used to selectively ignore certain subterms during the rewriting process.

The semantics of `cutrewrite`(e) is to replace the current redex with e but do not bother with looking for other redexes in the replacement term. This means that the replacement term will not be re-traversed immediately. Furthermore, the replaced term will not match any other left hand side sub-patterns except pattern variables and wildcards.

This feature is very useful for preventing the rewriter from looking inside certain terms.

To demonstrate, consider the simple following example:

```
datatype Exp = NUM of int
             | ADD of Exp, Exp
             | SUB of Exp, Exp
             | MUL of Exp, Exp
             | DIV of Exp, Exp
             | FINAL of Exp
             ;
Exp e = ADD(NUM(1),MUL(NUM(2),NUM(3)));
```

Suppose we want to find all numbers within the expression `e` and increment their values by 1. The straight forward way of writing:

```
rewrite (e) type (Exp) {
   NUM x: NUM(x+1)
}
```

will not work, since the replacement term is a new redex, and the rewriting system will not terminate.

We can reimplement the rewriting system as a two stage process. First, we mark all terms that we do not want to be changed; whenever we find a marked term, we execute a `cutrewrite` to prevent the term from being changed. Then we unmark the terms.

```
rewrite (e) type (Exp) {
   NUM x:   FINAL(NUM(x+1));
preorder:
   FINAL _: cutrewrite;
}
```

```
rewrite (e) type (Exp) {
   FINAL x: x
}
```

In the first rewrite statement, replacement terms that we want to stay fixed are encapsulated within the auxiliary `FINAL` constructor. Recall that preorder rules are tried before the subterms of a redex are reduced. Thus we can make sure all that all terms that are arguments to a `FINAL` term is left unaltered by adding a `preorder` rule that executes a `cutrewrite`[10] Finally, after the first rewrite statement finishes, we use the second rewrite statement to remove all `FINAL` terms generated in the first.

The semantics of `cutrewrite` needs further explanation. Consider the following rewrite system:

```
rewrite (e) type (Exp) {
   NUM x:          cutrewrite(NUM(x+1))
|  MUL(X, NUM 1):  X
|  MUL(NUM 1,X):   X
}
```

Given the term `MUL(NUM(0),NUM(2))`, the subterms `NUM(0)` and `NUM(1)` will be rewritten by the first rule into `NUM(1)` and `NUM(3)` respectively. Furthermore, the replacement term `NUM(1)` will *not* match the left subpattern in `MUL(NUM 1,X)`. Thus the result of the rewrite will be `MUL(NUM(1),NUM(3))`.

Tips: Since the semantics of a rewriting system with `cutrewrite` can be tricky to analyze, its use should be avoided as much as possible.

6.7 Conditional failure with `failrewrite`

From version 2.3.0 onward, the `failrewrite` statement may be used wherever a `rewrite` statement is expected. The `failrewrite` statement is an alternate way of writing conditional rewriting rules, and it is often useful when the condition to be checked is too complex to be written as a guard.

When a `failrewrite` statement is executed within the rhs action of a rewrite rule, the current pattern is rejected and the rewriter will proceed to try other matches.

For instance, suppose the current redex is `DIV(INT 0,INT 0)` and the following rules are specified within a rewriting system:

```
   DIV(INT X,INT Y): { if (Y == 0) failrewrite; else rewrite(NUM(X/Y); }
|  DIV(INT 0,Y):     NUM(0)
```

The execution of these rules will proceed as follows: both patterns match the redex, but the first rule will be tried first. Since `Y` is zero, the `failrewrite` statement is executed. The rewriter will proceed to try the second rule and the redex will be replaced with `NUM(0)`. Note that if we omit the `failrewrite` statement in the first rule, as in:

```
   DIV(INT X,INT Y): { if (Y != 0) rewrite(NUM(X/Y); }
|  DIV(INT 0,Y):     NUM(0)
```

the rewriter will commit on the first rule and no action will be performed on the redex.

Like `rewrite` and `cutrwerite`, `failrewrite` acts as a branching statement and all statements immediatedly following `failrewrite` are dead code.

6.8 Confluence and termination

Currently no automatic checking is available to make sure that a set of rewriting rules is confluent and will terminate. This is currently the user's responsibility. Some verification features will be developed in the future.

[10] If an expression argument is not given to a `cutrewrite` statement, then the redex is left unchanged.

6.9 Debugging Tree Rewriting

By default, **Prop** generates a macro `DEBUG_`C for each rewrite class named C. The user can redefine this macro in order to trace the replacements that are performed during the execution of a set of rewriting rules. The default definition of `DEBUG_`C is as follows:

`#define DEBUG_`$C(R, \mathit{redex}, \mathit{filename}, \mathit{linenumber}, \mathit{ruletext})\ R$

i.e. it simply returns the value of R.

The arguments to the macro are as follows:

- R is the new replacement term.
- *redex* is the current redex, before the replacement has been performed.
- *filename* is the file name of the current rewriting rule.
- *linenumber* is the location of the current rewriting rule.
- *ruletext* is a string that describes the current rule.

In order to activate tracing, the user can redefine the macro `DEBUG_FOO` (for rewrite class `FOO`) as follows:

```
#define DEBUG_FOO(a,b,c,d,e) print_rule(a,b,c,d,e)

template <class T>
  T print_rule(T replacement, T redex,
               const char * file_name,
               int line, const char * rule_text)
  { cerr << file_name << ':' << line << ": " << rule_text
         << " redex = " << redex << " replacement = " << replacement
         << endl;
    return replacement;
  }
```

Note that these definitions must appear before the `rewrite FOO` declaration.

6.10 Optimizing Tree Rewriting

An experimental rewriting optimizer based on partial evaluation techniques has been implemented in release 2.3.4 of **Prop**. The optimizer can be enabled with the option `-Orewriting`.

Informally speaking, the rewriting optimizer will try to locate "longer" reduction sequence and add these to your rewriting rules. The result is that during execution these bigger reduction steps are taken whenever possible.

For example, consider the `Wff` datatype defined in section 6.4.1. Suppose we have the following two rewriting rules in our rule set:

```
   Not(Not X): X
|  FORALL(X,A,P): Not(EXISTS(X,A,Not(P)))
```

Given an input term of the form `FORALL(X,A,Not(P))`, the term will first be reduced to `Not(EXISTS(X,A,Not(Not(P))))` via the second rule, then to `Not(EXISTS(X,A,P))` via the first rule. The optimizer can discover that this two step reduction can be compressed into one step if the rewriting system is augmented with the following rule:

```
   FORALL(X,A,Not P): Not(EXISTS(X,A,P))
```

Note that this augmented rule is a *specialization* of the two rules given by the user. In general, the rewriting optimizer will repeatedly look for useful specializations and add them to the user rule set[11]. These specialized rules will perform multiple reductions in one single step.

