

コンパイラ及び演習

関澤 俊弦

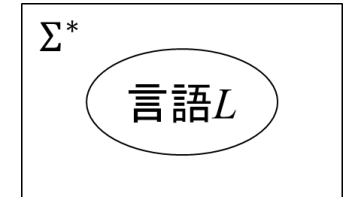
日本大学 工学部 情報工学科

復習

■ 言語処理系

■ 形式言語

- アルファベット, 記号列, スター閉包, Σ 上の言語



■ 形式文法

- 終端記号, 非終端記号, 書き換え規則

- 文法で生成される言語

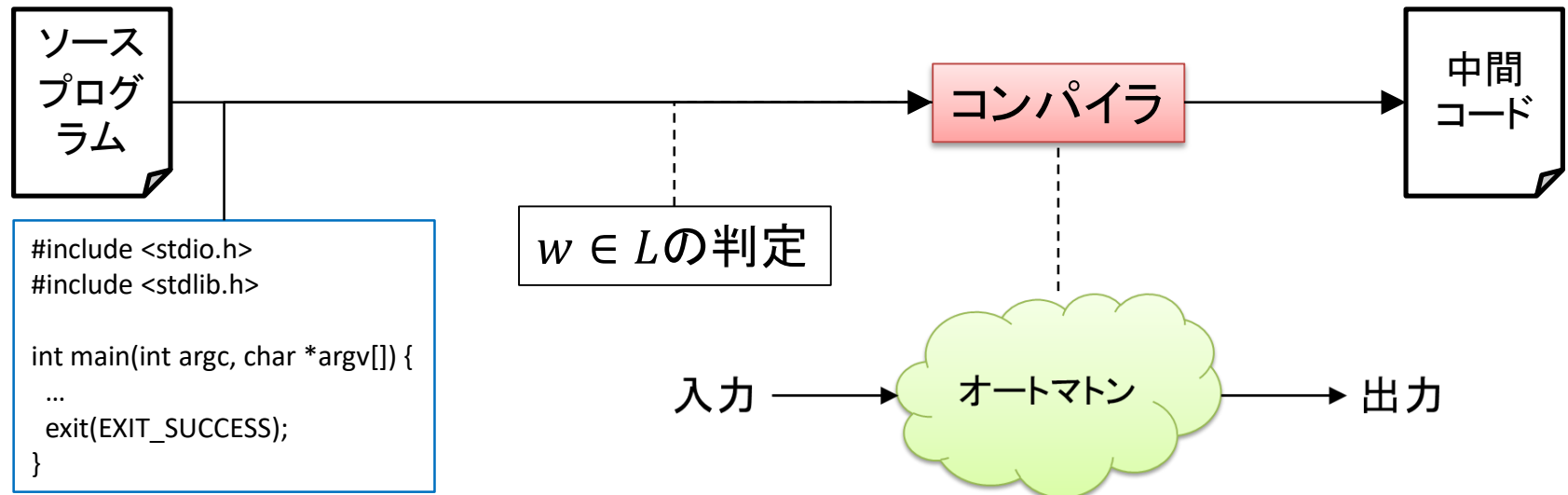
■ オートマトン

- 有限オートマトン, 正規言語

復習

■ これまでの実装(演習)

- コマンドラインからのファイル名の指定
- 語 $w \in L$ の判定
- 関数 nextChar
- 有限オートマトンの実装(状態と遷移)



- 形式文法
 - 正規文法
- BNF記法
- 字句解析
 - 字句, 字句解析の役割, 字句の読み取り
- プログラミング技報

- 形式文法
 - 正規文法
- BNF記法
- 字句解析
 - 字句, 字句解析の役割, 字句の読み取り
- プログラミング技報

言語を扱う方法

■ 「言語」を数学的に扱う方法

 p.115

□ 状態機械

- 状態で記憶し計算する機械(オートマトン)で扱う
- 言語は, 初期状態 q_0 を受理状態の一つに導く入力記号列の集合

$$L(M) = \{w \mid w \in \Sigma^*, (q_0, w) \vdash_M^* (q_n, \varepsilon), q_n \in F\}$$

