SableCC, An O-O Compiler Framework (Presentation on Kuno-zemi)

久野　靖*

2000.6.26

1　はじめに

□ Lex/Yacc --- 言語屋 (私) がコンパイラを作るのにとても重宝なツール

□ 文法を書いてみて「いい加減」ではないことがチェックできる

□ その文法でコードを書いてみて「書ける」ことがチェックできる

□ それだけで構文解析までコンパイラができたことになる

□ あとは意味解析と簡単なコード生成だけで「動かしてみる」ことができる

- 実際には「型のある言語」だと意味解析が大変だけど
- 実際には「並列言語」なんかだと生成コードとランタイムが大変だけど

1.1　最近はほとんどのコードを Java で書くようになったので…

□ Lex/Yacc は当然、C 言語が前提になっている→ Java では使えない

□ Java でも、Lex/Yacc みたいなツールが欲しい!

□ 探してみたけど…有名なのは Sun の「JavaCC」

- JavaCC は LL(k) 文法→ LL(k) なんか嫌いだ!

□ ずっと悩んでいたが…

- 実は「CUP」なんていうのもあったらしい→滝本さんが紹介してくださるそうで

□ 先日 JSPP'2000 という会議でコンパイラの大家、中田先生に会ったら

- → SableCC というのがおすすめだとのこと
- →じゃあ取り寄せて遊んでみよう→大変面白い→紹介させていただきます

1.2　本日の内容

□ SableCC の作者が書いた Thesis の紹介

- 「SableCC, An Objecdt-Oriented Compiler Framework」
- Étienne Gagnon, MS Thesis, McGill Univ., Montreal, 1998.
- つまり学生が修論で作ったもの→結構すごい!

□ 簡単なデモ

2　Introduction

□ コンパイラを作るのは大変→ツール (特にフロントエンド) が多く存在

□ Java でやりたい→ CUP(Lex/Yacc の Java 移植)、ANTLER(PCCTS の Java 移植版)

- 移植だと Java の特徴が活かされない部分も
- 以下に比較検討

2.1　Lex/Yacc

□ Lex → lexer、Yacc → parser、その移植版が JLex と CUP

□ 利点

- DFA ベースの lexer、LALR(1) パーザ
- ソース配付

□ 弱点

- 8 ビット文字限定
- JLex →マクロのバグ (字面の置き換えは難しい)
- JLex と CUP を協調させるのに手作業必要
- CPU のあいまい文法処理→初心者には危険
- パーザの動作コードが埋め込み方式→虫取りがしにくい
- 現在なら AST ベースの方がよい (安全、マルチパス化が簡単)
- 開発サイクルが複雑

*筑波大学大学院経営システム科学専攻

2.2 PCCTS(Purdue Compiler Construction Tool Set)

□ その Java 移植版→ ANTLER(パーザ)、DLG(lexer)、SORCERER(ツリー生成/変換)

□ Sun からも同様のツール→ JavaCC

□ 利点

- lexer と parser が統合
- 16 ビット文字 (JavaCC のみ)
- LL(k) の方がアクションは書きやすい (そうかなあ?)
- semantic predicate により文脈依存文法まで対応
- AST の自動生成
- ANTLER はソース公開 (JavaCC は非公開)

□ 弱点

- semantic predicate は扱いがむずかしい
- LL(k) 文法の記述力 (左再帰が許されない)
- 開発サイクルが複雑
- AST のデータ構造の正しさはプログラマに依存

2.3 SableCC では…

□ DFA lexer、LALR(1) パーザ、自動 AST 生成

□ もっと使いやすく OO を活かした構成

□ 生成コードとプログラマが書いたコードの分離 (OO の特徴を用いて)

□ 拡張 Visitor パターンの導入

3 Background

□ (正規表現と文脈自由文法の復習)

4 SableCC

4.1 introduction to SableCC

□ 最近のコンパイラに対する要求→マルチパス化、AST ベースの処理、発展性、保守性

□ lexer+parser はわりと「どうでもいい」ことに

□ その代り、意味解析以降がうまくできることに重点

□ そのため SableCC では…

- AST は勝手に作られる
- AST のノードは厳密に型づけ、壊されることがないように
- 各種解析はそれぞれ別のクラスとして tree walker のサブクラスとして作成
- 各種解析情報は解析クラス側で持たせ、AST は書き換えない→安全

4.2 general steps to build a compiler using SableCC

□ 記述ファイルを用意

□ SableCC で生成する

□ 生成されたクラスを継承して自前の作業クラス群を作る

□ コンパイラドライバ (main) を作る

□ コンパイルして動かす

4.3 specificaiton files ...

