

コンパイラ構成

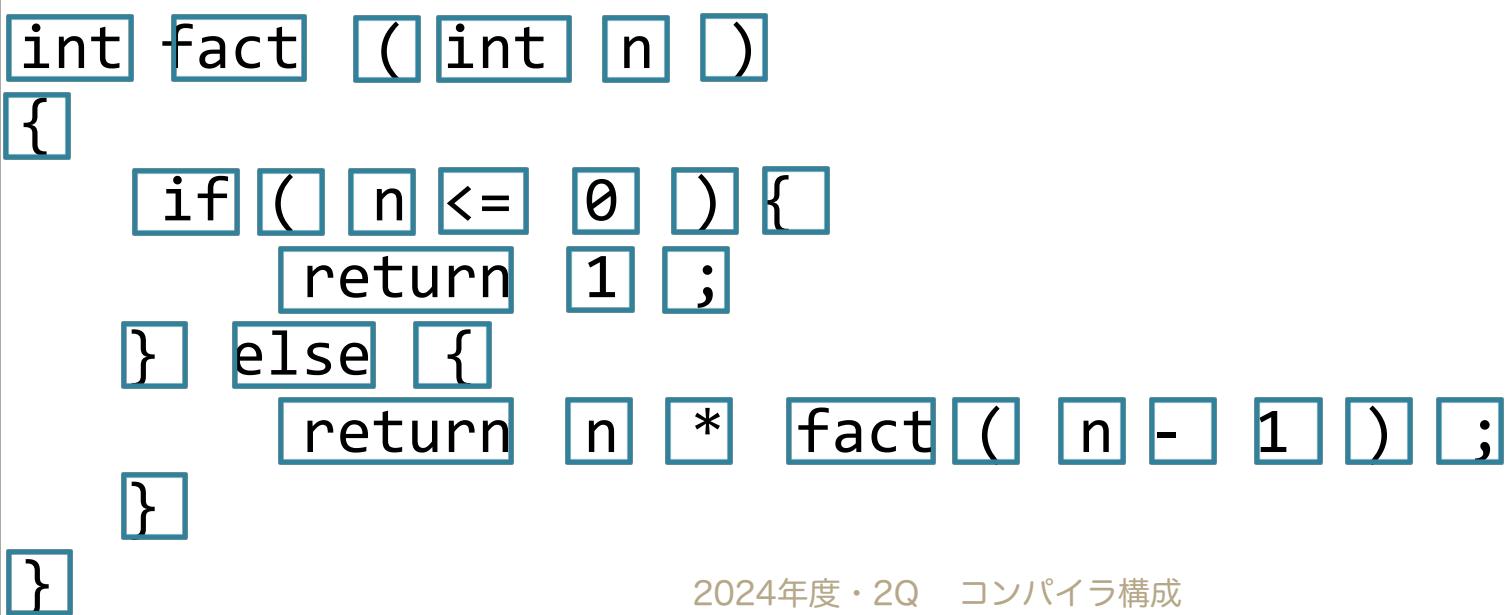
字句解析, 正規表現

情報工学系
権藤克彦



字句解析(lexical analysis)の概要(1/2)

- **字句要素**(lexeme)=プログラムの最小単位の文字列.
- **トークン**(token)=字句要素の種類
 - 予約語 (キーワード) , 識別子 (変数名など) , リテラル (定数) , 演算子, 区切り記号…
- **字句解析**=ソースコードを字句要素
(またはトークン列) に分解すること





字句解析の概要(2/2)

- 通常、字句要素は正規表現で定義する。
 - 10進表記の整数定数の正規表現例
 - $0 \mid ((1|2|3|4|5|6|7|8|9) \cdot (0|1|2|3|4|5|6|7|8|9)^*)$
 - PerlやLexの記法： $0|[1-9][0-9]^*$
 - 読み方「0, または『1から9のどれかで始まり, その後に, 0から9まで0個以上続く文字列』」
(0自身以外には、0で始まる整数定数は含まず)
- 正規表現で字句を定義→プログラムに機械的に変換可能
 - 正規表現
 - 非決定性オートマトン(NFA)
 - 決定性オートマトン(DFA)
 - 状態数を最小化したDFA



識別子(identifier)

unique

- 識別子 = 複数の対象から、特定のものをユニークに（一意に）指定する符号や名称。同姓同名は許さない。
 - 「IDカード」のIDは identity/identification 身分（保証）
- プログラミング言語では、**名前**を使う。
 - 例：変数名、関数名、クラス名、型名…
 - 同じ名前は許すが、スコープ規則などで厳密に区別

```
#include <stdio.h>
int a = 111;
int main (void)
{
    int a = 222;
    {
        int a = 333;
        printf ("%d\n", a);
    }
}
```

やっとバックスラッシュが円記号ではなく
バックスラッシュと表示できた



リテラル(literal)

- リテラル=定数のこと
 - 数値や文字列などのデータを直接的に表現
 - literal の意味は「文字通りの」
- C言語の例：
 - 整数リテラル：10, 0xFF00, 0477
 - 文字リテラル：'a', '\n'
 - 文字列リテラル："hello\n"

正規表現(regular expression)

- 正規表現 = 文字列集合を表現する文字列
- 文字列のパターンマッチや置換に使う



Perlの正規表現実行例（一行野郎で）

```
% perl -ne 'print "|",$&,"|\n" if /[1-9][0-9]*/'
```

aaa	マッチしない		
123		マッチした文字列	パターン
123	マッチした	が入る変数	
0123			
123	123の部分がマッチした		
01230			
1230	1230の部分がマッチした		
^D	Ctrl-d で入力終了		
%			



Perlの正規表現の記法 (1/2)

記法	意味	例	マッチする文字列
.	任意の一文字	.	a
[]	どれか一文字	[abc]	a
[^]	どれか一文字以外	[^abc]	d
-	範囲	[0-9]	5
$\alpha \beta$	連続する α と β	abc	abc
$\alpha \beta$	α または β	abc def	def
(α)	グループ化	(a b c)d	bd
α^*	α を0回以上	(a b c)*d	d
α^+	α を1回以上	(a b c)+d	abbd
$\alpha^?$	α を0~1回	(a b c)?d	d

α と β は任意の正規表現



Perlの正規表現の記法 (2/2)

記法	意味	例	マッチする文字列
^	行頭	^abc	行頭のabc
\$	行末	abc\$	行末のabc
\n	n番目のカッコの中身	([0-9])\1\1	111
$\alpha\{m\}$	α をm回	a{3}	aaa
$\alpha\{m,n\}$	α をm回以上, n回以下	a{3,5}	aaaa
\d	[0-9]と同じ	\d\d	12
\D	[^0-9]と同じ	\D\D	ab
\w	[a-zA-Z0-9_]と同じ	\w\w	a0
\s	空白文字 (スペース, タブ, 改行など)	\s	スペース
\メタ文字	メタ文字自身	\	

メタ文字 = ^ \$ \ | など, 正規表現を書くための記号



Perlの正規表現の優先順位

優先順位	正規表現の機能	例
高い	カッコ (グループ化)	(abc)
	量指定	a?, a*, a+, a{3, 5}
	連接, アンカー	abc, ^, \$
	選択	abc def
低い	アトム	a, [abc], \d, \1

