卒業研究報告書

題目

Java による PRAM コンパイラの拡張

指 導 教 員

石水 隆 講師

報告者

05-1-037-0138

藤枝正樹

近畿大学理工学部情報学科

平成 21 年 1 月 31 日提出

概要

現在、様々な分野において、膨大な量の計算や複雑な計算を短時間で処理することが求められている。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、あるいはアルゴリズムの最適化等が考えられるが、それぞれに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、ある問題を解く際に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。すなわち、プログラム全体をいくつかの部分に分割し、それぞれ別のプロセッサで実行する。このような計算のプログラムを書くことは並列プログラミングと言われ、そのための計算プラットフォームである並列コンピュータは、複数のプロセッサあるいは複数の独立したコンピュータを何らかの方法で結合して特別に設計することも可能である。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、実際に複数のプロセッサを同時に動かすことは非常に困難であり、複数のプロセッサにより並列処理を行うためのアルゴリズムである並列アルゴリズムの正当性および時間計算量を実験的に評価するのは容易ではない。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である PRAM コンパイラおよび PRAM インタプリタを作成する。

目次

1　序論

1.1 本研究の背景

1.1.1並列処理

現在、様々な分野で膨大な量の計算や複雑な計算を短時間で処理する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１万倍程度しか速くすることができない。また、アルゴリズムの最適化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は、そのような限界は本質的には存在しない。

並列処理(Parallel Processing)とは、ある問題を解く際に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、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、問題の計算にかかる処理時間の高速化を図るために、複数のプロセッサを用いて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。

以下に、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。

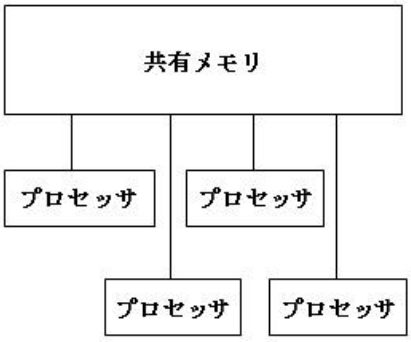

図 1：共有メモリ型並列計算機

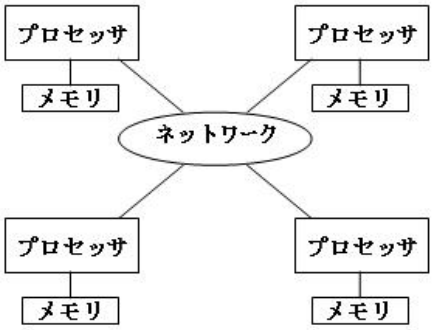

図 2：分散メモリ型並列計算機

1.1.3並列アルゴリズム

アルゴリズム（Algorithm）とは、問題を解くための手段を定めたものである。この手順は曖昧な点の残らないようきちんと定めたものでなければならない。手順を明確に定めてあれば、計算機をその手順どおり動かして問題を解かせることができる。アルゴリズムという概念は計算機とは無関係に成立するが、普通は計算機に問題を解かせるための手順である。

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。並列アルゴリズムは、どのようなデータを分割し、それをどのプロセッサに割り当てるか記述しなければならない。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く、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および解析は、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)[2]、Mesh[2]、HyperCube[2]、BSP(Bulk Synchronous Parallel)[3]などがある。

1.1.5 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 上で行われる。

PRAM は複数のプロセッサがメモリを共有するメモリ型並列計算モデルである。PRAM は代表的な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、全ての命令がその種類に関係なく、1 単位時間で行われる、1 命令毎に同期が取られる、通信のコストが一切発生しない、などの仮定が設けられた理想的なモデルである。PRAM は個々の演算による実行時間の違いや通信、同期のコストを無視した単純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ことができる。また、複数のプロセッサによる異なる位置のメモリセルへのアクセスに対しては全てのプロセッサが自由に読み書きを行なうことができる。一方、複数のプロセッサによる同一セルへのアクセスについてはそれをどう処理するかにより PRAM は以下の 4 つに分類される。

① 排他読み出し排他書き込み(EREW, Exclusive-Read Exclusive-Write)PRAM
メモリ位置へアクセスは排他的である。どの 2 つのプロセッサも同じメモリ位置から同時に読み出したり書き込んだりできない。

② 同時読み出し排他書き込み(CREW, Concurrent-Read Exclusive-Write)PRAM
複数のプロセッサが同時に同じメモリ位置から読み出すことができるが、書き込みの権利は排他的であり、どの 2 つのプロセッサも同じメモリ位置に同時に書き込むことはできない。

③ 排他読み出し同時書き込み(ERCW, Exclusive-Read Concurrent-Write)PRAM
複数のプロセッサが同時に同じメモリ位置に書き込むことができるが、読み出しは排他的である。

④ 同時読み込み同時書き込み(CRCW, Concurrent-Read Concurrent-Write)PRAM
複数のプロセッサによる同じメモリ位置への同時読み出し、同時書き込みが認められている。

また、CRCW-PRAM は同時書き込みが行われる際の処理方法で以下の 3 種に分類される。

A) 優先型(Priority)CRCWPRAM
同時書き込みが起こった時、最も優先順位が高いものが書き込みに成功する。

B) 任意型(Arbitrary) CRCWPRAM
同時書き込みが起こった時、どれか一つが書き込みに成功する。

C) 共通型(Common) CRCWPRAM
同時書き込みが起こった時、全てが同じ値書き込もうとした時に成功し、その他はエラーとする。

大規模なプロセッサでのメモリの共有化や、通信や同期の高速化には様々な問題が生じる。そのため、PRAM 自体の実現は非常に困難である。

1.1.6PRAM 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 要素から構成される。

① PRAM 用並列言語
並列命令に対応した高級言語。

② 並列アセンブラ
並列命令に対応したアセンブリコード。

③ PRAM コンパイラ
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 PVSM（Parallel Virtual Stack Machine）
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。

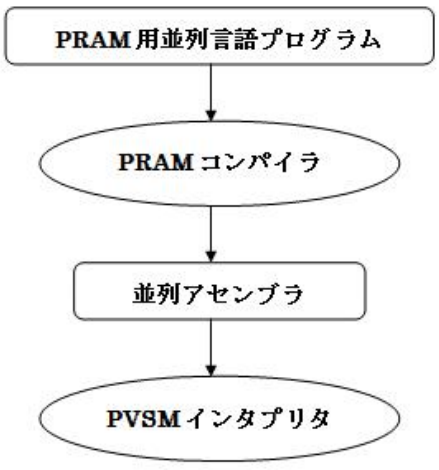

図 3：PRAM シミュレータによる処理の流れ

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 コンパイラを用いて PRAM 用並列言語プログラムを並列アセンブラプログラムに変換し、PVSM インタプリタ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。

1.1.7コンパイラとインタプリタ

コンパイラとは、プログラミング言語（高級言語）で書かれたプログラムをコンピュータが直接実行可能な形式（機械語のプログラム）に変換する処理を行うソフトウェアである。

インタプリタとは、プログラミング言語で書かれたソースコードを逐次解釈しながら実行するソフトウェアである。

1.2 本研究の目的

並列アルゴリズムは正当性を満たさなければならない。正当性とは、アルゴリズムが有限の時間内に正しい解を出力することである。正当性の理論的な検証は考えられる全てのデータに対して正しいことを保証し、その計算量も検証する必要がある。並列アルゴリズムの設計およびその計算量の解析は多くの場合 PRAM 上で行われるが、それを実現するには非常に困難である。そのため、並列アルゴリズムの正当性を実験的に証明することや、計算量の評価を実験的に行うことは難しい。そこで本研究では、Java 言語を用いて並列アルゴリズムの設計およびその計算量の解析の容易化を支援するために PRAM シミュレータの一部である PRAM コンパイラおよび PVSM インタプリタを設計する。

1.3 本報告書の構成

本報告書の構成を以下に述べる。

第 2 章では本研究で設計した PRAM コンパイラおよび PVSM インタプリタについて述べる。第 3 章では PRAM コンパイラおよび PVSM インタプリタの検証をする。ここではプログラムを実行することにより、本研究で設計したシミュレータの正当性を検証する。第 4 章では第 3 章で得られた実行結果を踏まえた上で、結論と今後の課題について考察する。

2　研究内容

2.1 PRAM 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 K07 言語を拡張した拡張 K07 言語を用いる｡ここで言う拡張 K07 言語とは、K07 言語が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 K07 言語に並列処理命令の parallel 文および実行中のプロセッサ番号を表示させる特殊記号$p を加えたものである。

parallel 文の文法は以下の通りである。

parallel(式①,式②)文

ここでの式①、式②とは int 型の評価値を持つ式であり、並列処理をする際に使用するプロセッサ番号が入る。また､ここでの文とは任意の文（ただし､parallel 文中に parallel 文は 2 重入れ子構造までを記述することができ、3 重以上の入れ子構造は認めないものとする）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以降に続く文の並列処理を行う命令である。

また、$p は実行中のプロセッサ番号を表示させる特殊記号であり、文中に特殊記号$p を記述すると$p は実行中のプロセッサ番号の値を持つ変数として処理され、parallel 文の 2 重入れ子構造で使用した場合、$p にはプロセッサ番号は 2 桁の数として表される。

PRAM 用並列言語プログラムでは、2 つの状態が存在する。プロセッサ 1 台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である。プログラムを開始した際は逐次状態であり、parallel 文を読み並列命令中の文は parallel 文が終了するまで並列状態として実行される。

付録 2 に本研究で作成した拡張 K07 言語の文法を示す。

2.2 並列用アセンブラ

本研究では、アセンブラとして付録 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。ここで言う拡張 VSM アセンブラとは、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以下の 3 つの命令を加えたものである。

① PARA
並列処理開始を開始する命令である｡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。

② 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。

③ 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、命令を実行しているプロセッサのプロセッサ番号をスタックトップに積む｡本研究で作成した VSM は parallel 文の 2 重入れ子構造に対応しているめ、プロセッサ番号は 2 桁の数として表される。

2.3 PRAM コンパイラ

本研究では、並列アルゴリズムの設計およびその計算量の解析の容易化を支援するために、PRAM シミュレータの一部として Java 言語を用いて 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、高水準言語である拡張 K07 言語から低級言語である拡張 VSM アセンブラを生成することができる。また、本研究で作成したコンパイラは PRAM の各分類に対応できるものであり、ユーザは各分類毎の PRAM での計算時間を求めることができる。

2.3.1PRAM コンパイラの構成

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。

2.3.2PRAM コンパイラの仕様

PRAM 用並列アルゴリズムを用いて記述したプログラムを[ファイル名].k とする。PRAM コンパイラを実行すると、[ファイル名].k が並列アセンブラに変換され、実行中に指定したファイルに出力される。なお、出力ファイルを指定しなかった場合、OpCode.asm に出力される。

2.4 PVSM インタプリタ

本研究では、作成した PRAM コンパイラに対応する PVSM インタプリタを拡張した。本研究で拡張した PVSM インタプリタを使用することによって、PRAM コンパイラを実行することにより出力された拡張 VSM アセンブラを実行することができる。

2.4.1PVSM インタプリタの構成

以下に本研究で拡張した PVSM インタプリタの構成を示す。

① VSM（Virtual Stack Machine）
本研究で使用した VSM は、parallel 文の 2 重入れ子構造に対応できるように、VSM i,j (0 ≦ i, j ＜10) のように 2 次元配列で表される。

② プロセッサ番号

parallel 文中で parallel 文が使用された場合、parallel 文によって与えられたプロセッサ番号を計算することにより、プロセッサ番号を図 4 に示すような 2 桁の 10 進数で表示する。

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

図 4：プロセッサ番号一覧

付録 5 に本研究で作成した PVSM インタプリタプログラムを示す。

2.4.2PVSM インタプリタの仕様

OpCode.asm を PRAM コンパイラにより作成された拡張 VSM アセンブラとする。PVSM インタプリタを実行すると、実行結果が表示される。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 K07 言語を実行させたときの計算時間および使用された総プロセッサ数が出力される。

また、本研究で作成した PVSM インタプリタは 1.1.5 で述べた PRAM の各分類に対応できるようにプログラム中にスイッチを設定してあり、このスイッチを変更することにより PRAM の分類を切り替えることができる。

3　実行結果および検証

本研究で設計した PRAM コンパイラおよび PVSM インタプリタの検証を行う。PRAM コンパイラおよび PVSM インタプリタ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、PRAM コンパイラおよび PVSM インタプリタを実行した。作成したプログラム例を図 5 に示す。

```
main(){
    parallel(0,5){
        parallel(0,3){
            writeint($p);
        }
    }
}
```

図 5：拡張 K07 言語プログラムの例

図 5 のプログラムでは、2 段階の parallel 文により、プロセッサ番号 0 番から 5 番の 6 台のプロセッサが、更にプロセッサ番号 0 番から 3 番の 4 台のプロセッサを用いて処理を実行する。また、$p を用いて並列で実行しているときのプロセッサ番号を認識させ、writeint により出力させている。このプログラムを、本研究で作成した PRAM コンパイラでコンパイルすることにより、図 6 に示す VSM アセンブラが生成される。

```
PUSHI   0
PUSHI   5
PARA
PUSHI   0
PUSHI   3
PARA
PUSHP
OUTPUT
SYNC
SYNC
HALT
```

図 6： PRAM コンパイラを実行することにより生成された VSM アセンブラ

図 6 の VSM アセンブラプログラムを本研究で作成した PVSM インタプリタで実行することにより、図 7 に示す実行結果が得られる。

```
0 1 2 3 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43 50 51 52 53
Execution time      : 11
Sequential time     : 4
Parallel time       : 7
Number of procesosrs: 24
```

図 7：PVSM インタプリタの実行結果

図 7 の実行結果より、実行にかかるステップ数が 11 であると計測され、図 5 の拡張 K07 言語プログラムを PRAM 上で実行させたときの実行時間が 11 であることが示される。また、総プロセッサ数が 24 であると計測され、使用されたプロセッサ数が 24 台であることが示される。

以上の結果から、本研究で作成した PRAM コンパイラおよび PVSM インタプリタによってプログラム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ことが確認できた。

