IronMeta User Manual and Code Documentation

Version 1.4

Generated by Doxygen 1.5.9

Fri Jul 17 20:39:16 2009

Contents

Chapter 1

IronMeta User Manual

IronMeta provides a programming language and application for generating pattern matchers on arbitrary streams of objects. It is an implementation of Alessandro Warth's `OMeta` system for C# on .NET.

IronMeta is available under the terms of the BSD License.

- The IronMeta Language
- The IronMeta Grammar
- `The IronMeta SourceForge page`

1.1 Contributors

IronMeta 1.4 was written entirely by Gordon Tisher.

1.2 Features

- Although the most common use for IronMeta is to build parsers on streams of text for use in compiling or other text processing, IronMeta can generate pattern matchers (more accurately, transducers) for any input and output type. You can use C# syntax directly in grammar rules to specify objects to match.
- IronMeta-generated parsers can function with strict `Parsing Expression Grammar` semantics, or can function as fully-backtracking recursive-descent parsers.
- Generated parsers are implemented as C# partial classes, allowing you to keep ancillary code in a separate file from your grammar.

- Unrestricted use of C# in semantic conditions and match actions.
- Higher-order rules: you can pass rules as parameters, and then use them in a pattern.
- Pattern matching on rule arguments: you can apply different rule bodies depending on the number and types of parameters.
- Flexible variables: variables in an IronMeta rule may be used to:
  - get the input of an expression they are bound to.
  - get the result or result list of an expression they are bound to.
  - match a rule passed as a parameter.
  - pass a rule on to another rule.
- As an enhancement over the base OMeta, IronMeta allows unlimited left-recursion, using Warth, Douglass and Millstein's `algorithm`, for all rules, even within parameter matching.

Current limitations:

- Error reporting is currently quite rudimentary. The software internally collects data regarding all failed matches, but the main program currently only reports the last error that ocurred at the rightmost position in the input.
- Performance is quite slow, as not much optimization has been done to date.

1.3 Changelog

1.4: Bug fixes and sample

- Fixed a bug when passing rules in a base class to another rule.
- Added a sample that parses a data file and produces an XmlDocument.

1.3: Optimization pass

- IronMeta is now about an order of magnitude faster.
- Fixed a bug when handling C# string literals.

1.2: Bug fixes and miscellaneous enhancements.

- Added a simple interactive shell to the Calc program.

- Fixed a bug that caused redundant evaluation of involved rules.
- Fixed a bug with parsers that don't declare a base class.
- Added "::=" as alternative to "=" for rules.
- Changed condition operator to single question mark to conform to other OMetas.
- C# object literals must be surrounded by curly braces.

1.1:

- Can now handle IEnumerable literals (e.g. strings, for a matcher of characters).

1.0: Initial release

1.4 Get IronMeta

IronMeta is currently only available for Windows.

You can download a zip file containing IronMeta binaries at the `SourceForge site`.

1.5 Programming with IronMeta

In order to use IronMeta, run `IronMeta.exe` on your IronMeta file.

In order to use an IronMeta-generated parser in your C# program, do the following:

- Compile the C# file. If your parser is named `Foo`, the generated class will be named `Foo.FooMatcher`.
- Create an object of type `Foo.FooMatcher`.
- If you wish to use the parser as a fully-backtracking recursive-descent parser, set the `Matcher<TInput,TResult>.StrictPEG` property to `false`.
- Create an `IEnumerable<TInput>` with your input data.
- Call the function `Matcher<TInput,TResult>.Match()` or `Matcher<TInput,TResult>.AllMatches()`, with your input stream and the name of the top-level rule you wish to use. These return an object (or `IEnumerable`) of type `Matcher<TInput,TResult>.MatchResult`, which contains information about the possible results.

Note:

If your parser is using the default strict PEG mode, there will only be one result. If you have turned off strict PEG mode, there may be more than one possible parse, in which case you will need to call `AllMatches()` to access all the possibilities. Note that if you are timing the program, parsing is deferred until you read each match from the enumerable.

