高度サイバーセキュリティPBL II

2023 正規表現

株式会社ティアフォー / 大阪大学

高野 祐輝

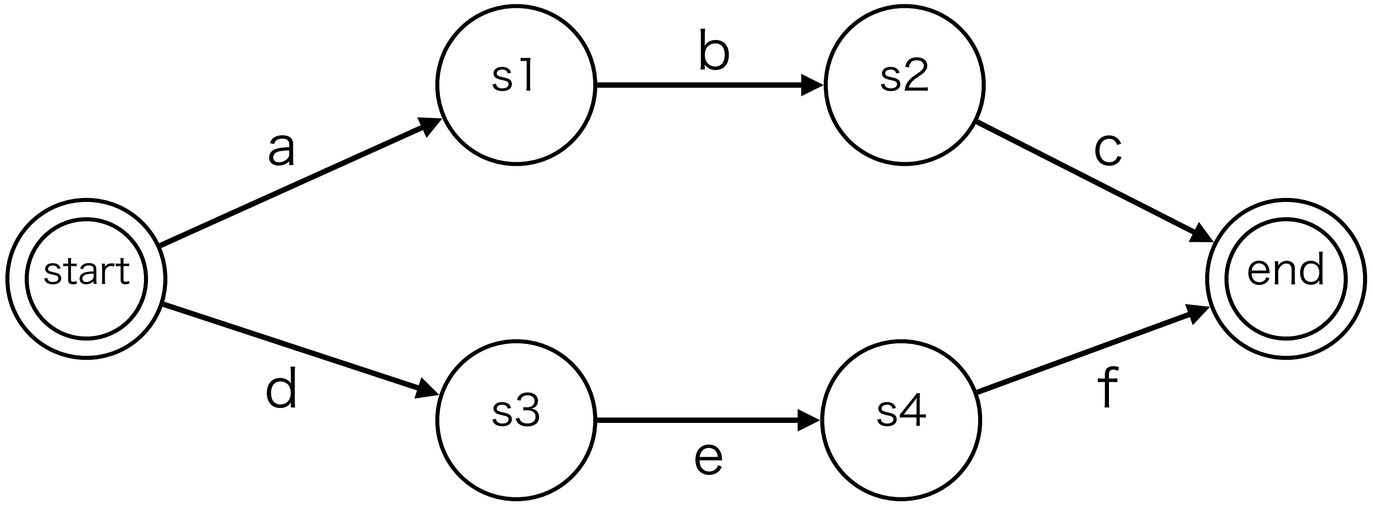
もくじ

- オートマトンと正規表現
- レジスタマシンと正規表現
- 深さ優先と幅優先探索
- 正規表現エンジンの実装

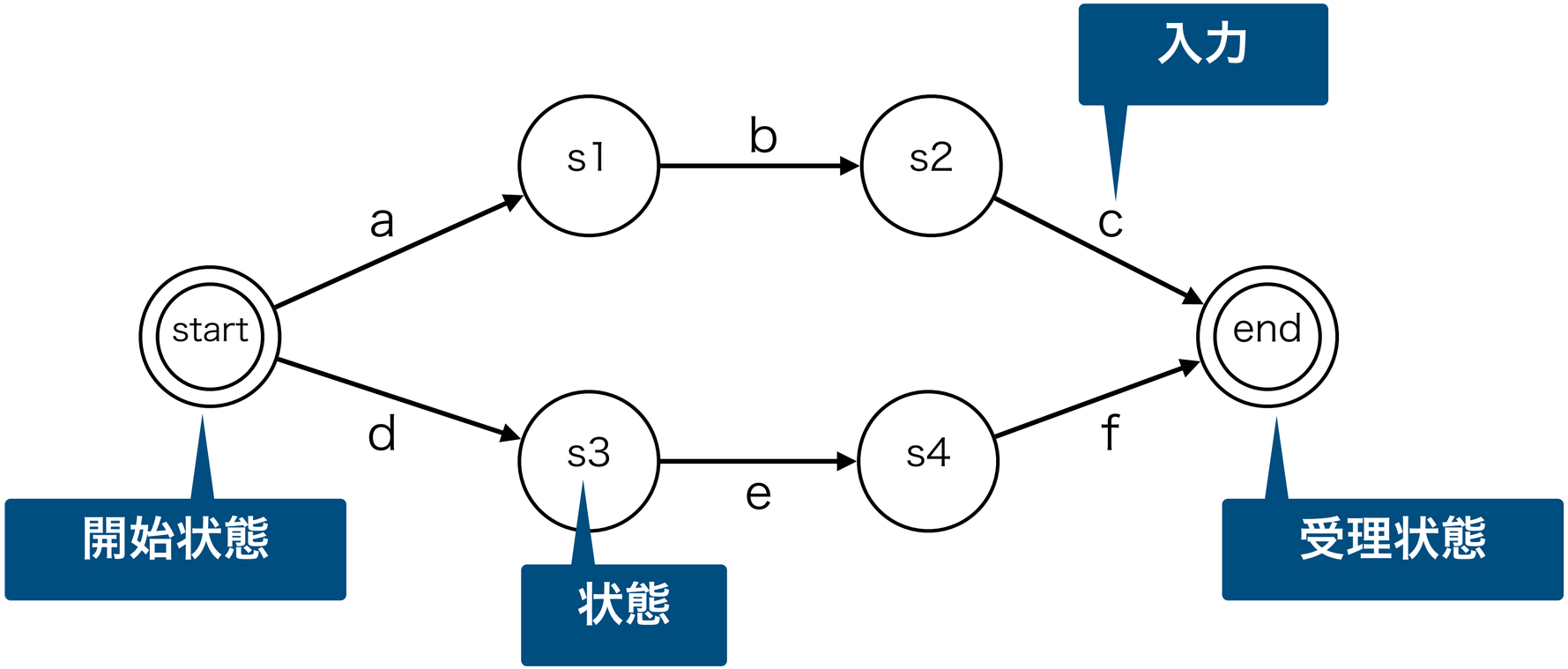
オートマトンと正規表現

有限オートマトン

- 有限な状態の遷移を表すために利用される
- ノードとエッジというグラフ構造で表現可能

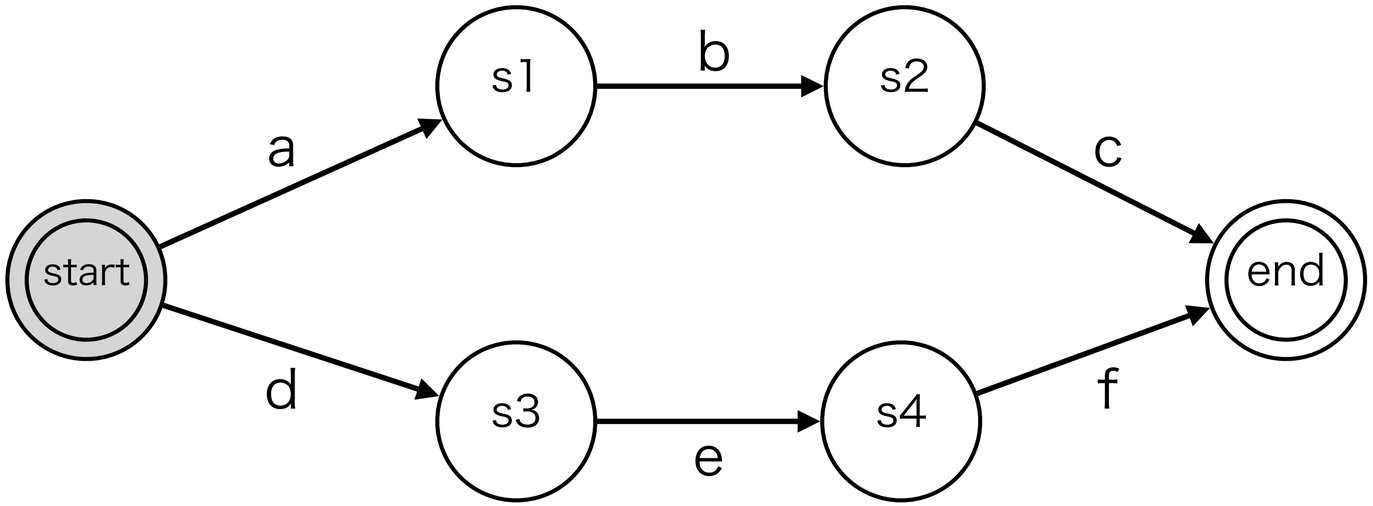


有限オートマトンの要素



有限オートマトンの遷移例（1/5）

- 入力文字列をabcとした場合



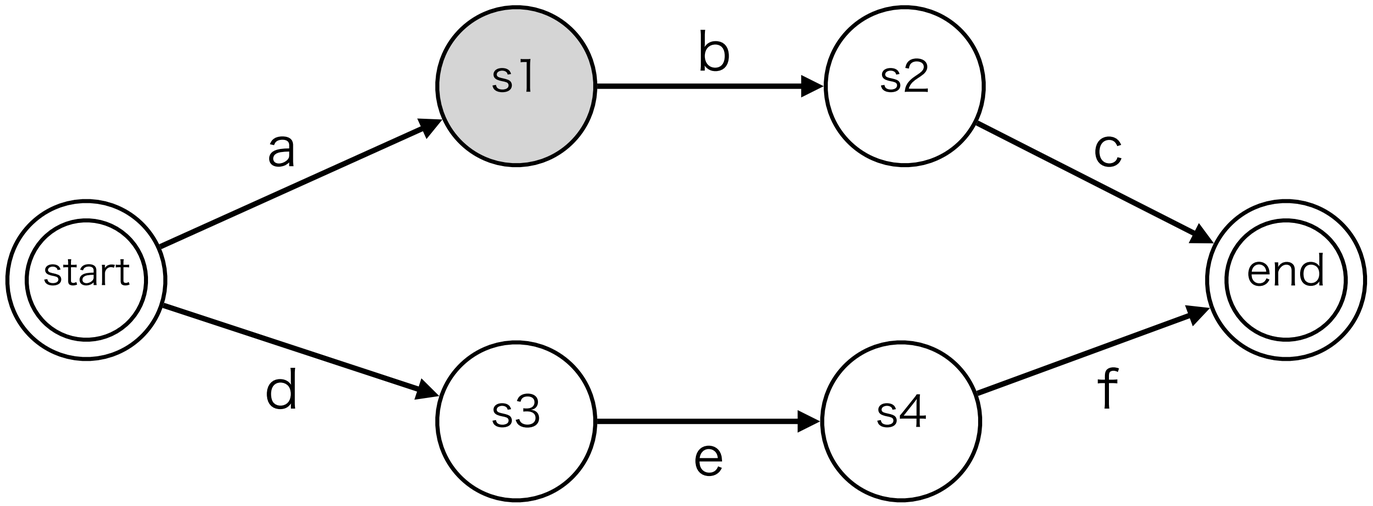
abc

↑

次入力

有限オートマトンの遷移例（2/5）

- 入力文字列をabcとした場合



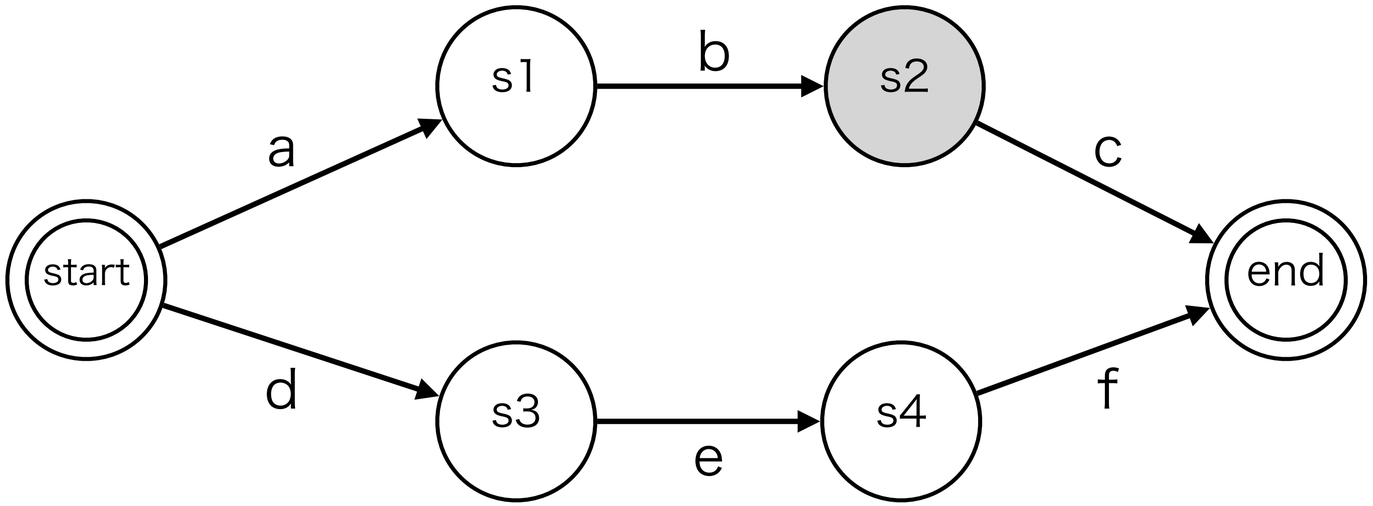
abc

↑

次入力

有限オートマトンの遷移例（3/5）

- 入力文字列をabcとした場合



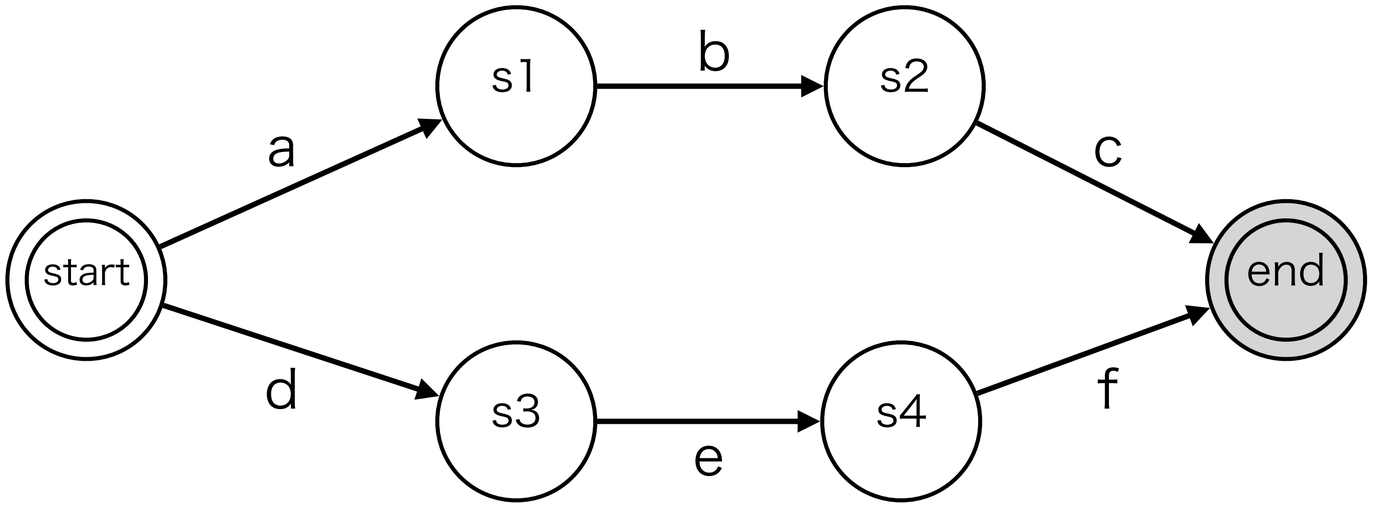
abc

↑

次入力

有限オートマトンの遷移例（4/5）

