システムソフトウェア特論'17 # 5 字句解析

久野　靖 *

2017.7.16

1　字句解析とは

1.1　字句解析の位置付け

既に学んだように、**字句解析** (lexical analysis) は言語処理系がソースコードを読み込む最初の部分であり、その主な仕事はソースコードを「名前」「定数」「記号」などのかたまり、ないし**トークン** (token) に分割することです。そして、言語処理系の中で字句解析を行なう部分を**字句解析器** (lexcal analyzer)、lexer、tokenizer などと呼びます。

なぜ字句解析器が必要なのでしょう。それは、これまでに見て来たように、BNF などによる文法記述は、変数、定数、記号などに単位を端記号 (ソースコードに出て来る単位) とするのが通例だからです。

ソースコードは文字の並びなわけですから、変数や定数なども非端記号として扱い、「文字」のレベルまで細かく定義することもできます。そのようにすれば、構文解析のときに文字を 1 文字ずつ読めばよいだけなので、字句解析器は不要です。しかし、そうすると次のような問題が現れて来ます。

- 文法が細かくなり読みづらくなる
- 1 文字単位で構文解析の処理を行なうと処理が遅くなる

CPU 性能が高くなった現在では、2 番目の問題はだいぶ軽減されていて、そのため構文と字句をまとめて扱うような処理系も使われるようになってはいます。しかし文法が細かくなることには変わりがないので、ここではまず、字句解析を分けて扱う伝統的な構成を取り上げています。

1.2　字句解析器の仕事

しかしそもそも、字句解析器ってそんなに大げさな名前がつくほどの仕事があるのでしょうか?

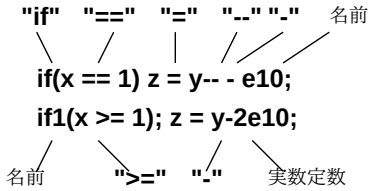

図 1: 字句解析の面倒さ

それが、あるのです。図 1 のように、ごく普通のプログラムでも、よく見ないと何の字句か決められないことが沢山あります。「`if(x == 1) z = y-- - e10;`」では、最初の `if` は if 文をあらわす特別な名前です。==は「等しい」比較演算子ですが、=1 つだけだと代入ですから、=がいくつあるかで区分する必要があります。--(減少演算子) と-(引き算) もそうです。そして、「`if1(x >= 1); z -`

*電気通信大学 情報理工学研究科

y-2e10;」と対比すると、if1 というのは普通の名前なので、こちらは x が 1 以上かどうかの結果をパラメタとして関数を呼び出しています。また、2e10 は実数定数 (2×10^{10}) ですが、e10 はただの名前 (変数) です。字句解析器はこれらの区分を正確につけながら、効率よく入力を読み取って行く必要があります。

1.3 代表的な字句

ここでは、字句解析器が扱う代表的な字句と、その扱いに際して問題になることを整理して示します。言語によってはそれらの一部が無かったり、扱いが大幅に違うこともありますが、そのような違いには深入りせず、あくまでも「代表的な」プログラミング言語では、ということで進めます。

- **識別子** (identifier) — 識別子とは要するに「名前」のことですが、プログラミング言語の世界では「名前 (name)」という用語を別の意味 (たとえば変数の実体などの意味) で用いることがあるので、それとの混乱を避ける意味で「識別子」を使います。

  識別子とは要するに、「どの変数や手続きであるか」を区別して指定するための文字列です。そして多くの言語では、識別子を「英字で始まり、英数字が並んだ長さ 1 以上の列」としています。ただし、何が英字なのかについては言語で異ります。たとえば C 言語では、英語のアルファベット 26 文字 (A-Za-z) に加えて、下線 (_) も英字としています。識別子に日本語文字 (ひらがな、漢字等) を許す言語もあり、その場合は規則はややこしくなります。[1]

- **予約語** (reserved words) — 予約語とは、識別子と同じ規則で認識できる語だけれど、特別な意味を持っていて通常の識別子としては使えないようなものを指します。まさに「予約」されているわけです。代表的なのは、C 言語における if、while、do、return など制御構造を表すのに使う語です。

  以前は、予約語を設けず、if や return などという名前の変数を使える言語も使われていましたが、言語処理系の作り方が面倒になるので、今では少なくなっています。予約語がある場合は、言語の仕様書にそのことが明記されます。

