卒業研究報告書

Java による PRAM コンパイラの作成

指 導 教 員 石水 隆 助手

03-1-47-162

神谷　道利

近畿大学理工学部情報学科

平成 19 年 2 月 5 日提出

概要

様々な分野で計算量の大きな問題を短時間で解く事が求められている。計算速度を向上させる方法としてはプロセッサの性能の向上、アルゴリズムの最適化等が存在するがそれぞれには限界が存在するため一定以上の向上が見受けられなくなってきている。そこで現在ではプロセッサの並列処理による高速化が注目されている。

並列処理とは、ある問題を解く際、その問題をより小さい複数の部分問題に分割し、その部分問題を複数のプロセッサで解く処理である。並列処理を行うことにより、単一のプロセッサによる逐次処理よりも高速に問題を解くことができ、またより複雑な問題を解く事ができる。

しかし、並列計算機は非常に高価であるため、並列アルゴリズムの設計およびその計算量の解析を実際の並列計算機を用いて実験的に行うことは困難である。そこで、本研究では並列アルゴリズムの実験的評価を容易化するツールである PRAM シミュレータの一部として PRAM コンパイラを作成する。

目次

1　序論 1

1.1　本研究の背景 1
1.2　本研究の目的 3
1.3　本報告書の構成 3

2　研究内容 4

2.1　PRAM 用並列言語 4
2.2　並列アセンブラ 4
2.3　PRAM コンパイラ 5

3　結果 6

4　考察・今後の課題 7

5　謝辞 8

付録 1　K05 言語の文法 10

付録 2　K05 言語の文法 11

付録 3　VSM アセンブラの文法 12

付録 4　PRAM コンパイラプログラム 14

1　序論

1.1　本研究の背景

1.1.1 並列処理

様々な分野で高速な計算機を用いて計算量の大きな問題を短時間でとくことが求められている。計算速度を上昇させるための方法の１つとしてプロセッサの性能の向上が考えられる。だがこれには 3.0×10^7m/s という光速の限界が存在し現在の１万倍ほどしか速くすることができない。またアルゴリズムの改良によって劇的な高速化が可能であるが、これには計算量の下界が存在する。ソーティングを例にとった場合、クイックソートによる計算量の $O(n \log n)$ が下界とされ、これ以上の改良は不可能とされている。だが、並列処理を用いた場合そのような限界は本質的に存在しない。

並列処理(Parallel Processing)とは、ある問題を解く際、その問題をより小さい複数の部分問題に分割し、その部分問題を複数のプロセッサで解く処理である。並列処理を行うことにより、単一のプロセッサによる逐次処理よりも高速に問題を解くことができ、またより複雑な問題を解く事ができる。

しかし、並列処理を行うためにはプロセッサ間での通信や同期、メモリへのアクセスなど並列独特の動きが必要となる。すなわち従来の逐次処理でのアルゴリズムを適応させることができないため、並列処理専用の並列アルゴリズム(Parallel Algorithm)が必要となる。

1.1.2 並列計算機

並列計算機(Parallel Computer)とは複数のプロセッサを持ち、並列処理を行える計算機である。並列計算機は大きく２つに分類される。図 1 のように全てのプロセッサが共通のメモリを使用して読み書きを行い、他のプロセッサ間で通信を行う共有メモリ型並列計算機(Shared Memory Parallel Computer)と、図 2 のように各プロセッサがそれぞれメモリを持ちネットワークを通じて通信を行う分散メモリ型並列計算機(Distributed Memory Parallel Computer)である。

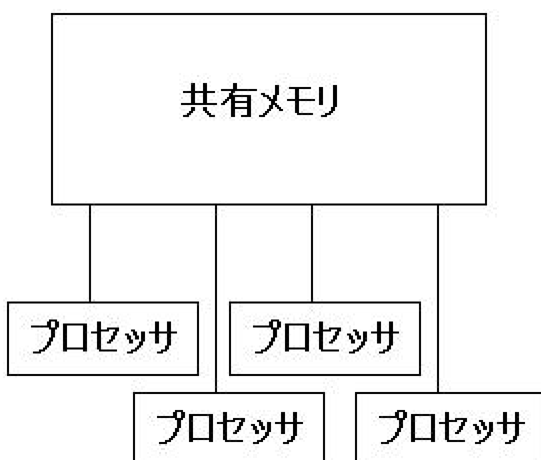

図 1 共有メモリ型並列計算機

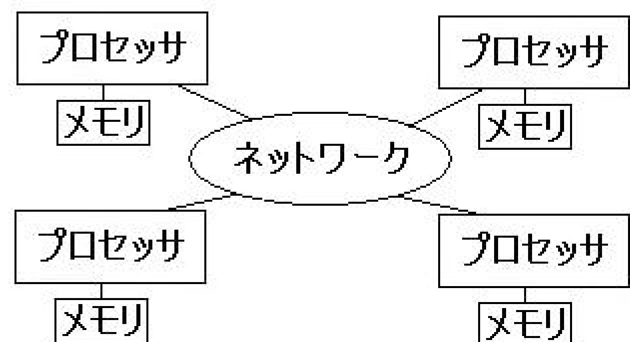

図 2 分散メモリ型並列計算機

1.1.3 並列計算モデル

並列アルゴリズムの設計、解析は、並列計算機を抽象化した並列計算モデル(Parallel Computing Model)上で行われる。代表的な並列計算モデルとして PRAM(Parallel Random Access Machine)、Mesh、HyperCube、BSP(Bulk Synchronous Parallel)などがある。

1.1.4 PRAM

並列アルゴリズムの設計およびその計算量の評価は多くの場合 PRAM(Parallel Random Access Machine)[2]上で行われる。

PRAM は共有メモリ型の並列計算モデルである。PRAM は演算命令、メモリアクセス命令、出力命令、全てのがその移動に関係なく 1 単位時間で行われる、1 命令ごとに同期が取られる、通信のコストが一切発生しない、等の仮定が設けられた理想的なモデルであるため、PRAM 上ではアルゴリズムの設計および評価を比較的容易に行う事ができる。しかし実際の大規模なプロセッサでのメモリ共有や、通信、プロセッサ間の同期の高速化は非常に困難であるため、PRAM アルゴリズムの実験的な評価を行うために PRAM シミュレータが必要となる

1.1.5 PRAM シミュレータ

PRAM アルゴリズムの実行をシミュレートする PRAM シミュレータは以下の４要素から成る①PRAM 用並列言語②並列アセンブラ③PRAM コンパイラ④PVSM（Parallel Virtual Stack Machine）。以下にそれぞれの説明を記す。

① 並列命令に対応した高級言語。
② 並列命令に対応したアセンブリコード。
③ ①の PRAM 用並列言語を②の並列アセンブリコードに変換するコンパイラ。
④ 並列アセンブラを実行するインタプリタ。

図 3 に PRAM シミュレータの実行の流れを示す。

ユーザは PRAM 用並列言語を用いて PRAM 用並列言語アルゴリズムを記述する。次に PRAM 用並列言語プログラムを PRAM コンパイラを用いて並列アセンブラプログラムに変換し PVSM を用いて並列アセンブラプログラムを実行することにより PRAM アルゴリズムの実行をシミュレートできる。

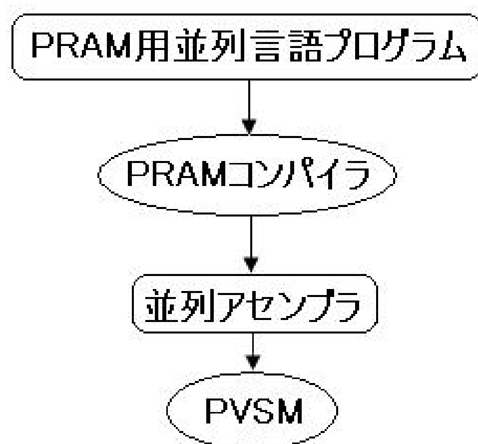

図 3 PRAM シミュレータによる実行の流れ

1.2　本研究の目的

本研究ではJAVA言語を用いてPRAMシミュレータの一部であるPRAMコンパイラの設計を行う。本研究で作成したPRAMコンパイラはPRAM 用並列言語に拡張したK05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（Virtual Stack Machine）のアセンブラコードに翻訳することができる。並列アセンブラプログラムをPVSMで実行することにより、PRAM用並列言語プログラムをPRAM上で動作させた場合の出力および実行時間を測定でき、PRAMアルゴリズムの計算量を実験的に評価することができる。

1.3　本報告書の構成

本報告書の構成を以下に述べる。2章では、本研究で作成したPRAMコンパイラについて述べる。3章では、PRAMコンパイラの検証を行う。またいくつかのプログラムを実行することにより作成したコンパイラの正当性を検証する。4 章では、3 章の結果を得た上で今後の課題を示す。

2　研究内容

2.1　PRAM 用並列言語

本研究で使用する PRAM 用並列言語は、JAVA 風の手続き言語である。この言語は付録 1 に示す K05 言語を並列処理に対応するように拡張した拡張 K05 言語である。拡張 K05 言語は K05 言語に並列処理を行う parallel 文および特殊記号「$p」を追加したものである。parallel 文の書式を以下に示す。

parallel（式１,式２）文

式１、式２は int 型の評価値を持つ式であり、文は parallel を含まない任意の文である。parallel 文はプロセッサ番号式１から式２までのプロセッサを用いて後ろに続く文を並列に実行する。また文中に特殊記号「$p」を記述すると「$p」は実行中のプロセッサ番号の値を持つ変数として処理される。

拡張 K05 言語プログラムは、０番目のプロセッサのみが命令を実行する逐次状態と parallel 文により指定された複数のプロセッサが命令を実行する並列状態を持つ。プログラム開始時は逐次状態で開始し、parallel 命令によって並列状態に移行し、文が終了すればまた逐次状態へ移行して実行する。付録 2 に拡張 K05 言語の文法を示す。

2.2　並列アセンブラ

本研究で使用する並列アセンブラは付録３に示す VSM アセンブラを並列処理に対応させるように拡張した拡張 VSM アセンブラである。拡張 VSM アセンブラは VSM アセンブラの命令セットに「PARA」「SYNC」「PUSHP」を加えたものである。各命令の働きを以下に示す。

PARA・・・・並列状態開始命令。逐次状態から並列状態へと移行する。
SYNC・・・・同期命令。並列状態の時プロセッサ間で同期を取り逐次状態に移行する。
PUSHP・・・プロセッサ番号挿入命令。スタックに命令を実行しているプロセッサのプロセッサ番号を入れる。