4　結論・今後の課題

本研究では、JAVA 言語を用いて並列アルゴリズムの設計およびその計算量の解析の容易化を支援するための PRAM シミュレータの一部である PRAM コンパイラおよび PVSM インタプリタを作成した。本研究で作成した PRAM コンパイラおよび PVSM インタプリタを用いることによって、PRAM アルゴリズムの実行にかかる時間や使用されたプロセッサ数を計測できる。つまり、PRAM シミュレータとしての最低限の能力を備えたシミュレータを実現することができる。

本研究で作成した PRAM コンパイラは parallel 命令を読んだ際に各々のプロセッサを各自並列化させることを可能にし、parallel 命令中に parallel 命令を実行できる 2 重入れ子構造の対応を実現させた。しかしこれは 2 段階までの parallel 命令にしか対応しておらず、今後の課題としては 3 段階以上の parallel 命令を可能にするよう機能を拡張することが考えられる。そのような拡張を行うことにより、さらにその計算量の解析の容易化を支援できる。また、プログラム実行中の VSM の様子をグラフィカルに表示させるように拡張することなども考えられる。

謝辞

本研究を進めるにあたり、石水隆先生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頂き、深く感謝の意を申し上げます。本当にありがとうございました。

参考文献

[1] “情報・コンピュータシステムプロジェクトⅠ指導書”、近畿大学理工学部情報学科、(平成 19 年)

[2] Joseph JáJá：“An Introduction to Parallel Algorithms”、Addison-Wesley(1992)

[3] L.G.Valiant：“A Bridging Model for Parallel Computation”、Comm.of the ACM(1990)

[4] “Java による PRAM コンパイラの作成”、近畿大学理工学部情報学科卒業論文、(平成 19 年)

付録

1. K07 言語の文法

以下に本研究で用いた K07 言語の文法を示す。

<Program>::= <Main_function>
<Main_function>::= ”main” “(“ “)” <Block>
<Block>::= “{“ {<Var_decl>} {<Statement>} “}”

<Var_decl>::= “int” ((NAME [“=” [“-“] INT]) | (NAME “[“ INT “]”))
{“,” ((NAME [“=” [“-“] INT]) | (NAME “[“ INT “]”))} “;”

<Statement>::= <If_statement> | <While_statement> | <Exp_statement>
| <Writechar_statement> | <Writeint_statement>
| “{“ {<Statement>} “}” | “;”

<If_statement>::= “if” “(“ <Expression> “)” <Statement>
<While_statement>::= “while” “(“ <Expression> “)” <Statement>
<Exp_statement>::= <Expression> “;”
<Writechar_statement>::= “writechar” “(“ <Expression> “)” “;”
<Writeint_statement>::= “writeint” “(“ <Expression> “)” “;”

<Expression>::= <Exp> [(“=” | “+=” | “*=” | “-=” | “/=” | “%=”) <Expression>]
<Exp>::= <Logical_term> {“||” <Logical_term>}
<Logical_term>::= <Logical_factor> {“&&” <Logical_factor>}
<Logical_factor>::= <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 [“[“ <Expression> “]”]
| (“--“ | “++”) NAME
| NAME (“--” | “++”)
| INT | CHAR
| “(“ <Expression> “)”
| “readchar” | “readint”

2. 拡張 K07 言語の文法

以下に本研究で用いた拡張 K07 言語の文法を示す。

<Program>::= <Main_function>
<Main_function>::= ”main” “(“ “)” <Block>
<Block>::= “{“ {<Var_decl>} {<Statement>} “}”

<Var_decl>::= “int” ((NAME [“=” [“-“] INT]) | (NAME “[“ INT “]”))
{“,” ((NAME [“=” [“-“] INT]) | (NAME “[“ INT “]”))} “;”

<Statement>::= <Parallel_statement> | <If_statement> | <While_statement> | <Exp_statement>
| <Writechar_statement> | <Writeint_statement>
| “{“ {<Statement>} “}” | “;”

<Parallel_statement>::= “parallel” “(“ <Expression> “,” <Expression> “)” <Statement>
<If_statement>::= “if” “(“ <Expression> “)” <Statement>
<While_statement>::= “while” “(“ <Expression> “)” <Statement>
<Exp_statement>::= <Expression> “;”
<Writechar_statement>::= “writechar” “(“ <Expression> “)” “;”
<Writeint_statement>::= “writeint” “(“ <Expression> “)” “;”

<Expression>::= <Exp> [(“=” | “+=” | “*=” | “-=” | “/=” | “%=”) <Expression>]
<Exp>::= <Logical_term> {“||” <Logical_term>}
<Logical_term>::= <Logical_factor> {“&&” <Logical_factor>}
<Logical_factor>::= <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>::= “$p” | NAME [“[“ <Expression> “]”]
| (“--“ | “++”) NAME
| NAME (“--” | “++”)
| INT | CHAR
| “(“ <Expression> “)”
| “readchar” | “readint”

3. VSM アセンブラの文法

以下に本研究で用いた VSM アセンブラの文法を示す。

```
BINOP: #define BINOP(OP) {Stack[SP-1] = Stack[SP-1] OP Stack[SP]; SP--;}
ASSGN:
        addr = Stack[--SP];
        Dseg[addr] = Stack[SP] = Stack[SP+1]
ADD: BINOP(+);
SUB: BINOP(-);
MUL: BINOP(*);
DIV:
        if (Stack[SP] == 0)
         {
                printf("Zero divider detected¥n");
                return -2;
         }
        BINOP(/);
MOD:
        if (Stack[SP] == 0)
         {
                printf("Zero divider detected¥n");
                retrun -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]];

4. PRAM コンパイラプログラム

以下に本研究で用いた PRAM コンパイラプログラムを示す。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）　FileScanner.java
（２）　Instruction.java
（３）　LexicalAnalyzer.java
（４）　Operator.java
（５）　Pram.java
（６）　PseudoIseg.java
（７）　Symbol.java
（８）　Token.java
（９）　Type.java
（１０）　Var.java
（１１）　VarTable.java

（１）　FileScanner.java

```
package pram;
import java.util.*;
import java.io.*;

class FileScanner {

        Scanner sourceFile;
        String line;
```

```
int lineNumber, columnNumber;
char currentCharacter, nextCharacter;

public FileScanner(String filename){
	try {
	sourceFile = new Scanner(new File(filename));
	lineNumber = 0;
	columnNumber = 0;
	nextCharacter = ' ';
	readInputFile();
	nextChar();
	}catch(Exception e){
		System.out.print(e);
		System.exit(1);
	}
}

public void closeFile(){
	this.sourceFile.close();
}

public void readInputFile(){
	try{
		line = this.sourceFile.nextLine();
	}catch(Exception e){
		closeFile();
		System.exit(1);
	}
}

public String currentLine(){
	return line;
}

public char lookAhead(){
	return nextCharacter;
}

public int currentLineNumber(){
```

```
            return lineNumber;
        }

        public char nextChar(){
            currentCharacter = nextCharacter;
            if(columnNumber < line.length()){
                nextCharacter = line.charAt(columnNumber++);
            }
            else if(sourceFile.hasNextLine() == false){
                        nextCharacter = '¥0';
            }
            else if(columnNumber >= line.length()){
                        readInputFile();
                        nextCharacter = '¥n';
                        lineNumber++;
                        columnNumber = 0;
            }
            return currentCharacter;
        }

}
```

（2） Instruction.java

```
package pram;

class Instruction
{
   String op;  //オペレータ
   int reg ;   //アドレス修飾用
   int addr;   //オペランド

   /** Operatorクラスの要素から命令オブジェクトを作るためのコンストラクタ
       @param opcode 文字列形式の命令
       @param flag その命令のアドレス修飾子
       @param address オペランド  */
   Instruction(Operator opcode, int flag, int address)
   {
      op = opcode.name();
      reg = flag;
```

```
        addr = address;
    }

    /** 文字列から命令オブジェクトを作るためのコンストラクタ
        @param opcode 文字列形式の命令
        @param flag その命令のアドレス修飾子
        @param address オペランド */
    Instruction(String opcode, int flag, int address)
    {
        op = opcode;
        reg = flag;
        addr = address;
    }

    //命令を出力するためのメソッド
    String printInstruction()
    {
        /* 命令を出力する際、命令のop部がop_oprndOuts
           内のものかどうかを区別する必要がある*/
        String op_oprndCodeList =
            "PUSH PUSHI POP POPI BLT BLE BEQ BNE BGE BGT JUMP";

        //命令のop部がop_oprndCodeList 内のものでなければ、opのみを出力
        if (op_oprndCodeList.indexOf(op) < 0)
            return  op;
        //命令のop部がop_oprndCodeList 何ものならば、opとaddrを出力
        else
            return  op + "¥t" + addr;
    }
}
```

（3） LexicalAnalyzer.java

```
package pram;
import static java.lang.Character.isLowerCase;
import static java.lang.Character.isUpperCase;
import static java.lang.Character.isDigit;
import static java.lang.Character.digit;
import static java.lang.System.err;
class LexicalAnalyzer {
```

```
private FileScanner sourceCode;    /* 解析対象となるファイルに対するスキャナ */

LexicalAnalyzer(String fileName){
        /* fileNameという名前のファイルのためのファイルスキャナを生成し、
           sourceCodeにより参照する                                    */
        sourceCode = new FileScanner(fileName);
}

/** トークンの切り出し */
Token nextToken(){
        /* トークンが切り出せたら、Tokenクラスのインスタンスを生成し、返り値とする
           トークンが切り出せなかったらメソッドsyntaxErrorを呼び出す          */
        String string = "";
        Token token = new Token(Symbol.NULL);
        char ch, nextCh;
do{
ch = sourceCode.nextChar();
}while(ch == ' ' || ch == '¥t' || ch == '¥n');

/** コメントの記述を可能とするプログラム */
do{
        // 「/*」で始まり、「*/」で終わる文字列をコメントとみなし、その文字列を読み飛ばす。
        if(ch == '/' && sourceCode.lookAhead() == '*'){
                ch = sourceCode.nextChar();
                do{
                        ch = sourceCode.nextChar();
                }while(!(ch == '*' && sourceCode.lookAhead() == '/'));
                ch = sourceCode.nextChar();
                do{
                        ch = sourceCode.nextChar();
                }while(ch == ' ' || ch == '¥t' || ch == '¥n');
        }
        // 「//」で始まる文字列をコメントとみなし、その文字列を読み飛ばす。
        else if(ch == '/' && sourceCode.lookAhead() == '/'){
                int line = sourceCode.currentLineNumber();
                ch = sourceCode.nextChar();
                do{
                        ch = sourceCode.nextChar();
                }while(line == sourceCode.currentLineNumber());
```

```
                ch = sourceCode.nextChar();
                do{
                        ch = sourceCode.nextChar();
                }while(ch == ' ' || ch == '¥t' || ch == '¥n');
        }
}while((ch == '/' && sourceCode.lookAhead() == '*') ||
                        (ch == '/' && sourceCode.lookAhead() == '/'));

if(ch == '¥0'){
        token = new Token(Symbol.EOF);
}else if(ch == '0')
        token = new Token(Symbol.INTEGER, 0);
 else if(isDigit(ch)){
        int val = digit(ch, 10);
        while(isDigit(sourceCode.lookAhead())){
                ch = this.sourceCode.nextChar();
                val = val*10 + digit(ch, 10);
        }
        token = new Token(Symbol.INTEGER, val);
}else if(isLowerCase(ch) || isUpperCase(ch)|| ch == '_'){
        string += ch;
        nextCh = sourceCode.lookAhead();
        while(isLowerCase(nextCh) || isUpperCase(nextCh) ||
                                        isDigit(nextCh)      || nextCh == '_'){
                ch = sourceCode.nextChar();
                string += ch;
                nextCh = sourceCode.lookAhead();
        }
        if(string.equals("main"))
                token = new Token(Symbol.MAIN);
        else if(string.equals("if"))
                token = new Token(Symbol.IF);
        else if(string.endsWith("else"))
                token = new Token(Symbol.ELSE);
        else if(string.equals("while"))
                token = new Token(Symbol.WHILE);
        else if(string.equals("readint"))
                token = new Token(Symbol.READINT);
        else if(string.equals("readchar"))
```

```
                token = new Token(Symbol.READCHAR);
        else if(string.equals("writeint"))
                token = new Token(Symbol.WRITEINT);
        else if(string.equals("writechar"))
                token = new Token(Symbol.WRITECHAR);
        else if(string.equals("int"))
                token = new Token(Symbol.INT);
        else if(string.equals("parallel"))
                token = new Token(Symbol.PARA);
        else
                token = new Token(Symbol.NAME,string);
}else{
switch(ch){
        case '=':
                if(sourceCode.lookAhead() == '='){
                        ch = this.sourceCode.nextChar();
                        token = new Token(Symbol.EQUAL);
                }else
                        token = new Token(Symbol.ASSIGN);
                break;
        case '!':
                if(sourceCode.lookAhead() == '='){
                        ch = this.sourceCode.nextChar();
                        token = new Token(Symbol.NOTEQ);
                }else
                        token = new Token(Symbol.NOT);
                break;
        case '<':
                if(sourceCode.lookAhead() == '='){
                        ch = this.sourceCode.nextChar();
                        token = new Token(Symbol.LESSEQ);
                }else
                        token = new Token(Symbol.LESS);
                break;
       case '>':
                if(sourceCode.lookAhead() == '='){
                        ch = this.sourceCode.nextChar();
                        token = new Token(Symbol.GREATEQ);
                }else
```

```
                token = new Token(Symbol.GREAT);
        break;
case '&':
        if(sourceCode.lookAhead() == '&'){
                ch = this.sourceCode.nextChar();
                token = new Token(Symbol.AND);
        }
        break;
case '|':
        if(sourceCode.lookAhead() == '|'){
                ch = this.sourceCode.nextChar();
                token = new Token(Symbol.OR);
        }
        break;
case '+':
        if(sourceCode.lookAhead() == '='){
                ch = this.sourceCode.nextChar();
                token = new Token(Symbol.ASSIGNADD);
        }else{
                if (sourceCode.lookAhead() == '+'){
                        ch = this.sourceCode.nextChar();
                        token = new Token(Symbol.INC);
                }else
                        token = new Token(Symbol.ADD);
        }
        break;
case '-':
        if(sourceCode.lookAhead() == '='){
                ch = this.sourceCode.nextChar();
                token = new Token(Symbol.ASSIGNSUB);
        }else if(sourceCode.lookAhead() == '-'){
                ch = this.sourceCode.nextChar();
                token = new Token(Symbol.DEC);
        }else
                token = new Token(Symbol.SUB);
        break;
case '*':
        if(sourceCode.lookAhead() == '='){
                ch = this.sourceCode.nextChar();
```

