平成 28 年度 修士論文

拡張可能な構文定義の枠組みの提案

東京工業大学数理・計算科学専攻
学籍番号 15M37087
桐山裕匡

指導教員
増原英彦

平成 29 年 1 月 19 日

概要

プログラミング言語には，言語ごとに得意な処理と不得意な処理がある．ある言語が不得意な処理を別の言語が持っている場合，2 つの言語を統合することで，不得意な処理がより少ない言語が作れる．例えば，SQL ではデータの検索を完結に記述することができるが，OS やコンパイラを作れない．一方で，C 言語では OS やコンパイラを作れるが，データの検索を記述するのは一般に困難である．C 言語に SQL の言語機能を統合することで，検索プログラムにも OS にも向いた言語を作ることができる．

本研究ではこのような統合言語の処理系を作る上で重要な，組合せ可能な構文解析器の実現に取り組む．構文解析器とは, 入力された文字列を, プログラムがより扱いやすい抽象構文木などのデータに変換するプログラムである．構文解析器を直接記述することは稀である．構文解析器の定義には, 構文定義から構文解析器を生成する yacc を用いる．あるいは, Haskell の Parsec[2] に代表されるパーザコンビネータライブラリを用いて記述される．

構文解析器を組み合わせることは難しい．例えば，SQL 文と C 言語を組み合わせた統合言語の構文解析器の実現には，既に定義済みの C の構文解析器のソースコードを直接書き換え，SQL の構文解析器のソースコードを貼り付ける必要がある．また，新言語の開発者は拡張を行うために，C と SQL の構文解析器のソースコードを入手でき，かつ，C と SQL の構文解析器ソースコードを読んで理解しなければならない．

また，構文解析器が生成したそれぞれの抽象構文木を安全に組み合わせる方法が必要になる．構文解析器を独立に定義を行った場合, それぞれの抽象構文木は型の不一致により組み合わせることができない. 組み合わせるには, 安全でない実行時の型変換を行うか, 抽象構文木の定義を変更するために既存のソースコードの変更をしなければならない．

本研究では、合成可能なパース規則を記述する言語を提案してこの問題を解決する．この言語は, 言語定義のモジュール分割と言語言語定義の組み合わせを行う機能を持つ．提案言語の形式化を行い, また Scala による実装を示した．

実装にあたっては, 構文解析器が生成する抽象構文木に Object algebra を用いることで値を拡張可能にし，パース規則の合成に mixin 継承を用いることで構文規則の拡張を可能にした．

謝辞

本論文を書くにあたり、多くの人におせわになりました．

指導教員の増原先生には，日頃からゼミや輪講で丁寧な指導をいただき，研究にもアドバイスをいただきました．青谷先生は研究のテーマについてのアドバイスや，本研究を進める上で欠かせない知識を鍛えてくれました．研究室の同期と先輩からは大きな刺激をもらい，また些細な質問や雑談にも付き合ってもらいました．皆様には感謝しきれない程の支えをいただきました．この場を借りてお礼申し上げます．

目次

第 1 章

はじめに

今日のプログラミング言語は様々な言語機能を拡張することで発展してきた.

拡張される機能は既存の言語から持ってこまれることもある.

言語実装者は負担をできる限り減らしたい.

コンパイラ実装者にとって, 言語機能の実装を分割する方法は課題となってきた.

プログラミング言語設計者にとって, 構文解析器は面倒な作業の一つであり, できる限り労力を割かずに抽象構文木の操作を実装する段階に進みたい. そのため, 言語設計者は構文解析器を書く手間を省くために yacc やパーサコンビネータライブラリを用いることで一部の労力を減らした.

プログラミング言語を設計する際に, 完全に新規な構文を利用することは稀で, 既存の構文の一部を真似ることが多い. そのため, 差分のみを記述することができれば大幅に労力を減らすことができる.

また, 別言語の機能を取り込む際には, 構文解析器を組み合わせて使うことができれば, 処理系の実装者は組み合わせ方のみを記述することで再実装の手間を減らすことができる.

しかし依然としてプログラミング言語処理系を作る時には, 構文解析器を新規に書くことが多く, 再利用されることは稀である.

我々は, 拡張や組み合わせによって, 新たな構文解析器を定義する際に問題が生じるため, 再利用されないと考えた.

この論文は以下のように構成される. 第2章には, 本論文を読む上で必要な背景知識として, オブジェクト指向言語における The Expression Problem と, その The Expression Problem の解決の 1 つである Object Algebra を説明する. また, 構文解析器の実現方法の一つとして Parser Combinator についても説明をする 第3章 では既存の構文解析器の問題を例を交えて説明する. 第5章 では, 提案言語の詳細とその意味論を定義する. 第6章では 第5章 の言語を Scala 上での実現方法を説明する. 第7章 では, 拡張の具体例として, MinCaml が XML 式を扱えるようにする方法を示す. 第9章 では, 本研究の結論と今後の課題を述べる.

第 2 章

背景知識

この章では、本研究を理解する上で必要となる, The Expression Problem, Object Algebra, Parser Combinator についての説明を行う.

2.1 The Expression Problem

The Expression Problem[9] は, プログラムの拡張性に関する古典的な問題である. プログラムとはデータと, データの変換を行う関数の 2 つの要素からなる. そのため, プログラムを拡張する場合には 2 つの拡張の方向性がある.

- 関数が扱うデータの種類の追加
- データに対する新しい関数の追加

1 つ目はプログラムが扱うデータの種類を増やすことによって, 既存の関数の振る舞いを拡張するものである. 2 つ目は定義済みのデータ型に対して新たな関数を追加することで, 新たな処理を追加するものである.

The Expression Problem は Philip Wadler によって初めて言及された. The Expression Problem は, 以下の 5 つの要件を満たしながらプログラムの拡張を行うにはどのようにすればよいか, という問題である.

1. **Extensibility in both Dimensions**: データの種類の拡張とそれに伴う既存の処理の拡張. 新たな処理の追加
2. **Strong static type safety**: 関数が扱えない種類のデータが適用されないようにする. これを静的にチェックする.
3. **No modification or duplication**: すでにあるコードへの変更や、コードの複製を行わない
4. **Separate Compilation and Type checking**: 静的検査はリンク時, 実行時より前に行われなければならない

5. **Independent extensibility**: 独立して作られたプログラムの拡張を一緒に使うことができる

The Expression Problem は, はじめ (1) から (4) の 4 つのみで定義されていたが, Odersky と Zenger らによって (5) が要件に追加された [6].

以下ではこの問題を具体例を用いて説明する. オブジェクト指向言語におけるプログラムの拡張の具体例として, 簡単な電卓プログラムを考える.

はじめに整数の加算のみができる電卓（以降, 加算電卓と呼ぶ）を作る. この電卓のプログラムが入力として受け取る式の文法は以下のとおりである.

```
expr ::= Plus(expr, expr)
       | Num( int )
```

Plus は 2 つの式をとり式を構築し,Num は整数値 1 つをとり式を構築する. この式の文法を Java 言語で定義すると, 図 2.1 のようになる. Exp インターフェースは値に評価可

```
interface Exp{ int eval(); }
class Num extends Exp {
  int x;
  public Num(int x){ this.x = x};
  public int eval(){ return x; }
}
class Plus extends Exp {
  Exp l, r;
  public Add(Exp left, Exp right) {
    this.l = left;
    this.r = right;
  }
  public int eval(){
    return l.eval() + r.eval();
  }
}
```

図2.1　電卓プログラムの定義

能な式を表すインターフェースで, eval() 関数を用いて式を値に評価する. この Exp クラスを Num クラスと Add クラスは継承している. Num クラスは整数 1 つを引数に取るコンストラクタを持ち, Add クラスは Exp 型の式 2 つを引数に取るコンストラクタを持つ.