2.3 PRAM コンパイラ

コンパイラとは高水準言語(high level programming language)で書かれたプログラムを計算機が実現可能な形式へ変換する変換機である。本研究では高水準言語である拡張 K05 言語を低級言語である拡張 VSM アセンブラに変換する。以下に本研究で作成した PRAM コンパイラの構成を示す。

① 字句解析部
原始プログラムを読み、空白や記号等で区切り単語を生成する。

② 構文解析部
字句解析で得た単語の文法的な関係をまとめ、分類する。

③ 制約検索部
プログラムから型情報を抽出し、使用時に矛盾がないかを調べる。

④ コード生成部
上記の情報を元にアセンブラコードを生成する。

付録 4 に本研究で作成した PRAM コンパイラプログラムを示す。以下に本研究で作成した PRAM コンパイラの使用方法を示す。
foo.k を PRAM 用並列アルゴリズム言語によって記述された PRAM 用アルゴリズムとする。以下のコマンドを実行すると、foo.k が並列アセンブラに変換され、outputfile に出力される。Outputfile を省略した場合は OpCode.asm に出力される。

```
＞　java Pram foo.k [outputfile]
```

PRAM コンパイラにより生成された並列アセンブラは PVSM インタプリタにより実行される。
以下に PVSM インタプリタの使用方法を示す。
foo.asm を拡張 VSM アセンブラとする。以下のコマンドにより foo.asm が PVSM により実行される

```
＞　java Pvsm　[-c] [foo.asm]
```

ファイル名 foo.asm を省略したときは OpCode.asm が実行される。
また、実行時にオプション-c を指定することにより、拡張 K05 言語で書かれた PRAM プログラムを PRAM 上で実行したときの PRAM で実行したときの実行時間を計測できる。

3　結果

本研究は､作成したコンパイラの正当性を検証するため､簡単な PRAM プログラムを作成し、それが正しくコンパイルされているか検証を行う。図 4 に拡張 K05 言語によるプログラム例を示す。

```
main(){
parallel(0,15){
                    write($p);
          }
}
```

図 4　拡張 K05 言語プログラム

図 4 のプログラム例はプロセッサ番号 0 番から 15 番までのプロセッサが実行中のプロセッサ番号の値を持つ変数として処理され､並列に実行するというプログラムである。本研究で作成されたコンパイラにより図 4 のプログラムは図 5 に示す拡張 VSM アセンブラに変換される。

```
PUSHI   0
PUSHI   15
PARA
PUSHP
OUTPUT
SYNC
HALT
```

図 5 拡張 VSM アセンブラプログラム

図 5 の拡張 VSM アセンブラを PVSM で実行することにより図 6 に示す実行結果が得られる。

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Execution time      : 7
```

図 6 PVSM インタプリタの実行結果

以上の結果から、本研究で作成した PRAM コンパイラは正しくコンパイルされ、並列処理された結果が出力される事を確認した。また PVSM インタプリタの出力により、図 4 のプログラムを PRAM 上に実行させたときの実行時間を測定できる。

4　考察・今後の課題

本研究では並列アルゴリズムの設計、正当性の証明、およびその計算量の評価を実験的に行うためのツールである PRAM シミュレータの一部として PRAM コンパイラを作成した。また PRAM アルゴリズムを記述できる拡張 K05 言語および拡張 VSM アセンブラを作成した。本研究で作成した PRAM コンパイラを用いることにより PRAM アルゴリズムの設計、正当性の証明、およびその計算量の評価を実験的に行うことが出来る。

しかし今回作成したシミュレータには対応していない JAVA 言語の命令が多数存在する。また parallel 文中で parallel を使用できない構造になっている。よってこれらに対応させてさらに多様なアルゴリズムに対応させることが今後の課題である。

5　謝辞

本研究を行うにあたり、並列処理の基礎から自分たちの研究内容まで様々なご指導ご鞭撻をいただき、石水隆助手には誠に感謝しております。また同じ研究を目的とした共同研究者の板東俊助君、池田直樹君には様々な相談にものって頂き、深い感謝、敬愛の気持ちを表します。

参考文献

[1]　平成１７年度第５セメスター　情報・コンピュータシステムプロジェクトＩ指導書
[2]　Joseph JáJá：”An Introduction to Parallel Algorithms”,　Addison-Wesley(1992)

付録1 K05 言語の文法

<Program>::=<Main_fanction>
<Main_fanction>::= ”main” “(” “)”<Block>
<Block>::= “{” { <Var_decl> } { <Statement> } “}”
<Var_decl>::= (“int” | “boolean”) NAME [“[” INT”]”]
{“,” NAME [“[“ INT “]”] } “;”
<Statement>::= <If_statement> | <While_statement> | <Assignment>
| <Write_statement> | <Writechar_statement>
| “{” { <Statement> } “}” | “;”
<If_statement>::= “(” <Expression> “)” <Statement>
<While_statement> :: = “while” “(” <Expression> “)” <Statement>
<Assignment>::= <Lefthand_side> “=” <Expression> “;”
<Writeint_statement>::= “writeint” “(” <Expression> “)” “;”
<Writechar_statement>::= “writechar””(” <Expression> “)” “;”
<Lefthand_side>::= NAME | NAME “[” <Expression> “]”
<Expression> ::= <Logical_term> { “| |“ <Logical_term> }
<Logical_term> ::= <Logical_factor> { “&&“ <Logical_factor> }
<Logical_factor> ::= <Arithmetic_expression>
| <Arithmetic_expression> “==“ <Arithmetic_expression>
| <Arithmetic_expression> “!=“ <Arithmetic_expression>
| <Arithmetic_expression> “<“ <Arithmetic_expression>
| <Arithmetic_expression> “<=“ <Arithmetic_expression>
| <Arithmetic_expression> “>“ <Arithmetic_expression>
| <Arithmetic_expression> “>=“ <Arithmetic_expression>
<Arithmetic_expression> ::= <Arithmetic_term> { (“+“ | “-“) <Arithmetic_term> }
<Arithmetic_term> ::= <Arithmetic_factor> { (“*“ | “/“ | “%“) <Arithmetic_factor> }
<Arithmetic_factor> ::= <Unsigned_factor> | “!“ <Arithmetic_factor>
| “-“ <Arithmetic_factor>
<Unsigned_factor> ::= NAME | NAME “[” <Expression> “]“ | INT | CHAR
| “(“ <Expression> “)“ | “readint“ | “readchar“ | “true“ | “false“

付録2　拡張 K05 言語の文法

<Program>::=<Main_fanction>

<Main_fanction>::= "main" "(" ")"<Block>

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

<Var_decl>::= ("int" | "boolean") NAME ["[" INT"]"]
{"," NAME ["[" INT "]"] } ";"

<Statement>::= <If_statement> | <While_statement> | <Para_statement>
|<Assignment>| <Write_statement>
| <Writechar_statement>| "{" { <Statement> } "}" | ";"

<If_statement>::= "(" <Expression> ")" <Statement>

<While_statement> :: = "while" "(" <Expression> ")" <Statement>

<Para_statement>::= "parallel" "(" <Expression> "," <Expression> ")" <Statement>

<Assignment>::= <Lefthand_side> "=" <Expression> ";"

<Writeint_statement>::= "writeint" "(" <Expression> ")" ";"

<Writechar_statement>::= "writechar""(" <Expression> ")" ";"

<Lefthand_side>::= NAME | NAME "[" <Expressi

<Expression> ::= <Logical_term> { "||" <Logical_term> }

<Logical_term> ::= <Logical_factor> { "&&" <Logical_factor> }

<Logical_factor> ::= <Arithmetic_expression>
| <Arithmetic_expression> "==" <Arithmetic_expression>
| <Arithmetic_expression> "!=" <Arithmetic_expression>
| <Arithmetic_expression> "<" <Arithmetic_expression>
| <Arithmetic_expression> "<=" <Arithmetic_expression>
| <Arithmetic_expression> ">" <Arithmetic_expression>
| <Arithmetic_expression> ">=" <Arithmetic_expression>

<Arithmetic_expression> ::= <Arithmetic_term> { ("+" | "-") <Arithmetic_term> }

<Arithmetic_term> ::= <Arithmetic_factor> { ("*" | "/" | "%") <Arithmetic_factor> }

<Arithmetic_factor> ::= <Unsigned_factor> | "!" <Arithmetic_factor>
| "-" <Arithmetic_factor>

<Unsigned_factor> ::= NAME | NAME "[" <Expression> "]" | INT | CHAR
|"$p"| "(" <Expression> ")" | "readint" | "readchar" | "true" | "false"

付録3 VSM アセンブラの文法

```
ASSGN:
        addr = Stack [--SP];
        Dseg[addr] = Stack[SP] = Stack [SP+1];
ADD:    BINOP(+);
SUM:    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¥n”, Stack [SP--]
OUTPUTC:printf(“%15c¥n”, Stack [SP--]
LOAD:   Stack [SP] = Dseg[Stack [SP]];
```

付録4 PRAM コンパイラプログラム

本研究用いた PRAM コンパイラプログラムをいかに示す。本研究で作成したプログラムは以下の構成からなる。

(1) InputFile.java
(2) Instraction.java
(3) InstractionSegment.java
(4) LexicalAnalyzer.java
(5) Operators.java
(6) Symbol.java
(7) TruthValue.java
(8) Type.java
(9) Var.java
(10) VarTable.java
(11) Pram.java

(1) InputFile.java

```
import ioTools.*;
import java.io.*;

public class InputFile {
    BufferedReader buffer;     /* 入力ファイルのバッファ */
    String line;               /* 入力ファイルの 1 行分の文字列 */
    int linenum;               /* 入力ファイルの行番号 */
    int columnnum;             /* 入力ファイルの列番号 */
    char currentc;             /* 読み込んだ文字 */
    char nextc;                /* 次に読み込む文字 */

    /* コンストラクタでは, inputFileName というファイルを開き,
         そのファイルを今後 buffer で参照する．また linenum,
         columnnum, currentc, nextc を初期化する */
    public InputFile(String inputFileName) {
        buffer = FileIo.fRead(inputFileName);
        linenum = 0;
        columnnum = 0;
        //入力ファイルから一行読む
        readInputFile();
        //最初の一文字目を読んで, その文字を nextc に格納する．
        nextc = ' ';
```

```
		nextChar();
	}

	//buffer から一行読み，文字列変数 line にその行を格納するメソッド.
	public void readInputFile() {
		try {
			line = buffer.readLine();
		} catch(IOException error_report) {
			/* 読込みエラーが発生したら，キャッチした例外を表示し，
			   ファイルを閉じ，処理系を終了させる */
			System.out.println(error_report);
			closeFile();
			System.exit(1);
		}
	}