- 入力文字列をabcとした場合



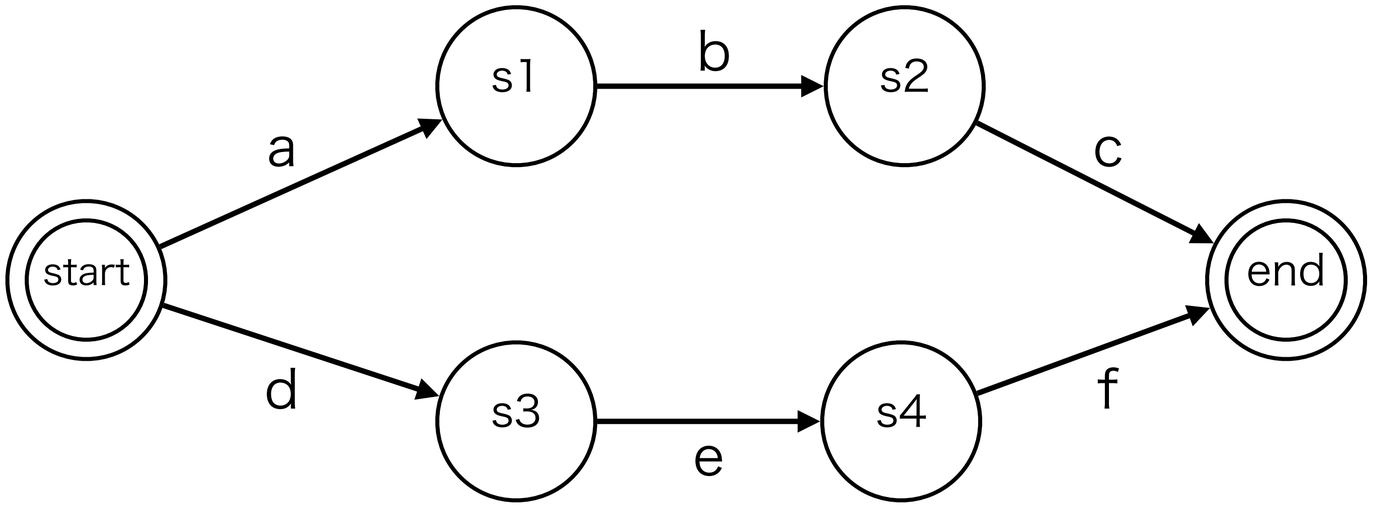
abc

↑

次入力

有限オートマトンの遷移例（5/5）

- 下記オートマトンは、abcまたはdefという入力を受理するオートマトン

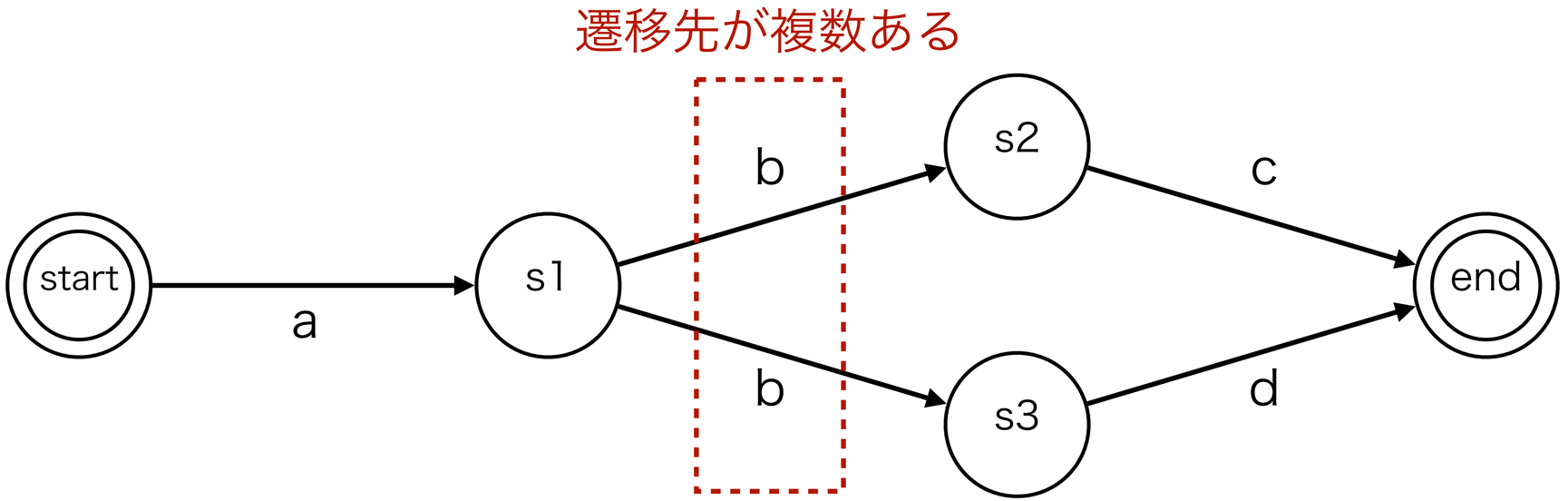


[決定性|非決定性]有限オートマトン

- 遷移先が必ず一つに限られるようなオートマトンを決定性有限オートマトン（Deterministic Finite Automaton, DFA）と呼ぶ
- 一方、遷移先が複数ある場合のオートマトンを非決定性有限オートマトン（Nondeterministic Finite Automaton, NFA）と呼ぶ

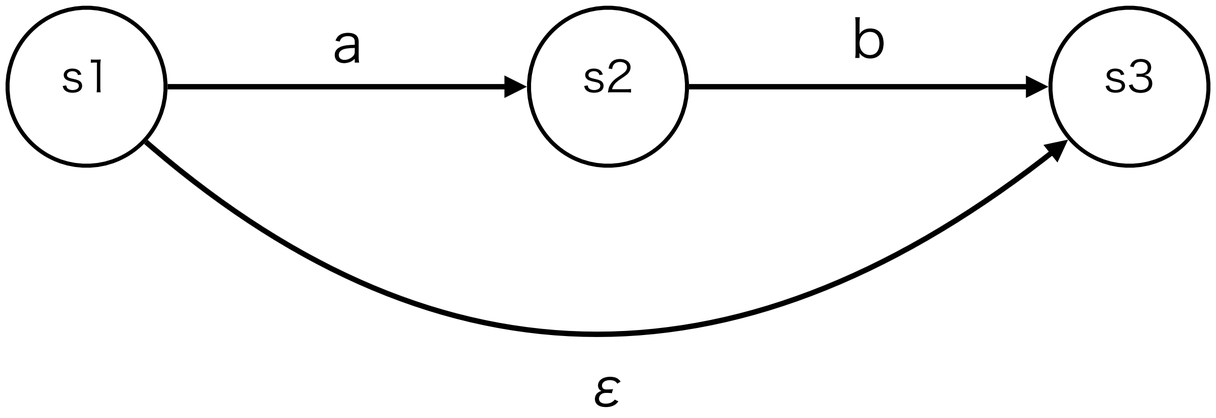
非決定性有限オートマトンの例

- 以下のオートマトンは、abc, abdという入力を受理する



ε 遷移

- 何も入力なしでも遷移する遷移を ε 遷移と呼ぶ
- 下図のs1からs3へは、入力無しで遷移可能



正規表現

- 文字のパターンを判別するための表現
- 受理できる言語の種類は有限オートマトンと等価だが、図や表ではなくテキストで表すことができるために利用される事が多い

簡単な正規表現用言語

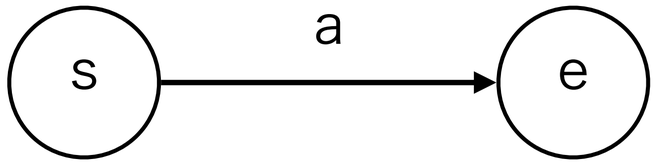
