Implementation Notes

Contents

0. Overview

The system operates in a number of phases.

1. Deduce the grammar of the language to be parsed from a set of `parsable` types, the equivalent of a set of EBNF productions, each with one or more alternatives and each alternative consisting of a sequence of components, e.g. punctuation, reserved words etc.
2. Build "first sets", the sets of lexemes which can begin strings described by each production, alternative and component. First sets are used to allow the parser to select alternatives without backtracking and to check that productions are not ambiguous. Definitions are also checked for left recursion at this stage.
3. Use the grammar to parse the input and build the objects it describes. Input to the parser is a stream of lexemes produced by the lexical analyser (`source_reader`) from the input text.

1. The `source_reader` class

As described above the purpose of the `source_reader` is to produce a stream of lexemes from the input text. The base class `lexeme` has a single field called `spelling`. All of the specific lexeme types are subtypes of this base class. They are documented in the following table.

Lexeme Type	Description
`punctuation`	Symbols not beginning with a letter or a digit, e.g. programming language operators, commas and semicolons etc. They can contain one or more characters, e.g. `:=` or `::=` or `!=`
`reserved_word`	Symbols beginning with a letter followed by further letters, digits or underscore characters, e.g. `begin end int32 a_word`
`number`	Symbols beginning with a digit possibly followed by further digits.
`real_number`	Symbols beginning with a digit possibly followed by further digits and containing a decimal point.
`identifier`	Symbols beginning with a letter followed by further letters, digits or underscore characters and which are not defined to be reserved words.
`string_literal`	Symbols beginning with a double quote followed by a character string and terminated by a double quote.
`char_literal`	Symbols beginning with a single quote followed by a single character and terminated by a single quote.

In addition to being read by the parser, these `lexeme` types are also used in conjunction with the `Parse` attribute to define fields of `parsable` types. For `number`, `real_number`, `identifier`, `string_literal` and `char_literal` lexemes the `source_reader` will assign the actual characters read from the input text to each lexeme's `spelling` field.

The `source_reader` constructors allow the input text to be defined by a `string`, a `FileStream` or a `StreamReader`. However, in the current implementation, regardless of which constructor is used, the input will be read and buffered in a single `string`.

The parser uses the following methods to interrogate the lexeme stream produced by the source reader.

```
internal bool symbol_is(Type kind)
```

Indicates if the current `lexeme` have the specified `Type`, e.g. `identifier`.

```
internal bool symbol_is(Type kind, string spelling)
```

Indicates if the current lexeme has the specified `Type` and `spelling`, e.g. `punctuation` with `spelling "!="`.

```
internal bool accept(Type kind)
```

If the current `lexeme` has the specified `Type` returns `true` and moves on to the next `lexeme` in the input. Otherwise returns `false`.

```
internal bool accept(Type kind, string spelling)
```

If the current `lexeme` has the specified `Type` and `spelling` returns `true` and moves on to the next `lexeme` in the input. Otherwise returns `false`.

```
internal void next_symbol()
```

Unconditionally moves on to the next lexeme in the input.

```
public void reset()
```

Moves to the first lexeme in the input. Not required after initial creation of the `source_reader`.

2. Grammar analysis

The details of the grammar represented by the `parsable` classes reachable from the initial call of `parsable.parse` or `parsable.define` are stored in a set of static fields belonging to the `parsable` class.

```
internal static Dictionary<string, production> productions;
internal static List<string>                   symbols;
internal static List<string>                   words;
internal static Stack<bool>                    checking_left_recursion_state;
internal static bool                           checking_left_recursion;
```

The dictionary of `productions` associates the name of each `parsable` class with a production (grammar definition). The name used is the short name of the type, e.g. `tree`. The two lists of strings are used to hold any punctuation symbols and reserved words discovered in field definitions marked with the `Parse` attribute. Before parsing begins, these two lists will be passed to the source_reader to allow it distinguish reserved words from identifiers and to know the set of valid punctuation symbols.