  字句解析器で予約語を認識するときには、直接認識する方法もありますし、まず識別子を認識して、それから表を検索して予約語だったら予約語として扱う、という方法もあります。

  このほか、一部の識別子を予約語ではないが予め意味の決まっている特別な識別子 — **疑似識別子** (pseudo identifier) として扱う言語もあります。

- **リテラル** (literal) — リテラルというのは「文字通り」という意味ですが、プログラミング言語では数値や文字などの定数を表すのに使います。たとえばソースコードに「123」とあれば、それはまさに 123 という整数の値を表すわけです。

  - 数値リテラル — 数値の定数は、整数と実数の 2 種類に分かれます。いずれもプラスやマイナスの符号がつくことがあります (プラス符号はない言語もある)。また、C 言語のように整数では通常の十進のほかに 8 進や 16 進のリテラルを許す言語もあります。
  - 文字 (列) リテラル — 文字や文字列の定数は、通常シングルクォート'...'やダブルクォート"..."で内容を囲んで表します。文字列しか扱わない言語では、どちらで囲んでもよいものが多いです。C や Java では、シングルクォートは文字リテラル、ダブルクォートは文字列リテラルとして区別しています。
  - 論理値リテラル — true、false は論理値のリテラルですが、予約語としている言語が多いです。

[1] 以下では日本語文字などは考えないことにします。

- **演算子** operator — 演算子はもともとは四則演算を表す`+`、`-`、`*`、`/`など 1 文字のものから始まりましたが、Algol で代入を表すのに`:=`を使うなど、2 文字のものも早くから含まれていました (Algol では`=`は「等しい」に使います)。C 以降は逆に`=`が代入、`==`が等しいですが、さらに`===`を持つ言語もあります。
- **区切り記号** (delimiter symgol) — 「`(`」、「`)`」などのかっこ類や「`,`」などは演算子ではありませんが (C 言語ではカンマは演算子にも使う)、言語を記述する上で必要な記号です。これは通常、区切り記号と呼びます。区切り記号にも 2 文字以上のものがある言語もあります。
- **空白** (blank space) — 多くの言語では識別子どうしがくっついていてはいけないので、識別子が連続する箇所には空白を入れます。ここで空白といっているのは、タブ文字や改行文字であってもよい場合が多いです。
- **注釈** (comment) — コメントは人間が読むだけでプログラムとしては扱わないので、空白と同等に扱う場合が多いです。コメントの規則も言語によりさまざまなので、扱いに工夫が必要な場合もあります。C 言語の「`/* ... */`」などもなかなか面倒です。

2 オートマトンに基づく字句解析

2.1 オートマトンに基づく字句解析のあらまし

既に学んだように、トークンの定義は正規表現で表現するのに適していて、さらに正規表現は有限オートマトンに変換でき、有限オートマトンはプログラムで効率よく実装できるのでした。本節ではこの話題をもうすこし詳しく取り上げて行きます。

前に触れたときは、正規表現から正規文法を経て有限オートマトンに変換していましたが、実はもっと直接的に正規表現をオートマトンに変換できます。まずその方法について説明します。

ただし、正規表現から素直に変換した有限オートマトンは**非決定性有限オーマトン** (non-deterministic finite automata, NFA) となります。非決定性というのは、ある 1 つの入力に対する状態遷移が 1 つではなく複数ある場合もある、ということです。

そしてプログラムで効率実装するには、1 つの入力に対して進むべき状態が 1 つでないと困ります。しかし幸いなことに、NFA を決定性有限オートマトン (DFA) に変換することができます。その方法についても説明しましょう。

そして、これらの変換を手で行なうのは非常に大変なので、実際には**字句解析器生成系** (lexcal analyzer generator) を使って正規表現のならびからそれらを認識する字句解析器を生成します。それがどのようなもので、どんな風に使うかも、経験しておきましょう。

2.2 正規表現から NFA への変換

ではまず、正規表現を NFA に変換する方法から見てみましょう。そのために例として、整数・実数の数値定数を定義する正規表現を図 2 に示します。これからこれを、NFA に変換してみます。

まず、IConst と RConst をそれぞれ「素直に」オートマトンに変換したものを図 3 に示します。

これらのオートマトンによってある文字列が実定数であるかどうかを判定できます。しかし実際に必要なのは、入力の文字の並びを順に見ていって「どの種のつづりがあったか」を知ることですね。

