情報処理演習 II

コンパイラの作成

第 1 回 lex を用いた字句解析器の作成

2005 年度担当: 中井央

1 はじめに

この演習では、プログラミング言語演習 I や II で習った内容を理解していることを前提にしている。

この演習ではプログラミング言語処理系を作る基礎を学ぶ。プログラミング言語処理系とは、一般には**コンパイラ (compiler)** と呼ばれているものである。コンパイラを構築する方法についてはかなり昔から論じられてきており、ソースプログラムを解析する技術はかなり確立されている。このため、そのプログラミング言語の仕様 (決め事) を与えることで、このような解析部分を自動的に生成することも可能となっている。

本演習は、すでに幅広く利用されている生成系を利用し、小さなプログラミング言語のコンパイラを作成しようというものである。

2 この演習の目的

コンパイラを使ったことはあっても作ってみようと思った人は少ないと思う。例えば、算術式を一行入力するとその答えを計算して求めてくれるプログラムを作ることを考えてみよう。ここでは扱えるのは四則演算だけにしておこう。`1+2` と入力したら、`1` と`+`と `2` に分解して、`1` や `2` は数値に直し、足し算をする。`1+2` くらいなら処理できそうに思われる。これが `13263+976` でもなんとか処理できるだろう。でも、四則演算を含む式はこのような短いものだけでなく、`1+2*3-4/5*6+7-8+9` のようなものもありえる。このとき、問題になるのは、`1+2*3` とあったら、先に `2*3` 計算して、`1` にその結果を足さないといけないことである。

仮にこれらをクリアしたとしよう。では、ちょっとグレードアップして今度は四則演算に括弧を使えるようにしたくなったとしよう。今度はもうちょっと難しくなる。括弧で括った中側にある式は先に計算をしないといけないからである。

さて、さらにこれらをクリアしたとしよう。今度は一旦計算した結果を覚えておく機能もつけたいとする。つまり、`a = 1+2*3` とやるとその結果を `a` に覚えておくのである。

このようなプログラムを作るにはどうすればよいか。これがこの演習での課題である。ここで学ぶことを発展させていくと、プログラミング言語のコンパイラを作成できるようになる。

3 コンパイラの概要

コンパイラを構成する部分はいくつもある。一般にはコンパイラで行われる処理には以下のものがある。

1. 字句解析
2. 構文解析
3. 意味解析
4. 最適化
5. コード生成

これらの言葉の中に明示的には表れていないが、このほかの重要な部分として記号表の管理がある。以下、これらについて説明する。

3.1 字句解析

字句解析 (lexical analisys) とは、ファイルに記述されているプログラムを意味のある最小単位に分解することである。わかりやすい言葉でいえば、単語に分けること、である。ここでいう「単語」のことを**トークン**(token) と呼ぶ。正確にはトークンは単語の種類のことをさす。自然言語の言葉でいえば、「品詞名」のことである。あるトークンを構成する実際の文字の並び

(文字列) のことはレクシム (lexeme) という。

たとえば、「Nakai is a human.」を品詞に分解すると、名詞、動詞、冠詞、名詞、となる。後にも述べるが、これらの品詞の並び方から文法に照らし合わせてどのような**構文**なのかを判断する。そうすると “Nakai” は名詞というトークンであり、N, a, k, a, i という文字の列であるレクシムを持つことになる。

3.2 構文解析

自然言語には文法がある。同じく、人造言語であるプログラミング言語にも文法がある。たとえば、`a = 3 - 4;` は許されても`= a - 4; 3` は意味が通らないため、コンパイラがエラーを通告する。

「文」を規定するものが「文法」(grammar) である。

分解されたトークンの並びが文法に則っているかどうか確かめるのが「構文解析」(syntax analisys または parsing) である。

文法といっても実は色々と種類がある。

一般にプログラミング言語を規定する文法としては、字句を規定するのに正規文法 (regular grammar) あるいは正規表現 (regular expression) を使い、構文を規定するのには文脈自由文法 (context free grammar) を用いる。これらについてはそれぞれの解析器についての節で述べる。

3.3 意味解析

プログラミング言語の意味とは、そのプログラミング言語でプログラムを書く際の決め事のことである。その決め事を守っているかどうかを調べることを意味解析という。

たとえば、C 言語などでは変数は宣言してからでないと使用することはできない。このようなことを調べるのは、これまでに述べた字句解析や構文解析では行えない。意味解析でチェックする主な項目を挙げると次のようになる。