The `checking_left_recursion_state` and the `checking_left_recursion` fields are used by the algorithm used to check if productions are left recursive. This will be described later.

3. The grammar classes

As stated previously each production is associated with one or more alternatives and each alternative is represented by a sequence of components. The `production` class contains a list of alternatives described by the `alternative` class. The `alternative` class contains a list of components. The `component` base class is inherited by each of the specific component classes, e.g. `identifier_component`, `number_component` etc. All of these classes support the same basic set of functions and have fields for storing their first sets and indicating if they may match empty sequences in the input, e.g. an optional or iterated component. The classes representing productions and alternatives support similar operations but do not inherit from `component` because they have slightly different requirements.

```
internal class component
{
  public lexeme_set firsts;
  public bool       maybe_empty;

  protected component()..

  public virtual void analyse()...

  public virtual void parse(source_reader source)...

  public virtual void unparse_object(parsable o, FieldInfo field, source_builder source)...

  public virtual void unparse(source_builder source)...
}
```

The `analyse` method is used recursively starting from the `production` associated with the top level `parsable` type. A `production` applies it to all of its alternatives. An `alternative` applies it to all its components. A `component` applies it to any sub-components. At the end of this process the `maybe_empty` state, i.e. may specify zero-length sequences, and the `firsts`, i.e. the initial symbols of any component, alternative or production, of every element will be known, assuming the grammar is neither left-recursive nor ambiguous.

The `parse` method is also applied recursively starting from the top level type. A `production` chooses an `alternative` to parse

based on the current lexeme available from the source_reader and the firsts of each alternative. If no alternative matches the current lexeme, and no alternatives maybe_empty, a parse_error exception is thrown. Once an alternative has been selected it applies the parse method sequentially to each of its components until the last one is reached or the next lexeme does not match a component, in which case a parse_error exception is thrown.

The unparse_object method is overridden by classes that inherit from component so that they can add appropriate output to the source_builder. The FieldInfo parameter is supplied by the unparse_object method of the alternative class as it iterates through its components. It provides access to the Reflection.FieldInfo for the field associated with the component. In conjunction with the parsable object parameter o this allows the field value to be accessed.

The unparse method is applied recursively from the top level production by calling the parsable.unparse_grammar method to build a formatted text of the whole grammar.

The complete set of component classes is as follows.

Component Class	Description
terminal_component	A non-empty component (maybe_empty == false) that must match a specific string of characters.
identifier_component	A non-empty component that matches a characters representing an identifier.
number_component	A non-empty component that matches a characters representing a simple integer number.
real_component	A non-empty component that matches a characters representing a real number possibly containing a decimal point.
reserved_word_component	A non-empty component that must match a specific string of characters representing a reserved_word.
string_literal_component	A non-empty component that matches a string of characters between double quotes.
char_literal_component	A non-empty component that matches a single character between single quotes.
iterated_component<t>	A possibly empty component that matches zero or more strings as defined by the production associated with type t.
optional_component<t>	A possibly empty component that matches zero or one string as defined by the production associated with type t
alt_component	A component that matches one of a set of punctuation symbols or reserved_words that represent values of an enumerated type. If the type includes an empty option the component maybe_empty.
non_terminal_component	A component that matches a string defined by a particular production. If the production maybe_empty, then the component maybe_empty.

4. The alternative class

The following redacted source of the alternative class shows how an alternative determines its firsts and also part of the left-recursion checking algorithm. It also shows that each alternative is associated with a specific Type. This is because alternatives are defined by sub-types of a parsable type. Where sub-types are used this way, a production will only be associated with the parsable base type, but each alternative will be associated with a specific sub-type of that base type. This allows the correct objects to be created as alternatives are selected during parsing.

```
internal class alternative
{
  public List<component> components;
  public lexeme_set      firsts;
  public bool            maybe_empty;
  public Type            alternative_type;

  public alternative(List<component> components, Type alternative_type)...

  public void analyse(production parent)
  {
    maybe_empty                = true;
    parent.maybe_left_recursive = true;
```

```
    parsable.start_checking_for_left_recursion();
    foreach (var c in components)
    {
      c.analyse();
      if (maybe_empty)
      {
        firsts.insert(c);
        if (!c.maybe_empty)
        {
          maybe_empty                      = false;
          parent.maybe_left_recursive      = false;
          parsable.checking_left_recursion = false;
        }
      }
    }
    parent.maybe_left_recursive = false;
    parsable.end_checking_for_left_recursion();
  }

  public void parse(source_reader source)...

  public void unparse_object(parsable o, source_builder source)...

  public void unparse(source_builder source)...
}
```

