コンパイラ

第9回 コード生成
― コード生成プログラムの作成 ―

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

コード生成プログラム

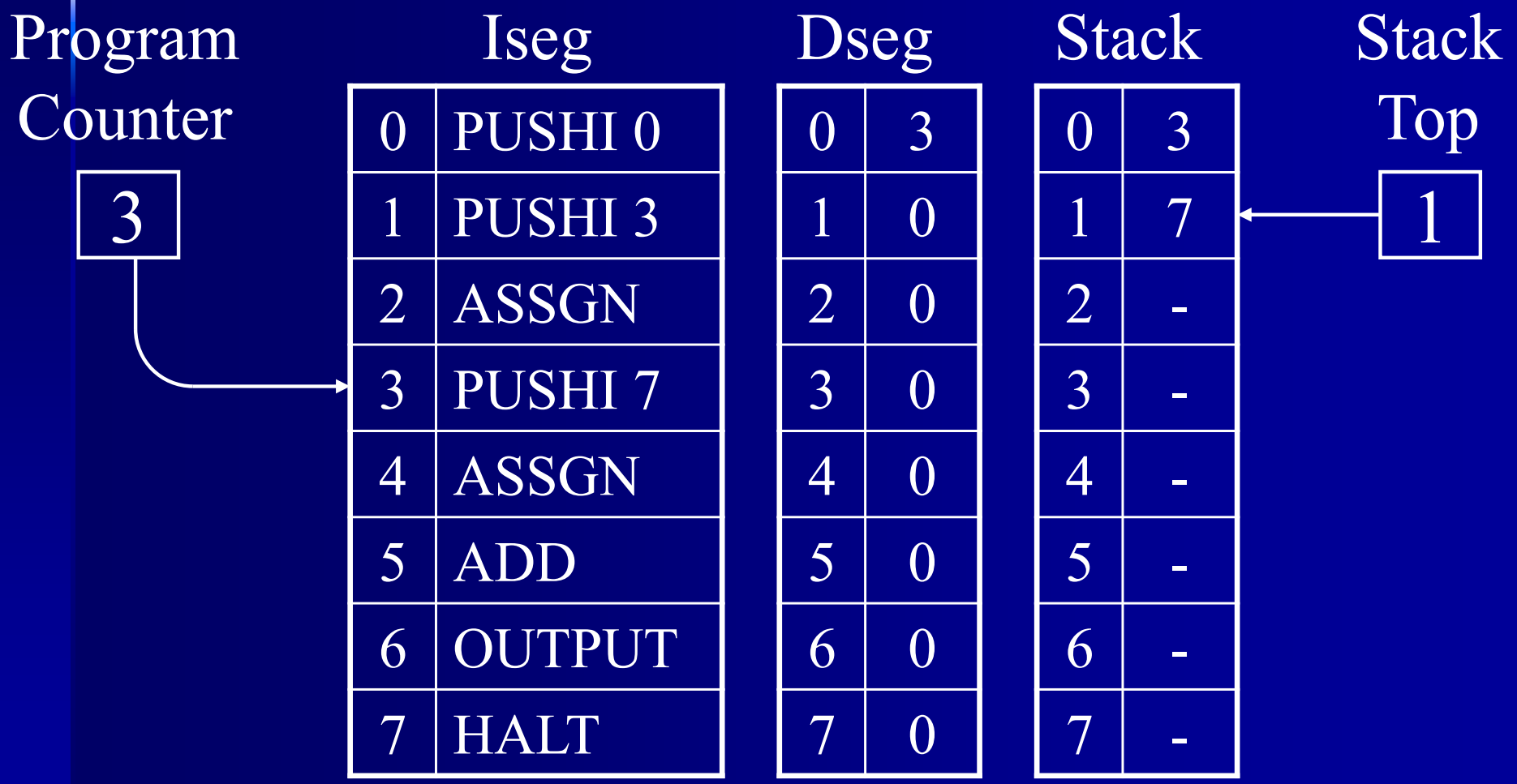
- Kc.java の各非終端記号解析用メソッドにコード生成を加える
  - 例 :非終端記号 <A> のコード生成

```
void parse<A> () {
  if (トークン列が<A>のマクロ構文と合致) {
     <A>のコード生成;
    /* VSM アセンブラのコードを Iseg に積む */
  } else syntaxError();
    /* マクロ構文と一致しなかった場合はエラー */
}
```

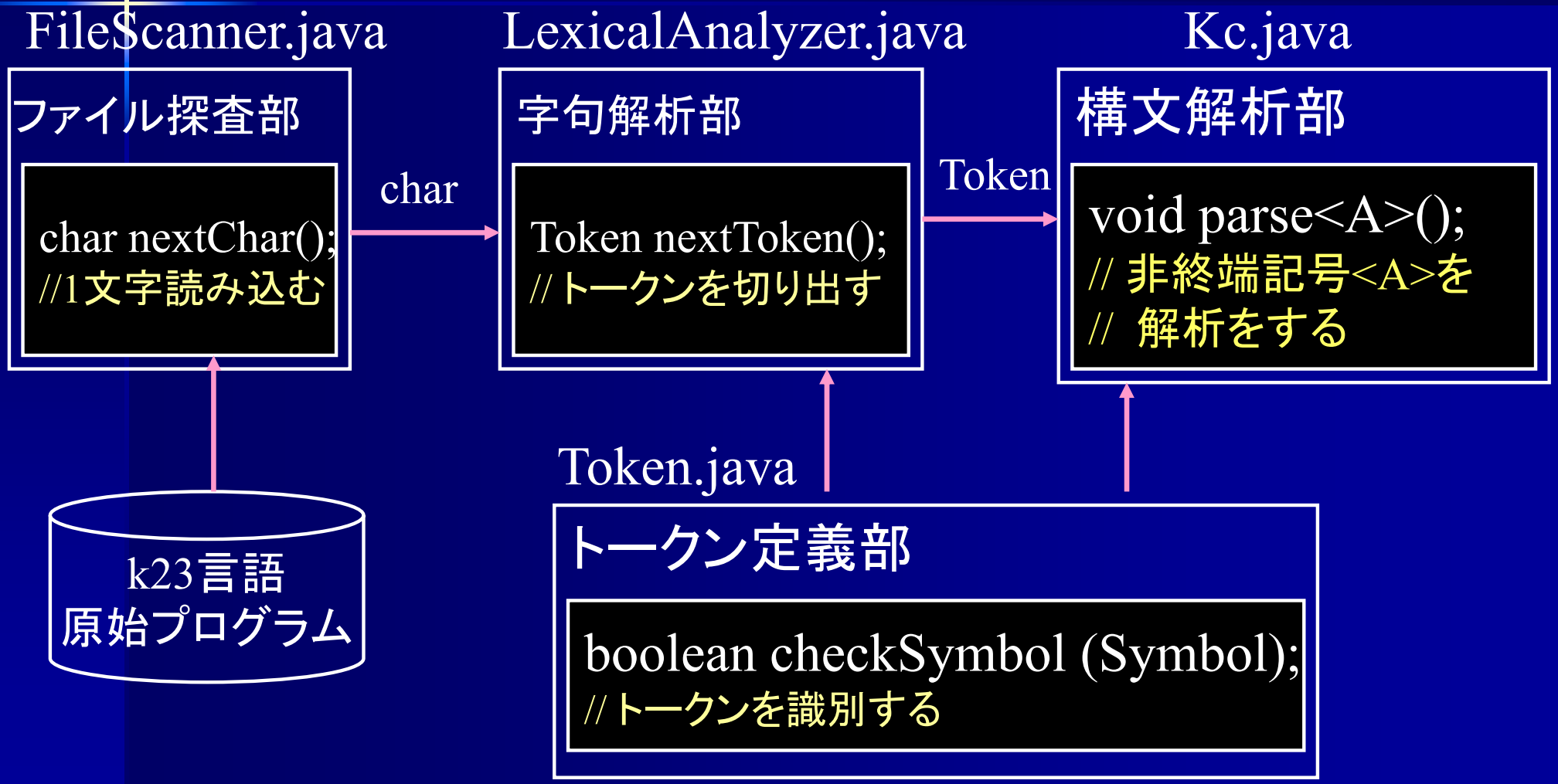
スタックマシン (stack machine)

