情報処理演習
PL0’ コンパイラ作成の手引

2008 年度担当: 中井央

2008.04.22

概 要

この演習では、pl0’ (ぴーえるぜろだっしゅ) という小さなプログラミング言語のコンパイラの作成を行なうことで、コンパイラ作成の手法を学ぶ。

1 準備

本演習では SableCC (以下、コマンド名である小文字の sablecc で表記する) を用いてコンパイラ作成を行なう。

1.1 sablecc の概要

sablecc はカナダの Étienne Gagnon が開発した処理系 [1] であり、Java をベースとしている。sablecc で開発した処理系における処理の流れでは、まず、字句解析と構文解析により解析木を作り、その後、それを走査し、意味解析等を行なう。解析木の走査は必要に応じて複数あってもよい。

sablecc に正規表現と文法を記述したファイルを与えると、字句解析クラス、構文解析クラス、構文解析をした結果として構築される解析木のノードとなるクラス、解析木を走査するクラス (の元となるクラス) が生成される。

意味解析器を作成するには、解析木を走査するクラスを継承して、クラスを作成する。解析木の走査は Visitor パターンをベースにして作られている。ここでは、この演習で利用する DepthFirstAdapter クラスをもとに話を進める。

木の各ノードを訪問する方法にはいくつか考えられるが、そのうちの 1 つは深さ優先探索 (Depth First traversal) というもので、ルートノードから探索を初め、まず、その最左の子ノードへ移動し、その後、さらにその最左の子へと探索を進める。葉ノードに至ったら、引き返し、親ノードから見て次の子ノードへ移動する。以下これを繰り返す。探索方法にはこの他、例えば、最左ではなく最右から行なう深さ優先探索や、同じレベル (兄弟) のノードを順に訪問した後、それらの子孫に同様に探索を行なう幅優先探索 (breadth first) などがある。

さて、深さ優先で探索する場合、あるノードに着目するとそのノードに対して何かアクションを起こすタイミングとしては次のものが考えられる。まず、そのノードへ最初に到達した時点、すなわち親ノードからそのノードへと到達した時点であり、同様にそのノードの全ての子孫を訪問し終って親ノードへ帰る時点の 2 つの時点が考えられる。これをそれぞれ、行きがけ、帰りがけと呼ぶことにしよう。また、子ノードが複数ある場合は、自身を起点として、各子ノードを訪問する間の時点が考えられる。

sablecc における走査のクラスでは、各ノード毎にこれら訪問時点ごとのアクションを意味するメソッドが用意されており、ユーザはそれらをオーバーライドすることでそれぞれの時点で具体的に操作をすることができる。

1.2 sablecc 利用のための準備

最新版の sablecc は `http://www.sablecc.org` から入手できるが、このテキスト執筆時の最新版を icho の上の私のホームにも置いている (sablecc-3.2.tar.gz)。sablecc を利用するには私のホーム上にパスを通すか自身で sablecc を導入するかする。

前者の場合、次のように入力する (bash : 標準のシェルの場合、$ はプロンプト)。

```
$ export PATH=$PATH:~nakai/sablecc-3.2/bin
```

後者の場合、最新版を入手し、ホームに展開する。icho 上の場合、次の手順で展開、設定できる。下記で tar の部分は最新版を入手した場合、そのファイル名に適宜置き換えて欲しい。

```
$ cd 展開したいディレクトリ
$ tar zxv ~nakai/sablecc-3.2.tar.gz
```

展開すると sablecc-3.2 というディレクトリができる。この中にはいくつかのファイルがあるが、以降で利用するのは、bin と lib である。bin の中には sablecc というファイルが入っており、中に次の一行がある。

```
java -jar lib/sablecc.jar $*
```

これを必要に応じて書き換える。ここではユーザのホーム直下に一式を展開したと仮定する。すると次のように書き換えれば良い。

```
java -jar ~/sablecc-3.2/lib/sablecc.jar $*
```

また、実行可能にするため、このファイルに実行許可を与えておく。

```
$ chmod a+rx sablecc
```

1.3 sablecc のサンプルプログラム

[1] にある例題プログラムを掲載し、簡単に解説する。

図 1 は sablecc に与える正規表現･文法 (以下、単に文法と記す) の記述である。1 行目の package postfix; は Java のパッケージの名称を表しており、生成されるソース一式がそのパッケージに収められることを意味している。

2 行目から、字句の定義が正規表現で行なわれている。Tokens は字句の定義のセクションの開始を示している。それぞれは
トークン名 = 正規表現 ;
という形になっている。正規表現として特殊なものとしては、文字の範囲の指定に [と] で括る表現が使える。Java がベースであるため、コード体系は Unicode となっている。

```
 1 Package postfix;
 2
 3 Tokens
 4   number = ['0' .. '9']+;
 5   plus = '+';
 6   minus = '-';
 7   mult = '*';
 8   div = '/';
 9   mod = '%';
10   l_par = '(';
11   r_par = ')';
12   blank = (' ' | 13 | 10)+;
13
14 Ignored Tokens
15   blank;
16
17 Productions
18   expr =
19     {factor} factor |
20     {plus} expr plus factor |
21     {minus} expr minus factor;
22
23   factor =
24     {term} term |
25     {mult} factor mult term |
26     {div} factor div term |
27     {mod} factor mod term;
28
29   term =
30     {number} number |
30     {expr} l_par expr r_par;
```

図 1: postfix.grammar

14 行目の Ignored Tokens は入力として無視して良いトークンの指定になっている。一般的には空白は読み飛ばすため、この例のように記述することが多い。

17 行目からが文法 (生成規則 = production rule) の記述となる。sablecc ではある左辺に対し、右辺が複数ある場合、構文解析時に作成される解析木のノードが一意にわかるよう名前をつけることになっている。例えば 18 行目から始まる expr を左辺とする規則は右辺が 3 つある。そのそれぞれに { と } で囲った名前が付けられている。

さて、文法の記述を終えるとそれを sablecc に与えることで、字句解析 (lexer)、構文解析 (parser)、解析木のノード (node)、走査 (analysis) の各パッケージが作られ、その中にそれぞれに必要なクラスが作成される。

```
$ sablecc postfix.grammar
```

もし、入力に誤りがあればエラーメッセージが出て実行が停止するのだが、エラーメッセージが非常にわ

```
 -- Generating parser for postfix.grammar in /home1/nakai/IP2
org.sablecc.sablecc.parser.ParserException: [4,23] expecting: ']'
        at org.sablecc.sablecc.parser.Parser.parse(Unknown Source)
        at org.sablecc.sablecc.SableCC.processGrammar(Unknown Source)
        at org.sablecc.sablecc.SableCC.processGrammar(Unknown Source)
        at org.sablecc.sablecc.SableCC.main(Unknown Source)
```