□ (実例で見ましょう)

5 Lexer

□ Lexer なんてちっとも面白くないから略

□ ただし日本語文字も使えるところはすごい (2 分木ベースの状態遷移)

□ Lex と同様、状態をいじくりながら遷移できる (私は嫌い)

6 Parser

□ いわゆるふつーの BNF による文法記述

□ EBNF →「X*」「X+」なども使える

□ Yacc 以来の「アクション記述」がない!

6.1 SableCC Naming Rules

□ ということは、後でよそから「このルール」「この選択肢」という指定が必要

- 指定というか実はノードのクラス名になる
- ルール「add = …」には「PAdd」という名前がつく
- ルール「add = xxxx」の右辺は選択肢がない場合「AAdd」という名前
- 選択肢がある場合には必ず名前を書く

```
expr = {add} expr plus term   ← AAddExpr
     | {sub} expr minus term  ← ASubExpr
     | {term} term            ← ATermExpr
```

□ 列のなかの「この要素」はノードのメソッド getXXX()/setXXX() でアクセス

- 列の中の各記号は通常その名前だけで区別できる
- 複数同じ記号が出て来る場合はそれにだけ名前をつける

```
record = [struct]:ident dot [field]:ident
```

← getStructIdent()/getFieldIdent() 等のメソッド名になる

6.2 EBNF

□ あんまり好きじゃないので略

□ 反復に対応するノードは Typed Linked List になる

6.3 the Parser class

□ Lexer を渡して生成し、メソッドは parse() だけ

```
Parser parser = new Perser(...lexer...);
Start tree = parser.parse();
```

7 Framework

8 the Visitor design pattern revisited

□ もともとの問題: if-then-else による場合分けがいや

```
void Selected(Shape obj) {
  if(obj instanceof Circle)
    System.out.println("a circle was selected");
  else if(obj instanceof Square)
    System.out.println("a square was selected");
  else
    System.out.println("a rectangle was selected");
}
```

□ ID 番号を用意して switch?

```
abstract class Shape {
  ...
  abstract int id();
}
class Rectangle extends Shape {
  ...
  static final int ID = 1;
  int id() { return ID; }
}
class Circle extends Shape {
  ...
  static final int ID = 2;
  int id() { return ID; }
}
```

□ switch 側は次のようになる。

```
void Selected(Shape obj) {
  switch(obj.id()) {
case Circle.ID:
    System.out.println("a circle was selected"); break;
case Square.ID:
    System.out.println("a square was selected"); break;
case Rectangle.ID:
    System.out.println("a rectangle was selected"); break;
  }
}
```

□ ID を管理するのがやっかい、switch の枝を増やす必要

□ 次のインタフェースを考える

```
interface Switch {
  void caseCircle(Circle obj);
  void caseSquare(Square obj);
  void caseRectangle(Rectangle obj);
}
```

□ Shape 側には apply() というメソッドを用意

```
abstract class Shape {
  ...
  abstract void apply(Switch sw);
}
```

□ 各図形ごとに override すればよい

```
class Circle extends Shape {
  ...
  void apply(Switch sw) { sw.caseCircle(this); }
}
```

□ 使うときは次のようにすればよい

```
void Selected(Shape obj) {
  obj.apply(new Switch() {
    void caseCircle(Circle obj) {
      System.out.println("a circle was selected"); }
    void caseSquare(Square obj) {
      System.out.println("a square was selected"); }
    void caseRectangle(Rectangle obj) {
      System.out.println("a rectangle was selected"); }
  });
}
```