それには、図 4 のように「本当の」初期状態を付加し、そこから各トークンを認識するオートマトンの初期状態に向かう空遷移 (空の入力列に対応する遷移) を加えます (併せて、各最終状態にそれはどのつづりに対応しているかの情報をつけ加えます)。

しかし、図 3.3 のようなオートマトンはそのままプログラムに変換して字句解析器とすることはできません。なぜなら、初期状態から出ている多数の空遷移のうちどれを選んでいいか分からないからです (例えば識別子などは次の 1 文字が英字かどうかでそちらへ行くかどうか決められますが、整定数と実定数の場合はずっと先まで見ないと区別できません)。

```
IConst = [ Sign ] Digit+
RConst = [ Sgin ] Digit+
           ( Expt [ Sign ] Digit+ | Dot Digit+ [ Expt [ Sign ] Digit+ ])
Sign   = "+" | "-"
Expt   = "E" | "e"
Digit  = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Dot    = "."
```

図 2: 数値定数を定義する正規表現

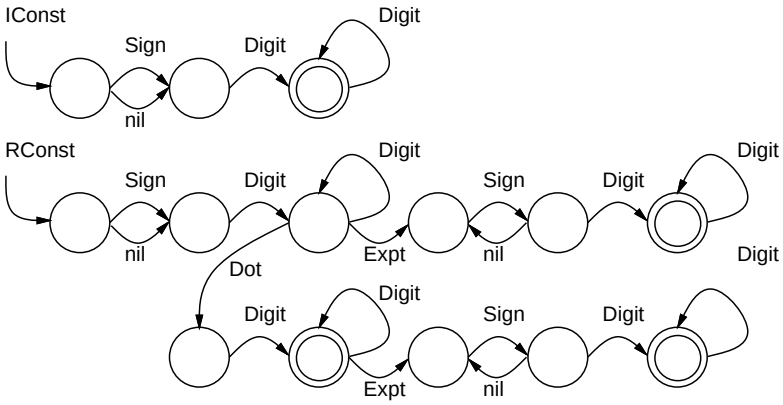

図 3: 図 3.1 に対応する有限オートマトン

このように次の状態が一意に決まらない有限オートマトンが NFA です。図 3 のオートマトンも実は NFA です (空遷移を進むかどうかは常に非決定的なので)。

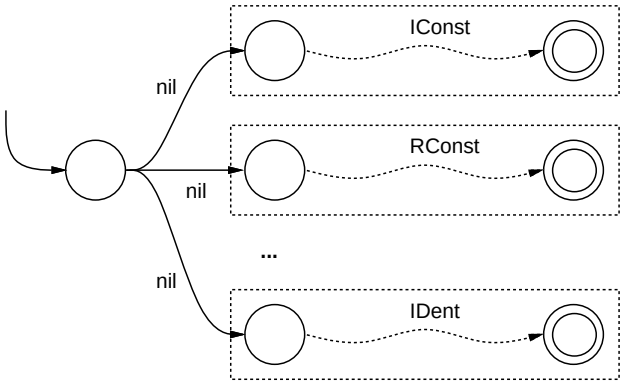

図 4: トークン認識のための有限オートマトン

これに対し、次の状態が一意に決まっているオートマトンを決定性有限オートマトンとよぶ。前章で述べた、容易にプログラムに変換できる有限オートマトンとは実は DFA のことである。しかし悲観するには及ばなくて、NFA を DFA に変換する方法が知られている。その前に、まず正規表現を NFA に機械的に変換する方法から見てみましょう。

図 3 の非決定性有限オートマトン (NFA) は図 2 の正規表現から手で構成しましたが、空遷移を多数作ってもかまわなければ、機械的にやるのは簡単です。そのやり方を図 5 に示します。