図 2: sablecc 実行時のエラーの例

かりにくい。図 1 の 4 行目の] を消して、sablecc に与えてみた実行結果 (の一部) を図 2 に示す。

この中で [4, 23] という記述があるが、要は 4 行目の 23 カラムでエラーが見つかったということである。この例の場合は expecting : ']' という記述があり、] が欠落していることが読みとれる。

さて、このサンプルは中置記法の算術式の入力に対し、その後置記法を出力するというものである。中置記法とは我々が日常使用している、被演算子の間に演算子を置く記法である。後置記法は被演算子の後に演算子を置く記法である。例えば、1+2 の後置記法は 1 2 + となり、1+2*3 の後置記法は 1 2 3 * + となる。後者の例は、1+2*3 = 1+(2*3) である点に注意されたい。後置記法は、この例を使うと、左から順に見ていき、演算子が見つかったところで、直前の 2 つ (もし演算子が単項演算子なら 1 つ) の被演算子に対してその演算子で演算を行なう。この例では 2 3 * の部分がそれにあたり、演算を行なった結果の 6 でそれらを置き換える。すると得られるのは 1 6 + であり、同様に左から見ていき、演算子が出てきたらその直前 2 つを被演算子として演算子をし、結果である 7 を得る。

SableCC でこれを実現する場合、まず、入力を字句解析、構文解析し、得られた構文解析木の上を順に訪問しながら対応する文字列を生成するように意味解析器を作る。この動作を深さ優先の走査にて行なうのが図 3 である。

図 3 は DepthFirstAdapter クラスを継承して、ユーザが作成する。これは図 1 で与えたのと同じパッケージに入れるため、先頭にその記述がある。そして、パッケージ内にはノード (node) と走査 (analysis) のためのパッケージがあり、それらを利用するため、それらを import しておく必要がある。

6 行目は、訪問したノードが葉ノードの場合の動作を記述している。具体的にはトークン number だった場合である。sablecc では、文法記述中のトークンに相当するノードのクラス名はその先頭が T となり、その後ろにトークン名が来るが、トークン名の先頭も大文字に変更されたものになる。この例ではトークンが見つかるとそれ自身を表示するようにしている。

10 行目のメソッドは outAPlusExpr となっている。

```
 1 package postfix;
 2 import postfix.analysis.*;
 3 import postfix.node.*;
 4 class Translation extends DepthFirstAdapter
 5 {
 6   public void caseTNumber(TNumber node)
 7   {// When we see a number, we print it.
 8       System.out.print(node);
 9   }
10   public void outAPlusExpr(APlusExpr node)
11   {// out of alternative {plus} in Expr,
      // we print the plus.
12       System.out.print(node.getPlus());
13   }
14   public void outAMinusExpr(AMinusExpr node)
15   {// out of alternative {minus} in Expr,
      // we print the minus.
16       System.out.print(node.getMinus());
17   }
18   public void outAMultFactor(AMultFactor node)
19   {// out of alternative {mult} in Factor,
      // we print the mult.
20       System.out.print(node.getMult());
21   }
22   public void outADivFactor(ADivFactor node)
23   {// out of alternative {div} in Factor,
      // we print the div.
24       System.out.print(node.getDiv());
25   }
26   public void outAModFactor(AModFactor node)
27   {// out of alternative {mod} in Factor,
      // we print the mod.
28       System.out.print(node.getMod());
29   }
30 }
```

図 3: Translation.java

これは、図 1 の 20 行目に対応している。

```
18   expr =
19     {factor} factor |
20     {plus} expr plus factor |
21     {minus} expr minus factor;
```

最初の out は帰りがけに操作をすることを意味している。それ以降は sablecc の命名規則に基づいて、A の後ろに指定した規則の名前 (この例では {plus}) ＋左辺の名前 (この例では expr) となっている。

走査のクラスは Visitor パターンに基づいているため、メソッドの引数は、処理の対象となるノードが渡されてくる。各ノードを表すクラスはその子ノードを得るためのアクセッサを用意している。この例では右辺には、expr, plus, factor という子があり、この例では必要なのは plus のノードであるので、getPlus が利用されている。

ここまでで文法から必要なクラスを生成し、また、それらに基づいて具体的な操作を DepthFirstAdaptor を継承したクラスとして作成した。最後にそれらをまとめてコンパイラとして完成させる。図 4 にそのドライバクラスを記述する。

図 4 の 12 〜 16 行目は字句解析器と構文解析器の生成である。標準入力を与えているが特定のファイル (main メソッドの引数など) を与えるように変更するのは容易であろう。生成された構文解析器クラスのインスタンスの parse メソッドを呼ぶと与えられた入力に対する解析木が生成される。解析木のルートは Start クラスとなっているため、18 行目では得られた解析木 (正確にはそのルートノード) を Start クラスの変数へ格納している。

20 行目では得られた木に対して Visitor を適用している。つまり、木を走査し、意味解析を行なっている。

さて、全体の流れは次のようになる。

1. 文法記述を作り、sablecc に与える。
2. 作成されたパッケージ内に意味解析クラスを作成する。
3. 作成されたパッケージ内にドライバクラスを作成する。
4. 全体をコンパイルし、実行する。

```
 1 package postfix;
 2 import postfix.parser.*;
 3 import postfix.lexer.*;
 4 import postfix.node.*;
 5 import java.io.*;
 6 public class Compiler
 7 {
 8   public static void main(String[] arguments){
 9     try {
10       System.out.println
           ("Type an arithmetic expression:");
11       // Create a Parser instance.
12       Parser p =
13       new Parser(
14           new Lexer(
15               new PushbackReader(
16                   new InputStreamReader
                         (System.in), 1024)));
17       // Parse the input.
18       Start tree = p.parse();
19       // Apply the translation.
20       tree.apply(new Translation());
21     }
22     catch(Exception e){
23       System.out.println(e.getMessage());
24     }
25   }
26 }
```

図 4: Compiler.java

演習

上記の流れに沿って、上述の例題プログラムを入力し、実行せよ。

2 PL0' コンパイラの作成

以下では、sablecc を用いて PL0' コンパイラを作成していく手順を示す。

2.1 準備

コンパイラを全て自分で作るのは大変なので、ここではある程度必要とされる機能を提供し、意味解析・コード生成の部分を各自に作成してもらう。

ここで提供する機能は次のものである。提供する機能一式は icho 上の~nakai/IP2/pl0d/pl0d に入っている。このディレクトリ内の README に各ファイルの簡単な説明がある。適宜コピーして使われたい。

1. 記号表クラス (LIST.java, SymTable.java)

2. ノード間の情報受渡しのためのクラス (Information.java)
3. コード生成に関連するクラス (Code.java, CodeGen.java, CodePtr.java, Mnemonic.java)
4. ドライバクラス (Main.java)

