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

03-1-47-248

池田　直樹

近畿大学理工学部情報学科

平成 19 年 2 月 5 日提出

概要

現在、多くの分野で膨大な量の計算・複雑な計算を高速で処理することが求められている。数年前に比べれば CPU の動作周波数が高速化し、高速な処理が可能になっているが、短時間では解けない問題が存在する。より高速に問題を解くためには、CPU をより高速で処理できるものにすればよい、しかし CPU の動作周波数の高速化には限界があるため、この方法では充分な高速化が見込めない。

このため、CPU の高速化以外の方法による高速化の手段として並列化が挙げられる。並列化は、複数の CPU を持つ並列計算機で処理を並列化することにより高速化を行うのである。その並列化を実現する為に用いられるのが並列アルゴリズムである。並列アルゴリズムとは、並列計算機で問題を解くためのアルゴリズムであり、多数の計算機を使用して高速な処理を行う際に用いられる。並列アルゴリズムを用いて並列化を行うことで計算時間の短縮・処理を行うサイズの拡大を可能にする。

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)で行われる。RAM は通信やコスト、個々の演算による実行時間の違いなどを無視した単純なものであるといったような並列処理機構が理想的に設計されているため、並列アルゴリズムの設計や評価を容易に行えるためである。しかし、実際に PRAM を実現して計算量の評価を行うことには様々な問題があり困難とされる。そこで本研究では、PRAM 上での計算量の評価を実験的に行う為のツールとして、PRAM シミュレータの一部である PRAM コンパイラの作成を行う。

目次

1　序論

1.1 本研究の背景

1.1.1並列処理

現在、地球環境のシミュレーション、天気予報で用いられる数値予報、量子力学に基づく分子設計、原始物理、宇宙物理のシミュレーション、流体の計算、経済・社会システムのシミュレーション、コンピュータグラフィックス、広域通信網、電子網等の最適設計、画像処理、大規模データベース、大規模知識ベースの検討などの様々な分野で、膨大な量の問題を高速かつ短時間で処理することが求められている。問題を高速で処理するには 2 通りの方法が挙げられる。1 つは使用するプロセッサの性能の向上である。近年ではプロセッサの発達が進み、処理速度が目覚しい向上を遂げた。しかし、これにも限界があり今以上の処理速度の著しい向上は期待できない。もう 1 つは、並列処理（Parallel Processing）を用いた処理の並列化である。並列処理とは、1 つの問題に対する処理を小さな処理に分割し、それを複数のプロセッサで並列に実行することをいう。近年、コンピュータハードウェアの値下がりによって複数のプロセッサを用いた並列計算機を構築することが可能になってきた。こういった傾向が進むにつれ並列処理による問題の処理への必要性が大幅に高くなっていくと思われる。

1.1.2並列計算機

並列計算機（Parall Computer）は、問題の計算にかかる処理時間の高速化をはかる為に複数のプロセッサを同時に動作させることができる計算機である。並列計算機には、共有メモリ型並列計算機（Shared Memory Parallel Computer）と分散メモリ型並列計算機（Distributed Memory Parallel Computer）がある。

図 1 および図 2 に共有メモリ型並列計算機と分散メモリ型並列計算機の概念図を示す。プロセッサ間の通信については、共有メモリ型並列計算機ではメモリを通して行われ、分散メモリ型並列計算機ではネットワークを通して行われる。

