組込み RTOS 向けアプリケーション開発支援ツール
TLV（トレース ログ ヴィジュアライザー）
フェーズ 5 リファクタリング仕様書

2010 年 6 月 1 日

改訂履歴

版番	日付	更新内容	更新者
1.0	10/6/01	新規作成	市原大輔

目次

1 はじめに

1.1 本書の目的

本書の目的は、文部科学省先導的 IT スペシャリスト育成推進プログラム「OJL による最先端技術適応能力を持つ IT 人材育成拠点の形成」プロジェクトにおける、OJL 科目ソフトウェア工学実践研究の研究テーマである「組込み RTOS 向けアプリケーション開発支援ツールの開発」に対して、その開発するソフトウェアに対する設計を記述することである。

本書は特に、フェーズ 5 におけるリファクタリング作業に関する記述を行う。

1.2 本書の適用範囲

本書は、組込み MPRTOS 向けアプリケーション開発支援ツールの開発プロジェクト（以下本プロジェクト）のフェーズ 5 におけるリファクタリング作業に関する記述を行う。

1.3 用語の定義/略語の説明

表 1　用語定義

用語・略語	定義・説明
TLV	Trace Log Visualizer
MPRTOS	マルチプロセッサ対応リアルタイムオペレーティングシステム
トレースログファイル	RTOS のトレースログ機能を用いて出力したトレースログや、シミュレータなどが出力するトレースログをファイルにしたもの
標準形式トレースログファイル	本ソフトウェアが扱うことの出来る形式をもつトレースログファイル。各種トレースログファイルは、この共通形式トレースログファイルに変換することにより本ソフトウェアで扱うことが出来るようになる。
変換ルール	トレースログファイルを標準形式トレースログファイルに変換する際に用いられるルール。
可視化ルール	標準形式トレースログファイルを可視化する際に用いられるルール。
TLV ファイル	本ソフトウェアが中間形式として用いるファイル。前述の標準形式トレースログファイルは、この TLV ファイルの一部である。
EBNF	Extended Backus Naur Form の略。
Parsec	関数型言語 Haskell のパーサ・コンビネータライブラリ。

1.4 概要

本書では、組込み MPRTOS 向けアプリケーション開発支援ツールのソフトウェアの仕様を記述する。本書は特に、フェーズ 5 におけるリファクタリング作業に関する記述を行う。

第 I 部
リファクタリング概要

2 実施理由

現状の TraceLog クラス (クラス概要は 3 参照) では、標準形式トレースログのパースにおいて、Listing1 のような複雑な正規表現を計 8 回適用している。これは、Object などの要素 1 つにつき 1 行の正規表現で取得するためである。こうすることで、Time や Value 等の有無に関わらず[*1]パースできるようにしている。しかし、このような複雑な正規表現は、可読性が低く保守が難しいため、リファクタリングを実施する。

リファクタリング後の TLV は、標準形式トレースログのパースを専用のパーサに任せる。パーサは、TLV 変換ルール・可視化ルールマニュアル (rule-maual.pdf) の「2.1.2 標準形式トレースログの定義」にある EBNF のようにコードを記述することで、構文を直感的に理解できるようになる。コード例を Listing2 に示す。

また、重複したコードおよび生成規則変更時の変更箇所も減らすことができるため、保守性も向上する。例えば、Time を囲む記号"["，"]"を"<"，">"にするとき、従来の正規表現を用いた方法では、8 行すべての正規表現を変更する必要があるが、リファクタリング後は 1 箇所で済むようになる。生成規則を追加する場合でも、従来の手法では 8 行の正規表現の中から変更箇所を探さねばならないが、リファクタリング後は、EBNF のように記述されているため、変更箇所が明確であるので見落としを減らせる。しかしながら、このような構文の変更は稀であるため、一番のメリットは構文を直感的に理解できるようになることである。

Listing 1 リファクタリング前のコード例

```
1 m = Regex.Match(_log, @"^(\[[^\]]+\])?(?<objectType>[^\[\]\(\)\.]+)\([^\)]+\)(\.[^\s]+)?$");
2         if (m.Success)
3                 ObjectType = m.Groups["objectType"].Value;
4         HasObjectType = m.Success;
```

Listing 2 リファクタリング後に表現したいコード例

```
1 var object_ =
2         ObjectTypeName().Char('(').AttributeCondition().Char(')')
3         .OR().
4         ObjectName();
```

3 対象

リファクタリング対象は、TraceLog クラス、特にそのコンストラクタである。この TraceLog クラスは、標準形式トレースログを構成するトークンを保持し、標準形式トレースログを用いた処理 (可視化表示など) を容易にする。