To make use of the optimizer, the rewriting system must have the following properties:

1. It must be strongly normalizing, i.e. it will terminate under all inputs.
2. It must be confluent, or else confluence is not necessary. Notice that the optimizer can and will alternate the reduction order of a rule set. Thus if a rewriting system is non-confluent, an optimized rewriting system may give a different result.

In addition, the following things should hold to maximize the effectiveness of the optimizer.

1. Indices (i.e. state caching) should be enabled. Otherwise the optimizer will not be able to optimize a rule.
2. Specializable rules must be simple, i.e. the right hand side must be written as an expression consist of only **Prop** datatypes. If the right hand side is a complex statement or if it involves external function calls, the optimizer will fail to analyze it properly.
3. Only bottom-up rules may be present.
4. Avoid conditional rules whenever possible.

Finally, it should be warned that an optimized rewriting system may generate a lot more code than the un-optimized one. The user should view the generated report and check that excessive specializations have not been performed.

[11] To ensure termination of the optimization process, augmented rules will not be further specialized.

7 User defined datatypes: Views

For many applications it is necessary to interface **Prop** to data structures predefined in libraries, or generated by other tools. In order to make it possible to work with these external data structures, **Prop**, from version 2.3.3. onward, provides a *view* mechanism to interface with these data structures. Using views, an externally defined type can be used within **Prop**'s pattern matching and rewriting mechanisms transparently, just as if it were defined by **Prop**. The **Prop** translator will call the appropriate interface functions specified by the user to manipulate these external datatypes.

7.1 A first example

. To illustrate the use of views, we'll begin with a simple example.

Suppose in our application we use the following tagged C union to represent our expression datatype. An expression is represented simply as a pointer to this structure. An null expression is represented simply as the NULL pointer.

```
enum exp_tag { Int = 1, Ident, Add, Sub, Mul, Div };
struct exp {
   enum exp_tag tag;
   union {
      int          number;
      const char * ident;
      struct { struct exp * l, * r } children;
   } u;
};
typedef struct exp * Exp;
/* User defined constructor functions */
Exp NONE = NULL;
extern Exp INT(int);
extern Exp IDENT(const char *);
extern Exp ADD(Exp, Exp);
extern Exp SUB(Exp, Exp);
extern Exp MUL(Exp, Exp);
extern Exp DIV(Exp, Exp);
```

Suppose this data structure is already used extensively in other modules, and we'd like to **Prop** pattern matching mechanisms while keeping the old code intact. We can accomplish this by declaring a view to this data structure, using a `datatype view` definition:

```
dataype Exp :: view =
   match (this ? this->tag : 0)
     view 0     => NONE
   | view Int   => INT(int = this->u.number)
   | view Ident => ID (const char * = this->u.ident)
   | view Add   => ADD(Exp = this->u.children.l, Exp = this->u.children.r)
   | view Sub   => SUB(Exp = this->u.children.l, Exp = this->u.children.r)
   | view Mul   => MUL(Exp = this->u.children.l, Exp = this->u.children.r)
   | view Div   => DIV(Exp = this->u.children.l, Exp = this->u.children.r)
   ;
```

In the above, a view named `Exp` is defined. Let's describe what the definition means: in general, a view definition has three main parts, the *view selector*, a set of *view transformation rules*, and a set *of view accessor expressions*. These parts perform the following tasks:

View selector The *view selector* is specified in the the `match` part of the definition. In this example, the expression associated with `match` is `this ? 0 : this->tag`. This returns a variant discrimination integer from `0` to `Div`. Note that the pseudo variable `this` refers to an object of type `Exp`.

View transformation rules Each of the clauses

`view`*exp*`=>`*term*

specifies a *view transformation rule*. These rules are used to specify how to transform a computed variant tag into a pattern. In general, expression *exp* must be a constant expression usable in a `case` statement.

View accessors In each of the constructor arguments, an *view accessor* expression can be defined to access the logical components of the constructor. In the rule:

```
| view Add   => ADD(Exp = this->u.children.l, Exp = this->u.children.r)
```

The expressions `this->u.children.l` and `this->u.children.r` tell **Prop** how to access the left and right child of the an addition expression node.

Note that since the variant `NONE` has no arguments, it is represented as a nullary constructor, and no accessors expressions are defined.

A pretty printing function for `struct exp` can be implemented in **Prop** using pattern matching as follows:

```
ostream& operator << (ostream& s, Exp e)
{  match (e)
   {  NONE:     { s << "none"; }
   |  INT i:    { s << i; }
   |  ID id:    { s << id; }
   |  ADD(a,b): { s << '(' << a << " + " << b << '); }
   |  SUB(a,b): { s << '(' << a << " - " << b << '); }
   |  MUL(a,b): { s << '(' << a << " * " << b << '); }
   |  DIV(a,b): { s << '(' << a << " / " << b << '); }
   }
   return s;
}
```

Note that this is exactly the same code we would've written if `Exp` is defined as a datatype rather than a view. This means that the user can reimplement the expression data structure as a **Prop**'s datatype at a latter time, without altering the **Prop** source code. In effect, the *logical* view of a datatype, and its *physical* implementation can separately. With this *representation transparency* capability, programs written in **Prop**'s pattern matching formalisms can evolve in an incrementally and pain-free manner.

The generated code for the above function will resemble the following output. Note that all the view selectors and view accessors are inlined into the code, and in general, views incur no overhead to their use.

```
ostream& operator << (ostream& s, Exp e)
{  switch (e ? e->tag : 0)
   {  case 0:      { s << "none"; }
      case Int:    { s << e->u.number; }
      case Ident:  { s << e->u.ident; }
      case Add:    { s << '(' << e->u.children.l
                       << " + " << e->children.r << '); }
      case Sub:    { s << '(' << e->u.children.l
                       << " - " << e->children.r << '); }
      case Mul:    { s << '(' << e->u.children.l
```

```
                    << " * " << e->children.r << '); }
      case Div:    { s << '(' << e->u.children.l
                    << " / " << e->children.r << '); }
   }
   return s;
}
```

7.2 Another view example

Suppose we use the following alternative implementation, a `Expression` class hierarchy to represent an expression in our AST. Note that `Exp` is an abstract base class inherited by the classes `Number`, `Operator`, `Identifer`, etc.

```
class Expression
{
public:

   // This class uses a ``wide'' interface
   virtual int  number() const         = 0;  // returns a number
   virtual const char * ident() const  = 0;  // returns an identifier

   // returns the ith subchild
   virtual const Expression * child(int) const = 0;
   // mutable version of above
   virtual Expression *& child(int) = 0;
};

typedef Expression * Exp;

class NullExp  : public Expression { ... };
class Number   : public Expression { ... };
class Operator : public Expression { ... };
class Identifier : public Expression { ... };
```