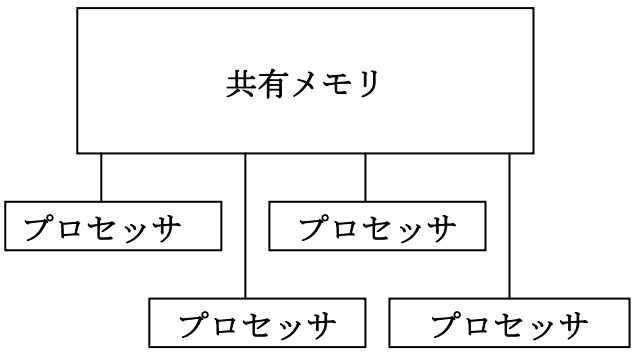

図 1 共有メモリ型並列計算機

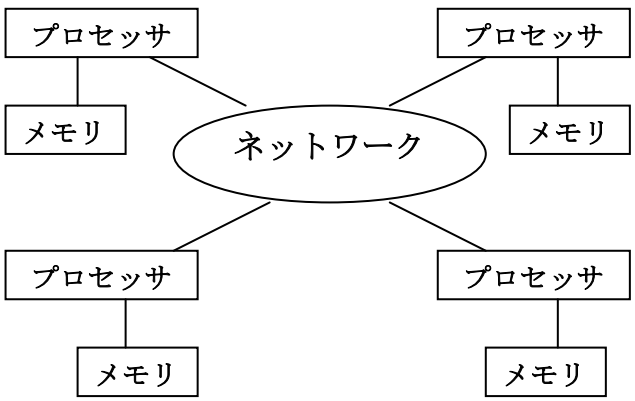

図 2　分散メモリ型並列計算機

1.1.3並列アルゴリズム

並列アルゴリズム（Parallel Algorithm）とは、複数のプロセッサを用いて並列に処理を行う計算方法をいう。任意の問題に対し、複数台のプロセッサを用いることでその台数分の時間短縮が期待できるかと言えばそうではない。複数のプロセッサを効率よく並列に利用することができなければ、うまく並列化することができず期待した時間短縮が実現されない。つまり、プロセッサ間のデータのやりとりや、メモリのアクセス、各プロセッサの同期など、並列特有の問題が生じる。そういった問題をなくす為に、逐次の処理で用いられる逐次アルゴリズムではなく効率よくプロセッサを並列に利用できる計算方法、つまりその問題に最適な並列アルゴリズムを用いる必要がある。

1.1.4 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。代数的な並列計算モデルとして PRAM（Parallel Random Access Machine)、Mesh、Hyper Cube、BSP(Bulk-Synchronous Parallel)モデル、CGM(Coase Grain Multi Computer)などがある。

1.1.5PRAM

並列アルゴリズムの研究では多くの場合並列計算モデルとして PRAM（Parallel Random Access Machine）[2]が用いられる。PRAM は共有メモリ型並列計算モデルであり、演算命令・メモリアクセス命令・出力命令などの命令が全て 1 単位時間で行われること、1 命令ごとに同期がとられること、通信コストが発生しないことなどの並列計算機構が理想的に設定されている。つまり、並列化を実現したときに関わる問題点を無視した計算モデルである為、並列アルゴリズムの設計を容易なものにし並列化に関しての評価を比較的容易なものにしている。しかし、並列計算機構理想的に設定されている為現実との差は大きく、実際に大規模なプロセッサでのメモリ共有化や、通信などの高速化などに問題がある。その為このような並列アルゴリズムを実現できる効率のよい並列計算モデル PRAM の実現は困難とされている。

PRAM は複数のプロセッサによる共有メモリへの自由な読み書きを行うことができる。共有メモリ上の同一のメモリアクセスに対する複数のプロセッサによる同時のアクセスの処理により、PRAM は以下の 3 種類に分類される。

① CRCW－PRAM（Concurrent-Read Concurrent-Write）
　複数のプロセッサによる同一メモリセルへの同時読み出し同時書き込みを認める。
② CREW－PRAM（Concurrent-Read Exclusive-Write）
　複数のプロセッサによる同一メモリセルへの同時読み出しは認めるが、書き込みは認めない。
③ EREW－PRAM（Exclusive-Read Exclusive-Write）
　メモリへのアクセスは排他的であり、どの２つのプロセッサも同一のメモリセルへの同時読み出しも同時書き込みも認めない。