- スタックマシン
  - Iseg[] : アセンブラプログラムを格納
  - Dseg[] : 実行中の変数値を格納
  - Stack[] : スタック(作業場所)
  - Program Counter : 現在の Iseg の実行位置
  - Stack Top : 現在のスタックの操作位置

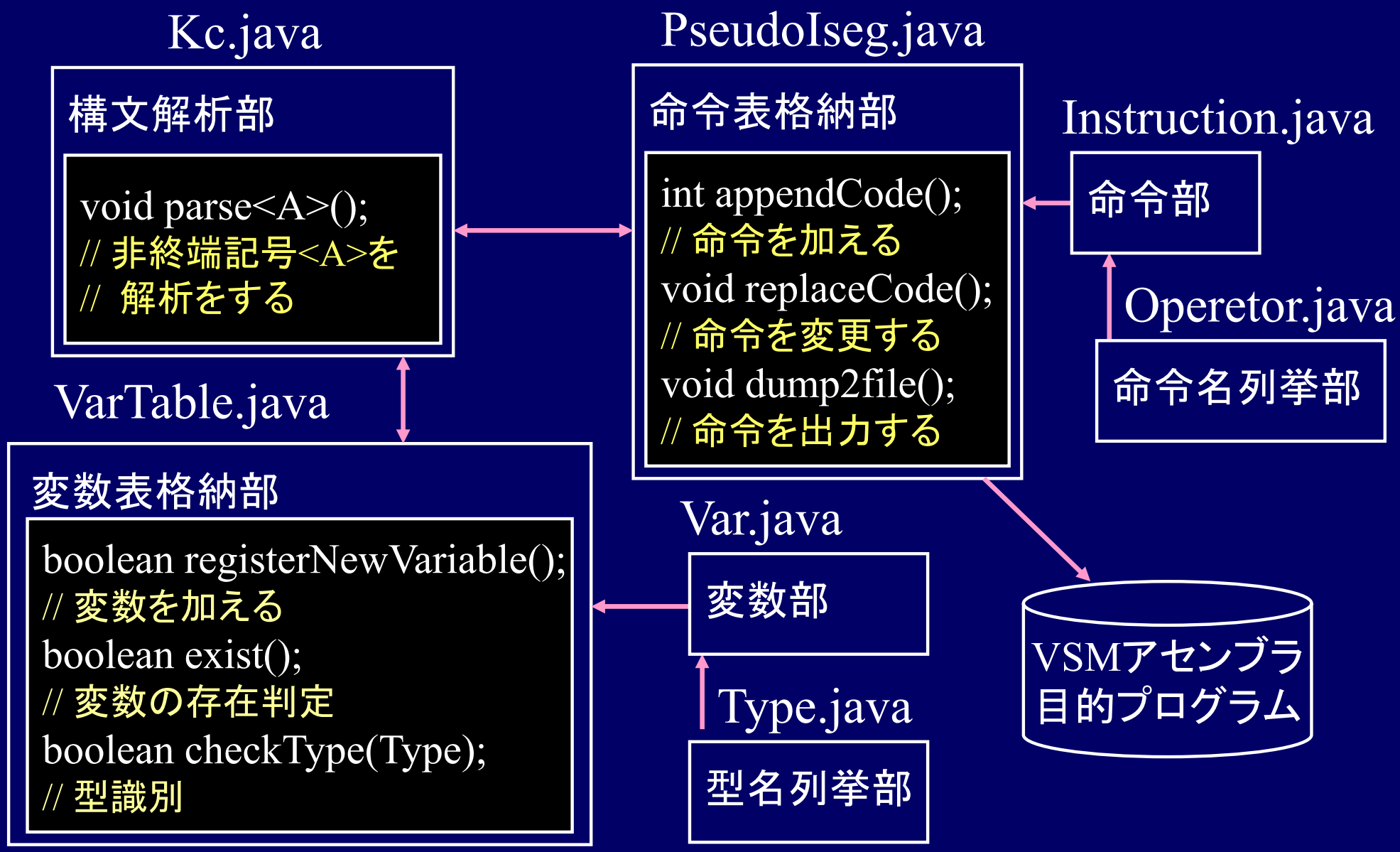
スタックマシン (stack machine)



プログラムの構造(構文解析系)



プログラムの構造(コード生成系)



PseudoIseg クラス

Kc			命令表格納部
pIseg	: ArrayList <Instruction>		# 命令表
pIsegPtr	: int		# カウンタ
PseudoIseg ()			# コンストラクタ
- setI (opcode : Operator, flag : int, addr : int)		: int	# 命令を格納
appendCode (opcode : Operator, addr : int)		: int	# 命令を格納
appendCode (opcode : Operator)		: int	# 命令を格納
getLastCodeAddress()		: int	# 命令末尾位置
dump()		: void	# 命令表表示
dump2file ()		: void	# 命令表出力
dump2file (outputFileName : String)		: void	# 命令表出力
replaceCode (ptr : int, op : Operator)		: void	# 命令を変更
replaceCode (ptr : int, addr : int)		: void	# 命令を変更
checkOperator (ptr : int, op : Operator)		: boolean	# 命令の一致判定
getOperand (ptr : int)		: int	# オペランドを得る
removeLastCode ()		: void	# 末尾の命令を削除

Iseg への命令の追加

- Iseg への命令の追加は
  PseudoIseg.appendCode (Operator, int)
  PseudoIseg.appendCode (Operator) を使用

```
/** @return 追加した命令の番地 */
int appendCode (Operator op, int addr)
int appendCode (Operator op)
```

例 : Iseg に PUSHI 10 , ADD を追加

```
iseg.appendCode (Operator.PUSHI, 10);
iseg.appendCode (Operator.ADD);
```

Iseg への命令の追加

例 :

```
iseg.appendCode (Operator.PUSHI, 10);
iseg.appendCode (Operator.PUSH, 0);
iseg.appendCode (Operator.ASSGN);
iseg.appendCode (Operator.REMOVE);
```

Iseg

```
0  PUSHI  10
1  PUSH    0
2  ASSGN
3  REMOVE
```

Iseg の命令の変更

- Iseg の命令の変更は

  PseudoIseg.replaceCode (int, Operator)
  PseudoIseg.replaceCode (int, int) を使用

```
void replaceCode (int ptr, Operator op)
void replaceCode (int ptr, int addr)
```

例 : Iseg の 20 番地の命令を に JUMP に変更

```
iseg.replaceCode (20, Operator.JUMP);
```

Iseg の15 番地のオペランドを 25 に変更

```
iseg.replaceCode (15, 25);
```

Iseg の命令の変更

Iseg

```
10  COMP
11  BGT    14
12  PUSHI  0
13  JUMP   15
14  PUSHI  1
15  BEQ
```

```
10  COMP
11  BLE    14
12  PUSHI  0
13  JUMP   15
14  PUSHI  1
15  BEQ
```

```
10  COMP
11  BLE    14
12  PUSHI  0
13  JUMP   15
14  PUSHI  1
15  BEQ    30
```

replaceCode (11, BLE);

replaceCode (15, 30);