	//入力ファイルを閉じるメソッド.
	public void closeFile() {
		try {
			buffer.close();
		} catch(IOException error_report) {
			System.out.println(error_report);
			System.exit(1);
		}
	}

	//次の文字を得るメソッド.（問題 2.7 で作成する）
	public char nextChar() {
		if (line == null) { //ファイル末に達したら'¥0'を返す.
			currentc = nextc;
			nextc = '¥0';
			return currentc;
		} else if (columnnum >= line.length()) { //行末に達したら'¥n'を返す
			readInputFile();
			currentc = nextc;
			nextc = '¥n';
			linenum++;
			columnnum = 0;
			return currentc;
```

```
        } else { //通常の動作．読んだ一文字を nextc に格納し，その値を返す．
            currentc = nextc;
            nextc = line.charAt(columnnum);
            columnnum++;
            return currentc;
        }
    }

    public static void main(String[] args) {
        InputFile inFile = new InputFile("bsort.k");
        do {
            do { //この do-while 文は問題 2.7 でのみ必要な処理である．
                inFile.nextChar();
                System.out.print(inFile.currentc);
            } while(inFile.currentc != '¥n' && inFile.currentc != '¥0');

            /* 次の 2 行は，問題 2.6 でのみ必要な処理である．
            System.out.println(inFile.line);
            inFile.readInputFile();
            */
        } while(inFile.line != null);
    }
}
```

(2) Instraction.java

```
class Instraction implements Operators {
        int operator;           /* オペレータ */
        int operand;            /* int 型オペランド */
        double doubleOperand;   /* double 型オペランド */
        String stringOperand;   /* String 型オペランド*/
        boolean register;       /* アドレス修飾 */

        // オペランドを持たない場合のコンストラクタ
        public Instraction (int op) {
                operator = op;
                operand = Integer.MAX_VALUE;
                doubleOperand = Double.NaN;
                stringOperand = "";
                register = false;
```

```
}

public Instraction (int op, boolean r) {
		operator = op;
		operand = Integer.MAX_VALUE;
		doubleOperand = Double.NaN;
		stringOperand = "";
		register = r;
}

// int 型オペランドを持つ場合のコンストラクタ
public Instraction (int op, int i) {
		operator = op;
		operand = i;
		doubleOperand = Double.NaN;
		stringOperand = "";
		register = false;
}

public Instraction (int op, int i, boolean r) {
		operator = op;
		operand = i;
		doubleOperand = Double.NaN;
		stringOperand = "";
		register = r;
}

// char 型オペランドを持つ場合のコンストラクタ
public Instraction (int op, char c) {
		operator = op;
		operand = (int) c;
		doubleOperand = Double.NaN;
		stringOperand = "";
		register = false;
}

public Instraction (int op, char c, boolean r) {
		operator = op;
		operand = (int) c;
```

```
		doubleOperand = Double.NaN;
		stringOperand = "";
		register = r;
	}

	// double 型オペランドを持つ場合のコンストラクタ
	public Instraction (int op, double d) {
		operator = op;
		operand = Integer.MAX_VALUE;
		doubleOperand = d;
		stringOperand = "";
		register = false;
	}

	public Instraction (int op, double d, boolean r) {
		operator = op;
		operand = Integer.MAX_VALUE;
		doubleOperand = d;
		stringOperand = "";
		register = r;
	}

	// String 型オペランドを持つ場合のコンストラクタ
	public Instraction (int op, String str) {
		operator = op;
		operand = Integer.MAX_VALUE;
		doubleOperand = Double.NaN;
		stringOperand = str;
		register = false;
	}

	public Instraction (int op, String str, boolean r) {
		operator = op;
		operand = Integer.MAX_VALUE;
		doubleOperand = Double.NaN;
		stringOperand = str;
		register = r;
	}
```

```
public String toString() {
	char c;
	switch(operator) {
	case PUSH:					/* int 型オペランドを持つ場合 */
	case PUSHI:
	case POP:
	case SETFR:
	case INCFR:
	case DECFR:
	case BEQ:
	case BNE:
	case BLE:
	case BLT:
	case BGE:
	case BGT:
	case JUMP:
	case CALL:
		return opName() + "¥t" + operand + "¥t";
	case PUSHC:					/* char 型オペランドを持つ場合 */
		c = (char) operand;
		switch (c) {
		case '¥0':
			return opName() + "¥t¥'¥¥0¥'¥t";
		case '¥b':
			return opName() + "¥t¥'¥¥b¥'¥t";
		case '¥n':
			return opName() + "¥t¥'¥¥n¥'¥t";
		case '¥r':
			return opName() + "¥t¥'¥¥r¥'¥t";
		case '¥t':
			return opName() + "¥t¥'¥¥t¥'¥t";
		default:
			return opName() + "¥t¥'" + c + "¥'¥t";
		}
	case PUSHD:					/* double 型オペランドを持つ場合 */
		return opName() + "¥t" + doubleOperand + "¥t";
	case PUSHB:					/* boolean 型オペランドを持つ場合 */
		return opName() + (operand != 0);
	case PUSHS:					/* String 型オペランドを持つ場合 */
```

```
                String str = "";
                for (int i=0; i<stringOperand.length(); i++) {
                        c = stringOperand.charAt(i);
                        switch (c) {
                        case '¥0':
                                str += "¥¥0";
                                break;
                        case '¥b':
                                str += "¥¥b";
                                break;
                        case '¥n':
                                str += "¥¥n";
                                break;
                        case '¥r':
                                str += "¥¥r";
                                break;
                        case '¥t':
                                str += "¥¥t";
                                break;
                        default:
                                str += stringOperand.charAt(i);
                                break;
                        }
                }
                return opName() + "¥t¥"" + str + "¥"¥t";
        default:
                return opName() + "¥t¥t";               /* オペランドを持たない場合 */
        }
    }

// オペランドコードをオペランド名に変換
    public String opName() {
        switch(operator) {
        case NOP:       return "NOP     ";      // no operation
        case ASSGN:     return "ASSGN   ";      // assign
        case ADD:       return "ADD     ";      // +
        case ADDLHS:    return "ADDLHS ";       // +=
        case SUB:       return "SUB     ";      // -
        case SUBLHS:    return "SUBLHS ";       // -=
```

```
case MUL:         return "MUL      ";       // *
case MULLHS:      return "MULLHS ";         // *=
case DIV:         return "DIV      ";       // /
case DIVLHS:      return "DIVLHS ";         // /=
case MOD:         return "MOD      ";       // %
case MODLHS:      return "MODLHS ";         // %=
case CSIGN:       return "CSIGN   ";        // 単項-
case AND:         return "AND      ";       // and
case OR:          return "OR       ";       // or
case NOT:         return "NOT      ";       // not
case XOR:         return "XOR      ";       // exclusive or
case COMP:        return "COMP     ";       // comp
case COPY:        return "COPY     ";       // copy
case PUSH:        return "PUSH     ";       // push
case PUSHI:       return "PUSHI   ";        // push integer
case PUSHC:       return "PUSHC   ";        // push char
case PUSHD:       return "PUSHD   ";        // push double
case PUSHB:       return "PUSHB   ";        // push boolean
case PUSHS:       return "PUSHS   ";        // push string
case POP:         return "POP      ";       // pop
case REMOVE:      return "REMOVE ";         // remove
case LOAD:        return "LOAD     ";       // load
case INC:         return "INC      ";       // ++
case DEC:         return "DEC      ";       // --
case PREINC:      return "PREINC ";         // 前置++
case PREDEC:      return "PREDEC ";         // 前置--
case POSTINC:     return "POSTINC";         // 後置++
case POSTDEC:     return "POSTDEC";         // 後置--
case SETFR:       return "SETFR   ";        // set frame register
case INCFR:       return "INCFR   ";        // inc frame register
case DECFR:       return "DECFR   ";        // dec frame register
case JUMP:        return "JUMP     ";       // jump
case BEQ:         return "BEQ      ";       // == ?
case BNE:         return "BNE      ";       // != ?
case BLT:         return "BLT      ";       // < ?
case BLE:         return "BLE      ";       // <= ?
case BGT:         return "BGT      ";       // > ?
case BGE:         return "BGE      ";       // >= ?
case CALL:        return "CALL     ";       // call
```

```
			case RET:		return "RET      ";	// return
			case INPUT:		return "INPUT   ";	// input integer
			case INPUTC:	return "INPUTC ";	// input character
			case INPUTD:	return "INPUTD ";	// input double
			case INPUTS:	return "INPUTS ";	// input string
			case OUTPUT:	return "OUTPUT ";	// output integer
			case OUTPUTC:	return "OUTPUTC";	// output character
			case OUTPUTD:	return "OUTPUTD";	// output double
			case OUTPUTL:	return "OUTPUTL";	// output line
			case OUTPUTS:	return "OUTPUTS";	// output string
			case CASTI:		return "CASTI   ";	// cast int
			case CASTC:		return "CASTC   ";	// cast char
			case CASTD:		return "CASTD   ";	// cast double
			case CASTB:		return "CASTB   ";	// cast boolean
			case CASTS:		return "CASTS   ";	// cast string;
			case RAND:		return "RAND    ";	// random
			case HALT:		return "HALT    ";	// halt
			case PARA:		return "PARA    ";	// parallel
			case SYNC:		return "SYNC    ";	// syncronous
			case PUSHP:		return "PUSHP   ";	// push processor number
			case EOF:		return "EOF      ";	// end of file
			default:		return "ERROR   ";	// error
			}
		}
}
```

(3) InstractionSegment.java

```
import ioTools.*;	//ファイル入出力用
import java.io.*;	//ファイル入出力用
import java.util.ArrayList; //ArrayList 処理用

public class InstractionSegment implements Operators {
		ArrayList iseg;
		int isegPtr;
		int size;

		boolean debugSW;

		public InstractionSegment(boolean dsw) {
```

```
		iseg = new ArrayList();
		isegPtr = 0;
		size = 0;
		debugSW = dsw;
	}

	// Iseg に Instraction 型命令を加える
	public int appendCode (Instraction inst) {
		if (size == isegPtr) {
			if (debugSW) System.out.println(isegPtr+": "+inst);
			iseg.add (inst);
			size++;
		} else {
			Instraction oldInst = ((Instraction) iseg.get(isegPtr));
			if (debugSW) System.out.print (isegPtr+": "+oldInst);
			iseg.remove (isegPtr);
			iseg.add (isegPtr, inst);
			if (debugSW) System.out.println ("-> "+inst);
		}
		isegPtr++;
		return isegPtr-1;
	}