現在、標準形式トレースログのパースに正規表現を用いている箇所をパーサを用いるように、TraceLog コンストラクタを変更する。

[*1] 可視化ルールファイルには、Time がない、Value がない、AttributeName がない等といった不完全な標準形式トレースログが記述されている。

4　変更内容

4.1　クラス構成

正規表現によるパース (Regex クラスおよび Match クラス) を廃止し、標準形式トレースログ用のパーサを作成して、TraceLog クラスはそのパーサへパース処理を委譲する。これにより、現在はパースの実装が TraceLog クラスに直書きされているためパースの実装方法を変更するのが困難であるのが、TraceLog クラスとパースの実装を分離することでパーサの交換が容易になり、パーサの実装方法変更が容易になる。

今回の変更に伴う影響範囲は、TraceLog クラス以外に存在しない。リファクタリング前後のクラス図を図 1,2 に示す。

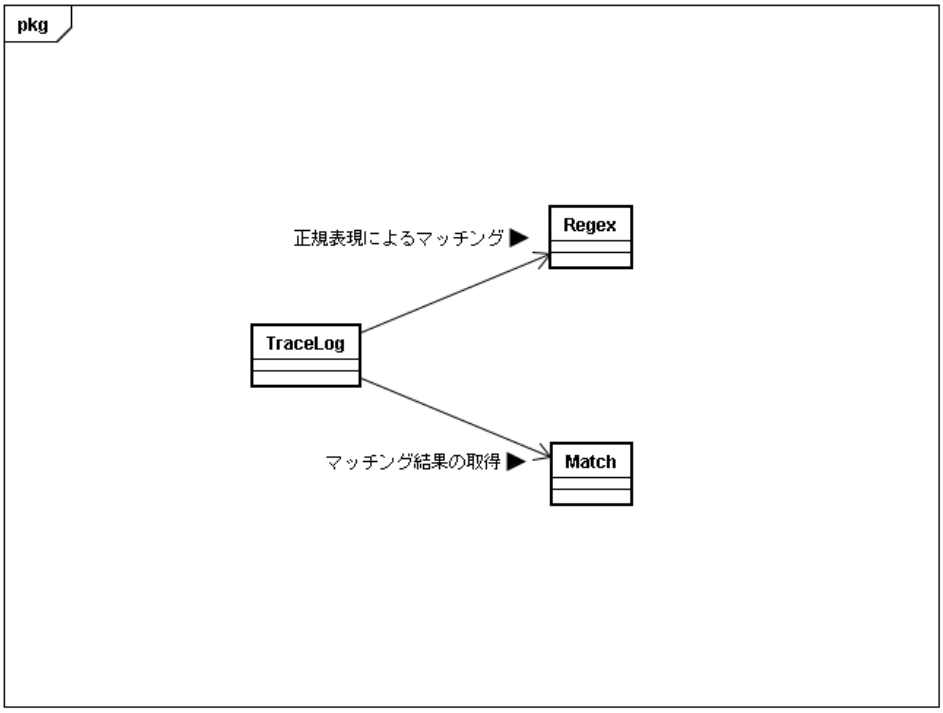

図 1　リファクタリング前のクラス構成

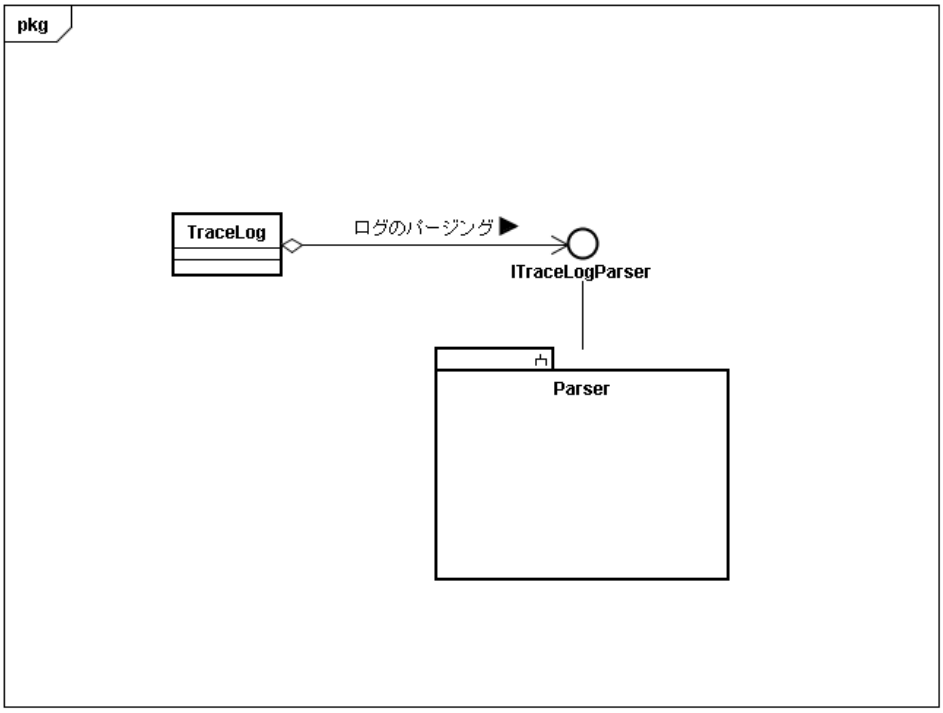

図 2　リファクタリング後のクラス構成

4.2　パーサの構成

パーサの実装は、コードで EBNF を表現し、直感的に理解できるようにする。

例として、Haskell のパーサコンビネータライブラリ Parsec を参考にしたパーサ [1] が挙げられる。このコードを使用するか、自作など他の実装方法を用いるかは、デバッグの容易さ、実装のしやすさ、可読性、実行速度などの要素により決定する。

第 II 部
詳細仕様

本章は、Phase5 で行ったリファクタリングにより変更・追加された箇所、および、リファクタリング後のシーケンスを説明し、今回採用しなかった手法との比較や今回の実装の採用理由を記述する。

5 コード設計

ここでは、リファクタリングの目標であった「EBNF を表現し、直感的に理解できる」コードの紹介と、小節にて実装に用いた手法・概念および主要メソッドの説明をする。

まず、「EBNF を表現」したコードの例を Listing3 に示す。

Listing 3 「コードで EBNF を表現」したコード例 (最終更新日現在)

```
1 var line = Char('[').Time().Char(']').Event();
2 ...
3 var object_ =
4         ObjectTypeName().Char('(').AttributeCondition().Char(')')
5         .OR().
6         ObjectName();
7 ...
8 var typeName = Many1(() => AnyCharOtherThan('(', ')', '.'));
```

これは、次の EBNF をパースすることを示している (rule-manual.pdf の 2. 1.2 参照)。

Listing 4 Listing3 が表す実際の EBNF

```
1 TraceLogLine = "[", Time, "]", Event;