- e1|e2：e1かe2
- e*：eの0回以上の繰り返し
- e+：eの1回以上の繰り返し
- e?：eが0回か1回出現する
- (e)：優先順位の変更

正規表現の例

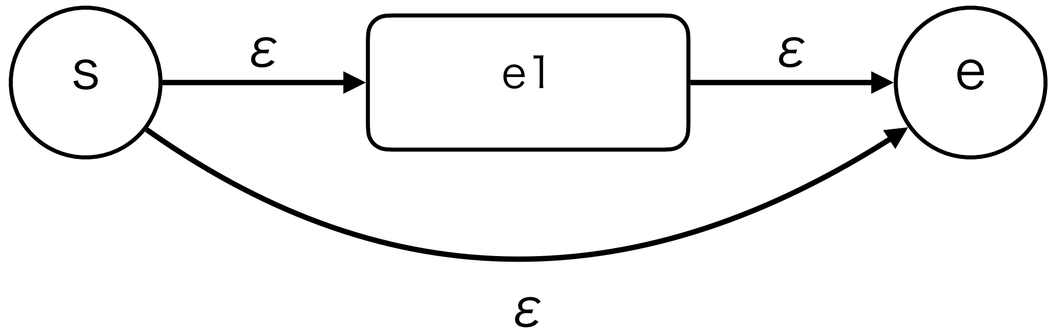
- “https?”：httpか、httpsという文字列にマッチする正規表現
- “(let)+”：let、letlet、letletletなどの文字列にマッチする正規表現
- “200|404”：200か、404という文字列にマッチする正規表現

正規表現からNFAへの変換

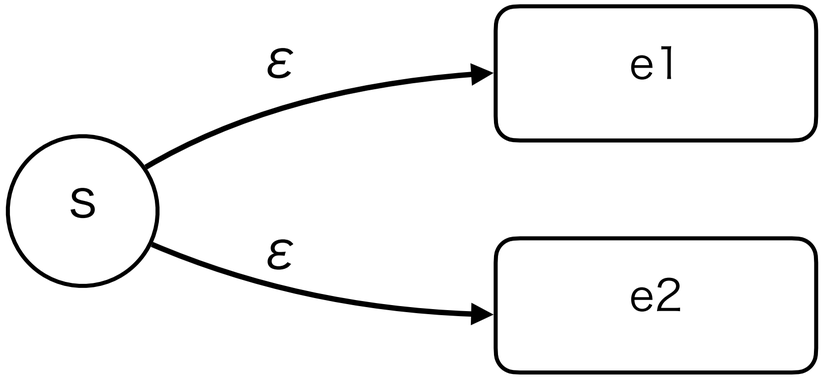
正規表現：a



正規表現：e1?

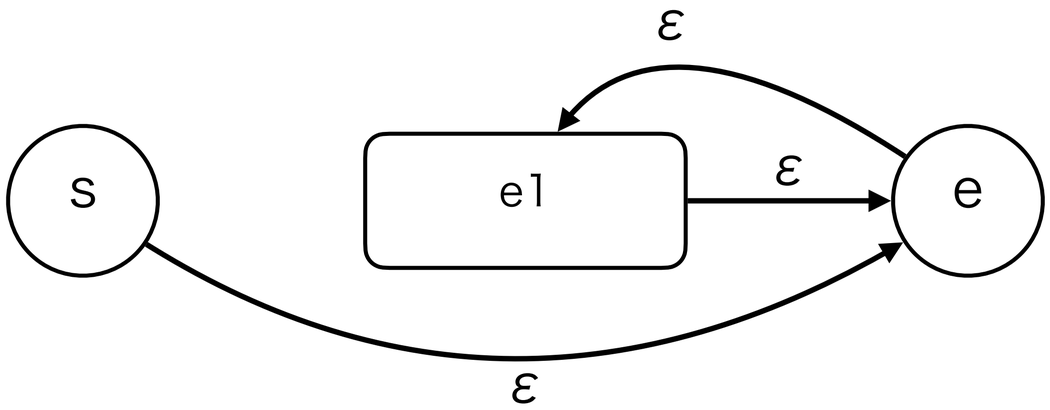


正規表現：e1|e2

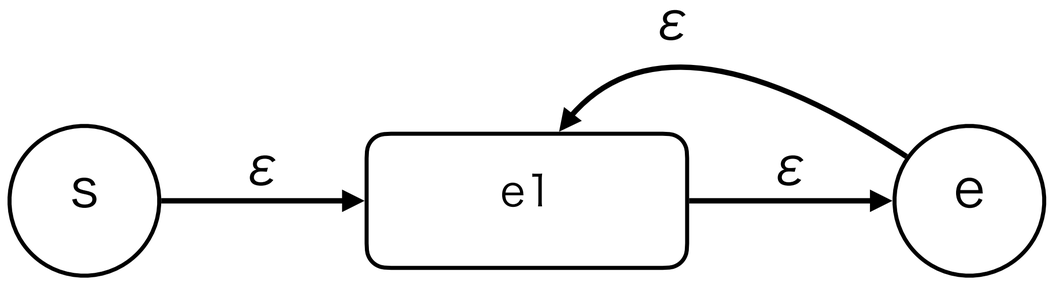


正規表現からNFAへの変換

正規表現：e1*



正規表現：e1+



レジスタマシンと正規表現

レジスタマシン

- 我々が普段使っているコンピュータはレジスタマシンと呼ばれている
- レジスタと呼ばれる有限個のメモリ領域を持つ
- 限定された命令セットを持つ
- 命令列を読み込んで実行する
- 正規表現エンジンはレジスタマシンで表現可能