// Iseg にオペランド無し命令を加える
	public int appendCode (int operator) {
		if (size == isegPtr) {
			Instraction inst = new Instraction (operator);
			if (debugSW) System.out.println (isegPtr+": "+inst);
			iseg.add (inst);
			size++;
		} else {
			Instraction inst = ((Instraction) iseg.get (isegPtr));
			if (debugSW) System.out.print (isegPtr+": "+inst);
			inst.operator = operator;
			inst.operand = Integer.MAX_VALUE;
			inst.doubleOperand = Double.NaN;
			inst.stringOperand = "";
			inst.register = false;
			if (debugSW) System.out.println ("-> "+inst);
```

```
		}
		isegPtr++;
		return isegPtr-1;
	}

	public int appendCode (int operator, boolean register) {
		if (size == isegPtr) {
			Instraction inst = new Instraction (operator, register);
			if (debugSW) System.out.println (isegPtr+": "+inst);
			iseg.add (inst);
			size++;
		} else {
			Instraction inst = ((Instraction) iseg.get (isegPtr));
			if (debugSW) System.out.print (isegPtr+": "+inst);
			inst.operator = operator;
			inst.operand = Integer.MAX_VALUE;
			inst.doubleOperand = Double.NaN;
			inst.stringOperand = "";
			inst.register = register;
			if (debugSW) System.out.println ("-> "+inst);
		}
		isegPtr++;
		return isegPtr-1;
	}

	// Iseg に int 型オペランド付命令を加える
	public int appendCode (int operator, int operand) {
		if (size == isegPtr) {
			Instraction inst = new Instraction (operator, operand);
			if (debugSW) System.out.println (isegPtr+": "+inst);
			iseg.add (inst);
			size++;
		} else {
			Instraction inst = ((Instraction) iseg.get (isegPtr));
			replace (isegPtr, inst, operator, operand, Double.NaN, "", false);
		}
		isegPtr++;
		return isegPtr-1;
	}
```

```
public int appendCode (int operator, int operand, boolean register) {
        if (size == isegPtr) {
                Instraction inst = new Instraction (operator, operand, register);
                if (debugSW) System.out.println (isegPtr+": "+inst);
                iseg.add (inst);
                size++;
        } else {
                Instraction inst = ((Instraction) iseg.get (isegPtr));
                replace (isegPtr, inst, operator, operand, Double.NaN, "", register);
        }
        isegPtr++;
        return isegPtr-1;
}

// Iseg に double 型オペランド付命令を加える
public int appendCode (int operator, double doubleOperand) {
        if (size == isegPtr) {
                Instraction inst = new Instraction (operator, doubleOperand);
                if (debugSW) System.out.println (isegPtr+": "+inst);
                iseg.add (inst);
                size++;
        } else {
                Instraction inst = ((Instraction) iseg.get (isegPtr));
                replace (isegPtr, inst, operator, Integer.MAX_VALUE, doubleOperand, "", false);
        }
        isegPtr++;
        return isegPtr-1;
}

public int appendCode (int operator, double doubleOperand, boolean register) {
        if (size == isegPtr) {
                Instraction inst = new Instraction (operator, doubleOperand, register);
                if (debugSW) System.out.println (isegPtr+": "+inst);
                iseg.add (inst);
                size++;
        } else {
                Instraction inst = ((Instraction) iseg.get (isegPtr));
                replace (isegPtr, inst, operator, Integer.MAX_VALUE, doubleOperand, "",
```

```
register);
                }
                isegPtr++;
                return isegPtr-1;
        }

        // Iseg に String 型オペランド付命令を加える
        public int appendCode (int operator, String stringOperand) {
                if (size == isegPtr) {
                        Instraction inst = new Instraction (operator, stringOperand);
                        if (debugSW) System.out.println (isegPtr+": "+inst);
                        iseg.add (inst);
                        size++;
                } else {
                        Instraction inst = ((Instraction) iseg.get (isegPtr));
                        replace (isegPtr, inst, operator, Integer.MAX_VALUE, Double.NaN,
stringOperand, false);
                }
                isegPtr++;
                return isegPtr-1;
        }

        public int appendCode (int operator, String stringOperand, boolean register) {
                if (size == isegPtr) {
                        Instraction inst = new Instraction (operator, stringOperand, register);
                        if (debugSW) System.out.println (isegPtr+": "+inst);
                        iseg.add (inst);
                        size++;
                } else {
                        Instraction inst = ((Instraction) iseg.get (isegPtr));
                        replace (isegPtr, inst, operator, Integer.MAX_VALUE, Double.NaN,
stringOperand, register);
                }
                isegPtr++;
                return isegPtr-1;
        }

        public int operator (int addr) {        /* オペレータを返す */
                if (addr < 0 || addr >= size)
```

```
			executeError("Illegal iseg address ", addr);
		return ((Instraction) iseg.get (addr)).operator;
	}

	public int operand (int addr) {		/*  オペランドを返す  */
		if (addr < 0 || addr >= size)
			executeError("Illegal iseg address ", addr);
		return ((Instraction) iseg.get (addr)).operand;
	}

	public char charOperand (int addr) {		/* char 型オペランドを返す  */
		if (addr < 0 || addr >= size)
			executeError("Illegal iseg address ", addr);
		return (char) ((Instraction) iseg.get (addr)).operand;
	}

	public double doubleOperand (int addr) {	/* double 型オペランドを返す  */
		if (addr < 0 || addr >= size)
			executeError("Illegal iseg address ", addr);
		return ((Instraction) iseg.get (addr)).doubleOperand;
	}

	public boolean boolOperand (int addr) {		/* boolean 型オペランドを返す  */
		if (addr < 0 || addr >= size)
			executeError("Illegal iseg address ", addr);
		return (((Instraction) iseg.get (addr)).operand != 0);
	}

	public String stringOperand (int addr) {	/* String 型オペランドを返す  */
		if (addr < 0 || addr >= size)
			executeError("Illegal iseg address ", addr);
		return ((Instraction) iseg.get (addr)).stringOperand;
	}

	public boolean register (int addr) { /*  レジスタを返す  */
		if (addr < 0 || addr >= size)
			executeError("Illegal iseg address ", addr);
		return ((Instraction) iseg.get (addr)).register;
	}
```

```
// 命令のジャンプ先のアドレスをチェック
	public void checkAddress (int addr) {
		Instraction inst = (Instraction) iseg.get(addr);
		switch(inst.operator) {
		case JUMP:
		case BEQ:
		case BNE:
		case BLT:
		case BLE:
		case BGT:
		case BGE:
		case CALL:
			if(inst.operand < 0 | | inst.operand >= size)
				syntaxError ("Illegal iseg address : " + inst, addr);
			break;
		default:
			break;
		}
	}

// 指定したアドレスの命令を表示
	public void print (int addr) {;
		System.out.print(addr + ": " + (Instraction) iseg.get (addr));
	}

// Iseg を表示
	public void dump() {
		for (int i=0; i<isegPtr; i++)
			System.out.println(i + ": " + (Instraction) iseg.get (i));
	}

// Iseg をデフォルトファイルに出力
	public void dump2file() {
		PrintWriter outputFile = FileIo.fWrite ("OpCode.asm", false);
		for (int i=0; i<isegPtr; i++)
			outputFile.println ((Instraction) iseg.get (i));
		outputFile.close();
	}
```

```
    // Iseg を指定したファイルに出力
        public void dump2file (String fileName) {
                PrintWriter outputFile = FileIo.fWrite (fileName, false);
                for (int i=0; i<isegPtr; i++)
                        outputFile.println ((Instraction) iseg.get (i));
                outputFile.close();
        }

        // 命令のオペレータ、オペランドを変更する
        void replace (int addr, Instraction inst, int operator, int operand, double doubleOperand, String
stringOperand, boolean register) {
                if (debugSW)
                        System.out.print (addr + ": " + inst);
                inst.operator = operator;
                inst.operand = operand;
                inst.doubleOperand = doubleOperand;
                inst.stringOperand = stringOperand;
                inst.register = register;
                if (debugSW)
                        System.out.println ("-> " + inst);
        }

    // addr 番目の命令の オペレータ，オペランド を operator, operand に変更する
        public void replaceCode (int addr, int operator, int operand) {
                Instraction inst = ((Instraction) iseg.get (addr));
                replace (addr, inst, operator, operand, Double.NaN, "", inst.register);
        }

        // addr 番目の命令のオペランド を operand に変更する
        public void replaceCode (int addr, int operand) {
                Instraction inst = ((Instraction) iseg.get (addr));
                replace (addr, inst, inst.operator, operand, Double.NaN, "", inst.register);
        }

        // addr 番目の命令の オペレータ，オペランド を operator, doubleOperand に変更する
        public void replaceCode (int addr, int operator, double doubleOperand) {
                Instraction inst = ((Instraction) iseg.get (addr));
                replace (addr, inst, operator, Integer.MAX_VALUE, doubleOperand, "", inst.register);
```

```
	}

	// addr 番目の命令のオペランド を doubleOperand に変更する
	public void replaceCode (int addr, double doubleOperand) {
		Instraction inst = ((Instraction) iseg.get (addr));
		replace (addr, inst, inst.operator, Integer.MAX_VALUE, doubleOperand, "",
inst.register);
	}

	// addr 番目の命令の オペレータ, オペランド を operator, stringOperand に変更する
	public void replaceCode (int addr, int operator, String stringOperand) {
		Instraction inst = ((Instraction) iseg.get (addr));
		replace (addr, inst, operator, Integer.MAX_VALUE, Double.NaN, stringOperand,
inst.register);
	}

	// addr 番目の命令のオペランド を stringOperand に変更する
	public void replaceCode (int addr, String stringOperand) {
		Instraction inst = ((Instraction) iseg.get (addr));
		replace (addr, inst, inst.operator, Integer.MAX_VALUE, Double.NaN, stringOperand,
inst.register);
	}

	// addr 番目の命令を inst に変更する
	public void replaceCode (int addr, Instraction inst) {
		Instraction oldInst = ((Instraction) iseg.get (addr));
		if (debugSW) System.out.print(addr+": "+oldInst);
		iseg.remove (addr);
		iseg.add (addr, inst);
		if (debugSW) System.out.println ("-> "+inst);
	}

	void syntaxError (String err_mes, int addr) {	/* 文法エラー */
		System.out.println ("Syntax error at line " + addr);
		System.out.println (err_mes);
		System.out.println ((Instraction) iseg.get (addr));
		System.exit(1);
	}
```