非終端記号 <A> のコード生成

- <A> ::= α (∈ (N∪T)*) の解析
  1. <A> ::= ε のとき
     コードは生成しない
  2. <A> ::= “a” (∈ T) のとき

```
if (token == “a”) {
    token = nextToken();
    “a” に対応する命令のコード(もしあれば);
} else syntaxError();
```

非終端記号 <A> のコード生成

- <A> ::= α (∈ (N∪T)*) のコード生成

  3. <A> ::= <B> (∈N) のとき

     1. ε∉First (<B>) のとき

```
if (token ∈ First (<B>)) parse<B>();
   else syntaxError();
```

     2. ε∈First (<B>) のとき

```
if (token ∈ (First (<B>)-ε)) parse<B>();
```

<B> のコード生成は parse<B> に任せる

非終端記号 <A> のコード生成

- <A> ::= α (∈ (N∪T)*) のコード生成

4. <A> ::= $\beta_1 \mid \beta_2 \mid \beta_3$ のとき

```
if (token ∈ First (β1)) {
    β1のコード;
} else if (token ∈ First (β2)) {
    β2のコード;
} else if (token ∈ First (β3)) {
    β3のコード;
} else syntaxError();
```

非終端記号 <A> のコード生成

- <A> ::= α (∈ (N∪T)*) のコード生成

  5. <A> ::= $\beta_1\beta_2\beta_3$ のとき

```
β1のコード;
β2のコード;
β3のコード;
```

非終端記号 <A> のコード生成

- <A> ::= α (∈ (N∪T)*) のコード生成

  6. <A> ::= {β} のとき

```
while (token ∈ First (β)) {
    βのコード;
}
```

非終端記号 <A> のコード生成

- <A> ::= α (∈ (N∪T)*) のコード生成

  7. <A> ::= [β] のとき

```
if (token ∈ First (β)) {
    βのコード;
}
```

else syntaxError(); は付けない

非終端記号 <A> (括弧) の解析

- <A> ::= α (∈ (N∪T)*) のコード生成

8. <A> ::= (β) のとき

```
βのコード;
```

<Unsigned>のコード生成

- 各終端記号に対応した命令を Iseg に積む

終端記号	命令
INTEGER	PUSHI
CHARACTER	PUSHI
NAME	PUSHI | PUSH
NAME “[” <Exp> “]”	PUSHI parseExp() ADD [LOAD]
“inputint”	INPUT
“inputchar”	INPUTC
“(” <Exp> “)”	parseExp()

整数, 文字のコード生成

- 整数の場合

```
appendCode (PUSHI, 整数値);
```

- 文字の場合

```
appendCode (PUSHI, 文字コード);
```

整数, 文字は共に PUSHI

整数値, 文字コードは Token.getIntValue() で得る

コード生成プログラム

- <Unsigned> ::= INTEGER | CHARACTER の場合

```
void parseUnsigned () {
    if (token == INTEGER) {
        int value = // tokenから整数値を得る
        token = nextToken();
        appendCode (PUSHI, value);
    } else (token == CHARACTER) {
        int charCode = // tokenから文字コードを得る
        token = nextToken();
        appendCode (PUSHI, charCode);
    } else if ...
```

整数と文字は
同一の処理

コード生成プログラム

- <Unsigned> ::= INTEGER | CHARACTER の場合

整数と文字は同一処理でOK

```
void parseUnsigned () {
    if (token == INTEGER || token == CHARACTER) {
        int value = // tokenから整数値or 文字コードを得る
        token = nextToken();
        appendCode (PUSHI, value);
    } else if ...
```

コード生成プログラム

- <Unsigned> ::= “inputint” | “inputchar” の場合

```
void parseUnsigned () {
        :
    else if (token == “inputint”) {
            token = nextToken();
            appendCode (INPUT);
    } else if (token == “inputchar”) {
            token = nextToken();
            appendCode (INPUTC);
    } else if ...
```

コード生成プログラム

- <Unsigned> ::= “(” <Exp> “)” の場合)

```
void parseUnsigned () {
        :
    else if (token == “(”) {
            token = nextToken();
            if (token ∈ First (<Exp>)) parseExp();
                else syntaxError();
            if (token == “)”) token = nextToken;
                else syntaxError();
    } else if ...
```

<Exp> のコード生成は parseExp() に任せる

演算のコード生成

- 演算のアセンブラコード
  - 演算子に対応したコードを最後に置く

例 : <Exp> ::= $\langle Term \rangle_1$ “+” $\langle Term \rangle_2$

<Term> ::= $\langle Factor \rangle_1$ “*” $\langle Factor \rangle_2$

<Exp>

$\langle Term \rangle_1$ のコード(右辺値) $\langle Term \rangle_2$ のコード(右辺値) ADD

<Term>

$\langle Factor \rangle_1$ のコード(右辺値) $\langle Factor \rangle_2$ のコード(右辺値) MUL

コード生成プログラム

- <Term> ::= <Factor> “*” <Factor> の場合

```
void parseTerm () {
    if (token ∈ First (<Factor>)) parseFactor();
        else syntaxError();
    if (token == “*”) token = nextToken();
        else syntaxError();
    if (token ∈ First (<Factor>)) parseFactor();
        else syntaxError();
    appendCode (MUL);
}
```

<Factor> のコードが詰まれる

最後に演算子のコードを詰む

結合性とコード

- a + b + c + d のコード

((a + b) + c) + d ?　　a + (b + (c + d)) ?

左結合的　　右結合的

a b + c + d +　　a b c d + + +

```
PUSH a のアドレス
PUSH b のアドレス
ADD
PUSH c のアドレス
ADD
PUSH d のアドレス
ADD
```

```
PUSH a のアドレス
PUSH b のアドレス
PUSH c のアドレス
PUSH d のアドレス
ADD
ADD
ADD
```

コード生成時に
結合性の
確認が必要

コード生成プログラム

- <Term> ::= <Factor> { “*” <Factor> } の場合 (左結合的)

```
void parseTerm () {
    if (token ∈ First (<Factor>)) parseFactor();
        else syntaxError();
    while (token == “*”) {
        token = nextToken();
        if (token ∈ First (<Factor>)) parseFactor();
            else syntaxError();
        appendCode (MUL);
    }
}
```

コード生成プログラム

- <Term> ::= <Factor> { “*” <Factor> } の場合 (右結合的)

```
void parseTerm () {
    int n=0;                              // “*” の個数カウント用
    if (token ∈ First (<Factor>)) parseFactor();
        else syntaxError();
    while (token == “*”) {
        ++n;                              // “*” の個数をカウントする
        token = nextToken();
        if (token ∈ First (<Factor>)) parseFactor();
            else syntaxError();
    }
    for (int i=0; i<n; ++i) appendCode (MUL);
}
```

コード生成プログラム

- <Factor> ::= <Unsigned> | “-” <Factor> の場合

```
void parseFactor () {
    if (token ∈ First (<Unsigned>)) {
        parseUnsigned();
    } else  if (token == “-”) {
        token = nextToken();
        if (token ∈ First (<Factor>)) parseFactor();
            else syntaxError();
        appendCode (CSIGN);
    } else syntaxError();
}
```

<Unsigned> のコードが詰まれる

単項演算子も同様に最後にコードを詰む

コード生成プログラム

- <Term> ::= <Factor> {(“*” | “/”) <Factor> } の場合

```
void parseTerm () {
  if (token ∈ First (<Factor>) parseFactor(); else syntaxError();
  while (token == “*” || token == “/”) {
    if (token == “*” ) { // “*” の場合
      token = nextToken();
      if (token ∈ First (<Factor>)) parseFactor(); else syntaxError();
      appendCode (MUL);
    } else {                    // “/” の場合
      token = nextToken();
      if (token ∈ First (<Factor>)) parseFactor(); else syntaxError();
      appendCode (DIV);
    }
  }
}
```

