2023 年度 第 5 セメスター

情報システムプロジェクトⅠ指導書

近畿大学 理工学部 情報学科

目次

1 実習全般について

1.1 まえがき

この実習では簡単な C 風の手続き型言語 (以下 K23 言語と呼ぶ) のコンパイラの作成を行なう．

実習の主目的は言語処理プログラムの構成法についての体験的な理解を得ることであるが，あわせて Java 言語によるプログラミングの諸技法の習得，定められた日程にあわせた段階的なソフトウェア作成の経験の取得を目的としているので，そのことをよく念頭に置いて実習に取り組むこと．

また，本実習は，講義科目である「コンパイラ」と密接な関係を持っており，講義と連動する形で実習を進める．「コンパイラ」の講義は必修科目ではないが，たとえ「コンパイラ」を履修しない場合でも，それ相応の自習が必要になることを注意しておく．

1.1.1 コンパイラについて

諸君はこれまでに，Java，C, Prolog 等いくつかのプログラミング言語に接してきた．これらの言語では，中置記法の数式など人間にとって読みやすい，あるいは書きやすい表現を用いることができるようになっている．このような言語を高水準言語と呼ぶ.

一方計算機は，機械語と呼ばれる 0 と 1 の並びで記述された命令によるプログラムしか解釈実行できない．また，各命令をニーモニック表現 (簡略化した英単語) に置き換えた言語がアセンブラである．機械語やアセンブラ言語を低水準言語と呼ぶ.

コンパイラとは，「高水準言語で書かれたプログラムを低水準言語のプログラムに変換する」プログラムのことであり，諸君は今までに履修してきた講義や実習で，自分で書いたプログラムを実行するために，いくどとなくコンパイラのお世話になってきているわけである．

1.1.2 実習の概要

本実習では，これまでの講義や実習で訓練してきたプログラミング技術の一つの集大成として，コンパイラ自体を作成する．具体的には K23 言語で書かれたソースプログラムを，スタック上での演算機能を備えた仮想計算機 VSM (Virtual Stack Machine) のアセンブラに翻訳するコンパイラを以下の手順にしたがって作成する．

(1) ファイル入力用のクラス FileScanner.java と，変数管理用のクラス VarTable.java の作成
(2) VSM の命令体系の理解を目的とした手作業によるコンパイルと，目的プログラムのシミュレータ上での実行
(3) 字句解析プログラム LexicalAnalyzer.java の作成
(4) 再帰降下 (LL(1)) 型構文解析法による構文解析プログラム Kc.java の作成
(5) Kc.java へのコード生成機能の埋め込み
(6) 拡張機能の実装

上記の順に，設定されたスケジュールにあわせてプログラム作成を進める．

1.2 K23 言語

本実習で作成するコンパイラの原始言語である K23 言語の概要と構文の形式的定義を与える．

1.2.1 概要

K23 は以下の特徴をもつ簡単な手続き型言語である．

(a) データ型は整数 (int) 型のみとする．真理値，文字も int 型データとして扱う．int 型データを真理値として扱う場合，0 は偽 (false)，それ以外の値は真 (true) と解釈するものとする．
(b) 配列は 1 次元のもののみ取扱う．
(c) 制御構造は while 文，for 文，if 文 (else 節なし) のみとし、その機能は C ないし Java に準拠する．
(d) 入出力は標準入出力のみとし，入力は式中に記述し，出力は出力文として扱う．
(e) 式の記述に用いる演算子はおおむね C ないし Java に準拠する．
(f) 式の記述に 10 進もしくは 16 進の定数を用いることができる．
(g) 関数定義，メソッド定義の機能は持たない．

K23 言語のソースプログラム例を，巻末付録 A (pp. 42〜48) に示す．このうちプログラム例 10 は小規模な字句解析プログラムの機能検査用のものであるが，残りは K23 言語の構文規則に則ったものである．なかでもプログラム例 23 bsort.k (p. 48) は配列 data に収められた整数値を昇順に並べかえるバブルソートプログラムになっており，K23 言語のほぼすべての機能が使用されている．端的に言えば，このプログラムを VSM のアセンブラに変換できるコンパイラを作成するのが今回の実習の一応のゴールということになる．

以下 K23 言語の形式的定義を与える．

1.2.2 マイクロ構文

K23 言語のマイクロ構文として以下のものを考える．

(a) K23 言語でキーワードとして扱う文字列は次の通り．

break, for, if, inputchar, inputint, int, main, outputchar, outputint, while

(b) 変数や配列の名前は英字 ('_' を含む) から始まる英字と数字からなる文字列とする．
(c) 整数は 10 進数および 16 進数とする．また 10 進数は適切に 0 抑制が行われているものとする．

これを EBNF 記法により形式的に定義したものを以下に示す．ただし，Alpha, Dec, Pdec はそれぞれ，'_' を含む大文字または小文字の英字，数字，1 から 9 までの数字を表し，Xdec は 16 進数の表記に用いる数字もしくは'A' から'F' までの英大文字を表す．また' ', '\t', '\n', '\0' はそれぞれ空白文字，水平タブ文字，改行文字，NULL 文字を表し，さらに Character は印字可能文字を表している．

```
K-Program ::= { Token | W-Space } '\0'
Token ::= NAME | INT | CHAR | KEYWORD | OPERATOR | DELIMITER
W-Space ::= ' ' | '\t' | '\n'

NAME ::= Alpha { Alpha | Dec }
```

```
INT  ::= ::= '0' | Pdec { Dec } | '0' 'x' Xdec { Xdec }
CHAR::= ''' Character '''

KEYWORD ::= 'b' 'r' 'e' 'a' 'k' | 'f' 'o' 'r' | 'i' 'f'
          | 'i' 'n' 'p' 'u' 't' 'c' 'h' 'a' 'r' | 'i' 'n' 'p' 'u' 't' 'i' 'n' 't'
          | 'i' 'n' 't' | 'm' 'a' 'i' 'n' | 'o' 'u' 't' 'p' 'u' 't' 'c' 'h' 'a' 'r'
          | 'o' 'u' 't' 'p' 'u' 't' 'i' 'n' 't' | 'w' 'h' 'i' 'l' 'e'

OPERATOR ::= '=' '=' | '!' '=' | '<' | '>' | '&' '&' | '|' '|' | '!'
           | '+' | '-' | '*' | '/' | '%'
           | '=' | '+' '=' | '-' '=' | '*' '=' | '/' '='
           | '+' '+' | '-' '-'

DELIMITER ::= ';' | '(' | ')' | '{' | '}' | '[' | ']' | ','
```

また，トークンの認識は最長一致の原則に基づいて行われるものとする．例えば文字列 “--” はトークン「-」が 2 つ連続すると認識されるのではなく，一つの「--」と認識されるものとする．また「interest」という文字列は「int」というキーワードと「erest」という名前として解釈されるのではなく，「interest」という名前と解釈される．

1.2.3 マクロ構文

K23 言語のマクロ構文の形式的定義を EBNF 記法で記述する．ただし「<」で始まり「>」で終わる文字列は一つの非終端記号を表すものとする．また「"」で囲まれた文字列は KEYWORD, OPERATOR ないし DELIMITER といったトークンそのものを表し，NAME，INT，CHAR はそれぞれ名前，非負整数，文字に対応するトークンを表している．また，EOF はファイル末を表す．

```
<Program> ::= <Main_function> EOF
<Main_function> ::= "main" "(" ")" <Block>
<Block> ::= "{" { <Var_decl> } { <Statement> } "}"

<Var_decl> ::= "int" <Name_list> ";"
<Name_list> ::= (<Name_list> "," <Name>) | <Name>
<Name> ::= NAME
        | ( NAME "=" <Constant> )
        | ( NAME "[" INT "]" )
        | ( NAME "[" "]" "=" "{" <Constant_list> "}" )

<Constant_list> ::= (<Constant_list> "," <Constant>) | <Constant>
<Constant> ::= ["-"] INT) | CHAR

<Statement> ::= <If_statement> | <While_statement>
              | <For_statement> | <Exp_statement>
              | <Outputchar_statement> | <Outputint_statement>
              | <Break_statement>
              | ("{" { <Statement> } "}") | ";"
```

```
<If_statement> ::= "if" "(" <Expression> ")" <Statement>
<While_statement> ::= "while" "(" <Expression> ")" <Statement>
<Exp_statement> ::= <Expression> ";"
<For_statement> ::= "for" "(" <Expression> ";" <Expression> ";" <Expression> ")" <Statement>
<Outputchar_statement> ::= "outputchar" "(" <Expression> ")" ";"
<Outputint_statement> ::= "outputint" "(" <Expression> ")" ";"
<Break_statement> ::= "break" ";"

<Expression> ::= <Exp> [("=" | "+=" | "-=" | "*=" | "/=") <Expression>]
<Exp> ::= (<Logical_term> "||" <Exp>) | <Logical_term>
<Logical_term> ::= (<Logical_factor > "&&" <Logical_term>) | <Logical_factor>
<Logical_factor> ::= <Arithmetic_expression>
                   [("==" | "!=" | "<" | ">") <Arithmetic_expression>]
<Arithmetic_expression> ::= (<Arithmetic_expression> ("+" | "-") <Arithmetic_term>)
                          | <Arithmetic_term>
<Arithmetic_term> ::= (<Arithmetic_term> ("*" | "/" | "%") <Arithmetic_factor>)
                    | <Arithmetic_factor>
<Arithmetic_factor> ::= <Unsigned_factor>
                      | ("-" <Arithmetic_factor>)
                      | ("!" <Arithmetic_factor>)

<Unsigned_factor> ::= NAME
                    | (NAME ("++" | "--"))
                    | (NAME "[" <Expression> "]")
                    | (("++" | "--") NAME)
                    | (("++" | "--") NAME "[" <Expression> "]")
                    | INT | CHAR
                    | ("(" <Expression> ")")
                    | "inputchar" | "inputint"
                    | <Sum_funcrtion>
                    | <Product_fuction>

<Sum_function> ::= "+" "(" <Expression_list> ")"
<Product_function> ::= "*" "(" <Expression_list> ")"
<Expression_list> ::= (<Expression_list> "," <Expression>) | <Expression>
```

さらに，以下の規則を満たすものとする．

- <Expression>中の代入演算子の左被演算子
  代入演算子 「=」, 「+=」, 「-=」,「*=」, 「/=」の左被演算子となる<Exp>は変数または配列要素，すなわち NAME もしくは NAME "[" <Expression> "]" の形のものでなければならない．例えば「x-y=z」は<Exp>にマッチする部分式が「x-y」となるのでこの規則を満たしておらず，構文エラーとみなす．
- break 文の記述位置
  break 文は while 文および for 文中にのみ記述できる．

なお, n 個の <Expression> $e_1, e_2, ..., e_n$ に対して， 和関数 $+(e_1, e_2, ..., e_n)$，積関数 $*(e_1, e_2, ..., e_n)$ はそ

れぞれ

$$+(e_1, e_2, ..., e_n) = (e_1) + (e_2) + ... + (e_n)$$

$$*(e_1, e_2, ..., e_n) = (e_1) * (e_2) * ... * (e_n)$$

と定義される．

1.3 自習レポートについて

コンパイラはおそらく諸君が初めて体験する大規模なプログラムになる．そのため実習時間だけの学習では，完成させることはなかなか難しいと思われる．従って本科目では，課外時間における学習も重要視する．このため，ほぼ毎週予習・復習課題をレポートとして課す．これらの自習レポートの内容は講義時間に説明するが，毎週共通している課題は以下の報告書の作成である．

- 予習課題として，当日の実習計画 (作業項目，各項目に費やす時間等) を作成してくる．また，復習として，当日実際に行った作業（実習日以降次回の実習日までに行った作業がある場合それも含める）の報告書を作成する．
- 計画や報告書は，第 1 回実習で配布する実習ノートに書く．

また，課題プログラムは，こちらで用意した専用の課題提出用プログラムを使って提出する．その他の方法での提出はできない．提出状況はそのプログラムを使って各自確認できるようになっている．詳細は実習時に説明する．

2 準備 (第 1〜3 週)

第 1〜3 週を実習の第一フェーズとし，準備として K23 言語，ターゲットマシン VSM(Virtual Stack Machine) の理解と，VSM アセンブラのプログラミングのノウハウを習得する．さらに，Java プログラミングの復習を兼ねて，コンパイラで必要となるいくつかのクラスのプログラムを作成する．

2.1 第 1 週

第 1 週目では，本実習の目的の理解と K23 言語の機能の理解を中心に実習を進める．

まずは講義形式で，本実習で作成するコンパイラの概要，実習の進め方，レポートの提出方法，到達目標等を説明する．

2.1.1 K23 言語プログラム

前にも述べたように，プログラム例 23 (p. 48) のバブルソートプログラムは K23 言語のほとんどの構文要素を含んでいる．このプログラムが正しく理解できていれば，K23 言語の機能についてほぼ理解していると考えてよい．そこでこのプログラムをざっと見ていくことにする．

- 2 行目 `int i, n=0, m=1, s, tmp, is_sorted=1, SIZE=20, data[20], product;` プログラムで使用する変数を宣言している．K23 言語の変数宣言は Java のようにプログラム中の任意の場所に記述できるのではなく，ブロックの先頭に置かれなければならない．この例では変数 `n`, `m`, `is_sorted`, `SIZE` にはそれぞれ初期値 0, 1, 1, 20 が設定され，`data` はサイズ 20 の配列として宣言されている．
- 3 行目 `int message[]= {'s','o','r','t'};`
  `message` は配列として宣言され，リスト `{'s','o','r','t'}` により，そのサイズが 4(リストの要素の数) であること，および各要素の初期値がそれぞれ文字定数`'s'`, `'o'`, `'r'`, `'t'` であることが指定されている．
- 4 行目 `;`
  いわゆる空文である．
- 5 行目 `outputchar('?');`
  文字定数を引数とした出力文の記述例である．
- 6 行目 `product = *(1,inputint,m);`
  キーボード入力の使用例である．入力は式文中に埋め込んで記述する．また，*(...) は括弧内の値の列の積を取る積関数の例である．
- 9 行目 `... i *= 0 ...`
  乗算代入演算子の使用例である．`i` の値が何であってもそれを 0 倍しているので 0 であり，「`i=0`」とした場合とプログラムの実行結果は同じであるが，生成されるアセンブラは異なる．
- 9 行目 `... ++i ...`
  スカラ変数に対する前置 `++` （`i` の値を 1 増やす）の記述例である．
- 9 行目から 13 行目 `for (i *= 0; i < SIZE; ++i) ...`
  `for` ループの記述例である．`for` 文はキーワード `for` に続いて，括弧内に初期設定式，繰り返し条件式，更新式を記述し，その後に本体の処理を記述するもので，最初に初期設定式が実行された後，繰り

返し条件式の値が 0(真理値 `false` を表す `int` 型データ) でない限り，本体および更新式を繰り返して実行する．この例では `i * =0` を実行することにより制御変数 `i` に初期値 0 を設定した後，`i` の値が `SIZE` の値より小さい間ループ本体と更新式 `++i` が繰り返して実行される．

- 10 行目 `... 0x0002F ...`
  16 進数の記述例である．
- 11 行目 `+(m,i)`
  括弧内の値の和を取る和関数の使用例である．
- 12 行目 `++data[i];`
  配列に対する前置 `++` (配列要素 `data[i]` の値を 1 増やす) の記述例である．
- 15 行目 `n= m= m* 0;`
  一つの式文中で複数の代入が行われる例である．
- 16 行目から 19 行目 `while (...) ...`
  `while` ループの記述例である．`while` ループは条件式の値が 0 でない限り，本体の処理を繰り返して実行する．
- 17 行目 `outputint(data[n]);`
  整数値の出力文の記述例である．
- 23 行目 `while (1) ...`
  条件式が常に 1(真) である while 文の例である．この場合，ループ中に break 文を記述することでループから脱出できる．
- 24 行目 `if (...) ...`
  `if` 文の記述例である．条件式の値が 0 でないときに限り，本体が実行される．
- 26 行目 `break;`
  `break` 文の記述例である．`break` 文があると，対応する `while` 文から脱出する．

