修 士 論 文

オブジェクト指向設計におけるデザインパターンの特定分野への適用に関する研究

指導教官　片山卓也 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

清水裕光

1997 年 2 月 14 日

目 次

第 1 章

序論

1.1 研究の背景

オブジェクトによって現実の世界を抽象化し、ソフトウェアを設計するオブジェクト指向設計の利点として部品の再利用が容易であるといわれている。しかし、オブジェクト指向に基づいて設計する際には、まずシステムを適切なオブジェクトに分割し、適切な粒度でクラスとしてまとめる。次にクラスのインタフェースや継承構造、そしてそれらの間の主要な関係を確立する必要がある。そのため、実際に再利用可能なソフトウェアを設計することには多くの困難を伴う。

オブジェクト設計には、多くの事例がある。これらの事例を調査し、繰り返し用いられている設計を体系化して利用できれば、オブジェクト指向設計をする際の再利用が容易になる。このように整理されたものはデザインパターンと呼ばれている [Gamma94]。

デザインパターンは、オブジェクト指向システムにおいて重要でかつ繰り返し現れる設計を、それぞれ体系的に名前付けし、説明を加え、評価した物である。デザインパターンを用いることによって、成功した設計の再利用が容易にできる。更に、デザインパターンはクラスやオブジェクト間の通信の仕様やそれの意図しているところを明確に示すので、システムの文書化や保守性の向上にも役立つ。

1.2 研究の目的

汎用部品としての個々のデザインパターンの有用性はある程度実証されているが、デザインパターンは抽象度が高く、実際のシステム設計に適用するための方法に関しては、十分な研究がなされていないのが現状である。そこで本研究では、特定のアプリケーションのためのフレームワークの構築を例とし、デザインパターンの適用に関する事例研究を行い、問題点を検討する。

1.3 本論文の構成

第2章では、フレームワークについて、及びデザインパターンを用いてフレームワークを設計することの意義と、予想される効果について説明する。第3章以降で、分野を特定して、デザインパターンを適用してフレームワークの構築を行い、第2章で述べた効果を確かめる。対象分野としては言語処理系を取り上げる。第3章では問題領域に関する説明を行い、フレームワークの設計において考慮すべき問題を整理する。第4章は、言語処理系の構成要素である字句解析器、構文解析器等についての、設計の方針、及び適用したデザインパターンについて説明する。また、第3章で考察した問題をどのように解決したかについて説明する。第5章は、第4章で構築したフレームワークの適用例を説明する。第6章は、第5章の適用例をもとに、第4章で構築したフレームワークの評価を行う。第7章でまとめと今後の課題を述べる。

第 2 章

デザインパターンとフレームワーク

この章では、フレームワークについて、及びデザインパターンを用いてフレームワークを設計することによって得られることについて説明する。

2.1 フレームワークとは

フレームワーク とは、ある特定のソフトウェアを対象にした再利用可能な設計プロダクトを構成するクラスの集合である [Deu89, JF88]。

フレームワークは、オブジェクト指向の考え方を取り入れたものの中では、現在最も再利用性の高い物である。

またフレームワークは、アプリケーションのアーキテクチャを形作る物である。すなわち、全体の構造、クラスとオブジェクトへの分割、主要な責任の割り付け、どのようにクラスとオブジェクトが協調するか、さらには制御スレッドなどを定義している。そのため、フレームワークは、アプリケーション分野に共通な設計上の決定事項を把握している。よって、フレームワークを用いてアプリケーションを構築することによって、コードに加えて、設計の再利用もできるようになる。その結果、より速くアプリケーションを構築できるようになり、さまざまなアプリケーションはいずれも似たアーキテクチャを有することになるため、保守しやすくなり、プログラマにとって、より一貫性が増す。

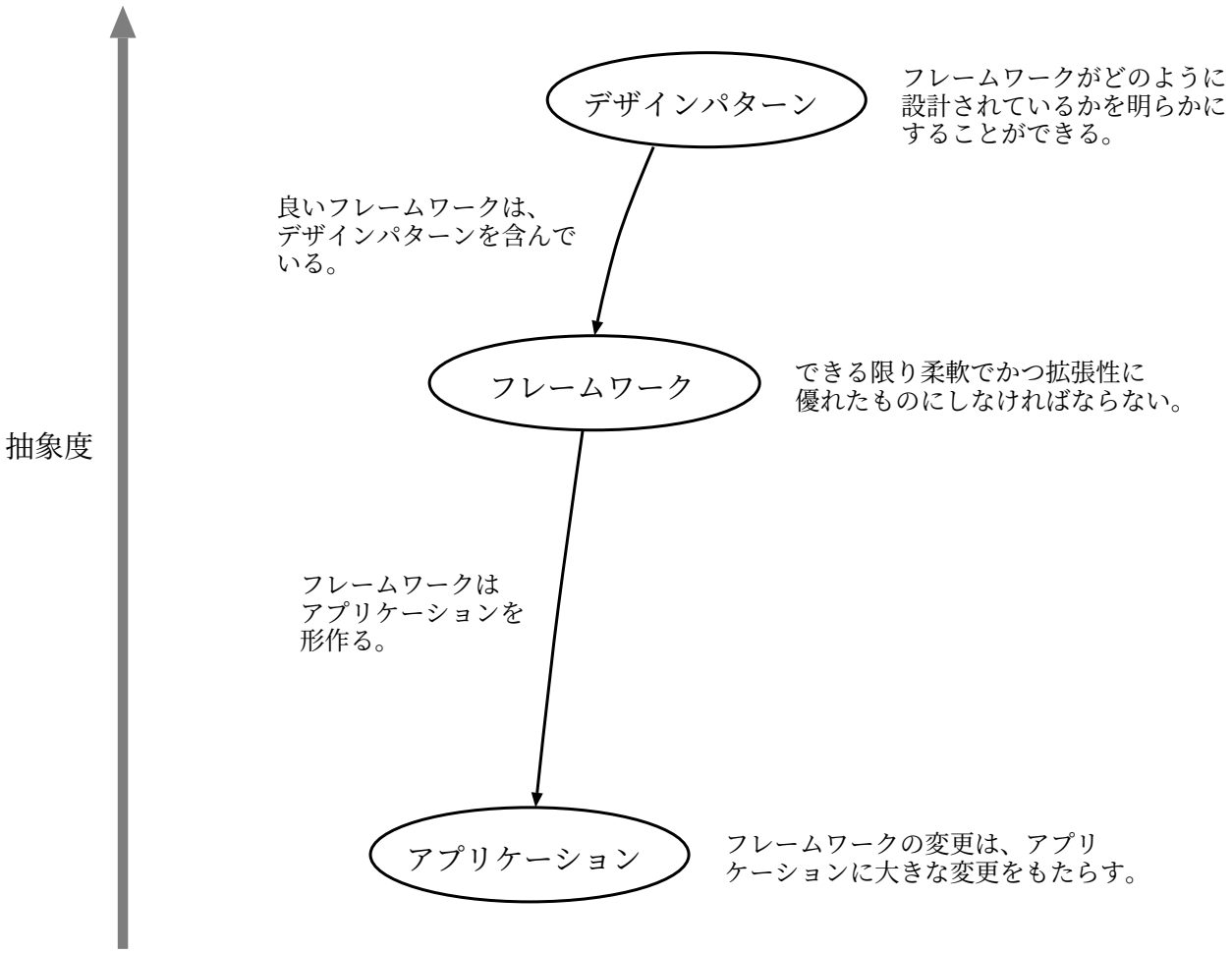

図 2.1: アプリケーションとデザインパターン、フレームワークとの関係

2.2 フレームワークを設計する上で注意すべき点

フレームワークを用いてアプリケーションを構築した場合、アプリケーションはフレームワークに依存しているので、フレームワークの変更によってアプリケーションに大きな変更をもたらす。したがってフレームワークは、できる限り柔軟でかつ拡張性に優れたものにし、フレームワークとアプリケーションの全ての結合度を低くしなければならない。これらのことから、フレームワークを設計するのは非常に難しい。

2.3 デザインパターンを用いてフレームワークを設計する

デザインパターンを用いて設計したフレームワークは、設計やコードの高度な再利用がしやすくなる。一般に、成熟したフレームワークはさまざまなデザインパターンを常に含んでいる。したがって、再設計なしでさまざまなアプリケーションに適合させるようなフレームワークの構築にはデザインパターンが多いに役立つ。

またフレームワークをデザインパターンで文書化した場合、パターンを知っている人はより早くそのフレームワークの構造を理解することができる。また、デザインパターンを知らない人にとっても、デザインパターンがフレームワークに加えた文書は理解の助けとなる。したがって、フレームワークとそれを構成する個々のクラスについて、それがどのように設計されているのかを明らかにすることができ、良く練られた設計についての情報を効率的に伝えことができる。

2.4 アプリケーションフレームワークを用いてアプリケーションを構築する

アプリケーションフレームワークとは、特定のアプリケーション領域に汎用的なソフトウェアシステムを包括しているようなフレームワークのことである [Pree95]。

問題領域に非依存であり、さまざまな分野で有用なフレームワークに GUI アプリケーションフレームワークがある。GUI アプリケーションフレームワークは、標準的な Look-and-Feel に必要とされる機構をほぼ備えた、再利用可能な空の (blank な) アプリケーションを提供する [Pree95]。アプリケーションフレームワークを用いてアプリケーションを構築する場合、フレームワークの抽象クラスと具象クラスを継承し、アプリケーションに適用させなければならない。

しかしながら、GUI アプリケーションフレームワークを用いて具体的なアプリケーションを構築しようとした場合、GUI アプリケーションフレームワークは抽象度が高く、対象となるアプリケーションに関する設計情報をほとんど持っていないため、プログラマが書かなければいけないコードの量は、決して少なくないのが現状である。このことは、GUI アプリケーションフレームワークはさまざまな問題領域に用いることができる有用なものであるが、実際のアプリケーションとの距離は、決して近くはないことを意味する。し

かしながら、GUI アプリケーションフレームワークを用いて構築できる、特定のアプリケーションのためのフレームワークをもちいることによって、プログラマの負担を軽くすることができる。なぜなら、特定のアプリケーションのためのフレームワークは、そのアプリケーションに関する設計情報を持っているからである。

そこで本研究では、特定のアプリケーションのためのフレームワークを、デザインパターンを用いて構築することにより、この特定のアプリケーション分野において、どのようにデザインパターンが適用できるかについて検討する。アプリケーション分野としては、言語処理系を考える。

次章では、言語処理系のフレームワークの構成について説明する。

第 3 章

言語処理系のためのフレームワークの構成

この章では、特定のアプリケーションのためのフレームワークの構成に関して議論する。対象とする分野は、図 3.1 で表せるような言語処理系とする。言語処理系を選んだ理由は、設計のための理論が確立しており、さまざまな分野への応用範囲が広いものであるからである。