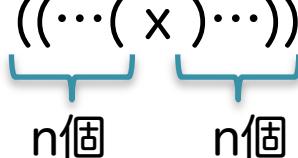
- ab^* は $a(b^*)$ と同じ
- $abc|def$ は $(abc)|(def)$ と同じ



正規表現の記述例

- alpha か beta にマッチ
 - alpha|beta
- alphaの後に間を置いて出現する beta にマッチ
 - alpha.+beta
- apple か apples にマッチ
 - apples?
- //で始まる一行コメントにマッチ
 - //.*\$
- 2016/1/1という形式の日付にマッチ
 - \d{4}/\d{1,2}/\d{1,2}
- カッコで囲まれた0文字以上の文字列にマッチ
 - \(.*\")

正規表現で表現できない文字列

- 規則が再帰的になるなど、有限個の状態で表現できない場合は、正規表現では表現できない。
- 例：開きカッコと閉じカッコがバランスした任意の式
 - $((\cdots(x)\cdots))$


n個 n個
- 文脈自由文法なら簡単に表現できる
 - $A \rightarrow ("A") | x$

優先順位の考慮

- 複数にマッチする場合は優先順位の考慮が必要
- 例：char は以下の両方にマッチする
 - 予約語 : char | else | goto | if | int | return | void | while
 - 識別子 : [a-zA-Z_][a-zA-Z0-9_]*
- 「文字列の否定」は記述が面倒なので、以下のどちらかで通常は対応
 - 先に予約語のマッチを試してから、識別子を試す
 - まとめて識別子で試し、予約語だったら別処理する



最長マッチと最短マッチ (1/4)

- 通常は最長マッチが選ばれる
- 例：パターン ab^* , 文字列 abbbb
 - マッチする部分の候補
 - a 最短マッチ
 - ab
 - abb
 - abbb
 - abbbb 最長マッチ
- 多くの場合, 最長マッチが便利だから
 - 例：10進数整数のパターン $[1-9][0-9]^*$, 文字列 12345+x
 - 1 最短マッチ (嬉しくない)
 - 12345 最長マッチ (嬉しい)



最長マッチと最短マッチ (2/4)

- 実装依存で最長にならない これは細かい話
→ 貪欲マッチ(greedy matching)だから
- 例：パターン **(a|ab)** **(b|bb)**, 文字列 abbb
 - abbb Perl
 - abbb
 - abbb
 - abbb sedなどのUNIXコマンド (POSIX)
- 貪欲マッチ
 1. 繰り返し表現 (*や+) を最大回数マッチさせる
(さらに最初に出て来た表現を優先)
 2. 失敗したら回数を減らす (バックトラック)
 3. それでもダメなら開始位置をずらす これは超大事！

注意：マッチのアルゴリズムやメタ文字は
言語やライブラリ毎に微妙に違う



最長マッチと最短マッチ (3/4)

- 最長マッチが困る場合がある
- 例：文字列リテラルのパターン ".*"
 - "111" 最短マッチ (嬉しい)
 - "111", "222" 最長マッチ (嬉しい)
- パターンを工夫すれば対応可能 (でも面倒)
 - 例："[^"]*" (中身はダブルクオート以外)
- Perlでは最短マッチのパターン記述可能
 - 例：パターン ".*?"
- GNU flexでは「状態」という機能を使う
- 手書きの字句解析プログラムなら最短マッチは簡単



最長マッチと最短マッチ (4/4)

- 文字列リテラルなどではエスケープシーケンスへ要対応
 - 例："abc\"def" (\でエスケープすれば"の出現OK)
- 対応例："(\\"|[^\"])*"
 - 意味：中は「\」か「"以外の文字」の0回以上の繰り返し
- エスケープシーケンス (escape sequence)
 - 特別な文字（列）を表現するための記法
 - C言語等では、文字列中に「"」を含めるには「\"」と書く
 - エスケープシーケンスがあるので正規表現はややこしい

アルファベットと文字列

- アルファベット(alphabet) Σ =文字の有限集合
 - 例： $\Sigma = \{0, 1\}$ は文字0と1からなるアルファベット
 - アルファベットは言語の構成単位を定義する
- 文字列(string)=アルファベット中の文字の0個以上の並び
- 空文字列(empty string)=長さが0の文字列. ϵ と書く
- 連接(concatenation)
 - 2つの文字列をつなげる演算
 - 文字列 s と t の連接を $s \cdot t$ と書く
 - 例： $010 \cdot 101 = 010101$
 - 任意の文字列 s に対して， $\epsilon \cdot s = s \cdot \epsilon = s$

正規表現の定義

- (狭義の) 正規表現の形式的定義
 - ε は正規表現. $a \in \Sigma$ の時, a は正規表現
 - r と s が正規表現の時, $r|s$, $r \cdot s$, r^* , (r) は正規表現
- 正規表現の例
 - $\Sigma = \{0, 1\}$ の時, $\varepsilon, 0, 1, 0 \cdot 1, 0^*, 0|1, (0|1)^* \cdot 1$ はどれも正規表現
- 捕捉
 - 優先順位は $* > \cdot > |$ (左ほど強い). $(0|(0 \cdot (1^*))) = 0|0 \cdot 1^*$
 - \cdot と $|$ は結合律 (associative law) が成り立つ
 - 例: $(0 \cdot 1) \cdot 0$ と $0 \cdot (1 \cdot 0)$ は同じなので $0 \cdot 1 \cdot 0$ と書いて良い
 - 誤解の恐れが無い時は, \cdot を省略. 例: $0 \cdot 1$ を 01 と書く.
 - $r \cdot r^*$ を r^+ とも書く. 糖衣構文 (syntax sugar)

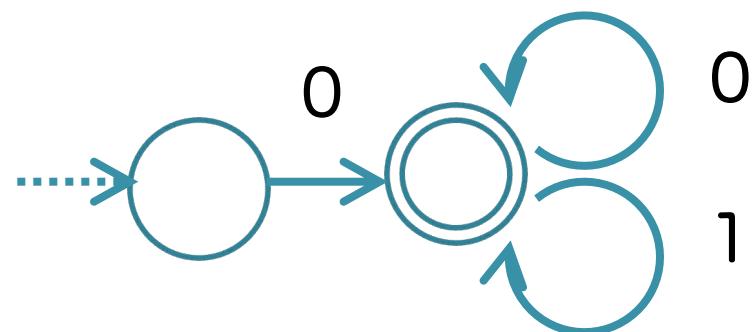


正規言語 (regular language)

- 正規表現は言語（文字列の集合）を表現する
 - 正規表現 r が表現する言語を正規言語といい, $L(r)$ と書く
- 正規言語の定義
 - $L(\varepsilon) = \{\varepsilon\}$. $a \in \Sigma$ の時, $L(a) = \{a\}$
 - r と s が正規表現の時,
 - $L(r|s) = L(r) \cup L(s)$ 和集合 (選択)
 - $L(r \cdot s) = \{x \cdot y \mid x \in L(r) \wedge y \in L(s)\}$ 連接
 - $L(r^*) = L(\varepsilon) \cup L(r) \cup L(r \cdot r) \cup L(r \cdot r \cdot r) \cup \dots$ 閉包 (0回以上の繰り返し)
 - $L((r)) = L(r)$ カッコ
- 例：
 - $L(0 \cdot (0|1)^*) = \{0, 00, 01, 000, 001, 010, 011, 0000, \dots\}$
 - 気持ち：0で始まり, その後に0か1が0回以上続く文字列