2
3 Resource = ResourceTypeName, "(", AttributeCondition, ")"
4          | ResourceName;
5
6 ResourceTypeName = /[0-9a-Z_]+/;
```

このように、Listing3 は EBNF を表現しており、正規表現によるパージングより可読性が向上している。Listing3 は完全なものではなく、実際には Listing5 のような解析メソッド内に記述され、それを組み合わせることで Listing3 を実現している。

Listing 5 解析メソッド例

```
1 public ITraceLogParser ObjectTypeName()
2 {
3         Begin();
4
5         var typeName = Many1(() => AnyCharOtherThan('(', ')', '.'));
6
7         typeName.ObjectTypeValue = Result();
8         return (ITraceLogParser)typeName.End();
9 }
```

5.1 利用した概念・手法

5.1.1 再帰下降パーサ

今回、実装したパーサは、スタックを利用した再帰下降パーサであり、バックトラッキングを用いている。これは、EBNF のようにトップダウンに記述されているものを表現するのに適しているが、バックトラッキングによる処理効率低下が懸念される。

5.1.2 パーサ・コンビネータ

パーサ・コンビネータとは、パーサとパーサを組み合わせることのできるコンビネータを指す。具体的には、Many メソッド、Many1 メソッド、OR メソッドが挙げられる。これにより、EBNF の表現や LL(k) 文法のパージングを可能にしている。しかしながら、LL(k) 文法を扱うため生成規則から左再帰性の除去が必要である。

5.1.3 NullObject パターン

インタフェースを実装しているが、何もしないクラスを用いるデザインパターン。多態性を利用したもので、Null かどうかを判別するコードを排除でき、処理の流れが明確になる。今回、これを用いることで EBNF のような記述を実現している。解析メソッドは、パーサクラスもしくはパーサクラス用の NullObject を返すことで次の解析メソッドを実行するかを決定する。すると、if 分をはさまずにメソッドチェーンのみで生成規則を表現できるようになる。

他に、Null による問題 (意図しない Null 参照による例外発生など) を防止できるというメリットもある。

5.2 基本メソッドの説明

5.2.1 Begin メソッド

解析メソッドの処理で最初に実行されるメソッド。

現在は、スタックへのパース結果およびポインタ[*2]の保存領域確保を行っている。

5.2.2 End メソッド

解析メソッドの処理で最後に実行されるメソッド。基本的に、パースを実行、つまり、生成規則を表すメソッドチェーンを実行して得られた戻り値 (パーサオブジェクトまたは NullObject) から呼ばれる。