コード生成プログラム

- <Term> ::= <Factor> {(“*” | “/”) <Factor> } の場合

```
void parseTerm () {
    if (token ∈ First (<Factor>) parseFactor();
        else syntaxError();
    while (token == “*” || token == “/”) {
        Symbol op = token.getSymbol();
        token = nextToken();
        if (token ∈ First (<Factor>)) parseFactor();
            else syntaxError();
        if (op == Symbol.MUL) appendCode (MUL);
            else appendCode (DIV);
    }
}
```

演算子を記憶

文のコード生成

- <St> ::= <If_St>
  | <While_St>
  | <Outputint_St>
  | <Outputchar_St>
  | <Exp_St>
  | “{” { <St> } “}”
  | “;”

このコード生成は
各parse<A>()に
任せる

ここは生成規則 6.に
従ってコード生成

“;” のコードは？

コード生成プログラム

```
void parseSt () {
    switch (token) {
        case First (<IfSt>)) :              parseIfSt();              break;
        case First (<WhileSt>)) :           parseWhileSt();           break;
        case First (<OutputintSt>)) :   parseOutputintSt();   break;
        case First (<OutputcharSt>)) :parseOutputcharSt(); break;
        case First (<ExpSt>)) :             parseExpSt();             break;
        case “{” :          “{” { <St> } “}” のコード生成;          break;
        case “;” :          空文のコード生成;                       break;
        default : syntaxError();
    }
}
```

空文のコード生成

<St> ::= “;” の場合

空文 = 何もしない = コード無し

```
void parseSt () {
    switch (token) {
            :
        case “;” : token = nextToken(); break;
            :
    }
}
```

トークンを読み飛ばすだけ

出力文のコード生成

<OuputintSt> ::= “outputint” “(” <Exp> “)” “;”

<Exp> のコード (右辺値)
OUTPUT
OUTPUTLN

<OuputcharSt> ::= “outputchar” “(” <Exp> “)” “;”

<Exp> のコード (右辺値)
OUTPUTC
OUTPUTLN

式文のコード生成

<ExpSt> ::= <Exp> “;” の場合

<Exp> のコード (右辺値)
REMOVE

スタックトップに残った
式の評価値を削除

```
void parseExpSt () {
    if (token ∈ First (<Exp>)) parseExp();
        else syntaxError();
    if (token == “;”) appendCode (REMOVE);
        else syntaxError();
}
```

“;” が来れば式終了 ⇒ 式の評価値はもう不要

変数宣言部のコード生成

- <Decl> ::= “int” NAME [“=” <Const>] “;”
- <Const> ::= [“-”] INTEGER | CHARACTER

初期値無しの変数/配列宣言

コード無し(変数表への登録のみ)

初期値ありの変数/配列宣言

変数表への登録

Dseg へのデータ代入コード生成

```
PUSHI <Const>の値
POP    NAMEの番地
```

コード生成には番地が必要

変数表への挿入

- 変数表への挿入は
  VarTable.registerNewVariable (Type, String, int) を使用

```
/** @ return 変数 name を登録できたか？ */
boolean registerNewVariable
      (Type type, String name, int size)
```

例：int i, a[5];

```
registerNewVariable (Type.INT, “i”, 1);
registerNewVariable (Type.ARRAYOFINT, “a”, 5);
```

変数の番地

- 変数の番地

  VarTable.getAddress (String) を使用

```
/** @ return 変数 name の番地 */
int getAddress (String name)
```

例 : 変数 i の番地

```
varTable.getAddress (“i”)
```

コード生成プログラム

- <Decl> ::= “int” NAME [“=” <Const>] “;” の場合

```
void parseVarDecl () {
    if (token == “int”) token = nextToken();
        else syntaxError();
    if (token == NAME) {
        String name = // tokenから変数名を得る
        token = nextToken();
    } else syntaxError();
    if (exist (name)) syntaxError ();       // 二重登録チェック
                                :
```

ここまでは初期値の有無に関係無く共通

コード生成プログラム

- <Decl> ::= “int” NAME [“=” <Const>] “;” の場合

```
if (token == “=”) { // 初期値代入がある場合
    token = nextToken();
    if (token ∈ First (<Const>)) parseConst();
       else syntaxError();
    resigterNewVariable (INT, name, 1);   // 変数表に登録
    int address = // 変数表を参照してnameの番地を得る
    appendCode (PUSHI, <Const>の値); // 初期値を積む
    appendCode (POP, address);              // Dseg に代入
} else resigterNewVariable (INT, name, 1);
if (token == “;”) nextToken; else syntaxError();
}
```

コード生成プログラム

- <Const> ::= [“-”] INTEGER | CHARACTER

```
/** @return 定数の値 */
int parseConst () {
    if (token == INTEGER) {
        int value = // tokenから整数値を得る
        token = nextToken();
        return value; // 整数値を返す
    } else if (token == “-”) {
        “-” INTEGER の解析; return 負の整数値;
    } else if (token == CHARCTER) {
        CHARACTER の解析; return 文字コード;
    } else syntaxError();
}
```

コード生成プログラム

- <Decl> ::= “int” NAME [“=” <Const>] “;” の場合

```
if (token == “=”) { // 初期値代入がある場合
   token = nextToken();
   int value;
   if (token ∈ First (<Const>)) value = parseConst();
      else syntaxError();
   resigterNewVariable (INT, name, 1);   // 変数表に登録
   int address = // 変数表を参照してnameの番地を得る
   appendCode (PUSHI, value);          // 初期値を積む
   appendCode (POP, address);             // Dseg に代入
} else resigterNewVariable (INT, name, 1);
if (token == “;”) nextToken; else syntaxError();
}
```

変数宣言部(配列)のコード生成

- <Decl> ::= “int” (NAME [“=” <Const>] “;”
  | NAME “[” INTEGER “]” “;”
  | NAME “[” “]” “=” “{” <Const> { “,” <Const> } “}” “;”)

例 : int a[] = { 10, 20, 30 };

名前	型	サイズ	番地
a	int []	3	5

```
PUSHI 10
POP    a[0]の番地
PUSHI 20
POP    a[1]の番地
PUSHI 30
POP    a[2]の番地
```

```
PUSHI 10
POP    5
PUSHI 20
POP    6
PUSHI 30
POP    7
```

番地を
1ずつ増加

変数宣言部(配列)のコード生成

- 変数表への登録にはサイズが必要
- コード生成には番地が必要

```
int a[] = { 10, 20, 30 };
```

サイズ未定

ここを読んでいる時点では
まだコード生成できない

“}” まで読めば
サイズ確定

“}” まで読んだ時点で変数表に登録する

⇩

各初期値は一旦作業用 ArrayList に保管

```
if (token == “int”) token = nextToken();
    else syntaxError();
 if (token == NAME) token = nextToken();
    else syntaxError();
 if (token == “[”) { // 配列の場合
    token = nextToken();
    if (token == INTEGER) { // 初期値無しの配列
        “[” INTEGER “]” “;” の解析
        変数表に登録
    } else if (token == “]”) {   // 初期値有りの配列
        “[” “]” “=” “{” <Const> { “,” <Const> } “}” “;” の解析
        変数表に登録
        コード生成
    } else syntaxError();
} else { // スカラー変数の場合
```

```
} else if (token == "]") {   // 初期値有りの配列
    token = nextToken();
    if (token == "=") token = nextToken(); else syntaxError();
    if (token == "{") token = nextToken(); else syntaxError();
    if (token ∈ First (<Const>)) int value = parseConst();
        else syntaxError();
    ArrayList<Integer> valueList = new ArrayList<Integer>();
    valueList.add (value);
    while ( token == ",") {
        token = nextToken();
        if (token ∈ First (<Const>)) value = parseConst();
            else syntaxError();
        valueList.add (value);
    }
    if (token == "}") token = nextToken(); else syntaxError();
        :
```