```
        void executeError (String err_mes, int addr) {     /* 実行時エラー */
                System.out.println ("Execute error at line " + addr);
                System.out.println (err_mes);
                System.out.println ((Instraction) iseg.get (addr));
                System.exit (1);
        }
}
```

(4) LexicalAnalyzer.java

```
public class LexicalAnalyzer implements Symbol {
    int ttype;              /* トークンの型 */
    int value;              /* 整数の場合その値 文字の場合文字コード */
    String name;            /* 変数の場合その名前 */
        String string;          /* 文字列の場合その文字列自身 */
    InputFile inFile;       /* InputFile クラスのインスタンス（入力ファイル) */


    //コンストラクタでは，入力ファイルの読込みと，各種初期化を行う．
    public LexicalAnalyzer(String fname) {
                //入力ファイルを開く
                inFile = new InputFile(fname);

                //フィールドの初期化
                value = 0;
                name = null;
        string = null;
    }


    public int nextToken() {                    /* 字句解析 次のトークンを得る */
        ttype = S_NULL;
        char c;

        do {                                        /* 空白をスキップ */
            c = inFile.nextChar();
        } while (c == ' ' | | c == '¥t' | | c == '¥n');

        if (c == '¥0') ttype = S_EOF;               /* End of file */
        else if (c == '0') {                        /* 符号無し整数(0) */
```

```
            value = 0;
            ttype = S_INTEGER;
        } else if (Character.isDigit(c)) {                    /* 符号無し整数 */
            value = extractIntValue(c);
            ttype = S_INTEGER;
        } else if (Character.isLowerCase(c) || Character.isUpperCase(c)
                    || c=='_') {
            String str = extractWord(c);
            switch (c) {
            case 'b':
                        if (str.equals("boolean")) ttype = S_BOOLEAN;/* boolean */
                        else ttype = S_NAME;                         /* 変数名 */
                break;

            case 'f':
                        if (str.equals("false")) ttype = S_FALSE;      /* false */
                        else if (str.equals ("for")) ttype = S_FOR;    /* for */
                        else ttype = S_NAME;                         /* 変数名 */
                break;

            case 'i':
                        if (str.equals("if")) ttype = S_IF;            /* if */
                        else if (str.equals("int")) ttype = S_INT;     /* int */
                        else ttype = S_NAME;                         /* 変数名 */
                break;

            case 'm':
                        if (str.equals("main")) ttype = S_MAIN;        /* main */
                        else ttype = S_NAME;                         /* 変数名 */
                break;

            case 'p':
                    if(str.equals("parallel")) ttype = S_PARALLEL; /* parallel */
            break;

            case 'r':
                if (str.equals("readint")) ttype = S_READINT;/* readint */
                        else    if    (str.equals("readchar"))    ttype    =    S_READCHAR;
/* readchar */
```

```
                    else ttype = S_NAME;                    /* 変数名 */
                break;

            case 's':
                if (str.equals ("String")) ttype = S_STRING;   /* String */
                    else ttype = S_NAME;                    /* 変数名 */
                    break;

            case 't':
                    if (str.equals("true")) ttype = S_TRUE;      /* true */
                    else ttype = S_NAME;                    /* 変数名 */
                break;

            case 'w':
                if (str.equals("while")) ttype = S_WHILE;     /* while */
                else if (str.equals ("write")) ttype = S_WRITE;                /* write */
                    else if (str.equals ("writedouble")) ttype = S_WRITEDOUBLE;
/* writedouble */
                    else if (str.equals ("writechar")) ttype = S_WRITECHAR;
/* writechar */
                    else if (str.equals ("writeint")) ttype = S_WRITEINT;              /*
writeint */
                    else if (str.equals ("writechar")) ttype = S_WRITECHAR;
/* writechar */
                    else if (str.equals ("writeln")) ttype = S_WRITELN;              /*
writeln */
                    else if (str.equals ("writestr")) ttype = S_WRITESTR;             /*
writestr */
                else ttype = S_NAME;                    /* 変数名 */
                break;

            default:
                    ttype = S_NAME;                    /* 変数名 */
                break;
            }
            name = str;
        } else {
            switch(c) {
            case '(':
```

```
        ttype = S_LPAREN;                          /* ( */
        break;
    case ')':
        ttype = S_RPAREN;                          /* ) */
        break;
    case '{':
        ttype = S_LBRACE;                          /* { */
        break;
    case '}':
        ttype = S_RBRACE;                          /* } */
        break;
    case '[':
        ttype = S_LBRACKET;                        /* [ */
        break;
    case ']':
        ttype = S_RBRACKET;                        /* ] */
        break;
    case ',':
        ttype = S_COMMA;                           /* , */
        break;
    case ';':
        ttype = S_SEMICOLON;                       /* ; */
        break;
    case '+':
        ttype = S_ADD;                             /* + */
        break;
    case '-':
        ttype = S_SUB;                             /* - */
        break;
    case '*':
        ttype = S_MUL;                             /* * */
        break;
    case '/':
        ttype = S_DIV;                             /* / */
        break;
    case '%':
        ttype = S_MOD;                             /* % */
        break;
    case '=':
```

```
        if (inFile.nextc == '=') {                    /* == */
            inFile.nextChar();
            ttype = S_EQUAL;
        } else ttype = S_ASSIGN;                          /* = */
        break;
    case '<':
        if (inFile.nextc == '=') {                    /* <= */
            inFile.nextChar();
            ttype = S_LESSEQ;
        } else ttype = S_LESS;                            /* < */
        break;
    case '>':
        if (inFile.nextc == '=') {                    /* >= */
            inFile.nextChar();
            ttype = S_GREATEQ;
        } else ttype = S_GREAT;                           /* < */
        break;
    case '!':
        if (inFile.nextc == '=') {                    /* != */
            inFile.nextChar();
            ttype = S_NOTEQ;
        } else ttype = S_NOT;                             /* ! */
        break;
    case '&':
        if(inFile.nextc == '&') {                     /* && */
            inFile.nextChar();
            ttype = S_AND;
        } else syntaxError();
        break;
    case '|':
        if (inFile.nextc == '|') {                    /* || */
            inFile.nextChar();
            ttype = S_OR;
        } else syntaxError();
        break;
    case '¥'':                                            /* 文字 */
        c = inFile.nextChar();
        if (c != '¥'' && inFile.nextc == '¥'') {
            inFile.nextChar();
```

```
                    ttype = S_CHARACTER;
                    value = (int) c;
                } else syntaxError();
                break;
            case '"' :
                String str = "";
                c = inFile.nextChar();
                while (c != '¥"') {
                        str += c;
                        c = inFile.nextChar();
                }
                ttype = S_STR;
                string = str;
                break;
                case '$':
                if (inFile.nextc == 'p') {                      /* $p */
                        inFile.nextChar();
                        ttype = S_PROCESSOR;
                } else syntaxError();
                break;


            default:
                syntaxError();
                break;
            }
        }
        return ttype;
}

public int extractIntValue(char c) {            /* c で始まる整数を得る */
        int v = Character.digit(c, 10);         /* 文字を整数に変換 */
        while (Character.isDigit(inFile.nextc)) {
            c = inFile.nextChar();
            v = v * 10 + Character.digit(c, 10);
        }
        return v;
}
```

```
	public String extractWord(char c) {		/* c で始まる文字列を得る */
		String s = String.valueOf(c);
		while (Character.isLowerCase(inFile.nextc)
				|| Character.isUpperCase(inFile.nextc)
				|| Character.isDigit(inFile.nextc)
				|| inFile.nextc=='_') {
			c = inFile.nextChar();
			s = s + c;
		}
		return s;
	}

	public void syntaxError() {			/* 文法エラー */
		System.out.println("Systax error at line " + inFile.linenum);
		System.out.println(inFile.line);
		inFile.closeFile();
		System.exit(1);
	}
}
```

(5) Operators.java

```
interface Operators {
		static final int NOP	=  0;	// no operation
		static final int ASSGN	=  1;	// assign
		static final int ADD	=  2;	// +
		static final int ADDLHS	=  3;	// +=
		static final int SUB	=  4;	// -
		static final int SUBLHS	=  5;	// -=
		static final int MUL	=  6;	// *
		static final int MULLHS	=  7;	// *=
		static final int DIV	=  8;	// /
		static final int DIVLHS	=  9;	// /=
		static final int MOD	= 10;	// %
		static final int MODLHS	= 11;	// %=
		static final int CSIGN	= 12;	// 単項-
		static final int AND	= 13;	// and
		static final int OR	= 14;	// or
		static final int NOT	= 15;	// not
```

```
static final int XOR       = 16;  // exclusive or
static final int COMP      = 17;  // comp
static final int COPY      = 18;  // copy
static final int PUSH      = 19;  // push
static final int PUSHI     = 20;  // push integer
static final int PUSHC     = 21;  // push character
static final int PUSHD     = 22;  // push double
static final int PUSHB     = 23;  // push boolean
static final int PUSHS     = 24;  // push string
static final int REMOVE    = 25;  // remove
static final int LOAD      = 26;  // load
static final int POP       = 27;  // pop
static final int INC       = 28;  // ++
static final int DEC       = 29;  // --
static final int PREINC    = 30;  // 前置++
static final int PREDEC    = 31;  // 前置--
static final int POSTINC = 32;  // 後置++
static final int POSTDEC = 33;  // 後置--
static final int SETFR     = 34;  // set frame register
static final int INCFR     = 35;  // inc frame register
static final int DECFR     = 36;  // dec frame register
static final int JUMP      = 37;  // jump
static final int BLT       = 38;  // < ?
static final int BLE       = 39;  // <= ?
static final int BEQ       = 40;  // == ?
static final int BNE       = 41;  // != ?
static final int BGE       = 42;  // > ?
static final int BGT       = 43;  // >= ?
static final int CALL      = 44;  // call
static final int RET       = 45;  // return
static final int INPUT     = 46;  // input integer
static final int INPUTC    = 47;  // input character
static final int INPUTD    = 48;  // input double
static final int INPUTS    = 49;  // input string
static final int OUTPUT    = 50;  // output integer
static final int OUTPUTC = 51;  // output character
static final int OUTPUTD = 52;  // output double
static final int OUTPUTB = 53;  // output boolean
static final int OUTPUTS = 54;  // output string
```