1. 宣言と名前の対応付け
2. 名前の有効範囲 (スコープ) の決定
3. 型の情報の収集と検査

宣言と名前の対応付けでは、使用部分で出てきた名前はちゃんと宣言されているか、また、宣言部分では二重宣言になっていないかを調べる。名前の有効範囲は宣言と名前の対応付けと関連する。C 言語では`{`と`}`で括った部分はその中だけの変数を宣言できる。それより外側で宣言された変数ももちろん使えるが、もし同じ名前の変数が内側にあればそちらが優先される。

意味解析ではそのようなことを管理する。

そして一般には中間コードと呼ばれるコードを生成する。これは目的コード (実行可能形式のコード) へ変換する前段階の、解析した情報に基づいたある 1 つの形式である。簡単なコンパイラの場合はここで目的コードまでを出力する。

3.4 最適化およびコード生成

簡単なコンパイラの場合、最適化は省略される。この演習では最適化を扱わないため、説明は省略する。

本演習で扱うコンパイラは仮想機械のコードを出力する。仮想機械はソフトウェアで構成した仮想的な計算機のことである。

4 字句解析器

演習の最初は字句解析器である。字句解析器とは字句解析を行うプログラムのことである。

まずは字句を表現するところから始める。その後、そのような表現を受け取って、字句解析を行うプログラムを自動生成するプログラム (生成系) を使った字句解析器の生成法を示す。

4.1 正規表現

前節でも述べたように字句を表現するには正規表現を用いる。ここで正規表現の定義を掲げておこう。ここでは**再帰的な**定義を挙げる。以降、文法を考える場合、再帰的な考え方は重要となる。このような考え方に慣れるようになってほしい。

正規表現の定義の前に言葉の定義しておく。

アルファベット (alphabet) とは言語を構成する文字の集合のことを言う。たとえば、我々が日常使っている算用数字の場合、アルファベットは`０，１，２，３，４，５，６，７，８，９`である。同様に英語のアルファ

ベットは a，...，z，A，...，Z である。日本語になると表現しきれないので省略する。

空列とは 0 個のアルファベットの要素の並びである。空列を ε で表す。この**「0 個の」**という表現も重要であるので覚えておいてほしい。

正規表現の定義

1. アルファベットの 1 つの要素は正規表現である。
2. ε は正規表現である。
3. r_1 と r_2 がどちらも正規表現であるとき、r_1r_2 (それらを順に並べたもの) は正規表現である。(連接。2 つの文字列をつないだものを表現したい場合)
4. r_1 と r_2 がどちらも正規表現であるとき、$r_1|r_2$ は正規表現である。(選択。どちらかの表現に一致させたい場合)
5. 正規表現 r に対して (r) は正規表現である。
6. 正規表現 r に対して $r*$ は正規表現である。ここで $*$ は r の 0 回以上の繰り返しを表す。

これだけではちょっとわかりにくいかもしれないからいくつか具体例を挙げる。

簡単なところで 2 進数を考えてみよう。2 進数の表現に現れるのは 0 か 1 の文字である。つまり 2 進数のアルファベットは $\{0, 1\}$ ということになる。

ここで定義の一番目から 0 または 1 は正規表現である。

さて、2 進数 01 というのを表す正規表現を考えよう。0 と 1 がそれぞれ正規表現であるから、01 は三番目の規則によって正規表現である。

しかし、ここまでででは特定の 2 進数を表現できただけで、あまり面白くないし、正規表現のありがたさも感じない。本当は 2 進数とはこういうものです、とびしっと言えるような表現がほしいところである。

まず、2 進数一桁を考えてみよう。2 進数 1 桁は 0 または 1 である。したがって 0|1 がその正規表現である。

実際には 2 進数は別に何桁あってもいい。2 進数 2 桁は (0|1)(0|1) となる。括弧で括っているのは優先順位をつけるためである。

これではまだ一般的に表したことにはならない。一般的には 2 進数とは、(0|1) が 1 個以上並んでいる。したがって、(0|1)(0|1)* と言う表現によって 2 進数を表したことになる。なお、ここで、便利な表記法として + を導入しよう。(0|1)(0|1)* の代わりに (0|1)+ と書けるとする。ちなみに 0|1+ とは異なる点に注意。