この数式の評価器は, 整数値ならば評価値としてその値, 2 つの式の和なら 2 つの式を評価した値の和を返す.

オブジェクト指向言語では, プログラムが扱う新しい種類の値を追加するのは容易である. 例えば, 式に新たに乗算を表す値を追加するには, Exp 型を継承した Subtract クラス (図 2.2) は, 既存のクラス定義, インターフェース定義に手を加えることなく追加できる.

```
class Subtract extends Exp{
  Exp l,r;
  public Subtract(Exp left, Exp right){
    this.l = l; this.r = r;
  }
  public int eval(){ return l.eval - r.eval()};
}
```

図2.2 新たに追加される引き算を表すデータ型の定義

データの拡張が簡単な一方, オブジェクト指向言語では関数の種類を追加することが難しい.

前述の電卓プログラムを拡張して, 数式を文字列に変換する機能を追加する. この時, 全ての Exp インターフェースを継承したクラスについて文字列に変換できる必要があるため, Exp インターフェースは以下のように変更する必要がある.

```
interface Exp{
  public int eval();
  public String toString();
}
```

また, この変更に基づいて Exp クラスを継承した Num, Plus, Minus クラス全てについて, toString() メソッドの実装を行う必要がある.

そのため, 新たな操作を追加したときには, 拡張前の基底クラスのメソッドの定義と継承したクラスの定義を書き換えなければならない. この拡張はすでに存在する定義を変更しているため, 「**すでにあるコードへの変更や, コードの複製を行わない**」の条件に反する.

この問題はデータに対する新たな操作を追加するときに起こる, オブジェクト指向言語での一般的な問題である. オブジェクト指向言語での The Expression Problem は, 既存のコードを変更せずに新たなメソッドの追加を安全に行えるか, という問題になる.

```
class Num extends Exp{
  ...
  public String toString(){ return x.toString();}
}
class Plus extends Exp {
  ...
  public String toString(){
    return "(" + l.toString + "+" r.toString() + ")";
  }
}
class Minus extends Exp {
  ...
  public String toString(){
    return "(" + l.toString + "-" r.toString() + ")";
  }
}
```

図2.3 拡張されるコード ((...) は省略した拡張前との重複部分)

2.2 Object Algebra

Object Algebra パターンは、ジェネリクス機構を持つオブジェクト指向言語における, The Expression Problem の解決となるデザインパターンの 1 つである. *Object Algebra* という単語は, Object Algebra インターフェースを実装したクラスを指す言葉である. 以下では, Object Algebra インターフェースを実装したクラスを Algebra, Object Algebra パターンを Object Algebra と呼んで区別する.

以下では2.1節と同様に, 電卓の例を用いて Object Algebra パターンの説明をする.

2.2.1 Object Algebra による値の定義

通常のオブジェクト指向プログラミングでは, まず数式を表す抽象クラスを定義し, その派生型として具体的なデータ型とそのコンストラクタを定義した (図 2.1). 一方, ObjectAlgebra ではコンストラクタを定義する代わりに, Object Algebra インターフェースを定義する. Object Algebra インターフェースは, コンストラクタと同じ型の引数を持ち, 抽象型を結果として持つ汎用インターフェースである.

2.1節 と同様の加算電卓を定義する. 数式を表す抽象クラスを定義する代わりに, これ

らの関数をメンバに持つ抽象クラスを定義する.

まず電卓の数式を表す Algebra を, ExpAlg として定義する (図 2.4). ExpAlg は型引数 A を取る. A は計算結果を表す型変数であり, たとえば数式を計算して整数の値を取り出す場合は A は Int で具体化され, 数式を表す文字列に変換する場合は String に具体化される.

ExpAlg は Num と Add の 2 つを Object Algebra インターフェースに持つ. Num は整数を 1 つ引数にとって A 型の計算結果を返す関数である. また Add は A 型の足し算の左辺の計算結果と A 型の右辺の計算結果を引数に取り, その結果から最終的な計算結果を作る関数である.

```
trait ExpAlg[A]{
  def Num : Int => A
  def Add : (A, A) => A
}
```

図2.4 Object Algebra によるデータの定義

図 2.5 は 1 + 2 + 3 を表す式の Object Algebra による表現である.

```
def six[A](f : ExpAlg[A] ) : A
= f.Add(f.Num(1), f.Add(f.Num(2), f.Num(3)))
```

図2.5 1 + 2 + 3 を表すデータの Object Algebra による表現

Object Algebra を用いたプログラムではコンストラクタを使用せず, 値を作る代わりに Algebra を用いて値を作る関数を定義する. six は型パラメータ A をとり, 引数に ExpAlg[A] 型の値 f をとって, 結果に A 型の値を返す関数として定義されている. 関数の本体では, 受け取った f の Object Algebra インターフェースを使って値を構築する.

2.2.2 Object Algebra の値に対する計算

図 2.5 のように, Object Algebra パターンにおいて, 値は Algebra を引数にとる関数として表現される. 一方で計算は,Object Algebra インターフェースを実装した値を引数に与えることで計算を行う.

図 2.5 で定義した six を評価して Int 型にする計算を定義したい. そのために, Object Algebra では, ExpAlg[A] インターフェースの型変数 A を Int で具体化した ExpAlg[Int] インターフェースを実装する. ExpAlg[Int] インターフェースを実装した ExpEval クラスを 図 2.6に示す.

```
class ExpEval extends ExpAlg[Int] {
  override def Add(l : Int, r: Int) : Int = l + r
  override def Val(i : Int) : Int = i
}
```

図2.6　Int 型の結果を持つ ExpAlg の実装

ExpAlg[A] の型変数 A を Int で具体化したため, Add の 2 つの引数の型は Int 型になっている. また, Add と Val の結果の型も Int になっている.

Add は 2 つの部分式の計算結果を引数にとり, 整数を結果として返す. Add の第一引数 l には, 加算式の左辺が ExpAlg[Int] によって計算された結果が渡され, 第二引数 r には加算式の右辺が評価された結果が渡される. Add では 2 つの引数を足した結果 l + r を返している.

Val は定数値を引数にとり, 整数を結果として返す. Val は受け取った引数 i をそのまま結果として返している.

この実装したクラスのインスタンスを six の引数に与えることで, 評価結果を得ることができる. 実際の計算は図 2.7 のようにして行う. 図 2.7 の 1 行目では, Algebra として ExpEval クラスのインスタンスを evalInt に代入している. この Algebra を six の引数に与えることで計算を行い, 計算結果を得ている.

```
> val evalInt = new ExpEval
> val result = six(evalInt)
result: Int = 6
```

図2.7　Object Algebra のインタプリタ上での計算例

six に対して別の計算を行うには, また別の Algebra を定義すれば良い.

例えば, six から文字列を得るためには, ExpAlg[String] を実装した Algebra を作り, そのインスタンスを six の引数に与えることで式を文字列に変換することができる (図 2.8, 図 2.9).