さらに、同時書きの処理に関して次のような分類ができる。

（1） 共通型（Common）CRCW－PRAM

全てのプロセッサが同じ値を書くときのみ同時書きを認める。

（2） 任意型（Arbitrary）CRCW－PRAM

どれか１つのプロセッサが書きに成功する。

（3） 優先型（Priority）CRCW－PRAM

最も優先度の高いプロセッサが書きに成功する。

本研究では、優先型 CRCW－PRAM を対象とする。

1.1.4PRAM シミュレータ

1.1.3 で述べたとおり、PRAM を実現し計算量などの評価を行うことは困難である。そこで、計算量などを実験的に評価する為に PRAM シミュレータが必要とされる。PRAM シミュレータには、PRAM アルゴリズムを記述する為の高級並列言語と並列アセンブラ、高級並列言語を並列アセンブラに変換するコンパイラ、その変換した並列アセンブラを実行する PVSM インタプリタから成る。図 3 に PRAM シミュレータによる処理の流れを示す。これらを使用することによって、PRAM 用並列言語プログラムを PRAM 上で実行した時の実行結果と実行時間などを出力することができ、PRAM での計算量の実験的な評価を行うことができる。

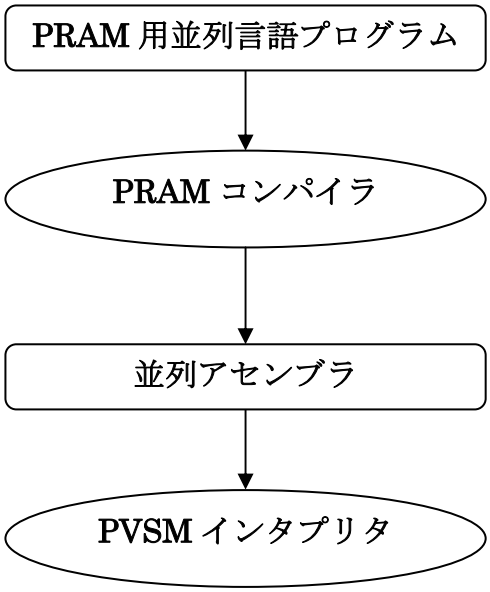

図 3　PRAM シミュレータによる処理の流れ

1.2 本研究の目的

本研究ではPRAM上での並列アルゴリズムの設計およびその計算量の実験的な評価を容易化するために 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。PRAM アルゴリズムの設計者は PRAM 用並列プログラムを作成し、本研究で設計した PRAM コンパイラで VSM アセンブラに変換し、変換した VSM アセンブラを PVSM で実行することにより計算量の実験的な評価を行うことができる。

1.3 本報告書の構成

以下に本報告書の構成を述べる。第 2 章では、本研究で設計した PRAM コンパイラについて述べる。第 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作成し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。第 4 章では、第 3 章での結果を踏まえて PRAM アルゴリズムおよび PRAM コンパイラの考察を行ない、今後の課題について考える。

2　研究内容

2.1PRAM 用並列言語

本研究では、高級並列言語として付録 1 に示す K05 言語を拡張した拡張 K05 言語を用いる。K05 言語は逐次用高級言語であるので、PRAM 用並列言語プログラムを記述するために、K05 言語に並列命令 parallel 文と特殊記号$p を加える。parallel 文の文法は以下の通りである。

parallel(式１,式２)　文

式１と式２は、int 型の式である。式１と式２には並列処理をする際に使用するプロセッサ番号がはいり、式１から式 2 のプロセッサを用いて以降に続く文を並列に実行する。文というのは parallel を含まない任意の文である。$p は、プロセッサ番号を表示させる特殊記号であり、$p を記述した場所における実行時のプロセッサ番号を表示する。付録 2 に拡張 K05 言語の文法を示す。

PRAM 用並列プログラムでは、始めは単一のプロセッサで逐次に処理が行われている。parallel 文を実行することで並列処理に切り替わり、parallel 文が終了すると再び逐次の処理に切り替わるといったようにプログラムが実行される。

2.2 並列用アセンブラ

本研究では、アセンブラに付録３に示す VSM アセンブラを拡張した拡張 VSM アセンブラを用いる。VSM アセンブラを並列に対応させるために、VSM アセンブラの命令セットに PARA・PUSHP・SYNC を加える。これらのアセンブラ命令は以下の意味を表す。

