システムソフトウェア特論'19 # 2 構文の定義

久野 靖 (電気通信大学)

2019.1.9

1 プログラミング言語の定義

1.1 言語の定義と標準

プログラミング言語を設計したり検討したり実装するためには、まず言語を正確・厳密に定義する必要があります。なぜでしょうか? たとえば言語 X の定義が正確・厳密でなかったとします。そうすると、その言語で書いたプログラムをある処理系 A で動かしたときと、別の処理系 B で動かしたときで、微妙に動作が違うかも知れません。いや、動作が違うどころか、そもそも A 用に書いた X 語のプログラムを B に持っていって動かそうとしたら、いきなりエラーで受け付けてもらえない (動き出すことすらできない)、ということも十分起こり得ます。それでは困りますね?

実はこれはコンピュータ科学の初期においては「普通のこと」でした。たとえば IBM の PL/I コンパイラ用に作ったプログラムを FACOM 用の PL1 コンパイラ (なぜか名前が違えてありました) で動かそうとしたらこの機能は無いよとかで受け付けてもらえないなどは普通でした。

今日では、プログラミング言語の定義方法についての知見が進歩したので、まず X の言語仕様を厳密に定義した上で標準化します。その上で、その言語仕様に合致するようにさまざまな処理系を作るので、どこの処理系でも同じプログラムが動く…はずなのですが、微妙に問題が残ることは今でもあります。具体的には次のようなことです。

- 作るときの都合上で一部の機能を「作っていない」処理系があったりする。
- 逆に便利さのためや研究のため「独自拡張」を追加した処理系があったりする。

どちらの場合でも、ある処理系用に作ったプログラムが他の処理系で動かないことは同じで、頭痛の種です。まあ、独自拡張は使わないでプログラムを書き、標準化に従っていない処理系を使わなくすればだいたい大丈夫ですが…(そのほか、処理系の実装ミスもありますけれど)。

実はこれに加えて、次のようなケースもあります。

- 言語仕様に「不定」「処理系依存」つまり仕様としては決めないので適当に作ってね、という箇所がある。

このようなところは処理系の方でとりあえず決めるので、ある処理系と他の処理系で動作が違うことになります。その問題が認識されたので、今の言語ではできるだけ処理系依存をなくすように厳密に記述しますが、そのために言語仕様が大きく複雑になるわりには、完全に定義するというのはなかなか困難という問題を抱えています。

1.2 プログラミング言語をどのように定義するか

では次の問題として、ではプログラミング言語はどのようにしたら正確・厳密に定義できるでしょうか。たとえば、次のような課題にあなたはどのように答えますか。

演習 2-0 外国の人 (日本語を勉強中) に、「元号で年月日を書くやりかた」を教えるとしたら、どのようにするか考えなさい。西暦への換算はとりあえず後にして、まずは表記方法だけでよい (曜日も不要)。

(実際に少しやってみてから、この先を読むようにしてください。)

さて、上の問題をどのようにしましたか。多くの人は、まず元号として「明治」「大正」「昭和」「平成」があり、その後に年の数字 (1〜2 桁) があり (あと「元年」を忘れないように)、「年」があり、次に月の数字 (1〜12) があり、「月」があり、日の数字 (1〜31) があり、「日」がある、のように書き方の規則を説明したと思います。

そしてその後で、ではそれがどのような意味になるかを説明するわけです。なぜでしょうか。それは、「281 平成 26 月日 1」みたいにでたらめな書き方では解釈しようがないからです。書き方の規則に従っていてはじめて、ここはどういう意味、ということが説明できるのです。

プログラミング言語でもまったく同じで、「書き方の規則」を定めてから、それのあとで「それぞれの書き方に対応する意味」を定めます。すなわち、構文と意味をそれぞれ定めることでプログラミング言語を定義する、ということですね。

- **構文** (syntax) — 書き方、ないし「文字をどのように並べたものがその言語の正しいプログラムであるか」を定めた規則。
- **意味** (semantics) — 構文に合致した書き方のそれぞれの意味、ないし「そのように書いた場合にどのように動作するのか」を定めた規則。