□ ただし次の弱点

- 要素型の追加が大変
- クラス階層を横切った visit ができない

8.1 extending the Visitor design pattern

□ 前記の弱点を克服するための拡張を行なう

```
interface Switch { }

interface Switchable { void apply(Switch sw); }

interface ShapeSwitch extends Switch {
  void caseCircle(Circle obj);
  void caseSquare(Square obj);
  void caseRectangle(Rectangle obj);
}
```

8.2 extending the Visitor design pattern(2)

□ 使う側は次のようになる

```
abstract class Shape implements Switchable {
  ...
}

class Circle extends Shape {
  ...
  void apply(Switch sw) {
    ((ShapeSwitch)sw).caseCircle(this);
  }
}
```

8.3 extending the Visitor design pattern(3)

□ では要素を拡張する場合は?

```
interface ExtendedShapeSwitch extends Switch {
  void caseOval(Oval obj);
}

class Oval extends Shape {
  ...
  void apply(Switch sw) {
    ((ExtendedShapeSwitch)sw).caseOval(this);
  }
}

interface AllShapesSwitch
    extends ShapeSwitch, ExtendedShapeSwitch {
  void Selected(Shape obj) {
    obj.apply(new AllShapesSwitch() {
      void caseCircle(Circle obj) {
        System.out.println("a circle was selected"); }
      void caseSquare(Square obj) {
        System.out.println("a square was selected"); }
      void caseRectangle(Rectangle obj) {
        System.out.println("a rectangle was selected"); }
      void caseOval(Oval obj) {
        System.out.println("an oval was selected"); }
    });
  }
}
```

8.4 SableCC and visitors

□ SableCC ではアダプタクラスを用意

```
class AllShapesSwitchAdapter implements AllShapesSwitch {
  void caseCircle(Circle obj) { defaultCase(obj); }
  void caseSquare(Square obj) { defaultCase(obj); }
  void caseRectangle(Rectangle obj) { defaultCase(obj); }
  void caseOval(Oval obj) { defaultCase(obj); }
  void defaultCase(Shape obj) { }
}
```

□ これを継承して必要なところだけオーバライド

```
void Selected(Shape obj) {
  obj.apply(new AllShapesSwitchAdapter() {
    void caseCircle(Circle obj) {
      System.out.println("a circle was selected"); }
    void defaultCase(Shape obj) {
      System.out.println("The selected object is not a circ
  });
}
```

8.5 AWT Walkers

□ AST をたどるアダプタを用意

```
class DephFirstAdapter extends AnalysisAdapter {
  void caseStart(Start node) {
    node.getAXxx().apply(this); // Xxx child of Start
    node.getEOF().apply(this); // EOF child of Start
  }
  void caseXxx(Xxx node) {
    node.getYyy().apply(this); // Yyy child of Xxx
    node.getZzz().apply(this); // Zzz child of Xxx
  }
  ...
}
```

□ これをカスタマイズするには?

```
class Action extends DepthFirstAdapter {
  void caseXxx(Xxx node) {
    ...
    action code
    node.getYyy().apply(this); // first child of Xxx
    node.getZzz().apply(this); // second child of Xxx
  }
  ...
}
```

8.6 AWT Walkers(2)

□ たどりのコードと混ざるのは嫌→分離

```
class DephFirstAdapter extends AnalysisAdapter {
  void caseStart(Start node) {
    inStart(node);
    node.getAXxx().apply(this); // first child of Start
    node.getEOF().apply(this); // second child of Start
    outStart(node);
  }
  void inStart(Start node) { }
  void outStart(Start node) { }
  void caseXxx(Xxx node) {
    inXxx(node);
    node.getYyy().apply(this); // first child of Xxx
    node.getZzz.apply(this); // second child of Xxx
    outXxx(node);
  }
  ...
}
```

8.7 additional features

□ さらに、AST そのものに解析情報を格納するのも嫌→分離

```
Object getIn(Node node);
Object getOut(Node node);
void setIn(Node node, Object in);
void setOut(Node node, Object out);
```