現在は、スタックのポップ、直前のスタックにあるパース結果との結合、NullObject のステータス初期化を行っている。パーサクラスか NullObject クラスかで動作が変わる。

5.2.3 Result メソッド

パースを実行して得られた文字列を返すメソッド。

5.2.4 OR メソッド

EBNF の記号"|"を表すメソッド。生成規則を表すメソッドチェーンで使用する。

[*2] パージング中の文字列の、解析の現在位置 (文字) を示すもの。

このメソッドより前の生成規則に当てはまらなかった場合、このメソッド内でバックトラッキングして、次の生成規則の適用を試みる。

5.2.5 Many メソッド

EBNF の記号"*"を表すメソッド。生成規則を表すメソッドチェーンで使用する。
引数に与えられた解析メソッドを 0 回以上適用させる。

5.2.6 Many1 メソッド

EBNF の記号"+"を表すメソッド。生成規則を表すメソッドチェーンで使用する。
引数に与えられた解析メソッドを 1 回以上適用させる。

6 クラス設計

リファクタリング後のクラス図を図 3 に示す。TraceLog クラス以外のクラス群が、前章の図 2 にあるサブシステムに相当する。

パーサは、基本的な機能を備えたクラスとパース対象に特化したクラスにわけることで、基本的な機能を備えたクラスの再利用ができるようになっている。そのため、それに該当する Parser クラスおよび NullObjectForParser クラスを継承する派生クラスを定義し、ITraceLogParser クラスのような特化したインタフェースを定義してそれぞれの派生クラスが実装することで、様々なパーサが作成可能である。