はじめに言語処理系について説明し、実際にフレームワークを構築する上で、考慮しなければならない点について考察する。

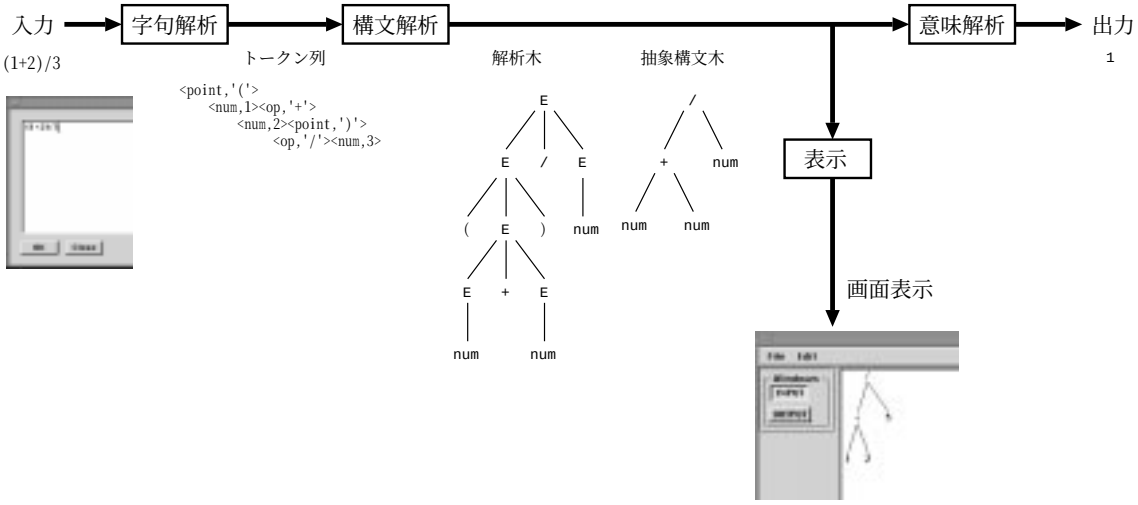

図 3.1: 対象とする言語処理系の構造

3.1 字句解析

字句解析器は、入力文字列を読み、トークンをなすレクシムに分け、トークン名と属性からなるトークンの列に変えて、構文解析器に渡す。

レクシムを表す正規表現から、有限オートマトンを作り、プログラム化することにより、字句解析器を作ることができる。

3.2 構文解析

構文解析器は、字句解析器から渡されるトークン列を文と見なして、それを文法にしたがって解析する。構文解析器の出力は **解析木** (parse tree) である。実際には解析木の作成を省略して直接、中間コードや目的コードを出力とする場合もあるが、ここでは解析木を作成する言語処理系を考える。また、意味解析 (3.3 節参照) や画面への出力のために、**抽象構文木** (abstract syntax tree) を、作成する。

構文解析の方法には、大きく分けて **下向き法** (top-down method) と **上向き法** (bottom-up method) があるが、ここでは下向き法を取り上げる。その理由として、人手で構文解析プログラムを作成するのに適しており、ほとんどのプログラミング言語は解析可能であることがあげられる。

下向き法は、文法の開始記号を根とする解析木から、生成規則を次々に適用することによって、最終的に文法記号を葉とする解析木を得る。下向き構文解析のできる文法として代表的なのが、**LL(k) 文法** である。k は、終端記号を k 個先読みすることを表す。ここでは $k = 1$ を用いる。また、本研究で構築する構文解析器は **再帰降下構文解析器** (recursive-descent parser) と呼ばれるものである。

3.3 意味解析

入力文字列に意味的誤りがないかを検査したり、言語の意味に関する情報を収集することを **意味解析** と呼ぶ。

意味解析の役割は、主に以下のようなものである。

(1) 名前[1]のスコープ (使用できる範囲) を解析し、名前の使用と宣言との対応をつける。

(2) 型の情報を集めたり、型の検査を行う。

(3) 制御の流れの検査を行う。

これらを通じて、意味解析では入力文字列の意味的正しさを検査し、必要となる情報を集める。

プログラミング言語で意味と呼ばれるものは二種類ある。一つは **静的意味** (static semantics) と呼ばれ、コンパイル時に分かるが文脈自由文法では表しきれない、あるいは表すと文法が大変複雑になるような性質のことである。もう一つは **動的意味** と呼ばれ、実行時になってはじめてわかる性質、例えば、0 で割ってはいけない、配列の添字が範囲を越えてはいけない、などの実行時の検査のことである。このうち意味解析で扱うのは静的意味である。

静的意味を定式化するための方法として代表的なものに **属性文法**がある。これは文脈自由文法と静的意味とを統合したものである。属性文法では、文脈自由文法の各文法記号に意味の情報を表す**属性** (attribute) を付加し、その属性に対する値の決め方を**意味規則** (semantic rule) として生成規則に付随させる。属性の値を計算することを **属性評価** (attribute evaluation) という。意味解析を行うプログラムを **意味解析器** (semantic analyzer) という。

3.4 言語処理系の設計問題

前節で、本研究で扱う言語処理系について説明した。この節では、実際に言語処理系のフレームワークを設計する上で、特に考慮しなければならない点について考察する。

- 文法の変更の容易性

  文脈自由文法の各文法記号は、終端記号と、非終端記号の二種類あり、それぞれ特有の性質を持っている。このことを考慮し、文法の変更を容易にできるようにする。

[1] 変数、定数、型、手続きなどの識別子

- 出力の柔軟性

  構文解析器の出力として、解析木を出力しなければならない。また、同時に抽象構文木も作成しなければならない。解析木や抽象構文木のノードは、言語によって任意に変化するが、解析木の作成の方法は一定である。よって、解析木はどのようなノードを作成する場合でも、一定の方法で行えるようにしなければならない。また抽象構文木の作成にあたっては、言語で扱うノードの種類を変えずに、木の作り方を変更するといった要求も起こり得る。したがって、抽象構文木の作成法を容易に変更できるようにしなければならない。

- 意味規則の利用

  文脈自由文法で表すのが難しい静的意味を定式化するための方法として、代表的なものに属性文法がある。一般に属性文法では、属性はさまざまな型をとり得る。例えば 5 章で示す例では、属性として変数名とその値を組とした型 (環境) を与えている。また、識別子の型の検査を必要とする言語では、識別子に付随する属性に型を表す値や型が正しいかを表す条件が必要となる。したがって、フレームワークを設計するにあたっては、属性の型を容易に変更できるようにしなければならない。

第 4 章

言語処理系のためのフレームワークの構築

この章では、字句解析器、構文解析器、意味解析器の設計において、適用したデザインパターンについて説明し、第 3 章で考察した問題をどのように解決したかについて説明する。

4.1 字句解析器の構築

トークンの共通の性質を定義する pToken クラス (図 4.1 参照) を定義する。アプリケーションでは扱うトークンの種類を基に、pToken クラスを継承することによって、必要なトークンの型を表すクラスを定義する。字句解析は Scanner クラスの Scan メソッドで行われ、pToken クラスのオブジェクトの並び (pTokenList クラスのオブジェクト) を出力する。リストの各要素は言語毎に定義されたトークンである ConcreteToken クラス[1]のオブジェクトである。

字句解析器の出力は pTokenList クラスのインスタンスである。pTokenList クラスのインスタンスは、pToken クラスのサブクラスのインスタンスを要素とするリストである。

[1] クラス名の先頭に “Concrete” の付くクラスは、アプリケーションにあわせて実装しなければならないクラスであり、アプリケーションによっては、複数のクラスになる。

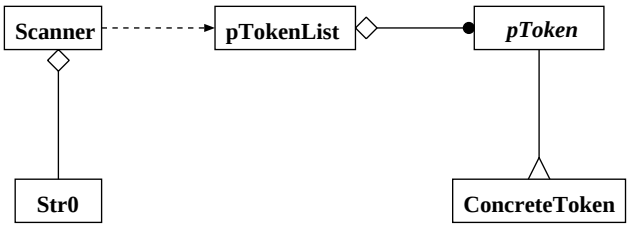

図 4.1: トークンと字句解析器

4.2 構文解析器の構築

4.2.1 抽象クラスと具象クラス

構文解析器を形成するクラスのクラス図は、図 4.2 である。

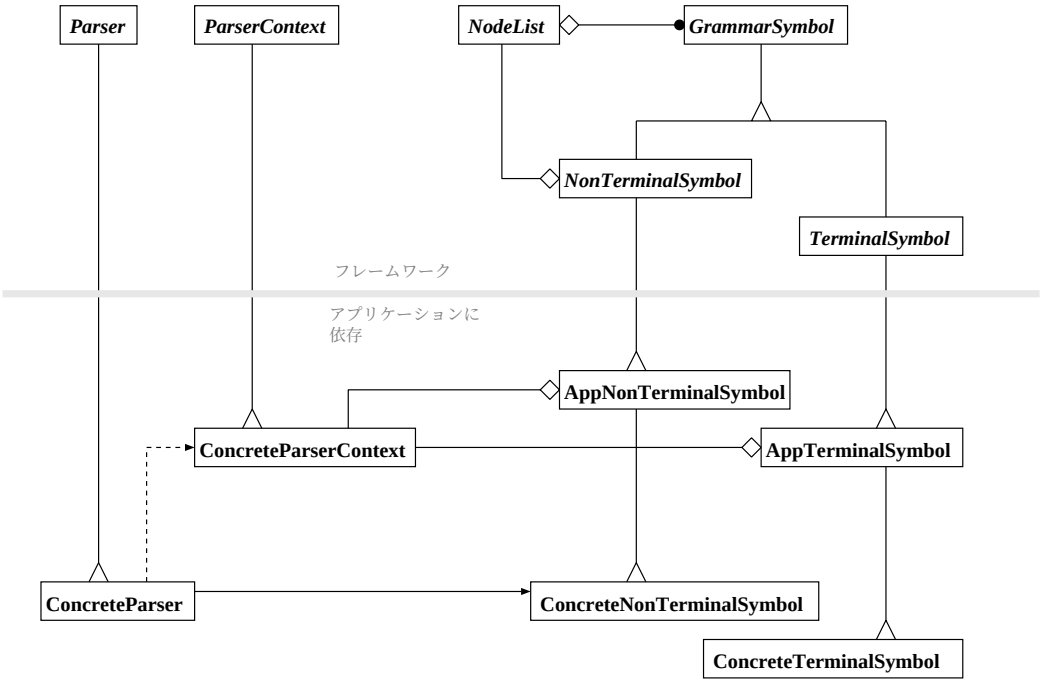

図 4.2: 構文解析器を形成するクラス

まず、以下のような抽象クラスを定義する。

Parser
構文解析器のメインを表すクラス。

ParserContext
構文解析中に必要な情報を定義するクラス。

GrammarSymbol
文法記号 (grammar symbol) の一般的性質を表すクラス。

NonTerminalSymbol
非終端記号 (nonterminal symbol) を表すノードの一般的性質を表すクラス。

TerminalSymbol
終端記号 (terminal symbol) を表すノードの一般的性質を表すクラス。

アプリケーションでは、これらのクラスを継承し、アプリケーションに適応させなければならない。図 4.2 のクラスの中で、アプリケーションに合わせて実装しなければならないクラスは以下の通りである。