□ そのほか各ノードについて利用可能なメソッド (表 6.1)

9 Case Studies

9.1 SableCC with SableCC

□ もちろん自分自身で書くのはお約束

□ Typed AST →型検査によるエラー発見は強力だった

□ version 1 では選択肢を番号で指定していた→間違いのもと→名前方式に

9.2 code profiling

□ 学生実験の課題として、Simple C にプロファイルコードを追加させた

- 学生は Java と OO ははじめて
- しかし学生にとってフレームワークを使うのは問題なかった
- 主な問題は JDK のインストールと CLASSPATH と…
- あとはコンパイラが始めてで AST の理解が困難だったとか
- ノードの番号指定が唯一の発見しにくいバグの原因だった

□ 対照群として McCAT コンパイラコンパイラを使った学生はもっと大変だった

- 実行時まで判らないエラー、AST が壊れていた場合、など

9.3 a Java frontend

□ JDK 1.0.2 仕様の言語のフロントエンドを作った

□ 仕様書通りの文法で OK。時間が掛かったのは名前をつけるところ

□ 動かしてみるとすごくメモリを食った→オブジェクト生成を少くチューニング→ 1.3M の Java ソースの処理が 55 分だったのを 99 秒に

9.4 fast points-to analysis of Simple C programs

□ AST 上でポインタ解析を行なうツールを作った

- パス 1 → AST を扱いやすいように変形
- パス 2 →最初の状態のストレージモデルを生成
- パス 3 →ストレージモデルを動かして最終状態を計算

□ AST 変形は次のような感じ

```
import ca.mcgill.sable.simple.analysis.*;
import ca.mcgill.sable.simple.node.*;
import ca.mcgill.sable.util.*;

class ModifyAST extends DepthFirstAdapter {
  public void outAIdentifierParameterDeclaration(
      AIdentifierParameterDeclaration node) {
    node.replaceBy(new AParameterDeclaration(
      new ATypedefTypeSpecifier(node.getIdentifier()),
        node.getDeclarator()));
  }
  public void outAAbstractIdentifierParameterDeclaration(
      AAbstractIdentifierParameterDeclaration node) {
    node.replaceBy(new AAbstractParameterDeclaration(
      new ATypedefTypeSpecifier(node.getIdentifier()),
        node.getAbstractDeclarator()));
  }
  ...
}
```

□ 定義通りにデータ構造を定義して値を設定…

```
import ca.mcgill.sable.util.*;
class Variable {
  ...
  void setType(Type t) {
    Variable e = this.ecr();
    e.type = t; // type(e) <- t
    for(Iterator i = e.pending.iterator(); i.hasNext();) {
      e.join((Variable) i.next()); // join(e,x)
    }
  }
  ...
}

Hashtable variables = new Hashtable(1);
...
// v = Ref(bottom, bottom)
variables.put(id, new Variable(new Ref()));
```

□ ポインタ解析規則ごとにルールをメソッドとして定義

```
private void rule1(Variable x, Variable y) {
  Variable ref1 = ((Ref) x.ecr().type).ref.ecr();
  Variable lam1 = ((Ref) x.ecr().type).lam.ecr();
  Variable ref2 = ((Ref) y.ecr().type).ref.ecr();
  Variable lam2 = ((Ref) y.ecr().type).lam.ecr();
  if(ref1 != ref2) { ref1.cjoin(ref2); }
  if(lam1 != lam2) { lam1.cjoin(lam2); }
}
```

これらを Simple C の書き換え種別ごとに場合分けして適用していく…

9.5 a framework for storing and retrieving analysis information

□ コンパイラでソースを解析し、解析情報をコメントの形でソースに付加

□ あとでそれを読み込めるようにする

- ソースとコメント中の解析情報とが「ごちゃまぜ」なので難しい
- SableCC では 2 つの lexer、parser を並行して動かせばよかった
- 両者の入力をもとに 1 つの AST を組み立てる

10 Conclusion and Future Work

□ まあよかったんじゃない?

□ エラーリカバリはまだないんです