有限オートマトン (1/3)

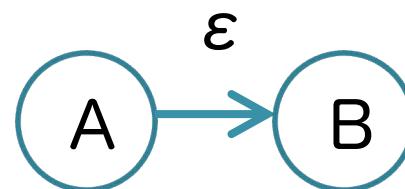
- 有限オートマトン(finite automaton)=仮想計算機の一種。有限個の状態を持ち、入力文字に従って状態を遷移して動作する。
- 状態遷移図(transition diagram)=有限オートマトンの図示
 - 例：AとBは状態(state)，矢印は入力文字に対する状態遷移(state transition)
 - 例：状態Aで入力文字0を受け取ると状態Bに移動
 - 2つの特別な状態：初期状態(initial state)と終状態(final state)
 - 初期状態=点線矢印の状態（1つ），終状態=2重丸の状態（複数）
 - 有限オートマトンの動作は初期状態から始まる





有限オートマトン (2/3)

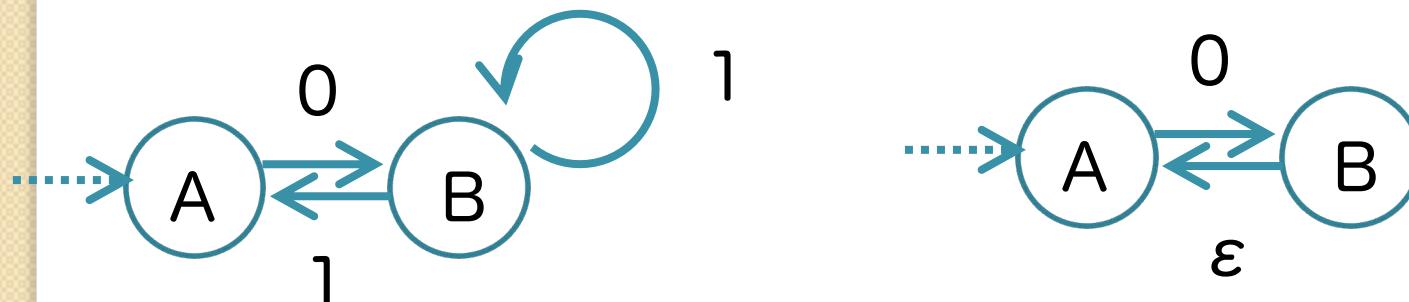
- 入力文字列 s に対する遷移先が終状態になる時、「有限オートマトンは文字列 s を受理(accept)する」という
 - 例：さっきの図の状態遷移図は入力文字列0010を受理する
 - 有限オートマトン M が受理する文字列集合を $L(M)$ と書く
- ϵ 遷移=入力文字無しでの状態遷移. 遷移しなくても良い.





有限オートマトン (3/3)

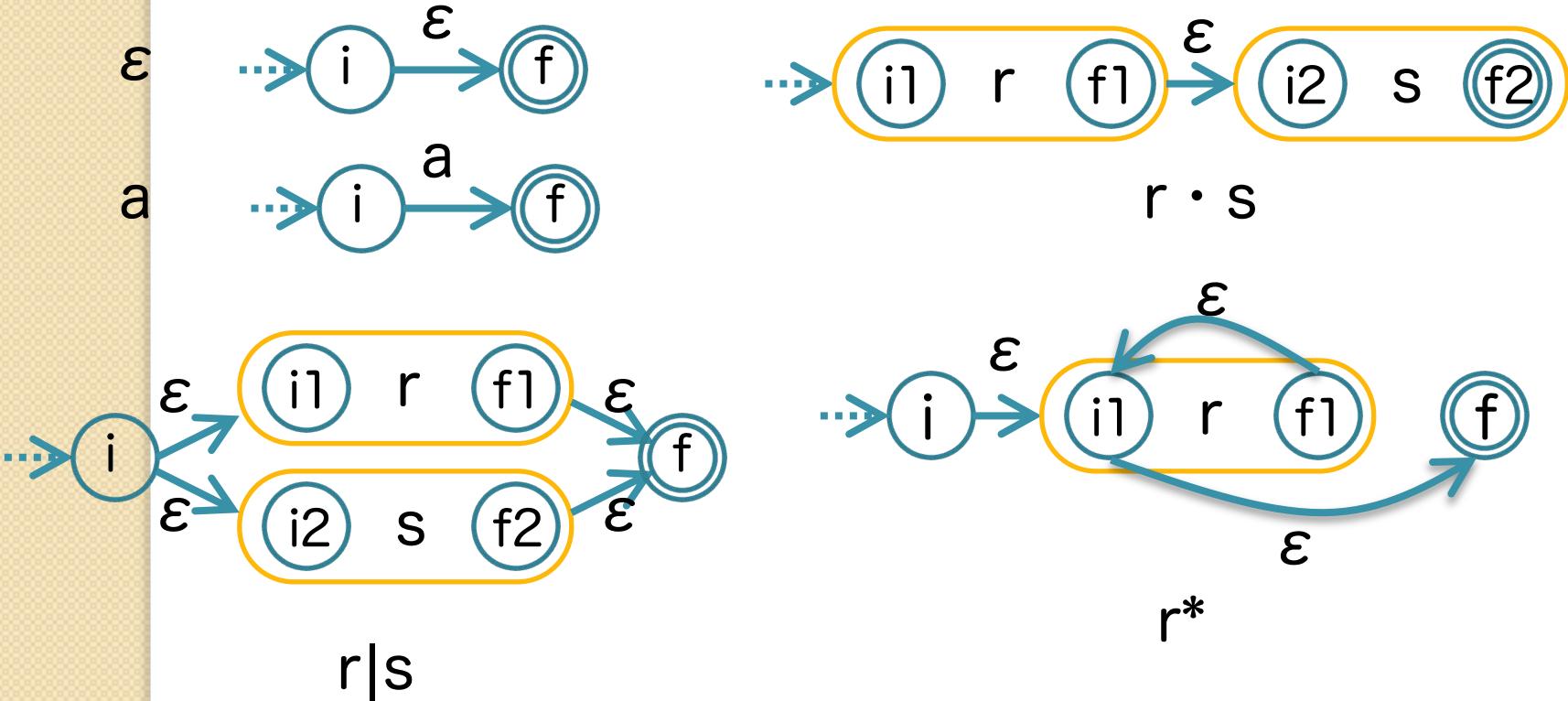
- 決定性有限オートマトン(DFA)
 - 入力文字列に対して遷移先の状態が一意に決まる有限オートマトン
 - 条件1： ϵ 遷移が無い
 - 条件2：各状態から同じ入力記号に対する遷移が複数無い
- 非決定性有限オートマトン(NFA)
 - DFAではない有限オートマトン。遷移先の状態が一意に決まらない。
 - 例1（左図）：入力文字列01に対して、AかBの状態にいる
 - 例2（右図）：入力文字列0に対して、AかBの状態にいる





