コンパイラ

第4回 字句解析
― 字句解析プログラムの作成 ―

http://www.info.kindai.ac.jp/compiler

E館3階E-331 内線5459

takasi-i@info.kindai.ac.jp

コンパイラの構造

- 字句解析系
- 構文解析系
- 制約検査系
- 中間コード生成系
- 最適化系
- 目的コード生成系

処理の流れ
情報システムプロジェクトIの場合

output (ab);

↓

字句解析系　　マイクロ構文の文法に従い解析

↓

“output” “(” 変数名 “)” “;”

↓

構文解析系　　マクロ構文の文法に従い解析

↓

<output_statement> ::= “output” “(” <exp> “)” “;”

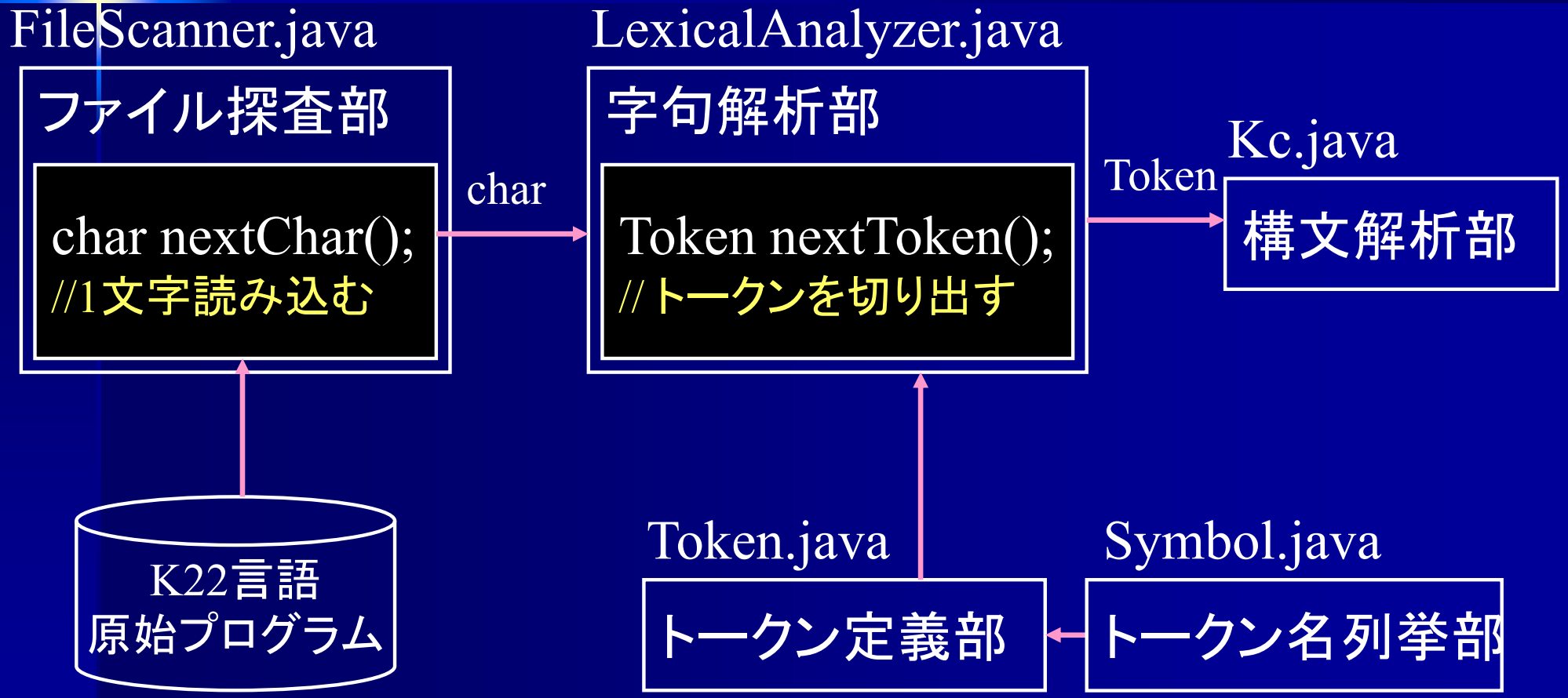
↓

コード生成系　　VSMアセンブラの文法に従い生成

↓

1. PUSH &ab　　2. OUTPUT

プログラムの構造(字句解析系)



FileScanner		# ファイル探査部
- sourceFile	: BufferedReader	# 入力ファイルの参照
- line	: String	# 行バッファ
- lineNumber	: int	# 行カウンタ
- columnNumber	: int	# 列カウンタ
- currentCharacter	: char	# 読み取り文字
- nextCharacter	: char	# 次の読み取り文字
FileScanner (sourceFileName : String)		# コンストラクタ
closeFile ()	: void	# 入力ファイルを閉じる
readNextLine ()	: void	# ファイルから1行読み込む
nextChar ()	: char	# 1文字切り出す
lookAhead ()	: char	# 次の読み取り文字を返す
getLine ()	: String	# 読み取り行を返す
scanAt ()	: String	# 読み取り位置を返す

文字の読み込み

FileScanner.java

```
import java.nio.file.*;
import java.io.*;
/**
 * 入力ファイルから文字列を読み出し、字句解析器に1文字ずつ渡すクラス
 */
class FileScanner {
    private BufferedReader sourceFile;  // 入力ファイルの参照
    private String line;                // 行バッファ
    private int lineNumber;             // 行カウンタ
    private int columnNumber;           // 列カウンタ
    private char currentCharacter; // 読み取り文字
    private char nextCharacter;         // 次の読み取り文字
}
```

文字の読み込み
(情報システムプロジェクトIの場合)

FileScanner.java

```
/**
 * 現在読んでいる文字の次の文字を返し、1文字読み進める
 * @return 次の文字 (行末なら '¥n', ファイル末なら '¥0' )
 */
 char nextChar () {
     currentCharacter = nextCharacter;
     走査位置を1文字進める;
     if (走査位置がファイル末か？) nextCharacter = '¥0';
     else if (走査位置が行末か？) nextCharacter = '¥n';
     else nextCharacter = 走査位置の文字;
     return currentCharacter;
 }
```

字句解析系 (lexical analyzer, scanner)

- 字句解析系
  - 空白、コメントを読み飛ばす
  - 単語(token)に区切る
  - マイクロ構文エラーを検出

```
if (ans >= 123 ) /* ansの値で分岐 */ (改行)
    (空白)output (‘1’) ;
```

予約語 “if”
左括弧 “(”
変数　“ans”
不等号 “>=”
整数　“123”
右括弧 “)”
予約語 “output”
　：

トークンの種類
(情報システムプロジェクトIの場合)

トークン		記号
区切り記号		; , ( ) { } [ ]
演算子	比較演算子	== != < > (<=) (>=)
	論理演算子	! && ||
	算術演算子	+ - * / %
	代入演算子	= += -= *= /= ++ --
名前		変数名
定数		整数 文字 (文字列)
予約語		main int if while for inputint inputchar outputint outputchar outputstr setstr (else) (do) (break) …

Token クラス

Token		# トークン定義部
- symbol	: Symbol	# トークンの種類
- intValue	: int	# トークンの値
- strValue	: String	# トークンの名前
Token (symbol : Symbol)		# コンストラクタ
Token (symbol : Symbol, intValue : int)		# コンストラクタ
Token (symbol : Symbol, strValue : String)		# コンストラクタ
checkSymbol (symbol : Symbol)	: boolean	# トークンの種類を比較
getSymbol ()	: Symbol	# トークンの種類を返す
getIntValue ()	: int	# トークンの値を返す
getStrValue ()	: String	# トークンの名前を返す

Token クラス

```
class Token {
    Symbol symbol; /* トークンの種類 */
    int intValue;        /* 整数値 または 文字コード */
    String strValue;    /* 変数名 または 文字列 */
}
```

トークン	symbol	intValue	strValue
main	MAIN		
==	EQUAL		
123	INTEGER	123	
‘a’	CHARACTER	97 (‘a’の文字コード)	
time	NAME		“time”

Tokenクラスのオブジェクト

```
Token token;
```

“main” のトークン

```
token = new Token (Symbol.MAIN);
```

“+” のトークン

```
token = new Token (Symbol.ADD);
```

以降は Symbol. は省略

トークンへの分轄

```
if (ans >= 123 ) /* ansの値で分岐 */ (改行)
    (空白)output (‘1’) ;
```

Token (IF)
Token (LPAREN)
Token (NAME)
Token (GREATEQ)
Token (INTEGER)
Token (RPAREN)