1.6 Reporting Bugs

If you come across a bug in IronMeta, please fill out a bug report at the `SourceForge bug tracker`.

1.7 Contributing to IronMeta

In order to build IronMeta, you must download the source code using a Subversion client. The following branches are available:

- Version 1.4: this branch contains the latest stable release of IronMeta:

  ```
  https://ironmeta.svn.sourceforge.net/svnroot/ironmeta/tags/1.4
  ```

- HEAD: Check out this branch to get the very latest code. This branch is where new development takes place. Note that code in this branch may not always work correctly!

  ```
  https://ironmeta.svn.sourceforge.net/svnroot/ironmeta/trunk
  ```

The top folder of the source code contains a Visual Studio 2008 solution file called `IronMeta.sln`. This includes three projects:

- `IronMeta.Matcher:` a library that contains the main packrat parsing functionality.
- `IronMeta:` the main IronMeta program.
- `Calc:` an example of the usual calculator program, made purposely more complicated so as to demonstrate some of IronMeta's advanced features.

Please send patches to the project admin listed at the `SourceForge website` or the IronMeta Development list, also available from SourceForge.

Chapter 2

The IronMeta Language

This section is an informal introduction to the features of the IronMeta language.

It uses the following IronMeta file named `Calc.ironmeta`, which is included in the IronMeta distribution.

The Calc grammar is much more complex than it needs to be, in order to demonstrate some of the advanced functionality of IronMeta.

```
// IronMeta Calculator Example

using System;
using System.Linq;

ironMeta Calc<char, int> : IronMeta.CharacterMatcher<int>
{
    Expression = Additive;

    Additive = Add | Sub | Multiplicative;

    DecimalDigit = .:c ?( c >= '0' && c <= '9' ) -> { return (int)c - '0'; };

    Add = BinaryOp(Additive, '+', Multiplicative) -> { return _IM_Result.Results.
      Aggregate((total, n) => total + n); };
    Sub = BinaryOp(Additive, '-', Multiplicative) -> { return _IM_Result.Results.
      Aggregate((total, n) => total - n); };

    Multiplicative = Multiply | Divide;
    Multiplicative = Number(DecimalDigit);

    Multiply = BinaryOp(Multiplicative, "*", Number, DecimalDigit)
            -> { return _IM_Result.Results.Aggregate((p, n) => p * n); };
    Divide = BinaryOp(Multiplicative, "/", Number, DecimalDigit)
            -> { return _IM_Result.Results.Aggregate((q, n) => q / n); };

    BinaryOp :first :op :second .?:type = first:a KW(op) second(type):b -> { retu
      rn new List<int> { a, b }; };

    Number :type = Digits(type):n Whitespace* -> { return n; };

    Digits :type = Digits(type):a type:b -> { return a*10 + b; };
    Digits :type = type;

    KW :str = str Whitespace*;
}
```

We will go through this example line by line to introduce the IronMeta language:

2.1 Comments

```
// IronMeta Calculator Example
```

You may include comments anywhere in the IronMeta file. They may also be in the C-style form:

```
/* C-Style Comment */
```

2.2 Preamble

```
using System;
using System.Linq;
```

You can include C# `using` statements at the beginning of an IronMeta file. IronMeta will automatically add `using` statements to its output to include the namespaces it needs.

2.3 Parser Declaration

```
ironMeta Calc<char, int> : IronMeta.CharacterMatcher<int>
```

An IronMeta parser always starts with the keyword `ironMeta`. Then comes the name of the parser (`Calc`, in this case), and the input and output types. The generated parser will take as input an IEnumerable of the input type, and return as output an IEnumerable of the output type.

In this case, the `Calc` parser will operate on a stream of `char` values, and output a stream of `int` values. We will define the parser so that the output list only includes one value.

Note:

You must always include the input and output types.

You may also optionally include a base class:

```
 : IronMeta.CharacterMatcher<int>
```

If you do not include a base class, your parser will inherit directly from `IronMeta.Matcher`. The `IronMeta.CharacterMatcher` class provides some specialized functionality for dealing with streams of characters.