正規表現からNFAへの変換

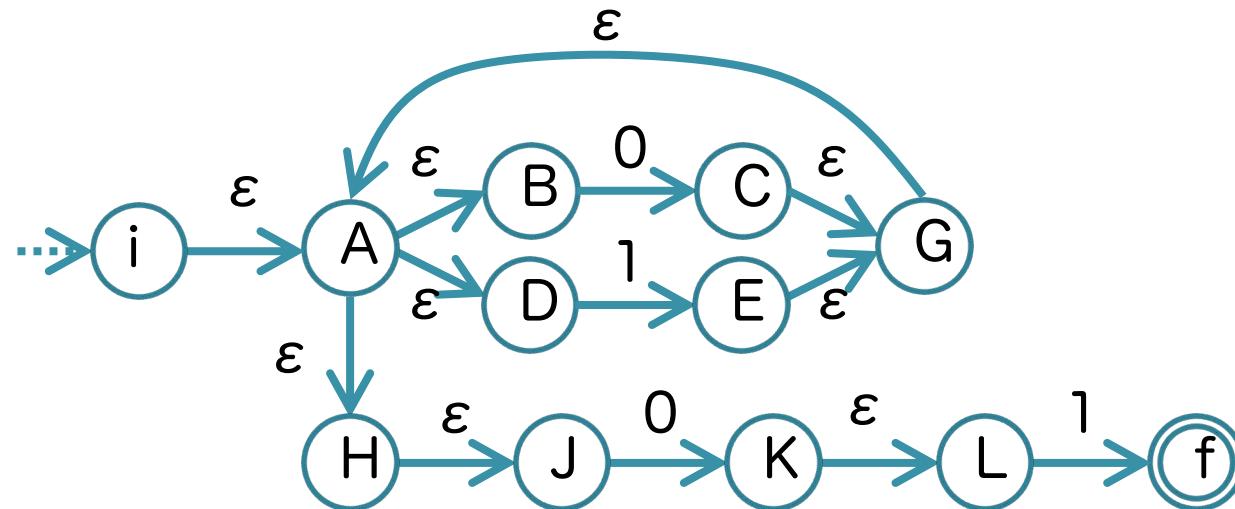
- $L(r)=L(M)$ の時、正規表現 r と有限オートマトン M は等価という
- 正規表現は機械的に等価な有限オートマトンに変換できる
- 正規表現からNFAへの変換 ($a \in \Sigma$, r と s は正規表現)





正規表現からNFAへの変換例

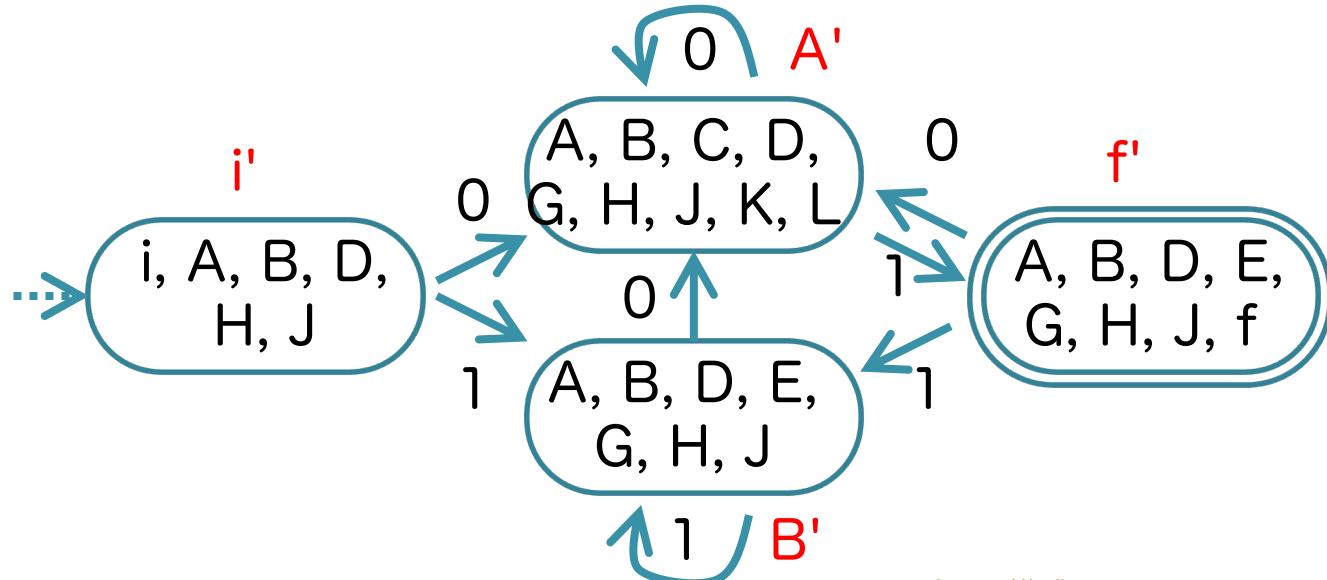
- 正規表現 $(0|1)^*01$ と等価なNFA





NFAからDFAへの変換 (1/2)

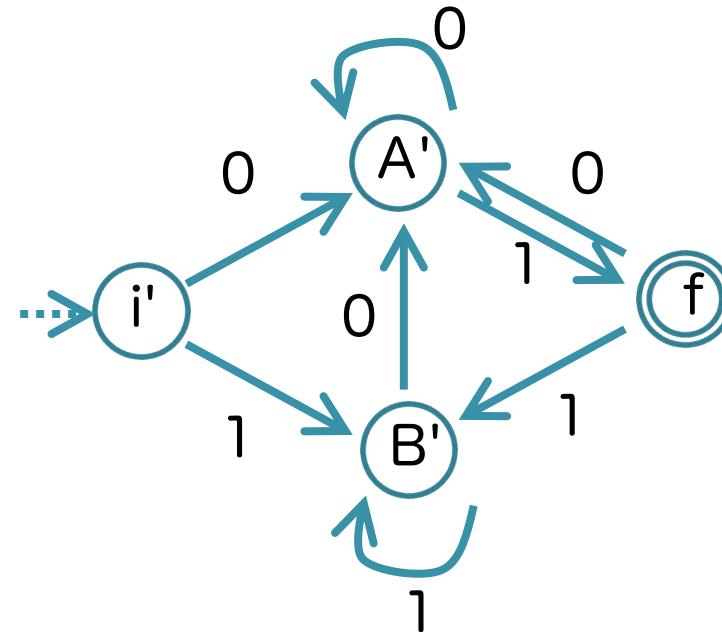
- NFAは機械的に等価なDFAに変換できる
- 方法 (直感的な説明)
 - 状態Aから同じ入力で複数の状態B₁, B₂, …, B_nに遷移可能な時、複数の状態B₁, B₂, …, B_nを1つにまとめてBとする
 - 上を初期状態から始めて、すべての遷移に対して繰り返す
- 例：前ページのNFAと等価なDFA





NFAからDFAへの変換 (2/2)