ConcreteParser
構文解析器のメインを表すクラス。

ConcreteParserContext
構文解析中に必要な情報を定義するクラス。

AppNonTerminalSymbol [2]
非終端記号を表すノードの、アプリケーションで共通な性質を表すクラス。

AppTerminalSymbol
終端記号を表すノードの、アプリケーションで共通な性質を表すクラス。

ConcreteNonTerminalSymbol
非終端記号を表すノードの性質を表すクラス。

ConcreteTerminalSymbol
終端記号を表すノードの性質を表すクラス。

構文解析器の設計には Interpreter パターン (図 4.3) を用いている。

[2] クラス名の先頭に “App” の付くクラスは、構築するアプリケーションで共通な性質を表すクラスであり、アプリケーションにあわせて実装しなければならない。

Interpreter パターン

デザインパターン [Gamma94] の Interpreter パターンは、言語に対して、文法表現と、それを使用して文を解釈するインタプリタを一緒に定義することを目的としている [Gamma94]。

Interpreter パターンの構造は、図 4.3 のようになっている。

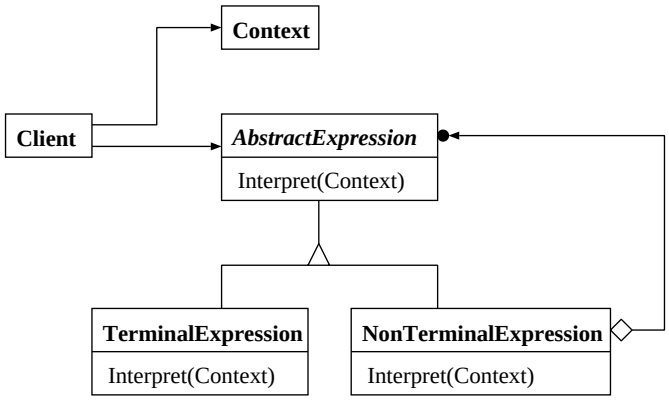

図 4.3: Interpreter パターン

Interpreter パターンでは、文法記号を表現するためにクラスを用いる。AbstractExpression クラスの各サブクラスの Interpret メソッドを定義することによって、インタプリタを作ることができる。

図 4.3 の構成要素は以下のようになっている。

- AbstractExpression クラス
  - 抽象構文木の全てのノードに共通な抽象化された Interpret メソッドを宣言する。
- TerminalExpression クラス
  - 文法中の終端記号に関する Interpret メソッドを実装する。
- NonTerminalExpression クラス
  - 文法中の非終端記号に関する Interpret メソッドを実装する。
- Context クラス
  - インタプリタにとって、グローバルな情報を含んでいる。

- Client クラス
  - 文法により定められた言語において、文を表現する抽象構文木を作る (または与えられる)。
  - Interpret メソッドの呼び出しを行う。

Interpreter パターンでは、文法規則を表現するためにクラスを用いているため、TerminalExpression クラス、または NonTerminalExpression クラスを継承することによって、文法を拡張することができる。

図 4.3 で現れるクラスと、構文解析器のクラスとの関係をまとめると、以下のようになる。

デザインパターンの Interpreter パターン	構文解析器のクラス
AbstractExpression	GrammarSymbol
TerminalExpression	TerminalSymbol
NonTerminalExpression	NonTerminalSymbol
Client	Parser
Context	ParserContext

これら構文解析器を形成する各クラスについては、以下の節で詳しく説明する。

4.2.2 解析木と抽象構文木の共通の性質を表すクラスの定義

解析木と抽象構文木には、いくつかの共通の性質がある。そのため、解析木と抽象構文木の一般的性質を表す Node クラスを定義する。

解析木を構成する文法記号を表すクラス、及び抽象構文木を表すクラスは、Node クラスを継承して定義する。

```
class Node : public Object
{
  protected:
    Node         *_father;    // ノードの親
    NodeList     *_sons;      // 子のノード
    int           _attr;      // ノードの種類の識別用
    const char   *_nodename;  // ノードの名前（表示用）
    NodeIterator *_nodeiter;  // ノードの走査用
    ValueList    *_in_vlist;  // 属性
    ValueList    *_out_vlist; // 属性評価後の結果
  public:
    virtual NodeIterator* CreateIterator(){}  // iterator の生成
    virtual void Evaluate(ValueList *, ValueList *){} // 意味解析を行なう
    Node* GetFather();    // 親のノードを得る
    Node* GetSon(int);    // 子のノードを得る
    virtual const char*   GetNodeName();    // ノードの名前を得る（表示用）
};
```

Evaluate は、意味解析を行うメソッドである (4.3 節参照)。

また CreateIterator メソッドは、子のノードを順に走査するための **iterator** を生成するメソッドである。

iterator とは、オブジェクト構造のアクセスしたり、構造中を走査したりする技法をサポートするためのものである [Gamma94]。iterator は、意味解析においては、属性を表すリスト構造や、解析木または抽象構文木といった木構造に対して必要となる。また、解析木や抽象構文木を画面に出力する場合にも必要となる。iterator の実装には、Iterator パターンが用いられる。Iterator パターンについては 4.4 節で説明する。

4.2.3 文法記号を表すクラスの生成

GrammarSymbol クラス

文法記号の一般的性質を定義するクラスである。文法記号は、非終端記号と終端記号をあわせたもので、これらを表すクラスは GrammarSymbol を継承し、それぞれ NonTerminalSymbol, TerminalSymbol クラスである。

GrammarSymbol クラスの宣言は以下のようになる。Parse メソッドは構文解析を行うメソッドである。

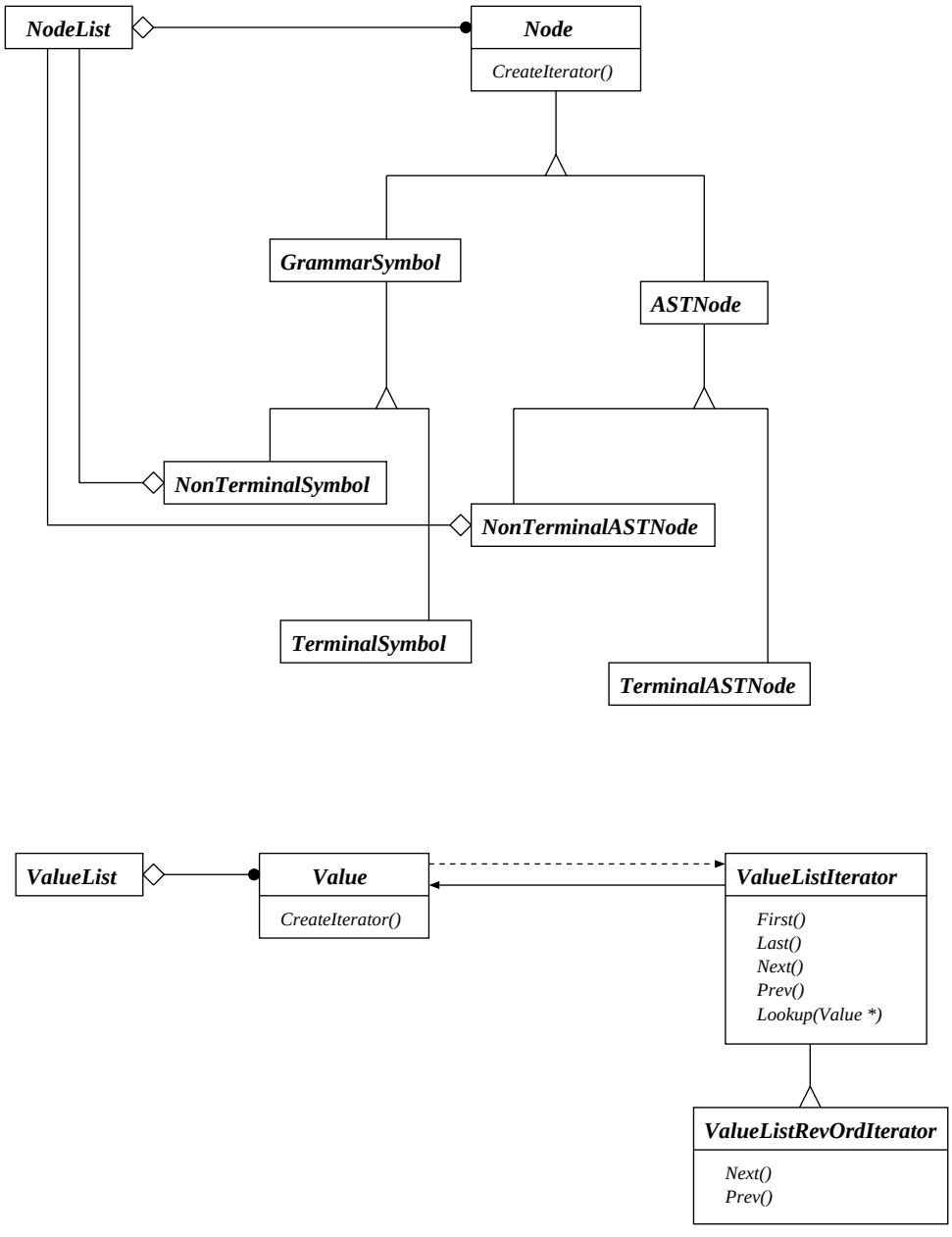

図 4.4: 解析木と抽象構文木の共通の性質を表すクラス

```
class GrammarSymbol : public Node
{
  public:
    // 構文解析を行う。
    virtual GrammarSymbol*         Parse(){};
    // iterator の生成
    virtual GrammarSymbolIterator* CreateIterator(){};
};
```

NonTerminalSymbol クラス

非終端記号 (nonterminal symbol) を表すノードの一般的性質を定義するクラスである。非終端記号とは、文法において、そこから終端記号の列が生成できるようなものである。プログラム言語での非終端記号は、言語のあるまとまった構成要素として一定の意味を持ち、構成要素間の階層構造を示すものが多い。

よって非終端記号では、生成規則にしたがって文法記号を表すクラスのオブジェクトを生成し、メンバー変数として GrammarSymbol クラスのオブジェクトのリストである GrammarSymbolList クラスのオブジェクトを持つ。

NonTerminalSymbol クラスの宣言は以下のようになる。

```
class NonTerminalSymbol : public GrammarSymbol
{
  public:
    GrammarSymbol*         Parse(){};
    GrammarSymbolIterator* CreateIterator();
};
```

TerminalSymbol クラス

終端記号を表すノードの一般的性質を定義するクラスである。終端記号とは、文法により生成される文 (sentence) を構成する基本単位となる記号である。プログラム言語では、終端記号は入力文字列の個々の文字ではなく、字句解析を行った結果、トークンとなるものである。