Token (OUTPUT)
Token (LPAREN)
Token (CHARACTER)
Token (RPAREN)
Token (SEMICOLON)

次のトークンを得る Token nextToken() メソッドを作成

値を持つトークン

- 値を持つトークン
  - 整数(整数値)
  - 文字(文字コード)
  - 変数名(文字列)

整数 1　12　256　⇒ Token (INTEGER)

しかし整数は値を区別する必要がある

Token (INTEGER, 1)
Token (INTEGER, 12)
Token (INTEGER, 256)

文字　‘a’　　変数名　time

Token (CHARACTER, ‘a’)　Token (NAME, “time”)

LexicalAnalyzer クラス

LexicalAnalyzer		# 字句解析部
- sourceFileScanner	: FileScanner	# 入力ファイルの参照
LexicalAnalyzer (sourceFileName : String)		# コンストラクタ
closeFile ()	: void	# 入力ファイルを閉じる
nextToken ()	: Token	# トークンを切り出す
analyzeAt ()	: String	# 読み取り位置を返す
- syntaxError ()	: void	# エラー検出時の処理

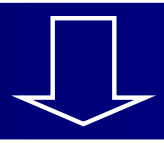
nextToken()

```
Token nextToken () {
   Token token;

   空白を読み飛ばす;
   トークンを切り出す;

   if (“++” を切り出した場合) token = // “++”のトークン生成
   else if (“+” を切り出した場合) token = // “+” のトークン生成
   else if (“if” を切り出した場合) token = // “if” のトークン生成
   else if (整数を切り出した場合) token = // 整数のトークン生成
   else if (名前を切り出した場合) token = // 名前のトークン生成
                  :              /* 以下各トークンに対する処理を else if で並べる */
   else syntaxError();    /* どのトークンとも一致しなかった場合はエラー */

   return token;
}
```

空白の読み飛ばし

- 空白,コメントは字句解析時に読み飛ばす
  - 空白：‘ ’(スペース) ‘¥n’(改行) ‘¥t’(タブ記号)
  - コメント(拡張課題)：“/* … */” “// … ¥n”

```
if (ans >= 123 ) /* ansの値で分岐 */ (改行)
    (空白)output (‘1’) ;
```



```
if(ans>=123)output(‘1’);
```

空白の読み飛ばし

- 字句解析部のプログラム

```
char currentChar;                    // 現在位置の文字

do {
    currentChar = nextChar();        // 次の文字を読み込む
} while (currentChar == ‘ ’);        // 空白文字の間ループ
```

‘¥n’, ‘¥t’ も同様に読み飛ばす

コメントも読み飛ばすが処理が少し難しい
(コメントは拡張課題)

LexicalAnalyzer.java

```
Token nextToken() {
  String word = “”;

  /* 文字列を単語に切り分ける */
  while (単語が続く間) {
      String word  += nextChar();
  }

  if (word が  “main”)   token = new Token (MAIN);
  else if (word が “if”)   token = new Token (IF);
  else if (word が “+”)   token = new Token (ADD);
  else if (word が “++”) token = new Token (INC);
  else if (word が “0”)   token = new Token (ZERO);
              :
  return token;
}
```

“単語が続く間”の判定はどのように行う？

単語への分割

英語の場合

School of Science and Engineering Kindai University

⇒単語間に空白があるので区切るのは簡単

日本語の場合

きんきだいがくりこうがくぶ

きんき : だいがく : りこうがくぶ　近畿 大学 理工学部

きんきだ : いがくり : こうが : くぶ　近畿だ イガ栗 黄河 九分

区切り方を正しく決定するのは困難

計算機言語の場合は？

単語への分割

計算機言語の場合

区切り記号で単語を判別できる

```
main () {
    int i, j, k;
        :
```

m a i n (

区切り記号 (が来たので
“main”で区切ると判別

単語への分割

どう区切る？

+++---===

+ + + - - - = = =

++ + -- - == =

+ ++ - -- = ==

++ + -- -= ==

文字列 “++” は INC？
それとも ADD ADD？

記号	トークン名
=	ASSIGN
==	EQUAL
+	ADD
-	SUB
+=	ASSIGNADD
-=	ASSIGNSUB
++	INC
--	DEC

最長一致で判断

最長一致(longest matching)

- 最長一致
  - 複数の字句の可能性がある

  ⇒そのうち最長の字句であると認識

  きょうとふきょうとし

  きょうとふ：きょうとし　　京都府：京都市

  きょうと：ふきょう：とし　京都：不況：都市

  きょう：とふ：きょ：うとし 今日：塗布：虚：疎し

  最長の“きょうとふ”と認識

最長一致

とうきょうとっきょきょかきょくきょかきょくちょう

とうきょうと：っきょきょかきょく～

東京都：っきょきょかきょく～

とうきょう：とっきょきょかきょく～

東京：特許：許可局：許可局長

自然言語は最長一致では解決できない場合もある

計算機言語は基本的に最長一致でOK

最長一致

+++---===

左から順に一致をチェック

++　+　--　-=　==

記号	トークン名
=	ASSIGN
==	EQUAL
+	ADD
-	SUB
+=	ASSIGNADD
-=	ASSIGNSUB
++	INC
--	DEC

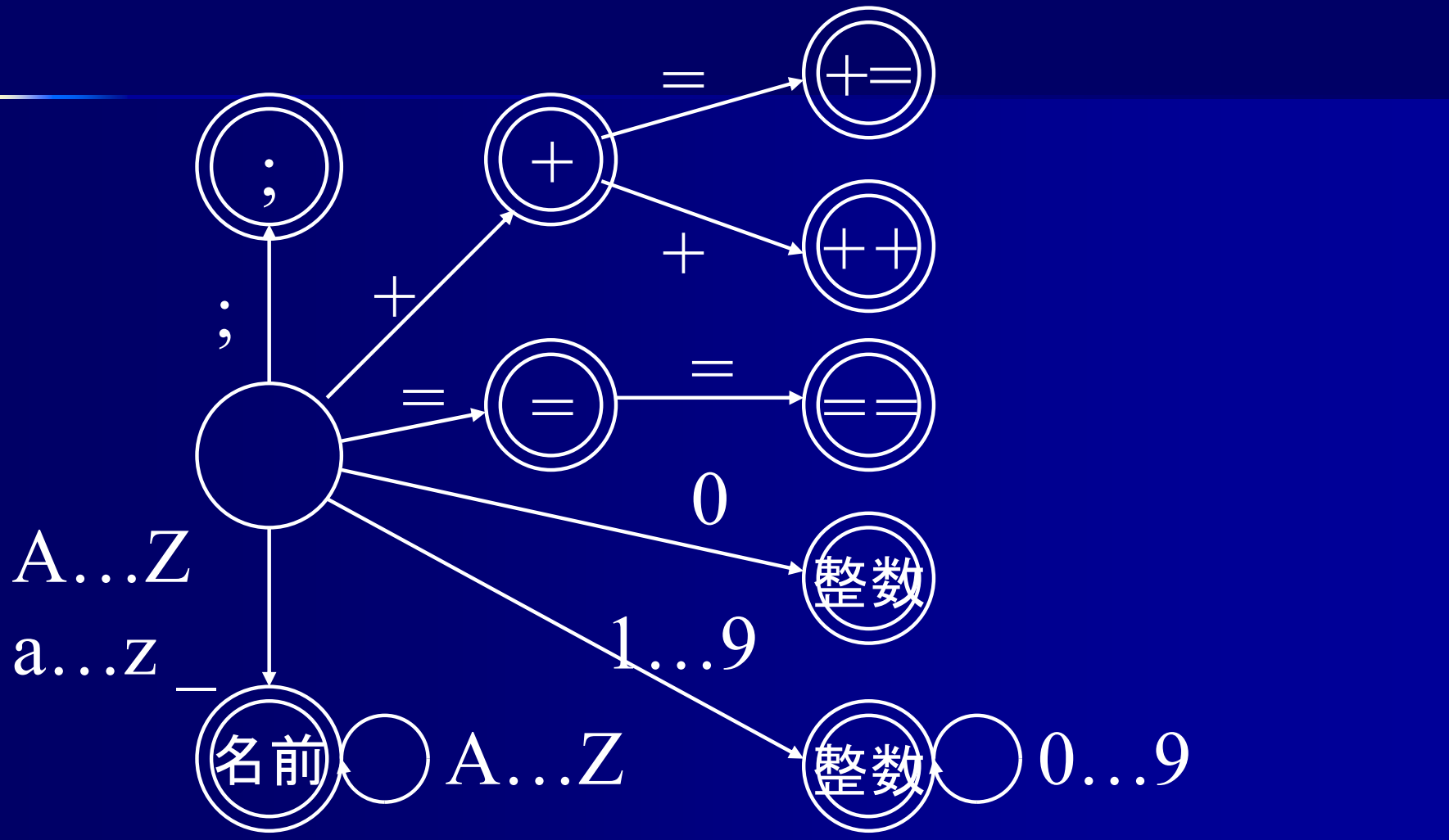
最長一致

a <= b	× a < = b	○ a <= b
+++	× + ++	○ ++ +
+ ++	× ++ +	○ + ++
-++	× -+ +	○ - ++
123	× 1 2 3	○ 123
main12	× main 12	○ main12 (変数名)