問題 2.1 `bsort.k` を以下の手順で実行せよ．

1. 教員側で用意した K23 コンパイラ (`Kc23.class`) を利用して，`bsort.k` をコンパイルせよ．

   ```
   $ cd ~/Documents/projI23/material
   $ java kc.Kc23 bsort.k bsort.asm
   ```

2. コンパイルにより作成された `bsort.asm` が，どのような命令列になっているかを確認せよ．
3. VSM シミュレータを用いて `bsort.asm` を実行せよ．

   ```
   $ ./vsm bsort.asm
   ```

   このプログラムの実行時には整数を一つ入力してやる必要がある．まずは 0 を入力しそのときの実行結果を確認する．次にざまざまな数値を入力した際の実行結果を確認すること．

問題 2.2 K23 言語の構文と機能が理解できたと思ったら，本当に正しく理解できているかを確認するために実際に自分でプログラムを作成して，実行してみよう．

1. 付録 A を参考に K23 言語の各種機能を活用したプログラムを emacs 等のエディタを用いて作成せよ．
2. 作成したプログラムをコンパイルせよ．例えば作成したプログラムのファイル名が foo.k の場合，以下

の様にすればコンパイルできる．

```
$ java kc.Kc23 foo.k foo.asm
```

3. コンパイルにより生成された VSM アセンブラプログラムが，どのような命令列になっているかを確認せよ．
4. VSM シミュレータを用いて VSM アセンブラプログラムを実行せよ．

```
$ ./vsm foo.asm
```

5. 構文エラーを含むプログラムをいくつか作成し，`Kc23.class` で処理させてみよ．

なお，ここで作成する K23 プログラムは，将来自分の作成したコンパイラの試験，デバグに用いることを想定して作成すること．

2.1.2 VSM の学習

第 1 週の残りの時間は，付録 D により，VSM の構造と動作，および逆ポーランド記法について学習すること．

2.1.3 宿題 (Java の自習課題)

第 2 週から始まるコンパイラ作成の準備として，以下に与える Java の自習課題を必ず行っておくこと．

問題 2.3（クラスとインスタンスの復習）「オブジェクト指向 Java プログラミング入門（第 2 版）」の 6.1〜6.3 節 (pp. 138–150) および 9.2〜9.3 節 (pp. 189–195) を熟読し，ソースコード 6.1〜 6.4 を打込んで動作を確認せよ．クラスとインスタンスの概念を，しっかりと思い出しておくこと．

問題 2.4（ファイル操作の予習） コンパイラではファイルの操作が必須である．「オブジェクト指向 Java プログラミング入門」の 10 章全部を熟読し，ソースコード 10.1 を打込んで動作を確認せよ．

2.2 第 2 週

第 1 週で，本実習で取り組む処理系が K23 言語のプログラムから VSM アセンブラプログラムへ変換するものだという事が理解できたと思う．コンパイラはファイルに保存された K23 言語プログラムを一行ずつ読み込み (ファイル入力)，変換後のアセンブラプログラムをファイルに書き出す (ファイル出力する) 必要がある．

第 2 週は，問題 2.4 で予習してきた File クラスと Scanner クラスを用いたファイル入力技法を基礎とし，本実習で作成する処理系のファイル入力に関わる部分を作成する．

2.2.1 K23 言語処理系の概要

本実習で作成する処理系は，主に次の 3 つのプログラムから構成される．

FileScanner.java　ファイル入力を行い，ファイルの各行から文字を一文字ずつ切り出して，その文字を LexicalAnalyzer に渡す．

LexicalAnalyzer.java　FileScanner から受け取った文字 (列) を意味のある単語として組み立て，その単語を分類する．この処理を字句解析と呼ぶ．その単語をトークンと呼ばれるデータとして Kc に渡す．

Kc.java　一つ以上のトークンを受け取り，それ (ら) を文法と照らし合わせて K23 言語プログラムの式や構造として認識する．この処理を構文解析と呼ぶ．さらに Kc は，解析の結果を基にして，対応する VSM アセンブラの命令を次々と生成する．

2.2.2 FileScanner の作成

ここでは実際に，FileScanner.java を作成する．FileScanner.java はファイルから文字列を読み出し，第 4 週以降で作成する字句解析器に文字を一文字ずつ渡す機能を持つクラスである．以下に指示する問題に分けて，段階的に作成する．

問題 2.5　次の仕様を満たすクラス FileScanner.java を作成せよ．

フィールド

- 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　：先読み文字

コンストラクタ

- FileScanner(String sourceFileName)　：`sourceFileName` をファイル名とするファイルを読み取るための `BufferedReader` を生成し，`sourceFile` で参照する．また，`lineNumber` を 0, `columnNumber` を -1 に，`nextCharacter` を`'\n'` (改行コード) にそれぞれ初期化する．その後，`nextChar()` を実行し，`nextCharacter` に `sourceFile` の 1 文字目を読み込む．ただし，問題 2.6 に取り掛かるまでは，`nextChar()` の実行部分はコメントアウトしておくこと．

メソッド

void closeFile() ：sourceFile で参照されているファイルを閉じる．

void readNextLine() ：sourceFile から readLine メソッドを用いて line に 1 行分読み込み，末尾に改行コード'\n' を連接する．もし，ファイル末まで達していれば line には null 値を格納する．また，読み込む際に例外 (読み込みエラー) が発生した場合，このメソッド内でその例外を表示し，ファイルを閉じ (closeFile())，プログラムを終了 (System.exit(1)) すること．

char lookAhead() ：nextCharacter 中の文字を返す．

String getLine() ：line フィールドの文字列を返す，

メインメソッド　このクラスのインスタンスを一つ作成し，そのインスタンスに，ファイル末 (getLine() で得られた文字列が null 値) になるまで，readNextLine() と getLine() で取り出した文字列の画面表示 (System.out.print) を繰り返す．ファイル末に達したら closeFile を行って終了する．なお，読み込むファイルには，プログラム例 23 (p. 48) に掲載されている bsort.k を用いること．実行結果は，bsort.k の内容が全て表示されるはずである．

問題 2.6　FileScanner.java に，一文字切り出し用のメソッド char nextChar() を追加せよ．nextChar() の直観的な動作を以下に示す．

初期状態

```
  m a i n ( ) {\n
↑ nextCharacter = '\n '
```

nextChar() 実行 1 回目 (コンストラクタ中で実行される)

```
  m a i n ( ) {\n
↑ currentCharacter = '\n', nextCharacter = 'm'
```

nextChar() 実行 2 回目

```
  m a i n ( ) {\n
  ↑ currentCharacter = 'm', nextCharacter = 'a'
```

nextChar() 実行 3 回目

```
  m a i n ( ) {\n
    ↑ currentCharacter = 'a', nextCharacter = 'i'
```

nextChar() 実行 4 回目

```
  m a i n ( ) {\n
      ↑ currentCharacter = 'i', nextCharacter = 'n'
```

nextChar() で一文字読み進める時，ファイルのどの部分を読んでいるかで次の 3 つの場合がある．

1. ファイル末に達した場合．
2. 行末に達した場合．
3. 行の途中の場合．

それぞれ異なった処理が必要となるが，共通しているのはどれも最初に currentCharacter に nextCharacter を代入することと，その currentCharacter を返り値とすることである．この他に，各々

の場合について，以下のような動作をさせること．

1. ファイル末に達していたら (nextCharacter が'\0')，共通動作以外は何もしない．
2. 行末に達していたら (nextCharacter が'\n')，readNextLine を実行して次の行を読む．次の行が null でなければ nextCharacter にその行の最初の文字を読み込み，lineNumber をインクレメントし，columnNumber を 0 にする．次の行が null の場合は nextCharacter にはファイル末を表す'\0'を入れる．
3. 行の途中 (上記以外) の場合は，columnNumber をインクレメントした後，nextCharacter に line の columnNumber 番目の文字を入れる．

nextChar() ができたら，問題 2.5 で作成したメインメソッドにおいて，行の表示部分と，readNextLine() の実行部分をコメントアウトし，その代りに nextChar() の戻り値を表示することで，bsort.k の全内容が表示できるようにせよ．文字の出力には System.out.print を用いること．

問題 2.7 FileScanner に以下のメソッドを追加せよ．

String scanAt() ： 現在入力ファイルのどの部分をスキャンしているのかを表現する文字列を返す．line フィールド，lineNumber フィールド，columnNumber フィールドを参照して，十分な情報をわかりやすく表示できるよう各自工夫すること．

2.2.3 動作確認

作成したプログラムは仕様通りに動くか動作確認が必要である．動作確認では以下の 2 つの項目をチェックする．

- 入力データとして仕様に合うものを用いた場合は，仕様通りの出力をする
- 入力データとして仕様を満たさないものを用いた場合は，入力データが不適切であることを表示する

入力データは，プログラムのどの部分をチェックするのかを考えて作成すること．想定されるバグに対して，どのような入力データを与えればそのバグを検出できるかを考えること．また，VSM アセンブラプログラムを出力するプログラムの動作確認は，VSM シミュレータを用いて出力された VSM アセンブラプログラムが仕様通りの動作をするかも確認すること．

プログラム例 1 FileScanner.java

```
package kc;
import java.nio.file.*;
import java.io.*;
class FileScanner {
    /* フィールドの定義 */

    /** 引数 sourceFileName で指定されたファイルを開き, sourceFile で参照し
     *  各フィールドを初期化する */
    FileScanner (String sourceFileName) {
        Path path = Paths.get (sourceFileName);
        // ファイルのオープン
        try {
            sourceFile = Files.newBufferedReader (path);
        } catch (IOException err_mes) {
            System.out.println (err_mes);
            System.exit (1);
        }
        /* 各フィールドの初期化 */
        /* nextChar によって nextCharacter に先頭文字を読み込む */
    }

    /** sourceFile で参照しているファイルを閉じる */
    void closeFile() {
        try {
            sourceFile.close();
        } catch (IOException err_mes) {
            System.out.println (err_mes);
            System.exit (1);
        }
    }

    /** sourceFile で参照しているファイルから一行読み line に格納する */
    void readNextLine() {
        try {
            if (sourceFile.ready()) { // sourceFile 中に未読の行があるかを確認
                /* nextLine メソッドで sourceFile から 1 行読み出し
                 * 読み出された文字列は改行コードを含まないので改めて改行コードをつけ直す */
            } else {
                /* 未読の行が無い場合の処理 */
            }
        } catch (IOException err_mes) {
            /* ファイルの読み出しエラーが発生したときの処理 */
        }
    }

    /*  以下仕様にある通りの動作を記述する */

}
```

2.3 第 3 週

ここでは，K23 言語プログラムを VSM アセンブラに変換する技法について学習する．

2.3.1 VSM オペレータセット

K23 言語の構文要素を VSM 命令の並びに変換するために，VSM アセンブラで使用するオペレータについて説明する．オペレータの名前と意味の一覧を 付録 C 章に掲載する．ここでは，実習に必要な主なオペレータをとりあげ，その動作を詳説する．

ASSGN:

```
addr = Stack[--SP];
Dseg[addr] = Stack[SP] = Stack[SP+1];
```

ここで SP は，スタックトップの位置を示す変数である．

例えばスタックトップに 3，スタックトップの一つ前（以下、スタックの 2 番目と呼ぶ）に 0 が積まれている場合，命令 ASSGN を実行すると，Dseg の 0 番地に 3 が格納されるが，この動作が上記 ASSGN の 2 行の C プログラムでどのように実現されているか，図 1 を使って説明する．

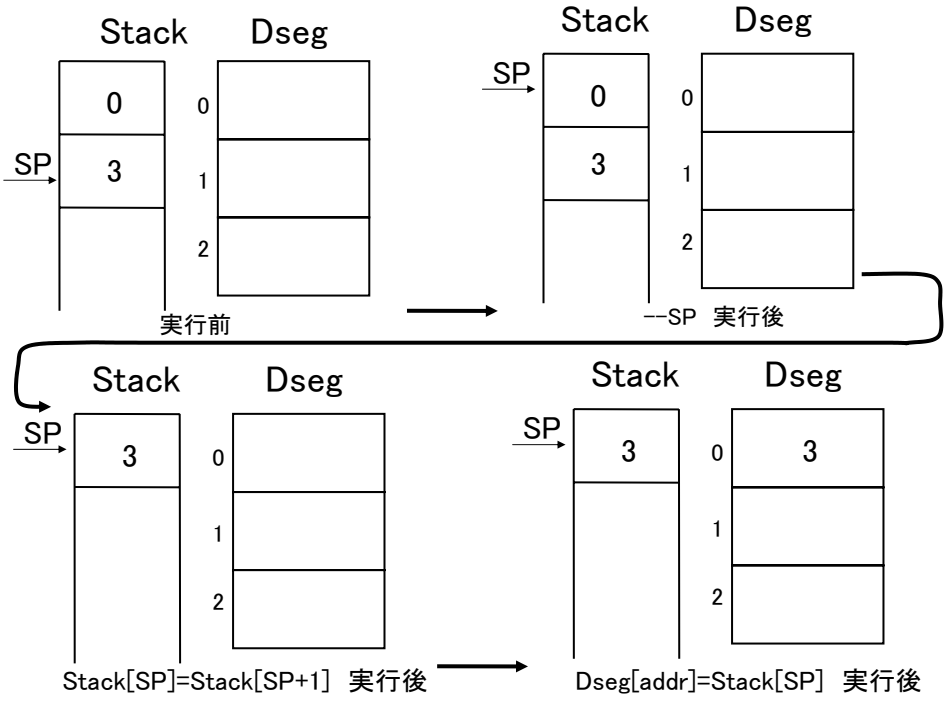


図 1 ASSGN の動作

図 1 に示す「実行前」の状態では，スタックトップに 3 が積まれており，SP がその位置を指している．この状態で ASSGN を実行すると，まず SP の値がデクレメントされ (「--SP 実行後」の状態)，SP が指す場所の中身，つまり 0 が 変数 addr に代入される．

次に，Dseg[addr] = Stack[SP] = Stack[SP+1] では，後ろの代入演算がまず実行される．すなわち，「実行前」までのスタックトップの値 Stack[SP+1] を，新たなスタックトップの値として Stack[SP] に代入される．この代入が実行された後の状態が図の「Stack[SP]=Stack[SP+1] 実行後」である．この状態で，Stack[SP+1] にももちろん 3 というデータは残っているが，これ以降は無視され，将来スタックに積まれる値によって上書きされる．変数 addr には，最初に 0 が保存されていたので，最後に Dseg[addr] = Stack[SP] によって，スタックトップの値 3 が Dseg の 0 番地に保存される．

ADD: BINOP(+);

BINOP は，次の命令で定義されるマクロである．

```
#define BINOP(OP) {Stack[SP-1] = Stack[SP-1] OP Stack[SP]; SP--;}
```

ADD，つまり BINOP(+) を実行すると上記マクロが展開され，OP の部分が + で置き換えられ，以下の 2 つの文に変わる．

```
Stack[SP-1] = Stack[SP-1] + Stack[SP]; SP--;
```

これらを実行すると，スタックの 2 番目の値にスタックトップの値が加算され，さらに SP-- によりスタックの 2 番目がスタックトップに変わるので，加算した結果がスタックトップに積まれている状態になる．次にあげる命令についても考え方は同様である．SUB(引き算)，MUL(掛け算)，DIV (割り算)，MOD(剰余演算)，AND (論理積)，OR(論理和)．

CSIGN: Stack[SP] = -Stack[SP];

スタックトップの値の符号 (正負) を反転させる命令である．

PUSH: Stack[++SP] = Dseg[addr];

オペランド addr をとり，Dseg の addr 番地の内容をスタックに積む命令である．

PUSHI: Stack[++SP] = addr;

オペランド addr で与えられる値を直接スタックに積む命令である．

REMOVE: --SP;

SP の値を一つ減らすことにより，スタックトップの内容を捨てる作用を持つ命令である．

POP: Dseg[addr] = Stack[SP--];

オペランド addr をとり，スタックトップの内容を取り出して Dseg の addr 番地に移す命令である．

COMP:

```
Stack[SP-1] = Stack[SP-1] > Stack[SP] ? 1 :
              Stack[SP-1] < Stack[SP] ? -1 : 0;
SP--;
```