TerminalSymbol クラスの宣言は以下のようになる。終端記号であるため、子のノードを持たない。

```
class TerminalSymbol : public GrammarSymbol
{
  public:
    GrammarSymbol*         Parse(){};
    GrammarSymbolIterator* CreateIterator();
}
```

Parser クラス

構文解析器のメインとなるクラスで、Parse メソッドによって、構文解析を開始する。

Parser クラスの宣言、及びコンストラクタは以下のようになる。

```
class Parser
{
  protected:
    ParserContext  *_context;   // 構文解析中に必要な情報を定義
    GrammarSymbol  *_start;     // 開始記号を表す文法記号
    GrammarSymbol  *_gs;        // 解析木
  public:
    Parser(TokenList *);
    virtual GrammarSymbol* Parse();  // 構文解析を行う。
};


Parser::Parser(TokenList *tl)
{
    // 構文解析中に必要な情報を定義するクラスである
    // ParserContext クラスのオブジェクトを生成
    _context = new ParserContext(tl);
}
```

Parser クラスのコンストラクタで、構文解析中に必要な情報を定義している ParserContext クラスのオブジェクトを生成する。Parser クラスを継承する ConcreteParser クラスでは、開始記号を表す文法記号を表すクラスのオブジェクトを生成し、変数 *_start に代入する。このとき Parser クラスのコンストラクタで生成した、ParserContext クラスのオブジェクトを引数に渡すようにする。このようにすることによって、文法記号を表す全てのオブジェクトで、構文解析中に必要な情報を保持できる。

Parser クラスの Parse メソッドは以下のようになる。

```
GrammarSymbol* Parser::Parse()
{
    // 構文解析を開始。
    // start symbol の Parse メソッドを呼び出す。
    _gs = _start->Parse();

    // 出来上がった抽象構文木を得る (必要とする場合のみ)。
    _ast = _start->GetAST();

    // 解析木を返す。
    return (_gs);
}
```

まず、開始記号のオブジェクトの Parse メソッドを実行する。次に、抽象構文木を作成した場合は、開始記号のオブジェクトの GetAST メソッドを実行し、出来上がった抽象構文木を得る。最後に、構文解析後の解析木を返す。ほとんどの場合は、このような手順

で構文解析を行うことができるため、ConcreteParser クラスでは、Parse メソッドをオーバーライドする必要はほとんど無い。

ParserContext クラス

構文解析中に必要な情報を定義しているクラスである。

ParserContext クラスの宣言は以下のようになる。

```
class ParserContext
{
  protected:
    pToken         *_token;     // 現在処理中のトークン
    pTokenIterator *_next;      // トークンを走査する
    int             _attr;      // トークンの属性
  public:
    ParserContext(pTokenList *);
    pToken* GetToken();
    int     GetAttr();
};
```

ParserContext クラスのコンストラクタ、及びメソッドは以下のようになる。ParserContext クラスのオブジェクトは、構文解析を開始する前に Parser クラスのコンストラクタで作られ、構文解析を行う前に必要な手続きを行う。

```
ParserContext::ParserContext(pTokenList *tl)
{
    _next = new Iter(tl->MakeIterator());
    GetToken();
}

pToken* ParserContext::GetToken()
{
    _token = (pToken*)((*_next)());
    _attr = _token->GetAttr();
}

int ParserContext::GetAttr()
{
    return (_attr);
}
```

コンストラクタは pTokenList のオブジェクトを引数に取り、トークン列を走査する iterator を作る。次に GetToken メソッドを実行し、トークンを先読みする。GetToken メ

ソッドは、トークン列から次のトークンを取り、トークンの属性を表す変数 _attr をセットする。GetAttr メソッドは、現在のトークンの属性を返す。

また、GrammarSymbol クラスの具象クラスである ConcreteNonTerminalSymbol 及び ConcreteTerminalSymbol クラスの Parse メソッドは、文法にしたがって実装され、構文解析を行い解析木を生成する。抽象構文木を必要とする場合は、同時に生成する。

4.2.4 抽象構文木の構成

抽象構文木を構成するノードを表すクラスは図 4.5 のようになる。

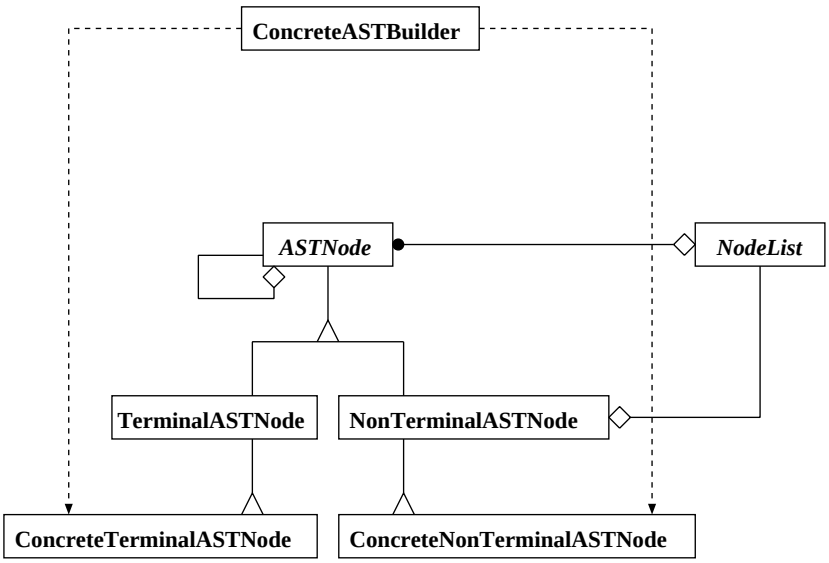

図 4.5: 抽象構文木のノード

図 4.5 のそれぞれのクラスについては、以下のようになっている。

ASTNode
抽象構文木のノードの一般的性質を定義した抽象クラス。

NodeList
Node クラス及びそのサブクラスのインスタンスのリスト。

NonTerminalASTNode
各非終端記号を表すノードの共通の性質を定義するクラス。

TerminalASTNode
各終端記号を表すノードの共通の性質を定義するクラス。

ConcreteNonTerminalASTNode
　各非終端記号を表すノードを定義するクラス。

ConcreteTerminalASTNode
　各終端記号を表すノードを定義するクラス。

ASTNode クラスの宣言は以下のようになる。

```
class ASTNode : public Object
{
  protected:
    Node         *_father;    // ノードの親
    NodeList     *_sons;      // ノードの子
    const char   *_nodename;  // ノードの名前（表示用）
    int           _attr;      // ノードの種類の識別用
    NodeIterator *_nodeiter;  // ノードの走査用
  public:
    virtual NodeIterator* CreateIterator();
};
```

NonTerminalASTNode クラスの宣言は以下のようになる。非終端記号を表すため、子のノードを持つ。

```
class NonTerminalNode : public ASTNode
{
  public:
    NodeIterator* CreateIterator();
};
```

TerminalASTNode クラスの宣言は以下のようになる。終端記号を表すため、子のノードを持たない。

```
class TerminalNode : public ASTNode
{
  public:
    NodeIterator* CreateIterator();
};
```

4.2.5　抽象構文木の生成

抽象構文木の生成には Builder パターンを用いる。Builder パターンは、多くの構成要素からなるオブジェクトを生成するアルゴリズムを、構成要素自体やそれらがどのように組み合わされるかということから独立しておきたい場合に適用することができる [Gamma94]。Builder パターンの構造は図 4.6 である。

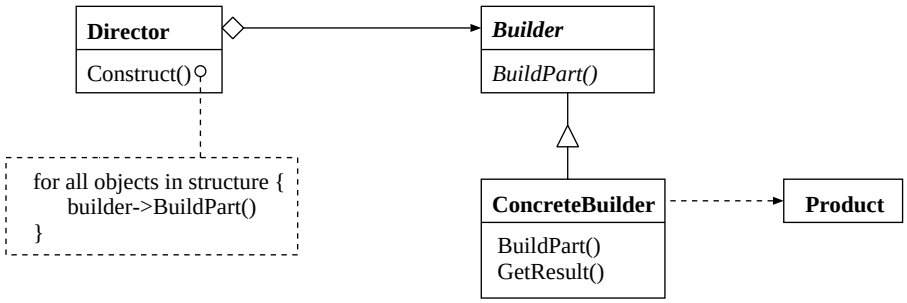

図 4.6: Builder パターン

図 4.6 の構成要素は以下のようになっている。

- Builder クラス
  - Product オブジェクトの構成要素を生成するための抽象化されたインタフェースを規定する。
- ConcreteBuilder クラス
  - Builder クラスのインタフェースを実装することで、Product オブジェクトの構成要素の生成や組合せを行う。
  - 自身が生成する表現を定義し、管理する。
- Director クラス
  - Builder クラスのインタフェースを使って、オブジェクトを生成する。
- Product クラス
  - 作成中の、多くの構成要素からなる複合オブジェクトを表す。ConcreteBuilder クラスは、Product オブジェクトの内部表現を作成し、また、それを組み立てる過程を定義している。
  - 構成要素を定義するクラス、及び構成要素を最終的な Product オブジェクトに組み合わせていくためのインタフェースを含んでいる。

図 4.6 で現れるクラスと、抽象構文木の作成部のクラスとの関係をまとめると、以下のようになる。

デザインパターンの Builder パターン	抽象構文木の 作成部のクラス
Builder	ASTBuilder
ConcreteBuilder	ConcreteASTBuilder
Director	ConcreteNonTerminalSymbol ConcreteTerminalSymbol
Product	ASTNode

また、文法記号を表すクラス及び抽象構文木の作成部のクラス図は、図 4.7 である。

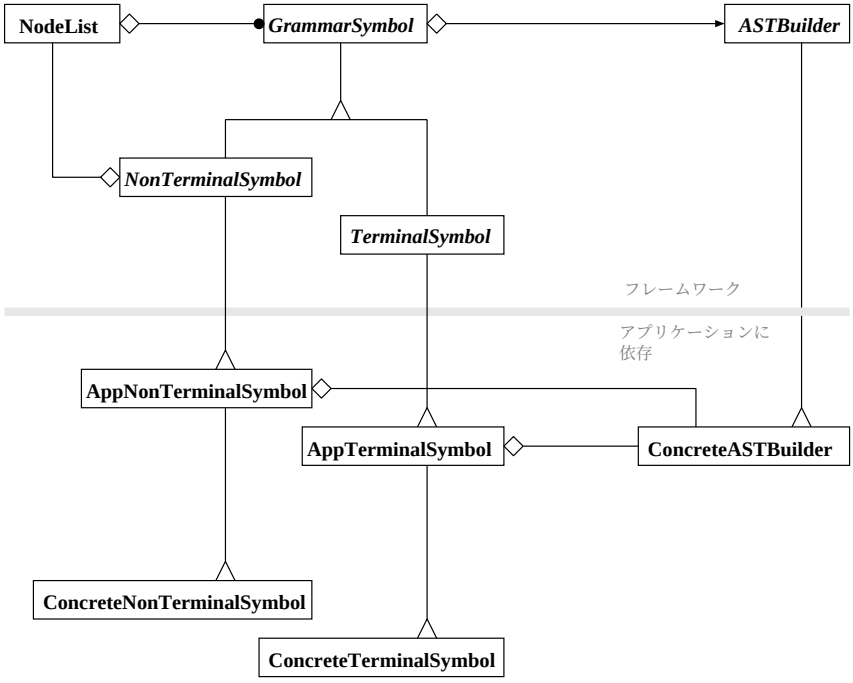

図 4.7: 文法記号を表すクラス及び抽象構文木の作成部

Builder パターンを用いることによって、抽象構文木の作成で用いている ASTBuilder クラスのサブクラスである ConcreteASTBuilder クラス (図 4.7 参照) を作り、文法記号を定義しているクラス (ConcreteNonTerminalSymbol, ConcreteTerminalSymbol クラス) から分離している。