2.4 Rules

```
    Expression = Additive;
```

An IronMeta rule consists of a name, an pattern for matching parameters, `"="`, a pattern for matching against the main input, and a terminating semicolon `";"` (for folks used to C#) or comma `","` (for folks used to OMeta):

```
IronMetaRule = "ironMeta" Pattern "=" Pattern (";" | ",");
```

In this case, the rule `Expression` has no parameters, and matches by calling another rule, `Additive`.

2.5 Matching Input

You can use the period `"."` to match any item of input, or you can use arbitrary C# expressions. The C# expressions may be a string literal, a character literal, or any other expression that is surrounded by curly braces:

```
MyPattern = 'a' "b" {3.14159} {new MyClass()};
```

IronMeta will use the standard C# `Equals()` method to match the items.

The pattern literal can also be an `IEnumerable` of the input type – IronMeta does not contain any special string-matching functionality; the fact that `string` implements `IEnumerable<char>` allows us to use C# strings directly.

This eliminates the need for the OMeta `token` function; just use a string literal, or if you are matching on something other than characters, use a list:

```
MyPattern = {new List<MyInputType>{ a, b, c }};
```

2.6 Sequence and Disjunction

```
Additive = Add | Sub | Multiplicative;
```

As is probably obvious from the other rules, you write a sequence of patterns by simply writing them one after the other, separated by whitespace.

To specify a choice between alternatives, separate them with `"|"`.

Note:

Unlike in other parser generator formalisms, separating expressions with a carriage return does NOT mean they are alternatives! You must always use the `"|"`.

2.7 Other Operators

You can modify the meaning of patterns with the following operators:

- `"?"` as a suffix will match zero or one times.

- "*" as a suffix will match zero or more times.
- "+" as a suffix will match one or more times.

In strict PEG mode these operators are all greedy – they will match as many times a possible and then return that result. If you want your parsers to be able to backtrack, you will need to disable strict PEG mode.

- "&" as a prefix will match an expression but NOT advance the match position. This allows for unlimited lookahead.
- "~" as a prefix will match if the expression does NOT match. It will not advance the match position.

2.8 Conditions and Actions

```
DecimalDigit = .:c ?( c >= '0' && c <= '9' ) -> { return (int)c - '0'; };
```

Here things get more interesting. This rule has only one expression, the period ".". This will match a single item of input. It is then bound to the variable `c` by means of the colon ":".

Note:

You can leave out the period if you are binding to a variable; that is, ":c" is equivalent to ".:c".

However, this rule will not actually match any character, because it contains a *condition*. A condition is written with "?" followed immediately by a C# expression in parentheses. The C# expression must evaluate to a `bool` value. Once the expression matches (in this case it will match anything), it is bound to the variable `c`, which is then available for use in your C# code.

The rule also contains an *action*. Actions are written with "->" followed by a C# block surrounded by curly braces. This block must contain a `return` statement that returns a value of the output type, or a `List<>` of the output type.

In this case, the variable `c` is explicitly cast to an `int` in order to force the variable to return its result, because otherwise C# would implicitly cast it to `char` because of the `'0'` in the expression.

Note:

If you do not provide an action for the expression, it will simply return the results of its patterns, as a list. Matching a single item will return `default(TResult)` by default, or you can pass a delegate or lambda function to the matcher when you create it that will convert values of the input type to the output type.

Be aware that an action only applies to the last expression in an OR expression. So the action in the following:

```
MyRule = One | Two | Three -> { my action };
```

will only run if the expression `Three` matches! If you want an action to apply on an OR, use parentheses:

```
MyRule = (One | Two | Three) -> { my action };
```

2.9 Variables

```
Digits :type = Digits(type):a type:b -> { return a*10 + b; };
```

Upon a successful match, variables will contain information about the results of the match of the expression they are bound to. In this example, because `a` & `b` are used in an expression containing an integer, they will automatically evaluate to the *results* of their expressions, because the result type of the Calc grammar is `int`.