COMP は，後で説明する BEQ や BLE 等の比較用命令，そして JUMP 命令と組合わせて，if 文や while

文などの制御構造を実現するための命令である．スタックの 2 番目の値 (Stack[SP-1]) とスタックトップの値 (Stack[SP]) を比較し，前者の方が大きければスタックの 2 番目の内容を 1 に，後者の方が大きければ −1 に，等しければ 0 に置き換える．そしてスタックポインタをデクレメントする．従って，比較した 2 つの値がスタックから消去され，1, − 1, 0 という比較結果を表す値だけがスタックトップに残る．

JUMP: `Pctr = addr;`

オペランド addr をとり，次に実行する命令を示す Pctr の値を addr とする．つまり Iseg の addr 番地までジャンプする．

BEQ: `if (Stack[SP--] == 0) Pctr = addr;`

オペランド addr をとり，スタックトップの値が 0 であれば，次に実行する命令を示す Pctr の値を addr とする (addr 番地へ分岐する)．あわせてスタックトップの内容を捨てる作用も持つ．BLT, BLE 等その他の比較用オペレータについては，付録 C を参照のこと．

INPUT: `scanf("%d%*c", &Stack[++SP]);`

キーボードから整数を一つ読み込み，スタックに積む命令．

INPUTC: `scanf("%c%*c", &Stack[++SP]);`

キーボードから文字を一つ読み込み，スタックに積む命令．

OUTPUT: `printf("%15d", Stack[SP--]);`

スタックトップにある int 型データを，整数として標準出力に出力する．スタックトップの内容を捨てる作用も持つ．

OUTPUTC: `printf("%c", Stack[SP--]);`

スタックトップにある int 型データを，文字として標準出力に出力する．スタックトップの内容を捨てる作用も持つ．

OUTPUTLN: `printf("\n");`

改行文字を標準出力に出力する．

LOAD: `Stack[SP]=Dseg[Stack[SP]];`

スタックトップにある整数を Dseg の番地と見なし，Dseg のその番地に保存されている整数で，スタックトップを置き換える命令．配列操作などで使用する．

COPY: `++SP; Stack[SP]=Stack[SP-1];`

スタックトップにある整数をもう一つスタックに積む命令．++ 演算子や +=演算子などで利用する．

例 2.1（if 文の実現方法） VSM アセンブラの命令をどのように組み合わせれば，if 文の動作を実現できるか例示する．左下のような K23 言語のプログラム文を考える．この文に対する VSM アセンブラの命令列は右下のようなものになる．

```
if (x == y)
      x = 3;
y = 4;
```

```
 8: PUSH    0
 9: PUSH    1
10: COMP
11: BEQ     14
12: PUSHI   0
13: JUMP    15
14: PUSHI   1
15: BEQ     20
16: PUSHI   0
17: PUSHI   3
18: ASSGN
19: REMOVE
20: PUSHI   1
21: PUSHI   4
22: ASSGN
23: REMOVE
```

以下，このアセンブラプログラムの動作を，図 2 を用いて説明する．

- すでに変数 x, y のための Dseg 上の領域はそれぞれ 0 番地と 1 番地に確保され，0 番地に 2，1 番地に 3 が入っているものとする．
- x と y の値を比較するために，スタックにまず x と y の値を PUSH する．このためのコードが VSM アセンブラの Pctr: 8 と 9 である．
- Pctr: 10 で，COMP 命令を用いてスタックトップとスタックの 2 番目の値を比較する．この操作の詳細については本節の COMP の説明を参照のこと．
- x == y が真の場合にはスタックに真理値 true を表す整数値 1 を，偽の場合にはスタックに真理値 false を表す整数値 0 を積まなければならない．

  x == y が真の場合　つまり，スタックトップに 0 がある場合には Pctr: 11 の BEQ により Pctr: 14 に分岐し，スタックに 1(すなわち true) をプッシュ (PUSHI) する．

  x == y が偽の場合　つまりスタックトップの値が 0 ではない場合には，Pctr: 11 の BEQ では分岐せず，Pctr: 12 でスタックに 0(すなわち false) をプッシュして，Pctr: 13 で JUMP 命令により Pctr: 15 に飛ぶ．

  この例の場合，x == y は成り立たないので，上記の「x == y が偽の場合」の動作を行う．
- if 文では，スタックトップが 1(x == y が真) の場合「x = 3」を実行し，スタックトップが 0(x == y が偽) の場合には「x = 3」を実行しない．x == y が偽の場合は x = 3 の処理 (Pctr: 16〜19) を飛ばして y = 4 を実行する．このために Pctr: 15 で BEQ のよりスタックトップの値が 0 であるかどうか

調べる．図 2 の場合，スタックトップの値が 0(x == y が偽) なので，Pctr: 20 に分岐する．

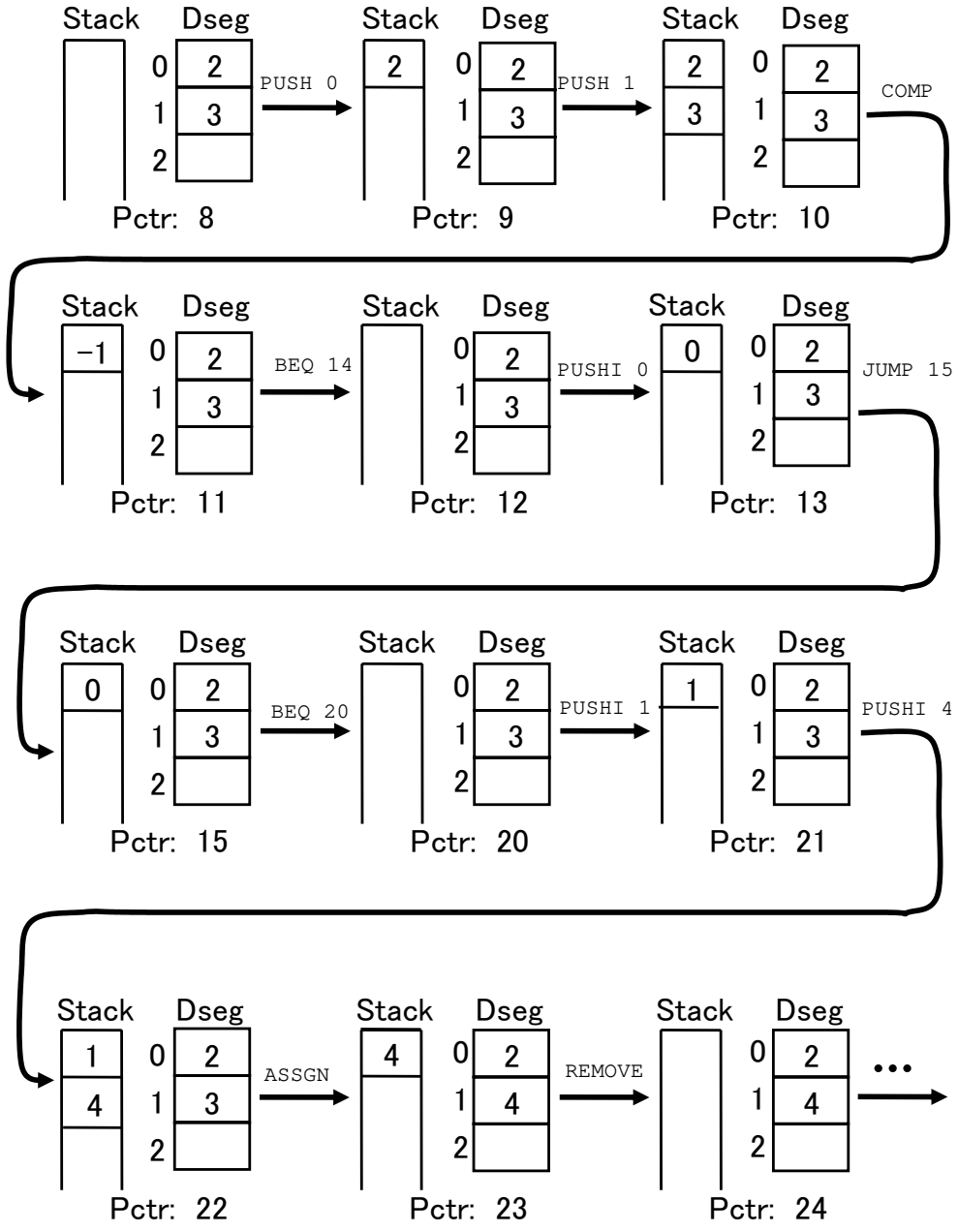


図 2 if 文の動作

- Pctr: 20 と 21 で，それぞれスタックに 1 と 4 を積んでいる．y = 4 の準備である．
- Pctr: 22 で，スタックトップの値を，スタックの 2 番目の値が示す Dseg の番地に格納する．これで y に 4 が代入されたことになる．
- Pctr: 23 で，不要になったスタックトップの値 4 を捨てている．

問題 2.8 例 2.1 では，if 文の条件式が成り立たない場合の VSM の動作を示した．if 文の条件式が成り立つ場合，例えば実行前の x と y の値がともに 2 である場合の VSM の動作を図 2 にならって図示せよ．

問題 2.9

1. 例 付録 D.1 と例 2.1 を参考に，以下に示す K23 言語プログラムを VSM アセンブラプログラムに変換せよ．教員側で用意した K23 コンパイラを使うのではなく，例を参考に手動で変換すること．

```
main() {
     int x, y;
     x = inputint;
     y = inputint;
     if (x < y)
         x = x + 1;
     if (y < x)
         y = y + 1;
     outputint(x);
     outputint(y);
}
```

2. この問題の 1. で作成したプログラムを実際に VSM で動作させ，変換が正しいことを確認せよ．(x<y の場合，x==y の場合，x>y の場合の 3 通りで確認し，結果を実習ノートに記述すること)

例 2.1 で比較を行う部分のアセンブラプログラムは以下の 5 行である．問題 2.9 の比較部分も同様のアセンブラプログラムを書けば良い．ただし，Pctr 11 のオペレータは，右の表に示す演算子に対応するオペレータに入れ替える．また，アセンブラプログラムを記述したアドレスに応じて Pctr 11, 13 の飛び先のアドレスは適切なものに書き換える必要がある．

```
10: COMP
11: BEQ     14 (3 行先へ)
12: PUSHI   0
13: JUMP    15 (2 行先へ)
14: PUSHI   1
```

演算子	オペレータ
==	BEQ
!=	BNE
<	BLT
<=	BLE
>	BGT
>=	BGE

2.3.2 VarTable の作成

変数を管理するためのプログラム，VarTable.java と Var.java を作成する．例付録 D.1 で見たように，K23 言語プログラム中に表れる全ての変数に対し，Dseg 上に領域を確保する必要がある．変数 i に Dseg の 0 番地を，変数 j に Dseg の 1 番地を，というように，変数名 (または配列名) と Dseg 上の番地の対応を管理することが，VarTable の主たる役割である．このためには，例えば一つの変数の情報を Var クラスの一つのインスタンスに保存することにし，VarTable のインスタンスは，Var のインスタンスを要素とする ArrayList を持つようにすればよい．

問題 2.10 以下に Var.java の仕様を示す．この仕様を満たすプログラムを作成せよ．

フィールド

private Type type ： Type は enum 型のクラスであり，Type.INT, Type.ARRAYOFINT, Type.NULL のいずれかの値を持つ (p.51 参照).

private String name ：変数名.

private int address ：Dseg 上のアドレス.

private int size ：配列の場合，そのサイズ.

コンストラクタ

Var(Type type, String name, int address, int size) ： 各フィールドを引数で与えられたもので初期化する.

メソッド 各フィールドを参照するための以下のゲッターメソッドを持つ.

Type getType()

String getName()

int getAddress()

int getSize()

問題 2.11 以下に VarTable.java の仕様を示す．この仕様を満たすように，p. 20 に掲載するプログラム例 2 を完成させよ.

フィールド

private ArrayList<Var> varList ：変数表

private int nextAddress ：次に登録される変数のアドレス

コンストラクタ

VarTable() ：ArrayList<Var> を一つ作り，varList で参照する．nextAddress を 0 に初期化する.

メソッド

private Var getVar(String name) ：varList 中から，引数で与えられた名前 name を持つ変数 (Var クラスのインスタンス) を探し，その参照を戻り値として返す．そのような変数が存在しない場合 null 値を返す．VarTable クラス内部からのみ呼び出される.

boolean exist(String name) ： 引数で与えられた名前 name を持つ変数が既に存在するかどうかを調べ，戻り値として返す.

boolean registerNewVariable(Type type, String name, int size) ：引数で与えられた型，名前，サイズを持つ変数を登録する．登録できたら戻り値 true を返す．既に varList 中に同じ名前の変数が存在する場合は登録せず，戻り値 false を返す.

int getAddress(String name) ：名前 name を持つ変数に与えられている Dseg のアドレスを求めて，戻り値として返す.

Type getType(String name) ：名前 name を持つ変数の型を戻り値として返す.

boolean checkType(String name, Type type) ：第 1 引数 name で与えられた変数の型が第 2 引数 type と一致するかを確認する.

int getSize(String name) ：名前 name を持つ変数のサイズを返す.

int size() ：変数表に登録されている変数の個数を返す.

void removeTail (int index) 引数で指定した位置から後の変数を変数表から削除する.

メインメソッド 以下に述べる動作を行うように作成する.

1. VarTable のインスタンスを一つ作成し，それを varTable という変数で参照する．
2. for 文を使って varTable に registerNewVariable を 4 回行わせ，varTable に要素を 4 つ追加する．追加する際の変数名は var0, var1, var2, var3, 型は整数型スカラー変数を表す INT, サイズは 1 でよい．
3. registerNewVariable を用いて varTable に要素を追加する．型は整数型配列変数を表す ARRAYOFINT, 変数名は var4, サイズは 10 とする．
4. varTable に追加した全ての変数に対し，checkType(変数名，Type.INT) が真ならば getType, getAddress を行い，その結果を表示する．表示の際には，それが何のデータなのかわかるように適当な説明のための文字列も出力すること．また，checkType(変数名，Type.INT) が真でない場合には checkType(変数名，Type.ARRAYOFINT) を調べ，これが真ならば getType, getAddress, getSize を行い，その結果を表示する．

プログラム例 2　VarTable.java

```
package kc;
import java.util.ArrayList;
class VarTable {
    /* フィールドの定義 */

    VarTable() {
        /* コンストラクタの処理を記述する */
    }
    /** 変数表から変数名で指定した変数を取り出すメソッド */
    private Var getVar(String name) {
        /* 表を検索し，name という名前の変数が存在すればその参照を返す，
           存在しなければ null を返す．*/
    }
    /** 既に変数表に登録されている変数名かどうかを調べるメソッド */
    boolean exist(String name) {
        /* name という変数名が表に存在すれば true，存在しなければ false を返す．*/
    }
    /** 表に新しい変数を登録するためのメソッド */
    boolean registerNewVariable(Type type, String name, int size) {
        /* 名前の重複チェック．既存の変数名なら false を返す．  */
        /* 型情報 type，変数名 name と 変数サイズ size から新たな Var の
           インスタンスを作り，それを ArrayList varList(表の本体) に追加する．*/
        /*  nextAddress を追加した変数のサイズ分だけ増やす．追加できたら true を返す．*/
    }


// 次ページに続く
```

```
    /** 変数表から変数のアドレスを得るメソッド */
    int getAddress(String name) {
        /* name という変数名が表に存在すれば，その変数のアドレスを返す．
           表にその変数が存在しない場合は -1 を返す．*/
    }
    /** 変数表から変数の型を得るメソッド */
    Type getType(String name) {
        /* name という変数名が表に存在すれば，その変数の Type を返す．
           表にその変数が存在しない場合は Type.NULL を返す．*/
    }
    /** 変数の型を確認するメソッド
    boolean checkType(String name, Type varType) {
        /* 変数名が name である要素の type フィールドが varType と一致するかを調べ
           一致すれば true，一致しなければ false を返す．*/
    }
    /** 表から変数のサイズを得るメソッド */
    int getSize(String name) {
        /* name という変数名が表に存在すれば，その変数のサイズを返す．
           表にその変数が存在しない場合は -1 を返す．*/
    }
    /** 変数表に登録されている変数の個数を返すメソッド */
    int size() {
        /* varList の size() に委譲する */
    }

    /** 引数で指定した位置から後の変数を変数表からを削除するメソッド */
    void removeTail (int index) {
        /* index と size() を比較し，同じまたは index の方が大きければ何もしない．
           nextAddress の値を，varList の index 番目に登録されている変数のアドレスの
           値にする．
           varList から index 番目〜size()-1 番目の変数を削除する．*/
    }

    public static void main(String[] args) {
        /* 仕様にある通りの動作を記述する */
    }
}
```