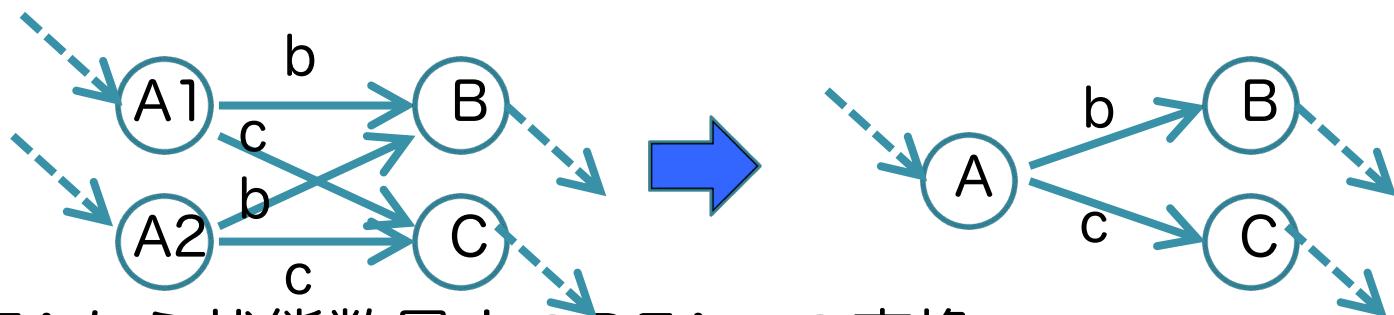
- 状態名を付け替えると以下になる





DFAの状態数の最小化

- DFAは機械的に等価で状態数が最小のDFAに変換可能
- アイデア：状態A1とA2が、すべての入力文字に対して同じ状態に遷移するなら、1つの状態Aにしてよい

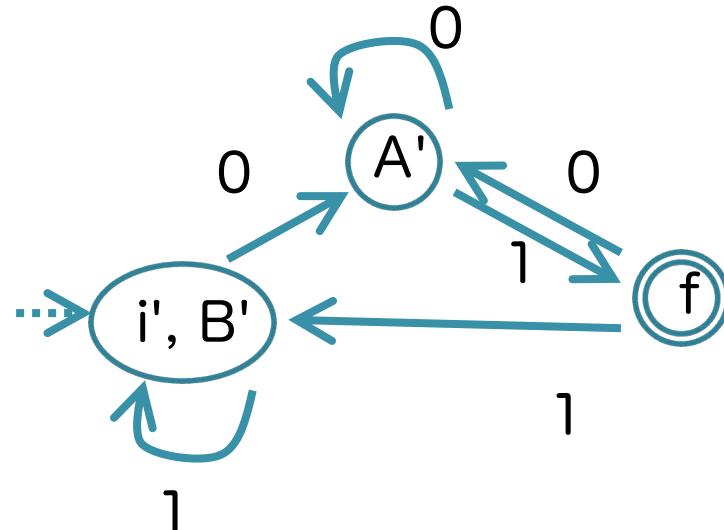


- DFAから状態数最小のDFAへの変換
 - 状態を2つに分割：終了状態とそれ以外（最低同じにできない）
 - 同グループに次の状態AとBがあれば分割
 - ある入力文字aに対して、Aは遷移するがBは遷移しない
 - ある入力文字aに対して、AとBは異なるグループに遷移する
 - 分割できなくなるまで上を繰り返す



DFAの状態数の最小化（例）

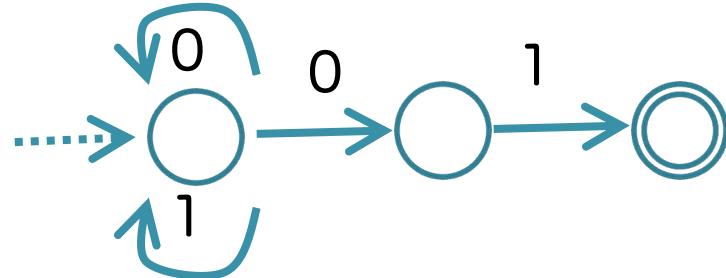
- 前の前のページのDFAの最小化
 - 状態A'はi'やB'とは異なる遷移→別グループに分割
 - 状態i'とB'は同じグループに遷移→分割終了。i'とB'を1つに



正規表現 $(0|1)^*01$



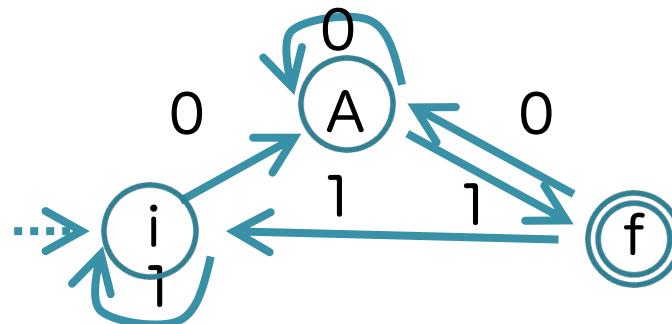
これは？



- NFAなので、これを最小DFAに変換すると前ページと同じになる
- 実は、ある正規言語を受理する最小DFAは唯一つ
 - 異なる正規表現の言語が同じかどうかは、最小DFAが同じかどうかを調べれば良い

DFAからプログラムへの変換

- DFAは等価なプログラムに変換可能
- 単純な方法：状態遷移図を状態遷移表に書き替える



	入力0	入力1
状態i (0)	状態A (1)	状態i (0)
状態A (1)	状態A (1)	状態f (2)
状態f (2)	状態A (1)	状態i (0)

- 状態遷移は配列アクセスでプログラムできる
 - `state = table [state][input];`
- この方法は素朴(naive)なので表が巨大化。
 - ほとんどの要素が空だから.
 - 効率良い表作成方法はある。ここでは省略。
 - 一般的には（表ではなく）リストやハッシュを使う。



プログラム例

```
#include <stdio.h>
#include <cassert.h>
int main (void)
{
    int table [3][2] = {{1, 0}, {1, 2}, {1, 0}};
    int c, state = 0;
    while ((c = getchar ()) != '\n') {
        state = table [state][c - '0'];
        if (state == -1)
            assert (0);
    }
    if (state == 2)
        printf ("accepted\n");
    else
        printf ("not accepted\n");
}
```



実行例

```
% gcc foo.c
% ./a.out
01
accepted
% ./a.out
10
not accepted
% ./a.out
11010
not accepted
% ./a.out
110101
accepted
%
```