2 進数と同様にして 10 進 (正整数) 数を表現してみよう。ただし、ここでは負の数は考慮しない。

(0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)*

または

(0|1|2|3|4|5|6|7|8|9)+

となる。

ここで (0|1|2|3|4|5|6|7|8|9) という表記は書くのが面倒くさい。これを略記する方法として `[0-9]` を導入する。数字の場合と同様に英文字のアルファベットも `[a-z]` のように記述できるとする。

また、同様の表記法として、任意の文字の集合を `[` と `]` を用いて表現できるとする。つまり、`a` と `h` と `o` の**どれか**を表現するには `[aho]` と書けばよいとする。

以上の略記を用いると、C 言語の変数として利用できる文字列の正規表現は次のようになる。

```
[_a-zA-Z][_a-zA-Z0-9]*
```

つまり、アンダースコア (_) か英文字のアルファベットの小文字大文字が先頭に来てもよい。それに続くことのできる文字はアンダースコアか英文字のアルファベットの小文字大文字に 0 から 9 までの数字である。

つまり、`_abc` や `a111` や `My_first_love` などは C 言語での名前 (識別子) である。

演習 1

以下のものを正規表現で表せ。なお、以下では負の数は考慮しないとする。

1. 通常、我々は数を表現するとき、先頭に 0 はつけない。つまり、`0012` などとはせず、`12` と書く。上記の 10 進数の正規表現では先頭から 0 がいくつか並ぶ可能性がある。先頭には 0 がこない 10 進数の表現を書け。
2. 実数を表す正規表現を書け。ここでは実数とは `3.14` などの数を指していて、数字の列の後ろに小数点 (`.`) が来てさらに数字の列が続くものとする。2 などの整数もここでは 2.0 と記述することとする。

4.2 字句解析器生成系 Lex

正規表現が一通りわかったところで字句解析器生成系 Lex について述べる (以下、コマンドを表す小文字の lex で表記する)。

まずは lex に与える記述 (これを lex 記述という) の例を示そう。

```
 1 %%
 2 [0-9]+          { printf("Number!!\n"); }
 3 [ \t\n]+        { /* do nothing */ }
 4 .+              { printf("other!!\n"); }
 5
 6 %%
 7
 8 main(){
 9   while(yylex()!=0){
10   }
11 }
```

図 1: lex の最初のサンプル

lex は字句解析器生成プログラムであり、正規表現に基づいた仕様書を受け取り、その正規表現を受理するオートマトンのプログラムを生成する。オートマトンとは日本語では状態遷移機械と呼ばれる。オートマトンについての詳細は他の教科書 ([1, 2] など) に譲る。

さて、図 1 をもとに説明をしよう。

まず、1 行目には%%がある。実はこれよりも上にはいくつかのことが記述できるのだが、ここでは省略している。たとえば、%{と%}で括ったものは、lex が出力する C プログラムの最初のほうにそのままコピーされる。%{と%}を使う例は後ほど紹介する。

また、正規表現に名前を付けることができる。これにより以下で記述される正規表現を簡潔に記すことができる。

たとえば、

```
D          [0-9]
```

のように記述しておくと [0-9]*などと書く代わりに D*のように記述できる。

2 行目には先ほど述べた整数の正規表現とそれに一致したとき (これをマッチ (match) という) の動作を表す C 言語断片が書かれている。ここでは数字 (の並び) を見つけたら、画面上に Number!!と出力する。

3 行目には、(半角) 空白、タブ (\t)、改行コード (\n) が 1 つ以上並んだものにマッチしたら、何もしないことを表している。

4 行目では、. が出てきているが、これは特殊な意味を持ち、**改行以外の任意の 1 文字**を表す。つまり、この行では「改行以外の任意の 1 文字」が 1 個以上並んだ文字列にマッチしたら other!!と出力することを表している。

6 行目の%%はここまでが正規表現とそれに対応するアクション (動作) の定義部分であることを表している。そして、%%以降には任意の C 言語のプログラム断片を書くことができる。アクションなどで必要となる関数はここに記述する (このことについては後述)。

8〜11 行目では main 関数を記述している。9 行目の yylex は lex が生成する字句解析関数である。yylex はある 1 つの字句を見つけるかファイルの終わりに達したら呼び出し側に制御を戻す。ファイルの終わりだった場合には 0 を返す。通常はアクションにおいてトークンに対応する数値を返却するように記述する。