PARA：並列処理開始を開始する命令である。逐次状態で PARA を読むと、スタックから PRAM 用並列プログラムの parallel 文で指定したプロセッサの台数のデータを取り出し、それを用いて並列状態に以降し処理が実行される。並列状態にあるとき PARA を読むと実行時エラーになる。

PUSHP：プロセッサ番号を挿入する命令である。この命令を読むと、並列状態で処理を実行している各プロセッサがプロセッサ番号をスタックトップに積む。

SYNC：同期をとる命令である。PARA により各プロセッサが並列処理を行っている時に SYNC 命令を読むと、処理中の各プロセッサが SYNC を読むまで動作を停止し、全てのプロセッサが SYNC 命令を読むと再び動作を開始する。並列処理中のプロセッサはこうして同期をとり逐次状態へ移行する。

2.3PRAM コンパイラ

本研究では Java を用いて PRAM コンパイラプログラムを作成した。本研究で設計した PRAM コンパイラを使用することによって、拡張 K05 言語から拡張 VSM アセンブラを生成することができる。

2.3.1PRAM コンパイラの構成

付録５に本研究で作成した PRAM コンパイラプログラムを示す。

今回 PRAM コンパイラを設計するにあたり、主にコンパイラを拡張したところを以下に示す。

① Pram.java

ここでは、構文解析を行う。字句解析部より字句の列を読み、それらを文法と照らし合わせて構文構造にまとめられる。今回作成した PRAM 用並列言語の命令などは、ここで構文構造をまとめられ命令に対するアセンブラコードを組み立てていく。

② Operators.java

今回、PRAM コンパイラの為に用意した PUSHP などのオペレータを VSM アセンブラ追加のどの時にそのまま用いる為に、各オペレータの対応付けを行っている。

③ Symbol.java

ここでは、字句解析プログラムが構文解析にトークンを渡す際にもちいるものであり、切り出したトークンを int 型のフィールドに格納する時、その値とトークンの対応づけを行う。

④ LexicalAnalyzer.java

ここでは、字句解析を行う。プログラムで書かれた命令などを読み、一字以上の文字を受け取り、それらを意味のある単語（Token）に変換する。さらにその単語を Pram に渡す。

2.3.2PRAM コンパイラの仕様

本研究で設計した PRAM コンパイラの仕様を以下に示す。

[ファイル名].k を PRAM 用並列アルゴリズムを使用し記述したプログラムとする。以下のコマンドを実行すると、[ファイル名].k が並列アセンブラに変換され、outputfile に出力される。foo.asm を省略した場合は、OpCode.asm に出力される。

>java Pram [ファイル名].k [foo.asm]

foo.asm を PRAM コンパイラにより作成された拡張された VSM アセンブラとする。このとき、以下のコマンドにより foo.asm を実行できる。

>java VSM　[-c][foo.asm]

ファイル名 foo.asm を省略した場合は、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 PRAM 上で拡張 K05 言語を実行させたときの計算時間も出力される。

3　検証

本章では本研究で設計した PRAM コンパイラの検証を行う。PRAM コンパイラの正当性を検証する為、PRAM 用並列言語を含むプログラムを作成して PRAM コンパイラでのコンパイルを実行した。作成したプログラムは図 4 に示す。

```
main(){

parallel(0,15){

        write($p);
        }
}
```

図 4　拡張 K05 言語プログラム

図 4 のプログラムでは parallel 文を使い、0~15 番までの 16 台のプロセッサで処理を実行する。そして、$p を使い並列で実行している時のプロセッサ番号を認識させ、write により出力させている。このプログラムをコンパイルすることにより図 5 に示す VSM アセンブラが生成される。

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

図 5　拡張 VSM アセンブラ

さらに、図 5 拡張 VSM アセンブラを PVSM で実行すると、図 6 に示す実行結果が出力される。

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

図 6　実行結果

この結果から、本研究で作成した PRAM コンパイラが正常に動作し、並列による処理が

実行されていることがわかる。また、図 4 の拡張 K05 言語プログラムを PRAM 上で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