意味から説明しましょう。意味についても数学のように厳密に記法から定めるやり方はあるのですが、この授業ではそれをやると大変すぎるので、「自然言語 (日本語) で」こういう書き方はこういう意味、というふうに説明することにします。ただし自然言語の弱点は曖昧さや不明瞭さですので、そのような問題が起きないように注意して記述します。

次に構文ですが、構文については何らかの決まった書き方を採用して、その書き方で記述することが普通です。それによって厳密性が保たれますし、言語の定義を読む人にとっても結局その方が簡潔で読みやすいのです。というわけで、この後はおもにその話題になります。

2 構文定義と構文木

2.1 BNF による構文定義

BNF(Backus Normal Form) とは、その名前通り Joun Backus というコンピュータ科学の先達が考案した「文法の書き方」です。**文法** (grammer) というと英文法のことが思い浮かぶかも知れませんが、「書き方の規則」という点では英語の書き方であってもプログラミング言語の書き方であっても別に変わりはないので、同じ用語を使います。英文法は English grammer ですが、プログラミング言語の場合は「grammer for programming language X」とかそういう感じで書きます。

さて話題を戻しますが、BNF では**メタ記号** (metasymbol) を用いて文法を記述します。メタ記号って? それは、たとえば C 言語ならそのプログラムには「+」とか「(」とかいろいろな記号が現れますね? BNF は C 言語などの言語の「書き方の規則を定める書き方」なので、その規則を記述するときに出て来る記号は「+などのプログラミング言語の記号」ではなく、「書き方の規則のための記号」つまり 1 レベル上の記号なのです。そういうものをメタ記号と言います。一般に「メタ (meta)」というのは「1 レベル上の」という意味ですから。

話が長くなると面白くないので細かいことは略し、とりあえず C 言語の関数定義 (の主に冒頭部分) を BNF で定義してみます (かなり略してあるので不正確ですが)。

```
関数定義 ::= 型指定 名前 ( パラメタ部 ) { 文列 }
型指定 ::= int | double | char | void | 型指定*
パラメタ部 ::= void | パラメタ
パラメタ列 ::= パラメタ | パラメタ列 , パラメタ
パラメタ ::= 型指定 名前 | 型指定 名前 []
文列 ::= ε | 文列 文
```

上の記述で「関数定義」というのはそういう文字が C 言語に直接現れるわけではなく、ここで C 言語の「関数定義というもの」を表しますよ、という指定なのでメタ記号です。この「関数定義」のように、プログラミング言語の中に直接現れないけれど、その言語の「〜というもの」を表すようなメタ記号のことを**非端記号** (nonterminal symbol) と呼びます。これに対し、丸かっことか `int` とか実際にプログラムの字面に現れるものは**端記号** (termnal symbol) と呼びます。そして、「::=」とか「|」とかはどちらでもない BNF の演算子 (メタ演算子) です (つまりプログラムには現れてこない)。なお、BNF にはいくつかもの流儀がありますが、ここでは一応筆者の好みの流儀でやっています (本質は同じです)。

「::=」は「左辺を右辺で定義する」なので、1 行目は「関数定義とは、型指定があり、名前があり、(があり、パラメタ部があり、) があり、{ があり、文列があり、} がある」と読みます。ここで C 言語の記号はそのままタイプフェースで書かれています。型指定、パラメタ列、文列はこの後で定義されます。それでは「名前」は? これは定義がないのですが、実は端記号で「C 言語の名前」(`main` とか `x` とか) を指します。ということは、「英字 (_を含む) で始まり、英数字の並んだ列」ということですね。このように、定義されていない名前はその言語の名前、文字列、数値などの端記号に対応しています。