まず初期状態と最終状態を用意します。次に、文字に対応する正規表現の場合、初期状態から最終状態へのその文字による遷移をつくればすみます。

これ以外の場合は全て、既存の NFA を空遷移でつなげて行きます。例えば列 XYZ に対応する NFA

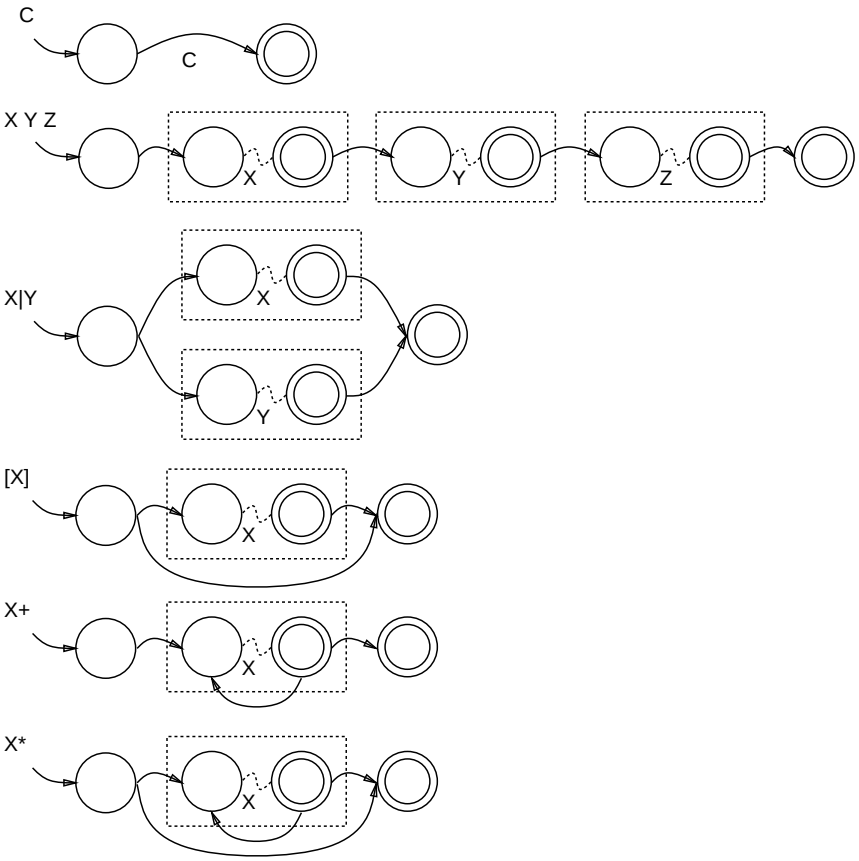

図 5: 正規表現を機械的に NFA に変換する方法

では、正規表現 X、Y、Z に対応する NFA をつくり、それぞれ前のものの最終状態から次のものの初期状態への空遷移をつなげます。全体の初期状態からは X の初期状態への空遷移、Z の最終状態からは全体の最終状態への空遷移を作ります。他のものも同様です。なお一般の NFA では最終状態は複数個有り得ますが、この方法で作っている限り、最終状態も初期状態同様 1 個だけです。

2.3 NFA から DFA への変換

次に NFA を DFA に変換するわけですが、その基本的なアイデアは次のようなものです。NFA では、ある状態である文字が来たときに進む「次の状態」が一般に複数個存在します。そこで、「NFA の状態の集合」をそれぞれ新たに 1 つの状態であると考えて有限オートマトンを構成します。すると、ある「もとの NFA の状態の集合」において、ある文字が来たときに進むことができる「もとの NFA の状態の集合」は (集合として)1 つですから、これは結果として DFA になります。

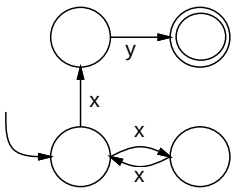

図 6: 簡単な NFA の例

例えば、図 6 のオートマトンは x が奇数個続いて最後に y がある語のみを受理しますが、最初の x が来たとき行く先が 2 つあるので NFA です。次に図 7 を見ると、この有限オートマトンの各状態は先の NFA の状態の集合 (斜線で塗られたもの) から成っています。初期状態は左の箱、つまり元の

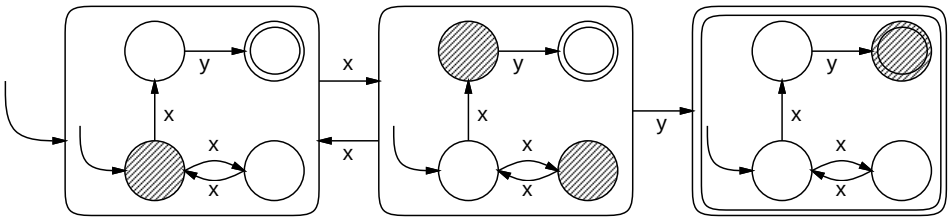

図 7: NFA の状態集合を状態とする DFA

NFA の初期状態だけが塗られたものです。

ここで x が来ると、元の NFA で行ける両方の状態が塗られた状態、つまり中央の箱へ来きます。ここで y が来ると、元の NFA で行けるのは右上の状態だけですから、それだけが塗られた右側の箱に来ます。一方 x が来た場合は、元の NFA で行けるのは初期状態ですから、左側の箱へ戻ります。