2.2 PL0 マシンについて

PL0 マシンはいわゆるスタックマシンと呼ばれるものである。スタックマシンでは後置記法の演算と同様に引数の処理をし、結果をスタックに積み、それをもとに演算を行なう形式をとる。基本的には演算を行なった結果はスタックに残される。例えば、中置記法の 1+2 に対応する後置記法 1 2 + では、1 をスタックに積み、2 をスタックに積み、+ は、積まれている 2 つの被演算子に対して、演算を行ない、結果をスタックに積む。

2.3 PL0' コンパイラの作成 (前半)

sablecc へ与える PL0' の文法記述を提供する。これを図 10 に示す。これは icho 上に次のように置いておく。

```
~nakai/IP2/pl0d/pl0d.grammar
```

また、コード生成にあたっては pl0 仮想機械のコードを出力するとする。ここでの pl0 仮想機械の仕様は中井の別の授業「プログラミング言語処理系」の「仮想機械」のテキストを参照されたい。また、icho 上にこの仮想機械のプログラムを置いた。

```
~nakai/IP2/pl0d/pl0i_src
```

にその一式が入っている。また、その中には実行可能形式もある。この仮想機械プログラムは `code.output` というファイルに仮想機械のコードが入っていることを前提としている。また、このファイルはカレントディレクトリにあることが前提となっている。仮想機械コードの例を示そう。

```
( INT, 0, 3 )
( LIT, 0, 3 )
( CSP, 0, 1 )
( OPR, 0, 0 )
```

これを `code.output` というファイルに記述し、pl0i を実行してみよう。これは、以下の最初に述べる例となっている PL0' のプログラムとして、`write 3.` と記述した結果である。最初の `( INT, 0, 3 )` は実行時に確保するメモリについての記述であり、ここでは 3 個の領域を確保している。3 はデフォルトで必要となる数である。変数が n 個あれば $3+n$ を確保する必要がある。`( LIT, 0, 3 )` は定数 3 をスタックに積むことを意味する。`( CSP, 0, 1 )` は組み込み関数である `write` (引数を画面へ出力) を呼び出す。`( OPR, 0, 0 )` は手続き呼出しの終了を意味するが、メインプログラム内に記述された場合は実行停止を意味する。

2.3.1 意味解析処理の概要

sablecc で意味解析をする際、各ノードを訪問し、そのノードで得た情報を他のノードを訪問している時に利用できるようにするには、どこかに保存しておく必要がある。sablecc (あるいは Visitor パターン) としては、各ノードに情報を保存するのではなく、各ノードに関連づけてどこかに保存するようにする。このため、ハッシュ(Hashtable クラス) を利用する。ハッシュは連想記憶を実現する方法の 1 つである。すなわち、あるデータと別のデータを関連付ける方法である。配列は同一の型の要素を複数まとめて扱うものであり、ある要素を指定するには整数のインデックスを用いる。ハッシュの場合、インデックスに相当するものに一般のオブジェクトを使用することができる。ここでは「ノード」と「情報」を関連付けて、保存しておき、必要になった際、「ノード」を与えることでハッシュからその「情報」が返されてくる。

Java でハッシュを利用するための記述は次のようになる。

```
public Hashbtable<Node, Information> codetable
  = new Hashtable<Node, Information>(10001);
```

`Node` はノードクラスである。`Information` はこの意味解析器において情報を受渡しするための器である。コード (列) を格納する `CodePtr`, 名前を格納する `String`, レベルや変数の個数などを格納する `int` の要素を持つ。詳細 (と言ってもコンストラクタとアクセッサくらい) はソースを参照されたい。

`CodePtr` は 1 つのコードを表す `Code` を要素とするリストクラスである。詳細はソースを参照されたい。

Code は、1 つのコードを表すクラスである。PL0 機械のコードは上述のように 3 つ組になっており、それぞれを格納する int の要素とリストを形成するためのポインタからなる。詳細はソースを参照されたい。

CodeGen は定数とラベルを作成するための static メソッドを定義している。定数とは PL0 機械のコードを表すものである。Mnemonic は PL0 機械のコード名と実際のコード (番号) の組合せを保持するためのものである。

PL0' では、変数、関数、定数という名前のカテゴリがあり、この他、スコープの切れ目を意味するブロックがある。記号表を表すクラス SymTable ではこれらの定数と記号表への登録などのメソッドが定義されている。以下、簡単にこれらを説明する。

addlist 記号表への一要素の登録。

search_all 記号表内全体での探索。

search_block 現在のブロック内だけの探索。

delete_block 現在のブロックを記号表から削除。

2.3.2 write 文

まず、write 文を実装し、定数を表示できるようにしてみよう。PL0' のプログラムの一番簡単なものは write 3. のような形になっている。すなわち、文とプログラムの終りを示すドット (.) からなる。文には色々あるが何かを実行したことがわかりやすいのは画面表示の命令であるため、write 文を実装する。実際には write の後ろに式が書けるが、最初は簡単に定数 1 つのみを扱うことにしよう。

PL0' の文法を見ると、定数の単独の式は f = number の規則による。一方、write 文は、statement の右辺が{write} write expression となっており、expression が左辺の規則では、右辺が {e} e となっているものが単一の式からなる。同様に f にたどり着くには t = {f} f と e = {t} t が適用される。

write 文の上位は、左辺の記号が statement なので、それを右辺に持つ block = declparts statement となり、プログラム全体は program = block dot となる。

以上が入力として引数が整定数である write 文に関与する生成規則である。そして、コンパイラとしてとりあえず、write 3. のような入力を受け付けるためにこれらの生成規則に関連する意味規則を作成していかなければならない。

では、まず初めに f = number の規則に対応する意味解析メソッドを定義しよう。なお、他のファイル同様、Translation.java の雛型を用意してある。コンストラクタ、ハッシュテーブルなどを設定するところまでは記述してあるので、これに各ノード毎のメソッドを入れていく作業を行なえば良い。これを図 5 に示す。

```
 1 package pl0d;
 2
 3 import java.util.*;
 4 import pl0d.node.*;
 5 import pl0d.analysis.*;
 6 import pl0d.*;
 7
 8 public class Translation extends DepthFirstAdapter
 9 {
10
11   public Hashtable<Node, Information> codetable
       = new Hashtable<Node, Information>(10001);
12   private SymTable st;
13   private int level = 0;
14
15   public void debug(String s){
16     System.out.println(s);
17   }
18
19   Translation(SymTable st){
20     this.st = st;
21   }
22
23 }
```