作業用ArrayList

一旦ArrayListに格納

ここまで来ればサイズ確定

```
        if (token == "}") token = nextToken(); else syntaxError();
        int size = valueList.size();   // 初期値の個数を得る
        registerNewVariable (ARRAYOFINT, name, size);
                                // サイズが確定したので変数表に登録
        int address = getAddress (name);
                                        // 配列の先頭のアドレスを得る
        for (i=0; i<size; ++i) {
            appendCode (PUSHI, valueList.get (i));
                                                // i番目の初期値を積む
            appnedCode (POP, address + i);      // Dseg に格納
        }
    } else syntaxError();
} else { // スカラー変数の場合
```

変数のコード生成

- <Unsigned> ::= NAME の場合

左辺値	右辺値
PHSHI NAME の番地	PHSH NAME の番地

- <Unsigned> ::= NAME “[” <Exp> “]” の場合

左辺値	右辺値
PHSHI NAME[0] の番地 <Exp> のコード ADD	PHSHI NAME[0] の番地 <Exp> のコード ADD LOAD

左辺値か右辺値かによりコードが異なる

コード生成プログラム

- <Unsigned> ::= NAME の場合

```
void parseUnsigned () {
      :
   else if (token == NAME) {
      String name = // tokenから変数名を得る
       int address = // 変数表を参照してnameの番地を得る
      token = nextToken();
      if (左辺値が必要な場合) {
          appendCode (PUSHI, address);    // 左辺値の場合
      } else {
          appendCode (PUSH, address);     // 右辺値の場合
      }
```

左辺値か右辺値かの判定が必要

左辺値の判定

- 左辺値が必要 = 代入の左辺にある

次に来るトークンが代入かどうかで判定

```
if (token == “=”) {
    appendCode (PUSHI, address); // 左辺値の場合
} else {
    appendCode (PUSH, address);   // 右辺値の場合
}
```

(注意) token = nextToken(); はしないこと

コード生成プログラム

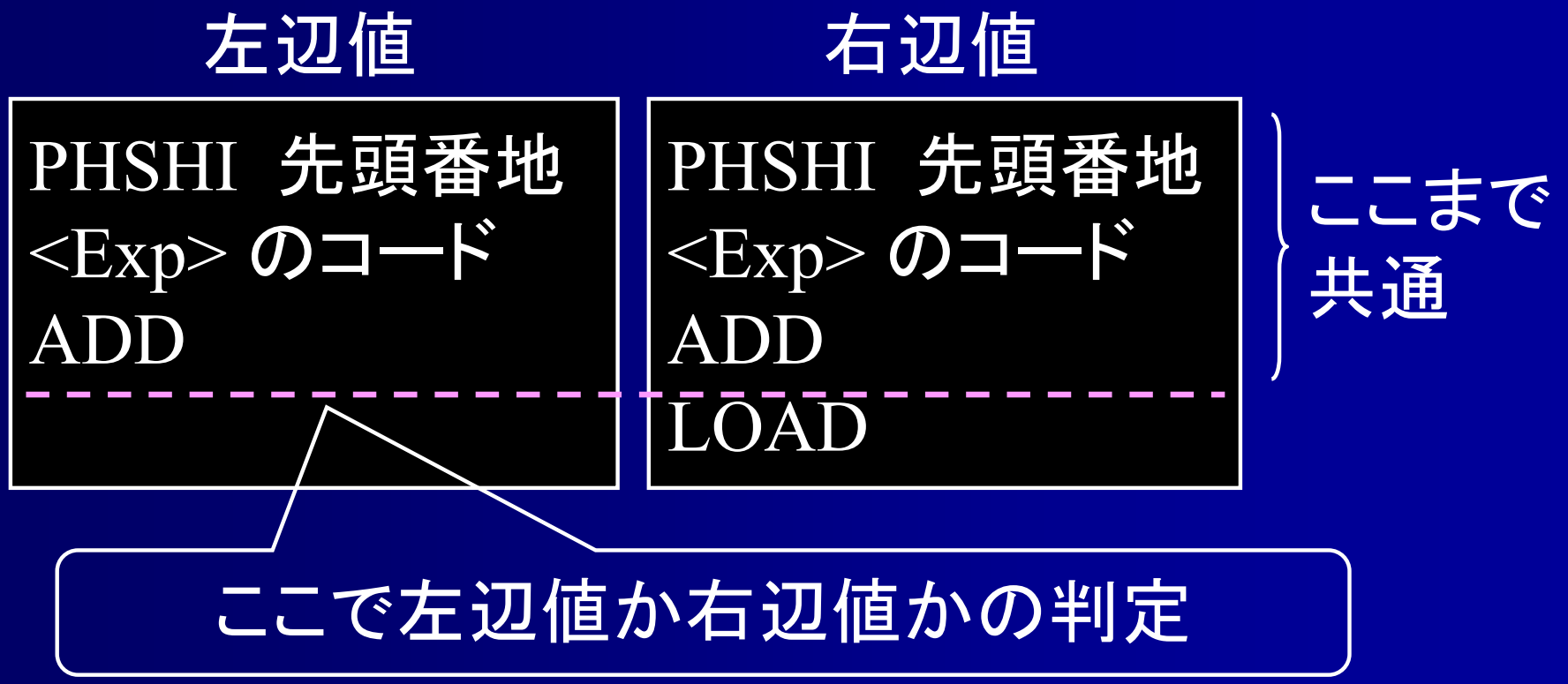
- <Unsigned> ::= NAME の場合

```
void parseUnsigned () {
        :
    else if (token == NAME) {
        String name = // tokenから変数名を得る
        int address = // 変数表を参照してnameの番地を得る
        token = nextToken();
        if (token == "=") {
            appendCode (PUSHI, address);    // 左辺値の場合
        } else {
            appendCode (PUSH, address);     // 右辺値の場合
        }
```

"+=" "-=" 等の場合も左辺値

配列のコード生成

- 配列のコード
  - <Unsigned> ::= NAME “[” <Exp> “]”



- <Unsigned> ::= NAME [“[” <Exp> “]”] の場合

```
else if (token == NAME) {
   String name = // tokenから変数名を得る
      int address = // 変数表を参照してnameの番地を得る
   token = nextToken();
   appendCode (PUSHI, address);     // 左辺値右辺値共通
   if (token == “[”) { // 配列の場合
      token = nextToken();
      if (token ∈ First (<Exp>)) parseExp(); else syntaxError();
      if (token == “]” ) token = nextToken(); else syntaxError();
      appendCode (ADD);
   }
   if (token != “=” )  // 次のトークンが代入以外
      appendCode (LOAD);
}
```

式の評価値が詰まれる

配列の左辺値が詰まれる

左辺値を右辺値に変換

代入のアセンブラコード

<Expression> ::= <Exp> [“=” <Expression>]

<Expression> → <Exp> の場合

```
<Exp> のコード (右辺値)
```

<Expression> → <Exp> “=” <Expression> の場合

```
<Exp> のコード (左辺値)
<Expression> のコード (右辺値)
ASSGN
```