2 行目は型指定ですが、ここで右辺に出て来る「|」は「または」を表します。つまり型指定は `int` か `double` か `char` か `void` か…その先は何でしょう? 「型指定の後に`*`」なので、たとえば `int` は型指定ですから、`int*`などが相当します。ところで `int*`も型指定と分かったので、ということは `int**` も型指定です。ということは`*`は何個あってもいいのですね。このように BNF では再帰 (定義の右辺の中に左辺が出て来る) を用いて繰り返しを表現します。

3 行目は簡単で、`void`(パラメタが無いことを表す) かパラメタ列かどちらかです。4 行目もまた再帰で、パラメタ列とは、α がパラメタ指定とすると、α だけでもいい (パラメタが 1 個) し、パラメタ列の後に「`,`」で区切って α をつけたものでもいいわけです。ということは「α」「α , α」「α , α , α」などカンマで区切った並びはこのようにして扱えるわけです。

5 行目はまた簡単で、パラメタ指定は型指定のあとに名前か、またはさらにその後に (配列を表す) `[]` をつけたものかどちらかということを意味します。

6 行目はまた目新しいもので、「ε (イプシロン)」が出てきます。これは伝統的に「空っぽの列」を意味することになっています。ということは、何もなくても文列ですし、「文列 文」は「ε 文」であってよいので、文 1 つでも文列ですし、あとは同様に再帰により何個の文の並びでも文列です。文が未定義ですが、そこから先はこれからの課題ということにしましょう。

演習 2-1 C 言語について次の部分の BNF を書いてみよ。

a. 文。式とかは後で作るものとしてよい。自分の知っている範囲の文だけでよい。
b. 式。とりあえず簡単にするため、整数定数、単独変数、四則演算だけでよい。
c. 上記に、配列、レコード、ポインタなども入れてみよ。
d. 上記に、関数呼び出しも入れてみよ。

2.2 文法と構文木

ある言語の文法が BNF で記述できたとして、実際に具体的なソースコードを読み込んだとき、それをどのように使うのでしょうか。それは「関数」とか「文」とか「式」などの非端記号と具体的なソースコードの文字列の範囲の対応をつける必要があります。たとえば、次の文法を見てください。

$$\text{列} ::= \epsilon \mid \texttt{a}\ \text{列}$$

これは、次のように考えると、「a が 0 個以上並んだ列」と分かります。

$$\text{列} \rightarrow \texttt{a}\ \text{列} \rightarrow \texttt{a a}\ \text{列} \rightarrow \texttt{a a a}\ \epsilon == \texttt{a a a}$$

ここで、入力「a a a」と構文の対応を、図**??**(1) のように木構で書き表すことができます。

この木構造は、(a) 下端には入力列 (と必要なら ϵ) が並んでいて、(b) ノードは端記号であり、(c) 各ノードから下向きの枝は必ずいずれかの文法規則にあてはまる (たとえば一番上の「列」からは「a」と「列」に枝が出ているが、それは「列 ::= a 列」に対応している)、という約束を守っています。このような木構造を**構文木** (syntax tree) と呼びます。

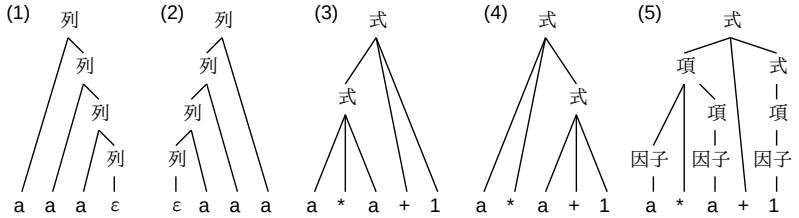

図 1: 構文木の例

ところで、先の類似品で次の文法を見てください。

列 ::= ϵ | 列 a

これも、「a が 0 個以上並んだ列」という点ではまったく同じですが、同じ入力に対応する構文木は図**??**(2) のようになります。このように、構文木は入力と文法の対応を正確に表現でき、便利なのです。