2.3　Parser Combinator

プログラムを書いていると小さな専用言語を処理しなければならない場面がある. 例えば, XML や JSON のような形式より単純な独自の形式を設定ファイルとして扱うような場面が考えられる. このようなときに, 設定ファイルからプログラムが扱いやすいデータに変換をする構文解析器 (Parser) が必要になる.

```
class ExpToStr extends ExpAlg[String] {
  override def Add(l : String, r : String ) : String
    = "(" + l.toString + "+" r.toString + ")"
  override def Val(i : Int) : Int = i.toString
}
```

図2.8 文字列を出力する Algebra の定義

```
> val expToStr = new ExpToStr
> val result = six(expToStr)
result: String = (1 + (2 + 3))
```

図2.9 文字列を出力する Algebra の計算例

構文解析器を定義するには, 独自の構文解析器を作るほかに, パーサ生成器 (Parser Generator) と呼ばれる外部プログラムによって構文解析器を自動生成する方法がある. 例えば C 言語で書かれた構文解析器を生成する yacc や bison といったプログラムがある. しかし, 新しいツールの使い方や, ツールの出力をプログラムにつなげる方法を学習しなければならない. また, 外部ツールの数が増えるに連れて組み合わせが複雑になる.

これらと異なる構文解析器を作る方法として, Parser Combinator がある. パーサ生成器がドメイン固有言語 (DSL) を用いて文法を定義しているのに対して, Parser Combinator は言語内 DSL を用いて文法を定義する. ここでの言語内 DSL とは, 構文解析器を組み立てるための部品となる関数や演算子の定義をまとめたライブラリである. Parser Combinator の演算子や関数は文脈自由文法を組み立てる手順に 1 対 1 の対応を持つため, 直感的な構文の組み合わせや可読性に優れたコードの記述の助けになる.

2.3.1 Parser Combinator による文法の定義

構文解析器は, 文字列を解析し, その結果を返す関数である. そのため,Parser Combinator は構文解析器を組み合わせて, 新たな構文解析器を作る高階関数として定義される.

以下では Parser Combinator の使用例を示す. いま, 図 2.10 にあるような, 整数と足し算からなる文法をもつ言語を考える.

| は選択可能な生成規則を区切り, シングルクオート (') で囲まれた文字列は具体的な文字列, 角括弧 (<>) で囲まれた記号は生成規則を表す. また、波括弧 {} は, 括弧で囲まれた内容の 0 回以上の繰り返しを表す.

expr(式) は 1 つの *term*(項)、または *term* の直後に + 演算子と *expr* が連続したもの

```
<expr> ::= <term> `+' <expr> | <term>
<term> ::= <num>
<num> ::= digit { digit }
<digit> ::= `0' | `1' | ... | `9'
```

図2.10 文法

である。また、*term* は *num*(整数) 一つからなる値であり、*num* は *digit*(数字) を 1 回以上繰り返したものになっている.

この言語の文字列を入力としてとり, この言語の抽象構文木を構築するプログラムを定義する. 図 2.10 に示された言語に対する Scala による構文解析器の定義を 図 2.11に示す.

算術式の構文解析器は `Arith` クラスとして定義されている. `Arith` クラスは `RegexParsers` を継承している. `RegexParsers` には構文解析器を記述するための関数と, 識別子, 文字列, 数値などの語句を認識するための基本的な構文解析器の定義が含まれている.

```
class Arith extends RegexParsers {
  def expr : Parser[Any] = term ~ "+" ~ expr | term
  def term : Parser[Any] = num
  def num : Parser[Any]  = """[0-9]+""".r
}
```

図2.11 Scala の Parser combinator ライブラリを用いた構文解析器の定義

`Arith` クラスには 3 つの定義が含まれており、これは算術式の生成規則を表している. この表現は図 2.10 と対応が取れる.

`|` 演算子と`~` 演算子は, 2 つの構文解析器から新たな構文解析器を作る。

`|` 演算子によって作られた構文解析器は、まず演算子の左の構文解器を実行し、その構文解析が成功した場合はその結果を返し、失敗した場合は演算子の右の構文解析器を実行する. この演算子は図 2.10における | と対応する.

また、`~` 演算子によって作られた構文解析器は、まず演算子の左の構文解器を実行し、成功した時に右側の構文解析器を実行する. 2 つの構文解析が成功する場合にのみ構文解析に成功する.

2.3.2 構文解析器の出力

結果を得るためには, `^^` という演算子を用いる. `^^` 演算子は構文解析器の処理結果を変換するための演算子である. 構文解析器を `P`, 変換関数を `f` とすると, `P ^^ f` は構文解

析器 P が解析した結果 X を f に適用し, f(X) を結果として返す.

例えば以下に示すのは浮動小数点を表す文字列を解析して, その結果を Double 型に変換して返す構文解析器である.

```
def doubleParser : Parser[Double] = floatingPointNumber ^^ (_.toDouble)
```

これを用いて, 図 2.11 の構文解析器の結果から抽象構文木を構築するためには, 以下のようにする.

まず, 抽象構文木を表す抽象クラス Exp を作る. 抽象構文木の各ノードを表すクラスとして, Exp クラスを継承した Add と Num クラスを定義する. Add クラスは加算を表し, + の左の式と右の式から値をつくるコンストラクタを持つ. Num クラスは, 整数 1 つから値を作るコンストラクタを持つ.

```
abstract class Exp
case class Add(l : Exp, r : Exp) extends Exp
case class Num(i : Int) extends Exp

class Arith extends RegexParsers {
  def expr : Parser[Exp] = (
      term ~ "+" ~ expr ^^ (case t ~ "+" ~ e => Add(t, e))
    | term
  )
  def term : Parser[Num]
    = num ^^ (x => Num(x))
  def num : Parser[Int]  = """[0-9]+""".r  ^^ (x => x.toInt)
}
```

num は 0 から 9 までの数字の列を解析し, その文字列に対応する整数を返す構文解析器となる. この num による構文解析の結果 x を用いて, term は整数 1 つからなる式 Num(x) を構築する. expr は, 加算式の + 演算子の左右の式をそれぞれ構文解析した結果 t,e を受け取り、加算を表す式 Add(t,e) を結果として返す.

第 3 章

問題

本章では, 組み合わせ可能な構文解析器を実現する際に発生する 5 つの問題について述べる.

3.1 名前の衝突

組み合わせられることを考えずに構文解析器を定義するとき, 構文解析器を独立に定義した時, 構文の名前が意図せず重複することがある.

```
exp  ::= term | term '+' exp
term ::= number | '(' exp ')'
```

```
exp  ::= term | term '*' exp
term ::= number | '(' exp ')'
```

図3.1　2 つの独立して定義された構文解析器

例えば図 3.1 では, 2 つの文法が定義されており, `exp` と `term` が左右同じ名前で定義されている.

この 2 つの構文解析器を同時に再利用している時, `exp` という名前がどちらを参照しているのかわからない. 複数の構文解析器を組み合わせる場合, 名前の重複する要素から, どちらを選ぶかを明示する方法が必要になる.

3.2 抽象構文木の型安全な組み合わせ

この問題は, 一般的に The Expression Problem として知られる問題である.

オブジェクト指向言語において, 抽象構文木の定義方法は一般的に以下のようになる. まず抽象構文木を表す抽象クラスを定義する. この抽象クラスは, trait, interface, abstract class として定義することができる. 次に抽象構文木のノードはこの抽象クラスを継承して定義する.