In order to provide a suitable interface to **Prop**'s view mechanism, we introduce an additional member function to return a variant tag:

```
class Expression
{
public:
   enum Variant { None, Int, Ident, Add, Sub, Mul, Div };
   virtual Variant get_variant() const = 0;
}
```

In addition, we assume the following constructor functions have been implemented:

```
extern Exp INT(int);
extern Exp IDENT(const char *);
extern Exp ADD(Exp, Exp);
extern Exp SUB(Exp, Exp);
extern Exp MUL(Exp, Exp);
extern Exp DIV(Exp, Exp);
```

Now, with these definitions in place, we can define a datatype view of the above `Expression` hierarchy using the following definition:

```
datatype Exp :: view
   = match (this->get_variant())
     view Expression::None  => NONE
   | view Expression::Int   => INT (int = this->number())
   | view Expression::Ident => ID  (const char * = this->ident())
   | view Expression::Add   => ADD (Exp = this->child(0),
                                    Exp = this->child(1))
   | view Expression::Sub   => SUB (Exp = this->child(0),
                                    Exp = this->child(1))
   | view Expression::Mul   => MUL (Exp = this->child(0),
                                    Exp = this->child(1))
   | view Expression::Div   => DIV (Exp = this->child(0),
                                    Exp = this->child(1))
   ;
```

An evaluation function can be written using **Prop**'s pattern matching construct:

```
int eval(Exp e, const Env& env)
{  match (e)
   {  NONE:          { return 0; }
   |  INT i:         { return i; }
   |  ID id:         { return env(id); }
   |  ADD (x,y):     { return eval(x,env) + eval(y,env); }
   |  SUB (x,y):     { return eval(x,env) - eval(y,env); }
   |  MUL (x,y):     { return eval(x,env) * eval(y,env); }
   |  DIV (x,INT 0): { cerr << "Division by zero"; return 0; }
   |  DIV (x,y):     { return eval(x,env) / eval(y,env); }
   }
}
```

Note that this code works equally well with the tagged union representation defined in section 7.1.

We can also use rewriting to transform an expression tree as follows

```
void simplify(Exp& e)
{
   rewrite (e) type (Exp)
   {  ADD(INT i, INT j): INT(i+j)
   |  SUB(INT i, INT j): INT(i-j)
   |  MUL(INT i, INT j): INT(i*j)
   |  DIV(INT i, INT j): INT(i/j)
   }
}
```

7.3 Syntax of view definitions

The general syntax of a view definition resembles that of a `datatype` definition:

Datatype_Spec ::= *Datatype_View_Spec*
Datatype_View_Spec ::= *Id* `::` `view` `=`
`match` `(` *Exp* `)`
View_Cons_Specs
View_Cons_Specs ::= `view` *Exp* `=>` *Cons_Spec* `|` ... `|` `view` *Exp* `=>` *Cons_Spec*

Default arguments in a *Cons_Spec* are interpreted as view accessors.

There are a few general rules to observe when defining a view:

1. A `datatype` `view` definition only defines the interface to an external data structure and no code is generated. In particular, the defined view name is not an actual C++ type. Thus the user is responsible for defining all appropriate type definitions.

2. All other qualifiers other than `view` are ignored. Thus automatic generation of pretty printers, garbage collection interface routines, etc. are disabled for views.

3. In general, the view accessors must be usable as lvalues if the user wants to modify a view datatype. This includes the situation when in-place rewriting is used.

4. Finally, the user are responsible for generating the constructor functions. In the two examples given above, the user should implement the functions `INT`, `ID`, etc.

8 Graph Types and Graph Rewriting

A typical compiler uses many types of graphs to represent internal program information. For instance, control flow graphs, data dependence graphs, call graphs, def/use chains are some of the most common examples. In order to automate the process of implementing and manipulating these graph structures, **Prop** provides a generic formalism– *graph types*– for specifying multisorted, attributed, hierarchical directed graphs in very high level manner. The data structure mapping process of translating high level graph specifications into concrete C++ classes is completely automated. Thus the user can concentrate on specifying the semantic relationships between the elements of the domain, instead of the implementation details.

The graph structure mapping process uses real-time set machine simulation[Pai89] and subtype analysis[CFH+91] to convert associative accesses involving sets, maps, and multimaps into hash free, worst-case constant time pointer manipulations. This optimization is performed transparently by the translator.

Manipulation of graphs structures are done in three ways:

object-oriented Using a standard interface generated from **Prop** the user can manipulate a graph in the usual procedural/object-oriented manner.

set formalism Using an embedded SETL[SDDS86]-like sublanguage, the user can manipulate the graphs using high level set operations such as set comprehension.

graph rewriting At the highest level, analysis and transformation can be carried out using the graph rewriting formalism.

In this section we'll describe each of these topics in detail.

8.1 Graph Types

A declaration is used to specify a graph structure with multiple sorts of labeled nodes and labeled directed edges. Attributes can be attached to both nodes and edges. The syntax of `graphtype` declarations is as follows:

Graph_Type ::= `graphtype` *Id*
  [`:` *Inherit_List]*
  [`::` *Graph_Mode* ... *Graph_Mode]*
`declare`
  `node:` *Node_Def* `|` ... `|` *Node_Def*
  `edge:` *Edge_Def* `|` ... `|` *Edge_Def*
`begin`
  Code
`end graphtype ;`

Graph_Mode ::=

Node_Def ::= *Id [[* `of` *] TypeExp]*

Edge_Def ::= *Id [* `of` *] Type_Exp* `->` *Type_Exp*
| *Id [* `of` *] Type_Exp* `<->` *Type_Exp*
| *Id [* `of` *] Type_Exp* `<=>` *Type_Exp*
| *Id [* `of` *] Type_Exp* `<=>*` *Type_Exp*

8.2 The Graph Interface

This section is incomplete.

8.3 Set Formalisms

This section is incomplete.

8.4 Graph Rewriting

This section is incomplete.

9 Running the Translator

The **Prop** translator is a program called `prop`. The translator uses the following command line syntax:

Running_Prop ::= `prop` [prop_options] *file* ... *file*

Here, *File* is a file with suffix `.p*`. By default, the output file will have the same name with the `p` extension removed. For example, a file named "foo.pC" will be translated into the file "foo.C" Similarly, a file named "bar.ph" will be translated into "bar.h"

9.1 Options

The available options to the translator are as follows:

`-G -GNU` Use GNU style error messages.

`-I`*`path`* Use the search path *path*. **Prop** will search *path* to look for **Prop** files with the suffix `.ph` inside a `#include` directive. Multiple paths can be specified by using multiple `-I` options.

`-l -no_line_directives` By default, **Prop** will try to generate `#line` directives to correlate the source and the translated output. This option suppresses this feature.

`-M -make_depends` Similarly to the `-M` option in `cc` and `cpp`. This option generates a suitable dependency list for Makefiles.

`-memory_usage` Print memory use during translation.