空白で区切られている

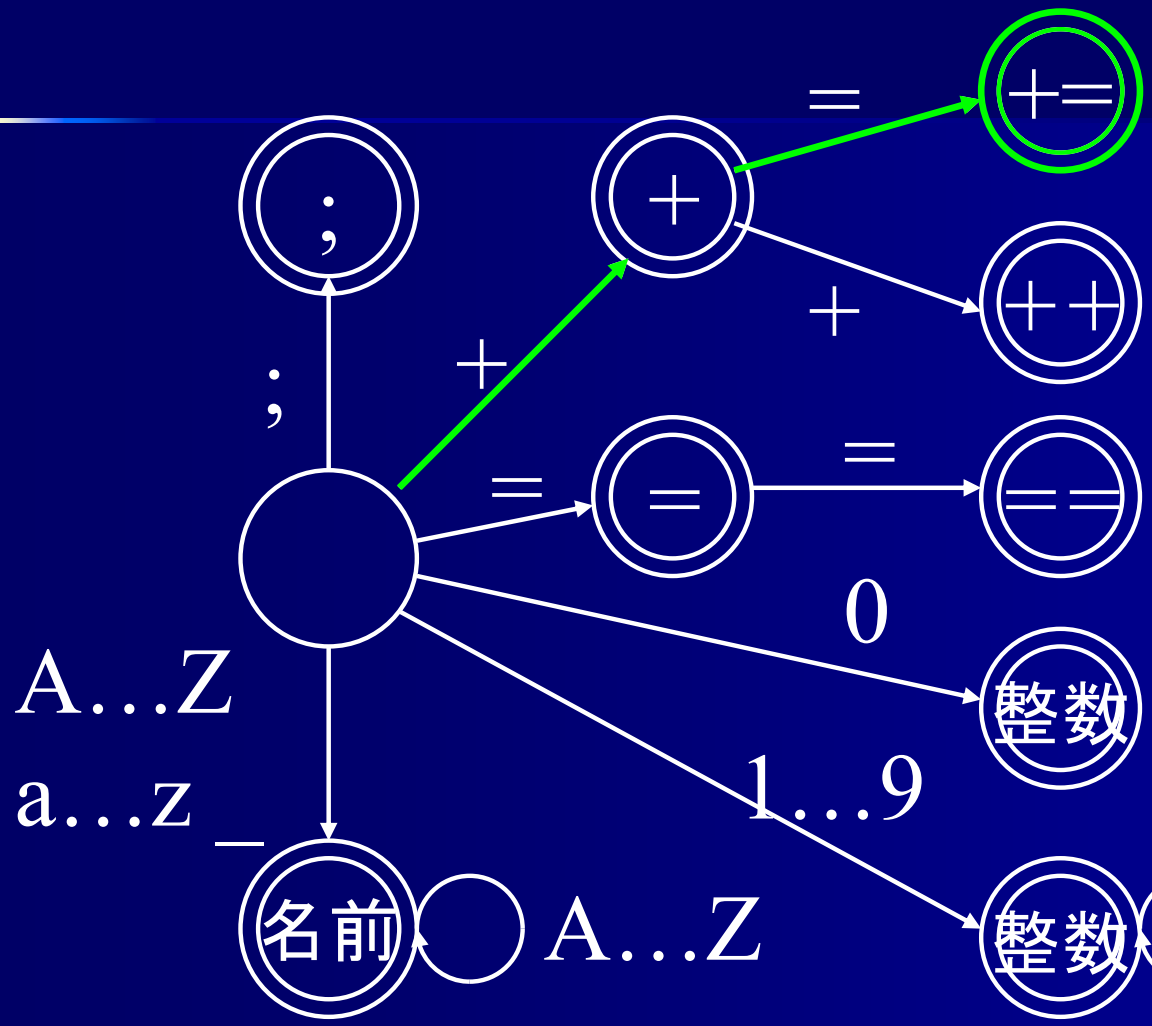
-+ というトークンは無い

字句解析オートマトン(一部)



矢印を辿れる限り
先へ進む

字句解析オートマトン(一部)



+= の場合

矢印を辿れる限り
先へ進む

トークンの識別

```
if (ans>=123) output (‘1’);
```

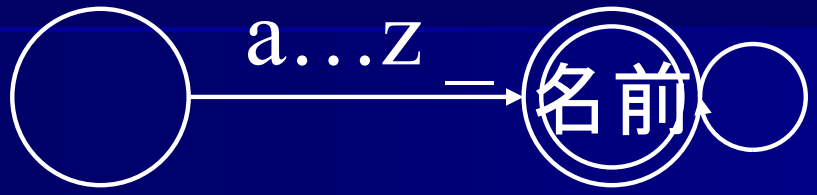
if (ans >=123) output (‘1’) ;

Token (IF)
Token (LPAREN)
Token (NAME, “ans”)
Token (GREATEQ)
Token (INTEGER, 123)
Token (RPAREN)

Token (OUTPUT)
Token (LPAREN)
Token (CHARACTER, ‘1’)
Token (RPAREN)
Token (SEMICOLON)

変数名が ans で終わるとどうやって判定する？
もしかしたら変数名 answer の一部かも？

トークンの識別



英数字が来る間ループ

youAre20YearsOld =

英数字以外

変数名 youAre20YearsOld と識別

最後に読んだ ‘=’ は？

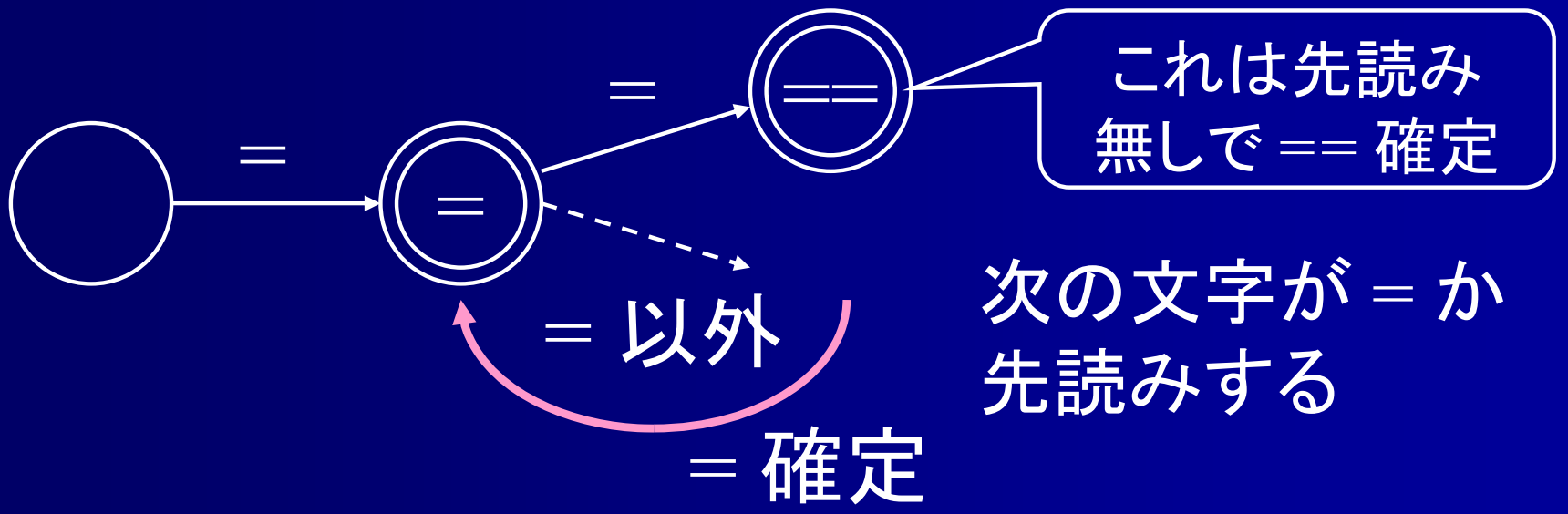
‘=’ は次のトークン(の一部)