図 3.2 は上記の定義の一例である. インターフェースとして`Exp` を, `Add` と `Lit` は`Exp`型の構文木のノードを表すクラスとして定義した.

```
abstract class Exp { int eval(); }
class Add extends Exp { ... }
class Lit extends Exp { ... }
```

図3.2 整数とその和からなる数式の抽象構文木の定義

このExp 型の値を構築する構文解析器を定義したものが図 3.3である.

lit は, 数値リテラルを受理し, Lit 型の値を返す構文解析器. add は + 演算子で繋がれた式を受理する構文解析器である. exp は, lit とadd の選択として定義した.

```
def exp : Parser[_X_] = add | lit;
def lit : Parser[Lit] = ..
def add : Parser[Add] = ..
```

図3.3 整数と和からなる式の構文解析器

この時, 構文解析器 exp の結果型 (図 3.3内の_X_) はどのように定義するのが良いだろうか. 素朴に考えると,_X_型は Add クラスと Lit クラスの共通の継承元である Exp 型を構文解析器の結果型とするのが自然になる.

しかし, 独立に定義された式を統合しようとした時, この方法では問題が生じる. 以下, ラムダ式に整数とその操作を加えた構文解析器を定義する例を用いて構文解析器の組み合わせの問題を説明する.

Exp 型とは独立に定義された, ラムダ式の抽象構文木型 LambdaExp が定義されている. 抽象構文木のノードには, 関数抽象 (Abs), 関数適用 (App), 変数Var がある.

```
abstract class LambdaExp {  }
class Abs extends LambdaExp { ... }
class App extends LambdaExp { ... }
class Var extends LambdaExp { ... }
```

図3.4 ラムダ式の抽象構文木型の定義

プログラマは, ラムダ式に整数の操作を加えたいと考える. この時, 何らかの方法で構文解析器 lambdaExp と expr を組み合わせ, 新たな構文解析器 lambdaIntExp を作ることが出来たとする.

```
def lambdaIntExp : Parser[?] = ...
```

```
def lambdaExp : Parser[LambdaExp] = lambdaAbs | lambdaApp | lambdaVar
def lambdaAbs : Parser[Abs] = ...
def lambdaApp : Parser[App] = ...
def lambdaVar : Parser[Var] = ...
```

図3.5 ラムダ式の構文解析器

このとき, lambdaIntExp には exp と lambdaExp の 2 つの式が入り混じっているため, 結果型は, LambdaExp と Exp の共通型にすることになる. この 2 つの型は共通の型を継承していないため, 新たに LambdaIntExp クラスを定義し, LambdaExp と Exp に継承させることになる. この場合, 既存のクラス定義が書かれたソースコードに手を加えなければならず, もはやライブラリとして利用できない.

もしくは, 構文解析結果を Object 型にすることも考えられる. しかし, この場合は実行時に型による条件分岐を行わなければならず, 本来コンパイル時にわかっているはずの型エラーが実行時に発生することになる.

いずれの状況も望ましくない.

3.3 再帰的構文の組み合わせ

実用的なプログラミング言語の構文には, 構文要素に再帰的な構造が現れるため, 言語定義を拡張した際に新たに再帰構造が作られることもある. 単純な拡張によって文法の再帰構造を新たに作るのは, 単純な方法では出来ない.

```
plus ::= term '+' plus | term
```

```
minus ::= term '-' minus | term
```

図3.6 plus と minus の文法定義

図 3.6 には, term を + で連結した式を定義した plus と, term を - で連結した式を定義した minus が定義されている. この plus と, minus を組み合わせて, + または - で term を連結した式 plus-minus を, 図 3.7 のように定義した

素朴には, plus と minus を | でつなげば十分に思えるが, その場合の定義は図 3.8のようになる. この定義は 図 3.7 とは異なる. plus の右辺と minus の右辺には二度と plus-minus は登場しないため, 図 3.8 は再帰的な構文になっていない.

図 3.7 のような定義を作るためには, 構文を組み合わせた際に文法の再帰関係を変更する必要がある.

```
plus-minus ::= term '+' plus-minus
             | term '-' plus-minus
             | term
```

図3.7　+ または - で　term を連結した plus-minus

```
plus ::= term '+' plus | term
minus ::= term '-' minus | term
plus-minus ::= plus
              | minus
```

図3.8　plus と minus を再利用して作られた plus-minus

3.4　構文の差分定義

既存の言語に対して新たな言語要素を追加するときに, もとの言語定義を再度貼り付けるべきではない. 可能ならば, 追加部分だけを記述したい.

直感的に考えると, < もとの定義 > | < 追加の定義 > のように書きたい. しかし, この書き方だと, 定義を追加したのか　, もとの定義との| をとったのかが区別できない.

3.5　分割コンパイル

プログラムが分割コンパイル可能であるとは, プログラムが以下の性質を持つことである.

- モジュールごとに独立してコンパイルできる
- コンパイル済みのモジュールをインポートできる
- インポートしたモジュールのインターフェースの整合性が保たれている

複数のプログラムを組み合わせて使う場合, プログラムの分割定義ができると便利になる.

分割定義したプログラムが, それぞれ独立に検査できるとバグの特定が容易になる. また, 分割定義した要素ごとにコンパイルができると, 全体の再コンパイルが必要なくなるために大幅な時間の節約ができる.

ソースコードを公開せずに, 言語機能をプラグインとして配布することができるため, プログラムのモジュールの再配布が容易になる. 詳細な実装を知らなくても, 提供されているインターフェースを用いることでプログラムを組み合わせることができる.

yacc や bison のようなコードジェネレータは, 文法の分割定義はできるが分割コード生成は出来ない.

第 4 章

解決

4.1　名前の衝突

名前の衝突の解決には, 名前空間を利用する. 各識別子の名前の集合をモジュールごとに分割することで, 名前の衝突を避ける.

```
module A {
exp  ::= term | term '+' exp
term ::= number | '(' exp ')'
}
```

```
module B {
exp  ::= term | term '*' exp
term ::= number | '(' exp ')'
}
```

図4.1　2つの独立して定義された構文解析器

図 4.1 は, 名前空間つきのモジュール宣言である. `module A { ... }` は, モジュール A の名前空間を宣言しており, `module B { ... }` はモジュール B の名前空間を宣言している.

4.2　型安全な構文の組み合わせ

型安全な構文の組み合わせには, Object Algebra を用いる. 抽象構文木のデータ型を Object Algebra を用いて表すことで, 構文解析器を組み合わせた際に自動的に抽象構文木のデータ型も変更されるようにした.

データ型の定義は, 通常の Object Algebra インターフェースを定義することで行う. 図 3.2 の抽象構文木は, Object Algebra では図 4.2のように書く.

これを用いると, 構文解析器は Object Algebra を引数に取り, 構文解析器を作る関数として定義される. 図 3.3 の構文解析器は, 図 4.3のように書き直せる.

同様に, 図 3.5は 図 4.4 のように書き直せる.