3 字句解析プログラムの作成 (第 4〜6 週)

字句解析プログラムは入力ファイル中の文字を FileScanner クラスを用いて一文字ずつ読んでいき，K23 言語のマイクロ構文に従って空白文字 (W-Space) を読み飛ばしてトークン (Token) を最長一致原則に基づいて順次切り出して，構文解析プログラムに渡す役割りを持つ．切り出したトークンに関する情報は Token クラスのインスタンスとして，構文解析プログラムに渡す．

3.1 字句解析プログラムの仕様

3.1.1 トークンクラスの仕様

切出したトークンを格納するクラス Token.java は以下の仕様を満たすものとする．

フィールド

private Symbol symbol ：そのトークンの種別を表す．Symbol は enum 型のクラス (p.50 参照)．

private int intValue ：トークンの種別が整数 (INTEGER) または文字 (CHARACTER) であるとき，その整数値あるいは文字コードを保持する．

private String strValue ：トークンの種別が名前 (NAME) または文字列 (STRING) であるとき，それを表す文字列を保持する．

コンストラクタ

Token(Symbol symbol) ：整数，文字，名前以外のトークンを生成するための，トークンの種別のみを引数とするコンストラクタ．

Token(Symbol symbol, int intValue) ：整数，文字のトークンを生成するための， トークンの種別と値 (整数値もしくは文字コード) を引数とするコンストラクタ．

Token(Symbol symbol, String strValue) ：名前，文字列のトークンを生成するための，トークンの種別と文字列を引数とするコンストラクタ．

メソッド

boolean checkSymbol(Symbol symbolType) ： symbol フィールドが，引数 symbolType とトークン種別と一致するかどうかを調べる．

Symbol getSymbol() ：symbol フィールドのゲッター

int getIntValue() ：intValue フィールドのゲッター

String getStrValue() ：strValue フィールドのゲッター

3.1.2 字句解析クラスの仕様

プログラム例 3 に示す字句解析プログラム LexicalAnalyzer.java を作成する．LexicalAnalyzer.java は以下の仕様を満たすものとする．

フィールド

private FileScanner sourceFileScanner ：ソースファイルに対するスキャナ

コンストラクタ

LexicalAnalyzer(String sourceFileName) ：ファイル名を引数とするコンストラクタ．

メソッド

Token nextToken() ： 次のトークンを切り出す．ファイル末に達している場合（'\0' を読んだ場合）は EOF を返す．入力がマイクロ構文に違反したためトークンの切り出しに失敗した場合，syntaxError() メソッドを呼び出す．

void closeFile() ：読んでいるファイルを閉じる．

String analyzeAt() ： 現在，入力ファイルのどの部分を解析中であるのかを表現する文字列を返す．sourceFileScanner の scanAt メソッドを呼び出せばよい．

private void syntaxError() ： 字句解析時に構文エラーを検出したときに呼ばれるメソッド．プログラム例 3 のとおりに作成すること．

プログラム例 3　LexicalAnalyzer.java

```
package kc;
class LexicalAnalyzer {
    private FileScanner sourceFileScanner;  /* ソースファイルに対するスキャナ */

    LexicalAnalyzer(String sourceFileName) {
        /* sourceFileName という名前のファイルのためのファイルスキャナを生成し，
                                              sourceFileScanner により参照する */
    }

    /** トークンの切り出し */
    Token nextToken() {
        /* トークンが切り出せたら，Token クラスのインスタンスを生成し，返り値とする
           ファイル末に達している場合は EOF を返す
           トークンが切り出せなかったらメソッド syntaxError を呼び出す */
    }

    /** 現在読んでいるファイルを閉じる（sourceFileScanner の closeFile に委譲）*/
    void closeFile() {
        sourceFileScanner.closeFile();
    }

    /** 現在の解析場所を表す文字列を返す（sourceFileScanner の scanAt に委譲）*/
    String analyzeAt() {
        return sourceFileScanner.scanAt();
    }

    private void syntaxError() {
        System.out.print(sourceFileScanner.scanAt());
        System.out.println("で字句解析プログラムが構文エラーを検出");
        closeFile();
        System.exit(1);
    }
}
```

3.2 第 4 週：SLexicalAnalyzer の作成

いきなり K23 言語のフルセットに対する字句解析プログラムを作るのは少々難しいかもしれない．一つの方法として，まずは手始めに K23 言語の一部のトークンを切り出す字句解析プログラムを作成し，それを拡張する形で K23 言語のフルセットに対する字句解析プログラムを作る方法が考えられる．

以下では，下記に示すマイクロ構文に対する字句解析プログラムの構成について考えて行くことにする．

```
Subset::= {Token | W-Space}'\0'
Token::= INT | OPERATOR
W-Space::= ' ' | '\t' | '\n'

INT::= '0' | Pdec {Dec}

OPERATOR::= '=' '=' | '!' '=' | '+' | '!' | '='
```

字句解析プログラムは W-Space を読み飛ばし，最長一致の原則に基づいてトークンを切り出すものである．W-Space の読み飛ばしは空白とタブと改行文字を読んだら何もせずに次の文字を読みにいけばよい．これを直接プログラム化するのは容易である．

一方，トークンを切り出すプログラムを構成するためには，まずはトークンを受理する DFA(決定性有限オートマトン) を構成し，その DFA をプログラム化するという手順を踏むのが一般的である．DFA の構成は教科書どおり正規表現から NFA(非決定性有限オートマトン) を求め，さらにそれを DFA に変換し，状態数を最小化するという方法をとってもよいが，もう少し直観的なやり方で構成できる．例えば以下のような考え方で構成すればよい (図 3 参照)．

- `INT::= '0' | Pdec {Dec}`

  より，以下の状態および状態遷移を導入する．
  - 初期状態で $1, \ldots, 9$ のいずれかを読むと「Pdec {Dec} を読んだ状態」へ遷移する．「Pdec {Dec} を読んだ状態」は受理状態であり，その状態で $0, \ldots, 9$ のいずれかを読むと「Pdec {Dec} を読んだ状態」へ遷移する．
  - 初期状態で'0' を読むと「'0' を読んだ状態」へ遷移する (図 3 の '0')．「'0' を読んだ状態」は受理状態であり，そこからの状態遷移は無い．
- `OPERATOR::= '=' '=' | '!' '=' | '+' | '!' | '='`

  より，以下の状態および状態遷移を導入する．
  - '+', '!', '=' がそれぞれトークンなので，初期状態で +, !, =を読むとそれぞれ「'+' を読んだ状態」,「'!' を読んだ状態」「'=' を読んだ状態」へ遷移するものとし，3 つとも受理状態とする．
  - '!' '=', '=' '=' がそれぞれトークンなので,「'!' を読んだ状態」「'=' を読んだ状態」で=を読むとそれぞれ「'!' '=' を読んだ状態」「'=' '=' を読んだ状態」へ遷移しそれぞれ受理状態とする．

最終的に得られた DFA が図 3 である．状態数の最小化についても，あまりこだわる必要は無い．

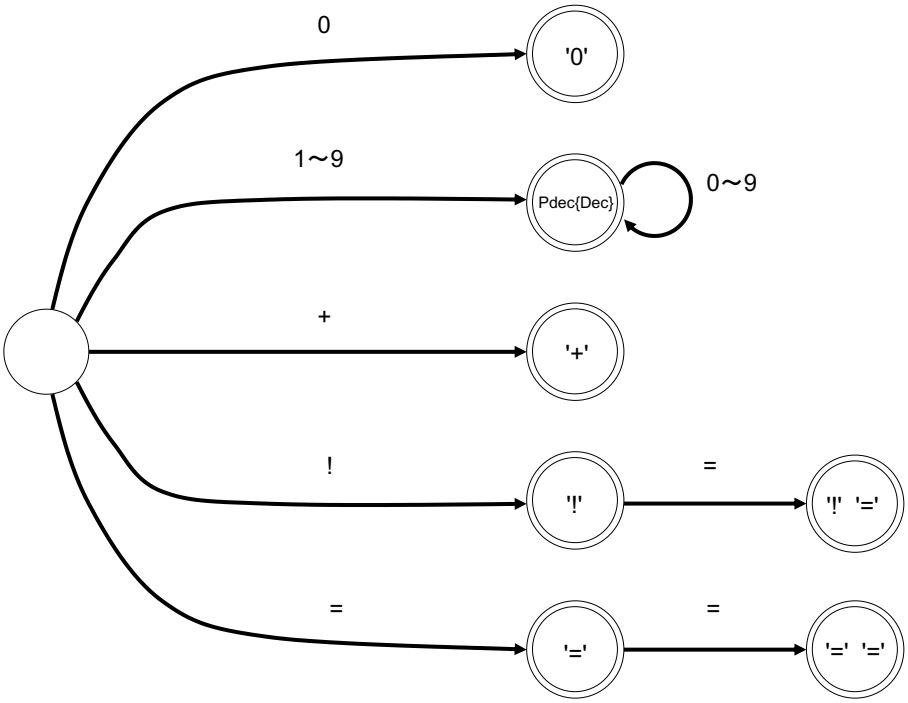


図 3 Subset のトークンを受理する DFA

次に，この DFA を基に字句解析プログラムを構成すればよい．いくつかの注意点，ヒントを以下に列挙しておく．

- 最長一致原則に基づいてトークンを受理するためには，1 文字先読みの機能が必要になる．例えば '!' を読んでも直ちにトークンとして切り出すのではなく，次の文字が '=' でないときに限って '!' をトークンとして切り出すようにしなければならない．このため FileScanner で `lookAhead` というメソッドを用意したのである．これを有効に活用すること．
- `Character` クラスのメソッド `isDigit`, `digit` を使うとプログラムをすっきり作ることができる．どのようなものか調べ，有効に活用せよ．情報源として，`http://docs.oracle.com/javase/jp/8/docs/api/` 等の資料を活用すれば良い．

問題 3.1（Token） まず，仕様に従って Token.java を作成せよ．SLexicalAnalyzer では名前および文字列は扱わないが，Token はこの段階から名前，文字列を扱うことを想定したものを作成すること．

問題 3.2（SLexicalAnalyzer） 上記のマイクロ構文に対する字句解析プログラム SLexicalAnalyzer.java を作成せよ．プログラムが構成できたら，こちらで用意する SLexerTester を使ってトークンの切り出しが正しく行なわれているかどうか，検査用に用意した `SLT.k` (p. 42) を入力として確認せよ．期待される出力は以下の通り．

実行結果

```
Integer 12
=
Integer 10
+
Integer 8
Integer 13
==
Integer 5
!
Integer 12
Integer 12
!=
Integer 0
Integer 0
Integer 13
==
Integer 5
Integer 14
!=
Integer 10
```

3.3 第 5〜6 週：LexicalAnalyzer の作成

SLexicalAnalyzer の動作確認ができれば，字句解析プログラム作成のためのもろもろのノウハウは身についているはずである．同様の手順で K23 言語全体に対する字句解析プログラムを作成せよ．

isDigit, digit と同様にプログラムを構成する上で有用となると思われる Character, String クラスのメソッドとして以下のものがあるので，有効に活用すること．

- isLowerCase
- isUpperCase
- equals
- valueOf

各々の使い方は，http://docs.oracle.com/javase/jp/8/docs/api/ 等の資料を参考にすればよい．また，char 型データも int 型のデータと同様に比較演算子を用いて大小比較が可能で，例えば char 型の変数 c の値が'B' のとき，'A' < c は真であり，'D' < c は偽である．16 進数の認識では char 型データの大小比較をうまく活用すること．

問題 3.3（LexicalAnalyzer） K23 言語に対する字句解析プログラム LexicalAnalyzer.java を作成せよ．プログラムが構成できたら LexerTester を使ってトークンの切り出しが正しく行なわれているかどうか確認せよ．

問題 3.4 さらに，さまざまなマイクロ構文違反を含む入力ファイルを作成し，LexicalAnalyzer.java がそれらの違反を検出できるかを LexerTester を使って確認せよ．

4 構文解析プログラムの作成 (第 7〜9 週)

字句解析プログラムが完成したら，いよいよコンパイラ本体，Kc.java の作成である．これは，まず再帰降下型構文解析法により構文解析プログラムを構成し，次にセマンティックアクションを埋め込むという手順で行うことになる．

構文解析プログラムは，基本的には以下の手順に従って作成すれば構成できる．

1. 与えられたマクロ構文の左再帰性の除去を行う．左再帰性の除去を行うと演算子の結合性の情報が失われるので，「a-b-c」のように同一優先順位の演算子が並んでいる場合は左右どちらが優先されるのかも記載すること．
2. さらにマクロ構文の左括り出しを行う．必要なら新しい非終端記号を導入する．
3. アルゴリズム付録 E.1 により各非終端記号 A について FIRST(A) を求める．これらさえ求めておけば，任意の文法記号列 α に対する FIRST(α) を簡単に求めることが出来る．
4. アルゴリズム付録 E.2 に従って各非終端記号 A に対するメソッド `parseA` を構成する．

4.1 第 7 週：準備

構文解析プログラムの準備として，マクロ構文の左再帰性の除去，左括り出し，各非終端記号に対する FIRST 集合を手作業により求める．実習ノート p.19 を利用して，それらの作業を実施せよ．

4.2 第 8〜9 週：プログラムの作成

4.2.1 構文解析クラスの仕様

この段階での Kc.java は以下の仕様を満たすものとする．

フィールド

private LexicalAnalyzer lexer ：使用する字句解析器

private Token token ：字句解析器から受け取ったトークン

コンストラクタ

Kc(String sourceFileName) ：ソースファイル名を引数とするコンストラクタ．

メソッド

void parseProgram() ：再帰降下型構文解析を行うメソッド．入力が構文規則に違反していることを検出した場合，エラーの内容を表す文字列を引数として `syntaxError` メソッドを呼び出す．

void closeFile() ：現在読んでいるファイルを閉じるメソッド．

private void syntaxError() ：構文エラーに関する情報を表示して，終了するメソッド．

メインメソッド コマンド行引数で指定された名前のファイルを構文解析する `Kc` クラスのインスタンスを生成し，そのインスタンスに構文解析させ，正常終了するとファイルのクローズを行う．

4.2.2 構文解析プログラムの構成

アルゴリズム付録 E.1, 付録 E.2 を用いて各非終端記号 A に対するメソッド parseA を構成する．プログラム全体の構成は，プログラム例 4(p. 28) のような形にすればよい．

プログラム例 4 Kc.java

```
package kc;
public class Kc {
    private LexicalAnalyzer lexer;    // 字句解析器
    private Token token;              // 字句解析器からもらうトークン

    Kc(String sourceFileName) {
        /* 字句解析器のインスタンス生成と lexer による参照 */
    }

    /** parseProgram 以下，各非終端記号 A に対する parseA メソッド */
    void parseProgram() {
        /* parseProgram の処理の記述 */
    }

    /** 現在読んでいるファイルを閉じる (lexer の closeFile に委譲) */
    void closeFile() {
        /* closeFile の処理の記述 */
    }

   /** 文法エラー出力 */
    private void syntaxError(String message) {
        System.out.print(lexer.analyzeAt());
        System.out.println("で構文解析プログラムが構文エラーを検出");
        System.out.println(message);
        closeFile();
        System.exit(1);
    }

    public static void main(String[] args) {
        Kc parser;                    // 構文解析器

        if (args.length == 0) {       // 実行時パラメータのチェック
            System.out.println("Usage: java kc.Kc file [objectfile]");
            System.exit(1);
        }

        parser = new Kc(args[0]);

        parser.parseProgram();        // 構文解析
        parser.closeFile();           // 入力ファイルのクローズ
    }
}
```

実際にプログラムがどのように構成されていくか，少しだけ見てみることにしよう．例として，以下に示す「式」についてのマクロ構文 (K23 の Expression とは異なるので注意すること) を考える．

```
<Expression> ::= <Exp>
                 [ ( "=" | "+=" | "-=" | "*=" | "/=" ) <Expression> ]
```

このような構文に対しては，アルゴリズム付録 E.2 の規則を用いて以下のような手順でプログラムが構成できる．

1. 規則 (5) より<Exp>と [("=" | "+=" | "-=" | "*=" | "/=") <Expression>] に対するコードを連接すればよい．
2. 規則 (3) より<Exp> に対するコードは，例えば「parseExp();」等でよい．<Expression>に対するコードも同様．
3. 規則 (2) より "="に対するコードは

   ```
   if (token.checkSymbol(Symbol.ASSIGN)) token = lexer.nextToken();
   else syntaxError("'=' が期待されます");
   ```

   でよい．"+=" などについてのコードも同じようにして生成できる．
4. 規則 (4) より ("=" | "+=" | "-=" | "*=" | "/=") に対するコードは

   ```
   switch (token.getSymbol()) {
   case ASSIGN:
       if (token.checkSymbol(Symbol.ASSIGN)) token = lexer.nextToken();
       else syntaxError("'=' が期待されます");
       break;
   case ASSIGNADD:
       .  .  .
   case ASSIGNDIV:
       if (token.checkSymbol(Symbol.ASSIGNDIV)) token = lexer.nextToken();
       else syntaxError("'/=' が期待されます"););
       break;
  }
   ```

   となる．
5. 規則 (7) より [("=" | "+=" | "-=" | "*=" | "/=") <Expression>] に対するコードは

   ```
   if (token.checkSymbol(Symbol.ASSIGN)
       || token.checkSymbol(Symbol.ASSIGNADD)
       || token.checkSymbol(Symbol.ASSIGNSUB)
       || token.checkSymbol(Symbol.ASSIGNMUL)
       || token.checkSymbol(Symbol.ASSIGNDIV)) {
       .  .  .
   }
   ```

とすればよい.