```
                token = new Token(Symbol.ASSIGNMUL);
            }else
                token = new Token(Symbol.MUL);
            break;
        case '/':
            if(sourceCode.lookAhead() == '='){
                ch = this.sourceCode.nextChar();
                token = new Token(Symbol.ASSIGNDIV);
            }else
                token = new Token(Symbol.DIV);
            break;
        case '%':
            if(sourceCode.lookAhead() == '='){
                ch = this.sourceCode.nextChar();
                token = new Token(Symbol.ASSIGNMOD);
            }else
                token = new Token(Symbol.MOD);
            break;
        case ';':
            token = new Token(Symbol.SEMICOLON);
            break;
        case '(':
            token = new Token(Symbol.LPAREN);
            break;
        case ')':
            token = new Token(Symbol.RPAREN);
            break;
        case '{':
            token = new Token(Symbol.LBRACE);
            break;
        case '}':
            token = new Token(Symbol.RBRACE);
            break;
        case '[':
            token =  new Token(Symbol.LBRACKET);
            break;
        case ']':
            token =  new Token(Symbol.RBRACKET);
            break;
```

```
        case ',':
                token = new Token(Symbol.COMMA);
                break;
        case '¥'':
                ch = this.sourceCode.nextChar();
                if(ch != '¥'' && sourceCode.lookAhead() == '¥''){
                        sourceCode.nextChar();
                        token = new Token(Symbol.CHARACTER, ch);
                }
                break;
        case '$':
                if(sourceCode.lookAhead() == 'p'){
                        ch = this.sourceCode.nextChar();
                        token = new Token(Symbol.PROCESSOR);
                }
                break;
        default:
                syntaxError();
                token = new Token(Symbol.NULL);
                break;
    }
}
return token;
}

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

void syntaxError(){
        /* 文法エラーであること、およびそれが何行目で発生したのかを標準出力に表示 */
        err.println(" 文法エラーです。" + (sourceCode.currentLineNumber()+1)
                                                +"行目でエラーが発生しました。");
        /* エラーの発生した行の内容を標準出力に表示 */
        err.println(" エラーが発生した行の内容を表示します。¥n" +
                                        sourceCode.currentLine());
        /* ファイルを閉じる*/
        closeFile();
```

```
                /* 終了する */
                System.exit(1);
        }
}
```

（4） Operator.java

```
package pram;
enum Operator
{
  NOP,
  ASSGN,
  ADD,
  ADDLHS,
  SUB,
  SUBLHS,
  MUL,
  MULLHS,
  DIV,
  DIVLHS,
  MOD,
  MODLHS,
  CSIGN,
  AND,
  OR,
  NOT,
  COMP,
  COPY,
  PUSH,
  PUSHI,
  REMOVE,
  POP,
  INC,
  DEC,
  SETFR,
  INCFR,
  DECFR,
  JUMP,
  BLT,
  BLE,
```

```
  BEQ,
  BNE,
  BGE,
  BGT,
  CALL,
  RET,
  HALT,
  INPUT,
  INPUTC,
  OUTPUT,
  OUTPUTC,
  ERROR,
  LOAD,
  PARA,
  SYNC,
  PUSHP
}
```

（5） Pram.java

```
package pram;

import static java.lang.System.out;
import static java.lang.System.err;
import static pram.Symbol.*;

public class Pram {
	private LexicalAnalyzer lexer; // 字句解析器
	private Token token; // 字句解析器からもらうトークン
	private VarTable variableTable; // 変数表

	private PseudoIseg iseg; // アセンブラコード表

	private String name = "";

	private int cnt = 0;

	static boolean inParallel;
	static int paraCnt = 0;
	public Pram(String sourceFileName) {
```

```
        /* 字句解析器のインスタンス生成と lexer による参照 */
        lexer = new LexicalAnalyzer(sourceFileName);
        /* 変数表のインスタンス生成と variableTablet による参照 */
        variableTable = new VarTable();
        /* アセンブラコード表のインスタンス生成と iseg による参照 */
        iseg = new PseudoIseg();
}

public static void main(String[] args) {
        Pram parser; // 構文解析器
        if (args.length == 0) { // 実行時パラメータのチェック
                out.println("Usage: java pram.Pram file [objectfile]");
                System.exit(1);
        }
        parser = new Pram(args[0]); // 構文解析器の生成
        inParallel = false;
        parser.parseProgram(); // 構文解析
        parser.closeFile(); // 入力ファイルのクローズ

        if (args.length == 1)
                parser.dump2file(); // OpCode.asm へ出力
        else
                parser.dump2file(args[1]); // 指定ファイルへ出力
}

// LexicalAnalyzer クラスのフィールド lexer を持ち、トークンの切り出しを行うメソッド
private void next(){
        token = lexer.nextToken();
}

/* symbol フィールドが、引数で与えられたトークン種別と一致するかどうかを
   確認するための boolean 型のメソッド */
private boolean check(Symbol s){
    return token.checkSymbol(s);
}
// check メソッドを用いて <Statement> の FIRST 集合をチェックする boolean 型メソッド
private boolean checkStatement(){
        return check(PARA)     ||
                check(IF)       ||    check(WHILE)  ||     check(NAME)     ||
```

```
                check(DEC)      ||    check(INC)    ||      check(INTEGER)  ||
                check(CHARACTER)||    check(LPAREN) ||      check(READCHAR) ||
                check(READINT)  ||    check(SUB)    ||      check(NOT)      ||
                check(WRITECHAR)||    check(WRITEINT)||     check(LBRACE) ||
                check(SEMICOLON);
}

// check メソッドを用いて <Expression> の FIRST 集合をチェックする boolean 型メソッド
private boolean checkExpression(){
        return check(NAME)     ||    check(DEC)      ||    check(INC)    ||
                check(INTEGER)  ||    check(CHARACTER)||    check(LPAREN) ||
                check(READCHAR) ||    check(READINT)  ||    check(SUB)    ||
                check(NOT);
}

/* check メソッドを用いて symbol フィールドが
   ("=" | "+=" | "*=" | "-=" | "/=" | "%=") かどうかをチャックするboolean型メソッド */
private boolean checkAssign(){
        return check(ASSIGN)    ||  check(ASSIGNADD)  ||  check(ASSIGNSUB)||
                check(ASSIGNMUL) ||  check(ASSIGNDIV)  ||  check(ASSIGNMOD);
}

/*  check メソッドを用いて symbol フィールドが
    ("==" | "!=" | "<" | "<=" | ">" | ">=") かどうかをチェックするboolean型メソッド */
private boolean checkEqual(){
        return check(EQUAL)  ||     check(NOTEQ)  ||     check(LESS)   ||
                check(LESSEQ) ||     check(GREAT)  ||     check(GREATEQ);
}

// checkメソッドを用いて <Unsigned_factor> の FIRST集合をチェックするboolean型メソッド
private boolean checkUnsigned_factor(){
        return check(PROCESSOR) || check(NAME) || check(DEC) || check(INC) ||
                check(INTEGER)    ||  check(CHARACTER)  ||  check(LPAREN)  ||
                check(READCHAR)   || check(READINT);
}
/* parseProgram 以下、各非終端記号 A に対する parseA メソッド */
private void parseProgram() {
        /* parseProgram の処理の記述 */
        parseMain_function();
```

```
        iseg.appendCode(Operator.HALT);
}

private void parseMain_function() {
        next();
        if (check(MAIN)) next();
        else syntaxError("main");
        if (check(LPAREN)) next();
        else syntaxError("(");
        if (check(RPAREN)) next();
        else syntaxError(")");
        parseBlock();
}

private void parseBlock() {
        if (check(LBRACE)) next();
        else syntaxError("{");
        while (check(INT)) parseVar_decl();
        while (checkStatement()) parseStatement();
        if (check(RBRACE)) next();
        else syntaxError("}");
}

private void parseVar_decl() {
        if (check(INT)) {
                next();
                if (check(NAME)) {
                        if (variableTable.exist(name = token.getName()) == true)
                                syntaxError("NAME");
                        else {
                                next();
                                parseType();
                        }
                } else syntaxError("NAME");
                while (check(COMMA)) {
                        next();
                        if (check(NAME)) {
                                if (variableTable.exist
                                        (name = token.getName()) == true)
```

```
                                        syntaxError("取り出した名前がすでに
                                                変数表に登録されています");
                                else {
                                        next();
                                        parseType();
                                }
                        } else syntaxError("NAME");
                }
        }
        if (check(SEMICOLON)) next();
        else syntaxError(";");
}

/*      取り出された名前がスカラー変数か配列変数かを決定することでその変数の型を決定し、
        配列変数ならそのサイズを求め、 addElement を用いて変数表に登録するメソッド      */
private void parseType(){
        int size=0, i=0, j=1;
        if (check(ASSIGN)) {
                token = lexer.nextToken();
                switch (token.getSymbol()) {
                case SUB:
                        if (check(SUB)) {
                                j = -j; // 符号反転
                                next();
                        } else syntaxError("-");
                case INTEGER:
                        while(token.getValue()
                                == 0 && token.getSymbol() == INTEGER) next();
                        if(check(INTEGER)){
                                i = token.getValue();
                                next();
                        }
                        else i = 0;
                        break;
                }
                variableTable.addElement(Type.INT, name, 1);
                iseg.appendCode(Operator.PUSHI, variableTable.getAddress(name));
                iseg.appendCode(Operator.PUSHI, i * j);
                iseg.appendCode(Operator.ASSGN);
```

```
                iseg.appendCode(Operator.REMOVE);
        } else if (check(LBRACKET)) {
                next();
                if(check(INTEGER)){
                        while(token.getValue() == 0 && check(INTEGER)) next();
                        if(check(INTEGER)){
                                size = token.getValue();
                                next();
                        }
                        else size = 0;
                }
                else syntaxError("配列の添字が欠落");
                if(check(RBRACKET)) next();
                else syntaxError("]");
                variableTable.addElement(Type.ARRAYOFINT, name, size);
        } else variableTable.addElement(Type.INT, name, 1);
}

private void parseStatement(){
        switch(token.getSymbol()){
        case PARA            : parseParallel_statement();       break;
        case IF              : parseIf_statement();             break;
        case WHILE           : parseWhile_statement();          break;
        case NAME            : parseExp_statement();            break;
        case DEC             : parseExp_statement();            break;
        case INC             : parseExp_statement();            break;
        case INTEGER         : parseExp_statement();            break;
        case CHARACTER       : parseExp_statement();            break;
        case LPAREN          : parseExp_statement();            break;
        case READCHAR        : parseExp_statement();            break;
        case READINT         : parseExp_statement();            break;
        case SUB             : parseExp_statement();            break;
        case NOT             : parseExp_statement();            break;
        case WRITECHAR       : parseWritechar_statement();      break;
        case WRITEINT        : parseWriteint_statement();       break;
        case LBRACE          :
                if(check(LBRACE)) next();
                else syntaxError("{");
                while(checkStatement()) parseStatement();
```

```
                if(check(RBRACE)) next();
                else syntaxError("}");
                break;
        case SEMICOLON       :
                if(check(SEMICOLON)) next();
                else syntaxError(";");
                break;
        }
}

private void parseParallel_statement() {
        paraCnt++;
        if(paraCnt>2 && inParallel) syntaxError("Not Ready Parallel");
        inParallel = true;
        if(check(PARA)) next();
        else syntaxError("parallel");
        if(check(LPAREN)) next();
        else syntaxError("(");
        if(checkExpression()) parseExpression();
        else syntaxError("parallel文の条件部の式が欠落");
        if(check(COMMA)) next();
        else syntaxError(",");
        if(checkExpression()) parseExpression();
        else syntaxError("parallel文の条件部の式が欠落");
        if(check(RPAREN)) next();
        else syntaxError(")");
        iseg.appendCode(Operator.PARA);
        parseStatement();
        iseg.appendCode(Operator.SYNC);
        inParallel = false;
}

private void parseIf_statement() {
        int ad1 = 0, ad2 = 0;
        if(check(IF)) next();
        else syntaxError("if");
        if (check(LPAREN)) next();
        else syntaxError("(");
        if(checkExpression()) parseExpression();
```

```
        else syntaxError("if文の条件部の式が欠落");
        if (check(RPAREN)) next();
        else syntaxError(")");
        ad1 = iseg.appendCode(Operator.BEQ);
        if (checkStatement()) {
                parseStatement();
                ad2 = iseg.appendCode(Operator.JUMP);
                iseg.replaceCode(ad1, iseg.pIsegPtr);
                if(check(ELSE)){
                        next();
                        if(checkStatement()){
                                parseStatement();
                                iseg.replaceCode(ad2, iseg.pIsegPtr);
                        }
                        else syntaxError("else文がそれに続く文を持っていません");
                }
        } else syntaxError("if文がそれに続く文を持っていません");
}

private void parseWhile_statement() {
        int ad1 = 0, ad2 = 0;
        if (check(WHILE)) next();
        else syntaxError("while");
        if (check(LPAREN)) next();
        else syntaxError("(");
        ad1 = iseg.pIsegPtr;
        if (checkExpression()) parseExpression();
        else syntaxError("while文の条件部の式が欠落");
        if (check(RPAREN)) next();
        else syntaxError(")");
        ad2 = iseg.appendCode(Operator.BEQ);
        parseStatement();
        iseg.appendCode(Operator.JUMP, ad1);
        iseg.replaceCode(ad2, iseg.pIsegPtr);
}
private void parseExp_statement() {
        parseExpression();
        if (check(SEMICOLON)) next();
        else syntaxError(";");
```

```
        iseg.appendCode(Operator.REMOVE);
}

private void parseWritechar_statement() {
        if (check(WRITECHAR)) next();
        else syntaxError("writechar");
        if (check(LPAREN)) next();
        else syntaxError("(");
        parseExpression();
        iseg.appendCode(Operator.OUTPUTC);
        if (check(RPAREN)) next();
        else syntaxError(")");
        if (check(SEMICOLON)) next();
        else syntaxError(";");
}

private void parseWriteint_statement() {
        if (check(WRITEINT)) next();
        else syntaxError("writeint");
        if (check(LPAREN)) next();
        else syntaxError("(");
        parseExpression();
        iseg.appendCode(Operator.OUTPUT);
        if (check(RPAREN)) next();
        else syntaxError(")");
        if (check(SEMICOLON)) next();
        else syntaxError(";");
}

private void parseExpression() {
        Operator op = Operator.NOP;
        parseExp();
        if (checkAssign()) {
                switch (token.getSymbol()) {
                case ASSIGN:
                        op = Operator.NOP;
                        if (check(ASSIGN)) next();
                        else syntaxError("=");
                        break;
```