構文を組み合わせる時には, 抽象構文木も組み合わせる必要があるため新たな Object Algebra インターフェースを定義する. `lambdaExp` と `expr` を組み合わせるときには,

```
trait ExprAlg[E] {
  def Add : (E, E) => E
  def Lit : E => E
}
```

図4.2 整数とその和からなる数式の Object Algebra による抽象構文木の定義

```
def exp : ExprAlg[X] => Parser[X] = {f => add(f) | lit(f)} ;
def lit : ExprAlg[X] => Parser[X] = ..
def add : ExprAlg[X] => Parser[X] = ..
```

図4.3 Object Algebra による構文解析器

```
def lambdaExp : LambdaAlg[X] => Parser[X]
    = (f) =>  lambdaAbs(f) | lambdaApp(f) | lambdaVar(f)
def lambdaAbs : LambdaAlg[X] => Parser[Abs] = ...
def
lambdaApp : LambdaAlg[X] => Parser[App] = ...
def lambdaVar : LambdaAlg[X] => Parser[Var] = ...
```

図4.4 ラムダ式の構文解析器

LambdaAlg と ExprAlg を組み合わせた Object Algebra を定義する.

```
trait ExprLambdaAlg[X] extends LambdaAlg[X] with ExprAlg[X] {}
```

4.3 差分定義

差分定義を実現するために, 構文定義に特殊な文法 `old.label` を追加する. **label** には, 構文のラベル名が入る. 以下ではその使い方を説明する.

まず, 掛け算からなる式 exp が以下のように定義されている.

```
exp ::= term | term '*' exp
```

この式に, 新たに足し算の差分を追加するには, 以下のように書く.

```
exp ::= old.exp | term '+' exp
```

差分を新たに追加した exp は　old.exp が拡張前の exp の右辺に展開される. そのため, この差分定義は以下のように展開される.

```
exp ::= term | term '*' exp | term '+' exp
```

4.4 再帰的構文の組み合わせ

新たに文法に相互再帰構造を作るには, 4.3節 で用いた old.**label** を用いる. 図 4.5 に定義された, plus と minus を組み合わせて, 足し算と引き算を組み合わせた構文を定義する.

```
plus ::= term '+' plus | term
```

```
minus ::= term '-' minus | term
```

図4.5　plus と minus の文法定義

新たに再帰構造を追加するには, plus に差分として minus を追加し, minus に差分として plus を追加する. これによって, 新たに相互再帰構造を作ることができる.

```
plus ::= plus.old
     | minus
minus ::= minus.old
      | plus
plus-minus ::= plus
```

図4.6　+ または - で　term を連結した plus-minus

4.5 分割コンパイル

提案言語の分割コンパイルは, 各モジュールを Scala のトレイトに変換することで行う. モジュールごとに独立したトレイトに変換されるため, 一つのモジュールを変換しても構文解析器全体の再コンパイルは行われない.

パーザジェネレータは複数の文法定義ファイルを読み込み, 1 つの構文解析器を生成す

る．そのため, 構文定義の一部が変更されたときに構文解析器全体の再コンパイルを行うことになる．そのため分割コンパイルを行うためにはパーザジェネレータを用いることはできない．

そのため，代わりに Parser Combinator ライブラリを用いて構文定義をすることにした．

また，　構文解析器を実行した結果の抽象構文木の表現には Object Algebra を用いる．これによって, 構文木に含まれるデータの種類が増えたとしても, 構文木の定義に手を加える必要がない．

第 5 章

提案

本研究では構文解析器の拡張性の問題の解決として構文解析器の上書き定義を利用する. 本章では上書き定義を操作とする, 構文解析器の定義ための言語を定義する. また, 分割コンパイルの方法を示す.

5.1 言語の定義

```
moduleDecl ::= module m {C}
t ::= t ⊕ t
   | t ⊗ t
   | old.l
   | l
   | string

C ::= l = t
   | C; C
   | import m
   | skip

v ::= v ⊕ v
   | v ⊗ v
   | string
   | l
```

図5.1 項の定義

この言語は, 項 t, コマンド C, モジュール定義 moduleDecl からなる.

5.1.1　moduleDecl

各モジュールは, moduleDecl 宣言で構成される. modlueDecl は　module m C の形で宣言される. m はモジュール名, C はモジュールに含まれる構文規則の宣言列である.

5.1.2　コマンド

コマンド C は 4 つの要素からなる.

`l = t`
: l に項 t を束縛することを表す.

$C_1;C_2$
: セミコロンによって 2 つのコマンドを結合し,　C_1, C_2 を順に解釈する.

`import m`
: 別の構文定義をインポートする

5.1.3　項

項 t は以下の 5 つの生成規則のいずれかからなる.

`t` $\oplus$ `t`
: 2 つの生成規則を $\oplus$　で組み合わせ, 選択を表す生成規則を作る.

`t` $\otimes$ `t`
: 2 つの生成規則を $\otimes$　で組み合わせ, 左の生成規則の直後に右の生成規則が連続することを表す.

`old.l`
: old.l は l=t の = の右辺のみに現れる. l に束縛されていた上書き前の項 t をストアから取り出すことを表す.

`l`
: l に束縛されていた項 t をストアから取り出す.

`string`
: 単純な文字列を表し, これに一致する文を受理する.

5.1.4　値

値 v は項とほとんど同じだが, old.l の規則のみが含まれていない.

5.2　ストアの定義

ストアはラベルから項への部分関数として定義される.

$$S ::= \emptyset \mid \mathrm{l} \mapsto \mathrm{t},\, S$$

図5.2 ストアの定義

図 5.2 にストアの定義を示した. $\emptyset$ は空のストアを表す. $l \mapsto t, S$ は, l を t に移し, それ以外は S と同様のストアを表す.

5.3 評価規則

$$\frac{\mathtt{C}_1|S_1 \to \mathtt{C}_1'|S_2}{\mathtt{C}_1\,;\mathtt{C}_2|S_1 \to \mathtt{C}_1'\,;\mathtt{C}_2|S_2} \quad \text{(Seq)}$$

$$\mathtt{l = t}|S \to \mathtt{skip}|\mathtt{l} \mapsto t[S(\mathtt{l})/\mathtt{old.l}], S \quad \text{(Assign)}$$

$$\mathtt{skip;C}|S \to \mathtt{C}|S \quad \text{(Seq-next)}$$

$$\frac{mod(\mathtt{m}) = \mathtt{C}}{\mathtt{import\ m}|S \to \mathtt{C}|S'} \quad \text{(Import)}$$

図5.3 推論規則

Seq 計算の逐次実行を表す推論規則である. ; で結合されたコマンドは, 常に前から評価される.

Seq-next
コマンド列が次のコマンドに遷移するための規則. `skip;C` のとき, `skip` を無視して次の計算に進む.

Assign
`l = t` のコマンドは, 既存のストアを上書きするコマンドである. この時, `t` の中の `old.l` の出現を, 上書き前のストアの定義に全て置き換える.

Import
Import は, すでに定義されたモジュールをストアに取り込むための規則である. `mod` 関数は, `m` のモジュール定義本体を取り出す関数である. コマンド `import m` がストア S のもとで実行されると, モジュール `m` の定義本体 `C` をストア S のもとで評価した結果 S' にストアが書き換わる.

5.4 文法定義の例

図 5.4 は提案言語による文法定義と文法拡張の例である. 図 5.4 には `A` と `B` の 2 つのモジュールが定義されている.

```
module A {
  digit = '1' ⊕ .. ⊕ '0';
  num =  digit
      ⊕ (digit ⊗ num);
  add = num ⊗ '+' ⊗ expr;
  expr = add ⊗ num;
};
module B {
  import A;
  minus = num ⊗ '-' ⊗ expr;
  expr = old.expr ⊕ minus;
}
```

図5.4　言語定義の例

モジュール A の定義

モジュール A は 4 つの代入文を (;) で結合したものから構成される.

digit は数字の定義である. '0' から '9' までの数字を ⊕ で結合した形で定義されており, 0 から 9 までの整数 1 文字を受理する.

num は整数を定義している. 整数は 1 つ以上の連続する数字の列として定義される. num は再帰的に定義されており, digit のみからなる規則と, num の先頭に digit を ⊗ で結合した規則を ⊕ で結合した形で定義される.