図 5: Translation.java

図 5 を簡単に解説しておく。1〜6 行目で必要なパッケージをインポートしておく。クラス Translation は DepthFirstAdapter を継承しているので走査のコードについては記載する必要はない。各ノードを訪問した際にどのような処理をするかをメソッドとして記述しておけば良い。11 行目ではハッシュテーブルのインスタンスを得ている。ハッシュテーブルは木の走査の途中において情報の受渡しに使われる。なお、JDK 5.0 から導入された総称を利用しているため、<Node, Information> のような記述が付加されている。

f = number の規則に対応するメソッド名は次のようになる。

```
public void outANumberF(ANumberF node)
```

ここではまず、トークンノード (Number) からテキスト情報を得て、それを数値に変換する。得られた値が n だとすると、整定数を表す PL0 マシンコードは (LIT, 0, n) なのでそのようなコードを作る。作ったコードを上位のノードに持ち上げて、他のコードと統合することになるため、持ち上げるためのクラスのオブジェクトを作り、このノードをキーとしてハッシュテーブルに登録する。

このメソッドの完成版を次に示す。

```
public void outANumberF(ANumberF node){
  int x = Integer.parseInt(node.getNumber().getText());
  CodePtr c = new CodePtr(new Code(CodeGen.O_LIT, 0, x));
  Information i = new Information(c);
  codetable.put(node, i);
}
```

ここでは 1 つのコードの作り方、このノードにおける情報の作り方に着目して欲しい。以降、基本的には同様の操作となっていく。

このコードを簡単に説明する。2 行目は、対応する規則 f = {number} number における右辺の number の部分のノードを、node.getNumber() によって取り出している。このノードは getText() というメソッドを持っており、その字面 (入力としての文字の列) を返す。これは字句解析により、数字が切り出されているので、それを利用して、Integer.parseInt により数値へ変換している。その結果を変数 x に格納している。3 行目では定数用のコードを作成している。4 行目では、得られたコードを上位のノードへ渡すため、ハッシュへ登録するための情報を表現するオブジェクトを生成している。5 行目で、そのオブジェクトをハッシュテーブルに登録している。

ここまでは number が子ノードであるノード f についての操作であった。今度は t = {f} t によって情報が 1 つ上のノードへ引き渡される部分を記述しよう。

この部分のメソッドは次のようになる。

```
public void outAFT(AFT node){
  Information tmp = codetable.get(node.getF());
  codetable.put(node, tmp);
}
```

こちらは非常にシンプルである。このノードですることは下位のノードの情報を得て、上位に渡すだけである。ハッシュテーブルから子ノードをキーとして (先ほど登録した) 情報 (Information のインスタンス) を取り出し、このノードをキーとしてハッシュテーブルに登録する。そうすると、e = {t} e に対応する意味処理も同様になる。その次は、expression = {e} e であるが、これも同様の処理となる。

次は statement = {write} write expression の処理である。

PL0 マシンでは、expression のコード、write のコードの順に評価が進む。すなわち、expression が表している算術式を計算し、その結果をスタックに積み、スタックトップの値を画面に表示する、という動きとなる。そうするとここで作成しなければならない意味処理は次のようになる。

1. expression から得られるコードを取り出す。情報はハッシュテーブルから子ノードを指定することで取り出す。取り出した情報にはコード列 (CodePtr のインスタンス) が含まれている (この場合、(LIT, 0, n))。
2. 取り出したコード列の末尾に write の呼び出しに相当するコードを追加する。これは CodePtr の add メソッドを使えば良い。ここで作るコードは (CSP, 0, 1) である。コードは Code クラスのインスタンスとして作る。作り方は上述の outANumberF を参照のこと。
3. こうして得られたコード列を更に上位に渡すため、Information のインスタンスを作り、ハッシュテーブルに格納する。

次は block = declparts statement での意味動作である。現時点では declparts 部分を無視して、作成することにしよう。本当は declparts には変数宣言や関数宣言が入っており、そこから得る情報を加味してここでのコードを作らなければならない。それらは後ほど考えることにし、ここでは statement に対応するコード、正確には write 文だけを含んだ最終的なコードができるようにしよう。

ブロックに入るとシステム用に 3、もし変数宣言などがあればその個数分が追加されて、(INT, 0, $3+n$) というコードを出力する必要がある。そして、ブロック内のコードはそのコードに続いて statement 部のコードが連結される形になる。

ここでも write 文のコードを作った時と同様にコードを作ろう。作ったコードは同様に Information クラスのインスタンスとして、更に上位に渡す必要がある。

さて、この項の最後として、program = block dot に対応した意味処理を作ろう。ここでも基本的な操作は同じであるが、ここでは、block のコードを得て、その後ろに関数呼出しの終了を意味する (OPR, 0, 0) を付け加える。前述のようにメイン関数の呼出しの終了＝プログラムの終了である。そして最後に得られたコード全体を画面へ表示する (出力先を code.output にしてもよい)。

CodePtr には出力用のメソッド void printcode() が用意されているので、得られたコード (CodePtr のインスタンス) から printcode を呼び出せば良い。

ここまでができると、write 3. のような入力に対して、次のような出力が得られる。

```
( INT, 0, 3 )
( LIT, 0, 3 )
( CSP, 0, 1 )
( OPR, 0, 0 )
```

演習

ここまでをプログラムとして実現せよ。プログラムができたら、適当な文を与えてみよ。

2.3.3 さらに算術式を使えるように

さて、次のステップとして、write の引数としてもう少し式が書けるようにしよう。まだ、変数は使用しないが、1+2*3 のような式を書けることを目標にする。

まだ、整定数のみなので、f = {number} number という規則については特に変わらない。次に t = {mult} t mult f | {div} t div f という規則についての意味処理を考える。右辺の t と f についての情報を得て、それらのコードをその順に並べ、最後に演算のコードを置くことでこれらの構文規則に対するコードが作られる。かけ算を表す PL0 コードは (OPR, 0, 4) であり、割算のそれは (OPR, 0, 5) である。これまで同様できたコードは情報として上位へ渡す。

足し算、引き算も同様に作れるであろう。足し算の PL0 コードは (OPR, 0, 2)、引き算のそれは (OPR, 0, 3) である。

ついでに expression を左辺とする規則まで作ってしまおう。単項のプラスは特に何もする必要がない。単項のマイナスは 0 からオペランドの値を引けば良い。

もう 1 つついでに括弧のある式 (f = {par} ...) の部分も作ってしまおう。

演習

ここまでのプログラムを実現せよ。ここまでで write 文に続いて定数をオペランドとする演算式を受け付けられるようになったので、様々な入力を与えて動作を確認してみよう。