ところで、次は式に対する次の文法を見てください。

式 ::= 式 + 式 | 式 * 式 | 1 | a

これを「a * a + 1」という入力にあてはめると、図**??**(3) と (4) のように 2 通りの構文木を作ることができます。このように、入力に対して複数の構文木が作れる文法のことを**曖昧** (ambiguous) である、と言います。通常は、曖昧な文法は避けるようにします。

では、先の例を曖昧でなくするにはどうしたらいいでしょうか。2 つの構文木を見比べて、どちらが適切だと思いますか? 「乗除算は加減算より強く結びつく」のですから、(3) の形になるのが正しそうです。そこで、次の文法を見てください。

式 ::= 項 | 項 + 式
項 ::= 因子 | 因子 * 項
因子 ::= 1 | a

このようにすると、図**??**(5) の構文木しか作れなくなります。つまり、曖昧でない文法に書き換えることができたわけです。そのかわり、文法はかなり複雑になっています。[1]

演習 2-2 構文木について次のことをやってみなさい。

a. 上の曖昧でない式の文法について、「a + a * a + 1」「a + 1 + a * a」に対する構文木を描き、確かに乗算が強く結び付くことを確認しなさい。

b. 上の曖昧でない式の文法では「a + a + a」の場合、左の足し算から順に実行する。これを右の足し算から実行するように文法を変更して構文木を描き、確認しなさい (足し算だとどちらが先でも良さそうだが、引き算だと違いがある)。

c. 上の曖昧でない文法の「因子」を次のように修正する。

因子 ::= 1 | a | (式)

[1] 「因子」を無くして項の右辺に a と 1 をそれぞれ書くこともできますが、「因子」の種類が増えて来たら大変なので普通はそういうことはしません。

これでかっこのついた式を適宜考え、構文木を描いてかっこの中が強く結び付くことを確認しなさい。

d. 次の文法を考える (「式」はこれまでのものを使う)。

文列 ::= ϵ | 文 文列
文 ::= 式 ; | while (式) 文 | if (式) 文 | { 文列 }

これで while 文と if 文の組み合わさったコードを適宜考え、構文木を描きなさい。

e. 上の文法の「文」に次の選択肢を追加する。

文 ::= if (式) 文 else 文

このとき、この文法が曖昧であることを示すような例を考え、2 通りの構文木を描きなさい。C 言語か Java 言語でこのようなコードが実際にどのように動作するか調べなさい。

3 再帰下降解析による構文検査

3.1 構文検査とは

ずっとお話だとつまらないので、ここで簡単な**構文検査器** (syntax checker) を Java で作ってみます。[2] 次のような制約を設けます。

- 「言語」は 1 行ぶんの文字列で、トークンはすべて 1 文字。

そこで、たとえば次のような BNF を考えます。

prog ::= ϵ | *ab prog*
ab ::= a b

prog のように斜体で示しているのが非端記号です。もともとの BNF では<program>のように角かっこで囲むのが正式な非端記号の表し方でしたが、複数フォントが使えるときはそれを活用する方が見やすいです。先に出て来た例では、日本語の言葉はそのままで非端記号ということで扱っていました。

それで、この BNF はちょっと考えれば分かるように「ab の 2 文字が 0 個以上反復した列」になります。実際にプログラムを動かしているところを示します。

```
% java Sam21 'abab'
true
% java Sam21 'aba'
false
% java Sam21 ''
true
%
```

このように、「形があってるかどうか」だけ調べるプログラムは**構文検査器** (syntax checker) と呼ばれます。コンパイラ作るときには検査だけでは済まないですが、しばらくは簡単なので検査器だけで試していきます。実際にプログラムを見ていきましょう。

3.2 文字単位の Toknizer

前回は単語単位でよむ Toknizer をやりましたが、今回は上記のように 1 文字ずつで扱うので 1 文字ずつ取るだけの Toknizer をつくります。ソースを見てみましょう。