モジュール B の定義

モジュール B は, expr に引き算を許すような拡張を定義する. モジュール B では, まずモジュール A をインポートする. このインポートした定義を上書きする形で文法を定義する.

minus は, expr に新たに追加する引き算を定義している. いま, minus は expr に組み込むことを想定しているので, 拡張後の expr 規則を利用したい. そのため, expr が出現する部分に old.expr を用いている.

同様に, expr はモジュール A の expr の定義に, 新たに minus を追加するため, old.expr ⊕ minus として定義されている.

第 6 章

実装

本章では, 提案言語の Scala への翻訳方法を示す. 構文解析器は Scala のパーザコンビネータライブラリを利用し, 言語の組み合わせは Scala の mixin 継承を用いた. また, 名前を用いた再帰にはクラスメンバの名前に依る参照を利用した.

6.1 提案言語の Scala への翻訳

構文解析の結果を表す値の表現には, Object Algebra を用いた. また, 構文解析器には scala-parser-combinator [5] ライブラリを用いた.

moduleDecl の翻訳

`moduelDecl A cmd` を以下のように翻訳する.

```
trait A { [|cmd|] }
```

import m の翻訳

コマンドに `import A` が含まれた時, 既存の構文解析の定義を展開する. モジュールを trait として表現したので, 継承によって定義を展開できる. モジュール X 中に `import A` のコマンドが含まれた場合, X は A を継承した trait として表される.

```
trait X extends A { ... }
```

⊕ の翻訳

⊕ は, 連続した二つの構文解析器を順に実行する意味を持つ. そのため `scala.util.parsing.combinator` パッケージの, | 演算子を用いた.

⊗ の翻訳

⊗ は，連続した二つの構文解析器を順に実行する意味を持つ．そのため scala.util.parsing.combinator パッケージの,~ 演算子を用いた．

old.<label> の翻訳

項 old.l は，拡張前に束縛されていた l に置換される．そのため，old.l は super.l に置き換えられる．

6.2　再利用可能な構文解析結果

```
trait IntAlg[A] {
  def lit(x : Int) : A;
  def add(e1 : A, e2 : A) : A;
}
```

図6.1　構文解析後の抽象構文木の定義

型安全で拡張可能なデータ型を実現する方法にはいくつかの実現方法がある．この拡張では，オブジェクト指向言語における実現方法として ObjectAlgebra を用いた．図 6.1 は整数と加算からなる数式の抽象構文木を表す Object Algebra の定義である．lit は整数 1 つを受取る Object Algebra インターフェース，add は Object Algebra インターフェースから構築された値 2 つを引数に取る Object Algebra インターフェースである．

図 6.2 は，整数と足し算からなる式の構文解析器を示した例である．各構文解析器は，IntAlgParser トレイトの中に宣言されている．

num は与えられた整数を表す文字列を Int 型の整数に変換する構文解析器である．litParser と addParser，expParser は，構文解析結果を IntAlg[a] を用いて構築する構文解析器である．

litParser は num の構文解析によって得られた整数を，IntAlg[a] が持つ Object Algebra インターフェース lit に渡して定数式を表す抽象構文木を作る．

addParser は定数式と数式が + で結合させた文字列を受理し,IntAlg[a] が持つ Object Algebra インターフェース add に渡して加算を表す抽象構文木を作る．+ の左の定数式を add の第一引数に，+ の右の数式を add の第 2 引数に渡して抽象構文木を作る．

expParser は加算式もしくは定数式を表す文字列を受理し，抽象構文木を IntAlg[a] を用いて構築する構文解析器である．addParser と litParser の選択 (|) を用いて定義している．

```
trait IntAlgParser extends RegexParsers {
  def num : Parser[Int]
    = """[0-9]+""".r ^^ ( x => x.toInt )
  def litParser[a](implicit f : IntAlg[a]) : Parser[a]
    = num ^^ { (x) => f.lit(x)}
  def addParser[a](implicit f : IntAlg[a]) : Parser[a]
    = (litParser ~ "+" ~ expParser) ^^ {
      case n ~ p ~ e => f.add(n, e)
    }
  def expParser[a](implicit f : IntAlg[a]) : Parser[a]
    = addParser | litParser
}
```

図6.2 構文解析器の定義

6.3 構文解析器の拡張例

図 6.2を拡張して, 新たに引き算を表す構文を追加する. 構文の追加とともに, 抽象構文木も変更され, 新たに引き算を表す項を定義する (図 6.3).

```
trait MinusAlg[A] {
  def minus(e1:A,e2:A) : A;
}
```

図6.3 抽象構文木を表す Algebra の拡張の定義

この結果を用いて, 引き算の式を受理し, 結果として MinusAlg によって作られた値を返す構文解析器を定義する.

図 6.4 は IntAlgParser を拡張し, 引き算を扱えるようにした MinusIntAlgParser の定義である. MinusIntAlgParser は IntAlgParser を継承したクラスで, 引き算を扱う構文解析器 minusParser と, expParser に対して引き算を扱えるようにする拡張の定義からなる.

minusParser の定義の右辺にある expParser は, 引き算の拡張を行った後の expParser が来てほしいので, そのまま expParser を用いている.

一方, expParser は old.exp ⊕ minus を super.expParser | minusParser に置き換えている.

```
trait MinusIntAlgParser extends IntAlgParser{
  def minusParser[a](implicit f : MinusAlg[a]) : Parser[a]
    = (litParser ~ "-" ~ expParser) ^^ {
      case n ~ "-" ~ e => f.minus(n,e)
    }
  override def expParser[a](implicit f : IntMinusAlg[a])
    = super.expParser(f) | minusParser(f)
}
```

図6.4　構文解析器の拡張の定義

このようにして言語定義の拡張を Scala のコードに翻訳できる.

6.4　提案言語の限界

提案言語では以下のことが出来ない

- 構文要素の削除
- 構文の選択規則の先頭, 末尾以外への挿入
- 左再帰の除去

第 7 章

MinCaml と XML の統合

本章ではプログラミング言語の統合の例を具体的に示す. MinCaml [7] に対して XML リテラルを導入する. XML の規格は大きいため, タグとタグ内のコンテンツのみからなる XML のサブセットを用いる.

MinCaml の構文解析器と XML の構文解析器をそれぞれ独立に定義し, 次に構文解析器の組み合わせを示す.

7.1 MinCaml の構文解析器と抽象構文木の定義

本節では, MinCaml の構文解析器とその抽象構文木の定義を示す.

7.1.1 syntax.scala

図 7.1 は, MinCaml の抽象構文木を定義するモジュールである. `SyntaxAlgebra` トレイトは, MinCaml の抽象構文木の Object Algebra による表現である. `Id.t` は型名を表す識別子, `Id.l` は変数名を表す識別子の型であり, それぞれ文字列として実装した.

7.1.2 type.scala

図 7.2 は, MinCaml の型に対する操作を定義したモジュールである.