なお、以下では同様にして、それぞれの部分を作成できたら、適宜、動作のチェックをしてみよう。チェックとしてどのようなことをすべきかを考えること。

2.3.4 複文

次は複文を扱う構文を処理しよう。複文とは、複数の文のことでこの文法においては statement = {begin} ... の部分がそれに相当する。関連するのは statement_list を左辺とする規則である。ここでは、この他に writeln も利用できるようにする。また、ここでは空文も扱うことにする。空文とは文字通り何もしない文であり、文法規則では statement = {no} がそれに当たる。

まず、writeln から処理をしよう。ここでは writeln のコードである (CSP, 0, 2) を作り、上位に渡す。

次に空文の処理をしよう。空文の処理は簡単で、コードのない情報を上位に渡すのみである。

次に statement_list の処理をしよう。まず、{single} であるが、これは下位で作成されたコードを上位に渡すのみとなる。{list} の方は右辺の statement_list と statement のコードを連結し、上位に渡す。ただし、ここでは空文に配慮しなければならない。前者 (右辺の statement_list) のコードが空であれば後者の情報をそのまま上位に渡す。後者のコードが空であるかどうかは気にしなくてよい。前者のコードが空でなければ前者と後者のコードをつなぐ。この場合もつなぐ作業は CodePtr の add メソッドが行なってくれるため、後者が空かどうかはチェックがいらない。

最後に {begin} の部分であるが、statement_list から得られるコードをそのまま上位に渡せば良い。

ここまででつぎのような入力を処理できるようになった。

```
begin
  write 3+4;
  writeln
end.
```

また、次のようなコードも受け入れられることを確認しよう。

```
begin
  ;
end.
```

すなわち、begin と end で囲った中に 2 つの空文があるプログラムである。ちなみにこれが大丈夫であればセミコロンが 10 個並んでいたとしても問題ない。

PL0' では、セミコロン (;) は文の終了を表すターミネータではなく、文の区切りを表すセパレータになっている点に注意しよう。

2.3.5 変数の導入 – 変数宣言 –

今度は変数を導入しよう。これにより、代入文などが利用できるようになり、より便利になる。

そのうちブロックの入れ子の概念が出てくるが、整数型の変数 level を用意し、ブロックに入ったらそれを 1 増やし、ブロックから出る際には 1 減らすようにすることで、いまどのレベルにいるかを記録できる。

それでは、まずは変数の宣言から見ていこう。宣言を扱うには記号表が必要であるが、ここでは用意している SymTable のインスタンスを使う。既に main メソッドの中でインスタンス化されているので、ここでは利用するのみとなる。

変数の宣言に関連する規則は、declparts, declpart, decl, vardecl を左辺とするものであり、さらに idlist という識別子の並びを表す生成規則も関連する。

この他、ブロックが 1 つのスコープになるのでブロックに入る直前もしくは直後にはブロック、すなわち新しいスコープに入ったことの処理をしなくてはならない。また、block を左辺とする規則では、ブロック内のコードをまとめる作業も行なうがその最初のコードはブロック内の変数の数が要求される。ここまでは (INT, 0, 3) としていたが、この 3 の部分に変数の個数 n が加わることになる。

では、最初に idlist = {single} id の意味規則を作成しよう。まず、二重登録にならないかどうかのチェックをする。これには SymTable に用意されている search_block メソッドを使用する。search_block は LIST 型の値を返す。null を返した場合は登録がなく、それ以外は登録されていることを意味する。LIST クラスは線形リストの要素となっており、ここでは識別子 (名前) の登録情報として、次のフィールドを持つ。

String name 識別子の名前

int kind 識別子の種類。ここでは、変数 (VARIABLE)、ブロック (BLOCK)、関数 (FUNC)、定数 (CONSTANT) の 4 種類。SymTable クラスで定義されている。

int offset 実行時環境におけるオフセット

int level スコープのレベル

params 関数の場合の引数の個数

LIST prev 1 つ前の要素

記号表への登録には addlist メソッドを使う。このシグネチャを次に挙げる。

```
public void addlist(String name,
                    int    kind,
                    int    offset,
                    int    level,
                    int    fparam){
```

最後に何個の変数が宣言されたかを最終的には得なければならない。これは変数用のメモリ確保の時点で必要となる。この規則の処理では情報として 1 個であることを上位に伝えれば良い。ここでは宣言のチェックと記号表への登録が仕事となり、コードを作ることはない。

以上からここでの処理の流れは次のようになる。

1. 名前の取得。これは葉ノードである id ノードが保持している。これを取得するには node.getId().getText() のようにする。

2. search_block により二重宣言でないか確認。

3. 二重宣言でなければ addlist を使って記号表へ登録。もし二重宣言ならばその旨を表示し、プログラムを終了する。オフセット (offset) は後で計算するため、ここでは 0 を指定しておく。

4. ここでは識別子 1 つを処理したため、Information のインスタンスを作って、情報として 1 を上位に伝える。

次は id_list = {list} の方の処理である。ここは基本的には id_list = {single} の場合と同じである。異なるのは右辺の id_list の持つ情報としての識別子の個数にここでの id に対応する 1 を加えたものがこの規則まで処理した際の識別子の個数となり、それを情報として上位に渡す。

vardecl を左辺とする規則では、単に情報を上位に渡すのみである。decl = {variable} も同様である。declpart = {decl} では、関数の宣言を処理した後は、それらのコードについての処理が必要になるが、現時点における変数宣言だけを処理するに当たっては情報を上位に渡すのみでよい。ここでは右辺のうち、declpart と decl であがってきた識別子の数を足したものが上位に渡す情報となる。

declpart = {no} の規則では宣言は 1 つもないため、変数の数も当然 0 となる。この情報を上位に渡す。

declparts を左辺とする規則まで還元が進むと、そのブロック内の宣言情報はすべて集めたことになる。この時点でバックパッチをし、各変数のオフセットを求める。これはすでに記号表クラス SymTable に持たせてあるので、単に個数を引数として、そのメソッド backpatch を呼び出せばよい。オフセットは PL0 マシンが実行する時に使用するメモリ上での相対的な位置のことである。関数の呼出しごとに必要なメモリが確保されるが、このとき、どの位置にどの変数があるかを決めなければならない。

ブロックに入ると、レベルが 1 つ上がるし、それにともない記号表でもブロックの区切りがわかるようにしなければならない。すなわち、ブロックに入る直前に記号表にブロック要素を登録し、レベルを 1 つ増やす作業をする。これは、ブロックに入る前に行なうため、inABlock に記述する。

block を左辺とする規則の意味処理は少し書き換えることになる。これまでは宣言部を無視してきたが宣言部から情報があがってくるので、これを利用する。現時点では宣言部のコードは存在しないため、単に宣言部から得られる変数の個数を取り出し、(INT, 0, 3) の 3 の部分にその個数を加えるようにする。