さて、一通りの説明が終わったところで、この lex 記述を打ち込み、lex に与えて、実行可能形式を作るまでをやってみよう。手順は以下のようになる。

1. 図 1 をエディタを起動して打ち込む。このとき、**正規表現とアクションの間は 1 つ以上空白 (ここでいう空白は半角空白かタブ) で区切る必要がある。**見易さのため、図 1 のようにタブを用いてアクション部分が同じ位置から始まるようにしておくのがよい。ここでは sample1.l というファイル名で保存することにする。
2. lex 記述を打ち終えたら、これを lex に与える。

   ```
   uni% lex sample1.l
   ```

   すると lex.yy.c という C 言語のソースプログラムが生成される。もし、エラーメッセージが出た場合にはエディタでの編集に戻ってエラー個所を修正する。
3. lex.yy.c が生成されたら、これを C コンパイラに与え、実行可能形式を作成する。

   ```
   uni% cc lex.yy.c -ll
   ```

   最後についている-ll は lex を使う際に必要なライブラリのリンクを指示している。

以上で実行可能形式 a.out ができたはずである。

演習 2

上記の a.out を実行し、色々な文字列を入れてみよう。文字列を打ち込んだら必ずリターンキーを押すこと。プログラムの終了は Ctrl-D である。

得られた結果が思った通りになったかどうか検討し、それぞれ入力した文字列に対してなぜそのような結果になるのか考察しよう。

4.3 特定文字列

字句解析器で切り出したい文字列としては特定の文字列の場合もある。

たとえば、C 言語では if や while といった文字列は予約語となっており、それらは英文字の列であっても識別子とはならず、特定の意味を持つ。これらを lex 記述に記述するのは簡単である。単に正規表現のところにそれを表す文字列を記述すればよい。ただし、lex が使う特殊な文字の場合、その効果を打ち消す必要がある。効果を打ち消すための記号は\である。たとえば、C 言語においては [は配列要素の指定をするための記号であり、これを lex として認識させたいとする。しかし、lex では [は文字の種類の指定に使われているので、そのまま記述することはできない。このとき、\[のように記述する。

あるいは、\を使う代わりに"で括る方法もある。"["のように記述するのである。上記の if や while などを書くときもこうしておくのがよい。すなわち、"if"や"while" のように記述しておくのである。

4.4 あいまいさと lex での規則

ここでは [0-9]+と .+という正規表現が出てきた。特に後者についてよく考えてみよう。. は「改行以外の任意の 1 文字」を表すから当然その中には [0-9] も含まれる。そうするとたとえば、123 というのは両方の正規表現にマッチするわけである。つまり、両方の規則に一致するというのは該当する規則が一意に定まらないということであり、このようなことを「あいまいである」という。lex ではこのような場合どうするかの解決の規則を設けている。それは、最初に書かれた正規表現の方を採用するというものである。

もう 1 つのパターンを考えてみよう。123abc と言う入力があった場合にこのプログラムはどう振舞うであろうか。

123 までは 2 行目と 4 行目の正規表現の両方にマッチする。しかし次の 1 文字 a を見たとき、ここでは 4 行目の正規表現にだけマッチする。つまり、123a は 2 行目の正規表現にはマッチしないが、4 行目の正規表現にはマッチする。以降、123abc まで同様である。

ここである人は、最初の 123 は 2 行目の正規表現に一致し、残りの abc が 4 行目の正規表現に一致するというように考えるかもしれない。

少し検討してみよう。例えば、C 言語の識別子 (名前) について考えてみよう。キーワードの if は別途"if"として定義が書かれているとする。fopen(ifile, "abc.c"); のように書かれているときにこのうちの ifile を if と ile に分けられても困るわけである。

このような問題を解決するために lex では**最長一致**という考え方を採用している。つまり、最も長い入力にマッチするものを採用するのである。このため、123abc は 4 行目の正規表現に一致すると考えるのである。

よく考えるとこれはもっともらしい考え方である。すなわち、最長一致でなければ、123 に対しては 3 回 Number!!が表示されてしまうのである。

4.5 …以外

文字列のパターンを考える際、「…以外の文字からなる文字列」を指定したい場合がある。ここでは「…以外」ということを表す方法を説明する。たとえば、a 以外を表すには [^a] とする。

数字と数字以外のものを切り分けることを考えてみよう。図 1 の最後の正規表現を「0-9 以外の並び」を表すように変更しよう。sample2.l と言うファイルに sample1.l をコピーしてから行うこと。