このように，生成規則の右辺に「|」や「[...]」,「{...}」を含む非終端記号 A に対する parseA の構成には FIRST(α) を求める必要が出て来るが，すでに各非終端記号 X について FIRST(X) を求めているので，たいして困難な作業ではないはずである.

以上より，parseExpression がプログラム例 5 のような形で構成できる.

プログラム例 5 メソッド parseExpression

```
private void parseExpression() {
    parseExp();
    if (token.checkSymbol(Symbol.ASSIGN)
        || token.checkSymbol(Symbol.ASSIGNADD)
        || token.checkSymbol(Symbol.ASSIGNSUB)
        || token.checkSymbol(Symbol.ASSIGNMUL)
        || token.checkSymbol(Symbol.ASSIGNDIV)) {
        switch (token.getSymbol()) {
        case ASSIGN:
            if (token.checkSymbol(Symbol.ASSIGN)) token = lexer.nextToken();
            else syntaxError("'=' が期待されます");
            break;
        case ASSIGNADD:
            . . .
        case ASSIGNDIV:
            if (token.checkSymbol(Symbol.ASSIGNDIV)) token = lexer.nextToken();
            else syntaxError("'/=' が期待されます");
            break;
        }
        parseExpression();
    }
}
```

問題 4.1（構文解析プログラムの作成） K23 言語の構文解析プログラムを作成せよ．プログラムが構成できたら，以下のような方法で動作試験を行なえ.

1. Kc を プログラムの引数として bsort.k を指定して実行し，エラーなく実行を完了できるか.
2. bsort.k に意図的に構文誤りを仕込んだプログラムの構文誤りを正しく検出，表示できるか.
3. さまざまな種類のマクロ構文違反 (問題 2.2 で作成したものなど) について，同様の作業を繰り返す.

注意事項とヒント

1. アルゴリズム付録 E.2 に厳密に従う必要は無く，むしろ後でセマンティックアクションを埋め込む際の便宜を考慮してプログラムが読みやすく，追加・変更が容易なように留意して作成するほうが良い．また，明らかに無駄な処理は省くこと．例えばアルゴリズム付録 E.2 に厳密に従うと，プログラム例 6 に示すようなコードが得られるが,

プログラム例 6

```
switch (token.getSymbol()) {
case ASSIGN:
    if (token.checkSymbol(Symbol.ASSIGN)) token = lexer.nextToken();
    else syntaxError("'=' が期待されます");
    . . .
```

無駄な処理を省いて，プログラム例 7 に示すようなコードを作成すれば良い．

プログラム例 7

```
switch (token.getSymbol()) {
case ASSIGN:
    token = lexer.nextToken();
    . . .
```

2. 段階的に作るとすれば，まずは，以下のようなマクロ構文についての構文解析プログラムを作り，変数宣言部のみからなるプログラムの解析，エラーチェックが正しく行なわれるか試してみるのも一つの手である．

```
<Program>::= <Main_function>
<Main_function>::= "main" "(" ")" <Block>
<Block>::= "{" {<Var_decl>} "}"
<Var_decl>::= "int" <Name_list> ";"
<Name_list>::= (<Name_list> "," <Name>) | <Name>
<Name> ::= NAME | (NAME "=" <Constant>) | (NAME "[" INT "]")
        | (NAME "[" "]" "=" "{" <Constant_list> "}")
<Constant_list> ::= (<Constant_list> "," <Constant>) | <Constant>
<Constant>::= ( ["-"] INT  ) | CHAR
```

この構文解析プログラムが正しく構成できれば，基本的なノウハウは身についているはずである．あとは parseBlock を

```
<Block>::= "{"  { <Var_decl> } { <Statement> } "}"
```

に対応する形に書き換え，parseStatement 以下のメソッドを順次追加していけばよい．

5 セマンティックアクション (第 10〜12 週)

5.1 プログラムの構成

構文解析プログラムが完成したら，あとはそこにセマンティックアクションを埋め込めばコンパイラの完成である．セマンティックアクションは大きく変数表の管理，参照と，アセンブラコードの生成に分けられる．変数表の管理，参照を支援するためのプログラムとして，すでに VarTable というクラスを作成した．アセンブラコードの生成を支援するプログラムとしては PseudoIseg というクラスを用意している．

セマンティックアクションの記述は VarTable と PseudoIseg のメソッドを順次呼び出す形で行なえる．

5.1.1 PseudoIseg

2 章で解説したように，アセンブラプログラムを VSM で実行する際，そのプログラムは Iseg と呼ばれるメモリ領域に読込まれる．K23 言語プログラムを解析し，各構文要素に対応する VSM 命令を順次生成し一つのアセンブラプログラムを得るために，本実習では Instruction.java と PseudoIseg.java というプログラムを用意した．Instruction.java は一つの VSM 命令を格納するオブジェクトのクラスであり，命令コードを収容するフィールド op, オペランドを収容するフィールド addr などからなる．Pseudo は「シュード」と発音し「仮の」という意味である．従って PseudoIseg(シュードアイセグ) とは，仮の Iseg という意味になる．PseudoIseg の機能は次の通り．

コンストラクタ　引数無しコンストラクタのみ

メソッド

- int appendCode (Operator opCode) ：オペランドを持たない命令を追加するメソッド．返り値は，追加した命令の番地．
- int appendCode (Operator opCode, int operand) ：オペランドを持つ命令を追加するメソッド．返り値は，追加した命令の番地．
- int getLastCodeAddress() ：pseudoIseg に最後に加えた命令の番地を返す．
- void dump() ：その pseudoIseg の内容をディスプレイに表示する．
- void dump2file() ：その pseudoIseg の内容をファイル OpCode.asm というファイルに出力する．
- void dump2file (String outputFileName) ：その pseudoIseg の内容を outputFileName で指定した名前のファイルに出力する．
- void replaceCode (int ptr, Operator opCode) ：ptr 番地の命令の命令コードを opCode に変更する．
- void replaceCode (int ptr, int operand) ：ptr 番地の命令のオペランドを operand に変更する．
- boolean checkOperator (int ptr, Operator opCode) ：ptr 番地の命令コードが opCode に一致するかを返す．
- Operator getOperator (int ptr) ：ptr 番地の命令コードを返す．
- int getOperand (int ptr) ：ptr 番地のオペランドを返す．
- void removeCode (int ptr) ：ptr 番地の命令を削除する
- void removeLastCode() ：pseudoIseg に最後に加えた命令を削除する．

PseudoIseg の実際の使用方法を，プログラム例 8 の Test1pIseg.java で例示する．詳しい説明について

は，そのプログラム内のコメントを参照すること．

p. 33 にあるプログラムの前半で，VSM 命令を pseudoIseg に格納する方法を例示している．

p. 34 にあるプログラムの後半で，pseudoIseg に格納された各命令に対する置き換えや表示，ファイル出力方法などを例示している．

プログラム例 8　Test1pIseg.java

```
/*  一つの命令は Instruction クラスのインスタンスに格納され，そのインスタンスが
    命令表（PseudoIseg クラスのインスタンス）に格納される．

    このプログラムでは，命令を作って表に格納したり，命令のオペレータやオペランドを
    書き換えたり，全命令を表示したり，ファイルに出力する方法を例示する */

public class Test1pIseg {
    public static void main(String[] args) {
        PseudoIseg iseg = new PseudoIseg();   //命令を格納する iseg を作る

        /* PseudoIseg のメソッド appendCode を用いて、iseg に各命令を格納していく．
           appendCode の返り値は int だが，例示には必要ないのでとりあえず返り値は
           無視して作っている */
        iseg.appendCode(Operator.PUSHI,1);
        iseg.appendCode(Operator.INPUT);
        iseg.appendCode(Operator.ASSGN);
        iseg.appendCode(Operator.REMOVE);
        iseg.appendCode(Operator.PUSHI, 2);
        iseg.appendCode(Operator.INPUTC);
        iseg.appendCode(Operator.ASSGN);
        iseg.appendCode(Operator.REMOVE);
        iseg.appendCode(Operator.PUSH, 1);
        iseg.appendCode(Operator.OUTPUT);
        iseg.appendCode(Operator.OUTPUTLN);
        iseg.appendCode(Operator.PUSH, 2);
        iseg.appendCode(Operator.OUTPUTC);
        iseg.appendCode(Operator.OUTPUTLN);
        iseg.appendCode(Operator.HALT);


    //次ページに続く
```

```
    /* 前頁までで作った pIseg 内のアセンブラプログラムを操作する例 */


        /* iseg 内の全命令を表示 */
        System.out.println("全命令の表示");
        iseg.dump();

        /* 4 番地の命令の Operator を PUSH に変更 */
        iseg.replaceCode(4, Operator.PUSH);
        System.out.println("\n 4 番地の命令の Operator を PUSH に変更"
                                  +"\n した後の全命令の表示");
        iseg.dump();

        /* 11 番地の命令の Operand を 5 に変更 */
        iseg.replaceCode(11,5);
        System.out.println("\n 11 番地の命令の Operand を 5 に変更"
                               +"\n した後の全命令の表示");
        iseg.dump();

        /* OpCode.asm というファイルに iseg 内の全命令を出力 */
        iseg.dump2file();
        System.out.println("OpCode.asm ファイルに命令を出力しました.");

        /* test.asm というファイルに iseg 内の全命令を出力 */
        String outputFileName = "test.asm";
        iseg.dump2file(outputFileName);
        System.out.println(outputFileName + "ファイルに命令を出力しました.");

    }
}
```

5.1.2 Kc.java の概形

構文解析プログラムに VarTable と PseudoIseg クラスのインスタンスの定義と生成，および生成したアセンブラプログラムのファイルへの書き出しを追加したコンパイラのプログラムの概形を，プログラム例 9 に示す.

プログラム例 9　Kc.java

```
package kc;
public class Kc {
    private LexicalAnalyzer lexer;    // 字句解析器
    private Token token;              // 字句解析器からもらうトークン
    private VarTable variableTable;   // 変数表
    private PseudoIseg iseg;          // アセンブラコード表

    public Kc(String sourceFileName) {
      /* 字句解析器のインスタンス生成と lexer による参照 */
      /* 変数表のインスタンス生成と variableTable による参照 */
      /* アセンブラコード表のインスタンス生成と iseg による参照 */
    }

    /** parseProgram 以下，各非終端記号 A に対する parseA メソッド*/
    void parseProgram() {
        /* parseProgram の処理の記述 */
    }

    /** 現在読んでいるファイルを閉じる (lexer の closeFile に委譲) */
    void closeFile {
        . . .
    }

    /** OpeCode.asm へ iseg の内容を出力する (iseg の dump2file に委譲) */
    void dump2file() {
        . . .
    }

    /** 指定されたファイルへ iseg の内容を出力する (iseg の dump2file に委譲) */
    void dump2file(String fileName) {
        . . .
    }

    /** 文法エラー出力 */
    private void syntaxError(String message) {
        . . .
    }

    public static void main(String[] args) {
        Kc parser;                    // 構文解析器
        if (args.length == 0) {       // 実行時パラメータのチェック
            System.out.println("Usage: java kc.Kc file [objectfile]");
            System.exit(0);
        }
        parser = new Kc(args[0]);     // 構文解析器の生成
        parser.parseProgram();        // 構文解析
        parser.closeFile();           // 入力ファイルのクローズ
        if (args.length == 1) parser.dump2file();  // OpCode.asm へ出力
        else parser.dump2file(args[1]);            // 指定ファイルへ出力
    }
}
```

5.2 セマンティックアクション記述の基本方針

5.2.1 変数表の構成と参照

変数表はソースプログラム中に現れる変数名または配列名と Dseg 上の番地との対応を管理するものである．変数表はソースプログラム中の<Var_decl>(変数宣言部) の部分で構成され，<Expression>(式) の部分で参照される．

変数表は基本的に以下のような考え方に基づいて構成すればよい．

1. <Var_decl> の中で checkSymbol(Symbol.NAME) が true であるトークンを認識すると，getStrValue() で取り出した名前がすでに変数表に登録されているかを exist メソッドを用いて調べ，登録されていれば構文エラーとする．
2. 未登録の名前であれば，以降の構文解析でスカラー変数か配列変数かを決定することでその変数の型を決定し，配列変数ならそのサイズを求め，registerNewVariable を用いて変数表に登録する．

一方，<Expression> の中で checkSymbol(Symbol.NAME) が true であるトークンを認識した場合は以下の処理を行なえば良い．

1. getStrVaue() で名前の文字列を取り出し，その名前がすでに変数表に登録されているかを exist メソッドを用いて調べ，登録されていなければエラーとする．
2. その変数がスカラー変数か配列変数かを構文的に決定し，checkType メソッドを用いて型の整合性を調べ，矛盾していればエラーとする．
3. 変数のアドレスを getAddress メソッドを用いて求め，コード生成に利用する．

5.2.2 左辺値と右辺値

コード生成について考える前に，式中の変数または配列要素 (以降，まとめて変数と呼ぶ) の解釈について触れる．通常のプログラミング言語では変数が代入演算子の左側に表れる場合は値を代入する場所 (Dseg の番地) を示すものと解釈され，それ以外の場所に出現した場合はその変数に格納されている値を表すものと解釈され，それぞれ左辺値，右辺値と呼ばれている．もちろん K23 言語でもそのような解釈にそってコード生成を行うことになる．

5.2.3 コード生成

コード生成の例として，以下の構文規則で規定されている式に対してどのようなコードを生成すればよいのか，そのためのセマンティックアクションをどう書けばよいのかを考えてみよう．

```
<Expression>::= <Exp> [("=" | "+=" | "-=" | "*=" | "/=") <Expression>]
```

すでに見たように構文解析プログラムで，プログラム例 5 (p. 30) のような parseExpression を作成しているはずである (以下のプログラムはプログラム例 5 を少し整理したものである)．

```
private void parseExpression() {
    parseExp();
    if (token.checkSymbol(Symbol.ASSIGN)
        || token.checkSymbol(Symbol.ASSIGNADD)
        || token.checkSymbol(Symbol.ASSIGNSUB)
        || token.checkSymbol(Symbol.ASSIGNMUL)
        || token.checkSymbol(Symbol.ASSIGNDIV)) {
        switch (token.getSymbol()) {
        case ASSIGN:
            token = lexer.nextToken();
            parseExpression();
            break;
        case ASSIGNADD:
            .  .  .
        case ASSIGNDIV:
            token = lexer.nextToken();
            parseExpression();
            break;
        }
    }
}
```