[2] 構文検査器ができればそれを修正して構文木を組み立てるようにもできるのですが、ここでは簡単のため検査だけです。

```
class CharToknizer {
  String str, tok;
  int pos = -1;
  boolean eof = false;
  public CharToknizer(String s) { str = s; fwd(); }
  public boolean isEof() { return eof; }
  public String curTok() { return tok; }
  public boolean chk(String s) { return tok.equals(s); }
  public void fwd() {
    if(eof) { return; }
    pos += 1;
    if(pos >= str.length()) { eof = true; tok = "$"; return; }
    tok = str.substring(pos, pos+1);
  }
  public boolean chkfwd(String s) {
    if(chk(s)) { fwd(); return true; } else { return false; }
  }
}
```

インタフェースは前回の Toknizer にほぼ合わせてあります。内部表現はファイルから読むのでなく、コンストラクタで文字列を渡すとそれをインスタンス変数 str に格納し、pos でそのどの位置を見ているかを管理します。最初は pos を-1 にしているのは、作ってすぐに fwd() を呼ぶことで最初の文字にセットするためです。

ほとんどの仕事は fwd() で行ないます。eof なら何もせず帰ります。そうでなければ、pos を 1 すすめ、もし文字列の終わりに来たら eof にします。そうでなければ、文字列のその位置を (長さ 1 の文字列として) 取り出します。文字列のメソッド length()、substring() の機能は API ドキュメントで確認してください。

さて、最後のメソッド chkfwd() は前回なかったものですが、chk() して OK だった場合はそのトークンを fwd() します。これは書きやすさのために用意しています (後で分かります)。

3.3 static の謎

この先実際のコードに入る前に、前回きちんと扱えなかった static について説明しましょう。Java では static とはひらたくいえば「インスタンスを作らなくても使える」という意味になります。

図??を見てください。クラス MyClass にはメソッド main と method1 がありますが、前者は static がついているので、MyClass のインスタンスを作らなくても呼び出せます。実際 main() はそうして呼び出していますね。これに対し、method1 はまず MyClass c = new MyClass(...) によりインスタンスを生成し、そのあと c.methods1(...) のようにして呼び出す形 (メッセージ送信記法) でしか呼び出せません。これが static の有無の違いです。

そして、変数 sx/sy と ix/iy の違いもこれに関係します。ix/iy はインスタンス変数で個々のインスタンスに付随して 1 つずつ存在するので、インスタンスのある状態でしかアクセスできません。ということは、インスタンスメソッド mehotd1 の中からは参照できますが、static メソッドである main の中からは参照できません。一方 sx/sy は static 変数なので、クラスに対応して 1 つだけ存在し (グローバル変数のようなもの)、method1 からでも main からでも参照できます。

さらに、これまではクラスは並べて書いてきましたが、クラスの中でクラスを定義することもできます。とくにクラス内で定義している static 変数をアクセスしたい場合はそのようにする必要があります。[3] これを**内部クラス** (inner class) と呼びます。