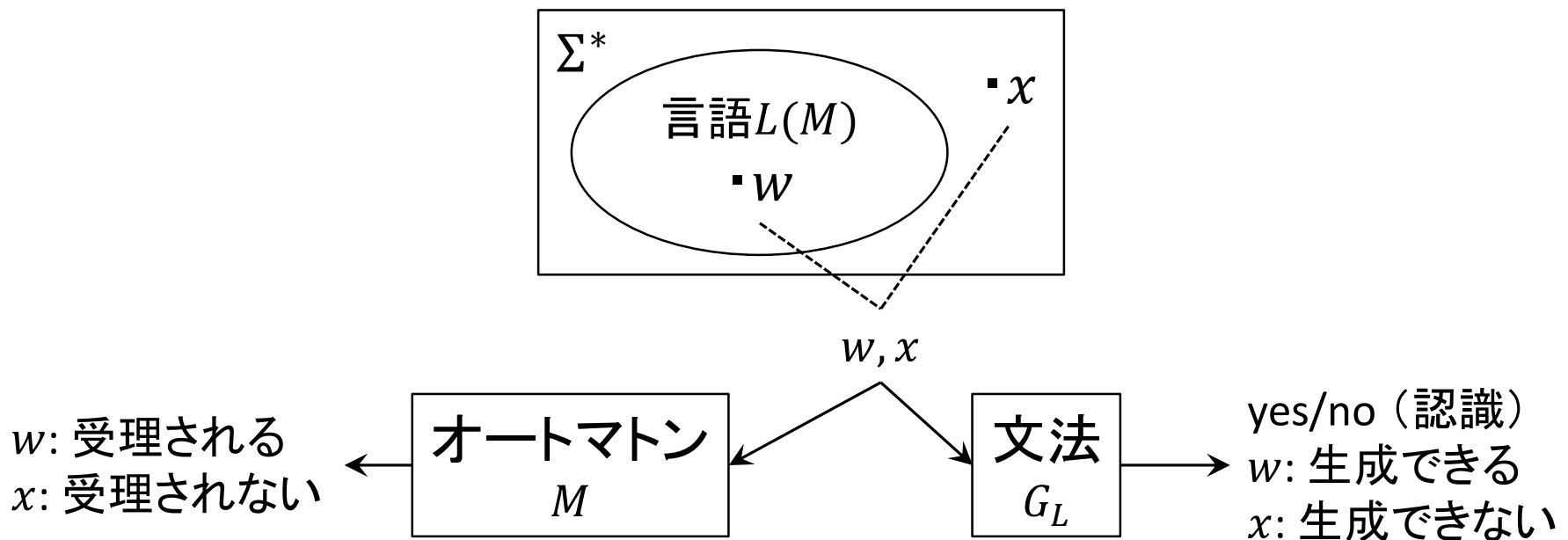
□ 形式文法

- 言語の文法を数学的に扱う

言語の生成装置としての形式文法

■ 文法を利用すると

- 文法の規則に従って文を生成できる
- 与えられた語が言語に属するか判定できる



(再掲)形式文法

■ 形式文法とは

- 形式的に定められた文法
- 語に関して基準を定め、基準を満たす語と満たさない語を識別するもの

■ 例

- 基準: 英文法 (英語を識別する)
 - Betty is a pretty girl.
- 基準: C言語の文法
 - $(a + b) * c$



紛れがない場合、形式文法を「文法」と呼ぶ

文の生成に関わる用語

■ 終端記号

- 文法により生成される文の構成要素

■ 非終端記号（構文変数）

- 文の生成時に使用される変数
 - 終端記号の集合と非終端記号の集合は共通部分をもたない

■ 初期記号（開始記号）

■ 書き換え規則（生成規則）



「記号」は「文字」「アルファベット」と呼ばれることもある

形式文法

■ 形式文法 $G = (N, \Sigma, P, S)$

 p.123

- N : 非終端記号の有限集合
- Σ : 終端記号の有限集合
- $P \subseteq V^* N V^* \times V^*$: 書き換え規則の有限集合
ここで, すべての記号の集合 $V = (N \cup \Sigma)$
- $S \in N$: 初期記号(開始記号)

補足: 書き換え規則 (1)

■ 書き換え規則 $p \in P$ の表記

- 正式には (α, β) . ただし, $\alpha \in V^*NV^*, \beta \in V^*$
- 通常, $p: \alpha \rightarrow \beta$ と書く
 - 「書き換え規則 p は, α を β に書き換える」
 - α を左辺, β を右辺という

■ 用語

- 終端規則
 - 右辺に非終端記号を持たない規則
- 非終端規則
 - 右辺に終端記号を持つ規則

補足: 書き換え規則 (2)

■ 適用可能(性)

- 記号列 ξ が非終端記号 α を含んでいる, すなわち, $\xi = \xi_1 \alpha \xi_2$ とする. このとき, 書き換え規則 $p: \alpha \rightarrow \beta$ は ξ に適用可能であると言う.

■ 書き換え規則の適用

- 記号列 $\xi = \xi_1 \alpha \xi_2$ に, 書き換え規則 $p: \alpha \rightarrow \beta$ を適用した結果 $\eta = \xi_1 \beta \xi_2$ である.

$$\begin{array}{c} \xi_1 \alpha \xi_2 \\ \Downarrow p: \alpha \rightarrow \beta \\ \xi_1 \beta \xi_2 \end{array}$$

補足: 書き換え規則 (3)

■ 書き換え規則 $\alpha \rightarrow \beta$

- 左辺 α が非終端記号を1つ以上含んでいれば, α を β に書き換えられる. このとき, α に含まれる終端記号も書き換えられる.

- 「終端記号」の意味を考えると不自然



■ 書き換え規則に関する制約

- 書き換えられるのは非終端記号のみ.
i.e., 終端記号は書き換えられない

用語と表記

■ 導出

- 書き換え規則の適用により, 語を生成すること
- 書き換え規則1回の適用を, 1回の導出という

■ 導出の記法

- 1回の導出を \Rightarrow と表わす

- 語 ξ を η に書き換えたとき, $\xi \Rightarrow \eta$ と表わす
- 適用した文法を明示する場合 $\xi \Rightarrow_G \eta$ と書く
- 適用した書き換え規則を明示する場合 $\xi \Rightarrow^p \eta$ と書く

- 0回以上の導出を \Rightarrow^* と表わす

- 例: $\xi \Rightarrow^* \eta$

語の導出, 文法で生成される言語

■ 文法 G で生成される語

- 初期記号から0回以上の導出によって得られた終端記号のみで構成された記号列
- 初期記号 S に対して, $S \Rightarrow^* w, w \in \Sigma^*$

■ 中間語

- $S \Rightarrow^* \xi \Rightarrow^* w$ に現われる語 $\xi \in (N \cup \Sigma)^*$

■ 文法 G で生成される言語

- 文法 G で生成される語の集合

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

正規文法 (RG: regular grammar)

■ $G = (N, \Sigma, P, S)$

□ N : 非終端記号の有限集合

□ Σ : 終端記号の有限集合

□ P : 書き換え規則の有限集合

- ただし, 書き換え規則が次の形式となっているもの
 $A \rightarrow aB$. あるいは, $A \rightarrow a$. ここで, $A, B \in N, a \in \Sigma$ であり, 例外として $S \rightarrow \varepsilon$ も許されるとする.