⇒次のトークンの識別のため再度読む必要あり

先読みを行う

先読み(lookahead)

- 先読み
  - トークンが終了するか否かを何文字か先の文字を読んで判定
    - 多くの言語では1文字先読みで判定できる



先読み

- Java, C, Pascal 等多くの言語
  - 1文字先読みすればトークンを識別可能
- FORTRAN
  - 無限に先読みが必要な可能性がある

FORTRAN90 の場合 (FORTRANでは空白は無視される)

```
DO  I=1, 20
```

コンマ

DO(予約語)　I(変数名)　＝　1　，　20

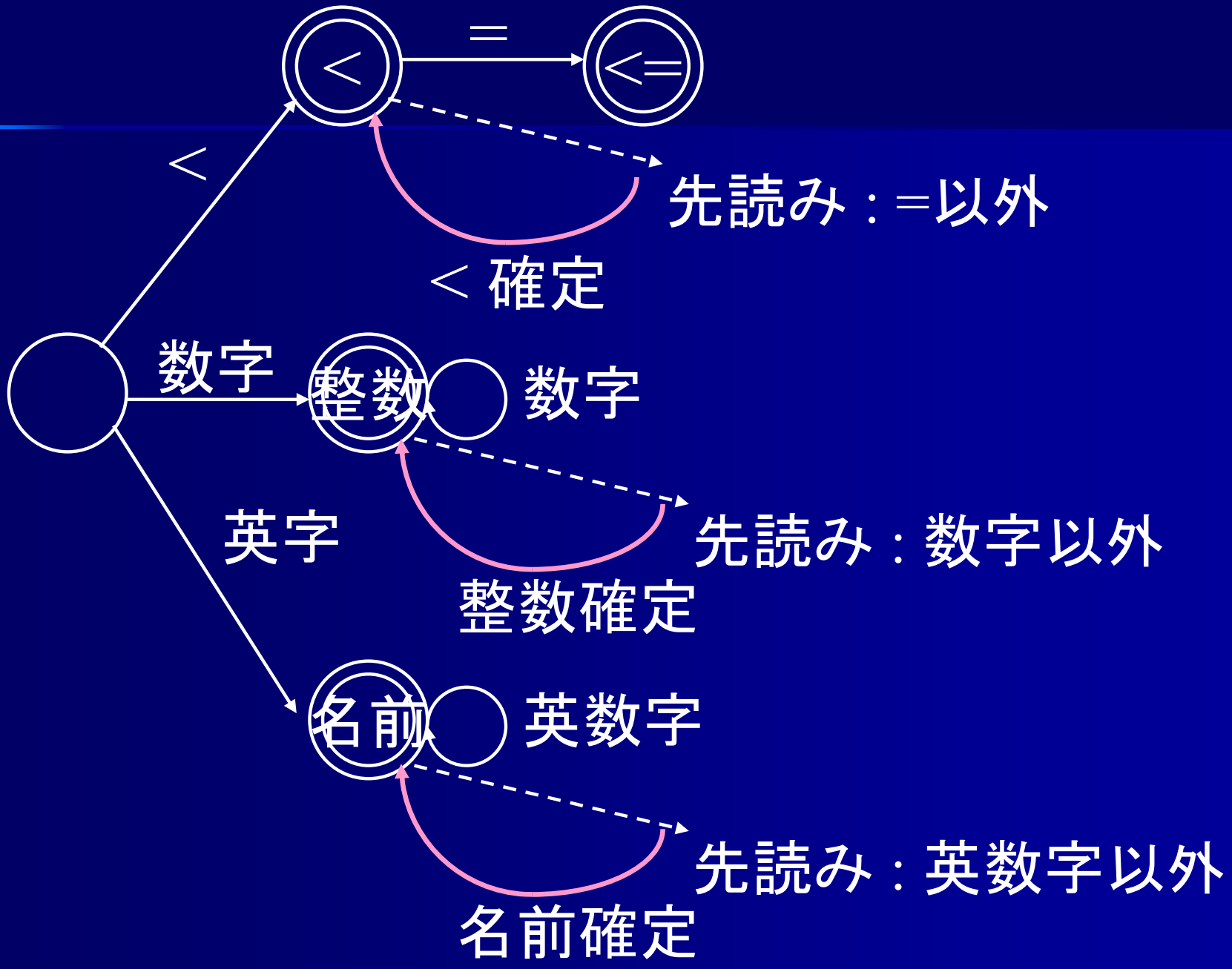
```
DO  I=1. 20
```

ピリオド

DOI(変数名)　＝　1.20

，．まで読まないと
DO と DOI を識別できない

トークンの識別



トークンの識別

```
if(ans>=123 )write (‘1’);
```

if(　　英数字以外が来た⇒ if で区切る

Token (IF)

(　　(は単独でトークン

Token (LPAREN)

ans>　英数字以外が来た⇒ ans で区切る

Token (NAME, “ans”)

トークンの識別

```
if (ans>=123) output (‘1’);
```

if (ans >=123) output (‘1’) ;

Token (IF)
Token (LPAREN)
Token (NAME, “ans”)
Token (GREATEQ)
Token (INTEGER, 123)
Token (RPAREN)

Token (OUTPUT)
Token (LPAREN)
Token (CHARACTER, ‘1’)
Token (RPAREN)
Token (SEMICOLON)

if は (を先読みして判定

(は (単独で判定可能

ans は > を先読みして判定

先読み
(情報システムプロジェクトIの場合)

FileScanner.java

```
/**
 * 現在読んでいる文字の次の文字を返す
 * @return 次の文字 (行末なら '¥n', ファイル末なら '¥0' )
 */
 char lookAhead () {
     if (次がファイル末か？) return '¥0';
     else if (次が行末か？) return '¥n';
     else return 次の文字;
 }
```

記号解析部分のプログラム

- 字句解析部のプログラム

```
char currentChar;        // 現在位置の文字
char nextChar();         // 次の文字を読み込み1文字進める
char lookAhead();        // 次の位置の文字の先読み
```

例 +, ++ の解析

```
if (currentChar == '+') {
    if (lookAhead() == '+') {          /* 1文字先読みする */
        nextChar();                    /* 2文字目の'+'を読む*/
        token = new Token (INC);       /* ++ と判定 */
    } else {
        token = new Token (ADD)        /* + と判定 */
    }
}
```

記号解析部分のプログラム

- 字句解析部のプログラム

```
char currentChar;        // 現在位置の文字
char nextChar();         // 次の文字を読み込み1文字進める
char lookAhead();        // 次の位置の文字の先読み
```

例 -, -= の解析

```
if (currentChar == '-') {
    if (lookAhead() == '=') {              /* 1文字先読みする */
        nextChar();                        /* 2文字目の'='を読む*/
        token = new Token (ASSIGNSUB);     /* -= と判定 */
    } else {
        token = new Token (SUB)            /* - と判定 */
    }
}
```

整数の解析

- 整数：数字の並び　　数字 ∈ {0…9}

123　Token (INTEGER, 123)

0 は単独で整数

(007 は　整数0　整数0　整数7 と識別)