本研究で設計した PRAM コンパイラを用いることで、プログラムを並列に実行することができ、並列アルゴリズムの計算量評価を実験的に行うことができる。しかし、コンパイラとしての機能面で見ると十分なものとは言い難い。parallel 文に関しては、文法の制限が多く、使いやすさにかけているところがある。$p に関しては、用いるときに書き方の制限が多く、これも使い易いとは言い切れない。

今後の課題としては、並列処理を行うプロセッサ数の宣言時に単一のプロセッサ番号を指定できる機能や、parallel 文中に parallel 文を用いられるような機能などを付け加えるといった機能の拡張を行うことが挙げられる。

5　謝辞

本研究をすすめるにあたり、石水隆助手には並列アルゴリズムや並列処理について多くのご指導ご鞭撻をいただき、また同研究室の皆にもいろいろとお世話になり深く感謝申し上げます。

6　参考文献

[1]　平成 17 年度情報・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 Wesle

付録

付録 1 K05 言語の文法

本研究で用いた K05 言語の文法を以下に示す。

<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 “[” <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 | $p | CHAR
| “(“ <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

本研究で作成した PRAM コンパイラのプログラムを以下に示す。なお、本研究で作成したプログラムは以下の構成からなる。

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

/* Pram.java */

```
public class Pram implements Operators, Symbol, Type, TruthValue {
    static LexicalAnalyzer lexer;
    static VarTable vt;
    static InstractionSegment iseg;

    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();
        parseProgram();

        lexer.inFile.closeFile();
```

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

    }

    public static void parseProgram(){
        parseMain_function();
        iseg.appendCode(HALT);
    }

    public static void parseMain_function() {
        if (lexer.ttype==S_MAIN) lexer.nextToken();
        else syntaxError();

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

        parseBlock();

    }

    public static void parseBlock(){
        if (lexer.ttype==S_LBRACE) lexer.nextToken();
        else syntaxError();

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

        while (lexer.ttype!=S_RBRACE)

            {
                parseStatement();
            }
```

```
        lexer.nextToken();

}

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();
        }


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();
                }
```

```
                if(vt.addElement(type,name,size)==false){
                    syntaxError();
                }
            }

        if (lexer.ttype==S_SEMICOLON) lexer.nextToken();
        else syntaxError();
}
public static void parseStatement(){
        switch (lexer.ttype)
            {
            case S_IF:
                parseIf_statement();
                break;

            case S_WHILE:
                parseWhile_statement();
                break;

            case S_FOR:
            parseFor_statement();
                break;

            case S_PARALLEL:
            parsePara_statement();
            break;

            case S_WRITE:
            parseWrite_statement();
            break;

            case S_NAME:
                parseAssignment();
                break;

            case S_WRITEINT:
```

```
                parseWriteint_statement();
                break;

            case S_WRITECHAR:
                parseWritechar_statement();
                break;

            case S_LBRACE:
                lexer.nextToken();
                while (lexer.ttype!=S_RBRACE)
                    {
                        parseStatement();
                    }

                if (lexer.ttype==S_RBRACE) lexer.nextToken();
                else syntaxError();
                break;

            case S_SEMICOLON:
                lexer.nextToken();
                break;

            default:
                syntaxError();
                break;
            }
}

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);

}

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

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

                parseAssignment2();

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

                int ad1 = iseg.isegPtr;

                ktest=parseExpression();
                if(ktest==T_INT || ktest==T_ARRAYOFINT){
                        syntaxError();
                }
                int ad3 = iseg.isegPtr;
                iseg.appendCode(BEQ);
                iseg.appendCode(JUMP);
                if(lexer.ttype==S_SEMICOLON)lexer.nextToken();
```

```
        else syntaxError();

        int ad2 = iseg.isegPtr;
        parseAssignment3();

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

        iseg.appendCode(JUMP,ad1);
        int ad4 = iseg.isegPtr;
        parseStatement();
        iseg.appendCode(JUMP,ad2);
        iseg.replaceCode(ad3,iseg.isegPtr);
        iseg.replaceCode(ad3+1,ad4);
}

public static void parsePara_statement(){
        int ktest=-1;
        if(lexer.ttype==S_PARALLEL) lexer.nextToken();
        else {
                syntaxError();