これで、無事全ての入力について行き先が一意に定まった有限オートマトン、つまり DFA が構成できました。最終状態は元の NFA での最終状態を含んだもの、つまり右側の箱だけとなります。

なお、この方法で生成した DFA は冗長な状態を持つことがあります。たとえば、図 8 は $(b+|c)aa$ を受理する DFA の例ですが、左も右も同じように動作するものの、左の方が状態が多くなっています。a がいちど現れた先は同じなので、そこは共通にすれば状態が減らせるわけで、右のものはそのようになっています。これを状態の最小化と言います。

状態の最小化を行なうアルゴリズムの概要だけ説明しておきます。DFA の中で互いに区別すべき状態をグループ化し (最初は異なる記号に対応する最終状態群とそれ以外のすべての状態群のグループができる)、次に繰り返し、異なるグループに進むような状態を分割していき、これ以上分割できなくなったところで各グループを 1 つの状態とすることで、最小の状態を持つ DFA を得ることができます。

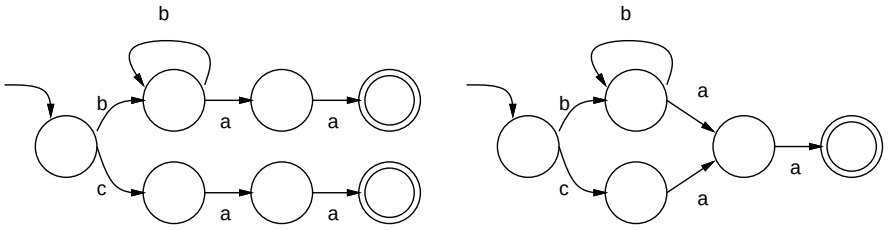

図 8: 冗長な状態を持つ DFA の例

3 字句解析器生成系

ここまでに述べた有限オートマトンの原理を利用して、実用的な字句解析器を自動生成するツールが多数作られています。Unix に含まれている字句解析器生成ツール **Lex**、それと同一仕様でつくられフリーソフトとして配布されている **Flex** などが代標的です。

これらはいずれも、正規表現から有限オートマトンを生成し、その構造を表の形で出力し、これと入力を走査しながらこの表をたどる (解釈実行する) ドライバルーチンを組み合せることで字句解析器を構成します。これらのツールで生成した字句解析器では、各正規表現ごとに、それを認識した際実行するプログラムの断片が指定できます。

Lex/Flex は C 言語のソースコードを生成しますが、この科目では Java を使っているので、同じ原理で Java による字句解析器を生成するフリーソフトである JFlex を取り上げ、どのようなものか体験してみます。

次に、ごく簡単な JFlex のソースファイルを示します。

```
%%
%class Lexer
%int
L = [A-Za-z_]
D = [0-9]
Ident = {L}({L}|{D})*
Iconst = [-+]?{D}+
String = \"(\\\"|[^\"])*\"
Blank = [ \t\n]+
%%
{Blank}   { /* ignore */ }
while     { return 4; }
{Ident}   { return 1; }
{Iconst}  { return 2; }
{String}  { return 3; }
```

JFlex では (Lex もそうですが)、「%%」でソースファイルをいくつかのセクションに区分し、また行の先頭にある「%名前」でさまざまな設定を行ないます。

最初のセクションは生成される Java ソースにそのまま取り込まれるので、package 文や import 文を書くのに使いますが、今回はとくに何も必要ないのでからっぽです。

2 番目のセクションの冒頭で、生成されるクラスの名前を Lexer とし、またトークンを取り出すメソッド yylex() の返値の型を int に指定しています。その後は、本体定義を書くのに利用するマクロ (定義) 群で、まず英字 L と数字 D を定義し、それを用いて「識別子 (英字のあとに 0 個以上の英数字)」「整数定数 (符号があってもなくてもよく、その後に数字の並び)」を定義しています。その次は文字列リテラルで、「"があり、\"または"以外の文字が 0 個以上あり、「"がある」と読みます。最後は空白で、「空白文字、タブ文字、改行文字の並び」です。