数値への変換, 数字の判定

```
int i;  char c;  boolean b;
```

- 数値への変換　文字 ‘1’ → 整数 1

```
i = Character.digit (c, 10);
```

10進数

```
i = c - ‘0’;
```

文字コード ‘0’ の値を引く

- 数字か？

```
b = Character.isDigit (c);
```

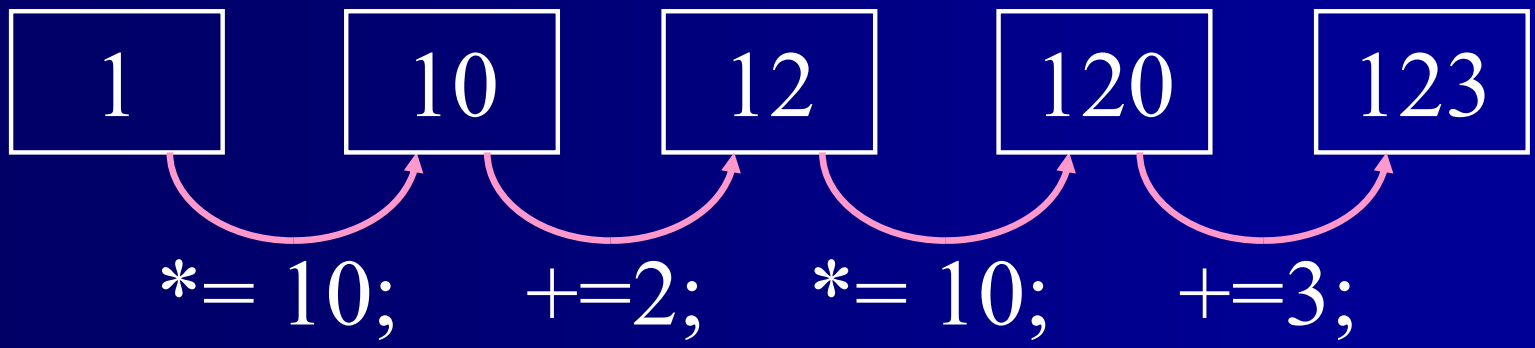
```
b = (‘0’ <= c && c <= ‘9’);
```

```
b = (Character.digit (c, 10) != -1);
```

数値への変換

- 2桁以上の整数値への変換
  - 「値を10倍して次の数値を足す」を繰り返す

例：123



数値への変換, 数字の判定(16進数)

```
int i;  char c;  boolean b;
```

- 数値への変換　文字 ‘c’ → 整数 12

```
i = Character.digit (c, 16);
```

16進数

```
i = c - ‘0’;          (c∈{0…9} のとき)
i = c – ‘A’ + 10;   (c∈{A…F} のとき)
```

- 数字か？

```
b = (‘0’ <= c && c <= ‘9’ || ‘A’<= c && c <= ‘F’);
```

```
b = (Character.digit (c, 16) != -1);
```

整数解析部分のプログラム

- 字句解析部のプログラム

```
if (currentChar ∈{‘0’…‘9’}) {
    int value = /* 文字currentCharを数値に変換 */
    while (lookAhead() ∈ {‘0’…‘9’}) { /* 次が数字の間ループ*/
        currentChar = /* 次の数字を読む*/
        value *= /* 数値を1桁ずらす */
        value += /* currentCharを数値に変換して加える */
    }
    token = /* 整数のトークン生成 */
}
```

しかしこれでは 007 を 整数7 と判別してしまう

⇒ 0 は別処理に

整数解析部分のプログラム

- 字句解析部のプログラム

```
if (currntChar == ‘0’) {          /* 先頭が0の場合は別処理 */
    token = /* 整数のトークン生成 */
} else if (currentChar ∈{‘1’…‘9’}) { /* 0以外の場合の処理 */
    int value = /* 文字currentCharを数値に変換 */
    while (lookAhead() ∈ {‘0’…‘9’} { /* 次が数字の間ループ*/
        currentChar = /* 次の数字を読む */
        value *= /* 数値を1桁ずらす */
        value += /* currentCharを数値に変換して加える*/
    }
    token = /* 整数のトークン生成 */
}
```

整数解析部分のプログラム
(値を文字列として記憶する場合)

```
if (currntChar == ‘0’) {          /* 先頭が0の場合は別処理 */
    token = /* 整数のトークン生成 */
} else if (currentChar ∈{‘1’…‘9’}) {
    String st = “” + currentChar;          /* 文字列として記憶 */
    while (lookAhead() ∈ {‘0’…‘9’} { /* 次が数字の間ループ*/
        currentChar = /* 次の数字を読む*/
        st += /* currentCharを文字列に追加 */
    }
    int value = /* 文字列stを整数に変換 */
    token = /* 整数のトークン生成 */
}
```

文字の解析

- 文字：‘(シングルクォート)*(任意の文字)’(シングルクォート)

‘a’　Token (CHARACTER, 97(‘a’の文字コード))

文字コードへの変換(Javaの場合)

```
int value;
char ch = ‘a’;

value = (int) ch;
```

文字解析部分のプログラム

- 字句解析部のプログラム

```
char currentChar;          // 現在位置の文字
char nextChar();           // 次の文字を読み込み1文字進める
char lookAhead();          // 次の位置の文字の先読み
```

```
if (currentChar == ‘¥‘’) {          /* 1文字目がシングルクォート*/
    currentChar = /* 2文字目を読む*/
    value = /* currentCharを数値に変換 */
    currentChar =  /* 3文字目を読む*/
    if (currentChar != ‘¥’’) {   /* 3文字目がシングルクォート以外*/
        syntaxError();              /* 字句解析エラー */
    }
    token = /* 文字のトークン生成 */
}
```

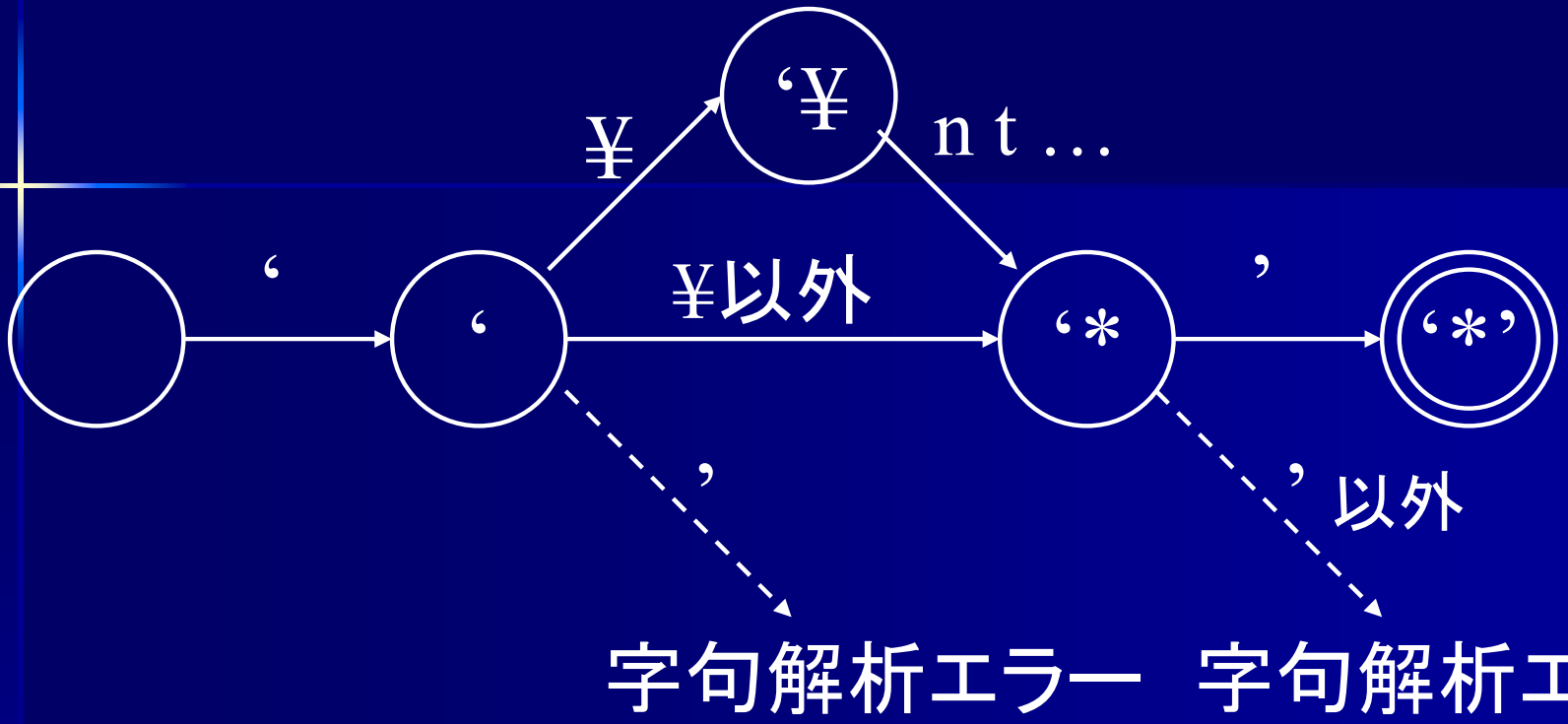
特殊文字

Javaの特殊文字

記号	意味
¥n	改行
¥t	タブ
¥r	行頭復帰
¥f	改ページ
¥b	バックスペース
¥¥	¥ (バックスラッシュ)
¥‘	‘ (シングルクオート)
¥“	“ (ダブルクオート)
¥uhhhh	文字コードhhhh(16進数)の文字