```
                case ASSIGNADD:
                        op = Operator.ADD;
                        if (check(ASSIGNADD)) next();
                        else syntaxError("+=");
                        break;
                case ASSIGNSUB:
                        op = Operator.SUB;
                        if (check(ASSIGNSUB)) next();
                        else syntaxError("-=");
                        break;
                case ASSIGNMUL:
                        op = Operator.MUL;
                        if (check(ASSIGNMUL)) next();
                        else syntaxError("*=");
                        break;
                case ASSIGNDIV:
                        op = Operator.DIV;
                        if (check(ASSIGNDIV)) next();
                        else syntaxError("/=");
                        break;
                case ASSIGNMOD:
                        op = Operator.MOD;
                        if (check(ASSIGNMOD)) next();
                        else syntaxError("%=");
                        break;
                }
                if (op != Operator.NOP) {
                        iseg.appendCode(Operator.COPY);
                        iseg.appendCode(Operator.LOAD);
                }
                parseExpression();
                if (op != Operator.NOP) iseg.appendCode(op);
                iseg.appendCode(Operator.ASSGN);
        }
}

private void parseExp() {
        parseLogical_term();
        while (check(OR)) {
```

```
                if (check(OR)) next();
                else syntaxError("||");
                parseLogical_term();
                iseg.appendCode(Operator.OR);
        }
}

private void parseLogical_term() {
        parseLogical_factor();
        while (check(AND)) {
                if (check(AND)) next();
                else syntaxError("&&");
                parseLogical_factor();
                iseg.appendCode(Operator.AND);
        }
}

private void parseLogical_factor() {
        Operator op = Operator.NOP;
        parseArithmetic_expression();
        if (checkEqual()) {
                switch (token.getSymbol()) {
                case EQUAL:
                        op = Operator.BEQ;
                        if (check(EQUAL)) next();
                        else syntaxError("==");
                        break;
                case NOTEQ:
                        op = Operator.BNE;
                        if (check(NOTEQ)) next();
                        else syntaxError("!=");
                        break;
                case LESS:
                        op = Operator.BLT;
                        if (check(LESS)) next();
                        else syntaxError("<");
                        break;
                case LESSEQ:
                        op = Operator.BLE;
```

```
                        if (check(LESSEQ)) next();
                        else syntaxError("<=");
                        break;
                case GREAT:
                        op = Operator.BGT;
                        if (check(GREAT)) next();
                        else syntaxError(">");
                        break;
                case GREATEQ:
                        op = Operator.BGE;
                        if (check(GREATEQ)) next();
                        else syntaxError(">=");
                        break;
                }
                parseArithmetic_expression();
                iseg.appendCode(Operator.COMP);
                int ad1 = iseg.appendCode(op);
                iseg.appendCode(Operator.PUSHI, 0);
                int ad2 = iseg.appendCode(Operator.JUMP);
                iseg.replaceCode(ad1, iseg.pIsegPtr);
                iseg.appendCode(Operator.PUSHI, 1);
                iseg.replaceCode(ad2, iseg.pIsegPtr);
        }
}

private void parseArithmetic_expression() {
        Operator op = Operator.NOP;
        parseArithmetic_term();
        while (check(ADD) || check(SUB)) {
                switch (token.getSymbol()) {
                case ADD:
                        op = Operator.ADD;
                        if (check(ADD)) next();
                        else syntaxError("+");
                        break;
                case SUB:
                        op = Operator.SUB;
                        if (check(SUB)) next();
                        else syntaxError("-");
```

```
                        break;
                }
                parseArithmetic_term();
                iseg.appendCode(op);
                if (checkAssign()) syntaxError("左辺に式があります");
        }
}

private void parseArithmetic_term() {
        Operator op = Operator.NOP;
        parseArithmetic_factor();
        while (check(MUL) || check(DIV) || check(MOD)) {
                switch (token.getSymbol()) {
                case MUL:
                        op = Operator.MUL;
                        if (check(MUL)) next();
                        else syntaxError("*");
                        break;
                case DIV:
                        op = Operator.DIV;
                        if (check(DIV)) next();
                        else syntaxError("/");
                        break;
                case MOD:
                        op = Operator.MOD;
                        if (check(MOD)) next();
                        else syntaxError("%");
                        break;
                }
                parseArithmetic_factor();
                iseg.appendCode(op);
                if (checkAssign()) syntaxError("左辺に式があります");
        }
}

private void parseArithmetic_factor() {
        if (checkUnsigned_factor()) {
                parseUnsigned_factor();
        } else if (check(SUB)) {
```

```
                if (check(SUB)) next();
                else syntaxError("-");
                parseArithmetic_factor();
                iseg.appendCode(Operator.CSIGN);
        } else if (check(NOT)) {
                if (check(NOT)) next();
                else syntaxError("!");
                parseArithmetic_factor();
                iseg.appendCode(Operator.NOT);
        }
}

private void parseUnsigned_factor() {
        Operator op = Operator.NOP;
        if (check(PROCESSOR)){
                iseg.appendCode(Operator.PUSHP);
                next();
        }else if (check(NAME)) {
                name = token.getName();
                if (!variableTable.exist(name)) syntaxError("変数が存在しません");
                next();
                if (check(LBRACKET)) {
                        if (!variableTable.checkType(name, Type.ARRAYOFINT)) {
                                syntaxError(name + "は配列変数ではありません");
                        }
                        iseg.appendCode(Operator.PUSHI,
                                        variableTable.getAddress(name));
                        if (check(LBRACKET)) next();
                        else syntaxError("[");
                        if (checkExpression()) parseExpression();
                        else syntaxError("配列の添字が欠落");
                        if (check(RBRACKET)) next();
                        else syntaxError("]");
                        iseg.appendCode(Operator.ADD);
                        if (!checkAssign()) iseg.appendCode(Operator.LOAD);
                } else if (check(DEC) || check(INC)) {
                        if (!variableTable.checkType(name, Type.INT))
                                syntaxError(name + "はスカラー変数ではありません");
                        switch (token.getSymbol()) {
```

```
                case DEC:
                        op = Operator.DEC;
                        if (check(DEC)) next();
                        else syntaxError("--");
                        break;
                case INC:
                        op = Operator.INC;
                        if (check(INC)) next();
                        else syntaxError("++");
                        break;
                }
                iseg.appendCode(Operator.PUSH,
                                variableTable.getAddress(name));
                iseg.appendCode(Operator.COPY);
                iseg.appendCode(op);
                iseg.appendCode(Operator.POP,
                                variableTable.getAddress(name));
        } else {
                if (!variableTable.checkType(name, Type.INT))
                        syntaxError(name + "はスカラー変数ではありません");
                if (checkAssign()) {
                        iseg.appendCode(Operator.PUSHI,
                                        variableTable.getAddress(name));
                } else iseg.appendCode(Operator.PUSH,
                                        variableTable.getAddress(name));
        }
} else if (check(DEC) || check(INC)) {
        switch (token.getSymbol()) {
        case DEC:
                op = Operator.DEC;
                if (check(DEC)) next();
                else syntaxError("--");
                break;
        case INC:
                op = Operator.INC;
                if (check(INC)) next();
                else syntaxError("++");
                break;
        }
```

```
        if (check(NAME)) {
                name = token.getName();
                if (!variableTable.exist(name))
                        syntaxError("変数が存在しません");
                next();
        } else syntaxError("NAME");
        iseg.appendCode(Operator.PUSH, variableTable.getAddress(name));
        iseg.appendCode(op);
        iseg.appendCode(Operator.COPY);
        iseg.appendCode(Operator.POP, variableTable.getAddress(name));
} else if (check(INTEGER)) {
        int i;
        while(token.getValue() == 0 && check(INTEGER)) next();
        if(check(INTEGER)){
                i = token.getValue();
                next();
        }
        else i = 0;
        iseg.appendCode(Operator.PUSHI, i);
        if (checkAssign()) syntaxError("整数に代入しようとしています");
} else if (check(CHARACTER)) {
        iseg.appendCode(Operator.PUSHI, token.getValue());
        if (check(CHARACTER)) next();
        else syntaxError("CHARACTER");
} else if (token.checkSymbol(Symbol.LPAREN)) {
        if (check(LPAREN)) next();
        else syntaxError("(");
        if (checkExpression()) {
                parseExpression();
        } else syntaxError("不正な計算式です");
        if (check(RPAREN)) next();
        else syntaxError(")");
} else if (check(READCHAR)) {
        iseg.appendCode(Operator.INPUTC);
        if (check(READCHAR)) next();
        else syntaxError("readchar");
} else if (check(READINT)) {
        iseg.appendCode(Operator.INPUT);
        if (check(READINT)) next();
```

```
                    else syntaxError("readint");
                    if (checkAssign()) syntaxError("予約語に代入しようとしています");
                }
        }

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

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

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

        /** 文法エラー出力 */
        private void syntaxError(String s) {
                cnt++;
                err.println("Error " + cnt + " : " + s);
                this.lexer.syntaxError();
                out.println();
        }
}
```

（6） PseudoIseg.java

```
package pram;
import java.io.*;   //ファイル入出力用
import java.util.*; //ArrayList用

class PseudoIseg
{
   ArrayList<Instruction> pIseg; //表本体
   int pIsegPtr; //命令数カウンタ
   PseudoIseg()
```

```
{
    pIseg = new ArrayList<Instruction>(); //表本体を作成
    pIsegPtr = 0;        //命令カウンタは 0
}

/* 一つの命令を作って、表に格納するメソッド、しかし実際には
   このあとで定義する appendCode や appendCode からのみ利用される */

int setI(Operator opcode, int flag, int addr)
 {
     /* オペレータ opcode, アドレス修飾子 flag, オペランド addr
        から一命令を作る。 */
     Instruction inst = new Instruction(opcode,flag,addr);

     //作った命令を表に追加し、カウンタをインクレメントする。
     pIseg.add(inst);
     pIsegPtr++;

     //返り値は、追加した命令の表内での位置。
     return pIsegPtr-1;
  }

//オペランドを必要とするオペレータを追加するメソッド。
 int appendCode(Operator opcode, int addr)
{
    return setI(opcode, 0, addr);
}

//オペランドを必要としないオペレータを追加するメソッド。
 int appendCode(Operator opcode)
{
    return setI(opcode, 0, 0);

}
 //表内の全ての命令を表示するメソッド。
 void dump()
{
    for (int i = 0; i < pIsegPtr; i++) {
           System.out.print(i + ": ");
```

```
            System.out.println(pIseg.get(i).printInstruction());
    }
}

/* 表内の全ての命令をファイルに出力するメソッド。引数なしで呼ばれた場合、
   ファイル名は "OpCode.asm" になる。 */
void dump2file()
{
            PrintWriter outputFile = null;
            try {
                    outputFile = new PrintWriter(
                                    new BufferedWriter(
                            new FileWriter("OpCode.asm")));
                    for (int i = 0; i < pIsegPtr; i++)
                            outputFile.println(
                                    (pIseg.get(i)).printInstruction());
        } catch(IOException exception) {
                System.out.println(exception);
        } finally {
         outputFile.close();
                    System.exit(1);
        }
}

/* 表内の全ての命令をファイルに出力するメソッド。引数ありで呼ばれた場合、
   ファイル名は その引数 になる。 */
    void dump2file(String outputFileName)
            {
                    PrintWriter outputFile = null;
                    try {
                            outputFile = new PrintWriter(
                                            new BufferedWriter(
                                    new FileWriter(outputFileName)));
                            for (int i = 0; i < pIsegPtr; i++)
                                    outputFile.println(
                                            (pIseg.get(i)).printInstruction());
                    } catch(IOException exception) {
                            System.out.println(exception);
                    } finally {
```

```
                        outputFile.close();
                        System.exit(1);
                    }
                }

    // ptr 番目の命令の オペレータ を op に変更するメソッド
    void replaceCode(int ptr, Operator op)
        {
            int oldReg = (pIseg.get(ptr)).reg;
            int oldAddr = (pIseg.get(ptr)).addr;
            Instruction inst = new Instruction(op, oldReg, oldAddr);
            pIseg.remove(ptr);
            pIseg.add(ptr,inst);
        }

    // ptr 番目の命令の オペランド を addrs に変更するメソッド
    void replaceCode(int ptr, int addrs)
    {
        int oldReg = pIseg.get(ptr).reg;
        String oldOp = pIseg.get(ptr).op;
        Instruction inst = new Instruction(oldOp, oldReg, addrs);
        pIseg.remove(ptr);
        pIseg.add(ptr, inst);
    }
}
```

（7） Symbol.java

```
package pram;
enum Symbol
{
   NULL,
   MAIN,              /* main */
   IF,                /* if */
   ELSE,              /* else */
   WHILE,             /* while */
   READINT,           /* readint */
   READCHAR,          /* readchar */
   WRITEINT,          /* writeint */
   WRITECHAR,         /* writechar */
```

```
    INT,                /* int */
    EQUAL,              /*,= */
    NOTEQ,              /* != */
    LESS,               /* < */
    GREAT,              /* > */
    LESSEQ,             /* <= */
    GREATEQ,            /* >= */
    AND,                /* && */
    OR,                 /* || */
    NOT,                /* ! */
    ADD,                /* + */
    SUB,                /* - */
    MUL,                /* * */
    DIV,                /* / */
    MOD,                /* % */
    ASSIGN,             /*, */
    ASSIGNADD,          /* += */
    ASSIGNSUB,          /* -= */
    ASSIGNMUL,          /* *= */
    ASSIGNDIV,          /* /= */
    ASSIGNMOD,          /* %= */
    INC,                /* ++ */
    DEC,                /* -- */
    SEMICOLON,          /* ; */
    LPAREN,             /* ( */
    RPAREN,             /* ) */
    LBRACE,             /* { */
    RBRACE,             /* } */
    LBRACKET,           /* [ */
    RBRACKET,           /* ] */
    COMMA,              /* , */
    INTEGER,            /* 整数 */
    CHARACTER,          /* 文字 */
    NAME,               /* 変数名 */
    EOF,                /* end of file */
    PARA,               /* parallel */
    PROCESSOR           /* $p */
}
```

（8） Token.java