7.1.3 camlparser.scala

図 7.3 – 7.5 に MinCaml の構文解析器の実装を示す. 住井の OCaml による MinCaml の実装の構文解析器の定義を基に, `exp` と `simple_exp` に左再帰の除去を施したのみで, 後の拡張のための変更は行っていない.

```
package syntax
import id.Id
import typename.Type

trait FunDefAlg[A]{
  type Syntax
  type TType
  def syntaxAlgebra : SyntaxAlgebra[Syntax]
  def typeAlgebra : Type.TypeAlgebra[TType]
  def fundef: (Id.t, TType, List[(Id.t, TType)], Syntax) => A
}

trait SyntaxAlgebra[Syntax]{
  type FunDef
  type TType
  def typeAlg   : Type.TypeAlgebra[TType]
  def funDefAlg : FunDefAlg[FunDef]
  def Unit      : Syntax
  def Bool      : Boolean => Syntax
  def Int       : Int => Syntax
  def Float     : Double => Syntax
  def Not       : Syntax => Syntax
  def Neg       : Syntax => Syntax
  def Add       : (Syntax, Syntax) => Syntax
  def Sub       : (Syntax, Syntax) => Syntax
  def FNeg      : Syntax => Syntax
  def FAdd      : (Syntax, Syntax) => Syntax
  def FSub      : (Syntax, Syntax) => Syntax
  def FMul      : (Syntax, Syntax) => Syntax
  def FDiv      : (Syntax, Syntax) => Syntax
  def Eq        : (Syntax, Syntax) => Syntax
  def LE        : (Syntax, Syntax) => Syntax
  def If        : (Syntax, Syntax, Syntax) => Syntax
  def Let       : (Id.t, Id.l, Syntax , Syntax) => Syntax
  def Var       : Id.t => Syntax
  def LetRec    : (FunDef , Syntax) => Syntax
  def App       : (Syntax, List[Syntax]) => Syntax
  def Tuple     : (List[Syntax]) => Syntax
  def LetTuple  : (List[(Id.t, TType)], Syntax, Syntax) => Syntax
  def Array     : (Syntax, Syntax) => Syntax
  def Get       : (Syntax, Syntax) => Syntax
  def Put       : (Syntax, Syntax, Syntax) => Syntax
}
```

図7.1　syntax.scala

```scala
package typename
object Type{
  trait TypeAlgebra[A]{
    def Unit : A
    def Bool : A
    def Int : A
    def Float : A
    def Fun : (List[A] , A) => A
    def Tuple : List[A] => A
    def Array : A => A
    def Var : Option[A] => A
  }
  // generate new type variable
  def genTyp[A](implicit f : TypeAlgebra[A]) : A  = {
    f.Var(None)
  }
}
```

図7.2 type.scala

7.2 XML のサブセットの構文解析器の定義

本節では, XML のサブセットとなる言語の構文解析器とその抽象構文木の定義を示す.

7.2.1 xml.scala

図 7.6 は, XML のサブセットとなる言語の構文解析器とその抽象構文木の定義が書かれたコードである.

7.3 MinCaml と XML の構文解析器の組み合わせ

図 7.7 に構文解析器の組み合わせを行うコードを示す.

`CamlXmlAlg` は, MinCaml の構文木と Xml の構文木を組み合わせた構文木である. Object Algebra を用いているため, `SyntaxAlgebra` と `XmlAlg` の 2 つを継承したクラスを新たに定義するだけで, 構文木の拡張が行える.

`Combined` は MinCaml と Xml の構文解析器を組み合わせたものである. MinCaml の式の中に XML 式を挿入する拡張を選択した. そのため, `exp_beta` を上書きし,

```
package camlparser
import scala.util.parsing.combinator._
import syntax._
class CamlParser extends RegexParsers {
  import typename._
  import id._
  def keywords: Parser[String]
    = ("if" | "then" | "else" | "not" | "true" | "false"
    | "let" | "in" | "rec" | "Array.create" | "Array.make"
    )
  def intParser: Parser[Int]
    = """(0|[1-9]\d*)""".r ^^ { _.toInt }
  def floatP : Parser[Double]
    = """[+-]?([0-9]*[.])?[0-9]+""" ^^{_.toDouble}
  def identP : Parser[String]
    = (not(keywords) ~> """[a-z]+""".r) ^^ { _.toString }
  def formal_args: Parser[List[String]]
    =  chainl1(identP ^^ {x=>List(x)}
                , "" ^^ {_ => (l:List[String],r:List[String]) => l++r})

  def fundef[A](implicit f : FunDefAlg[A]) : Parser[A]
    = identP ~ formal_args ~ "=" ~ exp(f.syntaxAlgebra) ^^
    {case fname ~ args ~ _ ~ body
     => f.fundef(fname,Type.genTyp(f.typeAlgebra)
                      ,args.map { s => (s,Type.genTyp(f.typeAlgebra)) }
                      ,body)}
  def funApp[A](implicit f : SyntaxAlgebra[A]) : Parser[A]
    =  exp_beta ~ actual_args ^^ { case e ~ arg => f.App(e,arg)}
  def actual_args[A](implicit f : SyntaxAlgebra[A]) : Parser[List[A]]
    = chainl1(simple_exp ^^ {x:A=>List(x)}
              , "" ^^ {_ => (l:List[A],r:List[A]) => l++r})
```

図7.3　camlparser.scala(1/3)

```
def exp_beta[A](implicit f : SyntaxAlgebra[A]) : Parser[A]
  =( simple_exp
       | ("not" ~> exp) ^^ { e => f.Not(e);}
       |  "-" ~> exp ^^ { e => f.Neg(e)}
       | ("if" ~> exp) ~ ("then" ~> exp) ~ ("else" ~> exp) ^^ {
         case p ~ tbody ~ fbody => f.If(p,tbody,fbody) }
       | ("let" ~> identP) ~ ("=" ~> exp) ~ ("in" ~> exp) ^^ {
         case i ~ e ~ b => f.Let(i,"",e,b) }
       | ("let"~"rec") ~> fundef(f.funDefAlg) ~ ("in" ~> exp) ^^ {
         case fundef ~ body => f.LetRec(fundef,body) }
  )
def exp[A](implicit f : SyntaxAlgebra[A]) : Parser[A]
  =(
        exp_beta ~ "+"  ~ exp ^^ {case l ~ _ ~ r => f.Add(l,r)}
       | exp_beta ~ "-"  ~ exp ^^ {case l ~ _ ~ r => f.Sub(l,r)}
       | exp_beta ~ "="  ~ exp ^^ {case l ~ _ ~ r => f.Eq(l,r)}
       | exp_beta ~ "<>" ~ exp ^^ {case l ~ _ ~ r => f.Not(f.Eq(l,r))}
       | exp_beta ~ "<"  ~ exp ^^ {case l ~ _ ~ r => f.Not(f.LE(r,l))}
       | exp_beta ~ ">"  ~ exp ^^ {case l ~ _ ~ r => f.Not(f.LE(l,r))}
       | exp_beta ~ "<=" ~ exp ^^ {case l ~ _ ~ r => f.LE(l,r)}
       | exp_beta ~ ">=" ~ exp ^^ {case l ~ _ ~ r => f.LE(r,l)}
       | exp_beta ~ "+." ~ exp ^^ {case l ~ _ ~ r => f.FAdd(l,r)}
       | exp_beta ~ "-." ~ exp ^^ {case l ~ _ ~ r => f.FSub(l,r)}
       | exp_beta ~ "*." ~ exp ^^ {case l ~ _ ~ r => f.FMul(l,r)}
       | exp_beta ~ "/." ~ exp ^^ {case l ~ _ ~ r => f.FDiv(l,r)}
       | funApp
       | exp_beta ~ ";" ~ exp ^^ {case l ~ _ ~ r => f.Let("?","()",l,r)}
       | exp_beta
  )
```