□ $S \in N$: 初期記号

例: 正規文法で生成される言語

■ $G = (N, \Sigma, P, S)$

- $N = \{S_0, S_1, S_2\}$

- $\Sigma = \{a\}$

- $P = \{S_0 \rightarrow aS_1, S_1 \rightarrow aS_2, S_2 \rightarrow aS_0, S_2 \rightarrow a\}$

- $S = \{S_0\}$

■ 語の導出

- $S \Rightarrow aS_1 \Rightarrow aaS_2 (= a^2 S_2) \Rightarrow aaa (= a^3)$

- $S \Rightarrow aS_1 \Rightarrow aaS_2 \Rightarrow aaaS_0 \Rightarrow^* aa$

■ G で生成される言語 $L(G)$

$$L(G) = \{a^{3n} \mid n \geq 1\}$$

例: 正規文法で生成される言語

■ 語の導出

$$\square S \Rightarrow aS_1 \Rightarrow aaS_2 (= a^2S_2) \Rightarrow aaa (= a^3)$$

$$\square S \Rightarrow aS_1 \Rightarrow a^2S_2 \Rightarrow a^3S_0 \Rightarrow^* a^6$$

□ ...

■ G で生成される言語 $L(G)$

$$L(G) = \{a^{3n} \mid n \geq 1\}$$

言語の扱い

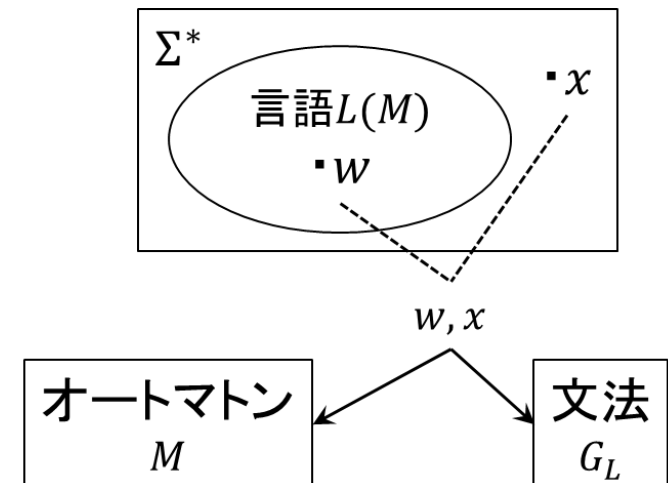
言語 L は語 w の集合として定義される

■ 状態機械 (オートマトン)

$$\square L(M) = \{w \mid w \in \Sigma^*, (q_0, w) \vdash_M^* (q_n, \varepsilon), q_n \in F\}$$

■ 形式文法

$$\square L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$



- 形式文法
 - 正規文法
- BNF記法
- 字句解析
 - 字句, 字句解析の役割, 字句の読み取り
- プログラミング技報

BNF記法

■ 通常の文字のみで文法を表現する記法

□ BNF: Backus-Naur Form, Backus-Normal Form

■ 構文規則

□ 非終端記号 ::= 定義₁ | 定義₂ | ... | 定義_n

- $A ::= B$ 左側の構文要素は, 右側で定義される
- CD C の後に D が続く
- $E | F$ ‘|’は「または」を意味する. E または F
- $\langle G \rangle$ 非終端記号
(付いていないときは終端記号)



::= は \rightarrow で表わされることもある.

非終端記号の $\langle \rangle$ は省略されることもある.

拡張BNF記法

■ 要素の繰り返しや省略などの表現を追加したBNF記法

- {} {}内の要素の0回以上の繰り返し
 - 1回以上の繰り返しを+, 0回以上の繰り返しを*で表わすこともある
- [] []内の要素は0または1個



BNF記法は、用途により様々な拡張があります

例: 拡張BNF記法 - 符号なし整数

$\langle \text{NonZeroDigit} \rangle$	$::=$	$1 2 3 4 5 6 7 8 9$
$\langle \text{Digit} \rangle$	$::=$	$0 \langle \text{NonZeroDigit} \rangle$
$\langle \text{Integer} \rangle$	$::=$	$\langle \text{Digit} \rangle \langle \text{NonZeroDigit} \rangle\langle \text{Integer} \rangle^*$



$\langle \text{Integer} \rangle$ は再帰的に定義されている点に注意

■ 例: 整数206の導出

$\langle \text{Integer} \rangle$	$::=$	$\langle \text{NonZeroDigit} \rangle\langle \text{Integer} \rangle^*$
	$::=$	$2 \quad \langle \text{Digit} \rangle\langle \text{Integer} \rangle^*$
	$::=$	$2 \quad 0 \quad \langle \text{Digit} \rangle$
	$::=$	$2 \quad 0 \quad 6$

例: 拡張BNF記法 - 符号なし整数

$\langle \text{NonZeroDigit} \rangle$	$::=$	$1 2 3 4 5 6 7 8 9$
$\langle \text{Digit} \rangle$	$::=$	$0 \langle \text{NonZeroDigit} \rangle$
$\langle \text{Integer} \rangle$	$::=$	$\langle \text{Digit} \rangle \langle \text{NonZeroDigit} \rangle\langle \text{Integer} \rangle^*$

■ 整数の導出

□ 1桁の整数. $\text{Integer} ::= \text{Digit}$

- $\text{Integer} ::= 0, 1, 2, \dots, 9$

□ 2桁の整数. $\text{Integer} ::= \text{NonZeroDigit Integer}$

- $\text{Integer} ::= (1, 2, \dots, 9)(0, 1, \dots, 9)$
 $::= 10, 11, \dots, 19, 20, 21, \dots, 29, \dots,$
 $90, 91, \dots, 99$

□ 3桁の整数 (以下同様)