3 番目のセクションが本体部分で、上での定義を利用しつつ (しなくてもよい)、次の形で動作を指定します。

正規表現 { Java コード }

ここで「Java コード」はトークンを返す関数 yylex() の中で特定のトークン (正規表現にあてはまる入力) が認識されたときに実行されるので、「return 1;」などと書くと yylex() から 1 が返されます。逆に return を書かないと引続き次のトークンの認識に進むので、「無視する」動作になります。

そういうわけで、ここでは空白は無視し、識別子、整数定数、文字列はそれぞれ 1、2、3 を返すようにしています。また、予約語の例も入れたかったので、while が来たら 4 を返します。このように、字句解析器では「どの種類のトークンか」という区分を返せばよいわけです。

これを処理してクラスファイルを作るには、まず jflex を動かし、エラーがなければ生成されている Lexer.java をコンパイルします。

```
% jflex sam51.jflex
Reading "sam51.jflex"
Constructing NFA : 42 states in NFA
Converting NFA to DFA :
.............
17 states before minimization, 14 states in minimized DFA
Old file "Lexer.java" saved as "Lexer.java~"
Writing code to "Lexer.java"
```

```
% javac Lexer.java
%
```

では次に、これを呼び出す Java プログラムを見てみます。

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

public class Sam51 {
  public static void main(String[] args) throws Exception {
    Lexer lex = new Lexer(new InputStreamReader(System.in));
    while(true) {
      int tok = lex.yylex();
      if(tok == Lexer.YYEOF) { break; }
      System.out.printf("%d %s\n", tok, lex.yytext());
    }
  }
}
```

クラス Lexer のインスタンスを生成するときに、Reader オブジェクトを渡す必要があります。Reader は java.io パッケージに定義されているインタフェースで、文字単位での入力機能を持つオブジェクトを定めています。具体的な Reader の種類として、ここでは System.in(InputStream オブジェクト) を元にした Reader を作り出すため InputStreamReader を使いました。

その先はすぐ無限ループで、メソッド yylex() を呼んでトークンを 1 つずつ取り出して行きます。そのトークンが Lexer で定義している定数 YYEOF と等しいならファイルの終わりです。そうでないなら、トークン番号とそのトークンに対応する文字列 (yytext() を呼ぶと返される) を表示します。

では実際に動かしてみましょう。