抽象構文木を生成するために、GrammarSymbol クラスに以下のようなメンバー変数、及びメソッドが定義されている。

```
ASTNode    *_ast;          // 生成した構文木
ASTBuilder *_astbuilder;   // 構文木を生成する

ASTNode* GetAST() { return _ast; }
void     SetAST(ASTNode *node) { _ast=node; }
```

抽象構文木を生成するクラス (ConcreteASTBuilder クラス) の一般的性質を定義したクラスである ASTBuilder クラスを継承し、アプリケーションに合わせて ConcreteAST-Builder クラスを実装しなければならない。また、AppNonTerminalSymbol クラス、及び AppTerminalSymbol クラスのコンストラクタで、ConcreteASTBuilder クラスのオブジェクトが生成されるように定義する。

ASTBuilder クラス

抽象構文木を生成するクラス (ConcreteASTBuilder クラス) の一般的性質を定義したクラスである。ASTBuilder クラスの宣言は以下のようになる。

BuildNode メソッドは、非終端記号を表すノードを生成するメソッドである。BuildNode メソッドは、第 2 引数に Value クラスのオブジェクトを持ち、非終端記号を表すノードを表すクラスのメンバー変数とする。

```
class ASTBuilder
{
  public:
    // 子のノードを持たない場合
    virtual Node* BuildNode(int, Value *)=0;

    // 2 分木の場合 (BuildTree メソッドを呼び出す)
    virtual Node* BuildBinaryTree(int, Node *, Node *);

    // 通常
    virtual Node* BuildTree(int, NodeList *)=0;
};
```

抽象構文木の部分木は、2 分木であることが多い。そのため、2 分木の抽象構文木を生成するメソッドである BuildBinaryTree メソッドを定義する。BuildBinaryTree メソッドは、BuildTree メソッドを呼び出す。

```
Node* ASTBuilder::BuildBinaryTree(int op, Node *n1, Node *n2)
{
    NodeList *sons = new NodeList();
    sons->Add(n1);  // 子のノードリストに追加
    sons->Add(n2);  // 子のノードリストに追加
    return ( BuildTree(op,sons) );
}
```

4.3 意味解析器の構築

意味解析は、解析木または抽象構文木に属性を付加し、属性評価を行うことによって行う。属性はインスタンス変数として、Value クラスのサブクラスのオブジェクトを持つクラスとして定義される。

Value クラスは、意味解析で扱う型の一般的性質を定義したクラスであり、意味解析で扱う型を表すクラスは全て Value クラスを継承することによって、定義される。例えば 5 章で示す例では、変数名とその値を組とした環境を表すために、Value クラスを継承して EnvValue クラスを定義している (5.5 節参照)。また、型の検査を必要とする言語では、型を表す値 (TypeValue クラス) や、型が正しいかを表す真偽値 (BoolValue クラス) を、属性を表すクラス (AttrValue クラス: TypeValue クラスと BoolValue クラスのインスタンスをメンバーとして持つ) 等を Value クラスを継承して定義する。

解析木または抽象構文木の意味解析にも、Interpreter パターン (図 4.3 参照) が有用である。

属性評価を行うために、GrammarSymbol クラス、及び ASTNode クラスには、以下のように属性と、属性評価後の結果を格納するメンバー変数がある。

```
ValueList *_in_vlist;  // 属性
ValueList *_out_vlist; // 属性評価後の結果
```

また、GrammarSymbol クラス、及び ASTNode クラスには、以下のように Evaluate メソッドが宣言されている。

```
virtual void Evaluate(ValueList *, ValueList *);
```

Evaluate メソッドの第 1 引数は属性、第 2 引数は評価後の結果である。

解析木に対して意味解析を行う場合は、解析木を構成するクラスである ConcreteNonTerminalSymbol 及び ConcreteTerminalSymbol クラス (図 4.7 参照) の Evaluate メソッドを

実装し、抽象構文木に対して行う場合は、抽象構文木を構成するクラスである ConcreteNonTerminalASTNode 及び ConcreteTerminalASTNode クラス (図 4.5 参照) の Evaluate メソッドを実装する。また、意味解析を行う場合、解析木や抽象構文木のノードを順に走査する必要がある。ノードの走査には、Iterator パターン (4.4 節参照) を用いている。

4.4 解析木 (抽象構文木) 表示部の構築

解析木または抽象構文木の表示部は、GrammarSymbol または Node クラスの iterator を定義し、表示部で木を順に走査することによって行う。

Iterator パターン

Iterator パターンは、集約オブジェクトが基にある内部表現を公開せずに、その要素を順にアクセスする方法を提供することが目的である [Gamma94]。Iterator パターンの構造は図 4.4 である。

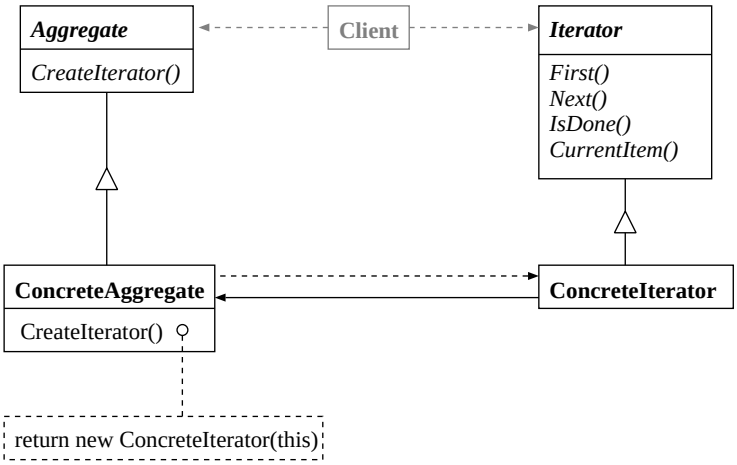

図 4.8: Iterator パターン

本研究で構築した言語処理系のためのフレームワークでは、既に挙げた通り、属性を表すクラスのオブジェクトのリストを表すクラス (ValueList クラスのサブクラス)、及び解析木 (GrammarSymbol クラスのサブクラス) または抽象構文木 (Node クラスのサブクラス) に対して意味解析や画面への表示を行うために、解析木のノードを表すクラスのオブ

ジェクトのリスト (GrammarSymbolList クラスのオブジェクト) や、抽象構文木のノードを表すクラスのオブジェクトのリスト (NodeList クラスのオブジェクト) を走査する必要がある。

これらリストのような集約オブジェクトは、内部構造を明かすことなく、要素にアクセスする方法を提供するべきである。更に、あるリストに対して何をしたいのかにより、リストを異なった方法で走査したくなるかもしれない。しかし、異なる走査ごとにメソッドを用意して、リストを表すクラスのインタフェースをふくらませていくことは、避けるべきである。

Iterator パターンでは、リストオブジェクトから、アクセスや走査のための責任を抜きだして、これを **iterator** オブジェクトに与える。iterator を定義しているクラスでは、リストの要素にアクセスするためのインタフェースを定義している。また iterator は、走査を記録しておく必要がある。すなわち iterator は、どの要素が既に走査されたのかを知っている。

図 4.4 の構成要素は、以下のようになっている。

- Iterator クラス
  - 要素のアクセスしたり走査したりするためのインタフェースを定義する。
- ConcreteIterator クラス
  - Iterator クラスで定義したインタフェースを実装する。
  - Aggregate オブジェクトの走査の際に、カレント要素を記録する。
- Aggregate クラス
  - Iterator オブジェクトを生成するためのインタフェースを定義する。
- ConcreteAggregate クラス
  - Aggregate クラスで定義したインタフェースに対して、適切な ConcreteIterator クラスのオブジェクトを生成して返すように実装する。

図 4.8 で現れるクラスと、フレームワークのクラスとの関係をまとめると、以下のようになる。

デザインパターンの Iterator パターン	フレームワークの クラス
Iterator	NodeIterator
	ValueListIterator
ConcreteIterator	NonTerminalNodeIterator TerminalNodeIterator
	ValueListRevOrdIterator
Aggregate	ASTNode
	GrammarSymbol
	ValueList
ConcreteAggregatet	ConcreteNonTerminalASTNode ConcreteTerminalASTNode
	ConcreteNonTerminalSymbol ConcreteTerminalSymbol

またクラス図は、図 4.9 のようになる。

フレームワークの各クラスについては、以下の通りである。

NodeIterator, ValueListIterator クラスは、それぞれ ASTNode, GrammarSymbol, ValueList クラスのオブジェクトの各要素にアクセスしたり走査したりするためのインタフェースを定義している。NodeIterator, ValueListIterator クラスのオブジェクトは、それぞれ ASTNode, GrammarSymbol, ValueList クラスのサブクラスの CreateIterator メソッドによって生成される。

解析木、及び抽象構文木には、非終端記号と終端記号の 2 つがある。そのため NodeIterator クラスを継承し、非終端記号用の iterator の性質を定義した NonTerminalNodeIterator クラスと、終端記号用の iterator の性質を定義した TerminalNodeIterator クラスを実装する。非終端記号には子のノードが存在しないため、子のノードを順に走査するメソッドである Next メソッドが呼ばれたとき、これ以上走査できないことを表す 0 を返すように実装されている。

4.3 節で既に説明した通り、属性を表すクラスは、Value クラスを継承して定義する。そのため、プログラム言語によっては ValueList クラスのオブジェクトを、要素の最後から逆順に走査できるようにする必要がある。このような走査を行うために、ValueListIterator