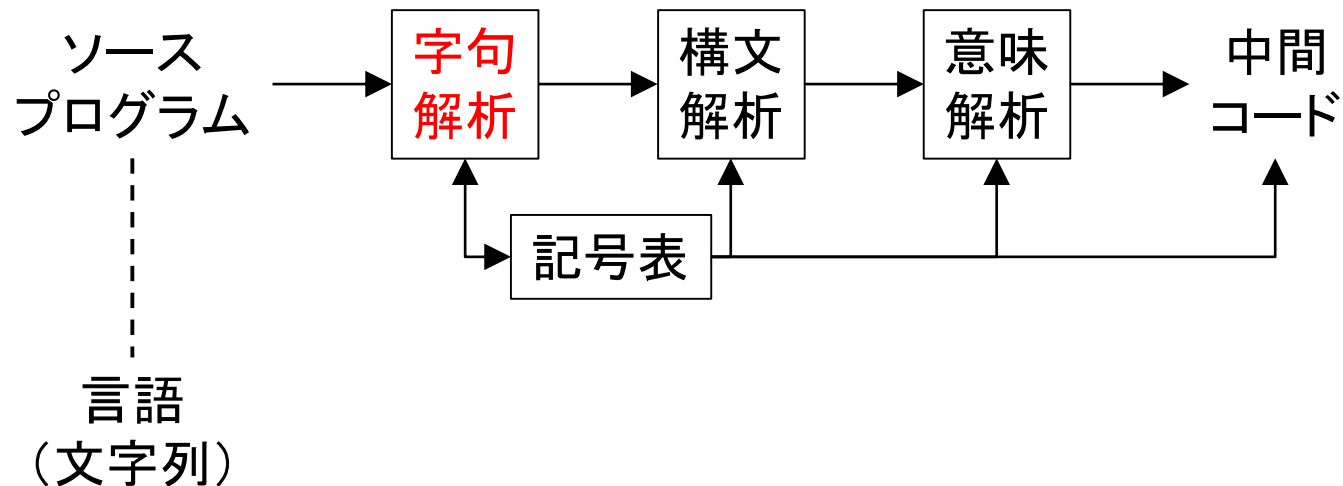
- 形式文法
 - 正規文法
- BNF記法
- 字句解析
 - 字句, 字句解析の役割, 字句の読み取り
- プログラミング技報

(再掲)コンパイラ: フロントエンド

■ ソースプログラム

□ CやJavaなどのプログラム言語で記述

- コンパイラから見ると意味を持った文(文字列)



字句・字句解析

■ 字句(トークン)

□ 意味のある語句の最小単位

- 言語は語の集合であるため, 字句が重要となる

■ 字句解析とは

□ ソースプログラムに含まれる字句の抽出

- 抽出された字句は, 後処理の構文解析で使用される



プログラムの可読性を高めるために挿入される, 改行・タブ・空白・コメント等は, 字句解析で取り除かれる

字句の種類

■ 字句の種類(プログラミング言語に依存)