Notice that the `analyse` method only adds components to its `firsts` until it gets to the first `component` that may not be empty. There is a static variable `checking_left_recursion` belonging to the `parsable` class (see above) that controls the checking of recursive references to productions. Before analysing any of its `components` the `analyse` method sets its `parent` production's state variable `parent.maybe_left_recursive` to `true`. If a reference to the `parent` is encountered while analysing the `components` of the `alternative`, and both the parent's `maybe_left_recursive` variable and the global `checking_left_recursion` variable are `true`, then the reference is left recursive and an exception will be thrown. The actual check will happen in the `analysis` method of the `non_terminal_component` class. The `parsable.start_checking_for_left_recursion` method is used to preserve the current state of the global `checking_left_recursion` variable on the global `checking_left_recursion_state` stack before setting it to `true`. This is necessary because recursive calls to `analyse` the alternatives of other productions may occur while analysing the `components` of the current `alternative`.

When the first non-empty `component` is encountered both the parent's `maybe_left_recursive` variable and the global `checking_left_recursion` variable are set to false so that any further recursive references to the parent production do not cause errors. After all the `components` of the `alternative` have been analysed, the `parsable.end_checking_for_left_recursion()` method is used to restore the original state of the global `checking_left_recursion` variable from the global `checking_left_recursion_state` stack. The parent's `maybe_left_recursive` variable is also set to `false` in case no non-empty `component` was encountered.

5. The `production` class

The following redacted version of the `production` class shows how a `production` analyses its `alternatives` and checks to determine if they are ambiguous. A new `production` is created and associated with each `parsable` type reachable from the root type. If the associated type has subtypes, the `add_alternative` method will be use to add their details to the `production`. Otherwise a single `alternative` is added based on the fields of the type.

The `analyse` method uses the `defined` field to prevent endless recursion when recursive references to the `production` occur within its definition. It analyses each of its `alternatives` in sequence. After each `alternative` has been analysed, its `firsts` will be defined and these are checked for any intersection with the cumulative `firsts` of the `production` itself. If there is an intersection, this indicates that the production's `alternatives` are ambiguous and an exception is thrown. If there is no intersection, the `firsts` of the `alternative` are added to the `firsts` of the `production`. In addition the `maybe_empty` status of the `production` is updated according to the `maybe_empty` status of the `alternative`.

```
internal class production
{
  public Type              t;
  public List<alternative> alternatives;
  public lexeme_set        firsts;
  public bool              maybe_empty;
  public bool              defined;
  public bool              maybe_left_recursive;
  public List<FieldInfo>   fields;

  public production(Type t)...

  public void add_alternative(List<component> components, Type alternative_type)...

  public void analyse()
  {
```

```
    if (!defined)
    {
      defined = true;
      foreach (var a in alternatives)
      {
        a.analyse(this);
        if (a.firsts.intersects(firsts))
          parsable.grammar_error("The alternatives of " + t.Name + " are ambigous");
        firsts.insert(a.firsts);
        maybe_empty = maybe_empty || a.maybe_empty;
      }
    }
  }

  public void parse(source_reader source)...

  public void unparse_object(parsable o, source_builder source)...

  public void unparse(source_builder source, int id_width)...
}
```

6. The `parsable` library public methods

The following redacted version of the `parsable` class shows the top-level methods used to define a grammar starting from the root of the `parsable` object hierarchy.