次に、図 3 にある各クラスの簡単な説明をする。

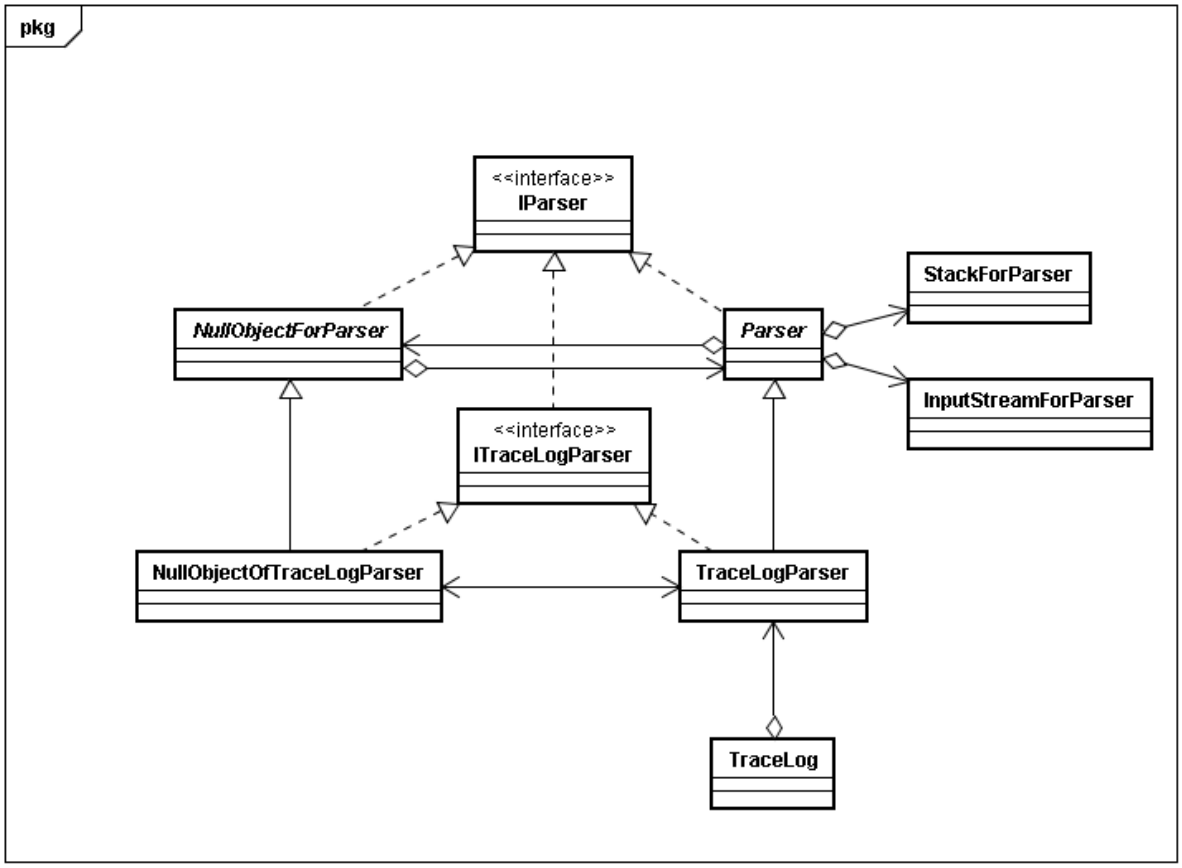

図 3　リファクタリング後のクラス構成

6.1　既設クラス

6.1.1　TraceLog クラス

標準形式トレースログをコード上で扱うためのクラス。

今回、そのコンストラクタをリファクタリングし、可読性の向上を行った。また、新たに `TraceLogParser` オブジェクトを保持する静的フィールドを追加し、パーサの生成を一回で済むようにした。

6.2　新設クラス

6.2.1　Parser クラス

パーサを実現するための基本的な機能を備えたクラス。

これを継承し、解析メソッドを記述することで専用のパーサを作成する。5.2 に示したメソッドの他に、アルファベットや数字の解析メソッドも有する。これらのメソッドは、単純に継承しただけでは戻り値の型が合わず、使用できない。よって、派生クラスに、処理を `Parser` クラスに委譲し、その結果を使用するインタフェース型にキャストして `return` する同名 (もしくは、それとわかる名前) のメソッドを用意して使用する。

6.2.2 StackForParser クラス

Parser クラスが利用する専用スタック。

パーサが今までのパース結果やポインタを退避させるのに用いる。できるだけ効率よく管理するために用意した。

6.2.3 InputStreamForParser クラス

パース対象を格納し、管理するクラス。

ポインタの制御やポインタの示す文字の提供などを行う。.NET Framework の標準ライブラリである StringReader クラスは、Read メソッドで文字を消費してしまって復元が難しいため、専用のクラスを用意した。

6.2.4 NullObjectForParser クラス

Parser クラスの NullObject を実現するための基本的な機能を備えたクラス。

これを継承し、解析メソッドを含むインタフェースを実装することで、専用のパーサ用 NullObject を作成する。

6.2.5 TraceLogParser クラス

標準形式トレースログ用のパーサクラス。

標準形式トレースログをパースするための解析メソッドや結果を取得するためのプロパティがある。

6.2.6 NullObjectOfTraceLogParser クラス

標準形式トレースログ用のパーサクラス用の NullObject クラス。

メソッドは基本的に何もしないが、効率やアルゴリズム上の理由などで処理を行う場合がある。ただし、NullObject としては異例なため、そのような場合は保守性が低下する恐れがある。

6.2.7 IParser クラス

Parser クラスと NullObjectForParser クラスを結ぶインタフェース。

これにより、メソッドチェーンによる EBNF のような表現を実現している。

6.2.8 ITraceLogParser クラス

TraceLogParser クラスと NullObjectOfTraceLogParser クラスを結ぶインタフェース。

これにより、メソッドチェーンによる EBNF のような表現を実現している。

7 シーケンス

例として、"[.]Task.state=="RUNNING""という誤ったログをパースするシーケンスを、途中までだが、図 4 に示す。解析メソッドの適用に成功すれば次の解析メソッドを試み、失敗すれば以降の解析メソッドは NullObject のものを呼ぶことでパーサの解析メソッドを適用しないようになっている。