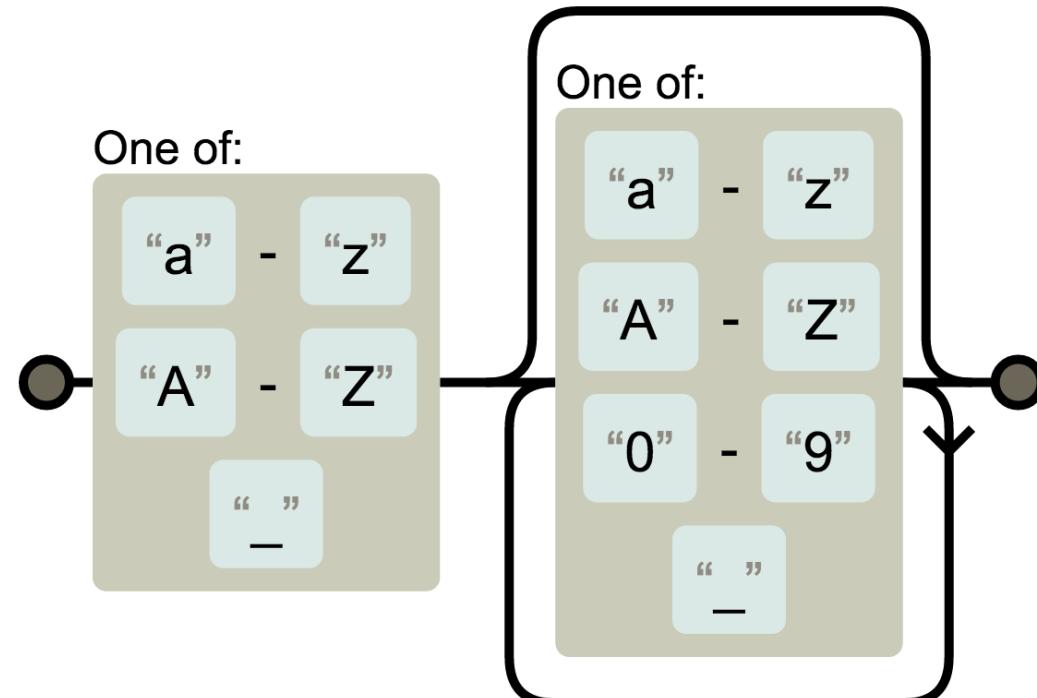
例題：簡易言語 XC

- この授業で例題とするプログラミング言語
- C言語のサブセット
- XCの字句要素
 - 予約語 : char, else, goto, if, int, long, return, void, while
 - 識別子 : [a-zA-Z_][a-zA-Z0-9_]*
 - 10進整数リテラル : 0|[1-9][0-9]*
 - 文字リテラル : '(\n | \\' | \\\\" | [^\\'])'
 - 文字列リテラル : " (\n | \\" | \\\\" | [^\\"])* "
 - 演算子, 区切り子 : ==, &&, ||, ;, :, {, }, ',', =, (,), &, !, -, +, *, /, <
- XCの字句解析器を以降で作る



レイルロード・ダイアグラム (1/3)

- 文脈自由文法や正規表現を図表示する方法.
- 例：識別子：[a-zA-Z_][a-zA-Z0-9_]*

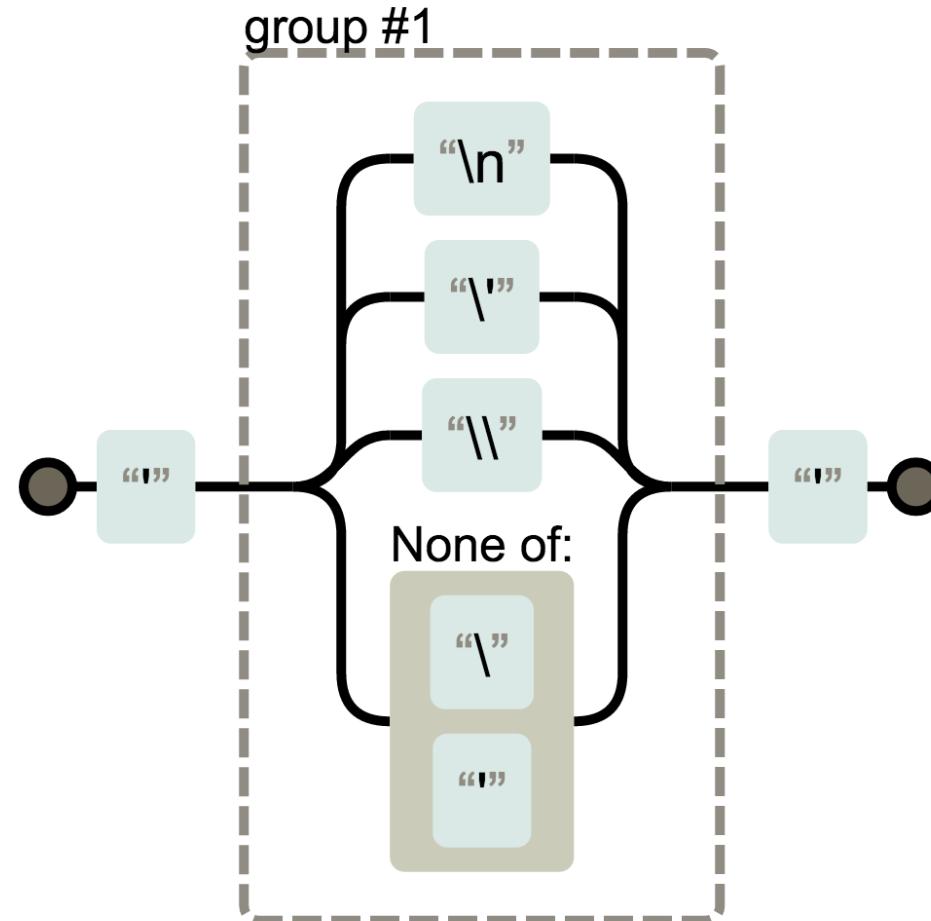


generated with <http://regexper.com/>



レイルロード・ダイアグラム (2/3)

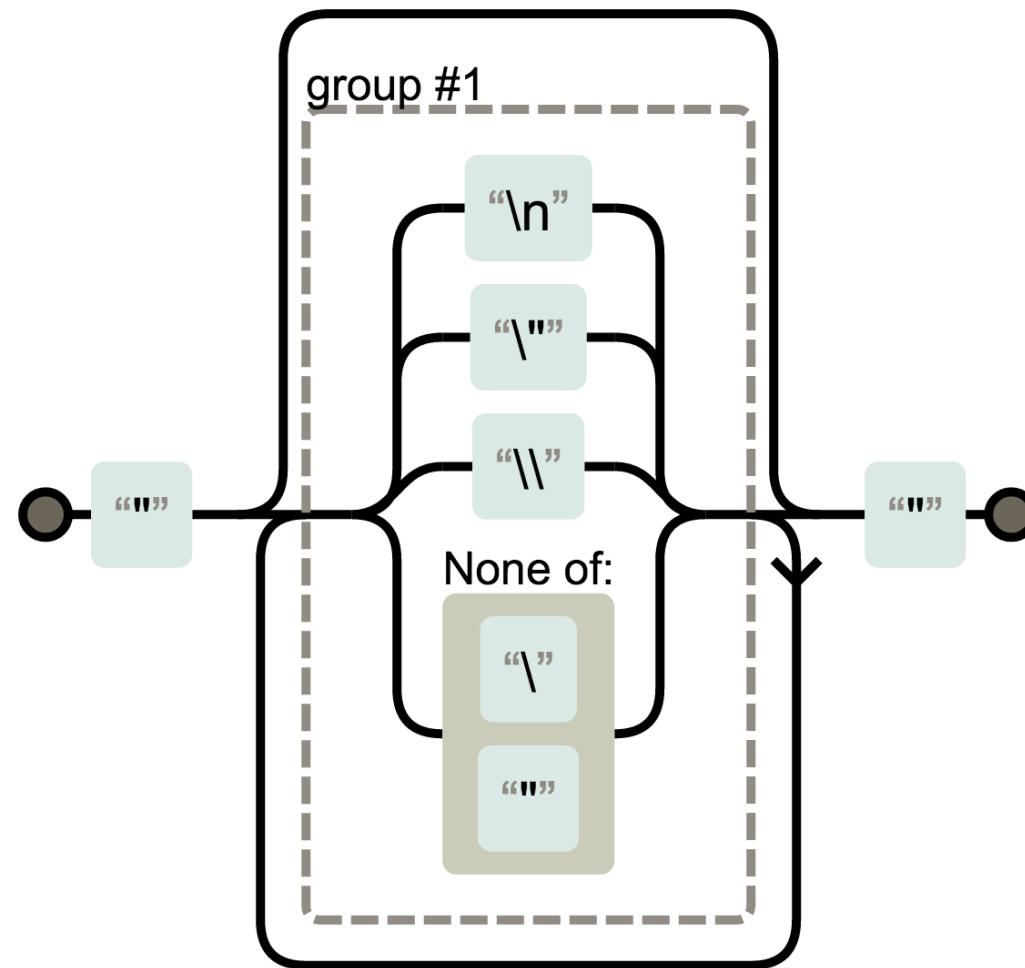
- 例：文字リテラル：'(\n | \\' | \\\\" | [^\\'])'