ここまでで変数の宣言を扱えるようになった。ここまでを実装したら、試しに変数宣言を入れたプログラムを与えてみよう。

```
var i, j, k;
.
```

これにより、次のような結果が出ていればよい。

```
( INT, 0, 6 )
( OPR, 0, 0 )
```

2.3.6 変数の導入 – 変数の使用 –

次は変数の使用の部分の処理を作る。変数を使用するところは read 文、式、代入文である。まずは read 文の実装を考えよう。

read 文は標準入力から値を得て、引数となる変数に代入する文である。この規則における意味解析処理は、まず、引数となる変数が宣言されているかを調べる。調べる範囲は現在のスコープだけではなく全てであるため、SymTable の search_all を使用する。ここでは、その変数が宣言されているかチェックし、宣言されていなければその旨のエラーメッセージを出して、プログラムを終了する。また、宣言されていたとしても変数として宣言されていなければ代入できないため、変数以外だった場合にもエラーメッセージを出力してプログラムを終了する。

read 文に対応する PL0 マシンのコードは、(CSP, 0, 0) である。これにより、PL0 インタプリタのスタック上に入力された値が積まれる。これを指定された変数に格納する。すなわち、その変数用に割り当てられた番地へと格納する命令を実行する。こちらは (STO, l, o) という形式になっている。l の部分には現在のレベルとその変数が宣言された時点でのレベルの差が入る。o にはその変数のオフセットが入る。その変数のレベル、オフセットともに記号表を引いた結果として search_all の返り値に存在する。これらを取り出し、それぞれ当てはめれば良い。CSP

と STO のコードをつなげたものがここでのコードとなり、それを上位に渡す。

次は算術式中の変数の使用を扱う。これができると、次のようなコードを試すことができる。

```
var i;
begin
  read i;
  write i;
  writeln
end.
```

該当する規則は f = {ident} である。定数宣言を扱っていないが、定数であることも想定して意味処理を行なうようにする。ここでの処理も基本的には read 文と同じであり、当該識別子が宣言されているか、変数か定数かチェックし、該当しなければエラーメッセージを出してプログラムを終了する。変数である場合、その変数に割り当てられているメモリ領域から値を読み出す。PL0 マシン上の動作としては取り出した値をスタックトップに積む。このための PL0 コードは (LOD, l, o) である。l と o は上述のものと同じである。もし、定数であった場合は LIT 命令により、定数をスタックに積む命令を作る。あとはそれらを上位に渡す。

次は代入文の意味処理を扱おう。statement = {ident} の部分がそれに相当する。ちなみに PL0' では代入の記号は := である。これまでと同様、ここでの処理の流れは次のようになる。

1. 代入文の左辺の変数が正しく宣言されているかのチェック
2. 代入文の右辺のコードの取得
3. 右辺のコードの最後に左辺の変数に相当するメモリ位置へ値を格納するコード (STO) をつなげる
4. 情報を上位へ渡す

ここまでで変数へ値を入れることができるようになった。これで、例えばつぎのようなサンプルプログラムを実行できる。

```
var i;
begin
  i:=1*2;
  i:=i*4;
  write i;
  writeln
end.
```

2.3.7 条件文

次は if や while といった条件文を扱えるようにしよう。条件文を扱うには条件式を評価できなければならない。図 10 では condition を左辺とする規則がそれに当たる。

ちなみにこれらの条件式は非結合的なので再帰的な文法になっていない (条件式の要素である式はもちろん再帰的な生成規則からなる)。

condition = {odd} の部分は、奇数かどうか判定するための組み込み関数 odd の適用である。これは引数が 1 つである単項演算子である。PL0' で利用できる他の条件式は二項演算子からなる。それらは、=, <>, <, >, <=, >= である。= は C 言語と違って代入記号ではなく、比較演算子 (等号) である。<> は等号の否定である[1]。

これらのコードは、算術式のコード生成と同様である。二項演算子の場合は、左の被演算子のコード、右の被演算子のコード、その演算のコードの順につないだものがその条件式のコードとなる。単項演算子の場合は、被演算子、その演算のコードをこの順につないだものがその条件式のコードとなる。各被演算子のコードの取得、その演算子のコードの生成、それらの合成 (連結)、そしてそれらを上位へ渡す、という手順はこれまでと同様である。ただし、ここでは右辺に同じ記号 (expression) が二回出てきているため、それらを区別するためにそれらに [left] と [right] が付けられている。このため、子ノードとしてそれぞれを得るには、getLeft(), getRight() を用いる。

各演算は (OPR, 0, a) の形式になっており、a と記号の対応は次の通り。

[1] 不等号とはいわない…。

記号	a
=	8
<>	9
<	10
>=	11
>	12
<=	13

さて、ここまでで条件式を扱えるようになった。次は if 文の意味処理を実装しよう。PL0' の if 文は else 部を持たない。

PL0 マシンでは条件ジャンプと呼ばれる命令が用意されている。これは直前に評価した内容が 0 であれば指定されたところへジャンプするというものである。また、ここで用意している PL0 マシンではラベルが利用できる。ジャンプ先としてラベルを使用できる。

if 文のコードは次のようになる。JPC は直前に評価した式の値 (スタックトップに載せられている) が 0(偽を表す) ならばラベルへとジャンプする。

```
…条件式のコード列…
( JPC, 0, ラベル )
… statement のコード列…
( LAB, 0, ラベル )
```

「ラベル」の部分には整数が来る。コンパイル中に一意な整数を生成するメソッドが CodeGen に用意されている。そのシグネチャは次の通り。

```
static int makelabel()
```

さて、ここまでで次のようなプログラムに対してコードを生成することができるようになった。

```
var i;
begin
  read i;
  if odd i then write i;
  writeln
end.
```

次は while 文を実装しよう。基本的な考え方は if 文と同じである。while 文のコードは次のような並びになる。

```
( LAB, 0, ラベル 0 )
…条件式のコード…
( JPC, 0, ラベル 1 )
… statement のコード列…
( JMP, 0, ラベル 0 )
( LAB, 0, ラベル 1 )
```

JMP は無条件のジャンプで、指定したラベルへ制御を移動する。

2.3.8 定数宣言

PL0' には次のような形式の定数宣言がある。

```
const max = 100;
```

定数宣言に関連する規則は、id_assign, id_assign_list, constdecl を左辺とするもの、および decl = {constant} である。

実際に識別子が現れるのは、id_assign を左辺とする規則のみであるので、それに対応する意味規則として、同一スコープ内に同じ識別子が宣言されていないかチェックし、同一名の登録がなければ定数 (st.CONSTANT) として記号表に登録する。この際、number で与えられた文字列から数値を作り出し、addlist の最後のパラメタ (LIST 内の params) に指定しておく。なお、これらの規則では特に情報を上位にあげる必要がないが、declpart = {decl} の規則における意味処理では、下位の変数の数を収集している。この際、decl = {constant} の規則からは 0 が返されなければならない。