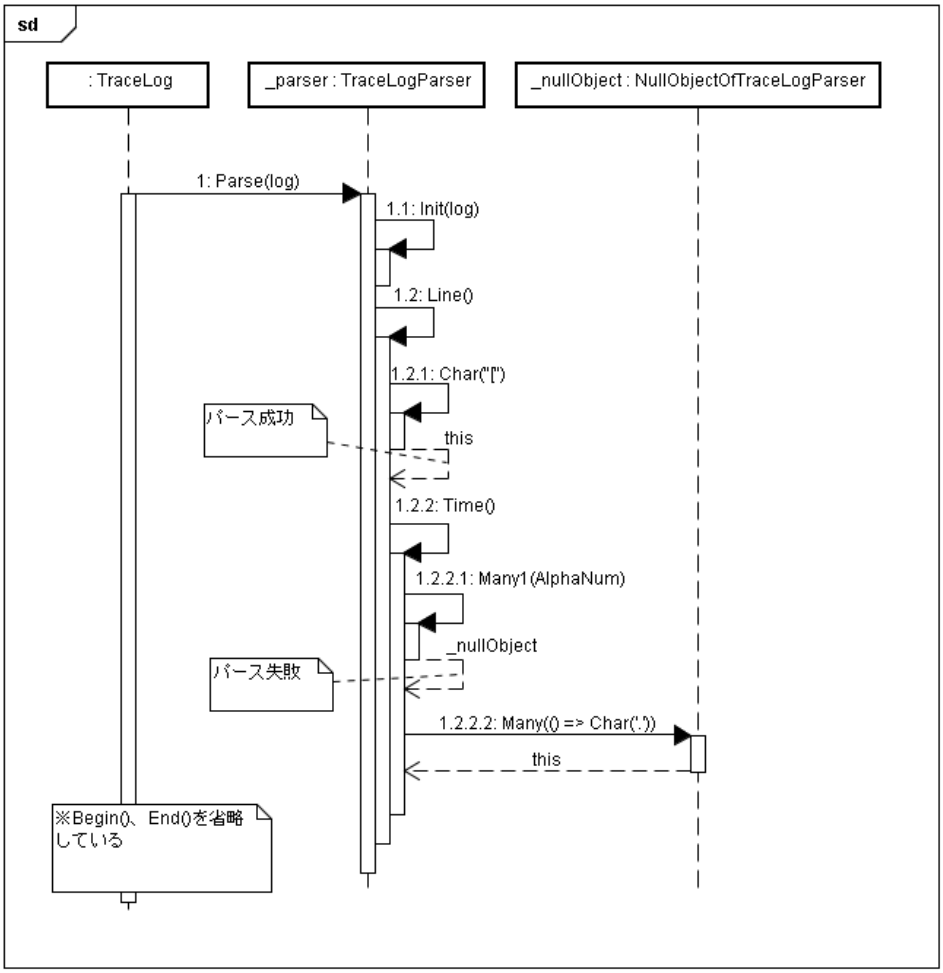

図 4　パージング時のシーケンス例 (途中まで)

また、失敗した状態で OR メソッドが呼ばれるとバックトラックし、OR メソッド以降の解析メソッドにより再度パースが行われる。その他の状態で呼ばれた時の OR メソッドの処理を図 5 に示す。3 つ目の場合は、例えば `hoge().OR().fuga().OR().piyo()` という生成規則があった場合、`hoge()` が成功した時の `fuga()` と `piyo()` の間にある `OR()` の挙動を示している。

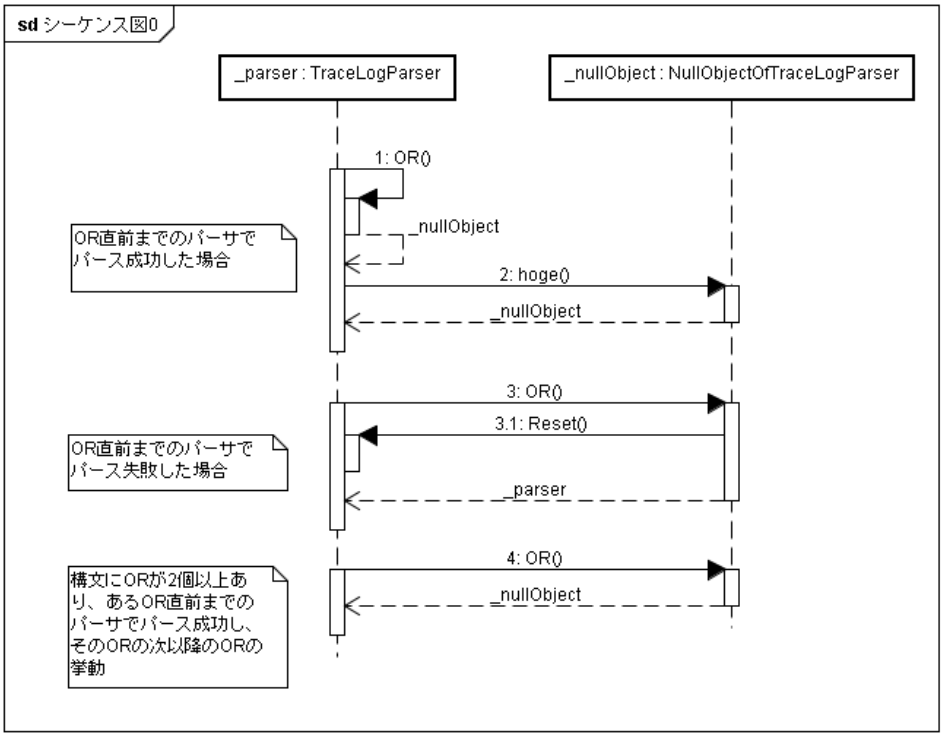

図 5　各状態での OR メソッドの処理

第 III 部
評価

本章は、4.2 で紹介した [1] を用いたパーサ (以降、C#版 Parsec)、前章で紹介したパーサ (以降、自作パーサ) を比較し、前章のパーサを選択した理由を記述する。また、それら二つのパーサおよび従来のパース手法で性能比較を示す。