```
package pram;
public class Token {
	private Symbol symbol;
	int value;
	String name;

	public Token(Symbol symbol){
		this.symbol = symbol;
	}

	public Token(Symbol symbol, int value){
		this.symbol = symbol;
		this.value = value;
	}

	public Token(Symbol symbol, String name){
		this.symbol = symbol;
		this.name = name;
	}

	boolean checkSymbol(Symbol symbol){
		if(this.symbol == symbol)
			return true;
		else return false;
	}

	public Symbol getSymbol(){
		return symbol;
	}

	public int getValue(){
		return value;
	}

	public String getName(){
		return name;
	}
}
```

（9） Type.java

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

（10） Var.java

```
package pram;
public class Var {
	private Type type;
	private String name;
	private int address;
	private int size;

	Var(Type type, String name, int address, int size){
		this.type = type;
		this.name = name;
		this.address = address;
		this.size = size;
	}

	public Type getType(){
		return type;
	}

	public String getName(){
		return name;
	}
	public int getAddress(){
		return address;
	}
	public int getSize(){
		return size;
	}
}
```

（１１）　VarTable.java

```
package pram;
import java.util.ArrayList;
public class VarTable {
	/* フィールドの定義 */
	ArrayList<Var> varList;
	int nextAddress;

	VarTable(){
		/* コンストラクタの処理を記述する */
		varList = new ArrayList<Var>();
		nextAddress = 0;
	}

	/** 表に新しい変数を登録するためのメソッド  */
	boolean addElement(Type type, String name, int size){
		/* 名前の重複チェック (既存の変数名なら false を返す) */
		if(exist(name)){
			return false;
		}else{
		/* 型情報 type、変数名 name と変数サイズ size から新たな Var の
		   インスタンスを作り、それを ArrayList varList (表の本体)に追加する */
		varList.add(new Var(type, name, nextAddress, size));
		/* nextAddress を追加した変数のサイズ分だけ増やす
		                                うまく追加できたら true を返す */
		nextAddress += size;
		return true;
		}
	}

	/** 既に変数表に登録されている変数名かどうかを調べるメソッド */
	boolean exist(String name){
		/* 表を最初から検索し、 name という変数名が既に表に存在すれば true、
		   存在しなければ false を返す */
		for(Var n : varList){
			if(n.getName().equals(name)){
				return true;
			}
		}
```

```
        return false;
}

/** 変数表から変数のアドレスを得るメソッド */
int getAddress(String name){
        /* 変数名 name の要素(Var.class のインスタンス)を表から探し、
           その要素の address フィールドの値を返す
           表にその変数が存在しない場合は -1 を返す */
        for(Var n : varList){
                if(n.getName().equals(name)){
                        return n.getAddress();
                }
        }
        return -1;
}

/** 変数表から変数の型を得るメソッド */
Type getType(String name){
        /* 変数名 name の要素を表から探し、その要素の type フィールドの値を返す
           表にその変数が存在しない場合は Type.NULL を返す */
        for(Var n : varList){
                if(n.getName().equals(name)){
                        return n.getType();
                }
        }
        return Type.NULL;
}

/** 変数の型を確認するメソッド */
boolean checkType(String name, Type varType){
        /* 変数名 name の要素の type フィールドが varType と一致するかどうかを調べ
           一致すれば true、一致しなければ false を返す */
        for(Var n : varList){
                if(n.getName().equals(name) && n.getType().equals(varType)){
                        return true;
                }
        }
        return false;
}
```

```
        /** 表から変数のサイズを得るメソッド */
        int getSize(String name){
                /* 変数名 name の要素を表から探し、その要素の size フィールドの値を返す
                   表にその変数が存在しない場合は -1 を返す */
                for(Var n : varList){
                        if(n.getName().equals(name)){
                                return n.getSize();
                        }
                }
                return -1;
        }

        public static void main(String[] args) {
                /* 仕様にある通りの動作を記述する */
                VarTable varTable = new VarTable();
                String [] addName ={"A","B","C","D","E"};
                for(String an: addName){
                        varTable.addElement(Type.INT, an ,1);
                        System.out.println("Address: " + varTable.getAddress(an)
                                                + "   Type: " + varTable.getType(an)
                                                        + "   Size: " + varTable.getSize(an));
                }
        }

}
```

5. PVSM インタプリタプログラム

以下に本研究で用いた PVSM インタプリタプログラムを示す。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）　DataSegment.java
（２）　InputFile.java
（３）　Instraction.java
（４）　InstractionSegment.java
（５）　OpCode.java
（６）　Operators.java
（７）　PramMode.java
（８）　Stack.java
（９）　Type.java
（１０）　VirtualStackMachine.java
（１１）　VSM.java
（１２）　VSMLexer.java

（１）　DataSegment.java

```
public class DataSegment implements PramMode, Type {
	static final int DSEGSIZE = 1024;
	Object[] dseg;		// メモリ
	Object[] tempDseg;	// 作業用配列
	boolean[] readCheck;	// 同時読みチェック
	int[] writeCheck;	// 同時書きチェック
	int maxAddr;		// 使用アドレスの最大値
	int mode;		// PRAM mode
	int size;		// サイズ

	public DataSegment(int m) {
		size = DSEGSIZE;
		dseg = new Object[DSEGSIZE];
		tempDseg = new Object[DSEGSIZE];
		readCheck = new boolean[DSEGSIZE];
		writeCheck = new int[DSEGSIZE];
		maxAddr = -1;
		mode = m;

		for (int i=0; i<DSEGSIZE; i++) { // データの初期化
			dseg[i] = tempDseg[i] = null;
```

```
                // nullで初期化
                // nullは便宜上int値0として扱う
                readCheck[i] = false;
                writeCheck[i] = 0;
            }
        }

    // VSM p がアドレス addr のint型データを読む
        // addr のデータがint型に変換できない場合はエラー
        public int read (int addr, int p, int q) {
            if (addr < 0 || addr >= size)
                executeError ("Illegal dseg address : " + addr, p, q);
            if (addr>maxAddr) maxAddr = addr;
            if (mode == M_EREW && readCheck[addr])        /* 同時読みのチェック */
                executeError ("Concurrent read at data segment ["
                                                        + addr + "]", p, q);
            readCheck[addr] = true;
            int val = 0;
            Object data = dseg[addr];
            if (data == null) return 0; // nullは便宜上整数値0として扱う;
            int type = getType (addr);
            switch (type) {
            case T_INT:
                val = ((Integer) data).intValue();
                break;
            case T_DOUBLE:                              /* double型はint型に変換 */
                val = ((Double) data).intValue();
                break;
            case T_CHAR:                                /* char型はint型に変換 */
                val = (int) ((Character) data).charValue();
                break;
            case T_BOOL:                                /* boolean型はint型に変換 */
                val = (((Boolean) data).booleanValue()) ? 1 : 0;
                break;
            case T_STRING:                              /* String型はint型に変換 */
                try {                                     /* 変換できなければエラー */
                    val = Integer.parseInt ((String) data);
                } catch (NumberFormatException error_report) {
                    executeError ("Data segment[" + addr
```

```
                        + "] is not integer value : type Mismatched", p, q);
                }
                break;
        default:
                executeError ("Data segment[" + addr
                        + "] is not integer value : type Mismatched", p, q);
                break;
        }
        return val;
}

// VSM p がアドレス addr のdouble型データを読む
// addr のデータがdouble型に変換できない場合はエラー
public double readDouble (int addr, int p, int q) {
        if (addr < 0 || addr >= size)
                executeError ("Illegal dseg address : " + addr, p, q);
        if (addr>maxAddr) maxAddr = addr;
        if (mode == M_EREW && readCheck[addr])        /* 同時読みのチェック */
                executeError ("Concurrent read at data segment ["
                                                        + addr + "]", p, q);
        readCheck[addr] = true;
        double doubleVal = 0.0;
        Object data = dseg[addr];
        if (data == null) return 0.0; // nullは便宜上整数値0として扱う;
        int type = getType (addr);
        switch (type) {
        case T_DOUBLE:
                doubleVal = ((Double) data).doubleValue();
                break;
        case T_INT:                                /* int型はdouble型に変換 */
                doubleVal = ((Integer) data).doubleValue();
                break;
        case T_CHAR:                               /* char型はdouble型に変換 */
                doubleVal = (double) ((Character) data).charValue();
                break;
        case T_BOOL:                               /* boolean型はdouble型に変換 */
                doubleVal = (((Boolean) data).booleanValue()) ? 1.0 : 0.0;
                break;
        case T_STRING:                             /* String型はdouble型に変換 */
```

```
                try {                                   /* 変換できなければエラー */
                        doubleVal = Double.parseDouble ((String) data);
                } catch (NumberFormatException error_report) {
                        executeError ("Data segment[" + addr + "] data is not double
                                                value : type Mismatched", p, q);
                }
                break;
        default:
                executeError ("Data segment[" + addr
                        + "] data is not double value : type Mismatched", p, q);
                break;
        }
        return doubleVal;
    }

// VSM p がアドレス addr のchar型データを読む
    // addr のデータがchar型に変換できない場合はエラー
    public char readChar (int addr, int p, int q) {
        if (addr < 0 || addr >= size)
                executeError ("Illegal dseg address : " + addr, p, q);
        if (addr>maxAddr) maxAddr = addr;
        if (mode == M_EREW && readCheck[addr])        /* 同時読みのチェック */
                executeError ("Concurrent read at data segment ["
                                                        + addr + "]", p, q);
        readCheck[addr] = true;
        char charVal = '¥0';
        Object data = dseg[addr];
        if (data == null) return '¥0'; // nullは便宜上整数値0として扱う;
        int type = getType (addr);
        switch (type) {
        case T_CHAR:
                charVal = ((Character) data).charValue();
                break;
        case T_INT:                                  /* int型はchar型に変換 */
                charVal = (char) ((Integer) data).intValue();
                break;
        case T_DOUBLE:                               /* double型はchar型に変換 */
                charVal = (char) ((Double) data).doubleValue();
                break;
```

```
            case T_BOOL:                                /* boolean型はchar型に変換 */
                    charVal = (((Boolean) data).booleanValue()) ? '¥1' : '¥0';
                    break;
            case T_STRING:                      /* String型は長さ1のときはchar型に変換 */
                    String str = (String) data;                 /* それ以外はエラー */
                    if (str.length() == 1)
                            charVal = str.charAt(0);
                    else executeError ("Data segment[" + addr
                                    + "] is not character : type Mismatched", p, q);
                    break;
            default:
                    executeError ("Data segment [" + addr
                                    + "] is not character : type Mismatched", p, q);
                    break;
            }
            return charVal;
    }

// VSM p がアドレス addr のboolean型データを読む
    public boolean readBool (int addr, int p, int q) {
            if (addr < 0 || addr >= size)
                    executeError ("Illegal dseg address : " + addr, p, q);
            if (addr>maxAddr) maxAddr = addr;
            if (mode == M_EREW && readCheck[addr])        /* 同時読みのチェック */
                    executeError ("Concurrent read at data segment ["
                                                    + addr + "]", p, q);
            readCheck[addr] = true;
            boolean boolVal = false;
            Object data = dseg[addr];
            if (data == null) return false; // nullは便宜上整数値0として扱う;
            int type = getType (addr);
            switch (type) {
            case T_BOOL:
                    boolVal = ((Boolean) data).booleanValue();
                    break;
            case T_INT:                     /* int型は0ならばfalse, それ以外はtrueを返す */
                    boolVal = ((Integer) data).intValue() != 0;
                    break;
            case T_DOUBLE:            /* double型は0.0ならばfalse, それ以外はtrueを返す */
```

```
                boolVal = ((Double) data).doubleValue() != 0.0;
                break;
            case T_CHAR:              /* char型は'¥0'ならばfalse, それ以外はtrueを返す */
                boolVal = ((Character) data).charValue() != '¥0';
                break;
            case T_STRING:        /* String型は"true"ならばtrue, それ以外はfalseを返す */
                boolVal = ((String) data).equalsIgnoreCase ("true");
                break;
            default:
                executeError ("Data segment[" + addr
                              + "] is not boolean : type Mismatched", p, q);
                break;
            }
            return boolVal;
    }

// VSM p がアドレス addr のString型データを読む
    public String readString (int addr, int p, int q) {
            if (addr < 0 || addr >= size)
                executeError ("Illegal dseg address : " + addr, p, q);
            if (addr>maxAddr) maxAddr = addr;
            if (mode == M_EREW && readCheck[addr])        /* 同時読みのチェック */
                executeError ("Concurrent read at data segment ["
                                                    + addr + "]", p, q);
            readCheck[addr] = true;
            String str = "";
            Object data = dseg[addr];
            if (data == null) return "0"; // nullは便宜上整数値0として扱う;
            int type = getType (addr);
            switch (type) {
            case T_STRING:
                str = (String) data;
                break;
            case T_INT:                                /* int型はString型に変換 */
                str = ((Integer) data).toString();
                break;
            case T_DOUBLE:                             /* double型はString型に変換 */
                str = ((Double) data).toString();
                break;
```

```
            case T_CHAR:                                /* char型はString型に変換 */
                    str = ((Character) data).toString();
                    break;
            case T_BOOL:                                /* boolean型はString型に変換 */
                    str = ((Boolean) data).toString();
                    break;
            default:                                    /* Object型はString型に変換 */
                    str = data.toString();
                    break;
            }
            return str;
    }

    // VSM p がアドレス addr のObject型データを読む
    public Object readObject (int addr, int p, int q) {
            if (addr < 0 || addr >= size)
                    executeError ("Illegal dseg address : " + addr, p, q);
            if (addr>maxAddr) maxAddr = addr;
            if (mode == M_EREW && readCheck[addr])        /* 同時読みのチェック */
                 executeError ("Concurrent read at data segment [" + addr + "]", p, q);
            readCheck[addr] = true;
            if (dseg[addr] == null) return new Integer (0);
            else return dseg[addr];
    }