図7.4 camlparser.scala(2/3)

```
  def simple_exp_beta[A](implicit f: SyntaxAlgebra[A]) : Parser[A]
    = ( "(" ~ exp ~ ")" ^^ { case _ ~ x ~ _ => x}
      | "()" ^^ {case _ => f.Unit}
      | ("true"|"false") ^^ {case x => f.Bool(x.toBoolean)}
      | intParser ^^ {case n => f.Int(n)}
      | floatP ^^ {case n => f.Float(n)}
      | identP ^^ {case s => f.Var(s)})
  def simple_exp[A](implicit f: SyntaxAlgebra[A]) : Parser[A]
  = ( simple_exp_beta
    | simple_exp_beta ~ "." ~ "(" ~ exp ~ ")"
      ^^ { case e ~ _ ~ _ ~ e2 ~ _ => f.Get(e,e2) }
    )
}
```

図7.5　camlparser.scala(3/3)

```
package xml
import scala.util.parsing.combinator._
import scala.xml.XML
trait XmlAlg[A]{
  def Elem : (String,A) => A
  def Str : String => A
}
object XMLParser extends RegexParsers {
  def ident : Parser[String] = """[a-z]+""".r
  def stag : Parser[String] = "<" ~> ident <~ ">"
  def etag(ident:String) : Parser[String] = "</" ~> ident <~ ">"
  def element[a](implicit f : XmlAlg[a]) : Parser[a]
    = stag.into((id:String) => content ~ etag(id)) ^^ {
      case c ~ etag => f.Elem(etag,c)}
  def content[A](implicit f:XmlAlg[A]): Parser[A]
    = ident ^^ {(s)=>f.Str(s)}
}
```

図7.6　xml.scala

CamlParser の exp_beta に, 新たに XML の要素 element を追加する.

このように, 僅かな変更だけで構文解析器の拡張が行える.

```
package combined
import camlparser._
import xml._
import syntax._

trait CamlXmlAlg[A] extends SyntaxAlgebra[A] with XmlAlg[A]{}

trait Combined extends CamlParser with XMLParser{
  def exp_beta[A](implicit f : CamlXmlAlg[A]) : Parser[A]
   = super.exp_beta(f) | super.element
}
```

図7.7 combine.scala

第 8 章

関連研究

本章では, 本研究に関連する研究について触れる.

8.1 拡張可能なコンパイラ定義のフレームワーク

プログラミング言語の環境として, Spoofax や JastAdd[4] がある.

JastAdd は参照属性文法 (Reference Attribute Grammar) に基づいたメタコンパイルシステムである. JastAdd では抽象構文を定義するための専用のプログラミング言語を用いて文法を定義し, それをコンパイルすることで構文解析器と抽象構文木を表すクラス定義を含む Java ソースコードを生成する. 抽象構文木に対する操作はアスペクトとして定義する.

これらのシステムはコード生成器として実装されており、プリプロセッサがプログラムを生成する. そのため, 抽象構文の定義が変更されるたびに Java ソースコードを生成し, 再コンパイルが必要となる. また, 分割コンパイルも不可能である.

我々の手法は構文解析器の分割定義と再利用が可能であり, また言語内 DSL として実現した点で異なっている. 提案手法は言語内 DSL として実現可能であり, 利用するために新たなプログラミング言語を学ばなくて良いため, 利用者の学習にかかる時間を短縮できる点で有利だと考えられる.

8.2 拡張可能なデータ型の定義方法

本研究では, 構文解析結果の抽象構文木拡張可能にするために Object Algebra を用いた. 拡張可能なデータ型の定義方法には, Object Algebra の他にも様々な方法が提案されている.

本研究は Scala のようなオブジェクト指向言語を用いて実装を行ったため, Object Algebra を採用した. 関数型言語に対して拡張可能な構文解析器を定義する場合は拡張可能なデータ型として別の表現を考える必要がある. 例えば Haskell では, Data Types à la Carte[8] や Tagless Final Interpreter[1], OCaml においては Polymorphic Variant [3]

などがある. いずれも The Expression Problem の解決となっており, 関数の拡張とデータの拡張を安全に行うことができる.

8.3　パーザコンビネータ

Parsec[2] のようなパーサコンビネータライブラリは, 文法定義を細かなパーツに分けてそれらを組み合わせることで構文解析器を組み立てる.

パーサコンビネータは汎用的な構文解析器をライブラリとして提供しているが, プログラミング言語によく現れるパターンをライブラリ化してはいない.

本研究では, パーザコンビネータの結果を拡張可能なデータ型として表現することで, より複雑な文法構造もライブラリ化できることを示した.

第 9 章

結論・課題

本章では本研究の結論と今後の課題について述べる.

9.1 結論

本研究では、合成可能なパース規則を記述する言語を提案した.

この言語は, 言語定義のモジュール分割と言語言語定義の組み合わせを行う機能を持つ. 提案言語の形式化を行った.

また Scala による提案言語の実装を行い, 組み合わせ可能な構文解析器の定義をすることができるようになった.

9.2 今後の課題

この節では本研究の課題点を述べる.

9.2.1 より厳密な意味論の定義

提案言語における操作は文法定義と文法拡張を行うにとどまっており, 構文解析器が生成する値の拡張についての操作を含んでいない.

提案言語の実装においては拡張可能なデータ型を用いており, 構文解析器の組み合わせを行うと組み合わせに応じてデータ型の拡張もされなければならない.

構文解析器の拡張の挙動をより厳密にするためにも, 構文解析器の組み合わせに伴うデータ型の拡張を行うことは今後の課題と考えられる.

9.2.2 The Expression Problem との関係性についての調査

The Expression Problem で対象にした問題は, データの種類の拡張と, 拡張されたデータに対する関数定義を拡張する方法をどのように実現するかと言うものだった. The Expression Problem では, 拡張可能な値から計算結果を得る方法をどのように実現する

かというものだった. 本研究で取り組んだ問題は, 入力から拡張可能な値を生成する点で The Expression Problem とは異なる問題に取り組んでいる.

The Expression Problem の解決には, Catamorphism を用いた複数の解決が提案されている. Catamorphism は, 代数的データ型の畳み込み操作のことであり, 再帰的データ型の計算を定義する.

Catamorphism の双対として Anamorphism がある. Anamorphism は値から再帰的なデータ型を構成する計算である. 構文解析器は文字列から抽象構文木を生成する点で, Anamorphism の一種として定義できる可能性がある. より厳密な意味論の定義のためにも, 形式化の可能性を探ることは今後の課題である.

参考文献

[1] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, Vol. 19, No. 5, pp. 509–543, September 2009.

[2] Erik Meijer Daan Leijen. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical report, 2001.

[3] Jacques Garrigue. Programming with polymorphic variants. In *In ACM Workshop on ML*, 1998.

[4] Görel Hedin and Eva Magnusson. Jastadd: An aspect-oriented compiler construction system. *Sci. Comput. Program.*, Vol. 47, No. 1, pp. 37–58, April 2003.

[5] Typesafe Inc. scala-parser-combinatros. `https://github.com/scala/scala-parser-combinators`, 2016.

[6] Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, January 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

[7] Eijiro Sumii. Mincaml: A simple and efficient compiler for a minimal functional language. In *Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education*, FDPE '05, pp. 27–38, New York, NY, USA, 2005. ACM.

[8] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, Vol. 18, No. 4, pp. 423–436, July 2008.

[9] Philip Wadler. The expression problem. Java-genericity Mailing List, November 1998.