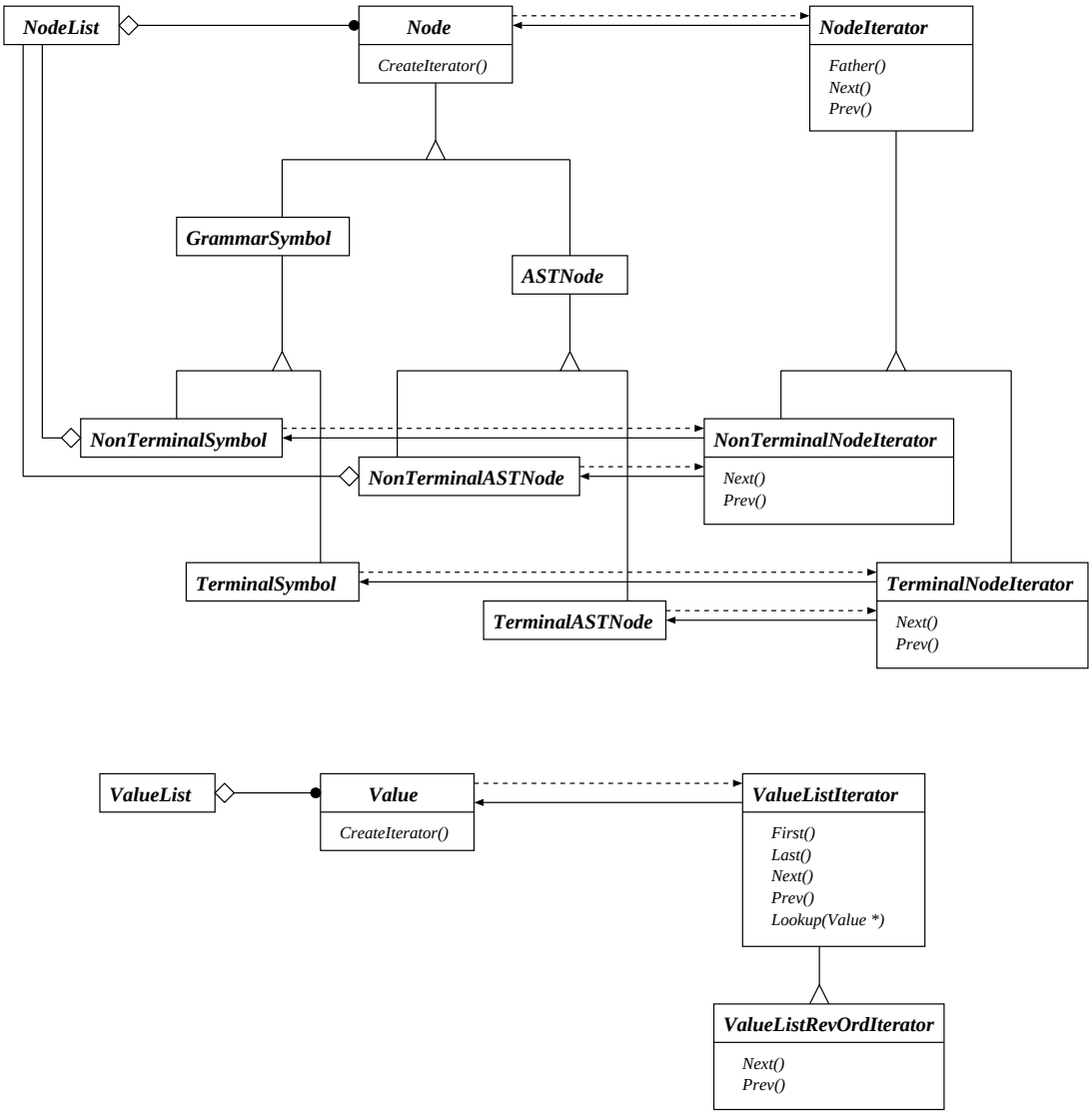

図 4.9: Iterator パターンの適用

クラスを継承し、ValueListRevOrdIterator クラスを定義している。また、属性解析後の結果は ValueList クラスのオブジェクトであるが、実際に必要なのは一番最後の要素のみという場合もある。このため、ValueListIterator クラスには、一番最後の要素を取り出す Last メソッドを実装している。

したがって、同じ集約オブジェクトに対して、異なった走査を行う場合、NodeIterator, ValueListIterator を継承して、新しいクラスを作り、CreateIterator メソッドで、新たに作ったクラスのオブジェクトを生成することによって、集約オブジェクトの内部構造を変

更することなく、走査方法の変更が可能である。

4.5 まとめ

言語処理系のためのフレームワークの構築で、以下のようなパターンを適用した。

- 言語に対して、文法表現と、それを使用して文を解釈するインタプリタを一緒に定義する **Interpreter パターン** によって、終端記号と、非終端記号のそれぞれの性質を定義したクラスを構築し、文法記号を表すクラスはこれらのクラスを継承して定義することによって、文法の変更を容易にし、意味解析を行う方法を提供した。
- **Iterator パターン**によって、属性を表すリストや、解析木、抽象構文木といった集約オブジェクトへアクセスしたり走査する責任を分離し、集約オブジェクトの内部構造を明かすことなく走査し、集約オブジェクトの内部構造を変更することなく、走査方法の変更を可能にした。
- **Builder パターン**は、抽象構文木の作成部を文法記号を表すクラスから分離し、変更を容易にした。

また、意味解析で扱うクラスを抽象化することによって、属性を表す型の変更が容易になることを示した。

したがって構築したフレームワークは、第 3 章で考察した、言語処理系のフレームワークを設計する上で、特に考慮しなければならない点を全て解決することが分かった。

第 5 章

フレームワークの適用例

以下のような簡単な仕様の言語を例に挙げ、第 4 章で構築したフレームワークの適用例を説明する。

5.1 言語の仕様

言語の仕様としては、フレームワークの全てのテストができる最小の例を考える。
言語は、以下のような仕様を持っており、算術式の評価を行う。

- 数値は整数型のみ。
- 演算は $+, -, *, /$ の 4 つ。
- let 式で束縛された変数は、let 式の本体でのみ出現する (自由変数は無い)。

5.2 アプリケーションで扱うデータ型

4.3 節で既に説明した通り、構文解析で扱う型は全て Value クラスを継承して定義する。
このアプリケーションでは、Value クラスを継承して IntValue, VarValue, EnvValue クラスを定義している。
IntValue クラスは、整数値を表す型であり、ConstNode クラス (図 5.4 参照) は、IntValue クラスのオブジェクトをメンバーとして持っている。また VarValue クラスは、変数名を

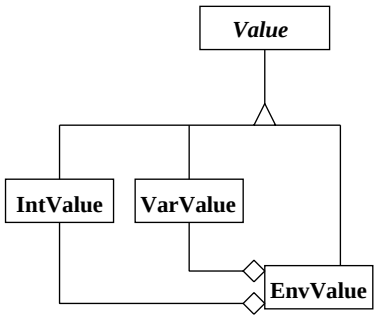

図 5.1: 扱う型を定義するクラス

表す型であり、VarNode クラス (図 5.4 参照) は、VarValue クラスのオブジェクトをメンバーとして持っている。

EnvValue クラスは、5.5 節で説明する、変数名とその値を組とした **環境** を表すクラスである。

5.3 字句解析器の作成

5.3.1 文法

言語の構文解析器の文法は、以下のようになっている。

$$
\begin{aligned}
E &\rightarrow T\,\{(+ \mid -)\,T\} \\
&\mid \text{let}\ \ DL\ \ \text{in}\ \ E \\
T &\rightarrow F\,\{(* \mid /)\,F\} \\
DL &\rightarrow D\,\{,\,D\} \\
D &\rightarrow \mathbf{ID}\ = E \\
F &\rightarrow \underline{(}\,E\,\underline{)} \mid \mathbf{ID} \mid \mathbf{CONST}
\end{aligned}
$$

ここで、{ } は、0 回以上のくりかえしを表すメタ記号である。また、終端記号の (と) をメタ記号と区別するため、終端記号の ( ) には下線を引いてある。

この文法を基にトークンの種類を決定する。トークンの種類は、表 5.1 のようになっている。

表 5.1: トークン

トークンの種類	レクシムの例	トークン名	パタン	属性
空白、改行	(空白)	(無視)	(空白 | 改行)$^+$	
識別子 (名前)	a b	**ID**	**letter** (**letter**|**digit**)*	文字列
数 (整数)	123	**CONST**	**digit**$^+$	整数値
キーワード	let in	**KW**	let in など	キーワード名
演算子	$+$ $-$ $*$ / など	**OP**	$+\|-\|*\|/$	演算子そのもの
句読点	( ) など	**POINT**	( ) など	句読点そのもの
入力文字列の終り		**EOL**		なし

$$\textbf{letter} \rightarrow a|b|\cdots|z|A|B|\cdots|Z$$
$$\textbf{digit} \rightarrow 0|1|\cdots|9$$

この表 5.1 を基に pToken クラスを継承し、トークンの性質を表すクラスを定義する。トークンの種類と、対応するクラスを表にまとめると以下の表のようになる。

トークンの種類	クラス名
識別子	IDToken
キーワード	KWToken
数 (整数)	ConstToken
演算子 句読点	TSToken

この表を基に字句解析を行う Scanner クラスを実装する。字句解析器の出力は pTokenList クラスのインスタンスである。

トークンと字句解析器を形成するクラスのクラス図は、図 5.2 のようになる。

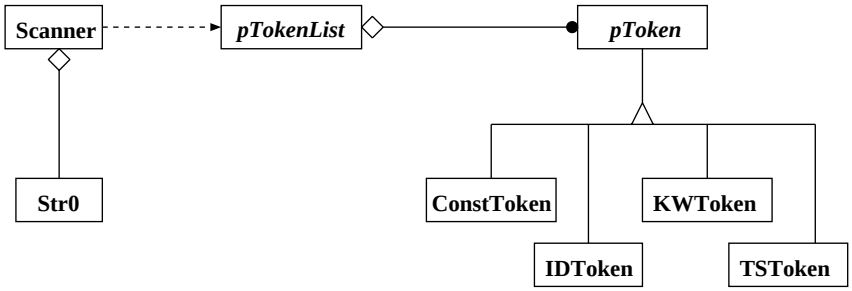

図 5.2: トークンと字句解析器

5.4 構文解析器の作成

5.4.1 構文解析器を形成するクラスの実装

図 4.2 の ConcreteParser, ConcreteParserContext, ConcreteNonTerminalSymbol, ConcreteTerminalSymbol クラス、及び図 4.7 の ConcreteASTBuilder クラスを実装する。実装後のそれぞれのクラス名は、以下の表のようになる。

クラス名		対応する
フレームワーク	アプリケーション	文法記号
ConcreteParser	CalcParser	—
ConcreteParserContext	CalcParserContext	—
ConcreteASTBuilder	CalcASTBuilder	—
ConcreteNonTerminalSymbol	ESymbol	E
	TSymbol	T
	DLSymbol	DL
	DSymbol	D
	FSymbol	F
ConcreteTerminalSymbol	IDSymbol	ID
	ConstSymbol	$CONST$
	OPSymbol	なし

CalcParser クラスの宣言は、以下のように Parser クラスを継承することによって、非常にシンプルになる。

CalcParser クラスのコンストラクタで、Parser クラスのコンストラクタを呼び出し、開始記号を表す文法記号を定義しているクラスのオブジェクトを生成する。開始記号は E であるため、ESymbol クラスのオブジェクトを生成する。

```
class CalcParser : public Parser
{
  public:
    CalcParser::CalcParser(TokenList *tl) : Parser(tl)
      {
          // 開始記号を表す文法記号を定義しているクラスの
          // オブジェクトを生成する。
          _start = new ESymbol();
      }
};
```