[3]正確に言えば、MyClass.sx のように外側クラス名を前置すれば外部からもアクセスできるようにもできますが、複雑

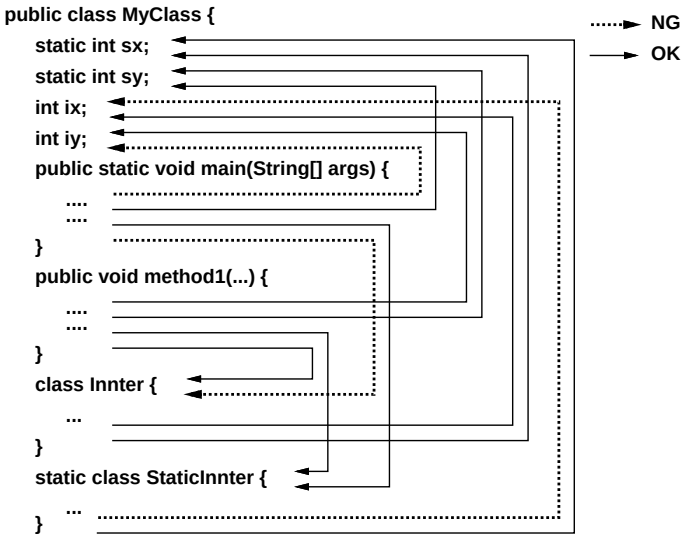

図 2: static の参照可否

内部クラスにも static ありと無しがあります。static なしの内部クラスは static なしのメソッド内からでしかインスタンスを生成できず、その中からはインスタンス変数 ix/iy がアクセスできます。つまり内部クラスのインスタンスは外側クラスのインスタンスに紐ついています。しかしこの機能は分かりにくいのであまり使いません。

一方、static つきの内部クラスは普通の (外側に書いた) クラスと同じで、static メソッドからでもインスタンスを生成でき、その中からは static のついた変数しか参照できません。通常はこちらを使うことが多いでしょう。

3.4 再帰下降解析器

ではいよいよ本体クラスを見ましょう。まず main では、コマンド引数の 1 番目の文字列を渡して CharToknizer を作り、static 変数 tok に入れます。これで、このクラス内のどのメソッドからも tok が参照できます。そして、最初のメソッドは「プログラム」に対応するものということにしたので、prog() を呼び、その正否 (渡した文字列が文法に合っていたか否か) を打ち出します。ただし、あてはまっても後ろにトークンが残ってはいけないので、さらに tok.isEof() であるという条件も必要です。

```
import java.util.*;
import java.io.*;

public class Sam21 {
  static CharToknizer tok;
  public static void main(String[] args) throws Exception {
    tok = new CharToknizer(args[0]);
    System.out.println(prog() && tok.isEof());
  }
  static boolean prog() {
    if(tok.chk("a")) {
      return ab() && prog();
    } else {
```

になるのでここでは扱っていません。

```
      return true;
    }
  }
  static boolean ab() {
    return tok.chkfwd("a") && tok.chkfwd("b");
  }
}
// ここに CharToknizer を入れる
```

さて、その後の 2 つのメソッドは、BNF に出て来る 2 つの端記号と同じ名前にしてあります。そして、それらが互いに再帰呼び出しをすることで、構文チェックができます。具体的には次のことが必要です。

1. 手続き P が呼び出され、成功した場合 (`true` を返す場合) は、ちょうどその P に対応するだけ入力を読み進む必要がある。`false` を返す場合はこのような制約はない。
2. 最初に呼び出される P については、`true` を返すときにはちょうど EOF まで読み終わっている必要がある。

このようなやりかたで再帰手続きの集まりによって構文検査 (または解析) を行なう方法を、**再帰下降解析** (recursive descent parsing) と呼びます。まあその理論は後の方で。

では、メソッド `prog()` から順に見ていきます。まず、「|」が含まれている規則の場合、そのどの枝 (選択肢) に行くかを決める必要があります。ここで説明している手法では「次の 1 トークンを見て決める」こととします (そのような文法だけを扱えます)。規則はこれですね。

$$prog \ ::= \ \epsilon \mid ab \ prog$$

そして、ϵ は何が来ても (空っぽでも) あてはまり得るのですぐ `true` を返しますが、しかし常にそうしてしまうと ab の枝が選ばれなくなります。なので、まず ab の枝かどうかをチェックし (それには次のトークンが「`a`」か調べます)、それが OK ならそちらを選びます。その枝であれば、`ab()` であり、かつさらに `prog()` であればいいわけです。OK でなければ ϵ の枝なので、すぐ `true` を返します。

メソッド `ab()` に対応する 2 番目の規則はこれです。

$$ab \ ::= \ \texttt{a b}$$

これはつまり、トークンがまず「`a`」、次に「`b`」であることを順に調べればいいです。この枝しかないので、調べたらすぐ `fwd()` で次のトークンに進みます。そのために `checkfwd()` が便利なわけです。どちらかが `false` を返したらこのメソッドもすぐ `false` を返します。