(できればここで以下を見ないで正規表現をまず考えよう。)

「0-9 以外の並び」を表す正規表現は [^0-9]+となる。図 1 の 4 行目をこのように変えて、字句解析器プログラムを作ってみよう。手順は次のようになる。

1. sample2.l を編集する

2. lex sample2.l
3. cc lex.yy.c -ll

では実行してみよう。123 と入力すれば Number!!と出力される。ところが abc(改行) と入力しても何も出てこない。続いて 456(改行) と入力すれば other!!と Number!!が出てくる。abc123 と入力すれば Number!!と other!!が出てくる。

このことについて少し考えてみよう。本当は改行を押すたびにそれぞれに切り分けてほしいところである。

気づいている人もいると思うが、[^0-9] の中には空白やタブ、改行コードも含まれている。最長一致の原則から行末が [0-9] 以外で終わっていたら行末の改行コードもそれに連続する文字列とみなされ、その次の行を見るまで結果は出力されないのである。

ここでは区切り文字は区切り文字として別個に認識してほしい。このため、図 1 の 4 行目は [^0-9 \t\n] とすべきである。

演習 3

ここで sample3.l として sample2.l をコピーし、以下の要件を満たす字句解析器を作成せよ。各出力はその最後で改行するようにせよ。

1. 整数を見つけたら画面に Number!!と出力する。
2. 空白 (半角空白、タブ、改行コード) は区切り文字とする。これらを見つけたときは何もしない。
3. +, -, *, /のいずれかを見つけたら、それぞれに対して plus, minus, mult, div のいずれかを出力する。
4. それ以外の任意の**1 文字**を見つけたら画面に error!!と出力せよ。

4.6 Lex のその他の機能

Lex で扱える正規表現には表 1 のものがある。

ここまでに紹介したことで大体の場合に対応できるが、ここでは更なる機能を紹介しておく。現時点ですべてを理解しなくてもよいが、そのような機能があることだけを知っていれば、後々、何か lex を使ってプログラムを作成しなければいけない場合、解決の目途がついてプログラミング作業が楽になる。

表 1: Lex で利用可能な正規表現

正規表現	機能
"a"	括られた文字列そのものを表す。
"abc"	
\	特殊文字の打ち消し。\自体の打消しは\\
\n	改行
\t	タブ
\b	バックスペース
[abc]	文字の集合。[と] で囲まれた文字のどれか 1 文字
[a-z]	[と] で囲まれている場合のみ-は範囲を表す。
[^abc]	[と] で囲まれている場合のみその先頭に^を書くことで以降に書いた文字種を除いた文字を表す。
^abc	この場合の^は**行の先頭**を意味する。上記の [と] で囲まれた場合とは別なので注意が必要である。
abc$	$は行末を意味する。
.	改行以外の任意の 1 文字
|	選択。a|b は a もしくは b を意味する。
?	直前の文字 (正規表現) があってもなくてもよいことを示す。ab?c は ac もしくは abc にマッチする。
*	直前の正規表現の 0 回以上の繰り返し。
+	直前の正規表現の 1 回以上の繰り返し。
a{1,5}	a の 1 から 5 回の繰り返し。
a{3}	a の 3 回の繰り返し。
a{2,}	a の 2 回以上の繰り返し。
()	優先順位を変更。(a|b)*と a|b*は意味が異なる。

余談だが、他の事柄も同様で、「○○を使えば××ができる」ということを知っていれば、後に自分がそのような問題を解決する際に非常に有益である。このため、まったく何もしないよりは、そのような機能があるということを一通り知っておくことには意義がある。

4.6.1 / オペレータ

さて、まずは/オペレータについて説明する。ある文字列 x に対して、その後ろに別のある文字列 y が続く場合に限り、x の部分がマッチしたことにしたいときがある。もう少し具体的に言うと ab は abcd というように後ろに cd が来る場合にだけ、ab として取り出したいとしよう。これを表すのが ab/cd という正規表現である。

4.6.2 yy...