正規表現用レジスタマシン

register	description
PC	program counter
SP	string pointer

opcode	operands	description
char	c	if *SP != c then fail; else SP++ and PC++
match		found
jmp	x	PC = x (jump to the address x)
split	x, y	clone (one thread's PC = x, and another's PC = y)

コード生成規則

regex	assembly
c	`char c`
e1 e2	`codes for e1` `codes for e2`

e1 | e2	`split L1, L2`
	`L1: codes for e1`
	`jmp L3`
	`L2: codes for e2`
	`L3:`

regex	assembly

e?	`split L1, L2`
	`L1: codes for e`
	`L2:`

e*	`L1: split L2, L3`
	`L2: codes for e`
	`jump L1`
	`L3:`

e+	`L1: codes for e`
	`split L1, L2`
	`L2:`

コード生成の例、文字編

正規表現：abcd

利用した規則

regex	assembly
c	char c

アセンブリコード

```
1: char a
2: char b
3: char c
4: char d
5: match
```

最後にmatchを挿入

正規表現用レジスタマシンの実行例

実行例、文字編（1/5）

入力：abcd

↑

SP

アセンブリコード

PC ⟶

```
1: char a
2: char b
3: char c
4: char d
5: match
```

step 1, 初期状態

実行例、文字編（2/5）

入力：abcd

アセンブリコード

```
1: char a
2: char b
3: char c
4: char d
5: match
```

PC ⟶ 2: char b

step 2

実行例、文字編（3/5）

入力：abcd

↑

SP

アセンブリコード

```
1: char a
2: char b
3: char c
4: char d
5: match
```

PC ⟶ 3: char c

step 3

実行例、文字編（4/5）

入力：abcd

SP

アセンブリコード

```
1: char a
2: char b
3: char c
4: char d
5: match
```

PC ⟶ 4: char d

step 4

実行例、文字編（5/5）

入力：abcd

アセンブリコード

```
1: char a
2: char b
3: char c
4: char d
5: match
```

PC ⟶ 5: match

step 5、マッチ

正規表現用レジスタマシンの失敗する実行例

失敗する実行例、文字編（1/2）

入力：adef

↑

SP

アセンブリコード

PC ⟶

```
1: char a
2: char b
3: char c
4: char d
5: match
```

step 1, 初期状態

失敗する実行例、文字編（2/2）

入力：adef

SP

アセンブリコード

PC ⟶

```
1: char a
2: char b
3: char c
4: char d
5: match
```

step 2、マッチ失敗

正規表現用レジスタマシンの失敗する実行例 その2

別の失敗例、文字編（1/2）

入力：abc

SP

アセンブリコード

```
1: char a
2: char b
3: char c
4: char d
5: match
```

PC ⟶ 3: char c

step 3

別の失敗例、文字編（2/2）

入力：abc

SP

アセンブリコード

```
1: char a
2: char b
3: char c
4: char d
5: match
```

PC ⟶ 4: char d

step 4, 入力文字がこれ以上無いので失敗

コード生成の例、or編

正規表現：ab|cd

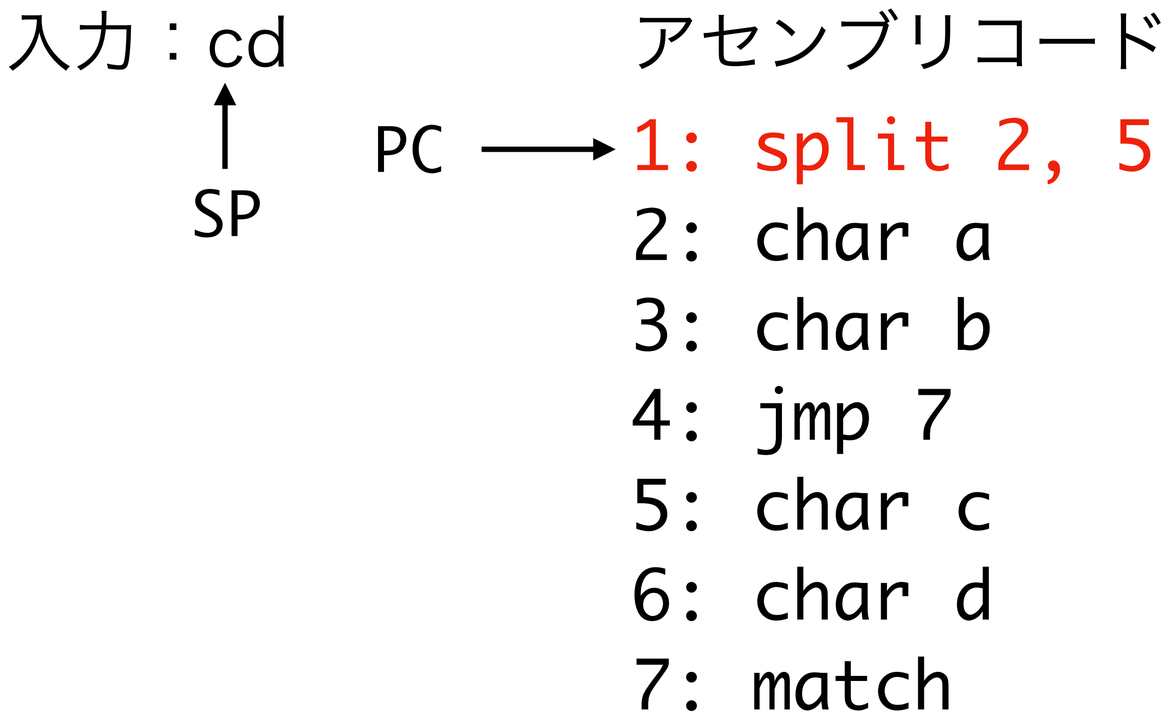
アセンブリコード

```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

利用した規則

regex	assembly
c	char c
e1 | e2	split L1, L2 L1: codes for e1 jmp L3 L2: codes for e2 L3:

実行例、or編（1/4）



step 1, 初期状態

実行例、or編（2/4）

thread 1:

入力：cd

SP

PC ⟶

アセンブリコード

```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

thread 2:

入力：cd

SP

PC ⟶

アセンブリコード

```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

step 2

実行例、or編（3/4）

thread 1:

入力：cd

SP

PC → Fail!

アセンブリコード

```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

thread 2:

入力：cd

SP

PC →

アセンブリコード

```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

step 3、thread 1は失敗

実行例、or編（4/4）

thread 2:

入力：cd

SP

アセンブリコード

```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

PC

step 3、マッチ終了

コード生成の例、?編

正規表現：a(bc)?d

アセンブリコード

```
1: char a
2: split 3, 5
3: char b
4: char c
5: char d
6: match
```

利用した規則

regex	assembly
c	char c
e?	split L1, L2 L1: codes for e L2:
e1 e2	codes for e1 codes for e2

コード生成の例、*編

正規表現：a(bc)*d

アセンブリコード

```
1: char a
2: split 3, 6
3: char b
4: char c
5: jmp 2
6: char d
7: match
```

利用した規則

regex	assembly
c	char c
e*	L1: split L2, L3 L2: codes for e jump L1 L3:
e1 e2	codes for e1 codes for e2

コード生成の例、+編

正規表現：a(bc)+d

アセンブリコード

```
1: char a
2: char b
3: char c
4: split 2, 5
5: char d
6: match
```

利用した規則

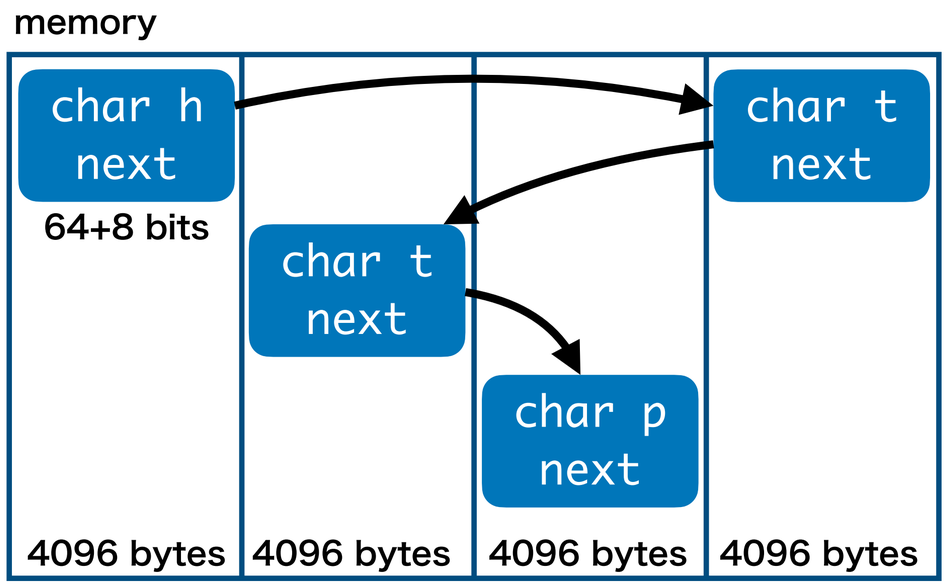
regex	assembly
c	char c
e+	L1: codes for e split L1, L2 L2:
e1 e2	codes for e1 codes for e2