文法記号を表すクラスである ConcreteNonTerminalSymbol 及び ConcreteTerminalSymbol クラスの Parser メソッドを実装する。例えば TSymbol, FSymbol クラスの Parse メソッドは、文法にしたがって以下のように実装される。

```
GrammarSymbol* TSymbol::Parse(ParserContext *context)
{
    // FSymbol のインスタンスを生成し、構文解析を行い、
    // 子のノードに追加
    _gslist->Add( ( new FSymbol() )->Parse(context) );

    while ( (_attr == '*') | (_attr ==  '/') )
      {
          _gslist->Add( new OPSymbol(_attr) ); // '*' または '/' を
                                               // 子のノードに追加
          GetToken();  // トークン列からトークンを読む
          _gslist->Add( ( new FSymbol() )->Parse(context) );
      }
    return (this);
}

GrammarSymbol* FSymbol::Parse(ParserContext *context)
{
    switch (_attr)     // トークンの種類を調べる
      {
        case '(':
          _gslist->Add( new OPSymbol(_attr) );
          GetToken();
          _gslist->Add( ( new ESymbol() )->Parse(context) );
          _gslist->Add( new OPSymbol(_attr) );  // _attr==')'
          GetToken();
        case eID:      // 識別子の場合
          _gslist->Add( new IDSymbol( (IDToken *)_token) );
          GetToken();  // トークン列からトークンを読む
          break;
        case eCONST:   // 整数の場合
          _gslist->Add( new ConstSymbol( (ConstToken *)_token ) );
          GetToken();  // トークン列からトークンを読む
          break;
      }
    return (this);
}
```

またクラス図は、図 5.3 である。

5.4.2 抽象構文木のノードを表すクラスの定義

ConcreteNonTerminalNode, ConcreteTerminalNode クラス (図 4.5 を参照) を実装する。それぞれのクラス名は表のようになる。

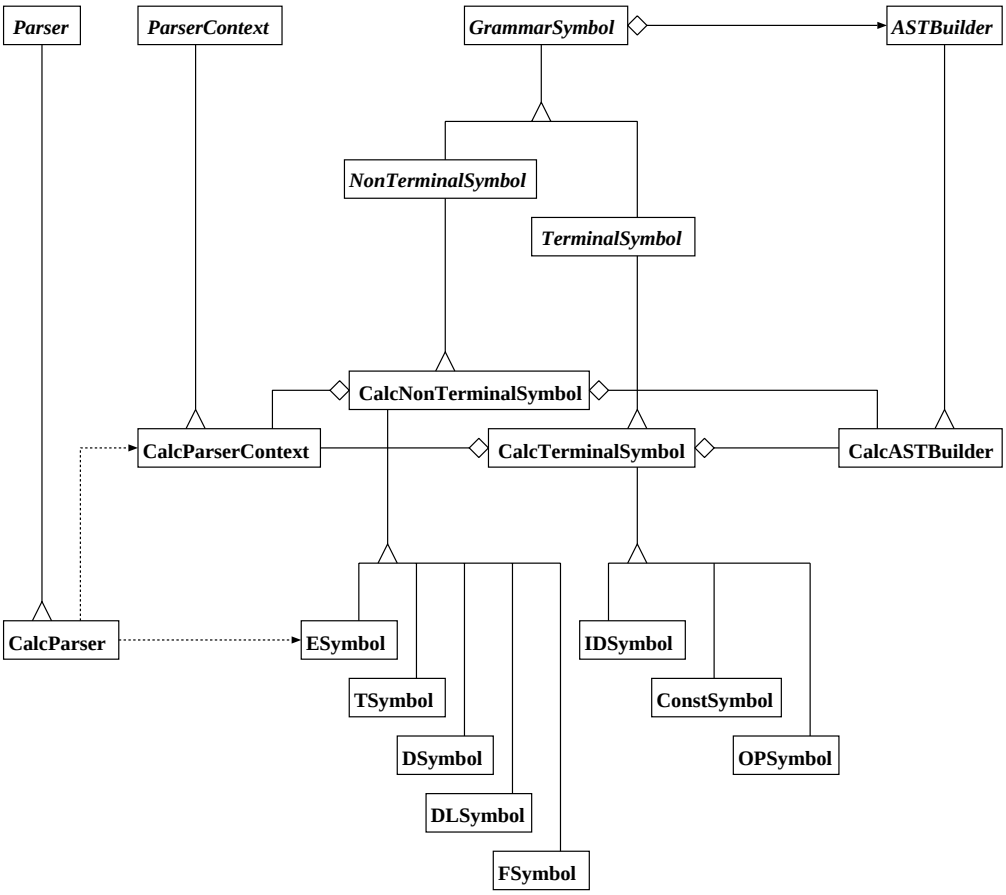

図 5.3: 構文解析器及び構文木の作成部

クラス名	
フレームワーク	アプリケーション
ConcreteNonTerminalNode	LetNode DeclListNode DeclNode ExpNode
ConcreteTerminalNode	VarNode ConstNode

また実装後のクラス図は、図 5.4 である。

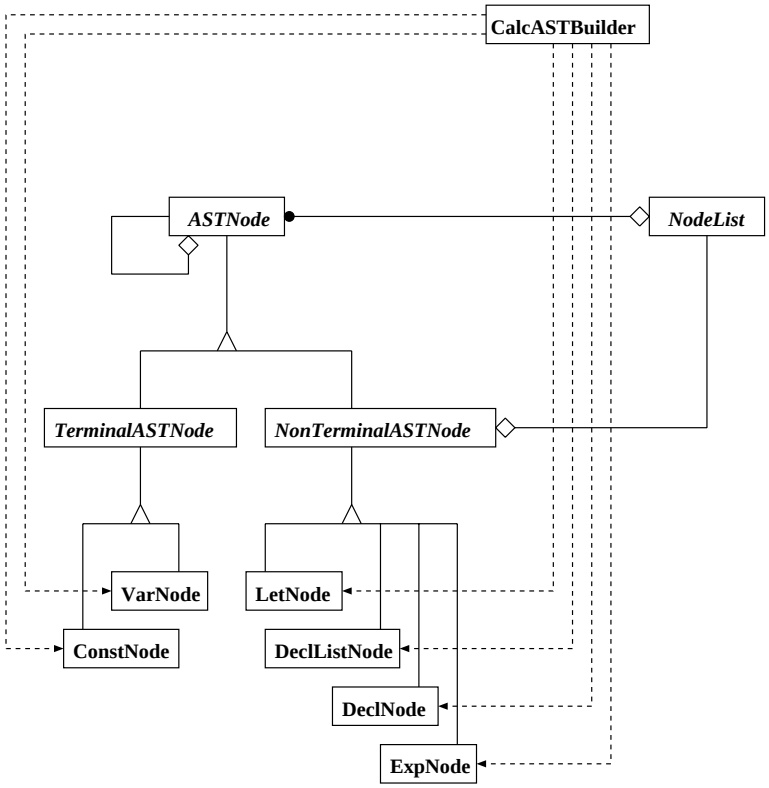

図 5.4: 抽象構文木

5.4.3 抽象構文木の生成

文法記号を表すクラスの Parse メソッドから、CalcASTBuilder クラスの BuildNode, BuildBinaryNode, BuildTree メソッドを呼び出し、構文木を生成する。BuildBinaryTree 及び BuildTree メソッドは、構文木の非終端ノードを表すクラスである LetNode, DeclListNode, DeclNode, ExpNode クラスのインスタンスを返し、BuildNode メソッドは、構文木の終端ノードを表すクラスである ConstNode, VarNode クラスのインスタンスを返す。

抽象構文木を生成する場合は、TSymbol クラスの Parse メソッドは以下のようになる。

```
GrammarSymbol* TSymbol::Parse(ParserContext* context)
{
    FSymbol *f;                    // 生成した FSymbol のインスタンス
    Node *n1,*n2;                  // 構文木のノード (一時保管用)

    f = new FSymbol();             // FSymbol のインスタンスを生成
    _gslist->Add( f->Parse(context) ); // 構文解析を行い子のノードに追加
    n1 = f->GetAST();              // 構文解析後の構文木のノードを得る
    while ( _attr == '*' || _attr == '/' )
      {
          _gslist->Add( new OPSymbol(_attr) );  // '*' または '/' を追加
          GetToken();              // トークン列からトークンを読む
          f =_new FSymbol();
          _gslist->Add( f->Parse(context) );
          n2 = f->GetAST();
          n1 = _astbuilder->BuildTree(_attr,n1,n2);
      }
    SetAST(n1);                    // n1 を構文木のノードにする
    return (this);
}
```

CalcASTBuilder クラスの BuildNode, BuildTree メソッドはそれぞれ以下のようになる。それぞれのメソッドの引数 op によって、適切なオブジェクトが生成される。

```
Node* CalcASTBuilder::BuildNode(int op, Value *value)
{
    switch (op)
      {
        case eCONST:
          return ( new ConstNode( (IntValue *)value ) );
        case eVAR:
          return ( new VarNode( (VarValue *)value ) );
      }
}
```

```
Node* CalcASTBuilder::BuildTree(int op, NodeList *sons)
{
    switch(op)
      {
        case eLET:       // キーワード 'let' の場合
          return ( new LetNode(sons) );
        case eDECLLIST:  // DECLLIST の場合
          return ( new DeclListNode(sons) );
        case eDECL:      // DECL の場合
          return ( new DeclNode(sons) );
        default:
          if ( isop(op) )  // '+','-','*','/' なら
            return ( new ExpNode(sons,op) );
      }
}
```

5.4.1 節のプログラムリストと比べると多少複雑になっているが、第 4 章で構築したフレームワークを適用することにより、各文法記号の Parse メソッドは、文法を機械的に変換することにより、実装可能である。

5.5 意味解析器の作成

この言語では、let 式で束縛された変数のみが let 式の本体で使用できる。また、式の型は整数型に限るため、整数型から実数型への変換などは行われない。

属性文法 (第 4.3 節参照) で、変数の値を構文木の各要素に付随するものが属性である。また、図 5.5 の *env* は、名前から格納情報への写像を表す **環境** (environment) を表す。

この言語処理系では抽象構文木に環境を付加し、属性評価を行う。環境を表すクラス EnvValueList は、IntValue 及び VarValue クラスのオブジェクトをメンバー変数として持っている EnvValue クラスのオブジェクトのリストである。

属性解析は次のような手順で行う。まず、環境を表す EnvValueList クラスのオブジェクトと、評価後の結果を表す IntValueList クラスのオブジェクトを空にする。そして、抽象構文木を前順に走査し、ノードの種類によって、以下のように行う。