The `reset` method allows the static variables of the class to be reinitialised to allow multiple grammars to be processed.

The `define` method constructs the grammar objects described previously for some `parsable` type `t` and passes the symbol and reserved word definition discovered in the `parsable` object field attributes to the `source_reader`, which is then reset ready for parsing to begin. If `define` is not called explicitly, it will be called by the `parse` method.

```
public class parsable
{
  internal static Dictionary<string, production> productions;
  internal static List<string>                   symbols;
  internal static List<string>                   words;
  internal static Stack<bool>                    checking_left_recursion_state;
  internal static bool                           checking_left_recursion;
  internal static Stack<object>                  elements;

  public static void reset()...

  public static void define<t>(source_reader source) where t: parsable, new()
  {
    analyse(typeof(t));
    foreach (production p in parsable.productions.Values) p.firsts.flattern();
    source.define_symbols(symbols);
    foreach (string w in words) source.add_reserved_word(w);
    source.reset();
  }

  public static t parse<t>(source_reader source) where t: parsable, new()
  {
    define<t>(source);
    production root = productions[typeof(t).Name];
    root.parse(source);
    if (!source.symbol_is(typeof(punctuation), "<eof>"))
      error("Extra symbols before end of source");
    return (t)elements.Pop();
  }

  public static void unparse_object<t>(t o, source_builder source) where t: parsable, new()...

  public static void unparse_grammar(source_builder source)...

  public static void unparse_firsts(source_builder source)...

  public static void parsed()
  {
  }
}
```

The internal `analyse` method used by `define` is responsible for traversing the `parsable` type hierarchy, constructing one `production` for each type and adding alternatives and their components according to the fields of the `parsable` objects that are marked with the `Parse` attribute.

```
internal static void analyse(Type definition)
{
  if (!productions.Keys.Contains(definition.Name))
  {
    add_production(definition);
    if (has_parsable_subtypes(definition))
      analyse_parsable_subtypes(definition);
    else if (has_parsable_fields(definition, definition))
      anaylse_parsable_fields(definition, definition);
    else
      grammar_error("Class " + definition.Name + " must have parsable fields or parsable subtypes");
    productions[definition.Name].analyse();
  }
}
```

This method is not only called by `define` at the top-level, but also for every `parsable` type reachable from the root type. It checks to see if a `production` for the type already exists in order to avoid re-analysis of the type due to recursive references. If not it adds a new `production` object to the list of `productions`. If the type has nested `parsable` subtypes the `analyse_parsable_subtypes` method will iterate through them and add an `alternative`, with appropriate components, to the new `production` for each one. If there are no `parsable` subtypes the `anaylse_parsable_fields` method iterates through the fields of the type marked with the `Parse` attribute and adds a single `alternative`, with the appropriate components, to the new `production`. If the type has neither `parsable` subtypes nor fields marked with the `Parse` attribute, an exception is raised.

7. The parsing process

The `parsable` class's `parse` method initiates the parsing process by calling `define` as described above to build the representation of the grammar. It then selects the `production` corresponding to its generic type parameter and calls its `parse` method. If the production's `parse` method returns without throwing an exception due to a parsing error, the `parse` method checks that the `source_reader` has reached the end of the `source`. If parsing the root `production` has terminated successfully, the top of the `elements` stack will contain an object which can be returned as the result of the parse.

```
public void parse(source_reader source)
{
  int alternative_index = 0;
  int elements_parsed   = 0;
  Type element_type     = t;
  if (firsts.can_accept(source) || maybe_empty)
  {
    foreach (var a in alternatives)
    {
      if (a.firsts.can_accept(source) || a.maybe_empty)
      {
        int first_element_index = parsable.elements.Count;
        a.parse(source);
        int final_element_index = parsable.elements.Count;
        elements_parsed = final_element_index - first_element_index;
        element_type    = a.alternative_type;
        break;
      }
      alternative_index = alternative_index + 1;
    }
  }
  else parsable.error("Missing " + t.Name);
  push_result(alternative_index, elements_parsed, element_type);
}
```