IronMeta variables are very flexible. They contain implicit cast operators to:

- A single value of the input type: this will return the last item in the list of results of the expression that the variable is bound to.
- A single value of the output type.
- A `List<>` of the input type.
- A `List<>` of the output type.

If your input and output types are the same, the implicit cast operators will only return the *inputs*, and you will need to use the explicit variable properties:

- `c.Inputs` returns the list of inputs that the parse pattern matched.
- `c.Results` returns the list of results that resulted from the match.
- `c.StartIndex` returns the index in the input stream at which the pattern started matching.
- `c.NextIndex` returns the first index in the input stream *after* the pattern match ended.

You can also use variables in a pattern, in which case they will match whatever input they matched when they were bound. Or, if they were bound to a rule in a parameter pattern (see below), they will call that rule. You can even pass parameters to them.

2.9.1 Built-In Variables

IronMeta automatically defines some variables for use in your C# code:

- `_IM_Result`: bound to the entire expression that your condition or action applies to.
- `_IM_StartIndex`: an `int` that holds the index at which the match starts.
- `_IM_NextIndex`: an `int` that holds the index after the match ends.

2.10 Multiple Rule Bodies

```
Multiplicative = Multiply | Divide;
Multiplicative = Number(DecimalDigit);
```

You can have multiple rule bodies; their patterns will be combined in one overall OR when that rule is called.

2.11 Parameters

```
Add = BinaryOp(Additive, '+', Multiplicative) -> { return _IM_Result.Results.
  Aggregate((total, n) => total + n); };
```

This rule shows that you can pass parameters to a rule. You can pass literal match patterns, rule names, or variables.

```
BinaryOp :first :op :second .?:type = first:a KW(op) second(type):b -> { retu
  rn new List<int> { a, b }; };
```

This rule demonstrates how to match parameters. The parameter part of a rule is actually a matching pattern no different than that on the right-hand side of the `"="` ! Using this fact, plus the ability to specify multiple rules with the same name, you can write rules that match differently depending on the number and kind of parameters they are passed.

2.12 Rules as Arguments

```
Add = BinaryOp(Additive, '+', Multiplicative) -> { return _IM_Result.Results.
  Aggregate((total, n) => total + n); };
BinaryOp :first :op :second .?:type = first:a KW(op) second(type):b -> { retu
  rn new List<int> { a, b }; };
```

These rules show that you can pass rules as parameters to other rules. To match against them, just capture them in a variable in your parameter pattern, and then use the variable as an expression in your pattern. You can pass parameters as usual.

2.13 Pattern Matching Arguments

The example above shows that you can define different behavior depending on the parameters to a rule.

2.14 List Folding

```
KW :str = str Whitespace*;
```

If you look at the rules that call this rule (indirectly through the `BinaryOp` rule), you'll see that they pass both a single character and a string:

```
Sub = BinaryOp(Additive, '-', Multiplicative) -> { return _IM_Result.Results.
  Aggregate((total, n) => total - n); };
Divide = BinaryOp(Multiplicative, "/", Number, DecimalDigit) -> { return _IM_
  Result.Results.Aggregate((q, n) => q / n); };
```

When matching against variables captured in parameters, variables containing single items or variables containing lists will match correctly.

2.15 Rule Inheritance

```
Number :type = type+:digits Whitespace* -> { return digits.Results.Aggregate(
  0, (sum, n) => sum*10 + n); };
```

You may have noticed that there is no rule in the file called `Whitspace!` Because IronMeta matchers are C# classes, and their rules are methods, they can inherit rules from base classes. You can call these rules just like any others.

In this case, the `CharacterMatcher` class contains a rule called `Whitespace` which matches any whitespace character, so the parser compiles and runs just fine.

2.16 CharacterMatcher

The `CharacterMatcher` class provides the following rules:

- `Whitespace:` matches a whitespace character.

- `EOL:` matches end-of-line. This function is useful in that it records the positions of the ends of lines for use later.
- `EOF:` matched end-of-file.