- <Expression> ::= <Exp> [“=” |<Expressopn>]

```
void parseExpression () {
    if (token ∈ First (<Exp>)) {
       boolean hasLeftValue = parseExp();
                        // <Exp> の左辺値の有無をコピー
       else syntaxError();
    if (token == “=”) {
       if (!hasLeftValue)  syntaxError (“左辺値がありません”);
                                  // 左辺値が無ければ制約エラー
       token = nextToken();
       if (token ∈ First (<Expression>))
          parseExpression(); else syntaxError();
       appendCode (ASSGN);
    }
}
```

代入のアセンブラコード

<Expression> ::= <Exp>
[(“=” | “+=” | “-=”) <Expression>]

<Expression> → <Exp> “+=” <Expression> の場合

<Exp> のコード (左辺値)
COPY
LOAD
<Expression> のコード (右辺値)
ADD
ASSGN

“=” の場合

<Exp>
<Expression>
ASSGN

- <Expression> ::= <Exp> [(“=” | “+=” | “-=”) <Expressopn>

```
void parseExpresson () {
  if (token ∈ First (<Exp>)) parseExp(); else syntaxError();
  if (token == “=” || token == “+=” || token == “-=”) {
     Symbol op = token.getSymbol(); // 演算子を記憶
     token = nextToken();
     if (op == “+=” || op == “-=”) {
        appendCode (COPY);
        appendCode (LOAD);
     }
     if (token ∈ First (<Expression>)) parseExpression();
        else syntaxError();
     if (op == “+=”) appendCode (ADD);
     else  if (op == “-=”) appendCode (SUB);
     appendCode (ASSGN);
}
```

条件式のアセンブラコード

<LFactor> ::= <Exp>$_1$ “==” <Exp>$_2$

<Exp>$_1$ == <Exp>$_2$ ならば 1

```
        <Exp>1 のコード (右辺値)
        <Exp>2 のコード (右辺値)
        COMP
        BEQ     (L1)
        PUSHI 0
        JUMP    (L2)
(L1)    PUSHI 1
(L2)
```

3番地先へジャンプ

2番地先へジャンプ

Iseg の番地が必要

Iseg の番地

- PseudoIseg.appendCode ()の返り値
  - 命令を積んだ Iseg のアドレス

例 :

```
iseg.appendCode (Operetor.PUSHI, 20);
iseg.appendCode (Operetor.INC);
```

返り値 = 30

返り値 = 31

Iseg

```
30  PUSHI  20
31  INC
```

条件式のアセンブラコード

```
void parseLFactor();
    if (token ∈ First (<AExp>)) parseAExp(
        else syntaxError();
    if (token == “==”) {
        token = nextToken();
        if (token ∈ First (<AExp>)) parseAEx
            else syntaxError();
        int compAddr = appendCode (COMP);
        appendCode (BEQ, compAddr+4) ;
        appendCode (PUSHI, 0);
        appendCode (JUMP, compAddr+5);
        appendCode (PUSHI, 1);
    }
}
```

```
200 COMP
201 BEQ     204
202 PUSHI 0
203 JUMP  205
204 PUSHI 1
205
```

COMP の
アドレスを得る

条件式のアセンブラコード

```
      COMP
      BEQ     (L1)
      PUSHI   0
      JUMP    (L2)
(L1)  PUSHI   1
(L2)
```

演算子	分岐コード
==	BEQ
!=	BNE
<=	BLE
<	BLT
>=	BGE
>	BGT

```
void parseLFactor();
    if (token ∈ First (<AExp>)) parseAExp(); else syntaxError();
    if (token == “==” || token == “<” || token == “<=”) {
        Symbol op = token.getSymbol(); // 演算子を記憶
        token = nextToken();
        if (token ∈ First (<AExp>)) parseAExp(); else syntaxError();
        int compAddr = appendCode (COMP);
        switch (op) {
            case “==” : appendCode (BEQ, compAddr+4) ; break;
            case “<” :   appendCode (BLT, compAddr+4) ; break;
            case “<=” : appendCode (BLE, compAddr+4) ; break;
        }
        appendCode (PUSHI, 0);
        appendCode (JUMP, compAddr+5);
        appendCode (PUSHI, 1);
    }
}
```

if 文のアセンブラコード

<If_St> ::= “if” “(” <Exp> “)” <St>

<Exp> のコード (右辺値)
BEQ (L)
<St> のコード
(L)

<St> の次の命令の番地に分岐

(L) の番地は <St> のコードを作るまで不明

後から番地を書き直す必要あり

PseudoIseg.replaceCode(); を使用

if 文のアセンブラコード

```
void parseIfSt() {
    if (token == “if”) token = nextToken(); else syntaxError();
    if (token == “(”) token = nextToken(); else syntaxError();
    if (token ∈ First (<Exp>)) parseExp(); else syntaxError();
    if (token == “)”) token = nextToken(); else syntaxError();
    int beqAddr = appendCode (BEQ, -1);
    if (token ∈ First (<St>)) parseSt(); else syntaxError();
    replaceCode (beqAddr, <St>の次の番地);
}
```

飛び先未定

<St> の次の番地が必要

⇒ PseudoIseg.getLastCodeAddress(); を使用

Iseg の番地

- Iseg の番地は

PseudoIseg.getLastCodeAddress(); を使用

```
/** @return 最後に積んだ命令の番地 */
int getLastCodeAddress()
```

例 :

```
iseg.appendCode (Operetor.PUSHI, 10);
int addr = iseg.getLastCodeAddress();
```

Iseg

```
50  PUSHI  10
```

返り値 = 50

if 文のアセンブラコード

```
void parseIfSt() {
    if (token == “if”) token = nextToken(); else syntaxError();
    if (token == “(”) token = nextToken(); else syntaxError();
    if (token ∈ First (<Exp>)) parseExp(); else syntaxError();
    if (token == “)”) token = nextToken(); else syntaxError();
    int beqAddr = appendCode (BEQ, -1);
    if (token ∈ First (<St>)) parseSt(); else syntaxError();
    int stLastAddr = getLastCodeAddress();
                    // <St> 部分のコードの末尾のコードのアドレスを得る
    replaceCode (beqAddr, stLastAddr+1);
}
```

while 文のアセンブラコード

<While_St> ::= “while” “(” <Exp> “)” <St>

```
(L1) <Exp> のコード (右辺値)
       BEQ  (L2)
        <St> のコード
       JUMP  (L1)
(L2)
```

JUMP の次の番地に分岐

条件式にジャンプ

(L2) の番地は <St> のコードを作るまで不明

⇩

後から番地を書き直す必要あり

while 文のアセンブラコード

```
void parseWhileSt() {
    if (token == “while”) token = nextToken(); else syntaxError();
    if (token == “(”) token = nextToken(); else syntaxError();
    int lastAddr = getLastCodeAddress();
                                        // 条件式直前の番地を記憶
    if (token ∈ First (<Exp>)) parseExp(); else syntaxError();
    if (token == “)”) token = nextToken(); else syntaxError();
    int beqAddr = appendCode (BEQ, -1); // 飛び先未定
    if (token ∈ First (<St>)) parseSt(); else syntaxError();
    int jumpAddr = appendCode (JUMP, lastAddr+1);
    replaceCode (beqAddr, jumpAddr+1);
}
```

for 文のアセンブラコード

<For_St> ::= “for”
“(” <Exp>$_1$ “;” <Exp>$_2$ “;” <Exp>$_3$ “)” <St>

```
        <Exp>1 のコード (右辺値)
        REMOVE
(L1)    <Exp>2 のコード (右辺値)
        BEQ   (L4)
        JUMP (L3)
(L2)    <Exp>3 のコード (右辺値)
        REMOVE
        JUMP (L1)
(L3)    <St> のコード
        JUMP (L2)
(L4)
```