If the `firsts` of the `production` include the current `lexeme` available from the `source_reader`, or the `production` maybe empty, the `parse` method iterates through the alternatives until it finds one that can start with the current `lexeme`. Only one `alternative` can satisfy this condition as the alternatives are known to be unambiguous by this stage. When the appropriate `alternative` is found, its `parse` method will be called. If the `alternative` can be parsed without throwing an exception because the source does not match the grammar, the `push_result` method is used to retrieve the values of the fields for this object from the `elements` stack. Using these it will create a new instance of the type corresponding to the `production` and assign values to its fields. The new object is then pushed onto the `elements` stack where it will be used to define other objects.

```
private void push_result(int alternative_index, int elements_parsed, Type element_type)
{
  if (t.Name != element_type.Name) return;
  ConstructorInfo constructor = element_type.GetConstructor(new Type[] { });
  if (constructor != null)
  {
    var parsed_object  = (parsable)constructor.Invoke(new object[] { });
    var field_elements = new Stack<object>();
    for (int i = 1; i <= elements_parsed; i = i + 1) field_elements.Push(parsable.elements.Pop());
    assign_fields(parsed_object, field_elements, alternative_index);
    parsed_object.parsed();
```

```
    parsable.elements.Push(parsed_object);
  }
  else
   parsable.grammar_error
   ("Class " + t.Name + " must have an accessible parameterless, default constuctor");
}
```

The guard at the beginning of `push_result` is required to handle the case where a subtype of the current `production` type has just been parsed. For example if some type `t` ha three subtypes `t1`, `t2` and `t3`. The grammar analyser will construct a production for `t` as show below.

```
  <t> ::= <t1> | <t2> | <t3>;
```

If one of the alternatives of <t> is parsed successfully, it will leave an object of the corresponding subtype on the elements stack. At the top-level this could be returned as the result of parsing the type `t`, or it could be assigned to a field of type `t` in another object. Creating an instance of the base type `t` would be superfluous.

The parse method of the alternative class is shown below.

```
public void parse(source_reader source)
{
  foreach (var c in components) c.parse(source);
}
```

This simply iterates through the components of the `alternative` calling the `parse` method of each one. If parsing any `component` fails to match the source, an exception will be thrown. Otherwise and appropriate set of objects representing the result of paring the `alternative` will be left on the `elements` stack.

Three representative examples of the `parse` methods for components are show below. The simplest case is for a `component` that matches a `lexeme` with a fixed spelling. In this case a `punctuation` symbol.

```
public override void parse(source_reader source)
{
  if (!source.accept(typeof(punctuation), spelling)) parsable.error("Missing " + spelling);
}
```

The `accept` method of `source_reader` either returns `true`, if the current `lexeme` is a `punctuation` symbol with the required `spelling`, and advances the `source` to the next `lexeme`, or it returns `false`. The value of `spelling` is a field of the `punctuation_component` setup by the grammar analyser from the second parameter of a `Parse` attribute associated with a field of type `punctuation` in a `parsable` class.

When a `component` matches some string without a fixed spelling it is necessary to return a value on the `elements` stack representing the details. Two examples are show below. The first is the `parse` method for the `identifier_component` class, in which case the `identifier lexeme` containing the actual spelling of the `identifier` is returned.

```
public override void parse(source_reader source)
{
  if (source.symbol_is(typeof(identifier)))
  {
    parsable.elements.Push(source.current);
    source.next_symbol();
  }
  else parsable.error("Missing identifier");
}
```

The second case shows the `parse` method of the `number_component` class. This parses the string recognised as a number into an `int` value and returns it on the `elements` stack.

```
public override void parse(source_reader source)
{
  if (source.symbol_is(typeof(number)))
  {
    parsable.elements.Push(int.Parse(((number)source.current).spelling));
    source.next_symbol();
  }
  else parsable.error("Missing number");
}
```