”=”や”+=”のような代入演算子は左辺の<Exp>に対応する変数 (または配列要素) に<Expression>に対応する値を代入あるいは加減乗除するものである．変数は変数表により Dseg 上の番地に対応づけられているので，結局は Dseg 上の特定の番地に値を代入する処理になる．ところで，parseExp()，parseExpression() にも適切なセマンティックアクションが埋め込まれていると想定すれば，

- メソッド parseExp() は単に式を構文解析するだけでなく，その式が単一の変数 (または配列要素) のみからなるものであり，かつ引き続くトークンが代入演算子である場合，その変数の左辺値をスタックにプッシュするためのアセンブラコードを出力するメソッドである．
- メソッド parseExpression() は式を構文解析するだけでなく，その式の現在の評価値をスタックにプッシュするアセンブラコードを出力するメソッドである．

と考えて良い．例えばあるプログラムの実行時に，式の実行前のスタックの状態が図 4(a) のようになっていたとする．このとき parseExp によって生成されたコード実行後のスタックの状態は，図 4(b) になり，さらに parseExppression によって生成されたコード実行後のスタックの状態は図 4(c) になっているはずである．ここで，

ASSGN

を実行すると Dseg の「左辺の番地」番地に「右辺の評価値」が代入され，スタックの状態は図 4(d) となる．

結局, 演算子 (token.getSymbol()) が Symbol.ASSIGN である場合には上記コードの parseExpression() の後に ASSGN 命令を生成するコードを付け加えれば，parseExpression 全体で

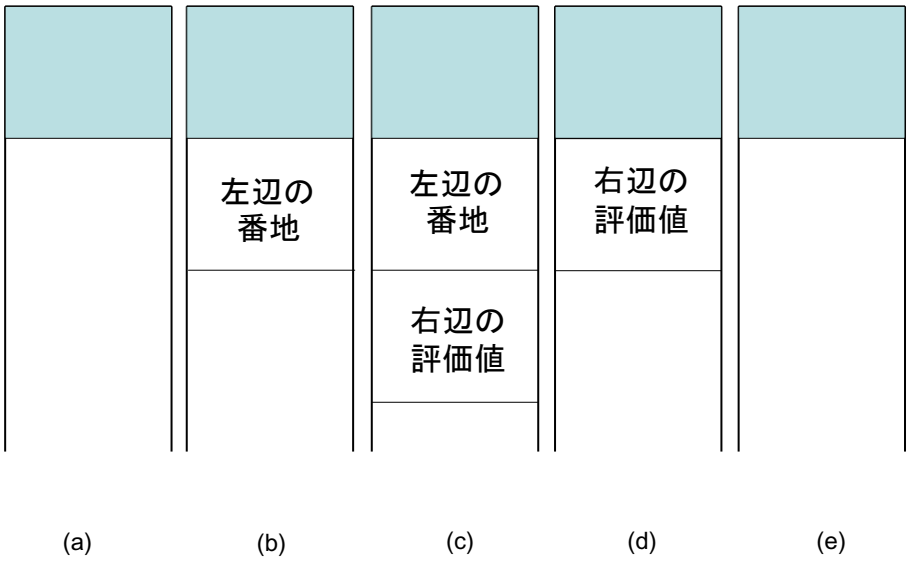


図 4 代入の実行手順

- 必要な代入処理を行い
- スタックに式の評価値をプッシュする

コードの生成が行なえることになる．ASSGN 命令を生成するコードは appendCode メソッドを用いればよい．

なお、このままではスタック上に式の評価値が残ったままなので，式文終了の";"を検出すると，スタック上の評価値を消去するために

```
REMOVE
```

を実行する．これによりスタックの状態は図 4(e) となる．

一方，演算子が"ASSIGNADD"である場合には，もう少し複雑で

1. parseExp() でその変数の左辺値をスタックにプッシュ
2. その番地をスタック上でコピー (COPY)
3. Dseg 上のスタックトップ番地のデータをロード (LOAD)
4. parseExpression() で式の現在の評価値をスタックにプッシュ
5. スタック上で演算を実行 (ADD)
6. ASSGN

という手順になる．

5.2.4 代入演算子の左被演算子の確認

K23 言語では代入演算子の左被演算子は変数 (または配列要素) でなければならないという制約がある．このため，式を構文解析する際，代入演算子の左に来る<Exp>が単一の変数 (または配列要素) のみからなるものであることを確認しなければならない．これを確認する方法について考える．

まず, 式の最小の構成単位である<Unsigned_factor>を構文解析する部分では, その<Unsigned_factor>が変数 (または配列要素) であるかどうかを決定できる．

<Exp>などのそれより大きな構成単位を構文解析する部分 (parseExp など) では，その部分式を解析するメソッド (parseLogicalTerm など) から，その部分式が単一の変数 (または配列要素) のみからなるかどうかをメソッドの戻り値として受け取るようにすればよい．それにより parseExp は自分自身が単一の変数 (または配列要素) のみからなるのかどうかを決定できる．

そして，parseExpression では，代入演算子を認識したとき，その左被演算子である<Exp>が単一の変数 (または配列要素) のみからなるかどうかを戻り値として受け取っているはずである．そうでない場合，制約を満たしていないと判断して，エラーメッセージを出力するようにすれば良い．

5.3 第 10〜12 週: セマンティックアクションの段階的な記述

以上に述べたような考え方で，各非終端記号に対して，どのようなコードを生成する必要があるかを考えて，そのコードを生成するためのセマンティックアクションの記述を行なっていけばコンパイラは完成である．しかしながら，モノによっては結構ややこしい．そこで，ある程度順序を踏んで作成していくのが良いと思われる．例えば以下のような順序が考えられる．

1. **代入を含まない算術式の評価と出力文のみ** すなわち，変数も代入も制御構造も大小比較も扱わない．これにより例えば printexp.k (p. 42) に対するアセンブラコードが出力できるはずである．
2. **代入演算子と変数 (配列は後回し)** ここで変数表の管理と参照が加わる．これにより assign.k (p. 42) に対するアセンブラコードの生成が行なえる．
3. **論理演算子，および while 文と if 文** ただし，まだ大小比較は扱わない．すなわち，whileif.k (p. 43) のコンパイルが行なえるようにする．
4. **大小比較** すなわち，comp.k (p. 43) をコンパイルできるようにする．これにより素数を順次求める prime.k (p. 45) のようなある程度実用性を持ったプログラムのコンパイルができるようになる．
5. **配列** これで簡易版バブルソートプログラム bsort2.k(p. 44) のコンパイルができるようになる．
6. **和関数，積関数** sumProduct.k (p. 45) のコンパイルができるようになる．
7. **++ 演算子，--演算子** primeInc.k (p. 45), arrayInc.k (p. 46), のコンパイルができるようになる．
8. **for 文** for.k (p. 46) のコンパイルができるようになる．
9. **配列への初期値代入** arrayInitAssign.k (p. 47) のコンパイルができるようになる．
10. **break 文** break.k (p. 47) のコンパイルができるようになる．
11. **総合確認** bsort.k (p. 48) のコンパイルができるようになり，当初の目標が達成できたことになる．
12. **代入演算子の左被演算子の確認** ここまで来れば一応完成と考えて良い．

上記 3. 4. では，2.3 節で学習した変換技法を参考にせよ．

問題 5.1（コンパイラの完成） 構文解析プログラムに順次セマンティックアクションを埋め込んで行き，Kc.java を完成せよ．プログラムができたら，`bsort.k` およびその他のソースプログラムのコンパイルと，生成されたコードの VSM による実行を行ない，正しく動作するかどうか確認せよ．

なお，`bsort.k` のコンパイル，実行さえできれば完全というわけではないので注意すること．最終的な動作試験は `bsort.k` より，もう少し意地の悪いプログラムで行なう予定なので，そのつもりで十分な動作試験と不具合の修正を行なっておくこと．

問題 5.2 代入演算子の左被演算子の確認が正しく行われているかを検査するために，制約を満たさないソースプログラムをいくつか作成し，作成したコンパイラがそれらの制約違反を正しく検出している事を確認せよ．

6 言語機能の拡張 (第 13 週以降)

問題 6.1 実習スケジュール通り，12 週目までで K23 言語のコンパイラを完成した者は，言語の機能拡張を考え，拡張 K23 言語に対するコンパイラを作成してもらいたい．拡張の内容としては例えば以下のようなものが考えられる．

- コメントの記述を可能とする．例えば「/*」で始まり，「*/」で終わる文字列はコメントとみなす．このためには字句解析プログラムで「/*」で始まり，「*/」で終わる文字列を読み飛ばすようにすればよい．
- さらに「//」から行末までをコメントみなして読み飛ばす．
- for 文を Java や C に準ずるものに拡張する．具体的には初期化式，繰り返し条件式，更新式を省略可能とする．(例: `for (;;)`) 繰り返し条件式が省略された場合は常に真であるとみなされ，break 文で脱出しない限り無限ループとなる，さらには初期化式，更新式の部分を「,」で区切った複数の式を記述できるようにするなどが考えられる．(例 : `for (i=0, j=100; i<j; ++i, --j)`)
- 2 次元配列の導入．
- 配列要素の後置 ++ の導入．
- if 文における else 節 の導入．
- do-while 文の導入．
- switch-case 文の導入．
- continue 文の導入．
- 定数の演算はコンパイル時に計算する．
- 変数宣言時に，`int n = i*j;` のように初期値として式を使えるようにする．
- 文字列操作の導入．例えば，`int s[] = ''hello'';` のように配列に文字列を初期値として代入できるようにする．この例の場合は `int s[] = { 'h', 'e', 'l' , 'l', 'o' }` と同じ意味である．また，`outputstr (s);` のように配列の中身を文字列として表示できるようにする．
- ブロック中に変数宣言できるようにする．
- for 文の初期式で変数宣言できるようにする．(例 : `for (int i=0; i<10; ++i)...`)
- 構文エラーを発見したとき，エラーメッセージを出力して終了するのではなく，通常のコンパイラのように適当にトークンの置き換え，挿入を行うことで構文解析を継続し，ソースプログラムの構文エラーをすべて検出するようにする．
- 宣言されたが使用されていない変数がある，break 文の直後など到達不能命令がある，といった文法上は認められるが不自然な記述がある場合にワーニングメッセージを出す．

上記のような拡張を実施する場合，Kc.java や LexicalAnalyzer.java だけでなく，教材として与えられている Symbol.java なども書き換える必要がある．拡張プログラムを提出する際は，それらのファイルも含めて提出すること．

付録 A　K23 言語サンプルプログラム

プログラム例 10　SLT.k (字句解析検査用プログラム)

```
1  12 = 10 + 8
2  13 == 5
3  !12
4  12 != 00
5  13==5
6  14!=10
```

プログラム例 11　printexp.k (数式出力プログラム)

```
1  main() {
2      outputint (- 3 + 4 * ( 5 + 6 / - - 2) % 10);
3  }
```

プログラム例 12　assign.k (代入例プログラム)

```
1  main () {
2      int i,j,k, l, product;
3      int c;
4
5      i = inputint;
6      j = inputint;
7      c = inputchar+1;
8      k = i+j;
9      l = product = i*j;
10     outputint (k);
11     outputint (l);
12     outputint (product);
13     outputchar (c);
14 }
```

プログラム例 13　whileif.k (制御構造例プログラム)

```
 0  main () {
 1      int i1, i2, i3, i, sum=0;
 2      i1= inputint;
 3      i2= inputint;
 4      i3= inputint;
 5      if (i1 && i2 && i3) {
 6          i = i1 * i2 * i3;
 7          while (i) {
 8              sum=sum+i;
 9              i=i-1;
10          }
11      }
12      outputint(sum);
13      if (i1 || i2 || i3) {
14        outputint(i1 + i2 + i3);
15      }
16  }
```

プログラム例 14　comp.k (比較例プログラム)

```
 1  main () {
 2      int i, sum;
 3      sum=0;
 4      i= inputint;
 5      if (i>0 && i<10) {
 6          while (i>0) {
 7              sum=sum+i;
 8              i=i-1;
 9          }
10      }
11      outputint (sum);
12  }
```

プログラム例 15 bsort2.k (簡易版バブルソートプログラム)

```
 1  main()  {
 2      int n=0, m=1, s, tmp, SIZE=20, data[20];
 3
 4      outputchar('?');
 5      s= (inputint * 1297 + 1) % 131 *2 -1;
 6
 7      while (n < SIZE)  {
 8          m= (m * 23 +0x002F) % s;
 9          data[n]=(m+n) % SIZE + 1;
10          n= n+1;
11      }
12
13      n= 0;
14      while (n < SIZE)  {
15          outputint(data[n]);
16          n=n+1;
17      }
18
19      outputchar(' ');
20      n= 0;
21      while (n < SIZE) {
22          m= SIZE-1;
23          while (n < m) {
24              if (data[m-1] > data[m]){
25                  tmp= data[m];
26                  data[m]= data[m-1];
27                  data[m-1]= tmp;
28              }
29              m=m-1;
30          }
31          n=n+1;
32      }
33
34      outputint(data[0]);
35      n= 1;
36      while (n < SIZE) {
37          outputint(data[n]);
38          n=n+1;
39      }
40  }
```

プログラム例 16　prime.k (素数選択プログラム)

```
 1  main() {
 2      int m,n, max;
 3      max = inputint;
 4      m=2;
 5      while (m<max) {
 6          n=2;
 7          while (!(m%n==0)) n = n + 1;
 8          if (m==n) outputint (m);
 9          m = m + 1;
10      }
11  }
```

プログラム例 17　sumProduct.k (和関数，積関数の例プログラム)

```
1 main() {
2     outputint ( *(2, 3, 5) );
3     outputint ( +(10, -20, 30, -40, 50) );
4     outputint ( + ( * (1), * (2, 2), * (3, 3, 3) ) );
5 }
```

プログラム例 18　primeInc.k (++ 演算子を含む素数選択プログラム)

```
 1  main() {
 2      int m,n, max;
 3      max = inputint;
 4      m=2;
 5      while (m<max) {
 6          n=2;
 7          while (!(m%n==0)) ++n;
 8          if (m==n) outputint (m);
 9          m++;
10      }
11  }
```

プログラム例 19 arrayInc.k (配列に対する ++ 演算を含むプログラム)

```
 1 main() {
 2    int i=7, a[7];
 3
 4    while (i)
 5       a[--i] = 10*i;
 6
 7    outputint (a[0]);
 8    while (i<6)
 9       outputint (a[++i]);
10
11    while (i>4)
12       --a[--i];
13
14    while (i>1)
15       ++a[--i];
16
17    a[i *= 0] += 5;
18
19    outputint (a[0]);
20    while (i<6)
21       outputint (a[++i]);
22 }
```

プログラム例 20 for.k (for 文の使用例)

```
 1 main() {
 2    int i, j, n;
 3
 4    for (i=1; i<11; i++) {
 5       n=0;
 6       for (j=1; j<i+1; ++j) {
 7          n += j;
 8       }
 9       outputint (n);
10    }
11 }
```

プログラム例 21　arrayInitAssign.k (配列への初期値代入の使用例)

```
 1  main() {
 2      int number[] = {-1, 0,1};
 3      int message[] = {'a', 'r', 'r', 'a', 'y'};
 4      int i=-1;
 5
 6      while (i< 3) {
 7          outputint(number[++i]);
 8      }
 9
10      i=-1;
11      while (i< 5) {
12          outputchar(message[++i]);
13      }
14  }
```

プログラム例 22　break.k (break 文の使用例)

```
 1 main () {
 2    int i=10;
 3
 4    while ( i > 0 ) {
 5       --i;
 6       if (i == 3) break;
 7       while (i == 5) break;
 8       outputint (i);
 9    }
10 }
```