Lex は後に述べる yacc と共同で使われるよう設計がされている。Lex には字句解析をする段階で得られた情報などを保持しておくいくつかの (大域) 変数が用意されている。

それらは接頭辞 (prefix)[1]が yy となっている。

そのような変数はいくつか用意されているが、ここでは yytext についてのみ記しておく。

yytext にはそのパターンにマッチしたときにマッチした文字列が格納される。実際には yytext は char* 型である。当然であるが、マッチしてから次のマッチまでの間だけ、マッチした文字列は保持されている。

いま、各単語の長さを調べ、また、全体で何単語あったかを出力する字句解析器を作成するとしよう。ここでは単語とは英文字の並びとする。そのようなプログラムは図 2 のようになる。

```
 1  %{
 2  int wordno=0;
 3  %}
 4  %%
 5  [A-Za-z]+   { printf("%4d %s\n",
                         strlen(yytext),
                         yytext);
                wordno++; }
 6  .           { /* do nothing */ }
 7  "\n"        { /* do nothing */ }
 8
 9  %%
10
11  main(){
12    while(yylex()!=0){
13    }
14
15    printf("the number of words is %d\n",
             wordno);
16  }
```

図 2: 単語の長さを調べるプログラム

まず、1〜3 行目で、単語数を数えるための変数を宣言している。

5 行目が単語に対応する正規表現である。英文字には小文字と大文字があるのでその両方のすべての文字種が 1 文字以上並んだものをここでは単語とみなしている。このとき、対応するアクションとして、その単語の長さとその単語を構成する文字列を表示している。

[1]文字列の先頭部分

そして、1 つの単語として認識されたので、単語を数える変数の値を 1 つ増やしている。

6 行目はそれ以外の (改行以外の)1 文字が来た場合の処理である。ここでは何もしない。

7 行目には"\n"がある。. は**改行を除く任意の 1 文字**であるから、改行が来たときの処理をする。ここでも今回のプログラムでは改行は何の意味も持たないので、何のアクションも起こさない。

演習 4

sample3.l をもとに sample4.l を以下の条件に従って作成せよ。

1. まず、実数を認識できるような正規表現を入れよ。ここで実数とは「数字の列、小数点、数字の列」となる正規表現で表すとする。対応するアクションとして Real!!と表示するようにせよ。
2. C 言語など多くの言語では、整数を実数として扱う場合、2. などの表記を許している。すなわち、整数に対して末尾に小数点をつけることで実数として読み取るのである。上の正規表現を書き直し、小数点以下の数字の列は**あってもなくてもよい**ものにせよ。

5 宿題

以下のものを作れ

1. ファイルの中の文字数、単語数、行数を数える wc というコマンドがある。lex を使ってこれを作ってみよう。ここでの単語とは半角空白 (' ')、タブ ('\t')、改行コード ('\n') のいずれかで区切られたその他の文字列のことである。図 2 を改造することでこのプログラムを作成せよ。
   そして自分が作ったプログラムと wc の結果を比較して、正しく動作することを確認せよ。
2. C 言語の識別子 (名前) を見つけたら画面上にそれが含まれる行番号とその名前を表示するプログラムを作れ。C 言語の予約語が見つかった場合は名前とはしない。例えば、if が見つかっても画面に表示はしない。
   これを満たす実行可能形式を

~nakai/Lesson/2001/IPP2/identifier として置いておく。このプログラムでは識別子に一致すると printf("%4d: %s\n", ... として表示するとした。
次のようにして、動作確認を行える。

```
uni% a.out < lex.yy.c > out1
uni% ~nakai/.../identifier < lex.yy.c > out2
uni% diff out1 out2
```

diff の結果として何も表示されなければ期待した結果になっている。

宿題には提出義務はない。

6 演習問題の解答

- 演習 1
  1. (0|[1-9][0-9]*)
  2. [1-9][0-9]*\.[0-9]+

- 演習 3

```
 1 %%
 2 [0-9]+  { printf("Number!!\n"); }
 3 [ \t\n]+        { /* do nothing */ }
 4
 5 "+"             { printf("plus\n"); }
 6 "-"             { printf("minus\n"); }
 7 "*"             { printf("mult\n"); }
 8 "/"             { printf("div\n"); }
 9
10 .               { printf("error!!\n"); }
11
12 %%
13
14 main(){
15   while(yylex()!=0){
16   }
17 }
```

図 3: 演習 3 の解答

- 演習 4

  正規表現のみ示す。

  1. [0-9]+\.[0-9]+
  2. [0-9]+\.([0-9]+)?

参考文献

[1] 中田育男: **コンパイラ**, オーム社, 1995.

[2] 佐々政孝: **プログラミング言語処理系**, 岩波書店, 1989.