8 他手法との比較

8.1 可読性

まず、C#版 Parsec のコードを次に示す。このパーサは、Haskell の Parsec を模したコードになっており、例えば Listing7 のように置き換えることができる。すなわち、6 では、`in` より右の解析メソッドを上から順に見たものが生成規則を表している。しかしながら、この記述は、コード上でクエリ式を記述できる LINQ を利用しているため、初見では理解しにくいということが現在の開発メンバとのレビューで出てきた。また、「右の解析メソッドを上から順に見たものが生成規則を表している」ことがわかったとしても、`select` が何をするのか、何が記述されているのかが理解しにくい。そのため、Haskell 経験者であればよいとは思われるが、そうでない者にとっては可読性が従来のものより低くなってしまう可能性がある。

自作パーサは、生成規則を表す解析メソッドのチェーンが EBNF に近いため、自作パーサの方が可読性が高いというレビュー結果が得られた。

Listing 6　C#版 Parsec のコード例

```
1  OBject = (from otn in ObjectTypeName
2            from u1 in Char('(')
3                    from ac in AttributeCondition
4                    from u2 in Char(')')
5                    select Base(otn).Append(u1).Append(ac).Append(u2).ToString())
6                   .OR(
7                   (from on in ObjectName
8                    select Base(on).ToString()));
9
10 // メソッドは、引数に基づいてクラスを新規作成して返すメソッド BaseStringBuilder
11 // クラスに関しては、StringBuilder.NET Framework ライブラリを参照
```

Listing 7　Listing6 を Haskell に置き換えたコード例

```
1  OBject = do{ otn <- objectTypeName
2             ; u1 <- char '('
3             ; ac <- attributeCondition
4             ; u2 <- char ')'
5             ;   return$ otn ++ u1 ++ ac ++ u2
6             }
7          <|>
8          do{ on <- objectName
9             ; return $ on
10            }
```

8.2 保守性

C#版 Parsec のクラス構造は、機能ごとによく分割・整理されており、また、文字列以外にも対応できるように設計されているため汎用性が高い。この C#版 Parsec およびパーサを変更するには、Parsec の実装に関する知識が必要になる。それに伴い、Haskell の知識が必要になる場合がある。よって、チューニングまで行おうとすると、事前知識がなければ学習コストがかかってしまう。

自作パーサは、文字列のパージングに特化したものであり、文字列化できないもののパージングはほぼ不可能に近い。しかしながら、TLV で用いる場合、文字列以外は現状では考えられないため、あまり問題にならない。変更の際のコストに関しては、自作パーサでも、NullObject との連携や解析メソッドのチェーンの理解に学習コストがかかる。

次に、生成規則の変更についてを考える。これについては、両方とも EBNF を表現したものであるため、基本的な手順や特徴は同じである。記述変更すべき箇所は、目的の生成規則を表現している箇所をみればよいので、従来の手法より変更箇所を発見しやすく、変更箇所の個数も削減できる。変更には、生成規則の追加、削除がある[*3]ので、この順序に見ていく。まず追加時には、次の手順を踏む。

1. 新規の生成規則にしたがって解析メソッドを作成する
2. 新規の解析メソッドを既存の生成規則に挿入する
3. C#版 Parsec の場合：新規の解析メソッドを挿入した既存の生成規則の select 句を書き換える

削除時には、次の手順を踏む。

1. 既存の生成規則から目的の解析メソッドを削除する
2. C#版 Parsec の場合：解析メソッドを削除した生成規則の select 句を書き換える。

以上のように、C#版 Parsec の場合は、自作パーサより多く記述しなければならない場合があるので、変更箇所数でいえば、自作パーサの方が優れる場合がある。

8.3 デバッグの容易性

C#版 Parsec は、かなりの割合をデリゲートが占める。そのため、デバッガによるステップ実行では、どのデリゲートが適用されているのかがわかりづらく、結果、処理の流れが把握しづらいため難しい。また、流れ自体も実装と関わっているためやはり Parsec などの知識が必要となる。

自作パーサは、コードを見るよりもデバッガによるステップ実行を行った方が分かりやすい。特に、取得したいパース結果の格納時の、次のコードの※ 1、2 である。

```
1 var object_ =
2         ObjectTypeName().Char('(').AttributeCondition().Char(')')
3         .OR().
4         ObjectName();
5
6 object_.ObjectValue = Result();      // ※1
7
8 // は、HasObjectTypeValueObjectTypeNameが真でも、ほかで失敗すれば偽である。()
```