‘¥n’ で1文字

(特殊文字を含む)文字の解析



K20言語のマイクロ構文では特殊文字は不要
(特殊文字は発展課題)

‘‘’：シングルクォート

‘“’：ダブルクォート

‘¥’：バックスラッシュ

```
if (currentChar == ‘¥‘’) {          /* 1文字目がシングルクォート*/
    currntChar = nextChar();                    /* 2文字目を読む*/
    if (nextChar == ‘¥¥’) {          /* 2文字目がバックスラッシュ */
        currentChar = nextChar();               /* 3文字目を読む*/
        if (currntChar == ‘n’) value = (int) ‘¥n’;
        else if (currentChar = ‘t’) value = (int) ‘¥t’;
            :
    } else {
        if (currentChar ==‘¥’’) syntaxError(); /* 2文字目が ‘ はエラー */
        else value = (int) currentChar; /* 2文字目の文字コードを記憶 */
    }
    currentChar = nextChar();                   /* 3文字目を読む*/
    if (currentChar != ‘¥‘’) syntaxError();
    token = /* 文字のトークン生成 */
}
```

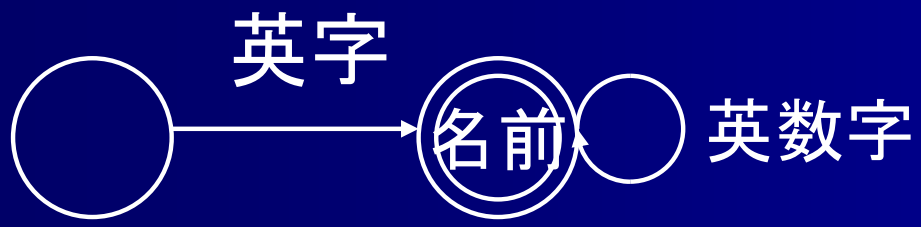
特殊文字の処理

変数名，予約語

- 変数名，予約語：英字の並び

  英字 $\in$ {a…z, A…Z, _}

  英数字 $\in$ {a…z, A…Z, _, 0…9}



変数名も予約語も文法上の制約は同じ

英字の判定

```
char c;  boolean b;
```

- 英小文字か？

```
b = Character.isLowerCase (c);
```

```
b = (‘a’ <= c && c <= ‘z’);
```

- 英大文字か？

```
b = Character.isUpperCase (c);
```

```
b = (‘A’ <= c && c <= ‘Z’);
```

変数名解析部分のプログラム

- 字句解析部のプログラム

```
char currentChar;          // 現在位置の文字
char nextChar();           // 次の文字を読み込み1文字進める
char lookAhead();          // 次の位置の文字の先読み
```

```
String name = “”;
if (currentChar ∈ {a…z, A…Z, _}) {          /* 1文字目が英字 */
    String name +=  /* currentCharを結合する */
    while (lookAhead() ∈ {a…z, A…Z, _, 0…9}) {
        currentChar = /* 次の文字を読み込む */
        name += /* currentCharを結合する */
    }
    token = /* 変数名のトークン生成*/
}
```

変数名解析部分のプログラム

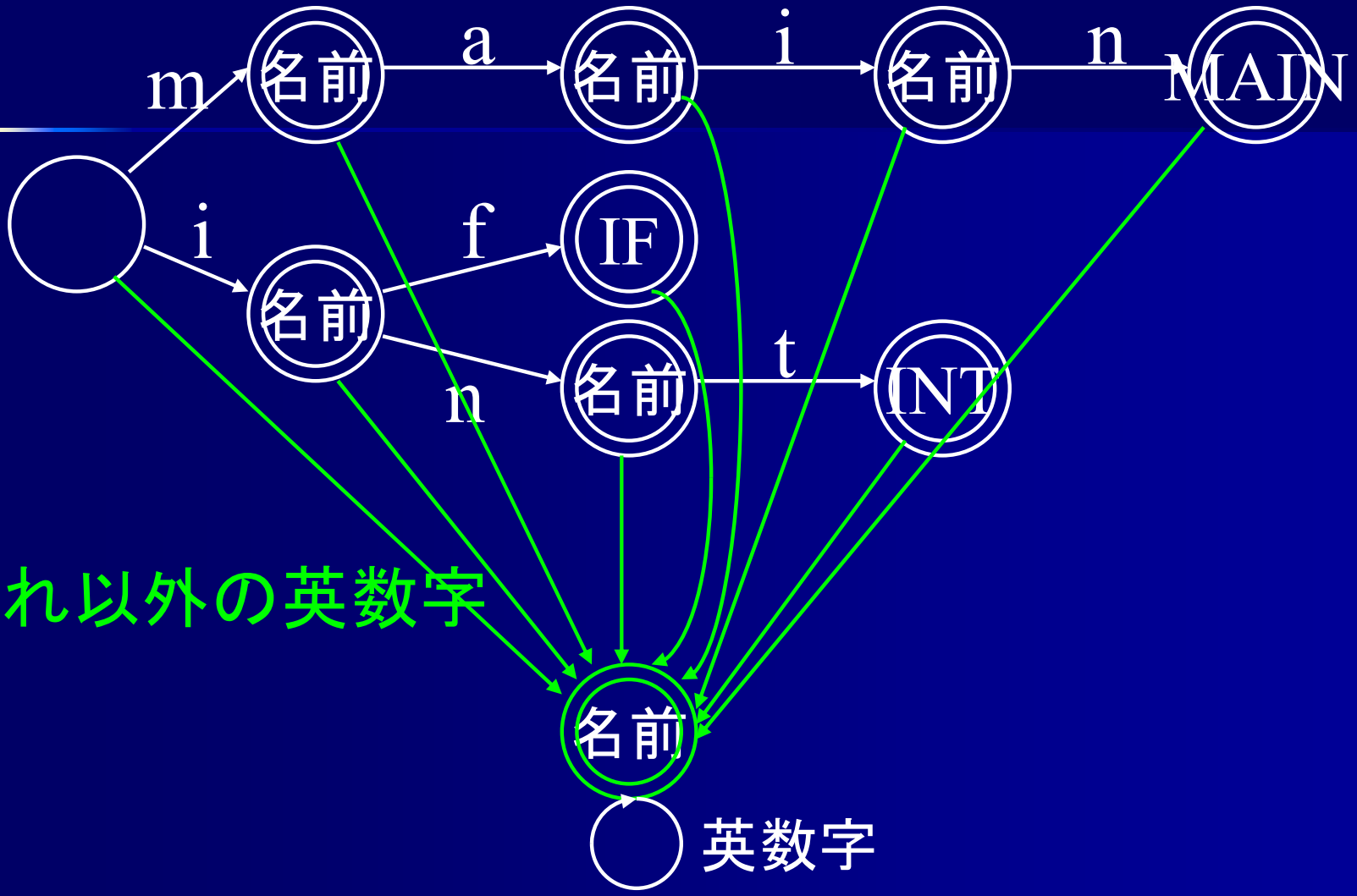
- 字句解析部のプログラム　　(大文字のみの場合)

```
char currentChar;          // 現在位置の文字
char nextChar();           // 次の文字を読み込み1文字進める
char lookAhead();          // 次の位置の文字の先読み
```

```
String name = “”;
if (Character.isUpperCase (currentChar)) {/* 1文字目が英字 */
    String name += currentChar;                    /* 文字を記憶 */
    while (Character.isUpperCase (lookAhead()) {
        currentChar = nextChar();              /* 次の文字を読み込む */
        name += currentChar;                           /* 文字を記憶 */
    }
    token = /* 変数名のトークン生成 */
}
```

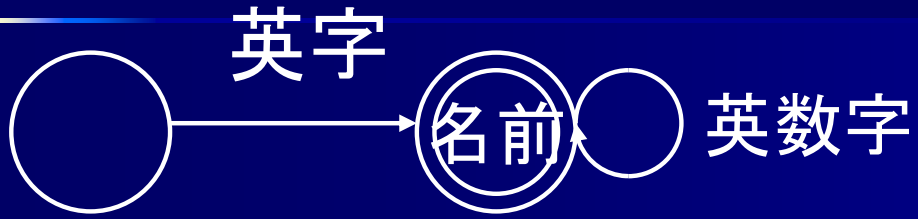
予約語はどうする？

変数名と予約語の解析



これでできなくはないが
状態数が膨大になり非常に面倒

変数名と予約語の解析



1. ひとまず全て名前候補として単語に切る
2. 予約語と一致する単語は予約語とする
3. 予約語と一致しなかった単語は変数名とする

I　if　IF　in　inc　int　inter

変数名,予約語解析部のプログラム

```
String name = “”;
if (currentChar ∈ {a…z, A…Z, _}) {
    String name += /* currentCharを結合する */
    while (lookAhead() ∈ {a…z, A…Z, _, 0…9}) {
        currentChar = /* 次の文字を読み込む */
        name += /* currentCharを結合する */
    } /* 名前候補として最後まで name に取り込む */
    if (name が “main” と一致)
        token = new Token (MAIN);        /* 予約語 main と判定*/
    else if (name が “if” と一致)
        token = new Token (IF);                /* 予約語 if と判定*/
    else token = /* 変数名のトークン生成   */
}
```