`-n -no_codegen` Don't output any code; perform semantic checking only.

`-N -non_linear` Use non-linear patterns during pattern matching. A non-linear pattern is one in which a variable occurs more than once. By default, **Prop** consider this an error.

`-o`*`outfile`* Send output to *outfile* instead of the default.

`-Ofast_string_match` Generate (potentially) faster string matching code by expanding string literal tests into character tests. By default, **Prop** generates string tests using `strcmp` and binary search.

`-Oadaptive_matching` Use the adaptive pattern matching algorithm[12].

`-Oinline_casts` Generate inline type casts instead of function calls that do the same. This may be faster for compilation.

`-Orewriting` Enable the rewriting optimizer. See also section 6.10 for details.

`-Otagged_pointer` Use a tagged pointer representation instead of representing variant tags explicitly in a structure field. For example, if there are two variants called `Foo` and `Bar` in a datatype. Then a pointer to `Foo` can be tagged with a low order bit 0 while a pointer to `Bar` can be tagged with a low order bit 1. Accessing the tag thus becomes just a simple bit test. This should save space and may be faster[13]

`-r -report` Generate a report whenever necessary. Parser, string matching, and rewriting constructs produce reports.

`-s -strict` Use strict semantic checking mode. All warnings are considered to be errors.

[12] The implementation may not be effective.
[13] However, this feature has not been fully tested.

`-S -save_space` Use space saving mode. Try not to use inline functions if code space can be saved. Datatype constructors will not be inlined in this mode. Instead, datatype constructors will be generated at where the corresponding `instantiate datatype` declaration occurs.

`-t -stdout` Send translated program to the standard output.

`-use_global_pool` Use global memory pool for allocation.

`-v`*num* Use verbose mode in report generation. This will provide more detailed information.

9.2 Error Messages

The following is a canonical list of all error messages generated by the **Prop** translator. Each message is prefixed by the file name and line number in which the error occurs. Most of these errors messages are self explanatory; more detailed explanations are provided below.

Error	Explanation
`unknown option` *option*	The translator does not recognize the command line *option*. Type *prop* to see a list of options.
`Error in command:` *command*	The translator fails when trying to execution *command*. When multiple files are specified in a command line, *prop* invokes itself on each of the file.
pat `with type` *type* `is not unifiable`	Pattern has not been declared to be a unifiable type.
`Sorry: pattern not supported in rewriting:` *pat*	Pattern *pat* is not supported. Currently, logical pattern connectives are not supported in rewriting. You'll have to rewrite them into non-logical form.
`Unknown complexity operator:` *op*	
`accessor is undefined for view pattern:` *pat*	An accessor function has not been declared for a constructor. When using datatype views
`arity mismatch (expecting` n`) in pattern:` *pat*	Pattern *pat* is expected to have arity n. The arity is the number of expressions that you're matching concurrently. If you are matching n objects then there must be n patterns for each rule.
`arity mismatch in logical pattern:` *pat*	Logical patterns do not match the arity of a pattern.
`bad constructor type` *type*	
`bad view constructor pattern:` *pat*	
`can't find include file` *file*	Additional search paths can be specified with the `-I` option
`component #`i `not found in constructor` *con*	Datatype constructor *con* does not have a component named $\#i$. This can happen when you write a record constructor expression and misspelt one of the labels.
`component` l `not found in constructor` *con*	Datatype constructor *con* does not have a labeled component l.
`constructor` *con* `already has print formats`	
`constructor` *con* `is not a class`	
`constructor` *con* `is not a lexeme`	A constructor is used as a terminal in the parser but it has not been declared to be a lexeme.
`constructor` *con* `doesn't take labeled arguments`	You're trying to use record constructor syntax on a constructor take does not take record arguments.
`constructor` *con* `is undefined`	
`cyclic type definition in type` *id* `=` *type*	Type abbreviations cannot be cyclic. For recursive types, use `datatype` definitions.
`datatype` *T* `is not a lexeme type`	T is used as a terminal within syntax class when it has not been declared as a lexeme type. All non-terminals must be a defined using the `lexeme` qualifier.
`determinism` *det* `not recognized`	

Error	Explanation
duplicated definition of pattern constructor *'con'*	The constructor *con* has already been in another datatype. *Prop* does not allow overloading of constructor names.
duplicated label *'l'* in expression: *exp*	A record type has one of its labels duplicated.
duplicated label *'l'* in type *type*	
duplicated pattern variable *'id'*. Use option -N	By default, pattern variables may not be duplicated within a pattern. Non-linear patterns are allowed only when the option -N is invoked.
edge e is not defined in graphtype *id*	
empty type list in rewrite (...) ...	
expecting c_1 ... c_2 (from line l) but found c_3 ... c_4 instead	Quotation symbols are not properly balanced.
expecting non-terminal *nt* to have type t_1 but found t_2	Non-terminal *nt* has already been defined to have a synthesized attribute type of t_1 but t_2 is found. This can happen if you have rules with the same lhs non-terminal previously defined.
expecting type t_1 but found t_2 in pattern variable *'v'*	*Prop* performs type inference on all the lhs patterns to determine the types of all variables. Type errors can occur if the patterns are miswritten.
expecting type t_1 but found t_2	
flexible vector pattern currently not supported in rewriting: *pat*	
format #i used on constructor *con*	
format #l used on non-record constructor *con*	
function name mismatch: expecting f ...	
illegal character c	
illegal context type: *type*	A context type in a matchscan statement must be previously defined to be datatype. The unit constructors of the datatype can be used as context values.
illegal context(s) in pattern *pat*	
illegal format '_' on constructor *con*	
illegal incomplete record pattern: *pat*	
illegal print format 'c' in constructor *con*	
illegal record label *'l'* in expression: *exp*	
illegal width in bitfield *'id* (n)'	
inherited attribute *'type'* can only be used in treeparser mode in rewrite class *id*	

Error	Explanation
`law id(...) = pat is not invertible`	Pattern *pat* cannot be treated as an expression. It may involve logical patterns and wild cards.
`law 'id': bound variable 'v' is absent in body exp`	A parameter variable v must be present within the body of a law.
`law 'id': type type cannot be used in parameter id`	
`lexeme id is undefined`	
`lexeme class id is undefined`	
`lexeme pattern is undefined for constructor con`	
`lexeme {id} already defined as regexp`	
`lexeme {id} is undefined in regexp`	
`missing '}' in regular expression regexp`	
`missing label 'l' in expression: con exp`	
`missing non-terminal in tree grammar rule: nt`	
`missing type type in the traversal list of rewrite class id`	Within the rewriting rules, you have used a pattern that involve a constructor of type *type* directly but no such types are defined in the rewrite class definition. You should add the type to the traversal list.
`missing type info in expression exp : type`	Sometimes *prop* fails to infer the type of an expression within a *match* statement. It such cases it is necessary to help out the translator by adding additional type information in the patterns. For example, rewrite some pattern p as $(p : \tau)$ where τ is the type of p.
`missing type info in function f type`	Similar to above
`missing type info in rule: f pat : type`	Similar to above.
`missing view selector for pattern: pat`	
`multiple mixed polarity pattern variable 'v'`	
`multiple unit constructors in polymorphic type id arg is not supported`	
`negative cost cost is illegal`	
`node id is not defined in graphtype id`	
`non lexeme type type in pattern pat`	
`non-relation type type in pattern: pat`	
`pattern is undefined for lexeme l`	
`pattern scope stack overflows`	
`pattern scope stack underflows`	

