ソフトウェア演習 1
第 2 回レポート課題

J2200071 齊藤 隆斗

2024 年 5 月 21 日

1　課題 3.7&3.8+ 応用課題 3.9

課題 3.7
算術式の構文解析ルーチンを参考にして正規表現の構文解析ルーチンを作成し、入力正規表現の構文木を表示するプログラムを作成せよ. 正規表現の*は、算術式の符号の場合と異なり、後置演算子であることに注意する必要がある.

課題 3.8
引数 -d0, -d1, -d2 を指定することによって、入力正規表現、トークン列、構文木の 3 つをそれぞれ表示できるように main 関数を変更せよ.

応用課題 3.9
通常の grep プログラムでは、連結の '.' は省略されるのが普通である. そのような場合、上の方法で正規表現の構文解析がうまく行く確かめよ. うまくいかないとすれば、どのように方針を修正すれば良いだろうか？

2　実行結果

課題 3.7 が正常に動作することを確認

実行例 1: テキストの例について確認

```
$ ./kadai2 'a.\|.b'
    LETTER(a)
CONC
        LETTER(|)
    CONC
        LETTER(b)
```

実行例 2: テキストの例について確認

```
$ ./kadai2 '(a|b*)|c'
        LETTER(a)
    VERT
            LETTER(b)
        AST
VERT
    LETTER(c)
```

実行例 3: 追加の例についても確認

```
$ ./kadai2 '((a|b)*|\e).\0'
                LETTER(a)
            VERT
                LETTER(b)
        AST
    VERT
        EPSILON
CONC
    EMPTY
```

実行例 4: 右結合であることを確認

```
$ ./kadai2 'abcd'
    LETTER(a)
CONC
        LETTER(b)
    CONC
            LETTER(c)
        CONC
            LETTER(d)
```

課題 3.8 が正常に動作することを確認

実行例 5: d0 オプションの動作について確認

```
$ ./kadai2 -d0 '(a|b*)|c'
(a|b*)|c
```

実行例 6: d1 オプションの動作について確認

```
$ ./kadai2 -d1 '(a|b*)|c'
LPAR
LETTER(a)
```

```
VERT
LETTER(b)
AST
RPAR
VERT
LETTER(c)
EOREG
```

実行例 7: d2 オプションの動作について確認

```
$ ./kadai2 -d2 '(a|b*)|c'
        LETTER(a)
    VERT
            LETTER(b)
        AST
VERT
    LETTER(c)
```

実行例 8: 予期しないオプションに対するエラー処理について確認

```
$ ./kadai2 -d3 '(a|b*)|c'
invalid option
```

応用課題 3.9 が正常に動作することを確認

実行例 9: テキストの例について確認

```
./kadai2 -d2 'ab*c'
    LETTER(a)
CONC
            LETTER(b)
        AST
    CONC
        LETTER(c)
```

実行例 10: 追加の例について確認

```
$ ./kadai2 -d2 'a(\0|\e)*|b\e'
        LETTER(a)
    CONC
                EMPTY
            VERT
                EPSILON
```

AST
VERT
LETTER(b)
CONC
EPSILON

3 プログラムの流れ

大まかなプログラムの流れは次のようになる.
はじめに、与えられた引数の検証を行う. 引数が存在しなかったり、引数がオプションと正規表現の 2 つよりも多くの引数が入力されてしまっている場合、そして、不当なオプションが与えられている場合はエラー処理を行う. オプションについて、d0 の場合は入力した正規表現をそのまま出力しプログラムを停止する、d1 の場合は入力した正規表現のトークン列を出力し停止する.d2 の場合は与えられた正規表現に対する構文木を出力し停止する. オプションが入力されなかったときはデフォルトの処理として、与えられた正規表現に対する構文木を出力する.
オプションが d0 または d1 であったときについてのプログラムの流れについて、d0 は正規表現を出力するだけ、d1 は前回の課題で扱った lexer() 関数を実行するだけであるから省略する. d2 または、オプションが与えられなかったとき、すなわち構文木を出力する場合についてのプログラムの流れについて大まかに見ていく. このサブルーチンは、一文字ずつ入力を読み込んでいき構文規則にしたがって構文木を生成していく. 生成の仕方は考察のセクションで触れる. 構文木を正常に生成し終われば、その構文木を出力して処理を停止する.

4 考察

この課題で考えたこと、工夫した点は 2 つある.
1 つ目は、サブルーチン parse() の再帰的な実装である. 再帰を用いず、手続き的に処理を行うとすると、VERT や CONC などの二項演算子の場合や、AST などの被演算子を前半に取るような一項演算子では、構文木を作成する際に文字を読み込んだ後にもう一度もどって処理を行う必要がある. 例えば AST の場合では、AST をルートに取る必要があるので、AST を読み込みルートに設定したうえで AST の被演算子をそのルートの子に設定する. 今回は再帰を用いており読み込んだ文字を逆行するようなことはせず、サブルーチンを再帰的に呼び出すことによって構文木を生成していく. 例として a|b について考える. まず、a を読み込み、eval_primary() によって処理が行われ、構文木のルートに a が格納される. この eval_primary() によって、| が読み込まれていて、再帰的に呼び出しているので呼び出し元の関数に戻っていき、条件文によって引っかかり、実行されるのは、eval_term() である. この関数は今まで作成した木を CONC をルートとした構文木において、CONC の左部分木として格納する. 同様にこの関数によって、次の文字は呼び出されているので b を再帰的に処理をすると LETTER(b) をルートとする子を持たない木が作成され、それを先程作成した CONC をルートとする右部分木に格納する. そして、次のトークンが EOREG となったので処理を parse() を終了する. このようにスタックを利用し、上位のサブルーチンを保留しておくことで再帰的に構文木を作成することができる.
2 つ目は、応用課題 3.9 における'.' を省略しても連結として処理できるようにするプログラムの拡張であ

る．実装のアイデアは、CONC の直後に現れるトークンは、EPMTY,EPSILON,LETTER または LPAR のいずれかであるから、eval_term() 関数において、curr_token が CONC でない場合について、これらの EMPTY,EPSILON,LETTER または LPAR のいずれかの場合について、CONC と同様の処理を行えば良いというアイデアである．CONC であった場合の処理については、get_token() によって curr_token を CONC からつぎの文字に移している．しかし、今回は CONC の直後に現れるトークンがすでに読み込まれていて、curr_token に格納されている状態であるから、get_token() 関数を呼び出す必要がないことに注意する．