splitの実装に関する考察

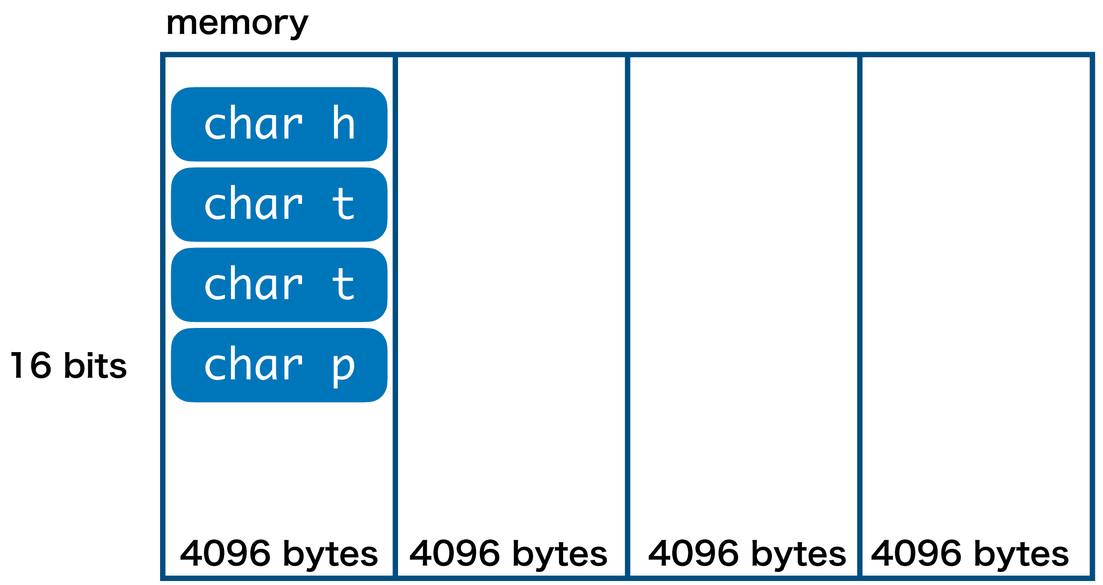
- 実際にOSのthreadを使う方法
  - OSのスレッド生成や同期処理は非常に重い
  - よほどうまく実装しないとシングルスレッドよりも遅くなる
- シングルスレッドで深さ優先探索で実装する方法（バックトラックとも呼ばれる）
  - 実装が簡単
  - しかし、非常に遅くなる場合がある
- シングルスレッドで幅優先探索で実装する方法
  - 実装が複雑
  - 多くの場合で高速に動作する

マシンコード生成の利点

- 構造体とポインタのグラフ形式で表現する場合、必要なメモリ量が多くなる
- CPU Cacheのlocalityが高くなり、CPU Cacheのヒット率が上がる