Error	Explanation
pattern variable 'v' has no binding at this point	
persist object id is undefined for *con*	
persistence pid *name* already allocated for type *type*	
persistence redefined for type *type*	
precedence symbol must be terminal: *term*	
predicate *p*: expecting type t_1 but found t_2	
redefinition of bitfield '*field*'.	
redefinition of constructor '*con*'	
redefinition of datatype *id*	
redefinition of lexeme class *id*	
replacement not in rewrite class: rewrite *exp*	
rewrite class *id* has already been defined	
rewrite class *id* is undefined	
rule r has incomplete type *type*	
rule r is of a non datatype: *type*	
syntax class *id* has already been defined	
syntax class *id* is undefined	
synthesized attribute '*type*' can only be used in treeparser mode in rewrite class *id*	
the class representation of constructor *id* has been elided	
this rule is never selected: r	
this rule is never used: r	
too few arguments *arg* in instantiation of type scheme *type*	
too many arguments *arg* in instantiation of type scheme *type*	
type *type* is not a datatype	
type *type* is undefined	

Error	Explanation
type *id* = *type* is not a datatype	
type *id* has already been defined as *type*	
type *id* has unknown class: *C*	
type *'type'* is not rewritable in treeparser mode rewrite class *id*	All datatypes used within the treeparser mode of rewriting must be defined with the `rewrite` qualifier.
type error in pattern *pat*: *type*	
type mismatch between nonterminal *nt*(type t_1) and rule r(type t_2)	
type mismatch in pattern: *pat*	
type mismatch in rule r	
type mismatch in rule r *pat*	
type or constructor *con* is undefined	
unable to apply pattern scheme *pat*	
unbalanced c_1 ... c_2 at end of file	
undefined non-terminal *nt*	
unification fails occurs check with t_1 and t_2	
unmatched ending quote c	
unrecognized quoted expression ‘*exp*‘	
unrecognized quoted pattern ‘*pat*‘	

A Garbage Collection in the Prop Library

In this appendix we describe the design and implementation of a garbage collector based on the Customisable Memory Management framework(CMM)[AF] in our **Prop** C++ class library. Like the previous approach, we have implemented a mostly copying conservative collector based on the work of Bartlett[Bar88]. Similar to CMM, our architecture provides a protocol to allow multiple garbage collectors using different algorithms to coexist in the same memory space. A few improvements are made to improve the performance, the flexibility and the functionality of our collector: to reduce retention due to false roots identification, blacklisting[Boe93] is used to identify troublesome heap addresses; the architecture of the system has been generalized so that it is now possible to have multiple instantiations of Bartlett-style heaps; finally, object finalization and weak pointer support are added. Our collector has been tested on gcc 2.5.8 under Linux, and SunOS 4.1.x[14]

A.1 Introduction

The **Prop** project involves the design and implementation of an environment and an extension language based on C++ for compiler construction and program transformation. An outgrowth of this project is the **Prop** C++ class library, which contains an extensive set of support algorithms and data structures. Since a typical compiler manipulates many complex tree, dag and graph objects with extended lifetime, manual memory management using C++'s constructors and destructors, reference counting smart pointers, or some other techniques is frequently complex and error prone. Furthermore, with the advent of new algorithms for garbage collection, manual memory management techniques do not necessarily provide better performance or space utilization. To make it possible to make use of garbage collection as a viable alternative for memory management in C++[ED93], we have implemented a garbage collection class hierarchy as part of the **Prop** library. The class library can be used directly by the users who'd like to program in C++; it can also be used as part of the runtime system of a highly level language implemented in C++, as we have done for **Prop**.

We have several good reasons to prefer garbage collection over manual memory management. The **Prop** language contains many declarative constructs and extensions such as algebraic datatypes, pattern matching, rewriting, and logical inference. When a user programs in **Prop** using these constructs, an applicative style of programming is the most natural paradigm.

A.2 The Framework

We base our design on the work on the Customisable Memory Management(CMM) system[AF]. In this framework, multiple independent heaps(including the usually non-collectable heap) can coexist with each other. Bartlett's mostly copying garbage collector is used as the primary collector. CMM extends the work of Bartlett[Bar88] by allowing cross heap pointers and unrestricted interior pointers.

However, all collectable objects are required to derive from a base class and reimplement a tracing method, which identifies the internal pointers of an object. This burden is usually quite small and in fact can be automated from the type definition[15]

One of the major advantages of the CMM framework is that multiple coorperating collectors can coexist at the same time, which makes it possible to customize the behavior of each collector with respect to the lifetime behavior of the objects. In [AF], examples are given in which the lifetime of certain class of objects exhibit first-in/first-out behavior. In this case a special collector can be written to take full advantage of this property.

[14] An alpha version also works under Solaris. Thanks to Paul Dietz for the patch.

[15] In **Prop**, a special keyword `collectable` is used to identify garbage collected classes and types. The **Prop** translator uses this type annotation to derive the appropriate tracing methods.

A.3 Our Framework

Our framework retains all the benefits and flexibilities of CMM, while extending it in several minor but useful ways:

- Like CMM, interior pointers(i.e. pointers to the middle of an object) and crossheap pointers (i.e. complex data structures linking nodes locating in multiple logical heaps.) are supported. Pointers to collectable objects are non-intrusive: i.e. they are normal C++ pointers and not encapsulated in templates.
- Also like CMM, we allow multiple garbage collectors using different algorithms to coexist. Unlike CMM, however, we allow multiple Bartlett-style collectors to be instantiated. Most of the services involving low level page management and object marking have been relegated to a separate heap manager.
- We provide support for finalization and weakpointers.
- We have implemented blacklisting[Boe93] to reduce the chance of false roots identification.

A.4 The Implementation

Our implementation has been completely written from scratch since we do not desire to utilize copyrighted code and, more importantly, we have to make a few important architectural changes: All low level memory management services, such as management of the page table and the object bitmap, is now relegated to the class `GCHeapManager`. Collectors now act as clients as of the heap manager and the Bartlett-style collector no longer has any special status.