```
% javac Sam51.java
% java Sam51
abc 123 while awhile 123while
1 abc    ←識別子
2 123    ←整数
4 while  ←予約語 while
1 awhile ←これは識別子
2 123    ←整数と while がくっついていても
4 while  ←トークンとしては分離される
"abc" "a\"bc" """"
3 "abc"   ←文字列
3 "a\"bc" ←「"」を含む文字列
3 ""      ←空文字列
3 ""
"aa       ←改行を含む文字列も OK
bc"
3 "aa
bc"
^D%       ← Ctrl-D を打つとファイルの終わりになる
```

演習 5-1 JFlex の例題を自分でも打ち込んで動かしてみよ。動いたら、次のような機能を追加してみよ。

a. 実数の数値定数を追加する。
b. さまざまな演算記号類やかっこ、カンマなどプログラミング言語の定義で必要なものを字句として追加する。
c. コメントを無視する機能。コメントの形式としては好きなものを使ってよい。
d. とくに何もしなければ、現在ソースコードの何行目にいるかは分からない。JFlex に組み込みの行番号機能もあるが、改行文字を独立したトークンとして認識することで何行目にいるかを併せて表示するようにしてみよ。
e. そのほか、JFlex のマニュアルなどを見て興味深いと思った機能があれば使ってみよ。

4 ハンドコーディングによる字句解析

実は字句解析というのはそれほど「難しい」作業ということはないので、字句解析部を全て手で書く、というのも十分実用的な方法です。実際、Lex などのツールによって生成された字句解析器は有限オートマトンを表現するデータ構造を入力に従ってたどりながら動作する、いわばインタプリタの形になります。一方、手で字句解析器を書いた場合には必然的に、現在プログラム上のどこを走っているかが状態に対応するので、実行速度の面ではこの方が有利です。

また、全てをプログラムとして書くわけなので、字句の認識と並行して様々な処理を行わせることができます。例えば数字を読み進めるのと並行して整定数の値を計算したり、名前の各文字を読み進めながらそれを文字列領域にコピーしたりできます。生成ツールに頼った場合にはこれらの文字はツールが定めた場所に蓄積され、つづりが認識された時点で改めて値を計算したりコピーを行うことになるので、結局頭から 2 回文字列を処理することになります。ソースコード中の名前や数値の数は非常に多いので、この差は結構無視できません。

この方法をとる場合には有限オートマトンがあまり複雑でないことが必要なので、Lex の場合のように予約語の認識を有限オートマトンによって行わせるのは困難です。そこで前述のように、予約語はいったん識別子として認識され、その後で簡単な表を引いて予約語かどうかを調べる方法が一般に使われます。

このように手で字句解析器を書く場合でも、どのようなものをトークンとして認識すべきかを正確に規定することはどのみち欠かすことができません。したがって、字句を正規文法や正規表現などで定義する、という道具立ては、このような場合にも十分役に立つのです。

では JFLex による字句解析器と同様に使える字句解析器を作ってみましょう。ここでは簡単のため、文字列は省略し、また符号つき整数の認識が単純化してあります。

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

public class Sam52 {
  public static void main(String[] args) throws Exception {
    Lexer lex = new Lexer(new InputStreamReader(System.in));
    while(true) {
      int tok = lex.yylex();
      if(tok == Lexer.YYEOF) { break; }
      System.out.printf("%d %s\n", tok, lex.yytext());
    }
  }
```

```
}
class Lexer {
  public static final int YYEOF = -1;
  Reader rd;
  int nc;
  String text = "";
  HashMap<String,Integer> map = new HashMap<String,Integer>();
  public Lexer(Reader r) throws Exception {
    rd = r; nc = rd.read(); map.put("while", 4);
  }

  public int yylex() {
    try {
      while(nc == ' ' || nc == '\t' || nc == '\n') { nc = rd.read(); }
      text = "";
      if(nc == YYEOF) {
        // do nothing
      } else if(alpha(nc)) {
        text += (char)nc; nc = rd.read();
        while(alpha(nc) || digit(nc)) { text += (char)nc; nc = rd.read(); }
        if(map.containsKey(text)) { return map.get(text); }
        return 1;
      } else if(sign(nc) || digit(nc)) {
        text += (char)nc; nc = rd.read();
        while(digit(nc)) { text += (char)nc; nc = rd.read(); }
        return 2;
      } else {
        System.err.printf("%x: invalid char\n", nc);
        nc = rd.read(); return yylex();
      }
    } catch(IOException ex) { }
    nc = YYEOF; return YYEOF;
  }

  public String yytext() { return text; }
  private boolean alpha(int c) {
    return c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z';
  }
  private boolean digit(int c) { return c >= '0' && c <= '9'; }
  private boolean sign(int c) { return c == '+' || c == '-'; }
}
```

メインの方は先と変わっていません。Lexer が作成したクラスです。コンストラクタでで Reader を受け取り、変数の初期設定をします。変数 map には、予約語のスペルと対応するトークン番号を登録しますが、ここでは while だけを登録しています。変数 nc には常に「次の 1 文字」が入っているようにしたので、コンストラクタ中でその 1 文字を読んでいます。

ほとんどの処理はメソッド `yylex()` の中で行なわれます。全体の構造としてまず、入力時の例外を補足するため try…catch で全体を囲み、例外が出た場合は最後に来て EOF を返します。最初に不要な空白類をスキップし、そのあと次の文字で分岐します。

- EOF なら下に抜けて末尾の処理に合流します。
- 英字なら名前を認識しますが、認識し終わったところで `map` を検索し、入っていれば格納されている番号を返し、そうでない場合は 1 を返します。
- 符号と数字なら数字を読み取って 2 を返します。

演習 5-2 上の例題をそのまま動かしなさい。動いたら、次のような改良を施してみなさい。

a. トークンとして「`(`」「`)`」「`=`」を追加する。番号は適当に定めてよい。
b. トークンとして「`>=`」「`<=`」「`!=`」「`==`」を追加する。もちろん「`=`」も使えること。
c. 例題では「`+`」「`-`」だけでも数値定数になってしまう。それはよくないので、「`+`」「`-`」は単独の演算子、後ろに数字がくっついていれば数値の符号というふうにしなさい。
d. 文字列が取れるようにしなさい。
e. そのほか、やってみたいと思う機能を追加しなさい。

5 課題 5A

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# 5」、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. 字句や字句解析についてどのように感じましたか。
Q2. オートマトンに基づく字句解析生成系・ハンドコーディングによる字句解析機についてどう思いましたか。
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。