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

        ktest=parseExpression();
        if(ktest==T_BOOL || ktest==T_ARRAYOFBOOL)
        {
                syntaxError();
        }

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

        ktest=parseExpression();
        if(ktest==T_BOOL || ktest==T_ARRAYOFBOOL){
                        syntaxError();
```

```
            }

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

            iseg.appendCode(PARA);
            parseStatement();
            iseg.appendCode(SYNC);

    }


    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

}
```

```
public static void parseAssignment(){
    int ktest1=-1;
    int ktest2=-1;
    ktest1=parseLefthand_side();
    if (lexer.ttype==S_ASSIGN){ lexer.nextToken();

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

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

    else syntaxError();
}

    public static void parseAssignment2(){
    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();
             }
        iseg.appendCode(ASSGN);
        iseg.appendCode(REMOVE);
}

        public static void parseAssignment3(){
        int ktest1=-1;
        int ktest2=-1;

        ktest1=parseLefthand_side();
        if (lexer.ttype==S_INC) lexer.nextToken();
        else syntaxError();

        iseg.appendCode(POSTINC);
                    iseg.appendCode(REMOVE);
}

        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_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:
        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 ();
        break;
                }
```

```
    }

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_BOOL||ktest==T_ARRAYOFBOOL){
        syntaxError();
    }
    if (lexer.ttype==S_RPAREN) lexer.nextToken();
    else syntaxError();

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

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_BOOL||ktest==T_ARRAYOFBOOL){
        syntaxError();
    }

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

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

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();
}

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:
                ktest=T_INT;
                iseg.appendCode(PUSHI,lexer.value);
                lexer.nextToken();
                break;

            case S_CHARACTER:
```

```
        ktest=T_INT;
        iseg.appendCode(PUSHI,lexer.value);
        lexer.nextToken();
        break;

case S_PROCESSOR:
ktest=T_INT;
        iseg.appendCode(PUSHP,lexer.value);
        lexer.nextToken();
break;

case S_LPAREN:
        lexer.nextToken();

        ktest = parseExpression();

        if(lexer.ttype==S_RPAREN){

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

        break;

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

case S_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;

            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);

}
```

```
    public static void syntaxError() {
        System.out.println(lexer.ttype);
        System.out.println("Syntax error at line " + lexer.inFile.linenum);
        System.out.println(lexer.inFile.line);
        lexer.inFile.closeFile();
        System.exit(1);
    }
}

/* InputFilejava */

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);
    }
}

/* 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
                }
        }
}

/* InstractionSegment */

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);
        }
}

/* 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 */
    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;
}


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;
        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 '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 '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("writeint"))
                    ttype = S_WRITEINT;                                 /* writeint */
                else if (str.equals("writechar"))
                    ttype = S_WRITECHAR;                                                    /*
writechar */
                else if (str.equals("write"))
                        ttype = S_WRITE;
                else ttype = S_NAME;                                    /*  変数名  */
                break;
            case 'p':
                if(str.equals("parallel")) ttype = S_PARALLEL; /*parallel*/
                else ttype = S_NAME;
                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 '+':
            if(inFile.nextc == '+'){
                    inFile.nextChar();
                    ttype = S_INC;
        }else       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 '$':
            if(inFile.nextc == 'p'){
                inFile.nextChar();
                ttype = S_PROCESSOR;
            } else syntaxError();
            break;
            case '¥':                                                   /* 文字 */
                c = inFile.nextChar();
                if (c != '¥' && inFile.nextc == '¥') {
                    inFile.nextChar();
                    ttype = S_CHARACTER;
                    value = (int) c;
                } 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);
    }
}

/* 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
}

/* 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 */
}

/* 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;
}

/* 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;
        }
}

/* 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])
                               );
        }
    }
}

/* TruthValue */

interface TruthValue{
    static final int V_TRUE = 1;
```

```
    static final int V_FALSE = 0;
}
```