[*3] 入れ替えは、追加・削除を組み合わせたものと考えられるため除外

```
9 object_.HasObjectTypeValue = false;  // ※2
```

例えば※ 1、2 は、object_が何を指すのかを推測して理解や記述をする必要がある。デバッガによるステップ実行を行うことで、object_が明確となり、処理が間違っていないか正確に確認できる。また、C#版 Parsec のようにデリゲートを多用しておらず、使用していてもシンプルなシーケンスであるため、どのメソッドまたはデリゲートを適用させているのかが C#版 Parsec よりシーケンスを追いやすい。

8.4 性能比較

8.4.1 処理速度

各手法間で処理速度の違いが出るかを調査した。TLV は、ユーザから高速化の要求を頂いているため、処理速度がリファクタリングにより著しく低下することは望ましくなく、逆にリファクタリングにより高速化することが望ましい。

以下の方法で行った。

用いるもの　各パーサを実装した TLV(計 3 つ)、ストップウォッチ
計測開始　「新規作成ウィザード」の「OK」ボタンを押した状態から離したとき
計測終了　「初期化中」という窓に切り替わったとき
計測順序　自作パーサ 3 回→ C#版 Parsec3 回→従来 3 回
入力 (.log)　6958 行 (TLV 付属のサンプルファイル fmp_long.log)
入力 (.res)　リソース数 28 (TLV 付属のサンプルファイル fmp_long.res)
備考　1 回の測定につき TLV の起動・終了を行う

実施環境は次の通り。

OS　Windows 7 Professional　x64
CPU　Core2Duo E8400(3.0GHz)
メモリ　DDR2-800　1GBx2+2GBx2

結果を表 2 に示す。

表 2　処理速度の測定結果

	1 回目	2 回目	3 回目
自作パーサ	2 分 26 秒	2 分 21 秒	2 分 22 秒
C#版 Parsec	2 分 40 秒	2 分 38 秒	2 分 38 秒
従来手法	2 分 25 秒	2 分 24 秒	2 分 24 秒

順位は自作パーサ、従来手法、C#版 Parsec の順であった。自作パーサは、処理時間が C#版 Parsec より約 11%、従来手法より約 2% 短く、処理速度面でも優れていることがわかる。

自作パーサが他より高速である理由は、次のものが考えられる。

- 文字列をそのまま扱うのではなく、文字列を文字に分解してコストが低い文字処理を行う
- 自作パーサ専用のスタックおよび入力ストリームのため無駄が少ない

- 文字列に特化することで、高速な文字列処理が行える StringBuilder クラスを採用できた

C#版 Parsec がこのような結果になった理由は、次のものが考えられる。

- クラスメソッド呼び出しよりコストのかかるデリゲートを多用している
- 汎用性を高めるために、Parsers クラスの Rep1 メソッドにて要素の結合を配列の結合で行っている (Parsec の実装に似せている)
- new を多用している (Result クラス、配列など)

8.4.2 解析能力

解析能力は、3 種類全ての手法で変わらない。自作パーサおよび C#版 Parsec は LL(K) 文法を解析でき、従来の手法は.NET Framework の正規表現が通常の正規表現より柔軟かつ強力にできている [2] ため、標準形式トレースログのパージングに問題はないからである。しかしながら、従来の手法で使用されていた正規表現では、括弧のネストを正確にパースするのに、現状以上に複雑な正規表現となってしまう[*4]。たとえば、次のコードは<,>のネストを正しくパースする正規表現 (詳細は [3] 参照) だが、これを現状の正規表現上の括弧のネストが起こりうる箇所に挿入することになる。

```
1 ^[^<>]*(((?'Open'<)[^<>]*)+((?'Close-Open'>)[^<>]*)+)*(?(Open)(?!))
```

したがって、従来の手法では現状より可読性が低くなってしまう。ゆえに、括弧のネストに対応するという将来性を考慮しても、自作パーサの有用性は高いと言える。

参考文献

[1] LukeH's WebLog : Monadic Parser Combinators using C# 3.0,
URL:
http://blogs.msdn.com/lukeh/archive/2007/08/19/monadic-parser-combinators-using-c-3-0.aspx

[2] .NET Framework の正規表現,
URL:http://msdn.microsoft.com/ja-jp/library/hs600312(v=VS.80).aspx

[3] グループ化構成体,
URL:
http://msdn.microsoft.com/ja-jp/library/bs2twtah%28VS.80%29.aspx#BalancingGroupDefinitionExample

[*4] 従来の手法では、括弧のネストのパースを実現していない。また、他の処理でも括弧のネストに対応しきれていない。