後で飛び先を決定

for 文のアセンブラコード

```
void parseForSt() {
    if (token == “for”) token = nextToken(); else syntaxError();
    if (token == “(”) token = nextToken(); else syntaxError();
    if (token ∈ First (<Exp>)) parseExp(); else syntaxError();
    if (token == “;”) token = nextToken(); else syntaxError();
    int removeAddr = appendCode (REMOVE);
                                        // 条件式直前の番地を記憶
    if (token ∈ First (<Exp>)) parseExp(); else syntaxError();
    if (token == “;”) token = nextToken(); else syntaxError();
    int beqAddr = appendCode (BEQ, -1);       // 飛び先未定
    int jumpAddr = appendCode (JUMP, -1);  // 飛び先未定
                    :
              (以下略)
```

break 文のアセンブラコード

<Break_St> ::= “break” “;”

JUMP (対応するループ, switch 文の外へ)

<Continue_St> ::= “continue” “;”

JUMP (対応するループの条件式へ)

(※) for 文は継続式(式3)へ

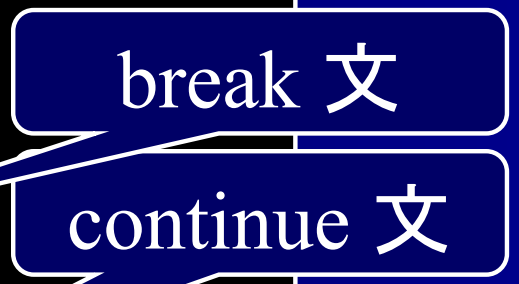
対応するループが無ければエラー

break 文のアセンブラコード
(while 文からの脱出の場合)

while (<Exp>)
{ <St_1> break ;
<St_2> continue
<St_3> }

の場合

(L1) <Exp> のコード (右辺値)
BEQ (L2)
<St_1> のコード
JUMP (L2)
<St_2> のコード
JUMP (L1)
<St_3> のコード
JUMP (L1)
(L2)



break 文 : ループ外へ
continue 文 : 継続式へ
while 文終了時に
break 文の飛び先決定

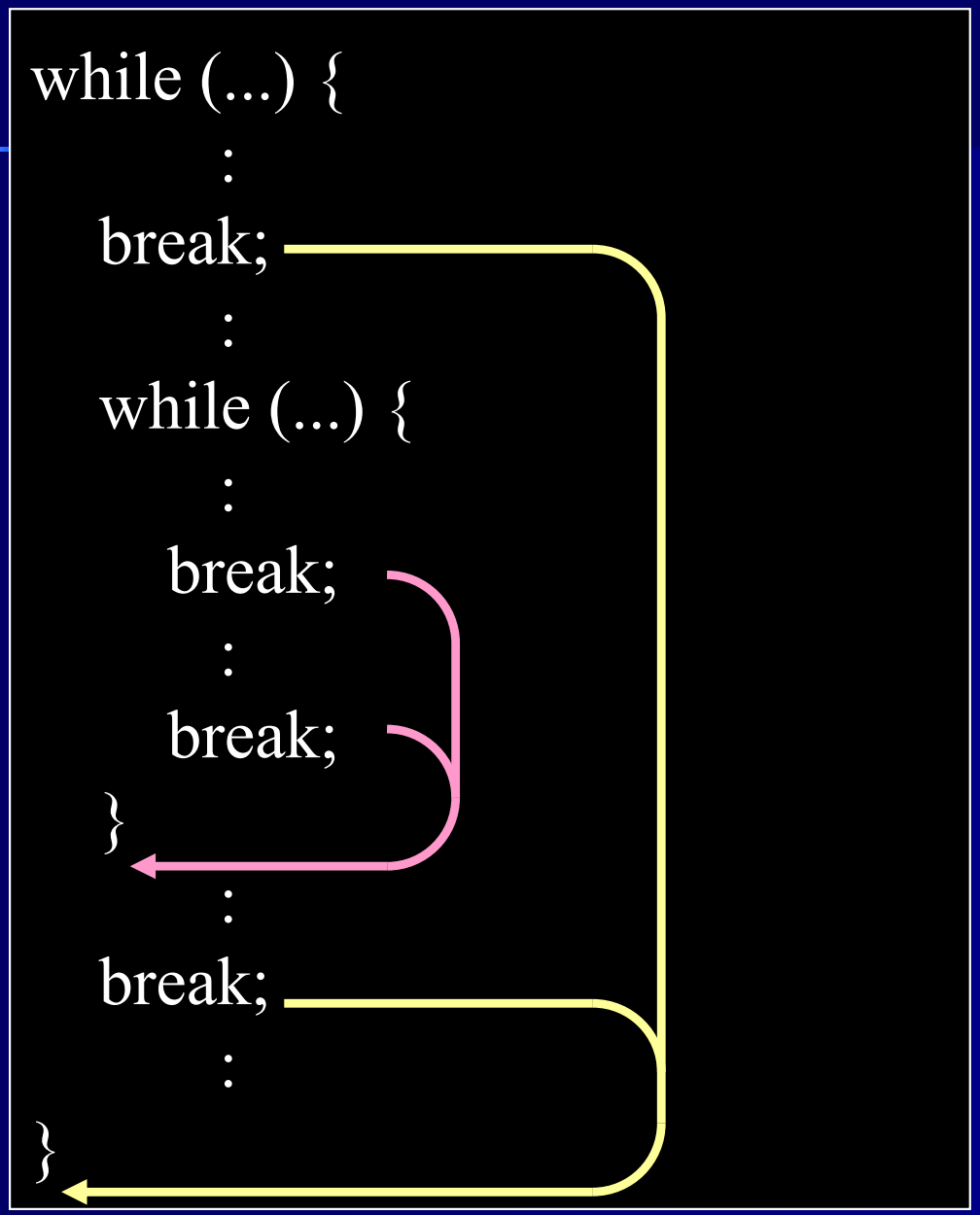
break文のコード生成

- ArrayList 型の大域変数を使用

```
ArrayList<Integer> breakAddrList;
                        /* break 文の JUMP 命令の番地を記憶する*/
boolean inLoop = false;                          /* ループ内部か？ */
```

```
parseBreak() {
   if (token == “break”) token = nextToken; else syntaxError();
   if (inLoop == false) syntaxError (“ループ内ではありません”)
   int addr = appendCode (JUMP, -1);          // 飛び先未定
   breakAddrList.add (addr);          // JUMP 命令の番地を記憶
   if (token == “;”) token = nextToken; else syntaxError()
}
```

break 文



break文は階層毎に
飛び先が異なる

⇩

階層毎に
飛び先を決定する
必要がある

```
parseWhile() {
                :
    boolean outerLoop = inLoop;     /* while文外部の情報を記憶 */
    ArrayList<Integer> outerList = breakAddrList;
    inLoop = true;                  /* フィールド変数の値をループ内部に */
    breakAddrList = new ArrayList<Integer>(); /* 空のリストを作成 */
    if (token ∈ first (<St>)) parseSt(); else syntaxError();
                            /* この<St>内はループ内部として処理される */
    int jumpAddr = appendCode (JUMP, /* 条件式へ*/ );
    for (int i = 0; i<breakAddrList.size(); ++i) {
                                    /* <St>内のbreak文の数だけ繰り返す */
        int breakAddr = breakAddrList.get (i);    /* break文の番地 */
        replaceCode (breakAddr, jumpAddr+1); /* ループ外へ */
    }
    inLoop = outerLoop;                 /* 外部のループ情報を復帰 */
    brealAddrList = outerList;
                :
```

```
parseForSt() {
                :
    boolean outerLoop = inLoop;     /* for文外部の情報を記憶 */
    ArrayList<Integer> outerList = breakAddrList;
    inLoop = true;                  /* フィールド変数の値をループ内部に */
    breakAddrList = new ArrayList<Integer>(); /* 空のリストを作成 */
    int tableSize = varTable.Size();        /*  変数表のサイズを記憶 */
                :
    for (int i = 0; i<breakAddrList.size(); ++i) {
                                    /* <St>内のbreak文の数だけ繰り返す */
        int breakAddr = breakAddrList.get (i);      /* break文の番地 */
        replaceCode (breakAddr, jumpAddr+1);   /* ループ外へ */
    }
    inLoop = outerLoop;                     /* 外部のループ情報を復帰 */
    brealAddrList = outerList;
    vatTable.removeTail (tableSize);            /* 変数表の末尾を削除 */
}
```

break 文

breakAddrList

100

```
while (...) {
        :
    break;
        :
    while (...) {
        :
        break;
        :
        break;
    }
        :
    break;
        :
}
```

```
        :
100 JUMP  ?
```