```
        static final int OUTPUTL = 55;   // output line
        static final int CASTI     = 56;   // cast to integer;
        static final int CASTC     = 57;   // cast to char;
        static final int CASTD     = 58;   // cast to double;
        static final int CASTB     = 59;   // cast to boolean;
        static final int CASTS     = 60;   // cast to string;
        static final int RAND      = 61;   // random
        static final int HALT      = 62;   // halt
        static final int PARA      = 63;   // parallel
        static final int SYNC      = 64;   // synchronous
        static final int PUSHP     = 65;   // push processor number
        static final int EOF       =255;   // end of file
        static final int ERROR     = -1;   // error
}
```

(6) Symbol.java

```
interface Symbol {
        static final int S_ERROR = -1;
        static final int S_NULL = 0;
        static final int S_MAIN = 1;            /* main */
        static final int S_IF = 2;              /* if */
        static final int S_ELSE = 3;            /* else */
        static final int S_WHILE = 4;           /* while */
        static final int S_FOR = 5;             /* for */
        static final int S_DO = 6;              /* do */
        static final int S_READINT = 7;         /* readint */
        static final int S_READCHAR = 8;        /* readchar */
        static final int S_READDOUBLE = 9;   /* readdouble */
        static final int S_READSTR = 10;        /* readstr */
        static final int S_WRITE = 11;          /* write */
        static final int S_WRITEINT = 12;       /* writeint */
        static final int S_WRITECHAR = 13;   /* writechar */
        static final int S_WRITEDOUBLE = 14;/* writedouble */
        static final int S_WRITESTR = 15;       /* writestr */
        static final int S_WRITELN = 16;        /* writeln */
        static final int S_RAND = 17;           /* rand */
        static final int S_INT = 18;            /* int */
        static final int S_CHAR = 19;           /* char */
```

```
static final int S_BOOLEAN = 20;       /* boolean */
static final int S_DOUBLE = 21;        /* double */
static final int S_STRING = 22;        /* String */
static final int S_EQUAL = 23;         /* == */
static final int S_NOTEQ = 24;         /* != */
static final int S_LESS = 25;          /* < */
static final int S_GREAT = 26;         /* > */
static final int S_LESSEQ = 27;        /* <= */
static final int S_GREATEQ = 28;       /* >= */
static final int S_AND = 29;           /* && */
static final int S_OR = 30;            /* || */
static final int S_NOT = 31;           /* ! */
static final int S_ADD =32;            /* + */
static final int S_SUB = 33;           /* - */
static final int S_MUL = 34;           /* * */
static final int S_DIV = 35;           /* / */
static final int S_MOD = 36;           /* % */
static final int S_ASSIGN = 37;        /* = */
static final int S_ADDLHS = 38;        /* += */
static final int S_SUBLHS = 39;        /* -= */
static final int S_MULLHS = 40;        /* *= */
static final int S_DIVLHS = 41;        /* /= */
static final int S_MODLHS = 42;        /* %= */
static final int S_INC = 43;           /* ++ */
static final int S_DEC = 44;           /* -- */
static final int S_SEMICOLON = 45;   /* ; */
static final int S_LPAREN = 46;        /* ( */
static final int S_RPAREN = 47;        /* ) */
static final int S_LBRACE = 48;        /* { */
static final int S_RBRACE = 49;        /* } */
static final int S_LBRACKET = 50;     /* [ */
static final int S_RBRACKET = 51;     /* ] */
static final int S_COMMA = 52;         /* , */
static final int S_INTEGER = 53;       /* 整数 */
static final int S_CHARACTER =54;     /* 文字 */
static final int S_NAME = 55;          /* 変数名 */
static final int S_TRUE = 56;          /* true */
static final int S_FALSE = 57;         /* false */
static final int S_DOUBLEVAL = 58;   /* 実数 */
```

```
        static final int S_STR= 59;          /*  文字列  */
        static final int S_PARALLEL = 60;    /* parallel */
        static final int S_PROCESSOR = 61;   /* $p */
        static final int S_EOF = 255;        /* end of file */
}
```

(7) TruthValue.java

```
interface TruthValue{
    static final int V_TRUE = 1;
    static final int V_FALSE = 0;
}
```

(8) Type.java

```
interface Type {
        static final int T_VOID          = 0;
        static final int T_INT           = 1;
        static final int T_ARRAYOFINT    = 2;
        static final int T_CHAR          = 3;
        static final int T_ARRAYOFCHAR   = 4;
        static final int T_BOOL          = 5;
        static final int T_ARRAYOFBOOL   = 6;
        static final int T_DOUBLE        = 7;
        static final int T_ARRAYOFDOUBLE = 8;
        static final int T_STRING        = 9;
        static final int T_ARRAYOFSTRING = 10;
        static final int T_ERROR         = 255;
}
```

(9) Var.java

```
public class Var {
    int type;              /*型*/
    String name;           /*  変数名  */
    int address;           /*  割り当てられるアドレス  */
    int size;              /*  配列型の場合そのサイズ*/

    public Var(int t, String n, int a, int s) {
```

```
        type = t;
        name = n;
        address = a;
        size = s;
    }
}
```

(10) VarTable.java

```
import java.util.Vector;

public class VarTable {
    Vector vt;
    int nextAddress;

    public VarTable() {
        vt = new Vector();
        vt.setSize(0);
        nextAddress = 0;
    }

    public boolean addElement(int t, String n, int s) { /* 表に変数挿入 */
        if(exist(n)) return false; /* 名前の重複チェック */
        vt.addElement(new Var(t, n, nextAddress, s));
        nextAddress += s;
        return true;
    }

    public boolean exist(String n) {        /* 既存の変数であるか */
        for(int i=0; i<vt.size(); i++)
            if(n.equals(((Var)vt.get(i)).name)) return true;
        return false;
    }

    public int getAddress(String n) {       /* 表から変数のアドレスを得る */
        int i;
        for(i=0; i<vt.size(); i++)
            if(n.equals(((Var)vt.get(i)).name)) break;
        if(i == vt.size()) return -1;   /* 表に存在しない場合 */
```

```
        else return ((Var)vt.get(i)).address;
    }

    public int getType(String n) {          /* 表から変数の型を得る */
        int i;
        for(i=0; i<vt.size(); i++)
            if(n.equals(((Var)vt.get(i)).name)) break;
        if(i == vt.size()) return -1;   /* 表に存在しない場合 */
        else return ((Var)vt.get(i)).type;
    }

    public int getSize(String n) {          /* 表から変数のサイズを得る */
        int i;
        for(i=0; i<vt.size(); i++)
            if(n.equals(((Var)vt.get(i)).name)) break;
        if(i == vt.size()) return -1;   /* 表に存在しない場合 */
        else return ((Var)vt.get(i)).size;
    }

    public static void main(String[] args) {
        VarTable varTable = new VarTable();
        String[]   varNames = {"p", "q", "r", "s", "t"};

        for(int i = 0; i < 5; i ++)
            varTable.addElement(0, varNames[i], 1);

        for(int i = 0; i < 5; i ++) {
            System.out.println("変数" + varNames[i] + "の¥n" +
                               "アドレス: " + varTable.getAddress(varNames[i])
                               + ",   型: " + varTable.getType(varNames[i])
                               + ",   サイズ: " + varTable.getSize(varNames[i])
                               );
        }
    }
}
```

(11) Pram.java

```
import ioTools.*;
```

```
public class Pram implements Operators, Symbol, Type, TruthValue {
	static LexicalAnalyzer lexer;
	static VarTable vt;
	static InstractionSegment iseg;
		static boolean inParallel;　//並列処理の有無を保持

	public static void main(String[] args){
		if (args.length == 0) {
				System.out.println
					("Usage: java Kc <file_name> [<objectfile_name]");
				System.exit(1);
			}

			lexer = new LexicalAnalyzer(args[0]);
			vt = new VarTable();
			iseg = new InstractionSegment(false);

			lexer.nextToken();
		inParallel = false;
			parseProgram();

			lexer.inFile.closeFile();

		if(args.length == 1){
			iseg.dump2file();
		}else {
			iseg.dump2file(args[1]);
		}

	}

		// プログラム文
	public static void parseProgram(){
			if (lexer.ttype==S_MAIN) lexer.nextToken();
			else syntaxError("Need main");
			parseMain_function();
			iseg.appendCode(HALT);
	}
```

```
	// メイン文
public static void parseMain_function() {
		if (lexer.ttype==S_LPAREN) lexer.nextToken();
	else syntaxError("Need (");
		if (lexer.ttype==S_RPAREN) lexer.nextToken();
		else syntaxError("Need )");

		parseBlock();
}

	// ブロック文
public static void parseBlock(){
	if (lexer.ttype==S_LBRACE) lexer.nextToken();
		else syntaxError("Need {");

		while (lexer.ttype==S_INT||lexer.ttype==S_BOOLEAN){
			parseVar_decl();
	  }

		while (lexer.ttype!=S_RBRACE){
			parseStatement();
		}

		lexer.nextToken();

}

	// 変数宣言
public static void parseVar_decl(){
	int type=0;
		String name = "";
		int size = 1;
	if (lexer.ttype==S_INT||lexer.ttype==S_BOOLEAN){
		type = lexer.ttype;
			if(type==S_INT)type=T_INT;
			if(type==S_BOOLEAN)type=T_BOOL;
			lexer.nextToken();
	}else syntaxError();
```

```
		if (lexer.ttype==S_NAME){
			name = lexer.name;
			if(vt.exist(name))syntaxError();
			lexer.nextToken();
		}else syntaxError();

		if (lexer.ttype==S_LBRACKET) {
			if(type==T_INT)type=T_ARRAYOFINT;
			if(type==T_BOOL)type=T_ARRAYOFBOOL;
			lexer.nextToken();

			if (lexer.ttype==S_INTEGER) {
				size=lexer.value;
				lexer.nextToken();
			}else syntaxError();

			if (lexer.ttype==S_RBRACKET) lexer.nextToken();
			else syntaxError("Need ]");
		}
	vt.addElement(type,name,size);	//変数登録

	while (lexer.ttype==S_COMMA){
		size=1;
			lexer.nextToken();
			if(type==T_ARRAYOFINT)type=T_INT;
			if(type==T_ARRAYOFBOOL)type=T_BOOL;

			if (lexer.ttype==S_NAME){
				name=lexer.name;
				if(vt.exist(name))syntaxError();
				lexer.nextToken();
			}else syntaxError();

		if (lexer.ttype==S_LBRACKET){
				if(type==T_INT)type=T_ARRAYOFINT;
				if(type==T_BOOL)type=T_ARRAYOFBOOL;
			lexer.nextToken();
				if (lexer.ttype==S_INTEGER){
```