The final example of a `component parse` method shows the `non_terminal_component parse` method. In this case the `non_terminal_component` class has a field of type `production` called `definition`. The grammar analyser will have set this field to a reference to the appropriate `production` so that its `parse` method, as described above, can be called here This will result in an object of the type associated with the `production` being pushed onto the `elements` stack.

```
public override void parse(source_reader source)
{
  definition.parse(source);
}
```

8. The `lexeme_set` class

An important part of grammar analysis is the construction of first sets for productions, alternatives and components. First sets are represented by instances of the `lexeme_set` class. An outline of the `lexeme_set` class is shown below. The fields of the class store the various kinds of elements included in a set. A list of `symbols` maintains all of the spellings of all the `punctuation` symbols contained in a set. A list of `words` contains the spellings of all the `reserved_words`, and a set of boolean values indicate if a set contains a number, real number, identifier, string literal or character literal.

A set is created with these lists empty and all the boolean values set to `false`, i.e. the set as a whole is empty. Values are added via the `insert` methods. In the simplest case a `terminal_component` representing a `punctuation` symbol might be inserted into the set. This would cause its `spelling` to be added to the list of `symbols` if it were not already present. Similarly inserting a `reserved_word_component` would add its `spelling` to the list of words if it were not already present. Inserting an `identifier_component` would set the `contains_id` field to `true`, and so on. It is also possible to `insert` all the elements of another set. Again no elements are duplicated. This is required in the definition of a `production` where the `firsts` of all its alternatives must be inserted into `firsts` of the `production`.

Where the definition of a `production` starts with a `non_terminal_component`, i.e. a reference to another production, the firsts of the referenced `production` may not be defined until later in grammar analysis. To handle this case the `lexeme_set` class keeps a list of unique productions whose `firsts` are required to be included in a set.

The `intersects` method is used to check that the alternatives of a `production` are not ambiguous. As each `alternative` is defined within a `production` a check is made to see if its `firsts` intersect, i.e. contain common elements, with the cumulative `firsts` of the parent `production`.

During parsing the `can_accept` method is used to determine if the current `lexeme` available from the `source_reader` is contained in a `lexeme_set`.

The `flatten` method is used at the end of grammar analysis when the `firsts` of all productions are known. It inserts the `firsts` of each of the productions in the set's `productions` list and then clears the list. This allows the `can_accept` method to be executed while parsing using only the simple `symbols` and `words` lists and the boolean fields without having to check the `firsts` of productions recursively.

```
class lexeme_set
{
  private List<string>     symbols;
  private List<string>     words;
  private List<production> productions;
  private bool             contains_number;
  private bool             contains_real;
  private bool             contains_id;
  private bool             contains_string;
  private bool             contains_char;

  public lexeme_set()...

  public void insert(component c)...

  public void insert(lexeme_set s)...

  public void insert(production p)...

  public bool intersects(lexeme_set s)...

  public bool can_accept(source_reader source)...

  public void flattern()...

  public string unparse()...
}
```

9. The `source_builder` class

The `source_builder` class implements a wrapper for the C# `StringBuilder` class and provides facilities for generating multi-line text with controlled indentation. It can be used in conjunction with the following interface defined by the library. Classes which inherit from `parsable` can also implement the `unparsable` interface to allow the original text to be regenerated after parsing, or to generate an alternative textual representation from the input.

```
public interface unpasable
```

```
{
  void unparse(source_builder source);
}
```

By itself, the `source_builder` class allows textual representations of a number of standard types, e.g. `string`, `int` and `double`, to be added to the output text. Textual representations of other types can be added by first converting them to strings. Text can also be added from files. The generated text is extracted by using the `ToString` method.

```
public class source_builder
{
  public int  indentation_size {get; set;}
  public bool single_line      {get; set;}

  public source_builder()...

  public void reset()...

  public override string ToString()...

  public void write(string value)...

  public void write_file(string path)...

  public void write(bool value)...

  public void write(int value)...

  public void write(double value)...

  public void new_line()...

  public void new_line(int n)...

  public void indent()...

  public void outdent()...

  public void indent(int indentation_size)...

  public void outdent(int indentation_size)...

  public void write(int index, params string[] values)...

  public delegate void action<t>(t e);

  public void iterate<t>(List<t> ts, action<t> writer)...

  public void separate<t>(List<t> ts, action<t> writer, string separator)...

  public void separate_lines<t>(List<t> ts, action<t> writer, int lines)...
}
```