It also provides the following utility functions:

- `_IM_GetText:` you can use this in a condition or action to get a string corresponding to the input matched by a pattern expression.
- `GetLineNumber():` you can use this after you have finished parsing to get the line number that a character at the given index is in.
- `GetLine():` you can use this after you have finished parsing to get a particular line of text from your input stream.

Chapter 3

The IronMeta Grammar

The following is the grammar used for parsing IronMeta files themselves, presented purely for interest.

```
//////////////////////////////////////////////////////////////////////
// Copyright (c) 2009, The IronMeta Project
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
//
//     * Redistributions of source code must retain the above
//       copyright notice, this list of conditions and the following
//       disclaimer.
//     * Redistributions in binary form must reproduce the above
//       copyright notice, this list of conditions and the following
//       disclaimer in the documentation and/or other materials
//       provided with the distribution.
//     * Neither the name of the IronMeta Project nor the names of its
//       contributors may be used to endorse or promote products
//       derived from this software without specific prior written
//       permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND  ANY EXPRESS OR  IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE  IMPLIED WARRANTIES OF  MERCHANTABILITY AND FITNESS
// FOR  A  PARTICULAR  PURPOSE  ARE DISCLAIMED. IN  NO EVENT SHALL THE
// COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
// INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
// BUT NOT  LIMITED TO, PROCUREMENT  OF SUBSTITUTE  GOODS  OR SERVICES;
// LOSS OF USE, DATA, OR  PROFITS; OR  BUSINESS  INTERRUPTION) HOWEVER
// CAUSED AND ON ANY THEORY OF  LIABILITY, WHETHER IN CONTRACT, STRICT
// LIABILITY, OR  TORT (INCLUDING NEGLIGENCE  OR OTHERWISE) ARISING IN
// ANY WAY OUT  OF THE  USE OF THIS SOFTWARE, EVEN  IF ADVISED  OF THE
// POSSIBILITY OF SUCH DAMAGE.
//
//////////////////////////////////////////////////////////////////////

ironMeta IronMeta<char, IronMeta.SyntaxNode> : IronMeta.CharacterMatcher<IronMeta
      .SyntaxNode>
{

    IronMetaFile = Spacing FilePreamble?:pre IronMetaParser*:parsers EOF
                -> { return new IronMetaFileNode(_IM_StartIndex, pre, parsers); }
      ;

    FilePreamble = ( UsingStatement )+;

    UsingStatement = KW("using") QualifiedIdentifier:id (KW(",") | KW(";"))
                -> { return new UsingStatementNode(_IM_StartIndex, _IM_GetText(id
      )); };

    IronMetaParser = KW("ironMeta") ParserDeclaration:decl ParserBody:body
                -> { return new ParserNode(_IM_StartIndex, decl, body); };

    ParserDeclaration = GenericIdentifier:name BaseClassDeclaration?:bc
                -> { return new ParserDeclarationNode(_IM_StartIndex, name, bc);
```

```
  };

BaseClassDeclaration = KW(":") GenericIdentifier:id
            -> { return id; };



ParserBody = KW("{") Rule*:rules KW("}")
            -> { return new ParserBodyNode(_IM_StartIndex, rules); };

Rule = KW("override")?:ovr Identifier:name Disjunction?:parms (KW("::=") | KW
  ("=")) Disjunction:body (KW(",") | KW(";"))
            -> {
                    bool isOverride = ovr.Results.Any();
                    SyntaxNode pNode = parms.Results.Any() ? (SyntaxNode)parm
  s : null;
                    return new RuleNode(_IM_StartIndex, isOverride, name, pNo
  de, body);
                };

Disjunction = Disjunction:a KW("|") ActionExpression:b
                -> { return new DisjunctionExpNode(_IM_StartIndex, a, b); }
            | ActionExpression;

ActionExpression = FailExpression;

ActionExpression = SequenceExpression:exp ((KW("->") | KW("=>")) &'{' CSharpC
  ode:action)
                -> { return new ActionExpNode(_IM_StartIndex, exp, action); }

            | SequenceExpression;

FailExpression = KW("!") (&'\"' CSharpCode:str)?
                -> { return new FailExpNode(_IM_StartIndex, str); };

SequenceExpression = SequenceExpression:a ConditionExpression:b
                -> { return new SequenceExpNode(_IM_StartIndex, a, b); }
            | ConditionExpression;

ConditionExpression = BoundTerm:exp ('?' &'(') CSharpCode:cond
                -> { return new ConditionExpNode(_IM_StartIndex, exp, cond);
  }
            | BoundTerm;