A.5 Architecture

The architecture of the memory management system is partitioned into a few classes, each responsible for providing distinct services:

- `GCHeapManager` — The heap manager. The heap manager manages the heap table, performs page level allocation and deallocation, and provides miscellaneous services like blacklisting. It also manages the object bitmaps.
- `GC` — The base class for all garbage collectors. This base class describes the protocol used by all the collector classes[16]
- `CGC` — The base class for conservative collectors. This class is inherited from class `GC` and implements some methods for locating the stack, heap, and static data areas.
- `BGC` — Bartlett-style mostly copying collector. This class is inherited from class `CGC` and implements the Bartlett mostly copying algorithm.
- `MarkSweepGC` — Mark/sweep style conservative collector. This class is inherited from class `CGC` and implements a mark/sweep collection algorithm.
- `WeakPointerManager` — The weakpointer manager. This class manages the weak pointer table and provides a few weak pointer scavenging services for the collectors.

A.6 The Programmatic Interface

The programmatic interface to the garbage collector involves two base classes, `GC` and `GCObject`. The base class `GC` provides an uniform interface to all types of collectors and memory managers while class `GCObject` provides the interface to all collectable classes. Table 1 contains a listing of the classes in our hierarchy.

[16] This base class is also inherited from class `Mem`, so that it adheres to the **Prop** memory management protocol.

Class	Purpose
GCObject	Collectable object base class
GC	Garbage collector base class
CGC	Conservative garbage collector base class
BGC	Bartlett style mostly copying collector
MarkSweepGC	A mark-sweep collector
UserHeap	A heap for pointerless object
GCVerifier	A heap walker that verifies the integrity of a structure

Table 1: Garbage Collection Classes.

Class `GCObject` is extremely simple: it redefines the operators `new` and `delete` to allocate memory from the default collector, and declares a virtual method "`trace`" to be defined by subclasses (more on this later.)

Memory for a `GCObject` is allocated and freed using the usually `new` and `delete` operators. Of course, freeing memory explicitly using `delete` is optional for many subclasses of `GC`.

```
class GCObject {
public:
   inline void * operator new(size_t n, GC& gc = *GC::default_gc)
      { return gc.m_alloc(n); }
   inline void * operator new(size_t n, size_t N, GC& gc = *GC::default_gc)
      { return gc.m_alloc(n > N ? n : N); }
   inline void   operator delete(void *);
   virtual void trace(GC *) = 0;
};
```

A.7 Memory Allocation

The base class `GC` is slightly more complex, as it has to provide a few different functionalities. The first service that class `GC` must provide is of course memory allocation and deallocation. As a time saving device we can specify what the default collector is using the methods `get_default_gc` and `set_default_gc`. Method `GCObject::new` will use this collector by default, unless placement syntax is used. Method `GCObject::delete`, on the other hand, can correctly infer the proper collector to use by the address of the object.

The low level methods to allocate and deallocate memory are `m_alloc` and `free` respectively. The programmers usually do not have to use these methods directly.

The method to invoke a garbage collection of a specific level is `collect(int level)`. This forces an explicit collection. Method `grow_heap(size_t)` can also be used to explicitly increase the heap size of a collector. Depending of the actual behavior of the subclasses, these methods may have different effects.

```
class GC {
protected:
   static GC * default_gc;
public:
   static GC&  get_default_gc()       { return *default_gc; }
   static void set_default_gc(GC& gc) { default_gc = &gc; }
   virtual void * m_alloc   (size_t) = 0;
   virtual void   free      (void *);
   virtual void   collect   (int level = 0) = 0;
   virtual void   grow_heap (size_t) = 0;
   static  void garbage_collect() { default_gc->collect(); }
```

```
    virtual void set_gc_ratio(int);
    virtual void set_initial_heap_size (size_t);
    virtual void set_min_heap_growth   (size_t);
};
```

A.8 The GC Protocol

The collector and collectable objects must cooperate with each other by abiding to a simple protocol:

- All objects that are to be garbage collected must be derived from `GCObject`. The application programmer must also supply a "**tracing**" method. The purpose of this method is to identify all internal pointers to other `GCObject`'s. This method is not used by the application programmer directly but only used internally by the garbage collectors.
- The tracing method of each collectable must in turn call the tracing method in the class `GC` with each pointer to a collectable object that the object owns:

  ```
  class GC {
  public:
     virtual GCObject * trace (GCObject *) = 0;
     inline  void trace (GCObject& obj);
  };
  ```

  Briefly, the rules are as follows:

  1. The tracing method of a collectable `Foo` has the following general form:

     ```
     void Foo::trace(GC * gc)
     {
        ...
     }
     ```

  2. If class `Foo` has a member that is a pointer `p` to a collectable object of type `Bar`, then add:

     ```
     p = (Bar)gc->trace(p);
     ```

     to the body of `Foo::trace`.
  3. If class `Foo` has a member object `x` that is a subclass of `GCObject`, also add:

     ```
     gc->trace(x);
     ```

     to the body of `Foo::trace`.
  4. If class `Foo` is derived from a class `Bar` that is a subclass of `GCObject`, add:

     ```
     Bar::trace(gc);
     ```

     to the body of `Foo::trace`.

  Notice that these methods can be arranged in any order.

This protocol can be used by both copying and non-copying collectors. In addition, the class `GCVerifier` also uses this protocol to walk the heap in order to verify the integrity of a garbage collected data structure.

A.9 Messages and Statistics

All garbage collectors use the following protocols for status reporting and statistics gathering.

```
class GC {
public:
   enum GCNotify {
      gc_no_notify,
      gc_notify_minor_collection,
      gc_notify_major_collection,
      gc_notify_weak_pointer_collection,
      gc_print_collection_time,
      gc_print_debugging_info
   }
   virtual int      verbosity() const;
   virtual void     set_verbosity(int);
   virtual ostream& get_console() const;
   virtual void     set_console(ostream&);
};

class GC {
public:
   struct Statistics {
      const char *   algorithm;
      const char *   version;
      size_t         bytes_used;
      size_t         bytes_managed;
      size_t         bytes_free;
      struct timeval gc_user_time;
      struct timeval gc_system_time;
      struct timeval total_gc_user_time;
      struct timeval total_gc_system_time;
   }
   virtual Statistics statistics();
};
```

Each collector has a current verbosity level, which can be set and reset using the methods `set_verbosity(int)` and `verbosity() const`. The verbosity level is actually a bit set containing flags of the type `GC::GCNotify`. A combination of these options can be used.

- `gc_no_notify` — no messages at all.
- `gc_notify_minor_collection` — all minor collections will be notified by printing a message on the current console.
- `gc_notify_major_collection` — similarly for all major collections.
- `gc_notify_weak_pointer_collection` — notify the user when weak pointers are being collected.
- `gc_print_collection_time` — this option will let the collector to print the time spent during collection (and finalization).
- `gc_print_debugging_info` — this option will print additional information such as stack and heap addresses during garbage collection.