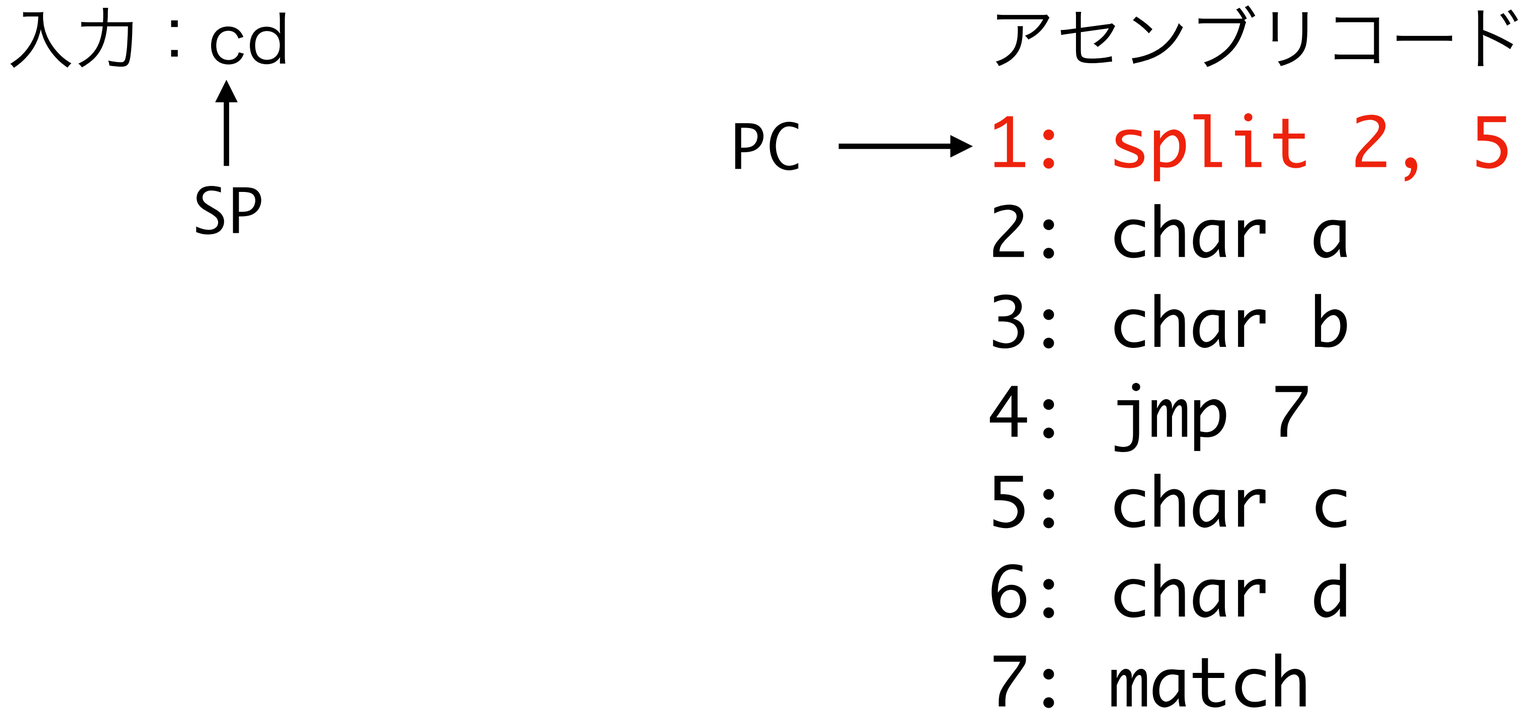
graph representation



machine code representation

深さ優先探索例

深さ優先探索例（1/7）



step 1, 初期状態

深さ優先探索例（2/7）

入力：cd

SP ↑

アセンブリコード

復帰位置 ⟶ 1: split 2, 5

PC ⟶ 2: char a

3: char b

4: jmp 7

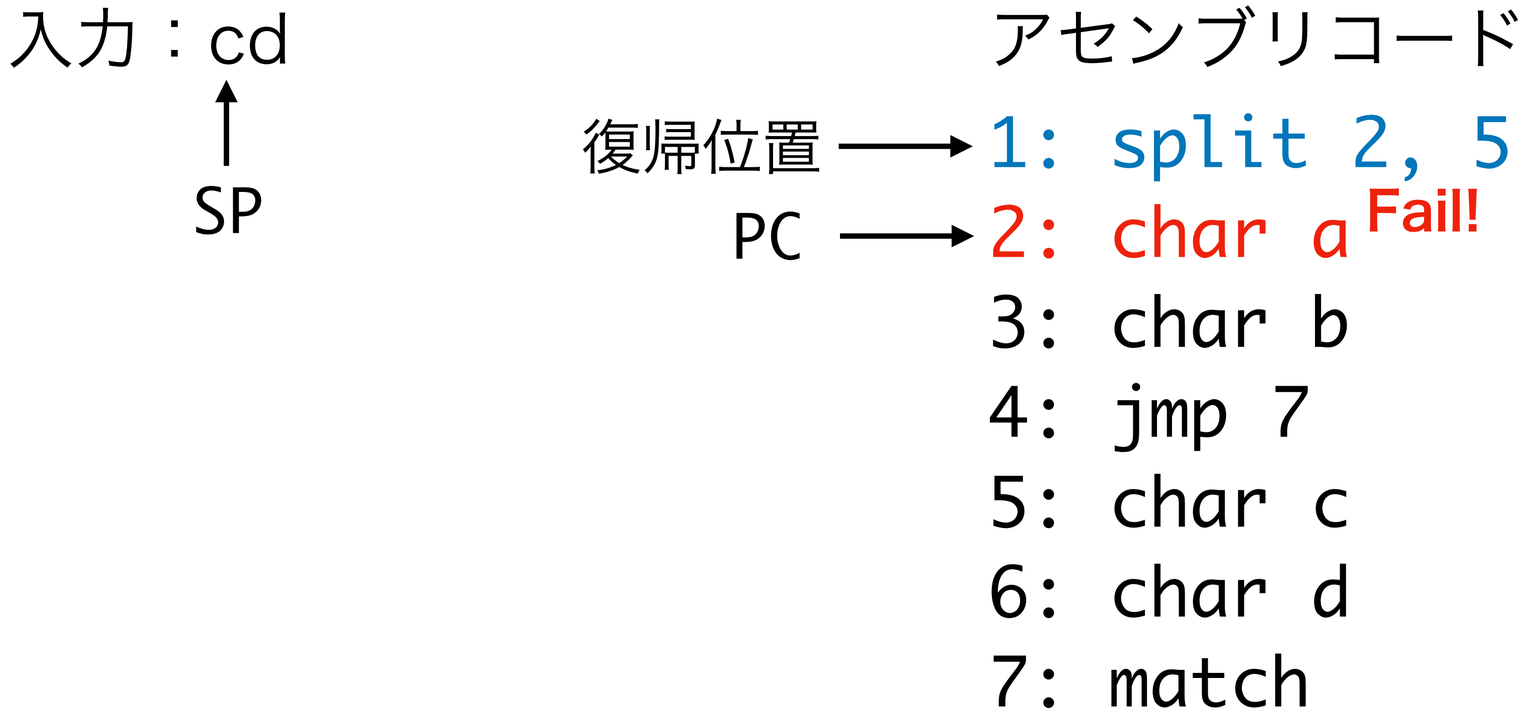
5: char c

6: char d

7: match

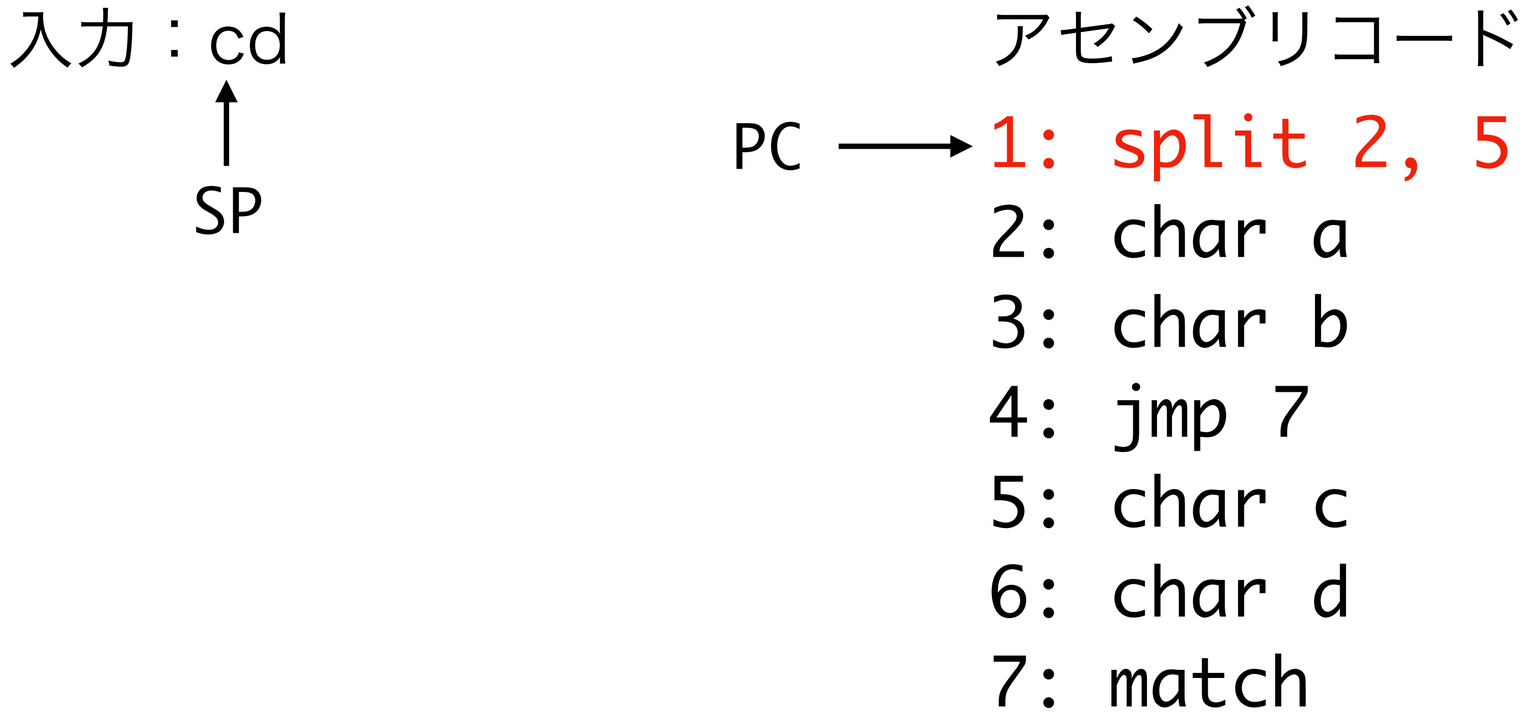
step 2

深さ優先探索例（3/7）



step 3、マッチ失敗したので復帰位置へ戻る

深さ優先探索例（4/7）



step 4

深さ優先探索例（5/7）

入力：cd

SP

アセンブリコード

```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

PC

step 5

深さ優先探索例（6/7）

入力：cd

SP

アセンブリコード

```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

PC

step 6

深さ優先探索例（7/7）

入力：cd

SP

アセンブリコード

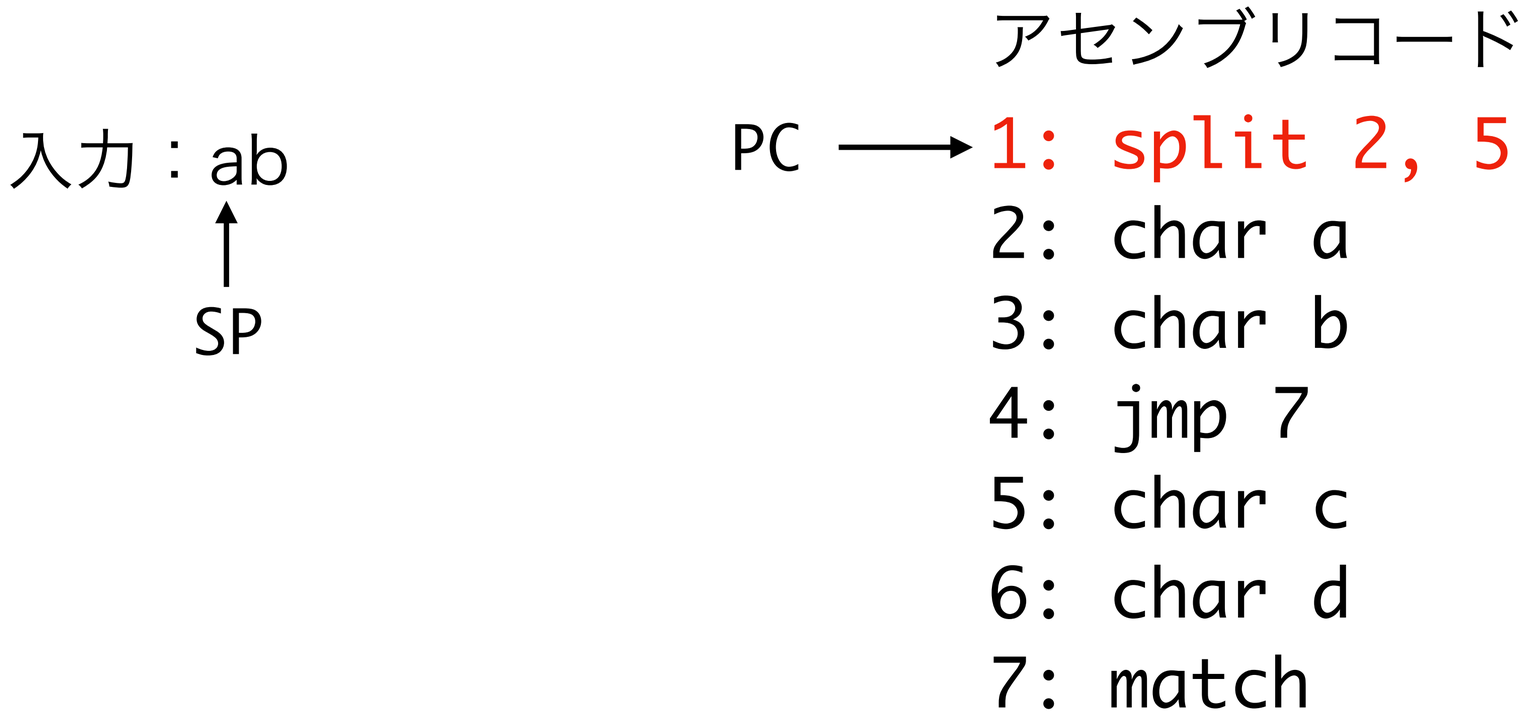
```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

PC ⟶ 7: match

step 7、マッチ

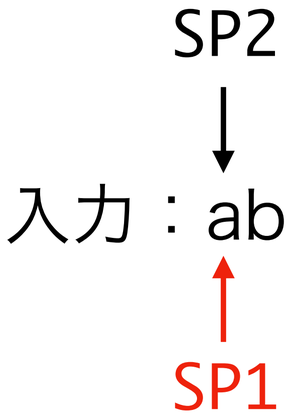
幅優先探索例

幅優先探索例（1/6）



step 1, 初期状態

幅優先探索例（2/6）



アセンブリコード

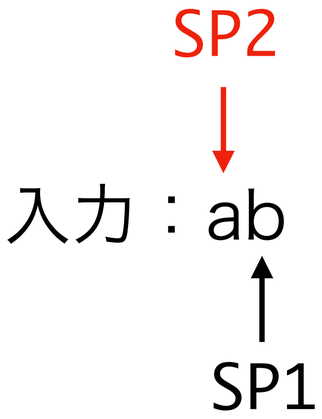
```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