また、識別子の使用 (f = {ident}) においては、その識別子を記号表から探索し、見つかった場合に定数かどうかチェックし、そうであれば記号表に収められているその値を取り出し (getParams を使う)、定数をスタックに積む PL0 マシンの命令 (LIT, 0, n) を生成する。

以上で関数宣言と関数の使用を除き、PL0' の機能を実装したことになる。

2.4 PL0' コンパイラの作成 (後半)

最後に残ったのは関数の宣言と関数呼び出しである。まず、関数宣言から処理を考えていこう。

2.4.1 関数宣言の処理

関数の宣言に関係する生成規則は、decl = {function}, funcdecl, fhead, fid である。まず、fid から見ていく。

fid は関数宣言における関数名を表している。ここでは記号表を探索し、同じ名前がないかチェックする。なければその識別子を記号表に登録する。見つかった場合には二重宣言のエラーであるため、その旨のメッセージを出力してプログラムを終了する。なお、ここではこのあと関数の引数が続き、再びブロックに入る。関数の引数はそのブロック内での変数宣言として扱うので、この規則 (fid = id) に対応する意味規則の最後にレベルの増加と記号表へのブロック要素の追加も行なう。

次は関数の引数を扱う規則を対象にしよう。これには fhead, param を左辺とするものがある。まず、param を左辺とする規則であるが、これは右辺が空か {id_list} id_list からなる。ここでは上位に引数の個数を渡す。空の場合はパラメタ無しなので、引数無しの Information のコンストラクタ呼び出しによりインスタンスを作り、上位に渡す。引数無しの場合、フィールド val は 0 に初期化される。id_list の場合、下位で変数の並びを扱っており、記号表に登録されていく。下位の id_list の情報から引数として並べられた変数の個数がわかる。ここではこの情報をそのまま上位に渡せば良い。

関数呼び出しの際のスタックの様子を図 6 に記す。仮に f(a, b, c) のように呼び出すとする。図では上側がスタックの底となっている[2]。

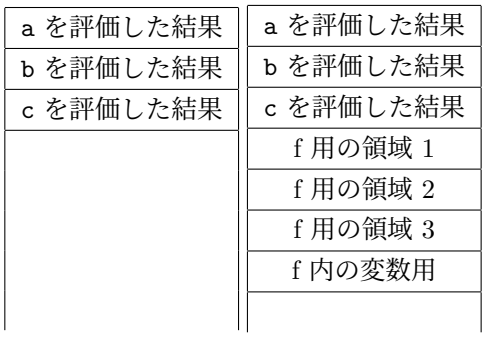

図 6: 関数呼び出し時の実行時スタックの様子

関数 f の呼び出し前には各引数を評価する。図 6 の左のようにそれぞれがスタックに積まれる。そして実際に関数のコードへ制御が移る。ブロックに入るとシステム用に 3 の領域が取られる。その他そのブロック内に変数宣言などがあればそれらがスタックに領域としてとられる。呼び出し後は「f 用の領域 1」を基準にし、各引数にアクセスする場合には、a は基準位置から見て -3 の位置に、b は基準位置から見て -2 の位置に、c は基準位置から見て -1 の位置にある。

さて、関数宣言の話に戻る。fhead を左辺とする規則では、関数名、引数について、記号表にバックパッチをあてる。すなわち、各引数が上述のように適切な位置を指し示すようにオフセットを設定する。また、関数呼び出しの際に引数の個数をチェックするが、そのために引数の個数を関数名を登録している要素に登録する必要がある。また、関数呼び出しの PL0 マシンコード (CAL) では、呼び出し先のラベルが必要となる。これもセットする。なお、実際のラベルの設定は 1 つ上位の funcdecl を左辺とする規則で行なう。これらの作業は SymTable の proc_params を呼び出すことで行なえる。

したがって、fhead を左辺とする規則では次のことを行なう。

1. ラベルの生成
2. 引数についての情報の取得
3. 引数の個数と生成したラベルによる proc_params の呼び出し
4. ラベルを上位へ持ち上げる。

ラベルは整数なので、変数の個数と同様に上位へ持ち上げれば良い。

関数宣言部である funcdecl を左辺とする規則の意味処理を考えよう。

ここでは下位の要素として、fhead と block がある。fhead からはラベルが上がってくる。まず、この関数へラベルを設定するコードを作る。次に block 部分のコードをそれに連結させる。最後に関数の終りを示す (OPR, 0, 0) を連結する。そしてこのコードを上位へ渡す。

関数宣言の上位の規則は decl = {function} である。この規則では単に下位からのコードを上位に受け渡せば良い。ここまでで関数宣言本体の意味処理は

[2]慣習上、このように書く。

終りだが、これらは更に上位に影響を与える。すなわち、declpart, declparts, block を左辺とする規則である。

declpart = {no} については、上位へは変数宣言がないという意味で個数 0 を情報として渡す。

declpart = {decl} では、下位で生成されたコードを上位に渡す必要が出てきた。また、右辺の構造が declpart decl のようになっているため、それぞれの非終端記号から情報を集め、合成する必要がある。ここでの処理の手順は次のようになる。

1. declpart decl のそれぞれから情報を得る。
2. 得られた情報から下位で宣言された変数の総数を求める。
3. それぞれのコードを統合するのだが、どちらかあるいは両方が null (コードがない) ということもあるので、それを考慮してコードを連結する。
4. 情報として、連結したコードと変数の総数を上位に渡す。

declparts を左辺とする規則では、基本的にこれまでと同じ処理で良い。情報として下位で得られたものをそのまま上位に渡す点に注意。

最後に block の意味処理であるが、ここでは declparts でコードが作られているかも知れない。このブロックの実行は statement 部から始まる。このため、次のような処理手順になる。

1. declparts, statement から情報を得る。declparts から得た情報から変数の個数を抽出しておく。
2. ラベルを作る。
3. 作ったラベルへのジャンプのコードを作る。
4. 上記のコードに declparts からのコードを連結する。
5. 上記のラベルのコードを作り、連結する。
6. INT 命令を $3+n$ になるようにする。n は上で求めた変数の個数。
7. statement から得られたコードを連結する。
8. 得られたコードを上位へ渡す。
9. ブロックから出るのでレベルを下げ、記号表から当該ブロックの情報を削除する。

2.4.2 関数呼び出し

関数呼び出しに関する文法規則は f = {emptypar}, f = {identpar} である。また、この下位には expressionlist を左辺とする規則がある。

