Reflexive Top Down Parsing and Unparsing for C# Objects

Introduction

The parseable library described here provides a simple means of parsing text representing C# objects and for unparsing these objects back into text.

Suppose we have a simple class representing a tree structure in which each node has a name represented by some type identifier, essentially a string, and a list of subtrees.

```
class tree
{
  private identifier  name;
  private List<tree>  subtrees;
}
```

We wish to parse text similar to the following to generate instances of the class tree.

A tree with no subtrees.

```
root()
```

A tree with two subtrees, the second of which itself has two subtrees. Note: White space is ignored by the parser.

```
root(
       a()
       b(
          b1()
          b2()
        )
    )
```

To achieve this we first make the tree class inherit from the class parsable, defined by the library, and decorate the original fields of the class with some attributes. Two extra “place holder” fields are required to indicate where punctuation symbols are required in the input text. The parsing process does not assign values to place holder fields.

```
using System.Collections.Generic;
using parsable_objects;

class tree: parsable
{
  [Parse(1)     ] private identifier  name;
  [Parse(2, "(")] private punctuation open;
  [Parse(3)     ] private List<tree>  subtrees;
  [Parse(4, ")")] private punctuation close;
}
```

The first parameter of the Parse attribute indicates the order in which fields are to be parsed as there is no clean, backwards comparable way of determining the order of declaration via reflection. Fields of type punctuation are used to indicate where fixed symbols are required in the source text. The actual symbol is specified in the Parse attribute. Given this new class definition, all that is required (minus some exception handling) to parse a file containing text in the format shown above is the following function.

```
using System.IO;
using parsable_objects;

...

  tree parse_tree(FileStream file)
  {
    return parsable.parse<tree>(new source_reader(file));
  }

...
```

The source_reader class implemented by the library provides lexical analysis facilities allowing symbols representing identifiers, reserved words, numbers, delimited strings, etc., to be read sequentially from a source text. It can be initialised from a string, a StreamReader or a FileStream. In the above example the type identifier represents any sequence of characters starting with a letter, and subsequently containing only letters, digits and underscores, that is not defined elsewhere as a reserved word. Identifier objects have a property called spelling which represents the string actually parsed, e.g. some_tree_value.name.spelling.

The tree example is very simple and only involves one parsable class. More complex examples might include multiple parsable classes, for example, representing the abstract syntax tree of an entire programming language.

How It Works

When the `parse` method is applied to a class which inherits from `parsable`, as in the `tree` example above, it analyses the fields marked with the `Parse` attribute (fields not marked with the `Parse` attribute are ignored) and recursively analyses the types of these fields to construct a representation of the grammar of the input language to be parsed. If required, this grammar can be reconstructed in text form using the `parsable.unparse_grammar` method. Using this in the context of the `tree` example would produce the following text in a variant of Extended Backus Naur Form (EBNF).

```
<tree> ::= <identifier> "(" { <tree> } ")";
```

In this text ::= is read as "is defined as". Identifiers in pointed brackets represent strings in the language defined by the grammar or by the library, e.g. <identifier>. Strings in double quotes represent fixed symbols in the language. Enclosing a symbol in braces indicates that it can occur zero or more times. In this example { <tree> } will be parsed to produce a `List<tree>` value. A similar notation involves enclosing a symbol in square brackets, e.g. [<tree>]. This will be parsed to produce an `Optional<tree>` value.

Once an internal representation of the grammar has been constructed the parse method uses it to read the input from the `source_reader` constructing objects as it discovers their source representations and initialising the fields marked with the `Parse` attribute. It then returns the root object it has constructed. The parse operation can be used repeatedly to parse different texts using the representation of the grammar produced by the first call. A `parsable`.reset method exists to allow the grammar representation to be rebuilt for a different set of `parsable` types.

Alternatives

Some types may have a number of alternatives structures, e.g. a value in an expression might be specified by a number or a variable. In a grammatical description of this situation the definition of a value would therefore specify the two alternatives.

```
<value> ::= <number> | <variable>;
```

A class representing value objects can specify its alternative structures as subtypes.

```
class value: parsable
{
  class numeric: value
  {
    [Parse(1)] private int number;
  }

  class variable: value
  {
    [Parse(1)] private identifier id;
  }
}
```

A full example of programming language like expressions can be found in the examples folder for this project. This shows how an abstract syntax tree for non-trivial expressions can be produced by parsing such that conversional operator precedence is enforced.