PC1 → 2: char a

PC2 → 5: char c

step 2

幅優先探索例（3/6）



アセンブリコード

```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

PC1 → 3: char b

PC2 → 5: char c Fail!

step 3、マッチ失敗したのでPC2、SP2は削除

幅優先探索例（4/6）

入力：ab

SP1

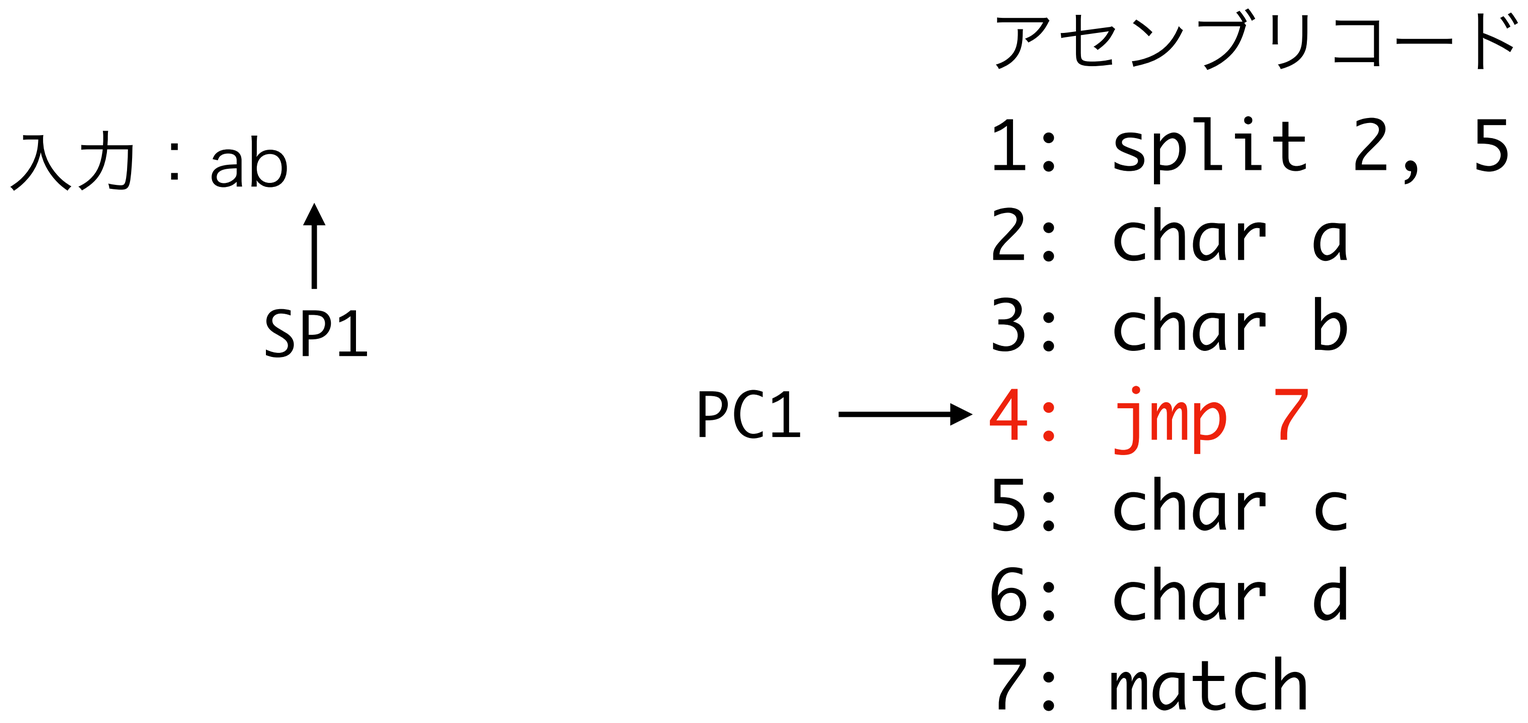
アセンブリコード

```
1: split 2, 5
2: char a
3: char b
4: jmp 7
5: char c
6: char d
7: match
```

PC1 → 3: char b

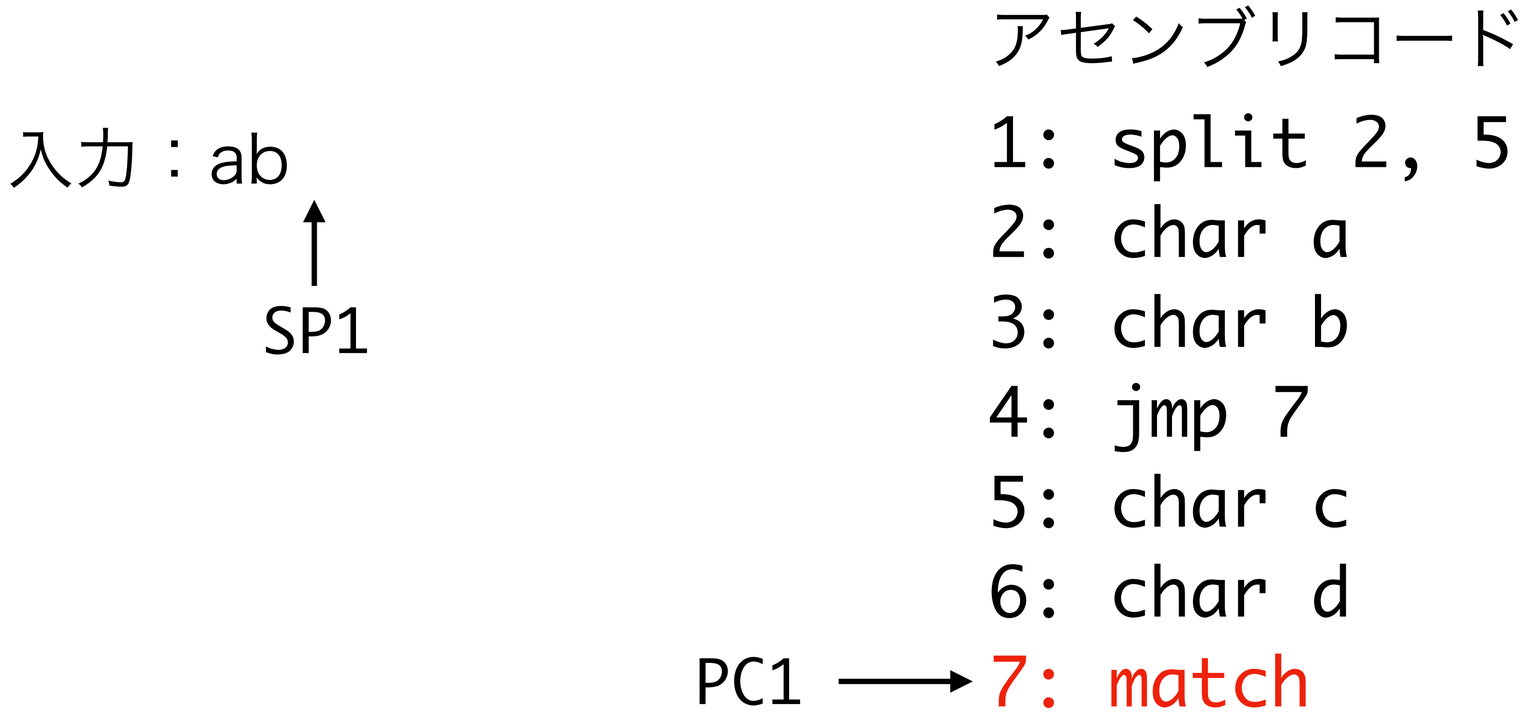
step 4

幅優先探索例（5/6）



step 5

幅優先探索例（6/6）

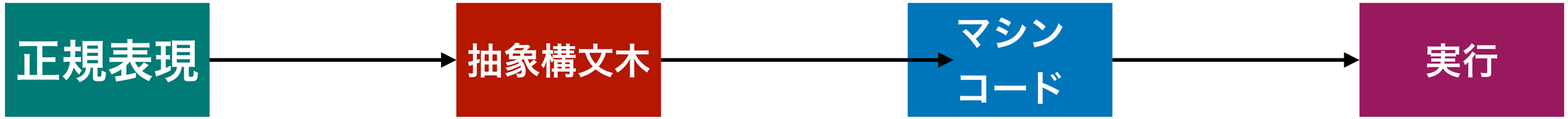


step 6、マッチ

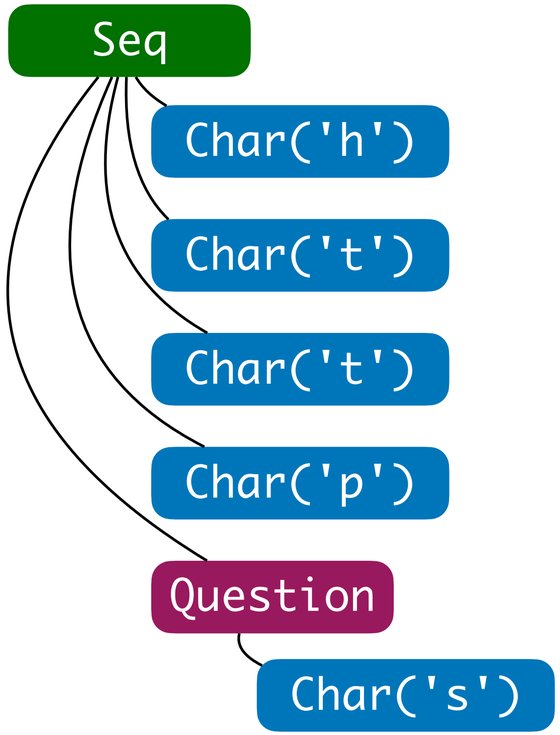
正規表現エンジンの実装

正規表現実行までの流れ

抽象構文木 (Abstract Syntax Tree, AST)



https?

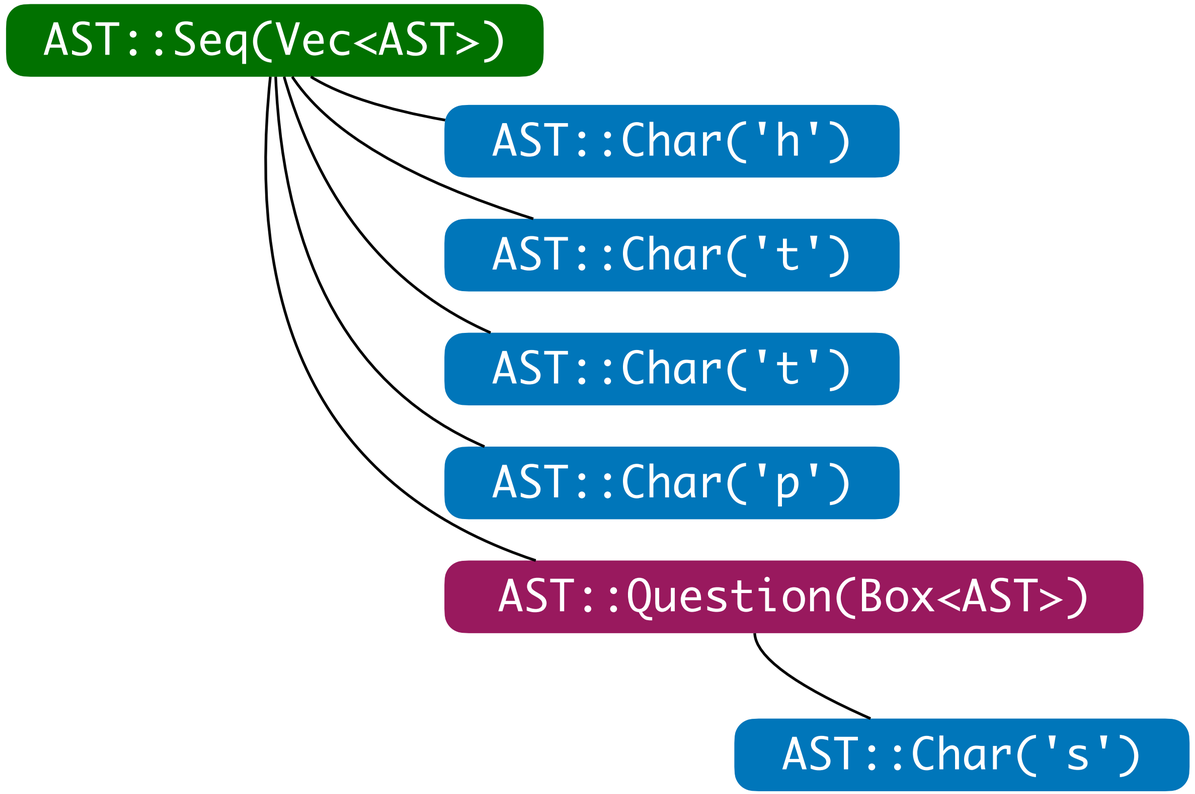