□ 予約語

- if, else, for, while など

□ 識別子

- 変数名, 関数名, プログラム名など

□ 定数

- 整数, 実数など

□ 文字列リテラル

- 文字型データによる値

□ 演算子

- 四則演算, 論理演算子など

□ 区切り文字

- 空白文字, 改行, カンマ, 括弧など

字句解析の役割

■ ソースプログラムに含まれる字句の抽出

□ トークンの抽出と、型などの情報の付加

1. トークンの抽出
2. 切り出したトークンに、型情報などを付加

□ コメントの除去

- ・ プログラムの動作には影響しないため

□ 改行文字の判定による行数管理

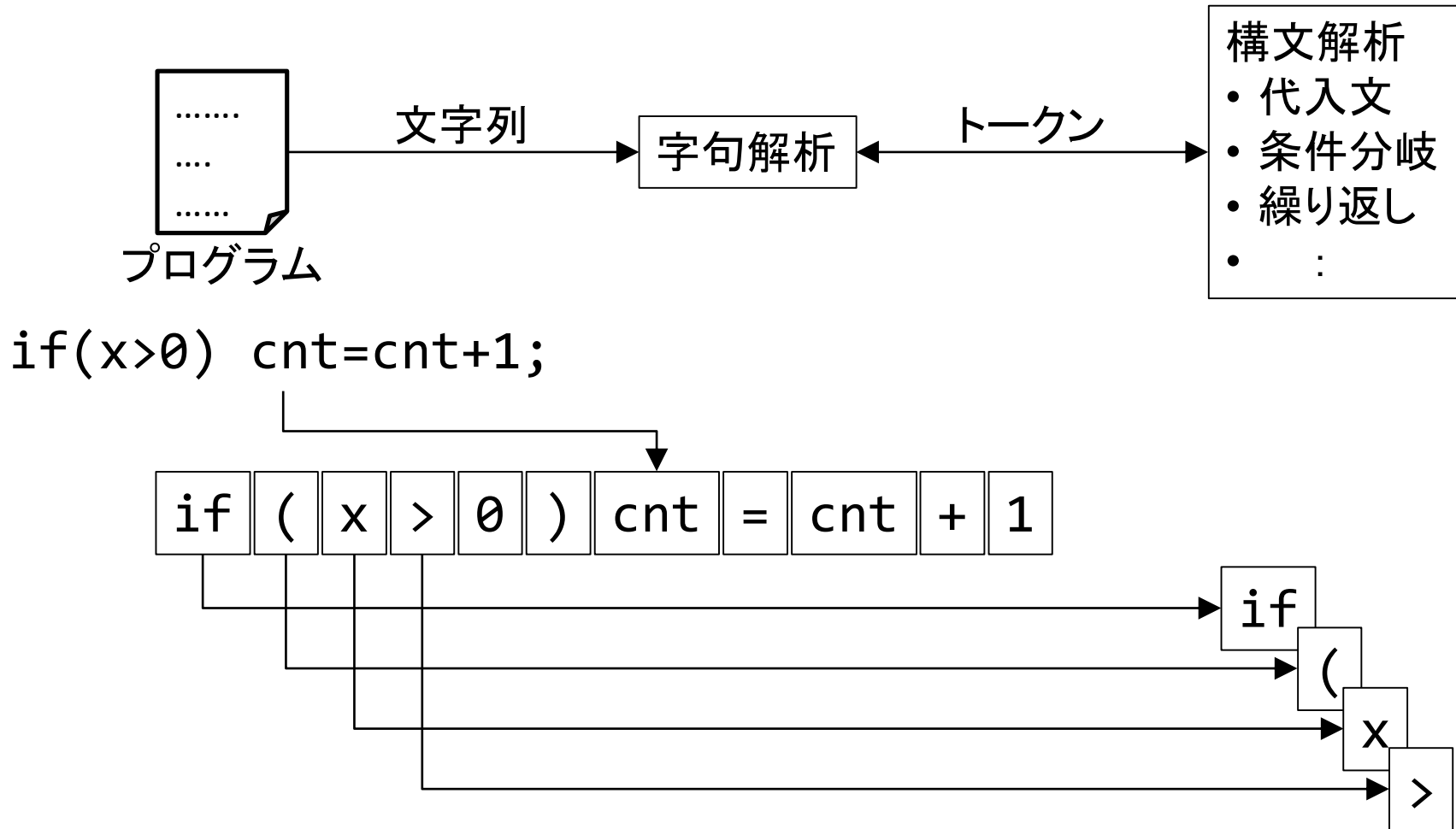
- ・ エラー個所の行番号表示等に用いる

□ 空白文字の削除

- ・ 区切り文字としての判定に使用した後

字句解析の役割

■ 構文解析で必要となるトークンの取得

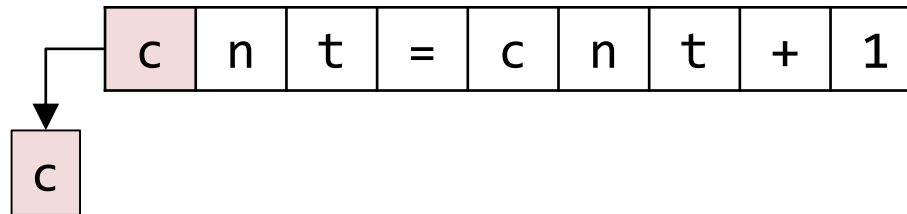


字句解析の手順

■ 文字の読み取り

□ プログラムから文字を1文字ずつ読み取る

- 1文字ずつ読み取るのは効率が悪いいため、実際はある程度(1行など), まとめて読み込む



■ 字句の読み取り

□ 読み取った文字の列から, 字句を抽出する

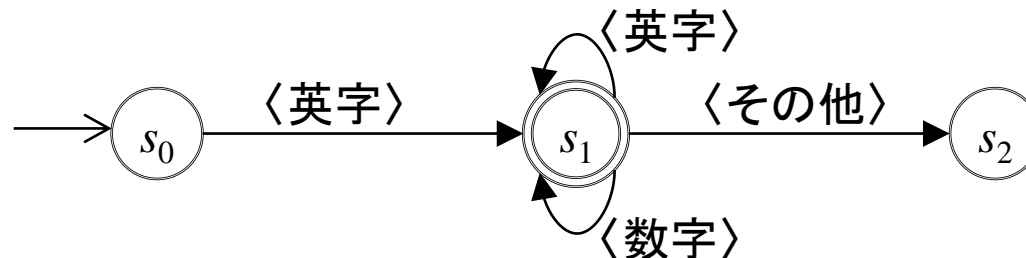
例: 字句の読み取り

■ 英字で始まる英数字の名前の読み取り

□ 構文規則

- 英字で始まる英数字の名前(文字列)

〈英字〉	::=	a b c ... z
〈数字〉	::=	1 2 3 4 5 6 7 8 9
〈名前〉	::=	〈英字〉{〈英字〉 〈数字〉}



- 形式文法
 - 正規文法
- BNF記法
- 字句解析
 - 字句, 字句解析の役割, 字句の読み取り
- プログラミング技報

文字列に対して1文字を書き込む関数

■ 文字列に対して1文字を書き込む関数

□ 書き込み個所, 文字を渡す

```
void writeStr(char **p, char c) {  
    **p = c;  
    (*p)++;  
}
```

配列のサイズチェック等の
エラー処理を行っていない
ことに注意



この関数を使わなくとも実装は可能です。関数名を適切に付けることによるプログラムの明確化, 関数の再利用性が考えられます。

演習4 Step2では便利かもしれません。

文字列に対して1文字を書き込む関数

■ 使用例

- グローバル変数 `char Str[]` へ, 文字 `ch` を順次書き込む(1文字ずつ順に書き込む)

```
char *pStr = Str;
:
ch = nextChar();
writeStr(&pStr, (char)ch);
:
ch = nextChar();
writeStr(&pStr, (char)ch);
```

Strへのポインタ
(`char Str[]`はグローバルとする)

Strへの書き込み
関数 `writeStr` を用いている

```
void writeStr(char **p, char c) {
    **p = c;
    (*p)++;
}
```