Automatic Unparsing

The easiest way to unparse objects produced by the parser back into text is to use the `unparse_object` method.

```
source_reader  input      = new source_reader("root(a() b())");
tree           tree_value = parsable.parse<tree>(input);
source_builder output     = new source_builder();
parsable.unparse_object(tree_value, output);
string output_text = output.ToString();
```

This would set the `output_text` string to the following text.

```
root ( a ( ) b ( ) )
```

As the parser ignores white space, it cannot reproduce an exact copy of the original input text. However if the unparsed output contained no white space, identifiers, numbers etc would run together so that the output could not be fed back into the parser. For this reason the `unparse_object` method places padding spaces between all components of the grammar.

Custom Unparsing

If more control is required over the exact formatting of the unparsed output, e.g. the unparsed output is not required to be in the same format as the input, or there are specific formatting requirements, the library provides the following interface to support unparsing.

```
public interface unpasable
{
  void unparse(source_builder source);
}
```

The source_builder class provides an inverse of the source_reader class. It contains methods for writing strings, and numeric values to the output text and various methods for inserting new lines and controlling indentation. The tree class can be extended as follows to allow it to unparse tree objects back into their textual form.

```
class tree: parsable, unpasable
{
  [Parse(1)      ] private identifier  name;
  [Parse(2, "(")] private punctuation open;
  [Parse(3)      ] private List<tree>  subtrees;
  [Parse(4, ")")] private punctuation close;

  public void unparse(source_builder source)
  {
    source.write(name.spelling);
    source.write("(");
      source.separate(subtrees, (t) => t.unparse(source), " ");
    source.write(")");
  }
}
```

The separate method takes a list of objects, a function to apply to each element in the list and a string value. In between, applications of the function, it writes the string value to the source. In this case that causes all the trees in the list of subtrees, if any, to be output separated by spaces.

The following method can be used to produce a textual version of any tree based on the extended class.

```
string tree_string(tree t)
{
  var source = new source_builder();
  t.unparse(source);
  return source.ToString();
}
```

It is also possible to call the automatic unparse_object method from within custom unparse methods where minimal reformatting is required, e.g. to produce a sequence of automatically unparsed objects as tab separated or newline separated lists.

The Parse Attribute

```
[AttributeUsage(AttributeTargets.Field, Inherited = false, AllowMultiple = false)]
public class Parse: Attribute
{
  public Parse(int order)...
  public Parse(int order, string spelling)...
  public Parse(int order, params string[] alternatives)...
}
```

Field types which may have the Parse attribute applied to them are as follows:-

Field Type	Attribute	Grammar Notation	Matches in Source
Any parsable type t	Parse(order)	<t>	The textual representation of an object of the indicated type.
int	Parse(order)	<number>	Digits denoting an integer value.
double	Parse(order)	<real_number>	Characters denoting a real value.
string	Parse(order)	<string_constant>	Characters enclosed in double quotes.
char	Parse(order)	<char_constant>	A character enclosed in single quotes.
reserved_word	Parse(order, "word")	"word"	The word indicated by the second parameter of the attribute, which must be a sequence of characters starting with a letter followed by further letters, digits or underscore characters.
punctuation	Parse(order, "!=")	"!="	The characters indicated by the second parameter of the attribute, which must be a sequence of characters not including letters or digits.
identifier	Parse(order)	<identifier>	Any sequence of characters starting with a letter followed by further letters, digits or underscore characters that is not a reserved word. The actual characters will be assigned to the spelling property of the identifier object.
List<t>	Parse(order)	{ <t> }	Where t is some parsable type. The textual representations of zero or more objects of the

			type `t`.
`List<t>`	`Parse(order, "separator")`	`{ <t> "separator" }`	As above but the separator string specified in the attribute must be present between the texts of the objects of type `t`.
`Optional<t>`	`Parse(order)`	`[ <t> ]`	The text of an object of any parable type `t` or nothing. The Optional type has a bool property called Defined which indicates if a representation of an object of type `t` was present. It has a Value property of `t` to provide access to the details if the representation of an object of type `t` was present.
Any enumerated type	`Parse(order, "+", "-", "*", "div", "mod")`	`("+" | "-" | "*" | "div" | "mod")`	Any of the symbols or reserved words specified in the attribute. The cardinality of the enumerated type must match the number of symbols or words specified.
Any enumerated type	`Parse(order, "+", "-", "")`	`[ ("+" | "-") ]`	As above but the empty symbol indicates that the other words or symbols may be omitted, e.g. `enum unary {plus, minus, none}`

Methods of the Parsable Class

Static Method	Description
`public static void reset()`	Clears all the grammar details of any previously specified `parsable` type
`public static void define<t>(source_reader source)`	Analyses type `t` to determine the grammar required to parse it, but without any parsing. Subsequent uses of the parse method will not need to recreate the grammar details.
`public static t parse<t>(source_reader source) where t: parsable, new()`	If this has not already been done, analyses type `t` to determine the grammar required to parse it. It them attempts to parse the source and returns a corresponding instance of type `t`.
`public static void unparse_object<t>(t o, source_builder source) where t: parsable, new()`	Unparses an object of `parsable` type `t` back into text. The grammar for `t` must have already been defined.
`public static void unparse_grammar(source_builder source)`	After uses of either the `define` or `parse` methods, writes a formatted version of the current grammar to the source.