プログラム例 23 bsort.k (バブルソートプログラム)

```
 1 main()  {
 2     int i, n=0, m=1, s, tmp, is_sorted=1, SIZE=20, data[20], product;
 3     int message[]= {'s','o','r','t'};
 4
 5     outputchar('?');
 6     product = *(1,inputint,m);
 7     s= (product * 1297 + 1) % 131 *2 -1;
 8
 9     for (i *= 0; i < SIZE; ++i)  {
10         m= (m * 23 +0x0002F) % s;
11         data[i]=+(m,i) % SIZE ;
12         ++data[i];
13     }
14
15     n= m= m* 0;
16    while (n < SIZE-1 || n==(SIZE-1)) {
17         outputint(data[n]);
18         ++n;
19     }
20
21     outputchar(' ');
22     i=0;
23     while (1) {
24         if (data[i] > data[i+1]) {
25             is_sorted = 0;
26             break;
27         }
28         ++i;
29         if (!(i - SIZE+1)) break;
30     }
31
32     if (is_sorted) {
33         outputchar('o');
34         outputchar('k');
35     }


// 次ページに続く
```

```
36     if (!is_sorted) {
37         i = -1;
38         while (i < 0x03) {
39             outputchar(message[++i]);
40         }
41         outputchar(' ');
42         n= 0;
43         while (!(SIZE < n + 1) ) {
44             m= SIZE-1;
45             while (n < m) {
46                 if (data[m-1]>data[m]){
47                     tmp= data[m];
48                     data[m]= data[m-1];
49                     data[m-1]= tmp;
50                 }
51                 m--;
52             }
53             n+=1;
54         }
55         for (i=0;  i<SIZE; i = + (i,-1,2))
56             outputint(data[i]);
57
58     }
59 }
```

付録 B　パッケージ kc 内 Java 言語プログラム

プログラム例 24　Symbol.java

```
package kc;
enum Symbol {
    NULL,
    MAIN,       /* main */
    IF,         /* if */
    WHILE,      /* while */
    FOR,        /* for */
    INPUTINT,   /* inputint */
    INPUTCHAR,  /* inputchar */
    OUTPUTINT,  /* outputint */
    OUTPUTCHAR, /* outputchar */
    BREAK,      /* break */
    INT,        /* int */
    EQUAL,      /* == */
    NOTEQ,      /* != */
    LESS,       /* < */
    GREAT,      /* > */
    AND,        /* && */
    OR,         /* || */
    NOT,        /* ! */
    ADD,        /* + */
    SUB,        /* - */
    MUL,        /* * */
    DIV,        /* / */
    MOD,        /* % */
    ASSIGN,     /* = */
    ASSIGNADD,  /* += */
    ASSIGNSUB,  /* -= */
    ASSIGNMUL,  /* *= */
    ASSIGNDIV,  /* /= */
    INC,        /* ++ */
    DEC,        /* -- */
    SEMICOLON,  /* ; */
    LPAREN,     /* ( */
    RPAREN,     /* ) */
    LBRACE,     /* { */
    RBRACE,     /* } */
    LBRACKET,   /* [ */
    RBRACKET,   /* ] */
    COMMA,      /* , */
    INTEGER,    /* 整数 */
    CHARACTER,  /* 文字 */
    NAME,       /* 変数名 */
    EOF         /* end of file */
}
```

プログラム例 25　Operator.java

```
package kc;
enum Operator {
  NOP,
  ASSGN,
  ADD,
  SUB,
  MUL,
  DIV,
  MOD,
  CSIGN,
  AND,
  OR,
  NOT,
  COMP,
  COPY,
  PUSH,
  PUSHI,
  REMOVE,
  POP,
  INC,
  DEC,
  JUMP,
  BLT,
  BLE,
  BEQ,
  BNE,
  BGE,
  BGT,
  HALT,
  INPUT,
  INPUTC,
  OUTPUT,
  OUTPUTC,
  OUTPUTLN,
  LOAD
}
```

プログラム例 26　Type.java

```
package kc;
enum Type {
    INT,
    ARRAYOFINT,
    NULL
}
```

付録 C　VSM オペレータセット

BINOP: `#define BINOP(OP) {Stack[SP-1] = Stack[SP-1] OP Stack[SP]; SP--;}`
ASSGN:

```
addr = Stack[--SP];
Dseg[addr] = Stack[SP] = Stack[SP+1];
```

ADD: `BINOP(+);`
SUB: `BINOP(-);`
MUL: `BINOP(*);`
DIV:

```
if (Stack[SP] == 0)
  {
     printf("Zero divider detected\n");
     return -2;
  }
BINOP(/);
```

MOD:

```
if (Stack[SP] == 0)
   {
     printf("Zero divider detected\n");
     return -2;
   }
BINOP(%);
```

CSIGN: `Stack[SP] = -Stack[SP];`
AND: `BINOP(&&);`
OR: `BINOP(||);`
NOT: `Stack[SP] = !Stack[SP];`
COPY: `++SP; Stack[SP] = Stack[SP-1];`
PUSH: `Stack[++SP] = Dseg[addr];`
PUSHI: `Stack[++SP] = addr;`
REMOVE: `--SP;`
POP: `Dseg[addr] = Stack[SP--];`
INC: `Stack[SP] = ++Stack[SP];`
DEC: `Stack[SP] = --Stack[SP];`

COMP:

```
Stack[SP-1] = Stack[SP-1] > Stack[SP] ? 1 :
Stack[SP-1] < Stack[SP] ? -1 : 0;
SP--;
```

BLT: `if (Stack[SP--] < 0) Pctr = addr;`
BLE: `if (Stack[SP--] <= 0) Pctr = addr;`
BEQ: `if (Stack[SP--] == 0) Pctr = addr;`
BNE: `if (Stack[SP--] != 0) Pctr = addr;`
BGE: `if (Stack[SP--] >= 0) Pctr = addr;`
BGT: `if (Stack[SP--] > 0) Pctr = addr;`
JUMP: `Pctr = addr;`
HALT: `return 0;`
INPUT: `scanf("%d%*c", &Stack[++SP]);`
INPUTC: `scanf("%c%*c", &Stack[++SP]);`
OUTPUT: `printf("%15d", Stack[SP--]);`
OUTPUTC: `printf("%c", Stack[SP--]);`
OUTPUTLN: `printf("\n");`
LOAD: `Stack[SP]=Dseg[Stack[SP]];`

付録 D　VSM

D.1　VSM の構造と動作

VSM とは，一般の計算機内部で行われているスタック (stack) を用いた計算方法を理解するために用意された抽象的なスタックマシンである．VSM の主要な構成要素は，Stack, Iseg, Dseg, Pctr の 4 つであり，この 4 つの要素の役割をここでしっかりと把握しておくこと．

Stack　後入れ先出し (LIFO: Last In First Out) 型のメモリ領域である．ここに計算に必要なデータを，あとで述べる Iseg 内の命令通りに push(書込み) したり，pop(取出し) することで計算が進む．図 5 の左図は，プログラムの実行中に「スタックトップ」の位置までデータがスタックに積まれている状態を示している．プログラム開始時は，スタックは空 (くう)，つまり何もデータが積まれていない状態である．なお，VSM ではスタックは下方に成長する．

Iseg　Instruction Segment の略．アセンブラプログラムが格納される．基本的には Iseg の先頭から順に命令 (オペレータとオペランドの組) が実行される．図 5 の中央図参照.

Pctr　Program Counter の略．次に実行する命令が，Iseg のどの番地に格納されているかを示すカウンタ．

Dseg　Data Segment の略．プログラムによって処理されるデータが格納される．図 5 の右図に示すように，0 から順に番地が付けられている．

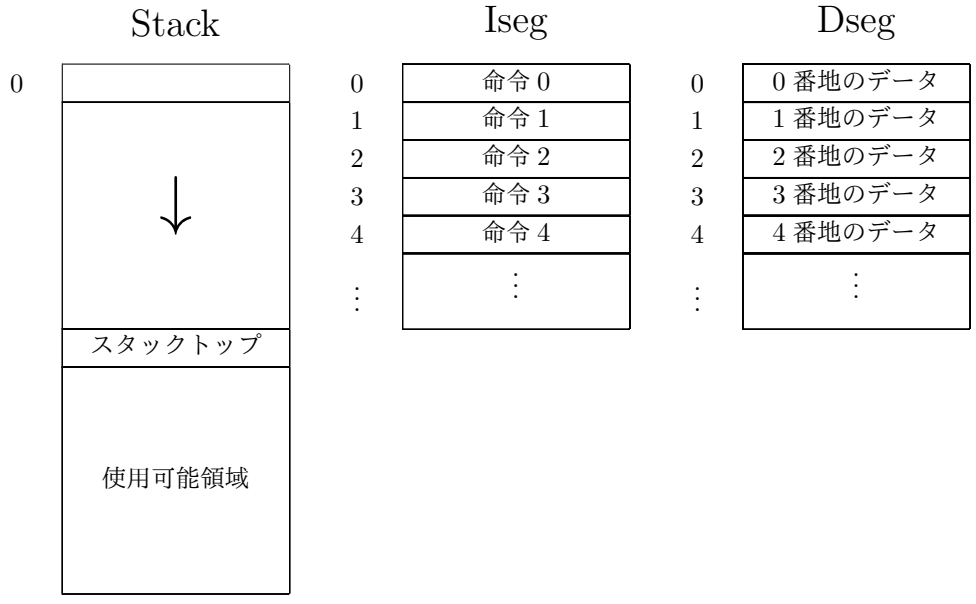


図 5　Stack，Iseg，Dseg

例 付録 D.1（VSM の動作例） VSM の動作を，具体的なプログラムを用いて例示する．左下の VSM アセンブラプログラムは，右下の K23 言語プログラムに対応している（実質的には，`i = 3;` から `outputint(j);` までの 3 行）．

```
PUSHI   0
PUSHI   3
ASSGN
REMOVE
PUSHI   1
PUSH    0
PUSHI   4
ADD
ASSGN
REMOVE
PUSH    1
OUTPUT
OUTPUTLN
HALT
```

```
main () {
    int i,j;
    i = 3;
    j = i + 4;
    outputint (j);
}
```

0	PUSHI 0
1	PUSHI 3
2	ASSGN
3	REMOVE
4	PUSHI 1
5	PUSH 0
6	PUSHI 4
7	ADD
8	ASSGN
9	REMOVE
10	PUSH 1
11	OUTPUT
12	OUTPUTLN
13	HALT

図 6　命令格納済の Iseg

このアセンブラプログラムが，図 6 に示すように Iseg に格納される．また，変数 i が Dseg の 0 番地に，変数 j が Dseg の 1 番地に，それぞれあらかじめ対応づけられているものとする．変数と Dseg の対応については，次週学習する．なお，各命令の正確な意味については，2.3.1 節と付録の付録 C 章を参照されたい．

- 図 7 の最上段左の図は，プログラム開始直前の VSM の各要素の状態を示している．次に実行する命令は最初の命令なので Pctr の値は 0, スタックは空，そして変数 i と j のために，Dseg の 0 番地と 1 番地が各々用意されている様子を示している．以下，Pctr の番号で図を参照する．
- Pctr: 1 は，最初の命令，すなわち Iseg の 0 番地の命令 PUSHI 0 を実行し，スタックに 0 を積んだ後の状態を示している．次の命令は Iseg の 1 番地 の PUSHI 3 なので，Pctr の値は 1 になっている．Pctr: 2 では，同様に PUSHI 3 を実行し，スタックに 3 を積んだ後の状態を表している．
- Pctr: 3 で，命令 ASSGN を実行し，スタックトップの値 (つまり整数 3) を，スタックトップの一つ前の値が示す Dseg の番地，つまり 0 番地に格納した後の状態を示す．この命令は同時に，スタックトップの値をスタックトップの一つ前の領域に格納し直し，スタックトップの値を取り除く作用も持つ．
- Pctr: 4 で，命令 REMOVE を実行し，スタックトップの値 (つまり 3) を，スタックから取り除いた後の状態を示す．
- Pctr: 5〜Pctr: 7 で，命令 PUSHI 1, PUSH 0, PUSHI 4 をそれぞれ実行した後の状態を示す．なお，PUSH 0 はスタックに 0 ではなく，Dseg の 0 番地の値，すなわち 3 をスタックに積む命令であることに注意されたい．
- Pctr: 8 で，命令 ADD，すなわちスタックトップの一つ前の値にスタックトップの値を足して，スタックトップの値を取り除いた後の状態を示す．
- Pctr: 9 で，命令 ASSGN を実行し，スタックトップの値 (つまり整数 7) を，スタックトップの一つ前の値が示す Dseg の番地，つまり 1 番地に格納した後の状態を示す．
- Pctr: 10 では， 命令 REMOVE でスタックトップの値を取り除いた状態を示している．
- Pctr: 11 では，命令 PUSH 1 で Dseg 1 番地の値をスタックに積んだ後の状態を示している．
- Pctr: 12 では，命令 OUTPUT でスタックトップの値，すなわち 7 を標準出力 (画面) に出力し，スタックトップの値を取り除いた後の状態を示している．
- 最後に Iseg の 12 番地の命令 OUTPUTLN で改行コードを出力し，13 番地の命令 HALT でプログラムを終了する．

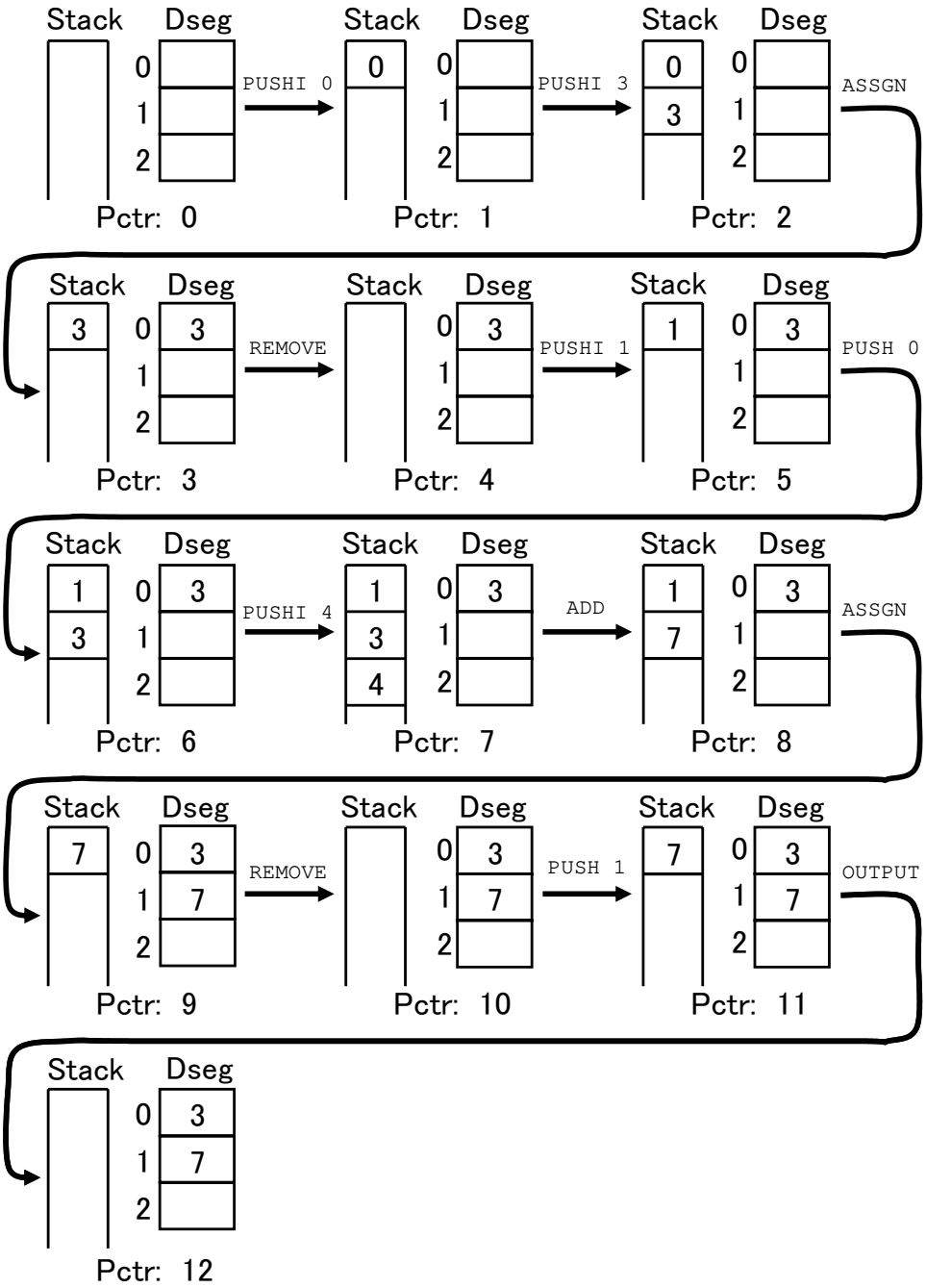

Stack
Dseg
PUSHI 0
PUSHI 3
ASSGN
Pctr: 0
Pctr: 1
Pctr: 2
REMOVE
PUSHI 1
PUSH 0
Pctr: 3
Pctr: 4
Pctr: 5
PUSHI 4
ADD
ASSGN
Pctr: 6
Pctr: 7
Pctr: 8
REMOVE
PUSH 1
OUTPUT
Pctr: 9
Pctr: 10
Pctr: 11
Pctr: 12