次の書き込み位置に
ポインタを移動する

まとめ

- 形式文法
 - 正規文法
- BNF記法
- 字句解析
 - 字句, 字句解析の役割, 字句の読み取り
- 演習

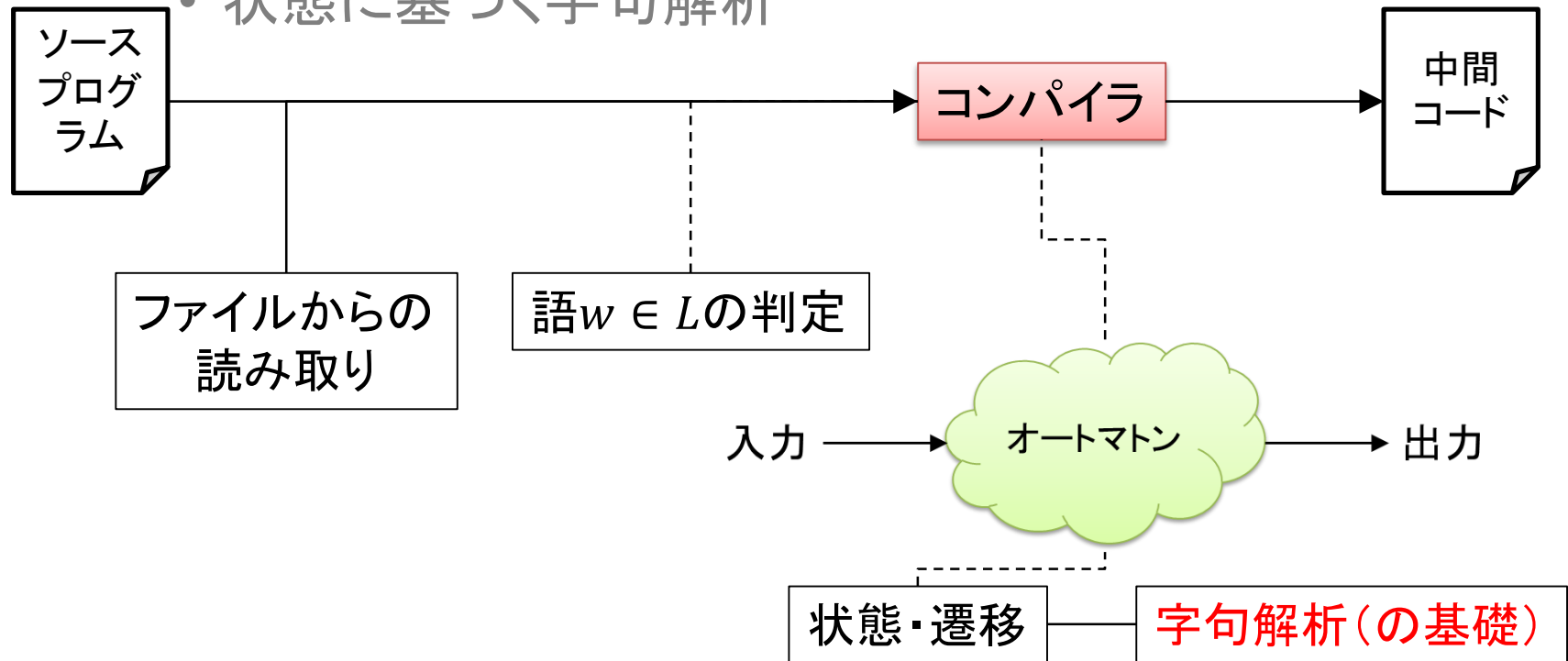
演習

演習の目的

■ 字句解析システムの構築

□ 字句の抽出の基礎を実装する

- 遷移に基づく字句解析
- 状態に基づく字句解析



字句解析システムの構築

- 符号なし2進数の字句を抽出する仕組みを実装する
 - トークンの抽出のみ
 - ・ 型情報の付加は行わない
 - 空白文字の削除
 - ・ 空白文字(スペース, タブ)
 - コメントの除去は行わない
 - 改行文字の判定による行数管理は行わない

字句解析システムの構築

■ 仕様

- 符号なし2進数は0を除き最上位は1である
- 記号列 w は, ファイルに記述された文字列(字句の列)の読み取りとする
 - ファイルを先頭から読み, トークンを切り出す
- ファイルには複数のトークンが記述されている
 - トークンの区切りは空白文字とする
空白文字: スペース, タブ($\backslash t$), ...
(`isspace`関数が真となる文字)
- 本演習では, ファイルには'0', '1', 空白文字, 改行文字以外の文字は書かれていないとする


字句解析システムの構築

■ 実装方針

オートマトンをベースとして考える.

1. 状態に基づく字句解析システム
2. 入力記号に基づく字句解析システム
(演習4)

状態に基づく字句解析

- 字句解析オートマトン M_{bn}
 - 入力を読み進め, 受理の可否を認識する
 - 2進数の値一つのみに対応
 - 拡張した字句解析オートマトン M_{bn+}
 - 複数の2進数の語を解析できるようにする
 - 語の区切りとして, 空白文字を設定する
- 
- M_{bn+} をシミュレートするプログラムの実装