Instance Method	Description
`public virtual void parsed()`	Every time the parser creates a new `parsable` object and defines the fields decorated with the `Parse` attribute, it calls this method on the new object. This allows further configuration of independent fields not assigned by the parser.

Grammar Requirements

The parser employs a top-down, single symbol lookahead strategy without backtracking to build objects on-the-fly as it parses. As is always the case for top-down parsing without backtracking, the grammar derived from a set of `parsable` object types must satisfy two constraints for this to be possible.

1. No ambiguity. It must be possible to determine which alternative version of an object to parse based only on the next symbol available from the source, The following example is invalid as both alternatives start with the same symbol.

   ```
   <t> ::= "a" "b" | "a" "c"
   ```

   This example is invalid if both <t1> and <t2> start with the same symbols.

   ```
   <t> ::= <t1> "b" | <t2> "c"
   ```

   This one is slightly more subtle as, regardless of what <t1> starts with, it is optional and so the second alternative may also

start with an "a".

```
<t> ::= "a" <t> | [ <t1> ] "a"
```

2. No left recursive definitions, e.g. `<t> ::= <t> "a"` is invalid as is `<t> ::= <t1> "a"` if <t1> may start with a <t>. Right recursive definitions are no a problem. The following defines zero or more <a>s terminated by a fullstop.

```
<aa> ::= <a> <aa> | "."
```

During grammar construction the library will check these constrains and generate an exception if they are not satisfied.

Parsing Exceptions

The library will generate exceptions of the following type in the event of problems with the grammar, or when the source text being parsed does not match the grammar, or in the event of an invalid call to one of the `source_builder` methods.

```
public class parse_error: Exception
{
  public enum error_kind {grammar, parser, source_builder};

  public error_kind kind { get; private set; }

  public parse_error(string message, error_kind kind): base(message)
  {
    this.kind = kind;
  }
}
```

The `kind` property can be used in a catch statement to distinguish the possible causes of an exception. If the exception is raised during parsing, the `message` property will indicate which symbol, reserved word or whole object was expected but not found, e.g. “Missing !=” or “Missing <expression>”. The `line` and `column` properties of the `source_reader` can be used to determine where in the source the problem lies. Alternatively the `position` property can be used to obtain an absolute offset in the text. The following method demonstrates the use of these facilities.

```
  void parse_tree(FileStream file)
  {
    source_reader source = new source_reader(file);
    try
    {
      tree t = parsable.parse<tree>(source);
      process_tree(t);
    }
    catch (parse_error error)
    {
      MessageBox.Show(error.Message + " at line " + source.line + " column " + source.column);
    }
  }
```

Requirements

Where fields marked with the `Parse` attribute are of type `List<t>` or `Optional<t>`, `t` must be a `parsable` type. Therefore the “obvious” definition for a field containing a `List<int>`, where the individual `int` value are to be separated by commas, is not possible and will produce an error.

```
  [Parse(1, ",")] private List<int> integers;
```

This restriction can be overcome by defining a `number` class with a single `int` field and replacing the `List<int>` field with a `List<number>` field. A similar change will allow `Optional<int>` fields to be parsed. These remarks apply equally to other non-parsable simple types, i.e. `string`, `char`, `double` and any enumerated types.

Parsable class types can have constant members, fields, properties and methods that are not accessed by the parser. They can also have independent constructors, including parameterless constructors. The parser uses the default parameterless constructor to create objects if no alternative exists. If an explicit parameterless constructor is specified it must have `public` accessibility. Also note that the constructor will be executed before any fields marked with the `Parse` attribute are assigned. The `parsable` instance method called `parsed` can be overridden to perform any initialisation that would otherwise be performed by the constructor. The `parsed` method can also be use to perform on-the-fly semantic checking.

Parsable class types use nested subtypes to represent alternatives. These nested subtypes must be private. More extensive uses of sub-typing in conjunction with `parasable` objects is currently undefined.