ソフトウェア演習 1
第 3 回課題

J2200071 齊藤 隆斗

1. 課題 4.5, 課題 4.7-4.9 のプログラム

課題 4.5

頂点 nv1, nv2 がすであったとして、ラベル label をもつ、頂点 nv1 から頂点 nv2 への辺をグラフ追加する関数 add_nedge 作成せよ.

課題 4.7

残りの関数を記述せよ.

課題 4.8

ptree_to_nfa 関数を記述せよ.

課題 4.9

構文木から NFA を生成するプログラムを完成させ、オプションで指定することによって、空遷移を含む NFA(ラベル付き有向グラフ+初期頂点+受理頂点) も表示できるように main 関数を変更せよ.

2. 実行結果

実行例 1: テキストの例について正常に動作するかどうかを確認

```
$ ./kadai3 -d3 'a|b*|c'
 a:   0 =>   1
\e:   1 =>  10
 b:   2 =>   3
\e:   2 =>   4
\e:   3 =>   2
\e:   4 =>   8
 c:   5 =>   6
\e:   6 =>   8
\e:   7 =>   2
\e:   7 =>   5
\e:   8 =>  10
\e:   9 =>   0
\e:   9 =>   7
Initial state:   9
Final state:  10
```

実行例 2: テキストの例について正常に動作するかどうかを確認

```
$ ./kadai3 -d3 'a.b*|c'
 a:   0 =>   1
\e:   1 =>   2
 b:   2 =>   3
\e:   2 =>   4
\e:   3 =>   2
\e:   4 =>   8
 c:   5 =>   6
```

```
\e:   6 =>   8
\e:   7 =>   0
\e:   7 =>   5
Initial state:   7
Final state:   8
```

実行例 3: EMPTY が含まれる場合に正常に動作するかどうかを確認

```
$ ./kadai3 -d3 '\0.a'
\e:   1 =>   2
 a:   2 =>   3
Initial state:   0
Final state:   3
```

実行例 4: EPSILON が含まれる場合に正常に動作するかどうかを確認

```
$ ./kadai3 -d3 '\e*'
\e:   0 =>   1
\e:   0 =>   2
\e:   1 =>   0
Initial state:   0
Final state:   2
```

実行例 5: 追加の例について正常に動作するかどうかを確認

```
$ ./kadai3 -d3 '(a|b)*.c'
 a:   0 =>   1
\e:   1 =>   5
 b:   2 =>   3
\e:   3 =>   5
\e:   4 =>   0
\e:   4 =>   2
\e:   4 =>   6
\e:   5 =>   4
\e:   6 =>   7
 c:   7 =>   8
Initial state:   4
Final state:   8
```

3. プログラムの流れの説明

今回のプログラムでは構文木から NFA に変換するようなプログラムを作成したので、d1, d2オプションが指定された場合のプログラムの流れは省略する. ここでは、d3 オプションが指定された場合、すなわち構文木を NFA へ変換し、 表示する関数 make_nfa()についてプログラムの流れについて説明する.

まず、第一回の課題で作成した関数 get_token()で、１つ目のトークンを取得する. その後、第 2 回課題で作成した関数 eval_expr()によって、そのトークンから構文木を生成する. この構文木が生成し終わった時点で curr_token の値が EOREG でない場合は解析エラーであるから、parse_error()関数によってエラー処理を行う. 正常に解析ができた場合は、関数 gen_nfa()によって NFA を構成する. この関数 gen_nfa()の詳細な流れは考察で触れる.その後、構成した NFA を表示する. そして、構文解析木と NFA で確保していたメモリを開放し、処理を停止する.

4. 考察

今回の課題で考えたこと、工夫したことは 2 つある.

1 つ目は関数 gen_nfa()のアイデアである. この関数は ptree_to_nfa()によって、得られた正規表現の構文解析木を NFA に変換し、 ptree_to_nfa()の関数の終了後に初期頂点と終了頂点をセットしてこの関数の処理を終了する. ここで、関数 ptree_to_nfa()が再帰的に呼び出されることに注意する. 例として正規表現$a|b$の場合について見ていく. この正規表現を構文解析した場合、ルートに|、ルートの左の子にa、 ルートの右の子にbとなるような木が生成される. よって、最初は ptree_to_nfa()にはルートのトークンである VERT が引数として渡される. これは case 文によって、VERT の処理が行われる.ここでは、まず ptree_to_nfa()に左の子を引数に渡し処理を行う. すなわち、LETTER(a)の NFA を生成する.この際にこの NFA の初期頂点と終了頂点の情報を含む構造体を n1 に格納しておく. 右の子 LETTER(b)についても同様の処理をおこない、この時に部分的に生成した NFA の初期状態と終了状態の情報を含む構造体を n2 に格納しておく. n1, n2 の部分的に生成した NFA の初期状態と終了状態の情報と部分的に生成した 2 つの NFA から gen_vert_nfa()関数によって$a|b$の NFA を構成する. このように NFA を再帰的に構成される流れについて考えた.

2 つ目は関数 add_nedge()の実装である.add_nedge()は nv1, nv2, label を 引数にとり、nv1 に、辺の行き先であるノード nv2 とラベル情報 label が付与された辺を追加する. この辺を追加する際には、まずノード nv1 に辺があるかどうかを調べる必要がある. これは、すでに他の辺が追加されている場合とまだ辺が追加されていない場合とで処理を分ける必要があるからである. まだ辺が追加されていない場合は、nv1 に辺の行き先 nv2 と辺のラベル label の情報を付与した要素をリンクすれば良い. すでに辺が追加されている場合は、nv1 にすでにリンクされている要素を辿っていき、次の要素へのポインタが NULL の要素に、nv1 に nv2 と label の情報を付与した要素をリンクする. このような流れによってノード nv1 からノード nv2 につながるようなラベル label が付与された辺を 追加することができると考えた.