The `reset` method clears any text that has already been generated. It is called automatically when a new `source_builder` is created but may be called subsequently to start a new text. The `new_line` method allows one or more new lines to be inserted into the output. Nested indentation is controlled by the `indent` and `outdent` methods. The following example shows how this works. Each call of `indent` should be matched by a call of `outdent` and these calls should occur before the next call of `new_line` where the indentation change is required.

```
source.write("{");
  source.indent();
  source.new_line ();
  source.write("one();");
  source.new_line ();
  source.write("two();");
  source.new_line ();
  source.outdent();
source.write("}");
```

This example would cause the following text to be generated.

```
{
  one();
  two();
}
```

Spaces are used to generate indentation, and the number of spaces is controlled by the `indentation_size` property which by default is 2. There are variants of the `indent` and `outdent` methods that allow the `indentation_size` to be overridden. Matching

calls of these methods should generally specify the same indentation size, e.g. `indent(7); …; outdent(7);` Temporarily setting the `single_line` property to `true` allows any multi-line formatting to be suppressed. This includes both newlines and indentation.

There are a number of methods designed to work with sequences of values. The following example would add the (zero-based) nth parameter `string` to the text. The string ??? is added if `n` is out of range.

```
source.write(n, "plus", "minus", "times", "divide");
```

Alternatively the string may be extracted from an array. The following would add the same text as the previous example.

```
string[] operators = new string[]{"plus", "minus", "times", "divide"};
source.write(n, operators);
```

The `iterate`, `separate` and `separate_lines` methods work with lists in conjunction with the `action` delegate. By way of an example a list of objects that implement the `unparsable` interface might be unparsed as follows.

```
List<unpasable> elements = …
source.iterate(elements, (e) => e.unparse(source));
```