レイルロード・ダイアグラム (3/3)

- 例：文字列リテラル：" (\n | \" | \\\\ | [^\\"])* "





malloc でファイルを全部メモリに (1/3)

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

off_t getfilesize (int fd)
{
    struct stat sbuf;
    fstat (fd, &sbuf);
    return sbuf.st_size; // ファイルサイズを返す
}
```

cat-malloc.c (1/2)



malloc でファイルを全部メモリに (2/3)

```
int main (int argc, char *argv[])
{
    FILE *fp; int i; off_t file_size; char *p;
    fp = fopen (argv [1], "r");
    file_size = getfilesize (fileno (fp));
    p = malloc (file_size);
    for (i = 0; i < file_size; i++) {
        p [i] = getc (fp);
    }
    for (i = 0; i < file_size; i++) {
        putchar (p [i]);
    }
    fclose (fp);
}
```

簡潔のためエラー処理は省略

cat-malloc.c (1/2)

malloc でファイルを全部メモリに (3/3)

```
% gcc cat-malloc.c
% ./a.out cat-malloc.c
#include <stdio.h>
#include <stdlib.h>
... (ファイルの内容が出力)
```

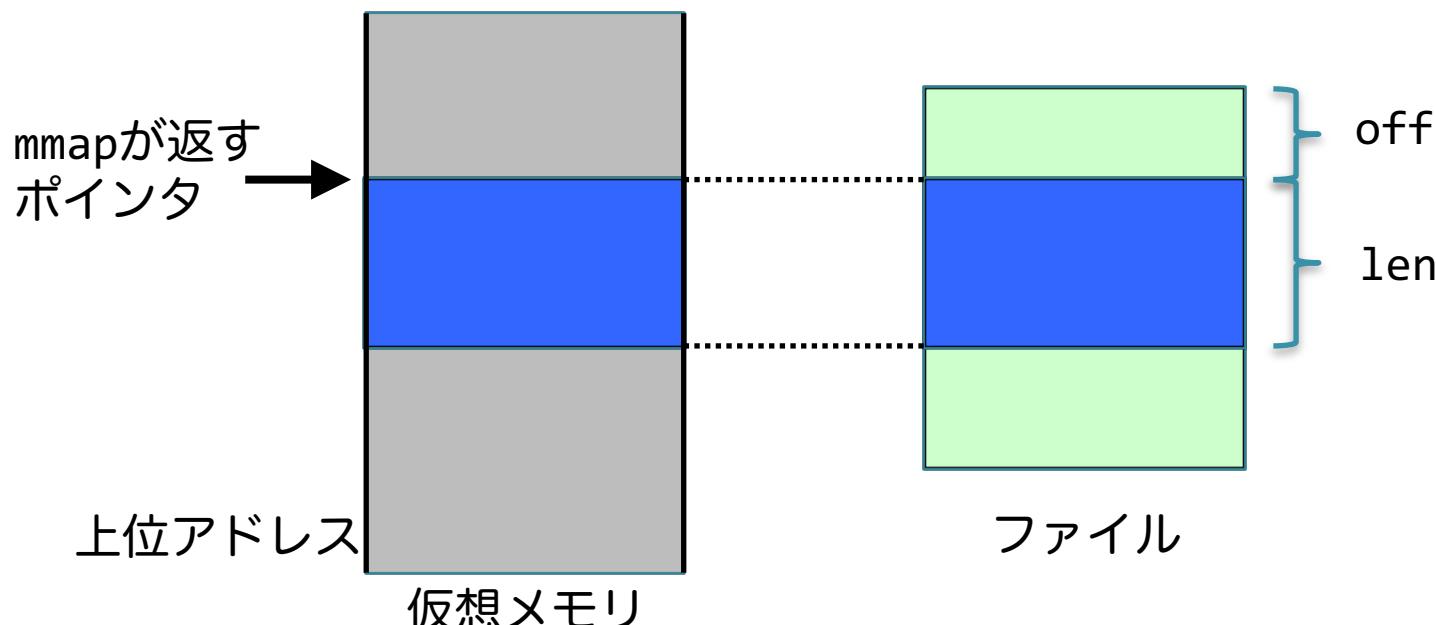
- ファイルを全部メモリに読み込みます, `getc/putc` で一文字ずつ読んだ方が効率は良い.
- でも1つの字句が何文字かは読んでみないと不明.
→ 全部メモリに読んでから処理の方が簡単.
- `malloc` でもできるが, 普通は `mmap` を使う.
 - ソースコード1つのサイズは通常は巨大にならない
→ どちらでもOK



mmap

```
#include <sys/mman.h>
void *mmap (void *addr, size_t len, int prot,
            int flags, int fildes, off_t off);
int munmap (void *addr, size_t len);
```

- **mmap**はファイルをメモリにマップ（対応づけ）し、プログラマがファイルをメモリとしてアクセス可能にする
 - 例： `putc ('a', fp);` ではなく， `*p = 'a';` と書ける.
 - マップ解除には `munmap` を使う.





mmapでファイル処理 (1/2)

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
char *map_file (char *filename)
{
    struct stat sbuf; char *ptr;
    int fd = open (filename, O_RDWR);
    fstat (fd, &sbuf);
    ptr = mmap (NULL, sbuf.st_size + 1, // +1 for '\0'
                PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
    ptr [sbuf.st_size] = '\0';
    return ptr;
}
```

MAP_PRIVATE ファイルはコピーして変更
MAP_SHARED ファイルそのものを変更
MAP_ANON メモリを確保

ファイル中に \0 はない前提

cat-mmap.c (1/2)



mmapでファイル処理 (2/2)

```
int main (int argc, char *argv[])
{
    char *ptr, c;
    ptr = map_file (argv [1]);
    while ((c = *ptr++) != '\0') {
        putchar (c);
    }
}
```

cat-mmap.c (2/2)