```
                    size=lexer.value;
                    lexer.nextToken();
                }else syntaxError();

                if (lexer.ttype==S_RBRACKET) lexer.nextToken();
                else syntaxError("Need ]");
        }
        if(vt.addElement(type,name,size)==false){
                syntaxError();
        }
    }

        if (lexer.ttype==S_SEMICOLON) lexer.nextToken();
        else syntaxError();
}

    //Statement 文の作成
public static void parseStatement(){
    switch (lexer.ttype){
        case S_IF:
            parseIf_statement();
            break;

      case S_WHILE:
            parseWhile_statement();
            break;

      case S_NAME:
            parseAssignment();
            break;

        case S_FOR:
            parseFor_statement();
            break;

        case S_WRITE:                           /* writeint 文  */
            parseWrite_statement();
            break;
```

```
    case S_WRITEINT:                          /* writeint 文 */
            parseWriteint_statement();
            break;

    case S_WRITECHAR:                         /* writechar 文 */
            parseWritechar_statement();
            break;

    case S_WRITEDOUBLE:                       /* writedouble 文 */
            parseWritedouble_statement();
            break;

case S_LBRACE:
            lexer.nextToken();
            while (lexer.ttype!=S_RBRACE) {
                    parseStatement();
            }
            if (lexer.ttype==S_RBRACE) lexer.nextToken();
            else syntaxError("Need }");
            break;

case S_SEMICOLON:
            lexer.nextToken();
            break;

    case S_PARALLEL:
            parseParallel_statement();
            break;

    case S_PROCESSOR:
            parseExpression();
            if (lexer.ttype != S_SEMICOLON) syntaxError ("Need ;");
            lexer.nextToken();
            break;