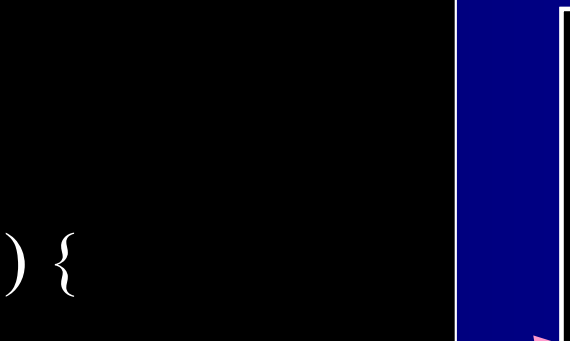
break 文

```
while (...) {
        :
    break;
        :
    while (...) {
        :
        break;
        :
        break;
    }
        :
    break;
        :
}
```

breakAddrList

100	
150	200

```
        :
100 JUMP  ?
        :
150 JUMP  ?
        :
200 JUMP  ?
```



break 文

```
while (...) {
        :
    break;
        :
    while (...) {
        :
        break;
        :
        break;
    }
        :
    break;
        :
}
```

breakAddrList

100	
150	200

```
        :
100 JUMP  ?
        :
150 JUMP  250
        :
200 JUMP  250
        :
250
```

break 文

breakAddrList

100	300

```
while (...) {
        :
    break;
        :
    while (...) {
        :
        break;
        :
        break;
    }
        :
    break;
        :
}
```

```
        :
100 JUMP  ?
        :
150 JUMP  250
        :
200 JUMP  250
        :
250
        :
300 JUMP  ?
```

break 文

breakAddrList

100	300

```
while (...) {
        :
    break;
        :
    while (...) {
        :
        break;
        :
        break;
    }
        :
    break;
        :
}
```

```
        :
100 JUMP  350
        :
150 JUMP  250
        :
200 JUMP  250
        :
250
        :
300 JUMP  350
        :
350
```

プログラム末到達時の処理

- プログラム末到達時にファイル末ならば
  コンパイル完了

```
void parseProgram () {
    if (token ∈ First (<MainFunction>))
        parseMainFunction();
        else syntaxError();
    if (token == “$”)
        appendCode (HALT);
        else syntaxError();
}
```

ファイル末を示すトークン

末尾に HALT を積む

Iseg からファイルへの出力

- Iseg からファイルへの出力は

  PseudoIseg.dump2file ()

  PseudoIseg.dump2file (String) を使用

```
void dump2file ()
void dump2file (String fileName)
```

例 : Iseg を OpCode.asm (デフォルト) に出力

```
iseg.dump2file ();
```

Iseg を xxx.asm に出力

```
iseg.dump2file (“xxx.asm”);
```

for 文のアセンブラコード

<For_St> ::= “for”
“(” <Var_decl> “;” $<Exp>_2$ “;” $<Exp>_3$ “)” <St>

```
        <Var_decl> のコード
(L1)    <Exp>2 のコード (右辺値)
        BEQ   (L4)
        JUMP (L3)
(L2)    <Exp>3 のコード (右辺値)
        REMOVE
        JUMP (L1)
(L3)    <St> のコード
        JUMP (L2)
(L4)
```

VarTable の管理

```
for (int i=0; i<n; ++i ) {

    :

}
```

int 型変数 i は
この内部のみで有効

for 文開始時に変数表のサイズを記憶
for 文終了時に変数表の末尾を削除

for 文のアセンブラコード

```
void parseForSt() {
    if (token == “for”) token = nextToken(); else syntaxError();
    int tableSize = varTable.Size();      /*  変数表のサイズを記憶 */
        :
        :
        :
    vatTable.removeTail (tableSize);      /* 変数表の末尾を削除 */
}
```

配列のアドレス

多次元配列の
アドレス計算は
各次元の大きさが必要

1次元配列

```
int a[N];
```

a[i] のアドレス : (a[0] のアドレス) + i

2次元配列

```
int a[M][N];
```

a[i][j] のアドレス : (a[0][0] のアドレス) + N*i + j

3次元配列

```
int a[L][M][N];
```

a[i][j][k] のアドレス : (a[0][0][0] のアドレス)
+ M*N*i + N*j + k

配列のアドレス

a[<Exp>$_1$]

```
PUSHI a[0] の番地
<Exp>1 のコード (右辺値)
ADD
```

a[<Exp>$_1$][<Exp>$_2$]

```
PUSHI a[0][0] の番地
<Exp>1 のコード (右辺値)
PUSHI  N
MUL
ADD
<Exp>2 のコード (右辺値)
ADD
```

a[<Exp>$_1$][<Exp>$_2$][<Exp>$_3$]

```
PUSHI a[0][0][0] の番地
<Exp>1 のコード (右辺値)
PUSHI M*N
MUL
ADD
<Exp>2 のコード (右辺値)
PUSHI N
MUL
ADD
<Exp>3 のコード (右辺値)
ADD
```

多次元配列への対応

- Var, VarTable
  - 各次元の大きさ、次元も登録できるようにする
- parseVarDecl()
  - 配列の次元、各次元の大きさも調べ、登録する
- parseUnsignedFactor()
  - [] の個数が登録された次元と一致するか確認する
  - 変数表から各次元の大きさを得て番地を計算する

Var.java の拡張

```
public class Var{
   private Type type;            // 型
   private String name;          // 変数名
   private int address;          // 番地
   private int size;             // サイズ
   private int sizeList[];       // 各次元のサイズ
   private int dimension;        // 配列の次元
                :
```

多次元配列の変数表

```
int i, j;
int a[10], b[5][6], c[2][3][4];
```

Type	name	address	size	sizeList	dim
int	i	0	1	null	0
int	j	1	1	null	0
array of int	a	2	10	{ 10 }	1
array of int	b	12	30	{ 5, 6 }	2
array of int	c	42	24	{ 2, 3, 4 }	3

```
int dimension = 0, size = 1;
ArrayList<Integer> sizeList = new ArrayList<Integer>();
while (token == “[”) {
    token = nextToken();
    ++dimension;                                    // 次元をカウント
    if (token == INTEGER) {
       size *=  token.getValue();                  // 全体の大きさを計算
       sizeList.add (token.getValue());          // 各次元の大きさを記憶
       token = nextToken();
    } else syntaxError();
    if (token == “]”) token = nextToken(); else syntaxError();
}
if (dimension == 0) { // スカラー変数の場合
    registerNewVariable (INT, name, 1, null, 0);
} else {                              // 配列の場合
    registerNewVariable
                  (ARRAYOFINT, name, size, sizeList, dimension);
}
```