- mmapを使うと、ファイル操作ではなく、
メモリ操作としてファイルにアクセスできる



regex.h (1/3)

- POSIX標準の文字列検索ライブラリ（置換は無し）
- 使い方
 - `regcomp` で正規表現をコンパイル（前処理）する
 - `regcomp` は、コンパイル結果を `regex_t` 構造体に格納
 - `regexec` で文字列検索を実行する（複数回可）
 - `regexec` は、マッチした位置情報を `regmatch_t` 構造体に格納
 - `.rm_so` にマッチ開始位置, `.rm_eo` にマッチ終了位置
 - 使い終わったパターンは `regfree` で解放（OSに返却）

```
#include <regex.h>
int regcomp (regex_t *preg, char *pattern, int cflags);
int regexec (regex_t *preg, char *string, size_t nmatch,
             regmatch_t pmatch[], int eflags);
void regfree (regex_t *preg);
```



regex.h (2/3) : 使用例

```
#include <stdio.h>
#include <stdlib.h>
#include <regex.h>
int main (void) {
    char    pattern [] = "^\"(\\\\\\\\\"|[^\\""])*\"";
    regex_t regex; regmatch_t regmatch;
    int ret; char buf [BUFSIZ];
    ret = regcomp (&regex, pattern, REG_EXTENDED);
    if (ret != 0) {
        regerror (ret, &regex, buf, sizeof (buf));
        fprintf (stderr, "regcomp: %s\n", buf);
        exit (1);
    }
    fgets (buf, sizeof (buf), stdin);
    ret = regexec (&regex, buf, 1, &regmatch, 0);
    if (ret == 0) { /* matched */
        buf [regmatch.rm_eo] = '\0';
        printf ("|%s|\n", buf);
    }
}
```

regcompに渡す時は、`^"(\\\"|[^\"])*"`



regex.h (3/3) : 実行例

```
% gcc -g foo.c
% ./a.out
"abc"
| "abc" | マッチした
% ./a.out
"abc\""
| "abc\"\" | \"があってもマッチした
% ./a.out
"abc\n"
\nがあるとマッチしない
% ./a.out
"abc" 行頭から始まってないのでマッチしない
%
```



エスケープの補足

- C言語の文字列リテラル中
 - \ を表すのに \\ と書く (エスケープ必要)
 - " を表すのに \" と書く (エスケープ必要)
- regcompに渡した後
 - " を表すのに " と書く (エスケープ不要)
 - [...]の外では, \ を表すのに \\ と書く (エスケープ必要)
 - [...]の中では, \ を表すのに \ と書く (エスケープ不要)

とてもわかりにくい。正規表現のエスケープ処理は難しい。
正規表現を使う時は、マニュアルを確認＆要十分なテスト。



XC字句解析器の方針

- ファイル処理には mmap を使用
- 文字列検索には regex.h を使用（手抜き）
 - ただし、コメントと空白文字類の読み飛ばしは手書きで
- 解析結果は以下の構造体の配列に格納

```
struct token {  
    int kind;          // トークンの種類(enum token_kind)  
    int offset_begin; // ファイル中のトークン開始位置  
    int offset_end;   // ファイル中のトークン終了位置  
    char *lexeme;     // トークン文字列  
};
```



enum token_kind

```
enum token_kind {
    TK_UNUSED      = 0,   TK_ID           = 1,   TK_INT          = 2,
    TK_CHAR         = 3,   TK_STRING        = 4,   TK_KW_CHAR       = 5,
    TK_KW_ELSE      = 6,   TK_KW_GOTO       = 7,   TK_KW_IF         = 8,
    TK_KW_INT       = 9,   TK_KW_RETURN     = 10,  TK_KW_VOID       = 11,
    TK_KW_WHILE     = 12,  TK_OP_EQ        = 13,  TK_OP_AND        = 14,
    TK_OP_OR        = 15,  TK_KW_LONG      = 16,
    // 以下は名前を付けずにそのまま使う
    // ';' ':' '{' '}' ',', '=' '(' ')' '&'
    // '!' '-' '+' '*' '/' '<'
};
```

enum は明示的に値を指定する必要はないが、
ここでは（デバッグを簡単にするため）明示的に指定



token_kind 文字列変換用配列

```
char *token_kind_string[] = {
    "UNUSED", "ID", "INT", "CHAR", "STRING",
    "char", "else", "goto", "if", "int",
    "return", "void", "while", "==", "&&", "||", "long",
    [';'] = ";", [':'] = ":" , ['{'] = "{", ['}'] = "}",
    [,'] = ",", [='] = "=" , ['('] = "(", [')'] = ")",
    [&'] = "&", ['!'] = "!", [-'] = "-", ['+'] = "+",
    [*'] = "*", ['/'] = "/", ['<'] = "<",
};
```

- `[';'] = ";"` は「配列 ';' 番目を ";" で初期化」の意味
- ';' の ASCII コードは 10 進数で 59 →
`[';'] = ";"` は、 `token_kind_string[59] = ";"` と同じ



xcc-lex.c

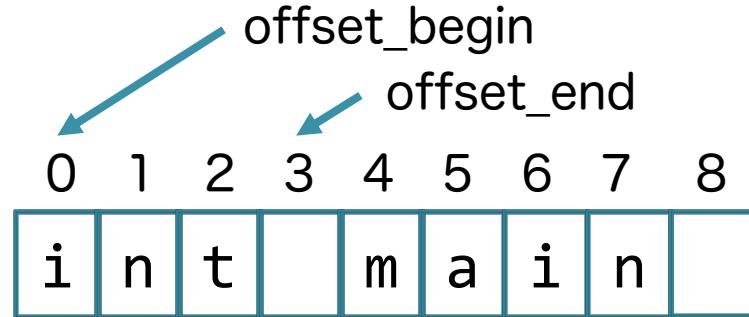
- 権藤が書いた XCC字句解析器
 - バグを見つけたら教えて下さい
- 演習課題1はこれをベースに使う
 - もちろん、字句解析器も自作しても良い



xcc-lex.c 実行例

```
% gcc -g xcc-lex.c
% cat test1/t2.c
int main ()
{
    int x;
    goto foo;
foo:
    return x;
}
% ./a.out test1/t2.c
0: 0-3: int (int)
1: 3-4: (WHITESPACE)
2: 4-8: main (ID)
3: 8-9: (WHITESPACE)
4: 9-10: ( ((()
5: 10-11: ) (()
```

以下略



0番目のトークンは0~3バイト目で
lexemeはint, kindもint





GNU flex

- 字句解析生成系の一つ
- 演習課題2ではGNU flexが生成した字句解析器を使う

```
/* integer literals (decimal only, no suffix) */
0|[1-9][0-9]*           { return INTEGER_CONSTANT; }

/* character literals */
'(\n|\'|\\\\|[^\\'])' { return
CHARACTER_CONSTANT; }
```

flexでは、 [...]中、 \-[]^は要エスケープ

xcc.l の一部