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

```
}

	//Parallel 文
	public static void parseParallel_statement(){
			if (inParallel) syntaxError("Not Ready Parallel");
			inParallel = true;
			if (lexer.nextToken() != S_LPAREN) syntaxError ("Need (");
			lexer.nextToken();
			parseExpression();
			if (lexer.ttype != S_COMMA) syntaxError ("Need ,");
			lexer.nextToken();
			parseExpression();
			if (lexer.ttype != S_RPAREN) syntaxError ("Need )");
			lexer.nextToken();
			iseg.appendCode (PARA);
			parseStatement();
			iseg.appendCode (SYNC);
			inParallel = false;
	}

	//for  文の作成
public static void parseFor_statement(){
			int ktest=-1;
			if (lexer.ttype==S_FOR) lexer.nextToken();
	else syntaxError();

			if (lexer.ttype!=S_LPAREN) syntaxError("Need (");
			if (lexer.nextToken() != S_SEMICOLON) {
				parseExpression();			/*  初期式  */
				System.out.println(lexer.value);
				if (lexer.ttype != S_SEMICOLON) syntaxError ("Need ;");
			}

/*	if (lexer.ttype == S_SEMICOLON) parseExpression();
				else syntaxError ("Need ;");
	*/
			int ad1=iseg.isegPtr;
			ktest = parseExpression();
```

```
		if (ktest!=T_BOOL){
			syntaxError();
		}

	/*	if (lexer.ttype == S_SEMICOLON) parseExpression();
				else syntaxError ("Need ;");
	*/
		parseStatement();
		if (lexer.ttype==S_RPAREN) lexer.nextToken();
		else syntaxError("Need )");

		int ad2=iseg.appendCode(BEQ);
		parseStatement();
		iseg.appendCode(JUMP,ad1);
		iseg.replaceCode(ad2,iseg.isegPtr); //BEQ の分岐先アドレスを書き換える
}

	//If 文の作成
public static void parseIf_statement(){
		int ktest=-1;
		if (lexer.ttype==S_IF) lexer.nextToken();
	else syntaxError();

		if (lexer.ttype==S_LPAREN) lexer.nextToken();
		else syntaxError();

		ktest=parseExpression();
		if(ktest==T_INT||ktest==T_ARRAYOFINT){
			syntaxError();
		}

		if (lexer.ttype==S_RPAREN){
				lexer.nextToken();
		} else syntaxError();

		int ad=iseg.appendCode(BEQ);
		parseStatement();
		iseg.replaceCode(ad,iseg.isegPtr);
}
```

```
    //While 文の作成
public static void parseWhile_statement(){
            int ktest=-1;
            if (lexer.ttype==S_WHILE) lexer.nextToken();
            else syntaxError();

    if (lexer.ttype==S_LPAREN) lexer.nextToken();
            else syntaxError();

            int ad1=iseg.isegPtr;

            ktest=parseExpression();
            if(ktest==T_INT||ktest==T_ARRAYOFINT){
                syntaxError();
            }
            if (lexer.ttype==S_RPAREN) {

                lexer.nextToken();
            }
            else syntaxError();

            int ad2=iseg.appendCode(BEQ);
            parseStatement();
            iseg.appendCode(JUMP,ad1);
            iseg.replaceCode(ad2,iseg.isegPtr); //BEQ の分岐先アドレスを書き換える

}

    //Assignment 文の作成
public static void parseAssignment(){
            int ktest1=-1;
            int ktest2=-1;
            ktest1=parseLefthand_side();
            if (lexer.ttype==S_ASSIGN) lexer.nextToken();
            else syntaxError();

            ktest2=parseExpression();
    if((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
```

```
            ( ktest2==T_INT || ktest2==T_ARRAYOFINT)){
        }
        else if((ktest1==T_BOOL || ktest1==T_ARRAYOFBOOL) &&
                (ktest2==T_BOOL || ktest2==T_ARRAYOFBOOL)){
        }else {
                syntaxError();
        }

        if (lexer.ttype==S_SEMICOLON) lexer.nextToken();
        else syntaxError();

        iseg.appendCode(ASSGN);
        iseg.appendCode(REMOVE);
}
    //Write 文の作成
public static void parseWrite_statement(){
        int ktest=-1;
        if (lexer.ttype==S_WRITE) lexer.nextToken();
        else syntaxError();

        if (lexer.ttype==S_LPAREN) lexer.nextToken();
        else syntaxError();

        ktest=parseExpression();
        if(ktest != T_INT && ktest != T_CHAR
                && ktest != T_DOUBLE && ktest != T_BOOL
                && ktest != T_STRING && ktest != T_ARRAYOFINT
                && ktest != T_ARRAYOFCHAR){
            syntaxError();
        }

        if (lexer.ttype==S_RPAREN) lexer.nextToken();
        else syntaxError();

        if (lexer.ttype==S_SEMICOLON) lexer.nextToken();
        else syntaxError();

    switch (ktest) {
        case T_INT:
```

```
    case T_ARRAYOFINT:
                iseg.appendCode (OUTPUT);
                break;
          case T_CHAR:
                iseg.appendCode (OUTPUTC);
                break;
          case T_DOUBLE:
                iseg.appendCode (OUTPUTD);
                break;
          case T_BOOL:
                iseg.appendCode (OUTPUTB);
                break;
          case T_STRING:
                iseg.appendCode (OUTPUTS);
                break;
          default:
                syntaxError ("Type mismatched");
                break;
          }
}


    //Writeint 文の作成
public static void parseWriteint_statement(){
          int ktest=-1;
          if (lexer.ttype==S_WRITEINT) lexer.nextToken();
          else syntaxError();

          if (lexer.ttype==S_LPAREN) lexer.nextToken();
          else syntaxError();

          ktest=parseExpression();
          if(ktest!=T_INT && ktest!=T_CHAR &&   ktest!= T_DOUBLE){
             syntaxError();
          }

          if (lexer.ttype==S_RPAREN) lexer.nextToken();
          else syntaxError();
```

```
                if (lexer.ttype==S_SEMICOLON) lexer.nextToken();
                else syntaxError();
                iseg.appendCode(OUTPUT);
}

      //Writechar 文の作成
public static void parseWritechar_statement(){
                int ktest=-1;
                if (lexer.ttype==S_WRITECHAR) lexer.nextToken();
                else syntaxError();

                if (lexer.ttype==S_LPAREN) lexer.nextToken();
                else syntaxError();

                ktest=parseExpression();
                if(ktest!=T_INT && ktest!=T_CHAR &&   ktest!= T_DOUBLE){
                     syntaxError();
                }

                if (lexer.ttype==S_RPAREN) lexer.nextToken();
                else syntaxError();



                if (lexer.ttype==S_SEMICOLON) lexer.nextToken();
                else syntaxError();
                iseg.appendCode(OUTPUTC);
}

      //Writedouble 文の作成
public static void parseWritedouble_statement(){
                int ktest=-1;
                if (lexer.ttype==S_WRITECHAR) lexer.nextToken();
                else syntaxError();

                if (lexer.ttype==S_LPAREN) lexer.nextToken();
                else syntaxError();

                ktest=parseExpression();
                if(ktest!=T_INT && ktest!=T_CHAR &&   ktest!= T_DOUBLE){
```

```
                syntaxError();
            }

            if (lexer.ttype==S_RPAREN) lexer.nextToken();
            else syntaxError();


            if (lexer.ttype==S_SEMICOLON) lexer.nextToken();
            else syntaxError();
            iseg.appendCode(OUTPUTD);
}

    public static int parseLefthand_side(){
            int ktest=-1;

            if (lexer.ttype==S_NAME){
                    if(vt.exist(lexer.name)==false){
                        syntaxError();
                    }
                    ktest = vt.getType(lexer.name);
                    iseg.appendCode(PUSHI,vt.getAddress(lexer.name));
                    lexer.nextToken();
            }else syntaxError();

            if (lexer.ttype==S_LBRACKET) {
                    parseLefthand_side2(ktest);
            }else {
                    if(ktest==T_ARRAYOFINT||ktest==T_ARRAYOFBOOL){
                            syntaxError();
                    }
            }
            return ktest;
}

public static void parseLefthand_side2(int ktest){
            if(lexer.ttype==S_LBRACKET){
                if(ktest==T_INT||ktest==T_BOOL){
                            syntaxError();
                    }
```

```
                lexer.nextToken();
            } else syntaxError();

            parseExpression();
            iseg.appendCode(ADD);
            if(lexer.ttype==S_RBRACKET){
                    lexer.nextToken();
            }
            else syntaxError();
}

    //Expression の作成
public static int parseExpression() {
            int ktest1=-1;
            int ktest2=-1;
            ktest1=parseLogical_term();

            while (lexer.ttype==S_OR){
                    if (lexer.ttype==S_OR) lexer.nextToken();
                    else syntaxError();

                    ktest2=parseLogical_term();
                    if((ktest1==T_INT || ktest1==T_ARRAYOFINT) ||
                        ( ktest2==T_INT || ktest2==T_ARRAYOFINT)){
                            syntaxError();
                        }
                    iseg.appendCode(OR);
            }
            return ktest1;
}

public static int parseLogical_term(){
            int ktest1=-1;
            int ktest2=-1;
            ktest1=parseLogical_factor();
            while (lexer.ttype==S_AND) {
                    lexer.nextToken();

                    ktest2=parseLogical_factor();
```

```
                if((ktest1==T_INT || ktest1==T_ARRAYOFINT) ||
                    ( ktest2==T_INT || ktest2==T_ARRAYOFINT)){
                                syntaxError();
                    }
                iseg.appendCode(AND);
        }
        return ktest1;
}

public static int parseLogical_factor(){

        int betype =-1;
        int pctr =0;
        int ktest1 = -1;
        int ktest2 = -1;
        ktest1=parseArithmetic_expression();
        betype=lexer.ttype;
        if(betype==S_EQUAL||betype==S_NOTEQ||betype==S_LESS||
            betype==S_LESSEQ||betype==S_GREAT||betype==S_GREATEQ){
            switch (betype){
                case S_EQUAL:
                lexer.nextToken();
                ktest2=parseArithmetic_expression();
                if((((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
                        ( ktest2==T_INT || ktest2==T_ARRAYOFINT))||
                        ((ktest1==T_BOOL || ktest1==T_ARRAYOFBOOL) &&
                        (ktest2==T_BOOL || ktest2==T_ARRAYOFBOOL)))==false){
                                syntaxError();
                        } else {
                                ktest1=T_BOOL;
                        }
                pctr = iseg.appendCode(COMP);
                iseg.appendCode(BEQ,pctr+4);
                break;

            case S_NOTEQ:
                lexer.nextToken();
                ktest2=parseArithmetic_expression();
                if((((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
```

```
                ( ktest2==T_INT || ktest2==T_ARRAYOFINT))||
                ((ktest1==T_BOOL || ktest1==T_ARRAYOFBOOL) &&
                (ktest2==T_BOOL || ktest2==T_ARRAYOFBOOL)))==false){
                        syntaxError();
                } else {
                        ktest1=T_BOOL;
                }
        pctr =iseg.appendCode(COMP);
        iseg.appendCode(BNE,pctr+4);
        break;

    case S_LESS:
        lexer.nextToken();
        ktest2=parseArithmetic_expression();
        if((((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
                ( ktest2==T_INT || ktest2==T_ARRAYOFINT))||
                ((ktest1==T_BOOL || ktest1==T_ARRAYOFBOOL) &&
                (ktest2==T_BOOL || ktest2==T_ARRAYOFBOOL)))==false) {
                        syntaxError();
                } else {
                        ktest1=T_BOOL;
                }
        pctr =iseg.appendCode(COMP);
        iseg.appendCode(BLT,pctr+4);
        break;

    case S_LESSEQ:
        lexer.nextToken();
        ktest2=parseArithmetic_expression();
        if((((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
                ( ktest2==T_INT || ktest2==T_ARRAYOFINT))||
                ((ktest1==T_BOOL || ktest1==T_ARRAYOFBOOL) &&
                (ktest2==T_BOOL || ktest2==T_ARRAYOFBOOL)))==false){
                        syntaxError();
                } else {
                        ktest1=T_BOOL;
                }
        pctr =iseg.appendCode(COMP);
        iseg.appendCode(BLE,pctr+4);
```

```
		break;

	case S_GREAT:
		lexer.nextToken();
		ktest2=parseArithmetic_expression();
		if((((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
				( ktest2==T_INT || ktest2==T_ARRAYOFINT))||
				((ktest1==T_BOOL || ktest1==T_ARRAYOFBOOL) &&
				(ktest2==T_BOOL || ktest2==T_ARRAYOFBOOL)))==false) {
					syntaxError();
				} else {
					ktest1=T_BOOL;
				}
		pctr =iseg.appendCode(COMP);
		iseg.appendCode(BGT,pctr+4);
		break;

	case S_GREATEQ:
		lexer.nextToken();
		ktest2=parseArithmetic_expression();
		if((((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
				( ktest2==T_INT || ktest2==T_ARRAYOFINT))||
				((ktest1==T_BOOL || ktest1==T_ARRAYOFBOOL) &&
				(ktest2==T_BOOL || ktest2==T_ARRAYOFBOOL)))==false){
					syntaxError();
				} else {
					ktest1=T_BOOL;
				}
		pctr =iseg.appendCode(COMP);
		iseg.appendCode(BGE,pctr+4);
		break;

	default:
		break;
	}
	iseg.appendCode(PUSHI, 0);
	iseg.appendCode(JUMP,pctr+5);
	iseg.appendCode(PUSHI, 1);
	}
```

```
    return ktest1;
}

public static int parseArithmetic_expression(){
            int betype=-1;
            int ktest1 = -1;
            int ktest2 = -1;
            ktest1=parseArithmetic_term();
            betype=lexer.ttype;
            while (lexer.ttype==S_ADD || lexer.ttype==S_SUB) {
                    switch (betype) {
                            case S_ADD:
                            lexer.nextToken();
                            ktest2=parseArithmetic_term();
                            if(((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
                                (ktest2==T_INT || ktest2==T_ARRAYOFINT))==false) {
                                    syntaxError();
                                }
                            iseg.appendCode(ADD);
                            break;

                        case S_SUB:
                            lexer.nextToken();
                            ktest2=parseArithmetic_term();
                            if(((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
                               ( ktest2==T_INT || ktest2==T_ARRAYOFINT))==false){
                                    syntaxError();
                                }
                            iseg.appendCode(SUB);
                            break;

                        default:
                            break;
                        }
                }
            return ktest1;
}

public static int parseArithmetic_term() {
```

```
int betype=-1;
int ktest1 = -1;
int ktest2 = -1;
ktest1=parseArithmetic_factor();
betype=lexer.ttype;
while (lexer.ttype==S_MUL || lexer.ttype==S_DIV || lexer.ttype==S_MOD)
    {
        switch (betype)
            {
            case S_MUL:
                lexer.nextToken();

                ktest2=parseArithmetic_factor();
                if(((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
                    ( ktest2==T_INT || ktest2==T_ARRAYOFINT))==false)
                     {
                         syntaxError();
                     }
                iseg.appendCode(MUL);
                break;

            case S_DIV:
                lexer.nextToken();


                ktest2=parseArithmetic_factor();
                if((ktest1!=T_INT && ktest1!=T_ARRAYOFINT) ||
                    ( ktest2!=T_INT && ktest2!=T_ARRAYOFINT))
                     {
                         syntaxError();
                     }
                iseg.appendCode(DIV);
                break;

            case S_MOD:
                lexer.nextToken();

                ktest2=parseArithmetic_factor();
                if(((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
```

```
                        (ktest2==T_INT || ktest2==T_ARRAYOFINT))==false) {
                            syntaxError();
                        }
                    iseg.appendCode(MOD);
                    break;

                default:
                    break;
                }
            }
        return ktest1;
}

public static int parseArithmetic_factor() {
            int ktest = -1;
            switch (lexer.ttype) {
                    case S_NOT:
                    lexer.nextToken();

                    ktest=parseArithmetic_factor();
                    iseg.appendCode(NOT);
                    break;

                case S_SUB:
                    lexer.nextToken();

                    ktest=parseArithmetic_factor();
                    iseg.appendCode(CSIGN);            //符号変換
                    break;

                default:
                    ktest=parseUnsigned_factor();
                }
            return ktest;

}

public static int parseUnsigned_factor(){
        String name = "";
```

```
int ktest = -1;

switch (lexer.ttype) {
case S_NAME:
            name = lexer.name;
            ktest=vt.getType(name);
            if(vt.exist(name)==false){
                syntaxError();
            }
            iseg.appendCode(PUSHI,vt.getAddress(name));
            lexer.nextToken();

                if(lexer.ttype==S_LBRACKET){
                    parseUnsigned_factor2(ktest);
                }
                iseg.appendCode(LOAD);
                break;

            case S_INTEGER:                          /* 定数 */
                ktest=T_INT;
                iseg.appendCode(PUSHI,lexer.value);
                lexer.nextToken();
                break;

            case S_CHARACTER:                        /* 文字型定数 */
                ktest=T_INT;
                iseg.appendCode(PUSHI,lexer.value);
                lexer.nextToken();
                break;

        case S_STR:                      /* 文字列 */
        ktest = T_STRING;
        iseg.appendCode (PUSHS, lexer.string);
                lexer.nextToken();
                break;

            case S_LPAREN:
                lexer.nextToken();
                ktest = parseExpression();
```

```
            if(lexer.ttype==S_RPAREN){
                lexer.nextToken();
            }else syntaxError();
            break;

        case S_READINT:                              /* readint */
            ktest = T_INT;
            iseg.appendCode(INPUT);
            lexer.nextToken();
            break;

        case S_READCHAR:                             /* readchar */
            ktest = T_INT;
            iseg.appendCode(INPUTC);
            lexer.nextToken();
            break;

        case S_TRUE:
            ktest = T_BOOL;
            iseg.appendCode(PUSHI,V_TRUE); //真を表す 1 が入る
            lexer.nextToken();
            break;



        case S_FALSE:
            ktest = T_BOOL;
            iseg.appendCode(PUSHI,V_FALSE); //偽を表す 0 が入る
            lexer.nextToken();
            break;

    case S_PROCESSOR:                                /* $p */
    ktest = T_INT;
    iseg.appendCode (PUSHP);
            lexer.nextToken();
            break;

        default:
            syntaxError();
            break;
```

```
		}
	return ktest;
}

public static void parseUnsigned_factor2(int ktest){
	if (lexer.ttype==S_LBRACKET) {
			if(ktest==T_INT||ktest==T_BOOL){
				syntaxError();
			}
			lexer.nextToken();
		} else syntaxError();
	parseExpression();

	if (lexer.ttype==S_RBRACKET){
			lexer.nextToken();
	} else syntaxError();

	iseg.appendCode(ADD);
}

// 文法エラー
public static void syntaxError() {
			System.out.println("Syntax error at line " + lexer.inFile.linenum);
			System.out.println(lexer.inFile.line);
			lexer.inFile.closeFile();
			System.exit(1);
}

	/* 文法エラーオーバーロード
	与えられた文字列を表示してエラー
	*/
	static void syntaxError(String err_mes) {
			System.out.println ("Systax error at line " + lexer.inFile.linenum);
			System.out.println (err_mes);
			System.out.println (lexer.inFile.line);
			lexer.inFile.closeFile();
			System.exit (1);
	}
}
```