ここまで共通

字句解析時のエラー処理

- トークンとして切り出せなかった
  ⇒字句解析エラー

```
if (“0” を切り出した場合) token = new Token (INTEGER, 0);
else if (“++” を切り出した場合) token = new Token (INC);
else if (“+” を切り出した場合) token = new Token (ADD);
else if (“if” を切り出した場合) token = new Token (IF);
                    :       /* 以下各トークンに対する処理を else if で並べる */
else syntaxError(); /* どのトークンとも一致しなかった場合はエラー */
```

エラー検出時はエラーメッセージを表示して停止

字句解析時のエラー処理

エラー検出時はエラーメッセージを表示して停止

```
private void syntaxError () {
    String err_mes = analyzeAt() + “でエラー検出”;
                    /* analyzeAt() を用いてエラーメッセージ作成 */
    System.out.println (err_mes);  /* エラーメッセージ表示 */
    closeFile ();                  /* 入力ファイルを閉じる */
    System.exit (0);               /* プロクラム停止 */
}
```

エラー箇所がユーザに分かり易いエラーメッセージを作成する

ファイル末到達時の処理

- ファイル末到達時は Token (EOF) を返す

ファイル末を示す文字

```
if (currntChar == '¥0') {          /* ファイル末の場合 */
    token = new Token (EOF);  /* 特殊なトークン EOF */
```

```
Token nextToken () {
  Token token = null;      /* ダミーの初期値 */
  do {
     currentChar = nextChar(); /* 1文字目を読み込む */
  while (currentChar == ‘ ’);    /* 空白以外を読み込むまで */

  if (currentChar == ‘+’) {
     if (lookAhead() == ‘+’) {
        nextChar();
        token = new Token (INC);
     } else token = newToken (ADD);
  }
  else if
                  :        /* 以下各トークンに対する処理を else if で並べる */
  else syntaxError();          /* どのトークンとも一致しなければエラー */
  return token;
}
```

コメント

- コメントの処理
  - 戦略1：空白と同様に処理
  - 戦略2：nextToken() を再帰呼び出しして
    次のトークンを読む

以下では @ ... @ をコメントとする

```
if (ans >= 123 ) @ コメント @
 output @ コメント @ (‘a’);
```

コメント：空白と同様に処理

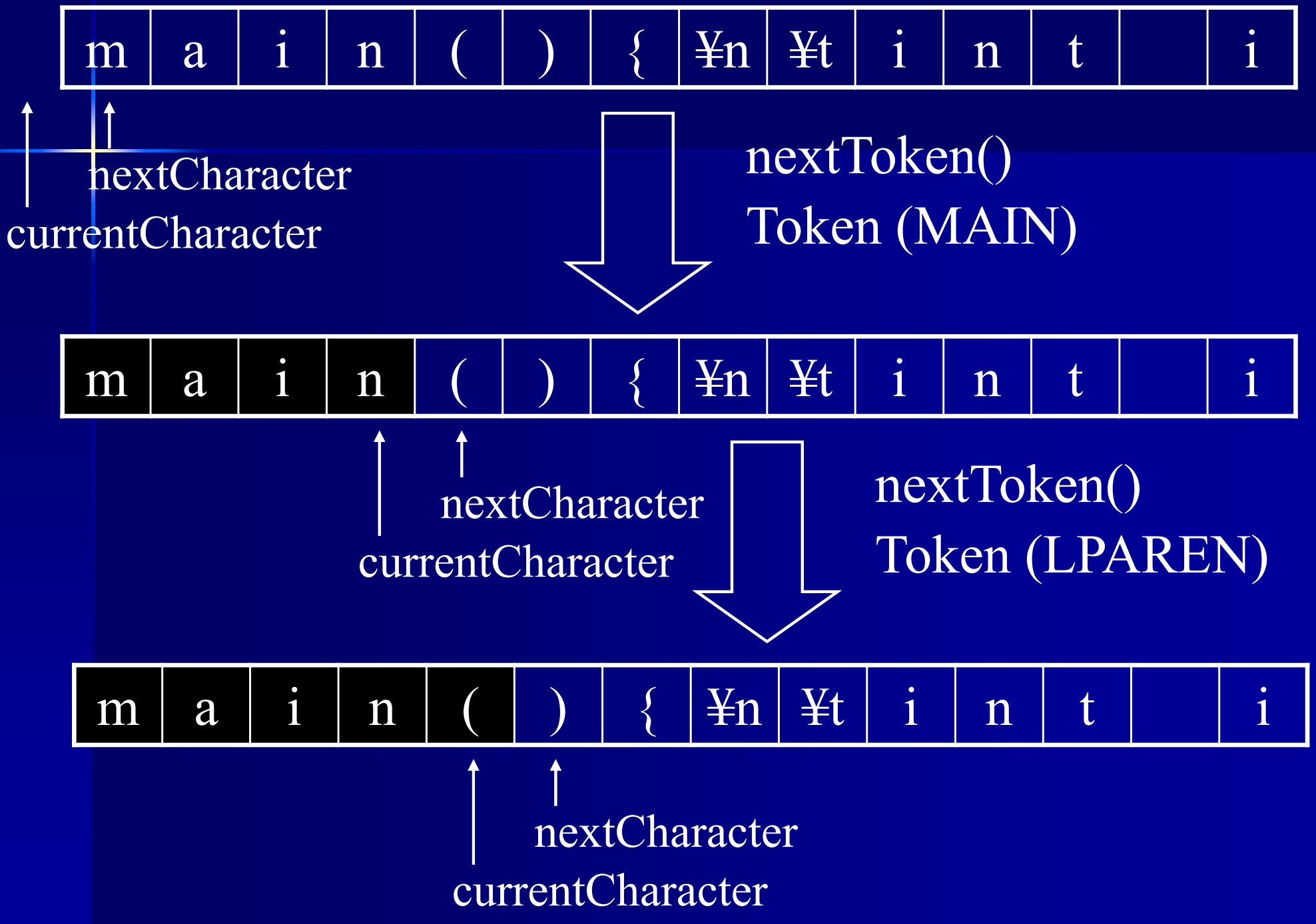
```
while (currentChar == ‘ ’ || currentChar == ‘@’) {
  if (currentChar == ‘ ’)          // 空白の場合
    currentChar = nextChar();      // 空白を読み飛ばす
  else {                           // @ の場合
    currentChar = nextChar();      // 1つめの @ を読み飛ばす
    while (currentChar != ‘@’)
      currntChar = nextChar();     // @以外を読み飛ばす
    currntChar = nextChar();       // 2つめの @ を読み飛ばす
  }
}
```

コメント：再帰で次のトークンを読む

```
if (currentChar == '@') {
   currentChar = nextChar();    // 1つめの @ を読み飛ばす
   while (currentChar != '@')
     currntChar = nextChar();   // @以外を読み飛ばす
   currntChar = nextChar();     // 2つめの @ を読み飛ばす
   token = nextToken();         // 再帰で次のトークンを読む
}
```

/* ... */ や // ... (改行) にする方法は
各自で考えること

nextToken()


m a i n ( ) { ¥n ¥t i n t i
nextCharacter
currentCharacter
nextToken()
Token (MAIN)
m a i n ( ) { ¥n ¥t i n t i
nextCharacter
currentCharacter
nextToken()
Token (LPAREN)
m a i n ( ) { ¥n ¥t i n t i
nextCharacter
currentCharacter

先読みを用いない字句解析

- 先読み文字が1文字の場合

⇒ 先読み無しでも字句解析可能

```
if (currentChar == '+') {
    currentChar = nextChar();  /* 読み進める */
    if (currentChar == '+') {    /* 現在の文字で判定 */
        nextChar();
        token = new Token (INC);
    } else token = newToken (ADD);
}
```