// VSM p がアドレス addr にint型データ val を書く
    public void write (int addr, int val, int p, int q) {
            if (addr < 0 || addr >= size)
                    executeError ("Illegal dseg address : " + addr, p, q);
            if (addr>maxAddr) maxAddr = addr;
            writeCheck[addr]++;
            if (writeCheck[addr]>1)
                    switch (mode) {
                    case M_EREW:
                    case M_CREW:
                            executeError ("Concurrent write at data segment ["
                                                            + addr + "]", p, q);
                            return;
                    case M_COMMON_CWCW:
```

```
                    // 共通型CRCWの場合、すでに書かれたデータと一致する場合のみ書き込まれる
                            // (同一のデータなので改めて書き込む必要は無い)
                            if (tempDseg[addr].getClass() != Integer.class)
                                    executeError ("Concurrent write at data segment ["
                                                                    + addr + "]", p, q);
                            else if (((Integer) tempDseg[addr]).intValue() != val)
                                    executeError ("Concurrent write at data segment ["
                                                                    + addr + "]", p, q);
                            return;
                    case M_ARBITRARY_CRCW:
                            // 任意型CRCWの場合、n個めのデータは確率1/nで書き込まれる
                            if ((Math.random () % writeCheck[addr]) < 1.0) break;
                            else return;
                    case M_PRIORITY_CRCW:
                            // 優先型CRCWの場合、すでにデータが(優先順位の高いプロセッサにより)
                            // 書かれている場合はデータを書き込まない
                            return ;
                    }
            tempDseg[addr] = new Integer (val);        /* 更新データはtmp配列に */
            return;
    }

// VSM p がアドレス addr にdouble型データ doubleVal を書く
    public void write (int addr, double doubleVal, int p, int q) {
            if (addr < 0 || addr >= size)
                    executeError ("Illegal dseg address : " + addr, p, q);
            if (addr>maxAddr) maxAddr = addr;
            writeCheck[addr]++;
            if (writeCheck[addr]>1)
                    switch (mode) {
                    case M_EREW:
                    case M_CREW:
                            executeError ("Concurrent write at data segment ["
                                                                    + addr + "]", p, q);
                            return;
                    case M_COMMON_CWCW:
                    // 共通型CRCWの場合、すでに書かれたデータと一致する場合のみ書き込まれる
                            // (同一のデータなので改めて書き込む必要は無い)
                            if (tempDseg[addr].getClass() != Double.class)
```

```
                        executeError ("Concurrent write at data segment ["
                                + addr + "]", p, q);
                    else if (((Double) tempDseg[addr]).doubleValue()
                                != doubleVal)
                        executeError ("Concurrent write at data segment ["
                                + addr + "]", p, q);
                    return;
                case M_ARBITRARY_CRCW:
                    // 任意型CRCWの場合、n個めのデータは確率1/nで書き込まれる
                    if ((Math.random () % writeCheck[addr]) < 1.0) break;
                    else return;
                case M_PRIORITY_CRCW:
                    // 優先型CRCWの場合、すでにデータが(優先順位の高いプロセッサにより)
                    // 書かれている場合はデータを書き込まない
                    return ;
                }
            tempDseg[addr] = new Double (doubleVal);    /* 更新データはtmp配列に */
            return;
        }

    // VSM p がアドレス addr にchar型データ charVal を書く
        public void write (int addr, char charVal, int p, int q) {
            if (addr < 0 || addr >= size)
                executeError ("Illegal dseg address : " + addr, p, q);
            if (addr>maxAddr) maxAddr = addr;
            writeCheck[addr]++;
            if (writeCheck[addr]>1)
                switch (mode) {
                case M_EREW:
                case M_CREW:
                    executeError ("Concurrent write at data segment ["
                                + addr + "]", p, q);
                    return;
                case M_COMMON_CWCW:
                    // 共通型CRCWの場合、すでに書かれたデータと一致する場合のみ書き込まれる
                    // (同一のデータなので改めて書き込む必要は無い)
                    if (tempDseg[addr].getClass() != Character.class)
                        executeError ("Concurrent write at data segment ["
                                + addr + "]", p, q);
```

```
                        else if (((Character) tempDseg[addr]).charValue()
                                                                != charVal)
                                executeError ("Concurrent write at data segment ["
                                                        + addr + "]", p, q);
                        return;
                case M_ARBITRARY_CRCW:
                        // 任意型CRCWの場合、n個めのデータは確率1/nで書き込まれる
                        if ((Math.random () % writeCheck[addr]) < 1.0) break;
                        else return;
                case M_PRIORITY_CRCW:
                        // 優先型CRCWの場合、すでにデータが(優先順位の高いプロセッサにより)
                        // 書かれている場合はデータを書き込まない
                        return ;
                }
            tempDseg[addr] = new Character(charVal);   /* 更新データはtmp配列に */
            return;
    }

// VSM p がアドレス addr にboolean型データ boolVal を書く
    public void write (int addr, boolean boolVal, int p, int q) {
            if (addr < 0 || addr >= size)
                    executeError ("Illegal dseg address : " + addr, p, q);
            if (addr>maxAddr) maxAddr = addr;
            writeCheck[addr]++;
                        if (writeCheck[addr]>1)
                switch (mode) {
                case M_EREW:
                case M_CREW:
                        executeError ("Concurrent write at data segment ["
                                                        + addr + "]", p, q);
                        return;
                case M_COMMON_CWCW:
                // 共通型CRCWの場合、すでに書かれたデータと一致する場合のみ書き込まれる
                        // (同一のデータなので改めて書き込む必要は無い)
                        if (tempDseg[addr].getClass() != Boolean.class)
                                executeError ("Concurrent write at data segment ["
                                                        + addr + "]", p, q);
                        else if (((Boolean) tempDseg[addr]).booleanValue()
                                                                != boolVal)
```

```
                        executeError ("Concurrent write at data segment ["
                                + addr + "]", p, q);
                    return;
                case M_ARBITRARY_CRCW:
                    // 任意型CRCWの場合、n個めのデータは確率1/nで書き込まれる
                    if ((Math.random () % writeCheck[addr]) < 1.0) break;
                    else return;
                case M_PRIORITY_CRCW:
                    // 優先型CRCWの場合、すでにデータが(優先順位の高いプロセッサにより)
                    // 書かれている場合はデータを書き込まない
                    return ;
                }
            tempDseg[addr] = Boolean.valueOf(boolVal);   /* 更新データはtmp配列に */
            return;
        }

    // VSM p がアドレス addr にString型データ str を書く
        public void write (int addr, String str, int p, int q) {
            if (addr < 0 || addr >= size)
                executeError ("Illegal dseg address : " + addr, p, q);
            if (addr>maxAddr) maxAddr = addr;
            writeCheck[addr]++;
            if (writeCheck[addr]>1)
                switch (mode) {
                case M_EREW:
                case M_CREW:
                    executeError ("Concurrent write at data segment ["
                                + addr + "]", p, q);
                    return;
                case M_COMMON_CWCW:
                // 共通型CRCWの場合、すでに書かれたデータと一致する場合のみ書き込まれる
                    // (同一のデータなので改めて書き込む必要は無い)
                    if (tempDseg[addr].getClass() != String.class)
                        executeError ("Concurrent write at data segment ["
                                + addr + "]", p, q);
                    else if (!((String) tempDseg[addr]).equals (str))
                        executeError ("Concurrent write at data segment ["
                                + addr + "]", p, q);
                    return;
```

```
                case M_ARBITRARY_CRCW:
                        // 任意型CRCWの場合、n個めのデータは確率1/nで書き込まれる
                        if ((Math.random () % writeCheck[addr]) < 1.0) break;
                        else return;
                case M_PRIORITY_CRCW:
                        // 優先型CRCWの場合、すでにデータが(優先順位の高いプロセッサにより)
                        // 書かれている場合はデータを書き込まない
                        return ;
                }
                tempDseg[addr] = str;                /* 更新データはtmp配列に */
        return;
    }

// VSM p がアドレス addr にObject型データ o を書く
    public void write (int addr, Object o, int p, int q) {
        if (addr < 0 || addr >= size)
                executeError ("Illegal dseg address : " + addr, p, q);
        if (addr>maxAddr) maxAddr = addr;
        writeCheck[addr]++;
        if (writeCheck[addr]>1)
                switch (mode) {
                case M_EREW:
                case M_CREW:
                case M_COMMON_CWCW:
                        executeError ("Concurrent write at data segment ["
                                                        + addr + "]", p, q);
                        return;
                case M_ARBITRARY_CRCW:
                        // 任意型CRCWの場合、n個めのデータは確率1/nで書き込まれる
                        if ((Math.random () % writeCheck[addr]) < 1.0) break;
                        else return;
                case M_PRIORITY_CRCW:
                        // 優先型CRCWの場合、すでにデータが(優先順位の高いプロセッサにより)
                        // 書かれている場合はデータを書き込まない
                        return ;
                }
                tempDseg[addr] = o;                /* 更新データはtmp配列に */
        return;
    }
```

```
// データ更新し、チェックフラグを初期化する
public void set() {
	for(int i=0; i<=maxAddr; i++) {
		dseg[i] = tempDseg[i];		/* tmp配列のデータをコピー */
		readCheck[i] = false;
		writeCheck[i] = 0;
	}
	return;
}

// Dsegの内容を表示する(デバグ用)
public void dump () {
	if (maxAddr == -1)
		System.out.println("Empty");
	else {
		for (int i=0; i<=maxAddr; i++) {
			Object data = dseg[i];
			If (data == null)continue;//初期化されていないDsegは表示しない
			if (i<10)
				System.out.print ("  " + i + "¥t");
			else if (i<100)
				System.out.print (" " + i + "¥t");
			else System.out.print (i + "¥t");
			if (data.getClass() == Integer.class)
				System.out.println (((Integer) data).intValue());
			else if (data.getClass() == Double.class)
				System.out.println (((Double) data).doubleValue());
			else if (data.getClass() == Character.class)
				System.out.println ("¥'"
					+ ((Character) data).charValue() + "¥'");
			else if (data.getClass() == Boolean.class)
				System.out.println
					(((Boolean) data).booleanValue());
			else if (data.getClass() == String.class)
				System.out.println ("¥"" + (String) data + "¥"");
			else System.out.println ("?");
		}
	}
}
```

```
        // Dsegアドレスaddrの内容を表示する(デバグ用)
        public void show (int addr) {
        }

        // アドレス addr のデータの型
        public int getType (int addr) {
                if (addr < 0 || addr >= size)
                        return T_ERROR;
                else {
                        Object o = dseg[addr];
                        if (o == null) return T_INT;
                        else if (o.getClass() == Integer.class) return T_INT;
                        else if (o.getClass() == Double.class) return T_DOUBLE;
                        else if (o.getClass() == Character.class) return T_CHAR;
                        else if (o.getClass() == Boolean.class) return T_BOOL;
                        else if (o.getClass() == String.class) return T_STRING;
                        else return T_ERROR;
                }
        }

        void executeError (String err_mes, int p, int q) {   /* 実行時エラー */
                System.out.println ("Processor " + p*10 + q);
                System.out.println ("Execute error at line " + VSM.vsm[p][q].pctr);
                System.out.println(err_mes);
                VSM.iseg.print(VSM.vsm[p][q].pctr);
                System.out.println();
                System.exit(1);
        }
}
```

（2） 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.
        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;
                }
        }
}
```

（3） 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
            }
        }
}
```

（4） 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 dumpToFile() {
            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 dumpToFile (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);
        }
}
```

（5） OpCode.java

```
class OpCode {
        String opName;
    int operator;
    int operand;

    public OpCode(String n, int i, int j) {
                opName = n;
                operator = i;
                operand = j;
    }

        public OpCode(String n, int i) {
                opName = n;
                operator = i;
                operand = -1;
    }

        public String toString() {
                if (operand == -1)
                        return opName;
                else return opName+'¥t'+operand;
        }
}
```

（6） 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
}
```

（7） PramMode.java

```
interface PramMode {
        static final int M_EREW = 0;
        static final int M_CREW = 1;
        static final int M_COMMON_CWCW = 2;
        static final int M_ARBITRARY_CRCW = 3;
```

```
        static final int M_PRIORITY_CRCW = 4;
}
```

（8） Stack.java

```
import java.util.ArrayList;  // ArrayList処理用

public class Stack implements Type {
        ArrayList stack;          // スタック
        int sp;                  // スタックポインタ
        int maxSp;               // スタックポインタの最大値
        int processorNumberX;       // プロセッサ番号
        int processorNumberY;

        // コンストラクタ
        // プロセッサpのスタックを作る
        public Stack (int p, int q) {
                stack = new ArrayList();
                sp = -1;
                maxSp = -1;
                processorNumberX = p;
                processorNumberY = q;
        }

        // スタックからint型データを取り出す
        // スタックの値がintに変換できない場合はエラー
        public int pop () {
                if (sp < 0) executeError ("Stack Underflow");;
                int val = 0;
                Object data = stack.get(sp);
                int type = getType();
                switch (type) {
                case T_INT:
                        val = ((Integer) data).intValue();
                        break;
                case T_DOUBLE:                          /* double型はint型に変換 */
                        val = ((Double) data).intValue();
                        break;
                case T_CHAR:                            /* char型はint型に変換 */
```

```
                val = (int) ((Character) data).charValue();
                break;
        case T_BOOL:                                /* boolean型はint型に変換 */
                val = (((Boolean) data).booleanValue()) ? 1 : 0;
        case T_STRING:                              /* String型はint型に変換 */
                try {                                 /* 変換できなければエラー */
                        val = Integer.parseInt ((String) data);
                } catch (NumberFormatException error_report) {
                        executeError ("Stack top data is not integer value :
                                                              type Mismatched");
                }
                break;
        default:
                executeError ("Stack top data is not integer value :
                                                              type Mismatched");
                break;
        }
        stack.remove(sp);
        sp--;
        return val;
}

// スタックからdoulbe型データを取り出す
// スタックの値がdoubleに変換できない場合はエラー
public double popDouble () {
        if (sp < 0) executeError ("Stack Underflow");;
        double doubleVal = 0.0;
        Object data = stack.get(sp);
        int type = getType();
        switch (type) {
        case T_DOUBLE:
                doubleVal = ((Double) data).doubleValue();
                break;
        case T_INT:                                 /* int型はdouble型に変換 */
                doubleVal = ((Integer) data).doubleValue();
                break;
        case T_CHAR:                                /* char型はdouble型に変換 */
                doubleVal = (double) ((Character) data).charValue();
                break;
```