字句解析オートマトン M_{bn+}

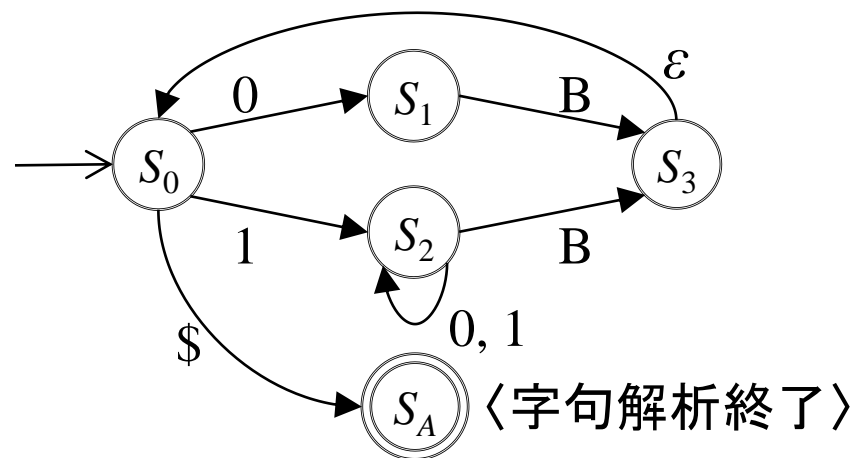
■ 追加の仕様

□ 追加された記号

- B: 1文字以上の空白 (スペース, タブ) を表わす
- \$: 文末記号を表わす

□ 2進数の値を1つのトークンとする

- 区切り文字をBで表わす



実装方針

■ 状態に基づく実装の問題点

- 状態をすべて用意する必要がある

- 2進数は状態数が少ないため問題ないが、プログラミング言語を扱うのは困難



■ 入力記号による遷移に基づく実装

- オートマトン上の遷移を考え、受理するために取りうる記号を処理する

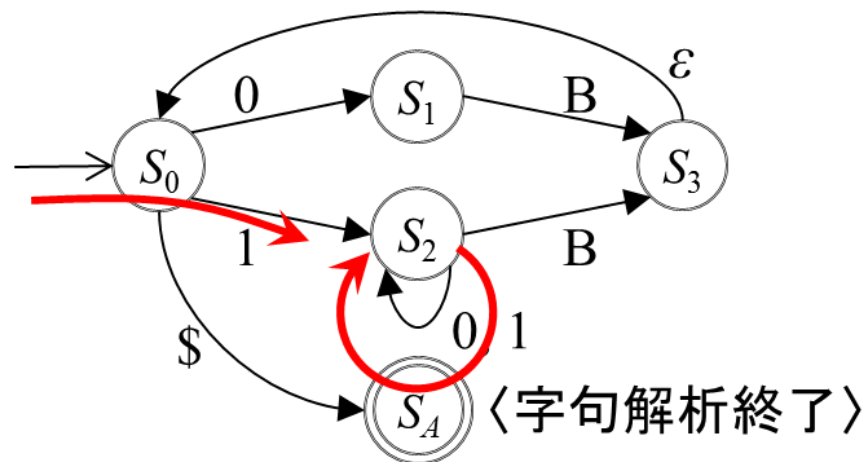
演習4: 字句解析器 lex_{bn}

■ 入力記号による遷移に基づく実装

□ 受理するために取りうる記号列を処理する

- "0", または
- "1の後に0または1が0個以上"

□ 例: 入力記号列1010





演習4: 字句解析器 lex_{bn}

■ 実装方針

- ファイルから1文字ずつ読み込む関数nextChar
(ファイル版)の実装
 - Lesson02のnextCharのファイル版
- 2進数の字句を抽出する関数nextBinaryTokenの実装

演習4: Step1: ファイルから1文字を取得

■ ファイルから記号1文字を順に取得する関数 nextCharの実装

- ファイルはオープンされているとする
- "compiler04_step1.c"

関数名	nextChar	
引数	void	
戻り値	int	FILE *fpに含まれる次の1文字
機能	FILE *fpで開かれているファイルから次の1文字を返す. • n回目に関数が呼ばれたとき, fpのn番目の文字. ただし, nがファイルに格納されている文字より大きい場合, EOFを返す. 戻り値の型がintであることに注意.	

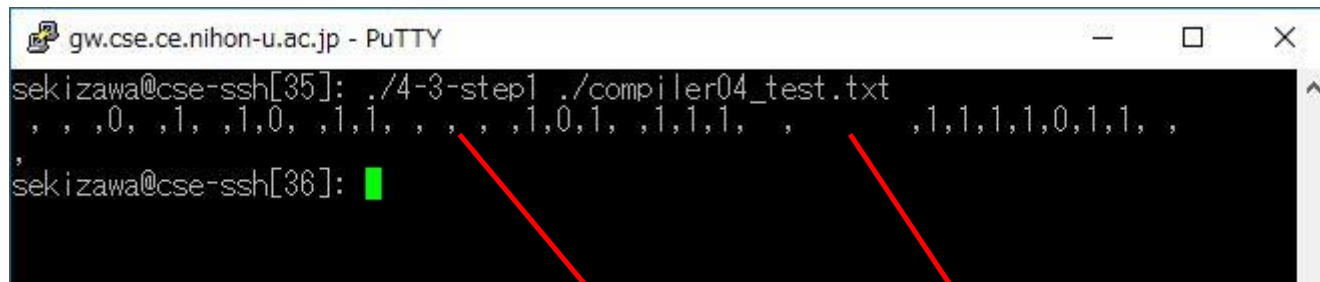
演習4: Step1: 表示の制約と実行例

■ Step1の制約

- 関数nextCharで取得した1文字を表示する
- 1文字の後にカンマ(,)を表示する

■ 例: 入力ファイル "compiler04_test.txt"