まず、引数無しの関数呼び出しである f = {emptypar} から始めよう。まず、右辺の最初の id が関数として宣言されているかチェックする。関数ではない場合、エラーメッセージを出力して終了する。関数として宣言されている場合、記号表を引いて得られた情報から引数の個数が 0 であるかどうかチェックする。もし違っていたら宣言と一致していないため、エラーメッセージを出力し、プログラムを終了する。ここまでのチェックに通れば、その関数呼び出しのコードを作り、上位へ渡す。関数呼び出しの PL0 マシンのコードは (CAL, l, o) となる。l の部分には呼び出し元の現在のレベルと記号表から得られた情報によるその関数のレベルの差が入る。o の部分にはラベルが入る。

また、関数から値を返すには return 文を使用するので、この意味処理を実装しよう。return に相当する PL0 マシンコード[3] は (RET, 0, a) の形をしている。a の部分には呼び出されている関数の引数の数が入る。これは記号表の中を末尾から順に探索し、現在のレベルより 1 つ小さい関数があればそこに情報として格納されている。これを行なうメソッドは SymTable に用意されている。searchf がそれで int 型の 1 つの引数を取る。与えるのは現在のレベルである。return 文では、キーワード return の後ろに式が来るため、ここで作成する PL0 コードは、その式のコードの後ろに return のコードを連結させたものとなる。そして、そのコードを上位に渡す。

さて、ここまでで次のようなサンプルを実行できるようになった。

```
function f()
return 101;
```

[3] この部分は中井による拡張。オリジナルの PL0 マシンにはない。

```
begin
  write f();
  writeln
end.
```

最後に引数ありの関数呼び出しを扱う。基本的には引数無しの関数呼び出しと同じである。異なるのは引数の個数チェックが下位の expressionlist から得られたものを用いて行なうところである。また、ここで生成するコードは expressionlist のコードのあとに関数呼び出しのコードをつなげたものである。

引数の処理は expressionlist を左辺とする規則で行なう。{single} の方では、下位の expression のコードと、引数の個数 1 を情報として上位に渡す。

{list} の方では、右辺の expressionlist と expression のそれぞれのコードをこの順で連結する。また、引数の個数は、右辺の expressionlist から得られる数 +1 となる。これらの情報を上位に渡す。

ここまででですべての構文について意味解析・コード生成を実装したことになる。関数呼び出しを使えるようになったことで再帰的なアルゴリズムを記述できるようになった。階乗計算を行なう PL0' プログラムを図 7 に示す。このプログラムでは最初に入力待ちになるので非負整数を入力するとその階乗を計算する。同様に、標準入力から与えた数 (番目) のフィボナッチ数を計算するプログラムを図 8 に、標準入力から与えた数 (番目) までのフィボナッチ数を計算するプログラムを図 9 に掲載する。

```
function f(n)
begin
  if n < 1 then return 1;
  return n * f(n - 1);
end;

var i;

begin
  read i;
  if i>0 then
    begin
      write f(i);
      writeln
    end
end.
```

図 7: 階乗計算を行なうプログラム

```
function fib(n)
begin
  if n = 1 then return 1;
  if n = 2 then return 2;
  return fib(n - 1) + fib(n - 2);
end;
var i;
begin
  read i;

  write fib(i);
  writeln
end.
```

図 8: n 番目のフィボナッチ数を計算するプログラム

```
function fib(n)
begin
  if n = 1 then return 1;
  if n = 2 then return 2;
  return fib(n - 1) + fib(n - 2);
end;
var i;
var j;
begin
  read i;
  j := 1;

  while j<=i do
    begin
      write fib(j);
      writeln;
      j := j+1;
    end
end.
```

図 9: n 番目までのフィボナッチ数を計算するプログラム

3 おわりに

時間が足りなくて、やっつけで作ったのでこちらが準備したクラスはあまりよいできとは言えません。余力があればこれらの改造 (改善) もしてみて下さい。

参考文献

[1] Étienne Gagnon: SABLECC, AN OBJECT-ORIENTED COMPILER FRAMEWORK, Master's thesis, McGill University, Montreal, 1998.

```
Package pl0d;

Helpers
    number = ['0' .. '9' ];

  tab = 9;
   cr = 13;
   lf = 10;

Tokens
    lpar = '(';
    rpar = ')';
    plus = '+';
   minus = '-';
    mult = '*';
     div = '/';
    semi = ';';
   comma = ',';
     dot = '.';
   coleq = ':=';
      eq = '=';
     neq = '<>';
      lt = '<';
      gt = '>';
      le = '<=';
      ge = '>=';
   const = 'const';
     var = 'var';
function = 'function';
   begin = 'begin';
     end = 'end';
      if = 'if';
    then = 'then';
   while = 'while';
      do = 'do';
  return = 'return';
    read = 'read';
   write = 'write';
 writeln = 'writeln';
     odd = 'odd';
      id = ['a'..'z'] (['a'..'z'] | ['0'..'9'])*;
  number = number+;

   blank = (' ' | tab | cr | lf)+;

Ignored Tokens
   blank;

Productions
         program = block dot;

           block = declparts statement;

       declparts = declpart;

        declpart = {no} |
                   {decl} declpart decl;

            decl = {constant} constdecl |
                   {variable} vardecl |
                   {function} funcdecl;
```

```
       constdecl = const id_assign_list semi;

   id_assign_list = {list} id_assign_list comma id_assign |
                    {single} id_assign;

        id_assign = id eq number;

          vardecl = var id_list semi;

          id_list = {list} id_list comma id |
                    {single} id;

         funcdecl = function fhead block semi;

            fhead = fid lpar param rpar ;

              fid = id;

            param = {no} |
                    {id_list} id_list;

        statement = {no} |
                    {ident} id coleq expression |
                    {begin} begin statement_list end |
                    {if} if condition then statement |
                    {while} while condition do statement |
                    {return} return expression |
                    {read}  read id |
                    {write} write expression |
                    {writeln} writeln;

   statement_list = {list} statement_list semi statement |
                    {single} statement;

        condition = {odd} odd expression |
          {eq}  [left]:expression eq  [right]:expression |
          {neq} [left]:expression neq [right]:expression |
          {lt}  [left]:expression lt  [right]:expression |
          {gt}  [left]:expression gt  [right]:expression |
          {le}  [left]:expression le  [right]:expression |
          {ge}  [left]:expression ge  [right]:expression;

       expression = {plus} plus e |
                    {minus} minus e |
                    {e} e;

                e = {plus} e plus t |
                    {minus} e minus t |
                    {t} t;

                t = {mult} t mult f |
                    {div} t div f |
                    {f} f;

                f = {number} number |
                    {ident} id |
                    {emptypar} id lpar rpar |
                    {identpar} id lpar expressionlist rpar |
                    {par} lpar expression rpar;

   expressionlist = {list} expressionlist comma expression |
                    {single} expression;
```

図 10: pl0d.grammar