先読みを用いない字句解析

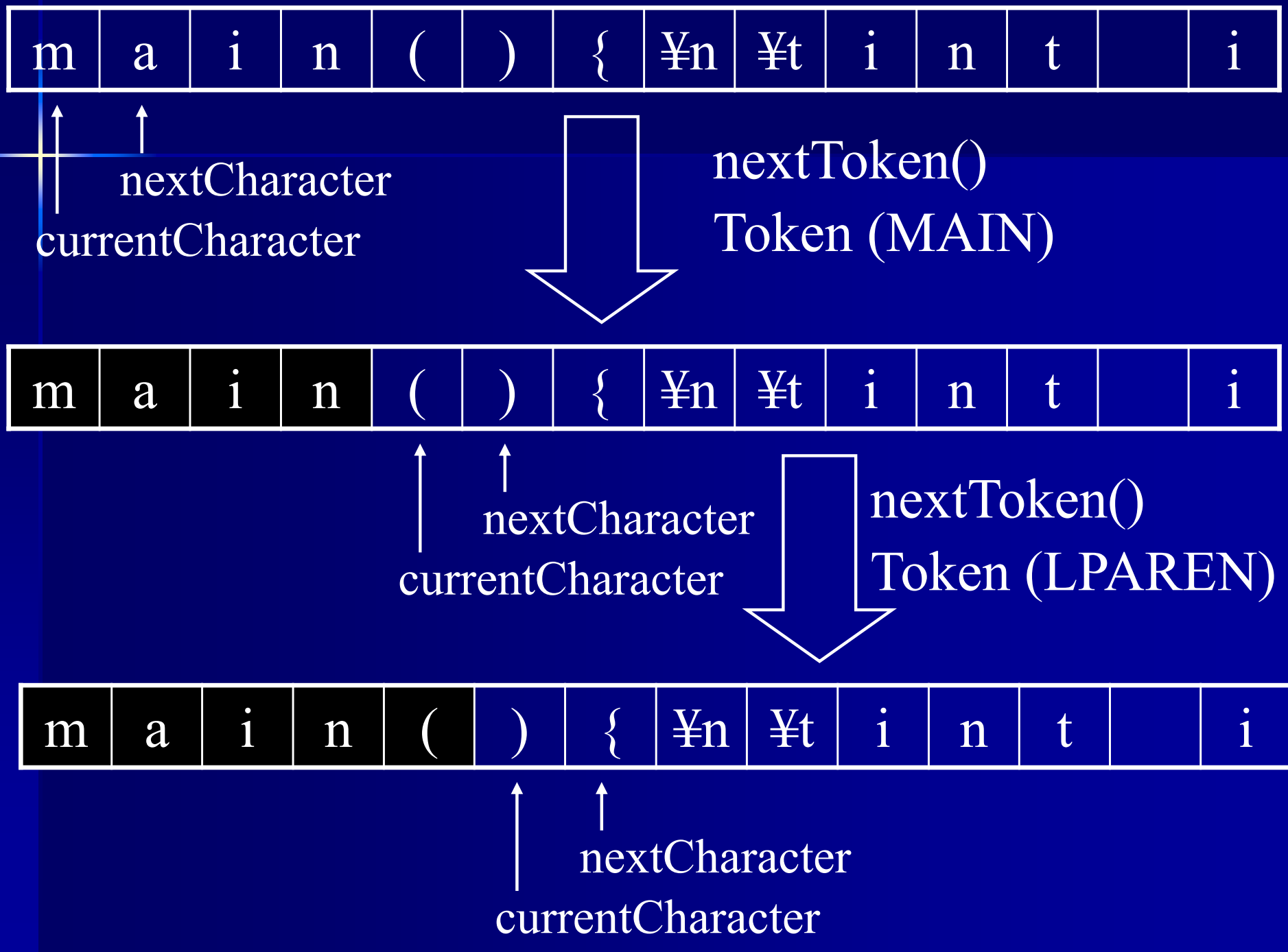
先読みあり

```
Token nextToken() {
  Token token;

 /* 1文字目は未読 */
  currentChar = nextChar(); // 1文字目

  if (currentChar == ‘+’) {
     if (lookAhead() == ‘+’) {
        nextChar();  // 2文字目
        token = new Token (INC);
     } else token = newToken (ADD);
  }

  return token;
}
```

先読み無し

```
Token nextToken() {
  Token token;

 /* 1文字目は既読 */

  if (currentChar == ‘+’) {
     currentChar = nextChar(); // 2文字目
     if (currentChar == ‘+’) {
        nextChar();  // 3文字目
        token = new Token (INC);
     } else token = newToken (ADD);
  }

  return token;
}
```

nextToken() (先読み無しの場合)



先読みあり

do-while文

```
Token nextToken () {
  Token token = null;
  do {
    currentChar = nextChar();     /* 1文字目を読み込む */
  } while (currentChar == ' ');   /* 空白を読み飛ばす */

  if (currentChar == '+') {
    if (lookAhead() == '+') {     /* 先読み文字で判定 */
      nextChar();                 /* 読み進める(2文字目) */
      token = new Token (INC);
    } else token = newToken (ADD);
  }
  return token;
}
```

先読み無し

while文

```
Token nextToken () {
  Token token = null;
  while (currentChar == ‘ ’)      /* 空白を読み飛ばす */
    currentChar = nextChar();

  if (currentChar == ‘+’) {
    currentChar = nextChar();  /* 読み進める(2文字目) */
    if (currentChar == ‘+’) {    /* 現在の文字で判定 */
      nextChar();                /* 読み進める(3文字目) */
      token = new Token (INC);
    } else token = newToken (ADD);
  }
  return token;
}
```

行単位での字句解析

- 文字単位ではなく行単位で解析する

```
String line;              // 現在解析中の行
String nextLine();  // 入力ファイルから次の行を読み込む
```

```
if (line.charAt (0) == '(') {   /* 先頭の文字で判定 */
    token = new Token (LPREN);
    line = line.substring (1);  /* line の先頭の1文字を削る */
}
```

記号解析部分のプログラム

- 字句解析部のプログラム

```
String line;                    // 現在解析中の行
```

例 +, ++ の解析

```
if (line.charAt (0) == '+' && line.charAt (1) == '+') {
                        /* 1文字目と2文字目の論理積で判定 */
    token = new Token (INC);    /* ++ と判定 */
    line = line.substring (2); /* line の先頭の2文字を削る */
} else if (line.charAt (0) == '+') {
    token = new Token (ADD);   /* + と判定 */
    line = line.substring (1);  /* line の先頭の1文字を削る */
}
```

変数名解析部分のプログラム

- 字句解析部のプログラム

```
String line;                    // 現在解析中の行
```

例 変数名の解析(手法1)

```
if (line.charAt (0) ∈ {a…z, A…Z, _}) { /* 1文字目が英字 */
    int i =1;
    while (line.charAt (i) ∈ {a…z, A…Z, _, 0…9}) ++i;
                        /* 連続する英数字の文字数をカウントする */
    name = line.substring (0, i);
                        /* line の先頭から i 文字目までが変数名 */
    line = line.substring (i);    /* line を name の文字数文削る */
    token = new Token (NAME, name);      /* 変数名と判定*/
}
```

変数名解析部分のプログラム

- 字句解析部のプログラム

```
String line;                    // 現在解析中の行
```

例 変数名の解析(手法2)

```
if (line.charAt (0) ∈ {a…z, A…Z, _}) { /* 1文字目が英字 */
    String[] subline = line.split ("[^a-zA-Z_0-9]");
                                        /* line を英数字以外で分割 */
    String name = subline[0];       /* 分割した先頭部分が名前 */
    line = line.substring (name.length());
                                    /* line を name の文字数文削る */
    token = new Token (NAME, name);      /* 変数名と判定*/
}
```

行単位での字句解析

```
Token nextToken () {
  Token token = null;

  while (line.charAt (0) == '¥n')
      line = readNextLine();    /* 行末なら次の行を読み込む */

  if (line.charAt (0) == '+' && line.charAt (1) == '+') {
      token = new Token (INC);        /* ++ と判定 */
      line = line.subset (2);         /* 先頭の2文字を削る */
  } else if (line.charAt (0) == '+') {
      token = new Token (ADD);        /* + と判定 */
      line = line.subset (1);         /* 先頭の1文字を削る */
  }
  return token;
}
```

行単位での字句解析の利点

- 2文字以上先読み可能
- 文字列操作用のメソッドを利用可能