```
        case T_BOOL:                                /* boolean型はdouble型に変換 */
                doubleVal = (((Boolean) data).booleanValue()) ? 1.0 : 0.0;
                break;
        case T_STRING:                              /* String型はdouble型に変換 */
                try {                                 /* 変換できなければエラー */
                        doubleVal = Double.parseDouble ((String) data);
                } catch (NumberFormatException error_report) {
                        executeError ("Stack top data is not double value :
                                                              type Mismatched");
                }
                break;
        default:
                executeError ("Stack top data is not double value :
                                                              type Mismatched");
                break;
        }
        stack.remove(sp);
        sp--;
        return doubleVal;
}

// スタックからchar型データを取り出す
// スタックの値がcharに変換できない場合はエラー
public char popChar () {
        if (sp < 0) executeError ("Stack Underflow");;
        char charVal = '¥0';
        Object data = stack.get(sp);
        int type = getType();
        switch (type) {
        case T_CHAR:
                charVal = ((Character) data).charValue();
                break;
        case T_INT:                                 /* int型はchar型に変換 */
                charVal = (char) ((Integer) data).intValue();
                break;
        case T_DOUBLE:                              /* double型はchar型に変換 */
                charVal = (char) ((Double) data).doubleValue();
                break;
        case T_BOOL:                                /* boolean型はchar型に変換 */
```

```
                charVal = (((Boolean) data).booleanValue()) ? '¥1' : '¥0';
                break;
        case T_STRING:                      /* String型は長さ1のときはchar型に変換 */
                String str = (String) data;              /* それ以外はエラー */
                if (str.length() == 1)
                        charVal = str.charAt(0);
                else executeError ("Stack top data is not character :
                                                        type Mismatched");
                break;
        default:
                executeError ("Stack top data is not character : type Mismatched");
                break;
        }
        stack.remove(sp);
        sp--;
        return charVal;
}

// スタックからboolean型データを取り出す
// スタックの値がbooleanに変換できない場合はエラー
public boolean popBool () {
        if (sp < 0) executeError ("Stack Underflow");;
        boolean boolVal = false;
        Object data = stack.get(sp);
        int type = getType();
        switch (type) {
        case T_BOOL:
                boolVal = ((Boolean) data).booleanValue();
                break;
        case T_INT:                    /* int型は0ならばfalse, それ以外はtrueを返す */
                boolVal = ((Integer) data).intValue() != 0;
                break;
        case T_DOUBLE:            /* double型は0.0ならばfalse, それ以外はtrueを返す */
                boolVal = ((Double) data).doubleValue() != 0.0;
                break;
        case T_CHAR:               /* char型は'¥0'ならばfalse, それ以外はtrueを返す */
                boolVal = ((Character) data).charValue() != '¥0';
                break;
        case T_STRING:        /* String型は"true"ならばtrue, それ以外はfalseを返す */
```

```
                boolVal = ((String) data).equalsIgnoreCase ("true");
                break;
        default:
                executeError ("Stack top data is not boolean : type Mismatched");
                break;
        }
        stack.remove(sp);
        sp--;
        return boolVal;
}

// スタックからString型データを取り出す
public String popString () {
        if (sp < 0) executeError ("Stack Underflow");
        String str = "";
        Object data = stack.get(sp);
        int type = getType();
        switch (type) {
        case T_STRING:
                str = (String) data;
                break;
        case T_INT:                                  /* int型はString型に変換 */
                str = ((Integer) data).toString();
                break;
        case T_DOUBLE:                               /* double型はString型に変換 */
                str = ((Double) data).toString();
                break;
        case T_CHAR:                                 /* char型はString型に変換 */
                str = ((Character) data).toString();
                break;
        case T_BOOL:                                 /* boolean型はString型に変換 */
                str = ((Boolean) data).toString();
                break;
        default:                                     /* Object型はString型に変換 */
                str = data.toString();
                break;
        }
        stack.remove(sp);
        sp--;
```

```
        return str;
}

// スタックからObject型データを取り出す
public Object popObject () {
        if (sp < 0) executeError ("Stack Underflow");
        Object o = stack.get(sp);
        stack.remove(sp);
        sp--;
        return o;
}

// スタックにint型データを挿入する
public void push (int val) {
        stack.add (new Integer (val));
        sp++;
        if (sp > maxSp) maxSp = sp;
        return;
}

// スタックにdouble型データを挿入する
public void push (double doubleVal) {
        stack.add (new Double (doubleVal));
        sp++;
        if (sp > maxSp) maxSp = sp;
        return;
}

// スタックにcharacter型データを挿入する
public void push (char charVal) {
        stack.add (new Character (charVal));
        sp++;
        if (sp > maxSp) maxSp = sp;
        return;
}

// スタックにboolean型データを挿入する
public void push (boolean boolVal) {
        stack.add (Boolean.valueOf (boolVal));
```

```
        sp++;
        if (sp > maxSp) maxSp = sp;
        return;
}

// スタックにString型データを挿入する
public void push (String str) {
        stack.add (str);
        sp++;
        if (sp > maxSp) maxSp = sp;
        return;
}

// スタックにObject型データを挿入する
public void push (Object o) {
        stack.add (o);
        sp++;
        if (sp > maxSp) maxSp = sp;
        return;
}

// スタックが空であるか
public boolean isEmpty () {
        return (sp < 0);
}

// スタックトップのデータの型
public int getType () {
        if (sp < 0) return T_ERROR;
        else {
                Object data = stack.get (sp);
                if (data.getClass() == Integer.class) return T_INT;
                else if (data.getClass() == Double.class) return T_DOUBLE;
                else if (data.getClass() == Character.class) return T_CHAR;
                else if (data.getClass() == Boolean.class) return T_BOOL;
                else if (data.getClass() == String.class) return T_STRING;
                else return T_ERROR;
        }
}
```

```
// スタックポインタiのデータの型
public int getType (int i) {
	if (i<0 || sp < i) return T_ERROR;
	else {
		Object data = stack.get (i);
		if (data.getClass() == Integer.class) return T_INT;
		else if (data.getClass() == Double.class) return T_DOUBLE;
		else if (data.getClass() == Character.class) return T_CHAR;
		else if (data.getClass() == Boolean.class) return T_BOOL;
		else if (data.getClass() == String.class) return T_STRING;
		else return T_ERROR;
	}
}

// スタックトップから2番めのデータの型
public int getSecondType () {
	if (sp < 1) return T_ERROR;
	else {
		Object data = stack.get (sp-1);
		if (data.getClass() == Integer.class) return T_INT;
		else if (data.getClass() == Double.class) return T_DOUBLE;
		else if (data.getClass() == Character.class) return T_CHAR;
		else if (data.getClass() == Boolean.class) return T_BOOL;
		else if (data.getClass() == String.class) return T_STRING;
		else return T_ERROR;
	}
}

// スタックトップを表示する
public String toString() {
	if (sp < 0) return "¥t¥t";
	else {
		String str = sp + "¥t";
		Object data = stack.get(sp);
		int type = getType();
		switch (type) {
		case T_INT:
			str += ((Integer) data).toString();
			break;
```

```
                case T_DOUBLE:
                        str += ((Double) data).toString();
                        break;
                case T_CHAR:
                        str += "¥'" + ((Character) data).toString() + "¥'";
                        break;
                case T_BOOL:
                        str += ((Boolean) data).toString();
                        break;
                case T_STRING:
                        str += "¥"" + (String) data + "¥"";
                        break;
                default:
                        str += "?";
                        break;
                }
                str += "¥t";
                return str;
        }
}
// スタックの内容を表示する(デバグ用)
public void dump () {
        if (sp < 0) System.out.println("Empty");
        else for (int i=0; i<=sp; i++) {
                String str = i + "¥t";
                Object data = stack.get(i);
                int type = getType(i);
                switch (type) {
                case T_INT:
                        str += ((Integer) data).toString();
                        break;
                case T_DOUBLE:
                        str += ((Double) data).toString();
                        break;
                case T_CHAR:
                        str += "¥'" + ((Character) data).toString() + "¥'";
                        break;
                case T_BOOL:
                        str += ((Boolean) data).toString();
```

```
                        break;
                    case T_STRING:
                        str += "¥"" + (String) data + "¥"";
                        break;
                    default:
                        str += "?";
                        break;
                    }
                    System.out.println(str);
                }
        }

        void executeError (String err_mes) {   /* 実行時エラー */
                System.out.println ("Processor " + processorNumberX*10
                                                        + processorNumberY);
                System.out.println ("Execute error at line "
                        + VSM.vsm[processorNumberX][processorNumberY].pctr);
                System.out.println (err_mes);
                VSM.iseg.print (VSM.vsm[processorNumberX][processorNumberY].pctr);
                System.out.println();
                System.exit(1);
        }
}
```

（9） 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;
}
```