BoundTerm = PrefixedTerm:exp KW(":") Identifier:id
                -> { return new BoundExpNode(_IM_StartIndex, exp, id); }
            | KW(":") Identifier:id
                -> { return new BoundExpNode(_IM_StartIndex, new AnyExpNode(_
  IM_StartIndex), id); }
            | PrefixedTerm;


PrefixedTerm = AndTerm | NotTerm | PostfixedTerm;

AndTerm = KW("&") PrefixedTerm:exp
        -> { return new PrefixedExpNode(_IM_StartIndex, exp, "LOOK"); };
```

```
NotTerm = KW("~") PrefixedTerm:exp
        -> { return new PrefixedExpNode(_IM_StartIndex, exp, "NOT"); };


PostfixedTerm = StarTerm | PlusTerm | QuestionTerm | Term;

StarTerm = PostfixedTerm:exp KW("*")
        -> { return new PostfixedExpNode(_IM_StartIndex, exp, "STAR"); };

PlusTerm = PostfixedTerm:exp KW("+")
        -> { return new PostfixedExpNode(_IM_StartIndex, exp, "PLUS"); };

QuestionTerm = PostfixedTerm:exp ('?' ~'(' Spacing)
        -> { return new PostfixedExpNode(_IM_StartIndex, exp, "QUES"); };


Term = ParenTerm | AnyTerm | RuleCall | CallOrVar | Literal;

ParenTerm = KW("(") Disjunction:exp KW(")")
        -> { return exp; };

AnyTerm = KW(".")
        -> { return new AnyExpNode(_IM_StartIndex); };

RuleCall = QualifiedIdentifier:name KW("(") ParameterList?:p KW(")")
        -> { return new RuleCallExpNode(_IM_StartIndex, _IM_GetText(name), p)
  ; };

ParameterList = Parameter (KW(",") Parameter)*
        -> { return _IM_Result.Results.Where(child => child is CallOrVarExpNo
  de || child is LiteralExpNode); };

Parameter = CallOrVar | Literal;

CallOrVar = QualifiedIdentifier
        -> { return new CallOrVarExpNode(_IM_StartIndex, _IM_Result); };

Literal = &('\"' | '\'' | '{') CSharpCode
        -> { return new LiteralExpNode(_IM_StartIndex, _IM_Result); };


CSharpCode = CSharpCodeItem:code Spacing
        -> { return new CSharpNode(_IM_StartIndex, _IM_GetText(code)); };

CSharpCodeItem = '{' ((~'}') (CSharpCodeItem | Comment | EOL | .))* '}'
               | '(' ((~')') (CSharpCodeItem | Comment | EOL | .))* ')'
               | '\"' ( ('\x5c' '\x5c') | ('\x5c' '\"') | ((~'\"') (EOL | .))
   )* '\"'
               | '\'' ( ('\x5c' '\x5c') | ('\x5c' '\'') | ((~'\'') (EOL | .))
   )* '\'';


GenericIdentifier = QualifiedIdentifier:id (KW("<") (GenericIdentifier (KW(",
  ") GenericIdentifier)*):p KW(">"))?
        -> {
                List<string> pl = p.Results
                                    .Where(node => node is IdentifierNode)
```

```
                                  .Select(node => node.Text).ToList();
                    IdentifierNode idn = (IdentifierNode)id;
                    return new IdentifierNode(_IM_StartIndex, idn.Name, idn.Quali
      fiers, pl);
                };

    QualifiedIdentifier = (Identifier KW("."))*:quals Identifier:name
            -> {
                    var ql = quals.Results.Where(node => node is IdentifierNode).
      Select(node => node.Text).ToList();
                    return new IdentifierNode(_IM_StartIndex, ((IdentifierNode)na
      me).Name, ql, null);
                };

    Identifier = (. ?(_IM_Result == '_' || System.Char.IsLetter(_IM_Result)))
                 (. ?(_IM_Result == '_' || System.Char.IsLetterOrDigit(_IM_Result
      )))*
                 Spacing
            -> { return new IdentifierNode(_IM_StartIndex, _IM_GetText(_IM_Result
      ).Trim()); };


    Spacing = (Comment | Whitespace)*:nodes
            -> { return new SpacingNode(_IM_StartIndex, nodes); };

    Comment = (  '/' '/' (~('\r'|'\n') .)* (EOL|EOF)
               | '/' '*' (~('*' '/') (EOL | .))* '*' '/' )
            -> { return new CommentNode(_IM_StartIndex, _IM_GetText(_IM_Result));
       };


    KW .*:kw = kw:str Spacing
            -> { return new KeywordNode(_IM_StartIndex, _IM_GetText(str)); };


    // EOL needs to be first, as otherwise it won't add the position to the list

    override Whitespace = EOL | . ?(System.Char.IsWhiteSpace(_IM_Result))
                                -> { return new TokenNode(_IM_StartIndex, TokenNo
      de.TokenType.WHITESPACE); };


    override EOL = ('\r' '\n' | '\r' ~'\n' | '\n')
            -> {
                    _IM_LineBeginPositions.Add(_IM_NextIndex);
                    return new TokenNode(_IM_StartIndex, TokenNode.TokenType.EOL)
      ;
                };

    override EOF = ~.
            -> { _IM_LineBeginPositions.Add(_IM_StartIndex); return new TokenNode
      (_IM_StartIndex, TokenNode.TokenType.EOF); };

}
```