- 入力ファイルと出力を比較してみてください



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[35]: ./4-3-step1 ./compiler04_test.txt
, , 0, , 1, , 1, 0, , 1, 1, , , , , 1, 0, 1, , 1, 1, 1, , , , , 1, 1, 1, 1, 0, 1, 1, , ,
sekizawa@cse-ssh[36]:
```

空白

タブ

演習4: 字句解析器 lex_{bn}

■ 追加の仕様

□ トークンの記号列へのポインタ(char *)を得る

□ トークンは最長一致とする

- 最長一致とは, 「入力記号列から得られる最も長い文字列をトークンとして選択する(切り出す)」こと
- 最長一致の例

記号列

1	1	1	1	0	1	1
---	---	---	---	---	---	---



最長一致では, トークン "1111011" を抽出する.
"1"や"1111", "111101"は最長一致ではない.

演習4: Step2: nextBinaryToken

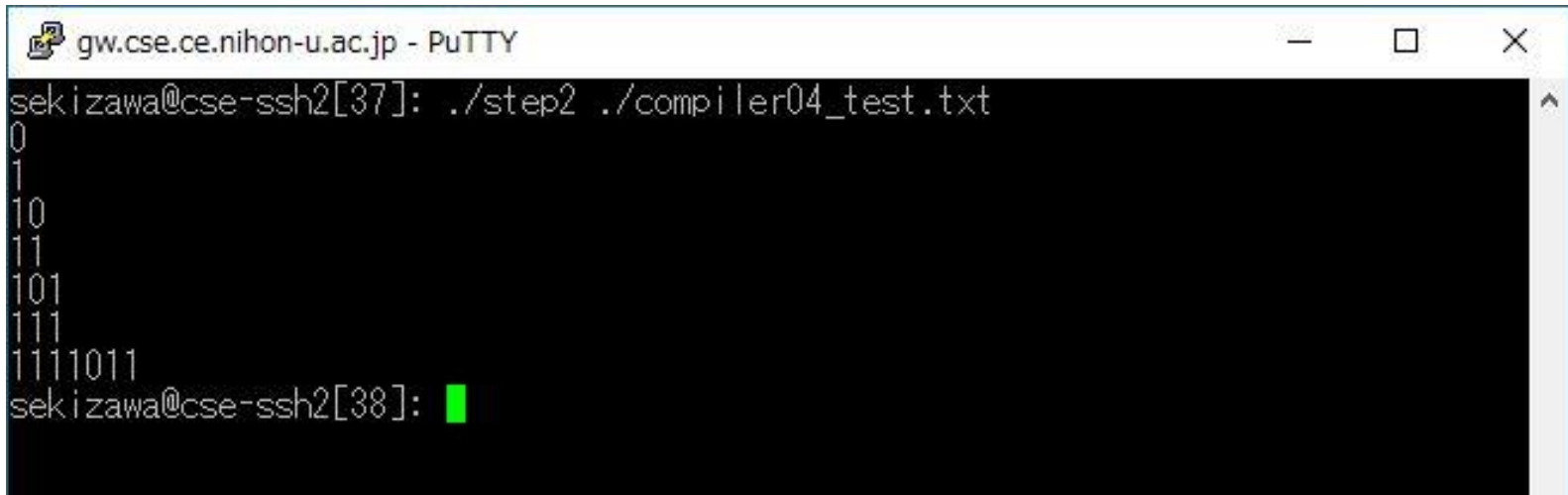
■ トークンを切り出す関数nextBinaryTokenの実装

□ "compiler04_step2.c"

関数名	nextBinaryToken	
引数	void	
戻り値	char *	トークンを格納した配列の先頭へのポインタ
機能	<p>FILE *fpで開かれているファイルから記号を1文字ずつ読み込み, 2進数を表わすトークンを取得する</p> <ul style="list-style-type: none">• トークンの文字列をグローバル変数char Str[]に格納する。(最後に'¥0'を書き込むのを忘れないように)• トークンを格納した配列の先頭へのポインタを返す• EOFに達した場合, NULLを返す• 空白は読み飛ばす	

演習4: Step2: 表示の制約と実行例

- Step2の制約
 - トークン毎に改行して表示する
- 例: "compiler04_test.txt"



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh2[37]: ./step2 ./compiler04_test.txt
0
1
10
11
101
111
1111011
sekizawa@cse-ssh2[38]:
```

演習4: Step3: binaryToken2int

- 2進数を表わすトークンを整数値に変換する関数binaryToken2intの実装
 - "compiler04_step3.c"

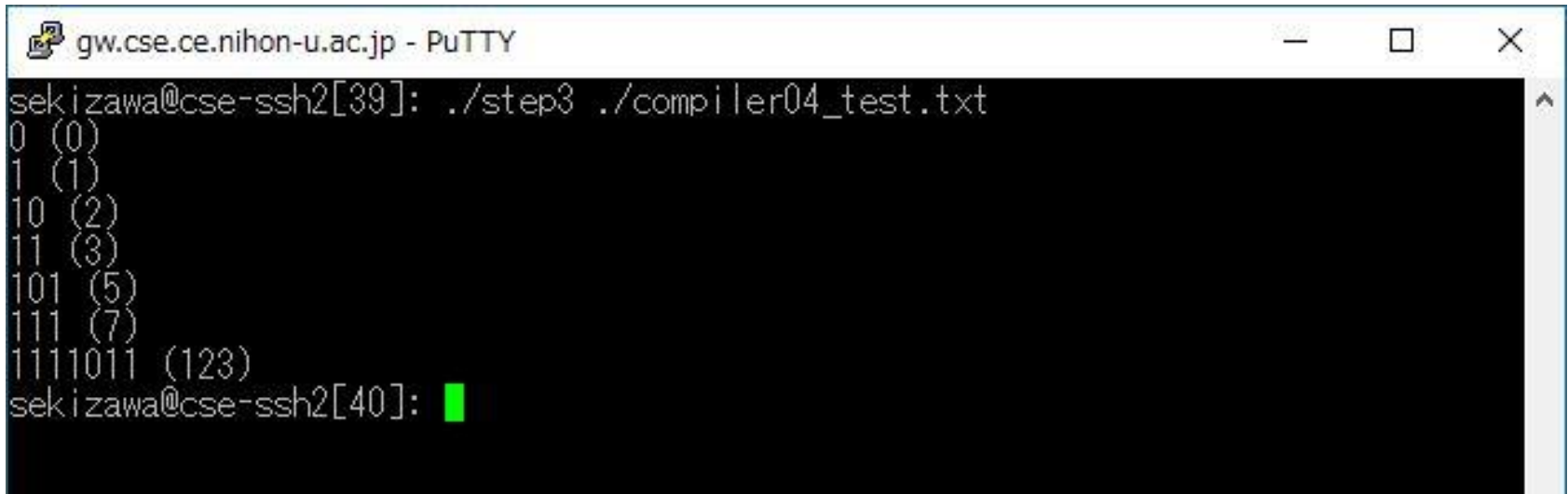
関数名	binaryToken2int	
引数	char *	トークンを格納した配列の先頭へのポインタ
戻り値	int	トークン(2進数)の値
機能	トークンを格納した配列の先頭へのポインタが指している配列が表わす符号なし2進数が表わす整数値を求める. • 配列が2進数でない場合, -1を返す	

演習4: Step3: 表示の制約と実行例

■ Step2の制約

- トークン毎に改行して表示する
- 続けて, 半角スペース1文字を入れた後, 括弧()内に10進数の値を表示する

■ 例: "compiler04_test.txt"



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh2[39]: ./step3 ./compiler04_test.txt
0 (0)
1 (1)
10 (2)
11 (3)
101 (5)
111 (7)
1111011 (123)
sekizawa@cse-ssh2[40]:
```