どうでしょうか。文法に対応して機械的に「どの枝か調べ」「その枝なら順番に処理する」という呼び出しをすれば完成すると思いませんか。このパターンを定式化したものを図**??**に示します。

まず、非端記号 S に対して `s()` というメソッドを作ります。そして、S を定義する規則の右辺の選択肢の数だけ if-else の選択肢を作り、どの選択肢に進むかを判定します。通常は、どこへ行くかは入力の現在位置のトークンを見ることで判断できます。ϵ は入力に対応しないので、ϵ の選択肢がある場合はそれは最後の else に対応させることになります (ϵ が無いなら順番に選択していって最後の 1 つを else に対応させてよい)。

それぞれの選択肢の中では、その選択肢にある記号の列に対応する呼び出しをおこない、それらがすべて `true` である場合にのみ `true` を返します。呼び出しは非端記号であればそれに対応するメソッドを呼び、端記号であれば `tok.chkfwd()` を呼べばよいです。

なぜこれでよいのでしょう。それぞれのメソッドは (複数の選択肢があれば適切な選択肢を選んだ上で)「そのメソッドに対応する端記号の列があったことを確認し、その分だけ入力を読み進める」働

```
S ::= ε | T11 x T13 | y T22 | T3

static boolean s() {
    if(T1 chosen) {
        return t11() && tok.chkfwd("x") && t3();
    } else if(T2 chosen) {
        return tok.chkfwd("y") && t22():
    } else if(T3 chosen) {
        return t3();
    } else {
        return true;
    }
}
```

図 3: 再帰下降解析用メソッドの構造

きを持ちます。それは具体的には、端記号については `tok.chkfwd()` を呼ぶことで行え、非端記号についてはそれに対応するメソッドを下請けに呼び出すことで行えます。ですから、最初の `prog()` に対して呼び出した結果が `true` であれば、`prog()` から始まる規則適用の列が正しくあてはまっていることになるわけです。

演習 2-3 上の例題を打ち込み、そのまま動かしなさい。動いたら、次の文法がどのような文字列を表現しているか考え、また再帰下降解析を実現して動かしてみなさい。

a. $prog$ ::= ϵ | ab $prog$
   ab ::= `a` | `bb`

b. $prog$ ::= `a` | `a` $prog$

c. $prog$ ::= ϵ | aa bb $prog$
   aa ::= `a` | `a` aa
   bb ::= `b` | `b` bb

d. $prog$ ::= `1` | `(` $prog$ `)`

e. $prog$ ::= $term$ | $term$ op $prog$
   $term$ ::= `a` | `1` | `(` $prog$ `)`
   op ::= `+` | `-` | `*` | `/`

f. その他自分が試してみたい文法

3.5 演習 2-3 の解答例

まだ慣れていないと思うので、演習 2-3 の各問題の解答例 (再帰下降解析用のメソッドのみ) を示して解説します。まず a. ですが、これは繰り返される要素が「a または bb」ということになります。

```
static boolean prog() {
  if(tok.chk("a") || tok.chk("b")) {
    return ab() && prog();
  } else {
    return true;
  }
```

```
  }
  static boolean ab() {
    if(tok.chk("a")) {
      return tok.chkfwd("a");
    } else {
      return tok.chkfwd("b") && tok.chkfwd("b");
    }
  }
```

prog() は同じでいいと一見思えますが、ab() に進むための条件は「次が a または b」になります。そして ab() の中では | が現れたのでどちらの枝かを選択してそれぞれで処理します。

次に b. ですが、これは「a が 1 個以上並んだもの」です。一見簡単そうですが、この規則は 1 文字目だけ見るとどちらも「a」なので選択できません (つまり先に挙げた規則の条件を満たさない)。しかし、a の先まで行ってから選択することはできます。

```
  static boolean prog() {
    if(tok.chk("a")) {
      tok.fwd();
      if(tok.chk("a")) {
        return prog();
      } else {
        return true;
      }
    } else {
      return false;
    }
  }
}
```