```
0  char h
1  char t
2  char t
3  char p
4  split 5, 6
5  char s
6  match
```

抽象構文木を表す型

```rust
/// 抽象構文木を表現するための型
#[derive(Debug)]
pub enum AST {
    Char(char),
    Plus(Box<AST>),
    Star(Box<AST>),
    Question(Box<AST>),
    Or(Box<AST>, Box<AST>),
    Seq(Vec<AST>),
}
```

https?の抽象構文木



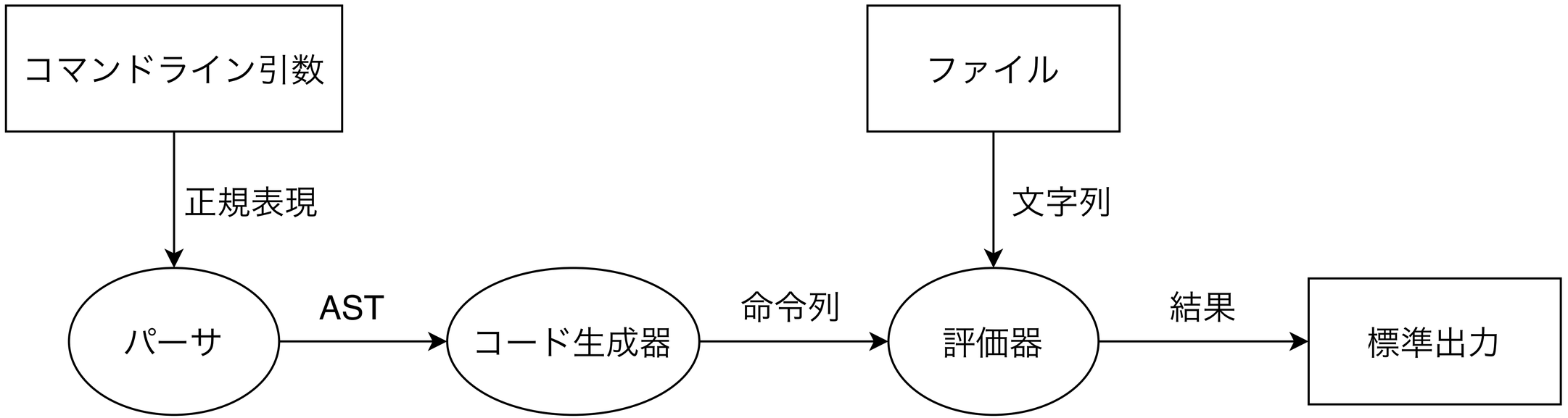
マシンコードを表す型

```rust
/// 命令列
#[derive(Debug)]
pub enum Instruction {
    Char(char),
    Match,
    Jump(usize),
    Split(usize, usize),
}
```

https?のマシンコード

```
0  Instruction::Char('h')
1  Instruction::Char('t')
2  Instruction::Char('t')
3  Instruction::Char('p')
4  Instruction::Split(5, 6)
5  Instruction::Char('s')
6  Instruction::Match
```

処理の流れ



パーサ

engine::parser::parse関数

```
/// 正規表現を抽象構文木に変換
pub fn parse(expr: &str) -> Result<AST, Box<ParseError>> { }
```

これは内部関数含めて実装済み

コード生成器

engine::codegen::get_code関数

```
/// コード生成を行う関数
pub fn get_code(ast: &AST) -> Result<Vec<Instruction>, Box<CodeGenError>> { }
```

内部関数のいくつかは未実装

評価器

engine::evaluator::eval関数

```
pub fn eval(inst: &[Instruction], line: &[char], is_depth: bool) -> Result<bool, Box<EvalError>> {
    if is_depth {
        eval_depth(inst, line, 0, 0)
    } else {
        eval_width(inst, line)
    }
}

/// 深さ優先探索で再帰的にマッチングを行う評価器
fn eval_depth(
    inst: &[Instruction],
    line: &[char],
    mut pc: usize,
    mut sp: usize,
) -> Result<bool, Box<EvalError>> { }
```

evalは内部でeval_depthを呼び出しているのみ。いくつかの処理が未実装

コード生成器用の型

```
/// コード生成器
#[derive(Default, Debug)]
struct Generator {
    pc: usize,
    insts: Vec<Instruction>,
}
```

1命令生成するたびに、pcをインクリメントし、生成した命令をinstsの末尾に追加していく

instsが最終的に生成されるコード列

コード生成器の各関数

Generator::gen_expr

```
/// ASTをパターン分けしコード生成を行う関数
fn gen_expr(&mut self, ast: &AST) -> Result<(), Box<CodeGenError>> {
    match ast {
        AST::Char(c) => self.gen_char(*c)?,
        AST::Or(e1, e2) => self.gen_or(e1, e2)?,
        AST::Plus(e) => self.gen_plus(e)?,
        AST::Star(e) => self.gen_star(e)?,
        AST::Question(e) => self.gen_question(e)?,
        AST::Seq(v) => self.gen_seq(v)?,
    }

    Ok(())
}
```

Generator::gen_char

```
/// char命令生成関数
fn gen_char(&mut self, c: char) -> Result<(), Box<CodeGenError>> {
    let inst = Instruction::Char(c);
    self.insts.push(inst);
    self.inc_pc()?;
    Ok(())
}
```

Instruction::Charを生成し、self.instsにpush、self.pcをインクリメント

self.inc_pcは、オーバーフローを検知しつつインクリメントする関数

Generator::gen_seq

```
/// 連続する正規表現のコード生成
fn gen_seq(&mut self, exprs: &[AST]) -> Result<(), Box<CodeGenError>> {
    for e in exprs {
        self.gen_expr(e)?;
    }

    Ok(())
}
```

再帰的にself.gen_exprを呼び出しコード生成するのみ

Generator::gen_or

- 実際のコードで説明
- 重要なのは、Instruction型の値を生成してコード生成して、inc_pcでpcをインクリメントすること
- さらに、行番号を保存しておくこと

演習（2日目）

2-1. src/engine/evaluator.rs内の、eval_depth関数を実装し、評価器を完成させよ

2-2. src/engine/codegen.rs内の、gen_plus、gen_question、gen_star関数を実装し、コード生成器を完成させよ

2-3. 既存の正規表現エンジンを参考にし、足りない機能を1つ以上追加せよ

　例：任意の文字を表す「.」、行頭を表す「^」、行末を表す「$」

2日目発表スライド

- 17:00〜
- 進捗、質問、感想などを書いてください

レポート

- デバッガ、正規表現の演習を行いレポートとしてまとめよ
  - ソースコードと解説
  - 講義の感想
- 締め切り：2023年7月23日（日）23:59（JST）