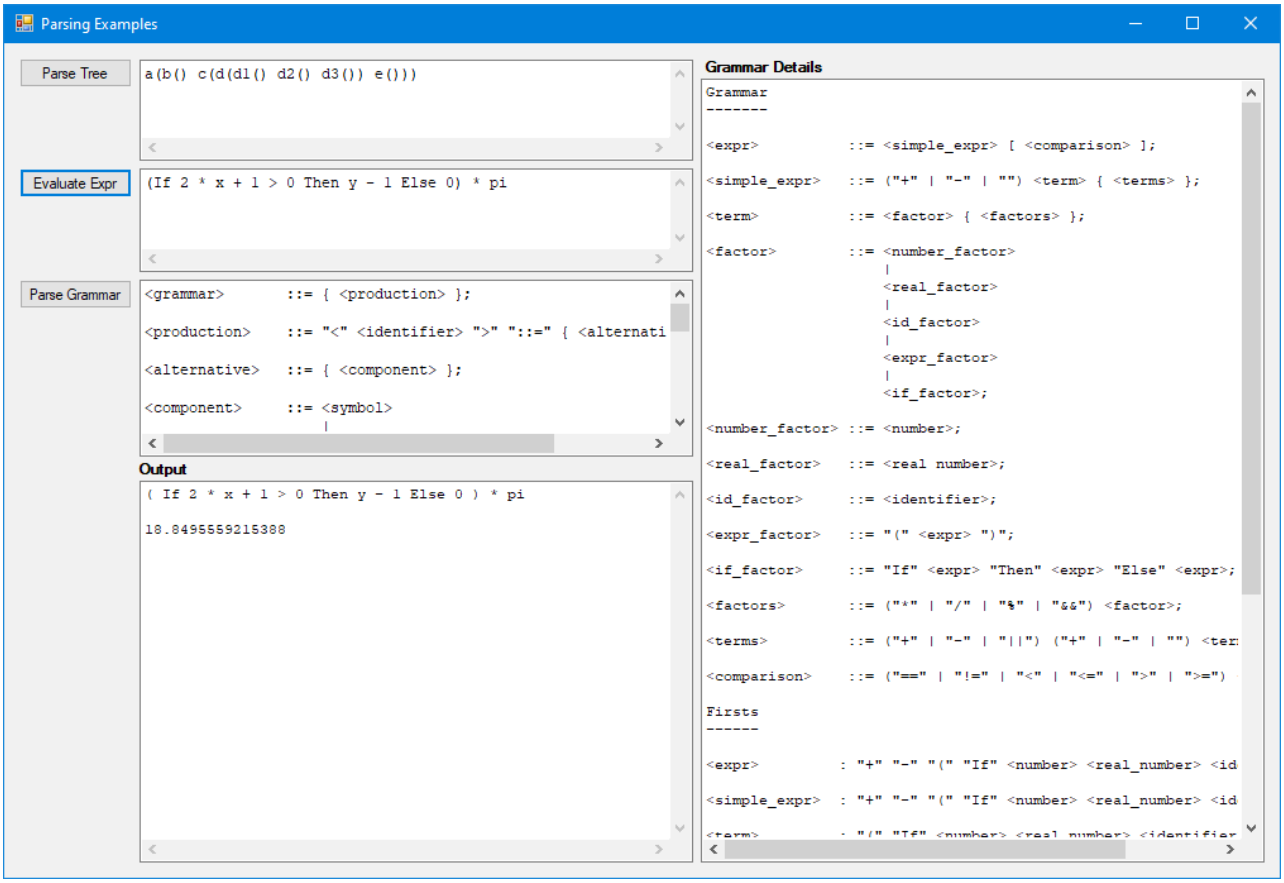
The previous example could have been accomplished using the `Linq` `foreach` method, however the following examples show how formatting can be inserted between `unparsable` elements. To insert new lines between, but not before or after, a list of elements the `separate_lines` method can be used.

```
List<unpasable> elements = …
source.separate_lines(elements, (e) => e.unparse(source));
```

To insert a specific separator between elements, the `separate` method can be used. The following example would add a bracketed, comma separated list of elements to the output. This assumes that each element `unparses` into text without new lines. If elements `unparse` into multi-line text, a more complex unparsing strategy, possibly involving indentation, would be required.

```
List<unpasable> elements = …
source.write("(");
source.separate(elements, (e) => e.unparse(source), ", ");
source.write(")");
```

10. Examples



The Examples folder contains a complete program which includes three different examples that can be run from the single form show below. The Parse Tree button loads the tree class described in the Reflexive introductory pdf document. It then parses the text in the

box to the right of the button. A textual representation of the grammar constructed from the tree class and the first set for trees are displayed in the Grammar Details box and the output from unparsing the resulting tree objects is displayed in the output box.

The Evaluate Expr button loads classes representing expressions similar to those found in many programming languages. These are organised such that the resulting grammar defines the precedence of the operators involved. It then parses an expression from the text in the box to its right. Each of the resulting objects supports an interface called `evaluator` that allows it to be evaluated. To simplify the example no type checking is performed, but this would be easy to add via another interface. The full grammar and firsts for each of the productions are displayed and the result of evaluating the expression is displayed in the output box.

The Parse Grammar button loads classes representing grammars similar, but not identical to, those used by the system itself. It then parses the text in the box to its right and unparses the result into the output box. This example demonstrates how the C# partial class mechanism can be used to completely separate the details of unparsing from the grammar data. The use of partial classes along with an interface provides a clean way to add functionality to a set of parsable classes. For example classes representing the abstract syntax tree of a programming language might be augmented by an interface and further partial class definitions to perform semantic checking, and yet another interface and further partial class definitions to perform code generation or direct execution of the AST.

The example program and the library itself are backward comparable (look it up!) to .Net Framework 3.5 and can be build with VS2008 or later.