図 7　VSM の動作

問題 D.1

1. 例付録 D.1 を参考に，左下のアセンブラプログラムが VSM でどのように処理されるか，Stack, Pctr, Dseg の各状態を図示し，説明を各図の下に記入せよ．最初の 4 つの PUSHI 命令は一つの図にまとめて良いが，それ以外の命令に対する動作は，別々に図示すること．
   (左のアセンブラプログラムは，右の K23 言語プログラムに対応している．変数 i に対 して，Dseg の 0 番地が割り当てられている)

```
PUSHI   0
PUSHI   3
PUSHI   4
PUSHI   2
MUL
ADD
ASSGN
REMOVE
PUSH    0
OUTPUT
OUTPUTLN
HALT
```

```
main () {
   int i;
   i = 3+4*2;
   outputint (i);
}
```

2. このアセンブラプログラムを，実際に VSM で動作させ正しい出力が得られることを確認し，実行結果を実習ノートに書け．実行方法は以下の通り:
   (a) このアセンブラプログラムを emacs を用いて入力し，プログラムファイルを作成する．ファイル名は ex1.asm とすること．
   (b) 以下のコマンドで，このファイル名を引数として VSM をターミナル上で実行する．

```
$ ./vsm ex1.asm
```

D.2 逆ポーランド記法とアセンブラ

通常我々が記述する数式は，例えば

$$1 + 2 \tag{1}$$

のように，二項演算子 (この場合は「+」) が，被演算子 (この場合「1」と 「2」) の中間に置かれている．このような表現方法を中置記法と呼ぶ．我々人間にとって中置記法はなじみ深い数式の表現方法であるが，スタックを用いて計算を行う場合，逆ポーランド記法 (後置記法) とよばれる表現方法で数式を表現すると都合が良い．(1) 式を逆ポーランド記法で表現すると (2) 式のようになる．

$$1\ 2\ + \tag{2}$$

中置記法で書かれた数式を VSM アセンブラへ変換，実行するには，以下の手順をとる．

1. 中置記法の式を木構造で表現する．

2. 木構造で表現された数式を，逆ポーランド記法へ変換する．
3. 逆ポーランド記法を，VSM アセンブラへ変換する．
4. 変換されたアセンブラプログラムを，vsm で実行する．

本節では，実際に中置記法で表現された式を後置記法に変換し，VSM アセンブラへ変換，実行する方法を，上記の順番で学習する．

例 付録 D.2（木構造への変換）

ここでは (3) 式を例にとって，中置記法による式の構造を木で表現してみよう．

$$1 + 2 * (3 + 4) - 5 \tag{3}$$

(3) 式の中で一番最初に計算されるのは $3 + 4$ なので，まずこの木を作る．$3 + 4$ は，3 と 4 が演算子 + でつながっているので，木は図 8 のようになる．

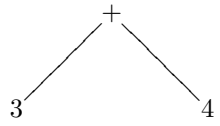


図 8　木の作成 (+ 分岐)

次に計算されるのは，2 の右の $*$ である．この部分は，2 と $(3 + 4)$ が $*$ でつながった式だと見なせるので，木は図 9 のようになる．

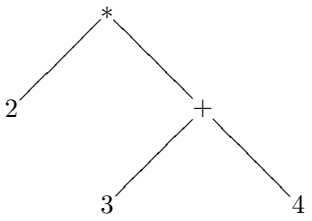


図 9　木の作成 ($*$ 分岐)

次に計算されるのは，2 の左の + である．この部分は，1 と $2 * (3 + 4)$ が + でつながった式だと見なせるので，木は図 10 のようになる．

最後に計算されるのは，5 の左の − である．この部分は，$1 + 2 * (3 + 4)$ と 5 が − でつながった式だと見なせるので，木は図 11 のようになる．

木ができあがったら，それを後順走査で辿る．直観的に木の後順走査出力 (以降単に「後順走査」と書く) とは，「左部分木の後順走査」「右部分木の後順走査」「根のラベル」を順番に並べたものである．例えば図 12 のような木があった場合，$3, 4, +$ が後順走査である．この要領で図 11 の後順走査を求める様子を，例付録 D.3 に示す．

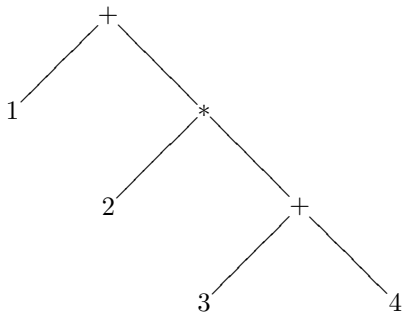


図 10　木の作成 (+ 分岐)

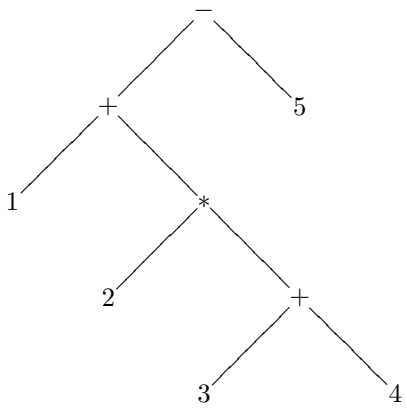


図 11　木の作成 (− 分岐を作って完成)

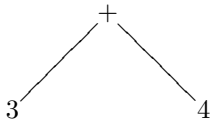


図 12　3，4，+ 後順走査

例 付録 D.3（木の後順走査）

図 11 を辿る順番を図 13 に図示する．

(1)　この木全体の後順走査は，「左部分木 (以降これを L1 と呼ぶ) の後順走査」「5」「−」の並びとなる．

(2)　L1 の左部分木は 1 のみであることより， L1 の後順走査は 「1」「L1 の右部分木 (以降 R1) の後順走査」「+ (L1 の根)」の並びとなる．

(3)　R1 の左部分木は 2 のみなので，R1 の後順走査は「2」「R1 の右部分木 (R2) の後順走査」「 ∗ (R1 の根)」の並びとなる．

(4) R2 の左部分木は 3 のみ，R2 の右部分木は 4 のみなので，R2 の後順走査は「3」「4」「+ (R2 の根)」の並びとなる．

(5) (4) の結果を (3) に適用することにより，R1 の後順走査は 2 3 4 ＋ ∗ となる．

(6) (5) の結果を (2) に適用することにより，L1 の後順走査は 1 2 3 4 ＋ ∗ ＋ となる．

(7) (6) の結果を (1) に適用することにより，木全体の後順走査は，式 (4) のようになる．

$$1\ 2\ 3\ 4\ +\ *\ +\ 5\ - \tag{4}$$

実際には木ができあがったら以下の単純な操作にしたがって，後順走査を求めることができる．

- 左優先の深さ優先探索をして訪問した頂点を順に記録していく．
- ただし内部頂点は最後に訪れたときのみ記録する．

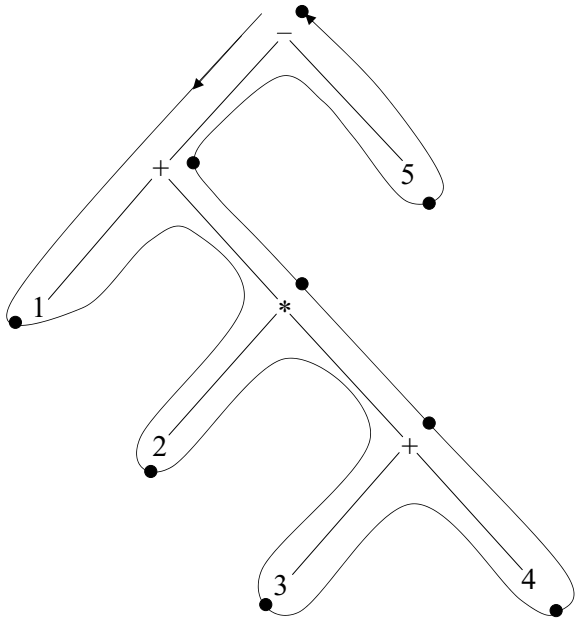


図 13 木の後順走査

式を逆ポーランド記法に変換できたら，VSM アセンブラへの変換は非常に簡単である．例 付録 D.4 で変換方法を例示する．

例 付録 D.4（アセンブラへの変換） 例 付録 D.3 で得られた逆ポーランド記法による式 (4) とそれに対応する VSM アセンブラプログラムを以下に記す．

1	PUSHI　1
2	PUSHI　2
3	PUSHI　3
4	PUSHI　4
+	ADD
*	MUL
+	ADD
5	PUSHI　5
-	SUB
	OUTPUT
	OUTPUTLN
	HALT

アセンブラプログラムの最後の 3 つの命令は，計算結果の出力と，プログラム停止のために付け加えたものである．それ以外の命令は，逆ポーランド記法で表現された式と一対一に対応しているのがわかるだろう．

問題 D.2

1. (5) 式で与えられる数式を，VSM アセンブラプログラムに変換し，さらに OUTPUT 命令，OUTPUTLN 命令と HALT 命令を付け加え，実際に VSM で動作させよ．実習ノートには，図 13 と同様の木構造の図を描き，逆ポーランド記法に変換した式，VSM アセンブラプログラム，VSM の実行結果を書くこと．なお，式中の演算子は Java, C に準じるものとして扱う．よって，演算子の優先順位は以下の規則に従う．
   - 括弧がある場合は括弧の中が最優先
   - 掛け算割り算は足し算引き算よりも優先
   - 掛け算割り算同士では左にあるものが優先，足し算引き算同士では左にあるものが優先

$$3 + 4 * (5 + 6)/3 \tag{5}$$

2. 1 番で求めたアセンブラプログラムが VSM で処理される過程を，問題 D.1 の 1 番と同じ要領で図示し，説明を各図の下に記入せよ．(Dseg は省略しても良い)

付録 E　再帰降下構文解析のためのアルゴリズム

E.1　FIRST 集合を求めるアルゴリズム

α を文法記号列とするとき，FIRST(α) は α から導出される先頭が終端記号である文法記号列の先頭の終端記号からなる集合である．ただし，α から ε が導出される場合には ε も FIRST(α) に含まれる．

FIRST 集合を求めるアルゴリズムを次に示す．

アルゴリズム　付録 E.1　FIRST 集合を求める

入力　EBNF G, 文法記号列 α

出力　FIRST(α)

手順　FIRST(α) が空集合から始め，FIRST(α) に追加するものがなくなるまで以下を繰り返す

(1)　α が空系列 ($\alpha = \varepsilon$) ならば，FIRST(α)=$\{\varepsilon\}$

(2)　α が終端記号 1 文字 ($\alpha = a$ とする) ならば，FIRST(α)=$\{a\}$

(3)　α が非終端記号 1 文字 ($\alpha = X$ とする) ならば，X を左辺に持つすべての生成規則 $X ::= \beta$ に対して FIRST(β) を求め，それらの和集合を FIRST(α) とする

(4)　α が長さ 2 以上の系列 ($\alpha = X\beta$ とする) ならば，FIRST(X) を求め，FIRST(α) に追加する．ただし，FIRST(X) が ε を含む場合は，ε は FIRST(α) に追加せず，FIRST(β) を求め，FIRST(α) に追加する

(5)　α が「$\beta \mid \gamma$」ならば，FIRST(β), FIRST(γ) を求め，それらを FIRST(α) に追加する

(6)　α が「$\{\beta\}$」または「$[\beta]$」ならば，FIRST(β) を求め，それらと ε を FIRST(α) に追加する

(7)　α が「(β)」ならば，FIRST(β) を求め, それらを FIRST(α) に追加する

E.2　再帰降下型構文解析法

任意の文法記号列 α について，FIRST(α) を求めることができれば，それを基に各非終端記号 A について A から導出できる終端記号列を解析する手続き `parseA` を構成することができる．

アルゴリズム　付録 E.2　再帰降下構文解析

入力　EBNF G

ただし，各非終端記号 A について A を左辺にもつ生成規則は一つしかないものとする．もし，複数ある場合には「|」演算子を用いて一つにまとめればよい．

出力　構文解析プログラム

手順　各非終端記号 A に対する生成規則を「$A ::= \gamma_A$」とする．各 A に対して以下の規則を用いて再帰的に A に対する構文解析手続き `parseA` を作る

(1)　$\gamma_A = \varepsilon$ ならば，何もコードを生成しない

(2)　$\gamma_A = a$(終端記号 1 文字) ならば，次のコードを生成する

```
if (token.checkSymbol(Symbol.a) token=lexer.nextToken();
                    else syntaxError();
```

(3) $\gamma_A = X$(非終端記号 1 文字) ならば，次のコードを生成する

```
parseX();
```

(4) γ_A = g_1 | g_2 | ... | g_n ならば，次のコードを生成する

```
switch(token.getSymbol()) {
case FIRST(g_1): "g_1"に対するコード;
     break;
case FIRST(g_2): "g_2"に対するコード;
     break;
     ...
case FIRST(g_n): "g_n"に対するコード;
     break;
}
```

ただし，FIRST(g_k) が ε を含んでいる場合 (そのような k は高々一つしかないはずである)，

```
case FIRST(g_k):
```

の行を

```
default: "g_k"に対するコード;
```

に置き換え，そのような k がなければ，

```
default: syntaxError();
     break;
```

を付け加える．

(5) γ_A =g_1 g_2 ··· g_n ならば，次のコードを生成する

```
{
    "g_1"に対するコード;
    "g_2"に対するコード;
     ...
    "g_n"に対するコード;
}
```

(6) γ_A ={g} ならば，次のコードを生成する

```
while (token.getSymbol() が FIRST(g) に属する) {
    "g"に対するコード;
}
```

(7) γ_A =[g] ならば，次のコードを生成する

```
if (token.getSymbol() が FIRST(g) に属する) {
    "g"に対するコード;
}
```

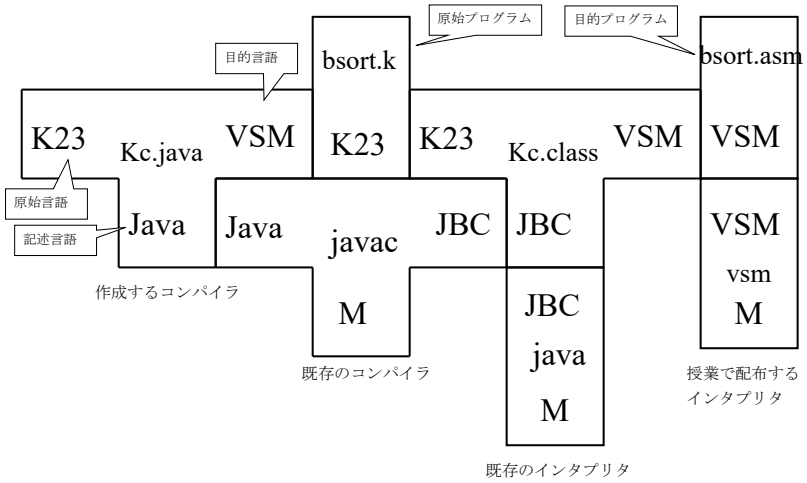
(8) γ_A =(g) ならば，次のコードを生成する

```
    "g"に対するコード;
```

付録 F　作成するコンパイラの構造

作成するコンパイラの T 図式

- 原始言語 : K23 言語(C 風言語)
- 目的言語 : VSM(Virtual Stack Machine)アセンブラ言語
- 記述言語 : Java



作成するコンパイラの構造