The current console of a collector is defaulted to the C++ stream `cerr`. It can be set and inspected with the methods `set_console(ostream&)` and `ostream& get_console()` respectively. For example, the user can redirect all garbage collection messages to a log file `"gc.log"` by executing the following during initialization:

```
ofstream gc_log("gc.log");
GC::get_default_gc().set_console(gc_log);
```

A.10 The Bartlett style mostly copying collector

Currently a Bartlett-style mostly copying collector has been implemented. The current version is non-generational but we expect that a generational version will be available in the near future.

The Bartlett-style collector is implemented as the concrete class `BGC`. Aside from all the services provided by class `GC`, `BGC` also provides the following method:

```
class BGC : public CGC {
public:
   virtual void set_gc_ratio(int);
   virtual void set_initial_heap_size (size_t);
   virtual void set_min_heap_growth   (size_t);
}
```

The gc ratio refers to the ratio of used heap space versus total heap space. Class `BGC` will invoke the garbage collection if this is exceeded. The default gc ratio for `BGC` is 75%.

The initial size of the heap and the minimal number of bytes to request from the system during a heap expansion can be adjusted using the methods `set_initial_heap_size` and `set_min_heap_growth` respectively. These methods are declared in the `GC` protocol. By default, the initial heap size for this collector is 128K and each heap expansion will increase the heap by 512K.

If an application uses a lot garbage collected storage it is a good idea to set the heap size to larger capacity during initialization. Otherwise, the collector will have to perform more garbage collection and heap expansions to reach a stable state.

A.11 The Mark Sweep collector

An alternative to the copying collector is the the mark sweep collector. Like the previous collector, this collector also uses conservative scanning to locate roots. Unlikely the Boehm collector, type accurate marking is used through the user supplied tracing method.

Since the default collector is of the `BGC` variety, the user must create an instance of `MarkSweepGC` if the mark sweep collector is to be used.

```
class MarkSweepGC : public CGC {
public:
   virtual void set_gc_ratio(int);
   virtual void set_initial_heap_size (size_t);
   virtual void set_min_heap_growth   (size_t);
}
```

The gc ratio in class `MarkSweepGC` determines whether heap expansion is to be performed after a collection. The default gc ratio is 50%, thus if after garbage collection the ratio of used versus total heap space exceeds one half, the heap will be expanded.

For most purposes the two collectors are interchangeable. Since all types of garbage collectors under our framework use the same protocol, applications can be written without fixing a specific garbage collection algorithm before hand.

A.12 Finalization

One common C++ idiom uses constructor and destructor for resource allocation: resources (including memory but may include others, such as file handles, graphical widgets, etc) that are acquired in a class constructor are released(finalized) in the class'es destructor.

There are, however, two opposing views in how finalization should be handled in a garbage collector. The first assumes that garbage collection simulates an infinite amount of memory and so automatic finalization is inapproriate. The second takes a more pragmatic view, and assumes that automatic finalization should be provided since otherwise explicit resource tracking must be provided by the user for collectable datatypes, making garbage collection much less useful than it can be.

We will refrain from participating in any religious and philosophical wars on which dogma is the "One True Way". nstead, both types of collectors are provided.

By default, all garbage collection classes do not perform finalization on garbage collected objects. If object finalization is desired then the user can do one of two things:

- Enable the finalization of the default heap by putting

  ```
  GC::get_default_gc().set_finalization(true);
  ```

  in the initialization code, or
- Instantiate a different instance of the garbage collector and allocate objects needing finalization from this collector. This may be a better method since not all objects need finalization and those that do not can still be allocated from the default collector. This way we only have to pay for the performance penalty (if any) proportional to the usage of finalization.

For example, if a collectable class named `Foo` should be properly finalized we can declare it in the following manner:

```
extern BGC bgc_f;  // somewhere an instance is defined

class Foo : public GCObject {
   Resource r;
public:
   Foo() { r.allocate(); /* allocate resource */ }
  ~Foo() { r.release(); /* release all resource */ }

   // Make object allocate from bgc_f instead
   // of the default non-collectable gc.
   void * operator new (size_t n) { return bgc_f.m_alloc(n); }
};
```

When an instance of class `Foo` is identified as garbage, its destructor will be called. [Aside: *currently issue such as finalization synchronization is not handled directly by the collector. So if there is synchronization constraints during finalization it must be handled by the object destructor. Future versions of* **Prop** *will provide subclasses with more sophisticated finalization mechanisms.*]

Notice that the default `delete` operator of all garbage collectable classes will call the object destructor explicitly. It is a good idea to use explicit `delete` if finalization is a rare need since this service involves an explicit scan of the garbage memory, which may degrade performace with excessive swapping: without finalization the garbage pages will not have to be referenced with a copying collector.

It should also be noted that since the collector is conservative, there is no guarantee that garbage will be identified as such: there is no guarantee that all garbage resources will be released. In general, however, the efficacy of the collector is quite good and so non-critical or plentiful resources can be safely finalized with this collector.

A.12.1 Weak Pointers

It is frequently very useful to be able to keep track of garbage collectable objects through the use of "**weak pointers**." Collectors ignore the presence of weak pointers to garbage collected objects; objects with only referencing weak pointers will still be collected. Weak pointers to objects that become garbage will be automatically reset to null.

Weak pointers are provided through a smart pointer template `WeakPointer`, whose definition is shown below:

```
template <class T> class WeakPointer {
   public:
      inline WeakPointer();
      inline WeakPointer(const WeakPointer<T>& wp);
      inline WeakPointer(T * ptr);
      inline WeakPointer<T>& operator = (const WeakPointer<T>& wp);
      inline WeakPointer<T>& operator = (T * ptr);
      inline bool is_null() const;
      inline operator const T * () const;
      inline operator       T * ();
      inline const T * operator -> () const;
      inline       T * operator -> ();
      inline const T&  operator *  () const;
      inline       T&  operator *  ();
   };
```

A weakpointer to a garbage collectable class `A` can be defined as `WeakPointer<A>`. If a **Prop** datatype `T` has been defined, a weakpointer to type `T` can defined using the `classof` keyword. For example:

```
datatype Wff :: collectable = True | False | And(...)  | ...;

WeakPointer<classof Wff> a = And(True,False);
```

A.12.2 The Heap Walker

There is a simple class called `GCVerify` that can be used to verify the integrity of a complex allocated data structure. This is frequently useful during the development of a new collector, or a complex class derive class of `GCObject`.