つまり、a の先まで行ってみて、さらにまた a なら prog() を再帰呼び出し、そうでなければそこまでで ture を返します。なお、そもそも a で始まらない場合は false です。このように、「途中まで共通になっている選択肢」があると少しややこしいことになります。

c. も実は aa() と bb() がその構造なのでそこがやっかいですが、prog() は簡単です。

```
  static boolean prog() {
    if(tok.chk("a")) {
      return aa() && bb() && prog();
    } else {
      return true;
    }
  }
  static boolean aa() {
    if(tok.chk("a")) {
      tok.fwd();
      if(tok.chk("a")) {
        return aa();
      } else {
        return true;
```

```
      }
    } else {
      return false;
    }
  }

  static boolean bb() {
    if(tok.chk("b")) {
      tok.fwd();
      if(tok.chk("b")) {
        return bb();
      } else {
        return true;
      }
    } else {
      return false;
    }
  }
}
```

次に d. はどうでしょうか。これは選択肢を選ぶのが簡単なので、作るのも簡単です。

```
  static boolean prog() {
    if(tok.chk("1")) {
      tok.fwd(); return true;
    } else if(tok.chk("(")) {
      return tok.chkfwd("(") && prog() && tok.chkfwd(")");
    } else {
      return false;
    }
  }
```

これは何ができるかというと、「1」の周囲をかっこで囲んだもので、かっこは何重でも大丈夫です。このような構造は BNF が得意とするところです。

最後の e. はかなり複雑です。実はこれは (不完全ですが)、算術式をイメージしています。*prog* が「式」に相当します。

```
  static boolean prog() {
    if(term()) {
      if(tok.chk("+")||tok.chk("-")||tok.chk("*")||tok.chk("/")) {
        return op() && prog();
      } else {
        return true;
      }
    } else {
      return false;
    }
  }
```

```
  static boolean term() {
    if(tok.chk("a")) {
      tok.fwd(); return true;
    } else if(tok.chk("1")) {
      tok.fwd(); return true;
    } else {
      return tok.chkfwd("(") && prog() && tok.chkfwd(")");
    }
  }
  static boolean op() {
    return tok.chkfwd("+") || tok.chkfwd("-") ||
           tok.chkfwd("*") || tok.chkfwd("/");
  }
}
```

prog() だけは先頭が共通の選択肢がありますが、あとは普通に区分できます。書き方はもうちょっと短くできるところが多いですが、多少長くなってもなるべく同じパターンで書くようにしています。

演習 2-4 先の演習の e. は式に対応しているが、演算が四則演算しかなかった。次のように強化してみよ。

a. C や Java では代入も演算の 1 つという位置付けである。代入も扱えるようにしてみよ。
b. if 文などで使うには比較演算子が不可欠である。比較演算子も使えるようにしてみよ。
c. 関数呼び出しも重要な式の要素である。関数呼び出しも使えるようにしてみよ。
d. その他、自分が追加したいと思う機能を追加してみよ。

演習 2-5 次のような文法の再帰下降解析器を構成してみよ。なお、*Expr* は上の演習の *prog* に対応しているものとする (名前だけ書き換えて使えばよい)。

a. $prog$ ::= { $statlist$ }
   $statlist$ ::= ϵ | $stat$ $statlist$
   $stat$ ::= $Expr$; | { $statlist$ }
b. 上記の *stat* の選択肢に次を追加せよ。
   $stat$ ::= `w` ($Expr$) $stat$ | `d` $stat$ `w` ($Expr$) ;
c. 上記の *stat* の選択肢に次を追加せよ。
   $stat$ ::= `i` ($Expr$) $stat$
   | `i` ($Expr$) $stat$ `e` $stat$
d. 上記の文法に好きなものを追加せよ。

4 課題 2A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 2」、学籍番号、氏名、提出日付。

- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. BNF によって言語の文法を規定するのはどう思いましたか。

  Q2. 再帰下降解析で構文へのあてはめができるということに納得しましたか。どこが難しかったですか。

  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。