- Let

  環境を左のノードに渡して評価を行う。左のノードでは、変数の束縛が行われており、環境が更新される。その後右のノードに環境を渡して評価を行う。

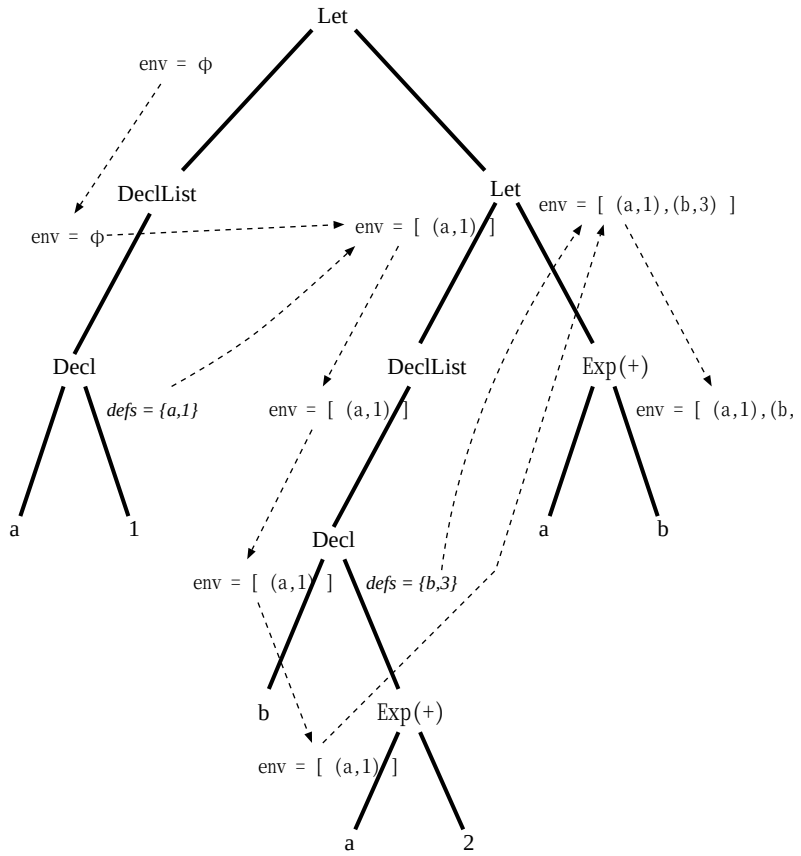

図 5.5: 属性付構文木

- DeclListNode

  このアプリケーションでは唯一このノードを 3 つ以上持つ。このためまず初めに環境を、一番左のノードに渡し、評価が終了したら、順に右のノードに渡す。全てのノードの評価が終了したら、親のノードに環境を渡す。

- DeclNode

  まず左のノードが変数を表す VarNode クラスのオブジェクトであるかどうか調べる。VarNode クラスのオブジェクトであれば変数名 (VarValue クラスのオブジェクト) をローカル変数に記録しておく。そうでなければエラーである。次に右のノードの評価を行い、記録しておいた変数名と評価結果 (IntValue クラスのオブジェクト) の組を環境に追加する。

- ExpNode

  左のノードの評価結果と、右のノードの評価結果を、ExpNode クラスのメンバー変数によって、加算、減算、乗算、除算のいずれかを行い、その結果 (IntValue クラスのオブジェクト) を、属性評価後の結果を表す IntValueList のオブジェクトに追加する。

- VarNode

  環境を表す EnvValueList を EnvValueListIterator クラスの Lookup メソッドによって後ろから順に、VarNode と同じ変数名が無いか調べる。後ろから順に調べるのは、let 式で同じ変数が束縛された場合、後に束縛された方を有効とするからである。EnvValueListIterator クラスは、ValueListRevOrdIterator クラスを継承し、Lookup メソッドを実装する。同じ変数名があった場合は、変数の値 (IntValue クラスのオブジェクト) を返す。同じ変数名が無かった場合は、エラーとなる。

- ConstNode

  ConstNode のメンバー変数である IntValue クラスのオブジェクトを返す。

また属性文法は以下のようになっている。

属性文法

$$
\begin{aligned}
&Let \ \rightarrow \ DeclList \ Exp \\
&\quad \{ \ Exp.env = newblock(DeclList.env, Exp.env) \ \} \\
&DeclList \ \rightarrow \ Decl_1 \ Decl_2 \ \cdots Decl_n \\
&\quad \{ \\
&\qquad for(i = 1; i < n; i++) \ \{ \ newblock(Decl_i.defs, \ Decl_{(i+1)}.env); \} \\
&\qquad newblock(Decl_n.defs, \ Decl_n.env); \\
&\quad \} \\
&Decl \ \rightarrow \ Var \ Exp \\
&\quad \{ \ defs = (\ Var.VarValue, \ Exp.Evaluate()\ )\} \\
&Let\{ \ in \ env, \ out \ env \ \} \\
&DeclList\{ \ in \ defs, \ out \ env \ \} \\
&Decl\{ \ in \ env, \ out \ defs \ \}
\end{aligned}
$$

$Exp\{\ in\ env,\ out\ env\ \}$
$Var\{\ in\ env,\ out\ env\ \}$
$Const\{\ in\ env,\ out\ env\ \}$

属性評価の結果は、Evaluate メソッドの第 2 引数に与えた IntValueList クラスのオブジェクトの最後の要素である IntValue のオブジェクトであり、InvValue クラスの GetNum メソッドを実行して、整数値を取り出し、出力する。

5.6 まとめ

全てのサブシステムは、第 4 章で構築したフレームワークを構成する抽象クラスを継承することで、構築可能であることを示した。その結果フレームワークは、言語処理系のアプリケーションを構築する上で必要とされる設計情報の大半を把握していることが分かった。また、本研究で構築したフレームワークを適用することによって、構文解析器の半自動生成が可能であることを示した。

第 6 章

評価

第 3 章では、以下のような言語処理系のためのフレームワークを設計する上での問題を考察し、第 4 章でフレームワークの設計を行い、以下のように解決した。

- 文法の変更の容易性

  文脈自由文法の各文法記号は、終端記号と、非終端記号の二種類あり、それぞれ特有の性質を持っている。このことを考慮し、文法の変更を容易にできるようにする。

  ⇓

  Interpreter パターンを用いることによって、終端記号と、非終端記号のそれぞれの性質を定義したクラスを構築し、文法記号を表すクラスはこれらのクラスを継承して定義することによって、文法の変更を容易にした。

- 出力の柔軟性

  解析木はどのようなノードを作成する場合でも、一定の方法で行えるようにしなければならない。また抽象構文木の作成法を容易に変更できるようにしなければならない。

  ⇓

  Interpreter パターンを用いて、構文解析を行うメソッドで、一定の方法で解析木の作成を可能にした。また、Builder パターンは、抽象構文木の作成部を文法記号を表すクラスから分離し、抽象構文木の作成方法の変更を容易にした。

- 意味規則の利用

  一般に属性文法では、属性はさまざまな型をとり得る。したがって、フレームワークを設計するにあたっては、属性の型を容易に変更できるようにしなければならない。

  ⇓

  意味解析で扱う型の一般的性質を定義したクラスを作り、属性を表すクラスは、このクラスを継承して定義することによって、さまざまな型を扱えるようになった。

第 5 章で示した適用例によって、このフレームワークは、言語処理系のアプリケーションを構築する上で必要とされる設計情報の大半を把握しており、特に構文解析器は、文法を機械的に変換することによって、構築可能であることが判明した。よってこのフレームワークを用いれば、言語処理系のアプリケーションを短時間で作ることが可能である。

デザインパターンを適用して実際のシステムを構築する場合、対象となるシステムの設計問題に適しているパターンを選択し、そのパターンを使うことによる結果や、設計上のその他の制約を考慮した上でも適用できるかどうかを十分検討する必要がある。例えば、文法記号を表すクラスのオブジェクトを走査するメソッドを、Composite pattern を適用して文法記号を表すクラスに定義するか、Iterator pattern を適用して分離するかといった問題がある。Composite pattern を適用した場合、文法記号を走査するメソッドは、文法記号を表すクラスに定義する。そのため、Iterator pattern を適用した場合と比べて、クラス数が少なくなるという利点がある。また、Iterator pattern を適用した場合、文法記号を表すクラスのオブジェクトを走査するクラスを分離するため、操作方法を再利用したり、操作方法のみを変更したりすることが可能である。文法記号には終端記号と非終端記号の二種類あるため、走査方法を共通にするのは無理がある。また、走査方法の変更が必要となる可能性があるため、本研究で構築したフレームワークでは、文法記号を表すクラスのオブジェクトを走査するクラスを分離した。

したがって、デザインパターンを特定分野に適用した例について、詳細に記述した文書は、同じようなシステムを構築する際に多いに有用である。さらに、デザインパターンでは触れられていない問題、例えば、構文解析中に文法が動的に変化するような場合の文書化は、今後待たれる課題である。

第 7 章

おわりに

7.1 まとめ

本研究では、デザインパターンを適用し、言語処理系のためのフレームワークを構築した。その結果、Gamma の 23 個のデザインパターンのみで、言語処理系のためのフレームワークは十分構築可能であり、考察した設計問題をすべて解決できることを示した。また、デザインパターンの適用に関する問題点を検討した。

7.2 今後の課題

- フレームワークの完成度をより高めるためには、更に適用例を増やす必要がある。
- デザインパターン [Gamma94] と比べると、フレームワークやアプリケーションの設計に関する記述が不十分であり、分かりにくいと思われる。そのため、設計に関する記述をさらに充実させる必要がある。

謝辞

本研究を行うにあたり、終始御指導いただきました片山卓也教授には、心からの感謝を申し上げます。また、夜遅くまでプログラムのデバッグに協力してくださった鈴木正人助手には深く御礼申し上げます。そして、本研究に関して数々の助言をいただいた片山研究室の皆様に感謝致します。

参考文献

[Gamma94] Gamma E., Helm R., Johnson R. and Vlissides J. (1994). Design Patternss - Microarchitecture for Reusable Object-Oriented Software Reading, Massachusetts: Addison-Wesley. 邦訳「オブジェクト指向における再利用のためのデザインパターン」1995, ソフトバンク

[Pree95] Wolfgang Pree. Design Patterns for Object Oriented Software Development. The ACM Press, a Division of the Association for Computing Machinery, Inc.(ACM). 1995. 邦訳「デザインパターンプログラミング」1996, トッパン

[Sato95] 佐藤啓太「クラスライブラリ自由自在」1995, トッパン

[Sassa89] 佐々政孝, プログラミング言語処理系, 岩波講座ソフトウェア科学 5, 1989.

[Deu89] L. Peter Deutch. Design reuse and frameworks in the Smalltalk-80 system. In Ted J. Biggeststaff and Alan J. Perlis, editors, *Software Reusability, Volume II: Applications and Experience*, pp.57-71. Addison-Wesley, Reading, MA, 1989

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming,* 1(2):22-35, June/July 1988.