Chapter 4

BSD License

Chapter 5

Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 6

Class Documentation

6.1 Matcher< TInput, TResult > Class Template Reference

Base class for IronMeta matchers.

Classes

- class **ActionCombinatorList**
- class **AndCombinator**
- class **AnyCombinator**
- class **ArgsCombinator**
- class **CallItemCombinator**
- class **Combinator**
- class **ConditionCombinator**
- class **EmptyCombinator**
- class **FailCombinator**
- class **LiteralCombinator**
- class **LookCombinator**
- class **MatchItem**
- class **MatchItemStream**
- class MatchResult

  Holds a match result from applying a parser to an input stream.

- class **Memo**
- class **NotCombinator**
- class **OrCombinator**

- class **RefCombinator**
- class **StarCombinator**
- class **VarCombinator**

Public Member Functions

- IEnumerable< MatchResult > AllMatches (IEnumerable< TInput > inputStream, string productionName)

    Returns the results of matching a given production against a stream of input.

- MatchResult Match (IEnumerable< TInput > inputStream, string productionName)

    Returns the result of matching a given parser production against a stream of input.

Static Public Member Functions

- static void **WriteIndent** (int index, int indent, int iter, string format, params object[ ] args)

Protected Member Functions

- Production **FindProduction** (string name)
- Matcher (Func< TInput, TResult > convertItem, bool strictPEG)

    Construct a new Matcher object.

- delegate IEnumerable< MatchItem > **Production** (int indent, IEnumerable< MatchItem > _inputs, int _index, IEnumerable< MatchItem > _args, Memo _memo)

Static Protected Member Functions

- static Combinator **_ACTION** (Combinator a, Func< MatchItem, IEnumerable< TResult >> action)
- static Combinator **_ACTION** (Combinator a, Func< MatchItem, TResult > action)
- static Combinator **_AND** (params Combinator[ ] combinators)
- static Combinator **_ANY** ()
- static Combinator **_ARGS** (Combinator arg_pattern, IEnumerable< MatchItem > actual_args, Combinator body_pattern)
- static Combinator **_CALL** (MatchItem v, IEnumerable< MatchItem > actual_args)