The interface of this class is shown below.

```
class GCVerifier : protected GC {
public:
   virtual bool is_valid_pointer          (GCObject *);
   virtual bool is_valid_interior_pointer (GCObject *);
   virtual bool is_valid_structure        (GCObject *);
   virtual bool is_valid_pointer          (GCObject *, GC *);
   virtual bool is_valid_interior_pointer (GCObject *, GC *);
   virtual bool is_valid_structure        (GCObject *, GC *);
   size_t number_of_nodes() const;
};
```

Class `GCVerify` is derived from class `GC` so that the same object tracing protocol can be used. Briefly, three different methods are provided for verification:

- Call to `is_valid_pointer(p,gc)` verifies that `p` is a valid pointer to an object within the collector `gc`.
- Call to `is_valid_pointer(p,gc)` verifies that `p` is a valid interior pointer to an object within the collector `gc`.
- Call to `is_valid_structure(p,gc)` verifies the entire structure reachable by `p` is valid. This method uses `GCObject::trace` to locate the correct sub-structures.

There are alternative versions of the same methods that assumes the default collector is used. Furthermore

- Method `number_of_nodes()` returns the node count of the last structure traced using `is_valid_structure`, and
- If `set_verbosity(GC::gc_print_debugging_info)` is used then error messages will be printed during structure tracing.

References

[AF] Guiseppe Attardi and Tito Flagella. A customisable memory management framework. Technical report, University of Pisa.

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, California, Febrarry 1988.

[Bar89] Joel F. Bartlett. Mostly-copying collection picks up generations and C++. Technical Report TN-12, DEC Western Research Laboratory, Palo Alto, California, October 1989.

[Boe93] Hans-Juergen Boehm. Space efficient conservative garbage collection. *ACM SIGPLAN PLDI*, pages 197–206, 1993.

[CFH+91] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type transofrmation and data structure choice. *In B. Moeller, editor, Constructing Programs From Specifications*, pages 126–164, 1991.

[ED93] J.R. Ellis and D.L. Detlefs. Safe, efficient garbage collection for C++. Technical Report CSL-93-4, Xerox Parc, 1993.

[HMM86] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1986.

[Les75] M. E. Lesk. LEX: a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[Pai89] R. Paige. Real-time simulation of a set machine on a RAM. *In N. Janicki and W. Koczkodaj, editor, Computing and Information*, II:69–73, May 1989.

[Pax90] V. Paxson. Using flex — a fast lexical analyzer. Technical report, The Regents of the University of California, May 1990.

[San94] Georg Sander. Graph layout through the VCG tool. Technical Report A03/94, Universität des Saarlandes, October 1994.

[San95] Georg Sander. *VCG: Visualization of Compiler Graphs (v.1.30)*, 1995.

[SDDS86] J. Schwartz, R. Dewar, D. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[Str91] Bjarne Stroustrup. *The C++ Programming Language. Second Edition*. Addison-Welsey, 1991.

Index

PROP_EXPLICIT_TEMPLATE_INSTANTIATION, 25

Classes
- BGC, 64, 68
- CGC, 64
- GCHeapManager, 64
- GCObject, 64, 66
- GCRewriteCache, 44
- GCVerifier, 66
- GCVerify, 70
- GC, 64–66, 68, 70
- IOLexerBuffer, 12
- IOLexerStack, 12
- LexerBuffer, 11
- MarkSweepGC, 64
- Mem, 64
- RewriteCache, 44
- WeakPointerManager, 64
- WeakPointer, 70

Command line options
- -G -GNU, 56
- -I*path*, 56
- -M -make_depends, 56
- -N -non_linear, 56
- -Oadaptive_matching, 56
- -Ofast_string_match, 56
- -Oinline_casts, 56
- -Orewriting, 47, 56
- -Otagged_pointer, 56
- -S -save_space, 57
- -fno-implicit-templates, 25
- -l -no_line_directives, 56
- -memory_usage, 56
- -n -no_codegen, 56
- -o*outfile*, 56
- -r -report, 56
- -r, 16
- -s -strict, 56
- -save_space, 25
- -t -stdout, 57
- -use_global_pool, 57
- -v*num*, 57
- -v, 16

graph type, 54

state caching, 42

Syntax
- Keywords
  - $*n*, 16
  - $$, 16
  - applicative, 37
  - before:, 38
  - bottomup:, 38
  - classof, 28, 70
  - collectable, 23, 29
  - cutrewrite, 45
  - datatype view, 49
  - datatype, 9, 23
  - edge:, 54
  - expect:, 14
  - extern, 23, 25, 42
  - failrewrite, 46
  - graphtype, 54
  - index:, 42
  - inline, 23, 25
  - instantiate datatype, 25, 30, 57
  - instantiate, 25
  - law, 23
  - left:, 14
  - lexeme class, 8, 9
  - lexeme, 8, 9, 23
  - matchall, 25
  - matchscan*, 10
  - matchscan, 10
  - match, 25, 50
  - node:, 54
  - persistent, 23, 30
  - postorder:, 38
  - preorder:, 38
  - refine persistent, 30
  - refine, 28
  - relation, 32
  - rewrite class, 37
  - rewrite, 23, 37, 41, 42
  - right:, 14
  - syntax class, 13
  - syntax, 14
  - this, 50
  - topdown:, 38
  - topdown, 37
  - treeparser, 37
  - type, 41
  - view, 50, 53
  - where type, 23
  - while, 26
  - with, 27
- Non–terminals
  - *Char*, 5

Cons_Specs, 23
Cons_Spec, 23
Context_Spec, 10
Cost, 25
Datatype_Decl, 23
Datatype_Qualifiers, 23
Datatype_Qualifier, 23
Datatype_Spec, 23, 53
Datatype_View_Spec, 53
Edge_Def, 54
Expect_Decl, 14
Graph_Mode, 54
Graph_Type, 54
Guard, 25
Id, 5
Index_Decl, 42
Index_Spec, 42
Instantiate_Decl, 25
Integer, 5
Lab_Pat, 27
Lab_Type_Exp, 24
Law_Arg, 24
Law_Spec, 24
Lexeme_Class_Decl, 8
Lexeme_Class_Eq, 8
Lexeme_Decl, 8
Lexeme_Eq, 8
Lexeme_Spec, 8
Match_Action, 25
Match_Mode, 25
Match_Rule, 25
Matchscan_Action, 10
Matchscan_Mode, 10
Matchscan_Rule, 10
Matchscan, 10
Match, 25
Node_Def, 54
One_Alt, 15
Operator, 14
PatArg, 27
Pat_Var, 27
Pat, 27
Placement_Cons, 29
Precedence_Decl, 14
Production_Rule, 15
Quark, 5
Real, 5
Refine_Decl, 29
Refine_Spec, 29
Rewrite_Action, 37
Rewrite_Class_Decl, 37
Rewrite_Decl, 37
Rewrite_Mode, 37
Rewrite_Modifier, 37
Rewrite_Rule, 37
Rewrite_Stmt, 41
Running_Prop, 56
Simple_Cons_Spec, 23
Stmt, 5
String, 5
Symbol, 15
Syntax_Class_Decl, 13
Syntax_Decl, 14
Token_Spec, 9
Tokens_Decl, 9
Type_Exp, 24
Type_Qualifier, 24
Type_Spec, 24, 25
View_Cons_Specs, 53