- static Combinator **_CALL** (Production p)
- static Combinator **_CALL** (Production p, IEnumerable< MatchItem > actual_-args)
- static Combinator **_CONDITION** (Combinator a, Func< MatchItem, bool > condition)
- static Combinator **_EMPTY** ()
- static Combinator **_FAIL** (string message)
- static Combinator **_FAIL** ()
- static Combinator **_LITERAL** (IEnumerable< TInput > items)
- static Combinator **_LITERAL** (TInput item)
- static Combinator **_LOOK** (Combinator a)
- static Combinator **_NOT** (Combinator a)
- static Combinator **_OR** (params Combinator[ ] combinators)
- static Combinator **_PLUS** (Combinator a)
- static Combinator **_QUES** (Combinator a)
- static Combinator **_REF** (MatchItem v)
- static Combinator **_REF** (MatchItem v, Matcher< TInput, TResult > matcher)
- static Combinator **_REF** (MatchItem v, string name)
- static Combinator **_REF** (MatchItem v, string name, Matcher< TInput, TResult > matcher)
- static Combinator **_STAR** (Combinator a)
- static Combinator **_VAR** (Combinator a, MatchItem v)

Protected Attributes

- List< Combinator > **CachedCombinators** = new List<Combinator>()
- Func< TInput, TResult > **CONV**

Properties

- bool StrictPEG `[get, set]`

  Determines whether or not the matcher will use strict Parsing Expression Grammar semantics.

6.1.1 Detailed Description

template<TInput, TResult> class IronMeta::Matcher< TInput, TResult >

Base class for IronMeta matchers.

Template Parameters:

TInput The type of input to the matcher.

TResult The type each match will result in.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 Matcher (Func< TInput, TResult > *convertItem*, bool *strictPEG*) `[protected]`

Construct a new Matcher object.

Parameters:

convertItem A delegate that holds a function that converts from the input type to the output type.

strictPEG Whether or not to use strict Parsing Expression Grammar semantics.

6.1.3 Member Function Documentation

6.1.3.1 IEnumerable<MatchResult> AllMatches (IEnumerable< TInput > *inputStream*, string *productionName*)

Returns the results of matching a given production against a stream of input.

If the parser is using strict PEG matching, there will only be one result. If not, there may be a set of results.

Parameters:

inputStream The input to the parser.

productionName The production (i.e. rule) of the parser to use.

6.1.3.2 MatchResult Match (IEnumerable< TInput > *inputStream*, string *productionName*)

Returns the result of matching a given parser production against a stream of input.

If the parser is using strict PEG matching, there will only be one result. If not, use AllMatches() to get all the possible parses.

Parameters:

inputStream The input to the parser.

productionName The production (i.e. rule) of the parser to use.

The documentation for this class was generated from the following file:

- Matcher.cs

6.2 MatchResult Class Reference

Holds a match result from applying a parser to an input stream.

Properties

- string Error `[get]`
  The error that caused the match to fail, if it failed.
- int ErrorIndex `[get]`
  The index in the input stream at which the error occurred.
- int NextIndex `[get]`
  The index in the input stream after the last item matched.
- TResult Result `[get]`
  The last result in the result list.
- IEnumerable< TResult > Results `[get]`
  The result of the match; possibly as a list.
- int StartIndex `[get]`
  The index in the input stream at which the match started (usually 0).
- bool Success `[get]`
  Indicates whether or not the match succeeded.

6.2.1 Detailed Description

template<TInput, TResult> class IronMeta::Matcher< TInput, TResult >::MatchResult

Holds a match result from applying a parser to an input stream.

6.2.2 Property Documentation

6.2.2.1 TResult Result `[get]`

The last result in the result list.

Will throw if the match did not succeed.

6.2.2.2 IEnumerable<TResult> Results [get]

The result of the match; possibly as a list.

Will throw if the match did not succeed.

The documentation for this class was generated from the following file:

- Matcher.cs

Index