（１０）　VirtualStackMachine.java

```
import ioTools.*;

public class VirtualStackMachine implements Operators, Type, PramMode {
        InstractionSegment iseg;// Instranciton Segment
        DataSegment dseg;      // Data Segment
        Stack stack;           // Stack
        int pctr;              // Program Counter
        boolean isRunning;     // status
        boolean fregP;
        int freg;              // Frame Register
        int minFreg;
        int callCtr;           // Call Counter
        int insCtr;            // Instrument Counter
        int processorNumberX;    // Processor Number
        int processorNumberY;

        public VirtualStackMachine (InstractionSegment is, DataSegment ds,
                                                    int p, int q, boolean fp) {
                iseg = is;
                dseg = ds;
                stack = new Stack (p, q);
                pctr = 0;
                isRunning = false;
                callCtr = 0;
                insCtr = 0;
                processorNumberX = p;
                processorNumberY = q;
                freg = 0;
                fregP = fp;
                minFreg = ds.size;
        }

        //isegのpctr番地の命令を実行
        public int execute (boolean traceSW) {
                int operator = iseg.operator (pctr);
                int operand = iseg.operand (pctr);
                if (iseg.register(pctr) && fregP)
                        operand += freg;
```

```
double doubleOperand = iseg.doubleOperand (pctr);
String stringOperand = iseg.stringOperand (pctr);

if (isRunning) {  // 状態が実行中のときのみ実行する
        int i;
        int val1, val2, addr;
        char charVal1, charVal2;
        double doubleVal1, doubleVal2;
        boolean boolVal1, boolVal2;
        String str1, str2;
        Object obj;
        int type1, type2;
        insCtr++;

        if (traceSW) {
                if (processorNumberX < 10)
                        System.out.print(" " + processorNumberX
                                        + processorNumberY + "¥t");
                else System.out.print(processorNumberX
                                        + processorNumberY + "¥t");
                iseg.print(pctr);
        }

        switch(operator) { // isegの命令により分岐
        case NOP:     /* No operation */
                pctr++;
                break;
        case ASSGN:    /* assign */
                obj = stack.popObject();
                addr = stack.pop();
                dseg.write (addr, obj, processorNumberX, processorNumberY);
                stack.push (obj);
                pctr++;
                break;
        case ADD:     /* + */
                type1 = stack.getType();
                type2 = stack.getSecondType();
                if (type1 == T_ERROR || type2 == T_ERROR)
                        executeError ("Type mismatch");
```

```
            else if (type1 == T_STRING || type2 == T_STRING) {
            /* string型に変換 */
                    str1 = stack.popString();
                    str2 = stack.popString();
                    stack.push (str2 + str1);
            } else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
            /* double型に変換 */
                    doubleVal1 = stack.popDouble();
                    doubleVal2 = stack.popDouble();
                    stack.push (doubleVal2 + doubleVal1);
            } else {
                    val1 = stack.pop();
                    val2 = stack.pop();
                    stack.push (val2 + val1);
            }
            pctr++;
            break;
        case ADDLHS:   /* += */
            obj = stack.popObject();
            addr = stack.pop();
            type1 = getType (obj);
            type2 = dseg.getType (addr);
            if (type1 == T_ERROR || type2 == T_ERROR)
                    executeError ("Type mismatch");
            else if (type1 == T_STRING || type2 == T_STRING) {
                    str1 = objToString (obj);
                    str2 = dseg.readString
                            (addr, processorNumberX, processorNumberY);
                    str1 = str2 + str1;
                    dseg.write (addr, str1,
                                    processorNumberX, processorNumberY);
                    stack.push (str2 + str1);
            } else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
                    doubleVal1 = objToDouble (obj);
                    doubleVal2 = dseg.readDouble
                            (addr, processorNumberX, processorNumberY);
                    doubleVal1 = doubleVal2 + doubleVal1;
                    dseg.write (addr, doubleVal1,
                                    processorNumberX, processorNumberY);
```

```
                stack.push (doubleVal1);
            } else {
                val1 = objToInt (obj);
                val2 = dseg.read (addr, processorNumberX,
                                                processorNumberY);
                val1 = val2 + val1;
                dseg.write (addr, val1, processorNumberX,
                                                processorNumberY);
                stack.push (val1);
            }
            pctr++;
            break;
        case SUB:     /* - */
            type1 = stack.getType();
            type2 = stack.getSecondType();
            if (type1 == T_ERROR || type2 == T_ERROR ||
                        type1 == T_STRING || type2 == T_STRING)
                executeError ("Type mismatch");
            else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
                doubleVal1 = stack.popDouble();
                doubleVal2 = stack.popDouble();
                stack.push (doubleVal2 - doubleVal1);
            } else {
                val1 = stack.pop();
                val2 = stack.pop();
                stack.push (val2 - val1);
            }
            pctr++;
            break;
        case SUBLHS:   /* -= */
            obj = stack.popObject();
            addr = stack.pop();
            type1 = getType (obj);
            type2 = dseg.getType (addr);
            if (type1 == T_ERROR || type2 == T_ERROR ||
                        type1 == T_STRING || type2 == T_STRING)
                executeError ("Type mismatch");
            else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
                doubleVal1 = objToDouble (obj);
```

```
                doubleVal2 = dseg.readDouble
                        (addr, processorNumberX, processorNumberY);
                doubleVal1 = doubleVal2 - doubleVal1;
                dseg.write (addr, doubleVal1,
                                processorNumberX, processorNumberY);
                stack.push (doubleVal1);
            } else {
                val1 = objToInt (obj);
                val2 = dseg.read (addr, processorNumberX,
                                                processorNumberY);
                val1 = val2 - val1;
                dseg.write (addr, val1, processorNumberX,
                                                processorNumberY);
                stack.push (val1);
            }
            pctr++;
            break;
        case MUL:     /* * */
            type1 = stack.getType();
            type2 = stack.getSecondType();
            if (type1 == T_ERROR || type2 == T_ERROR ||
                        type1 == T_STRING || type2 == T_STRING)
                executeError ("Type mismatch");
            else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
                doubleVal1 = stack.popDouble();
                doubleVal2 = stack.popDouble();
                stack.push (doubleVal2 * doubleVal1);
            } else {
                val1 = stack.pop();
                val2 = stack.pop();
                stack.push (val2 * val1);
            }
            pctr++;
            break;
        case MULLHS:   /* *= */
            obj = stack.popObject();
            addr = stack.pop();
            type1 = getType (obj);
            type2 = dseg.getType (addr);
```

```
            if (type1 == T_ERROR || type2 == T_ERROR ||
                        type1 == T_STRING || type2 == T_STRING)
                    executeError ("Type mismatch");
            else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
                    doubleVal1 = objToDouble (obj);
                    doubleVal2 = dseg.readDouble
                            (addr, processorNumberX, processorNumberY);
                    doubleVal1 = doubleVal2 * doubleVal1;
                    dseg.write (addr, doubleVal1,
                                processorNumberX, processorNumberY);
                    stack.push (doubleVal1);
            } else {
                    val1 = objToInt (obj);
                    val2 = dseg.read (addr, processorNumberX,
                                                processorNumberY);
                    val1 = val2 * val1;
                    dseg.write (addr, val1, processorNumberX,
                                                processorNumberY);
                    stack.push (val1);
            }
            pctr++;
            break;
    case DIV:     /* / */
            type1 = stack.getType();
            type2 = stack.getSecondType();
            if (type1 == T_ERROR || type2 == T_ERROR ||
                        type1 == T_STRING || type2 == T_STRING)
                    executeError ("Type mismatch");
            else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
                    doubleVal1 = stack.popDouble();
                    if (doubleVal1 == 0.0)
                            executeError("Zero divider detected");
                    doubleVal2 = stack.popDouble();
                    stack.push (doubleVal2 / doubleVal1);
            } else {
                    val1 = stack.pop();
                    if (val1 == 0)
                            executeError("Zero divider detected");
                    val2 = stack.pop();
```

```
                stack.push (val2 / val1);
            }
            pctr++;
            break;
        case DIVLHS:   /* /= */
            obj = stack.popObject();
            addr = stack.pop();
            type1 = getType (obj);
            type2 = dseg.getType (addr);
            if (type1 == T_ERROR || type2 == T_ERROR ||
                    type1 == T_STRING || type2 == T_STRING)
                executeError ("Type mismatch");
            else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
                doubleVal1 = objToDouble (obj);
                if (doubleVal1 == 0.0)
                    executeError ("Zero divider detected");
                doubleVal2 = dseg.readDouble
                    (addr, processorNumberX, processorNumberY);
                doubleVal1 = doubleVal2 / doubleVal1;
                dseg.write (addr, doubleVal1,
                        processorNumberX, processorNumberY);
                stack.push (doubleVal1);
            } else {
                val1 = objToInt (obj);
                if (val1 == 0)
                    executeError ("Zero divider detected");
                val2 = dseg.read (addr, processorNumberX,
                                        processorNumberY);
                val1 = val2 / val1;
                dseg.write (addr, val1, processorNumberX,
                                        processorNumberY);
                stack.push (val1);
            }
            pctr++;
            break;
        case MOD:     /* % */
            type1 = stack.getType();
            type2 = stack.getSecondType();
            if (type1 == T_ERROR || type2 == T_ERROR ||
```

```
                        type1 == T_STRING || type2 == T_STRING)
                executeError ("Type mismatch");
            else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
                doubleVal1 = stack.popDouble();
                if (doubleVal1 == 0.0)
                        executeError ("Zero divider detected");
                doubleVal2 = stack.popDouble();
                stack.push (doubleVal2 % doubleVal1);
            } else {
                val1 = stack.pop();
                if (val1 == 0)
                        executeError ("Zero divider detected");
                val2 = stack.pop();
                stack.push (val2 % val1);
            }
            pctr++;
            break;
        case MODLHS:   /* %= */
            obj = stack.popObject();
            addr = stack.pop();
            type1 = getType (obj);
            type2 = dseg.getType (addr);
            if (type1 == T_ERROR || type2 == T_ERROR ||
                        type1 == T_STRING || type2 == T_STRING)
                executeError ("Type mismatch");
            else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
                doubleVal1 = objToDouble (obj);
                if (doubleVal1 == 0.0)
                        executeError ("Zero divider detected");
                doubleVal2 = dseg.readDouble
                        (addr, processorNumberX, processorNumberY);
                doubleVal1 = doubleVal2 % doubleVal1;
                dseg.write (addr, doubleVal1,
                            processorNumberX, processorNumberY);
                stack.push (doubleVal1);
            } else {
                val1 = objToInt (obj);
                if (val1 == 0)
                        executeError ("Zero divider detected");
```

```
                val2 = dseg.read (addr, processorNumberX,
                                                processorNumberY);
                val1 = val2 % val1;
                dseg.write (addr, val1, processorNumberX,
                                                processorNumberY);
                stack.push (val1);
        }
        pctr++;
        break;
    case CSIGN:   /* 単項- */
        type1 = stack.getType();
        if (type1 == T_ERROR || type1 == T_STRING)
                executeError ("Type mismatch");
        else if (type1 == T_DOUBLE) {
                doubleVal1 = stack.popDouble();
                stack.push (-doubleVal1);
        } else {
                val1 = stack.pop();
                stack.push (-val1);
        }
        pctr++;
        break;
    case AND:     /* and */
        type1 = stack.getType();
        type2 = stack.getSecondType();
        if (type1 == T_ERROR || type2 == T_ERROR ||
                        type1 == T_STRING || type2 == T_STRING)
                executeError ("Type mismatch");
        else {
                boolVal1 = stack.popBool();
                boolVal2 = stack.popBool();
                stack.push (boolVal2 && boolVal1);
        }
        pctr++;
        break;
    case OR:      /* or */
        type1 = stack.getType();
        type2 = stack.getSecondType();
        if (type1 == T_ERROR || type2 == T_ERROR ||
```

```
                    type1 == T_STRING || type2 == T_STRING)
                executeError ("Type mismatch");
            else {
                boolVal1 = stack.popBool();
                boolVal2 = stack.popBool();
                stack.push (boolVal2 || boolVal1);
            }
            pctr++;
            break;
        case NOT:    /* not */
            type1 = stack.getType();
            if (type1 == T_ERROR ||  type1 == T_STRING)
                executeError ("Type mismatch");
            else {
                boolVal1 = stack.popBool();
                stack.push(!boolVal1);
            }
            pctr++;
            break;
        case XOR:    /* exclucive or */
            type1 = stack.getType();
            type2 = stack.getSecondType();
            if (type1 == T_ERROR || type2 == T_ERROR ||
                    type1 == T_STRING || type2 == T_STRING)
                executeError ("Type mismatch");
            else {
                boolVal1 = stack.popBool();
                boolVal2 = stack.popBool();
                stack.push ((boolVal2 && !boolVal1) ||
                                    (!boolVal2 && boolVal1));
            }
            pctr++;
            break;
        case COMP:    /* comp */
            type1 = stack.getType();
            type2 = stack.getSecondType();
            if (type1 == T_ERROR || type2 == T_ERROR)
                executeError ("Type mismatch");
            else if (type1 == T_STRING || type2 == T_STRING) {
```

```
                str1 = stack.popString();
                str2 = stack.popString();
                stack.push (str2.compareTo (str1));
        } else if (type1 == T_DOUBLE || type2 == T_DOUBLE) {
                doubleVal1 = stack.popDouble();
                doubleVal2 = stack.popDouble();
                if (doubleVal2 == doubleVal1) stack.push (0);
                else if (doubleVal2 > doubleVal1) stack.push (1);
                else stack.push (-1);
        } else {
                val1 = stack.pop();
                val2 = stack.pop();
                if (val2 == val1) stack.push (0);
                else if (val2 > val1) stack.push (1);
                else stack.push (-1);
        }
        pctr++;
        break;
case COPY:    /* copy */
        obj = stack.popObject();
        stack.push (obj);
        stack.push (obj);
        pctr++;
        break;
case PUSH:    /* push */
        stack.push (dseg.readObject (operand,
                        processorNumberX, processorNumberY));
        pctr++;
        break;
case PUSHI:   /* push integer */
        stack.push (operand);
        pctr++;
        break;
case PUSHC:   /* push character */
        stack.push ((char) operand);
        pctr++;
        break;
case PUSHD:   /* push double */
        stack.push (doubleOperand);
```

```
            pctr++;
            break;
        case PUSHB:    /* push boolean */
            stack.push (operand != 0);
            pctr++;
            break;
        case PUSHS:    /* push string */
            stack.push (stringOperand);
            pctr++;
            break;
        case PUSHP:    /* push processor number */
            stack.push (processorNumberX*10 + processorNumberY);
            pctr++;
            break;
        case POP:      /* pop */
            dseg.write(operand, stack.popObject(),
                        processorNumberX, processorNumberY);
            pctr++;
            break;
        case REMOVE:   /* remove */
            stack.popObject();
            pctr++;
            break;
        case LOAD:     /* load */
            addr = stack.pop();
            stack.push (dseg.readObject (addr,
                        processorNumberX, processorNumberY));
            pctr++;
            break;
        case INC:      /* ++ */
            type1 = stack.getType();
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                val1 = stack.pop();
                val1++;
                stack.push (val1);
                break;
```

```
            case T_DOUBLE:
                    doubleVal1 = stack.popDouble();
                    doubleVal1++;
                    stack.push (doubleVal1);
                    break;
            default:
                    executeError ("Type mismatch");
                    break;
            }
            pctr++;
            break;
        case DEC:     /* -- */
            type1 = stack.getType();
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                    val1 = stack.pop();
                    val1--;
                    stack.push (val1);
                    break;
            case T_DOUBLE:
                    doubleVal1 = stack.popDouble();
                    doubleVal1--;
                    stack.push(doubleVal1);
                    break;
            default:
                    executeError ("Type mismatch");
                    break;
            }
            pctr++;
            break;
        case PREINC:   /* 前置++ */
            addr = stack.pop();
            type1 = dseg.getType (addr);
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
```

```
                val1 = dseg.read (addr, processorNumberX,
                                                processorNumberY);
                val1++;
                dseg.write (addr, val1, processorNumberX,
                                                processorNumberY);
                stack.push (val1);
                break;
            case T_DOUBLE:
                doubleVal1 = dseg.readDouble (addr,
                             processorNumberX,processorNumberY);
                doubleVal1++;
                dseg.write (addr, doubleVal1,
                             processorNumberX, processorNumberY);
                stack.push (doubleVal1);
                break;
            default:
                executeError ("Type mismatch");
                break;
            }
            pctr++;
            break;
        case POSTINC:  /* 後置++ */
            addr = stack.pop();
            type1 = dseg.getType (addr);
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                val1 = dseg.read (addr, processorNumberX,
                                                processorNumberY);
                stack.push (val1);
                val1++;
                dseg.write (addr, val1, processorNumberX,
                                                processorNumberY);
                break;
            case T_DOUBLE:
                doubleVal1 = dseg.readDouble
                       (addr, processorNumberX, processorNumberY);
                stack.push (doubleVal1);
```

```
                doubleVal1++;
                dseg.write (addr, doubleVal1,
                                processorNumberX, processorNumberY);
                break;
            default:
                executeError ("Type mismatch");
                break;
            }
            pctr++;
            break;
        case PREDEC:  /* 前置-- */
            addr = stack.pop();
            type1 = dseg.getType (addr);
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                val1 = dseg.read (addr, processorNumberX,
                                        processorNumberY);
                val1--;
                dseg.write (addr, val1, processorNumberX,
                                        processorNumberY);
                stack.push (val1);
                break;
            case T_DOUBLE:
                doubleVal1 = dseg.readDouble
                        (addr, processorNumberX, processorNumberY);
                doubleVal1--;
                dseg.write (addr, doubleVal1,
                                processorNumberX, processorNumberY);
                stack.push (doubleVal1);
                break;
            default:
                executeError ("Type mismatch");
                break;
            }
            pctr++;
            break;
        case POSTDEC:  /* 後置-- */
```

```
            addr = stack.pop();
            type1 = dseg.getType (addr);
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                  val1 = dseg.read (addr, processorNumberX,
                                          processorNumberY);
                  stack.push (val1);
                  val1--;
                  dseg.write (addr, val1, processorNumberX,
                                          processorNumberY);
                  break;
            case T_DOUBLE:
                  doubleVal1 = dseg.readDouble
                        (addr, processorNumberX, processorNumberY);
                  stack.push (doubleVal1);
                  doubleVal1--;
                  dseg.write (addr, doubleVal1,
                              processorNumberX, processorNumberY);
                  break;
            default:
                  executeError ("Type mismatch");
                  break;
            }
            pctr++;
            break;
      case SETFR:    /* set frame register */
            freg = operand;
            if (freg > dseg.size)
                  executeError ("Freg overflow");
            else if (freg < minFreg) minFreg = freg;
            pctr++;
            break;
      case INCFR:    /* inc frame register */
            freg += operand;
            if (freg > dseg.size)
                  executeError ("Freg overflow");
            pctr++;
```

```
            break;
        case DECFR:    /* dec frame register */
            freg -= operand;
            if (freg < minFreg) minFreg = freg;
            pctr++;
            break;
        case JUMP:     /* jump */
            pctr = operand;
            break;
        case BEQ:      /* ==0 ? */
            type1 = stack.getType();
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                val1 = stack.pop();
                if (val1 == 0) pctr = operand;
                else pctr++;
                break;
            case T_DOUBLE:
                doubleVal1 = stack.popDouble();
                if (doubleVal1 == 0.0) pctr = operand;
                else pctr++;
                break;
            default:
                executeError ("Type mismatch");
                break;
            }
            break;
        case BNE:      /* !=0 ? */
            type1 = stack.getType();
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                val1 = stack.pop();
                if (val1 != 0) pctr = operand;
                else pctr++;
                break;
```

```
            case T_DOUBLE:
                    doubleVal1 = stack.popDouble();
                    if (doubleVal1 != 0.0) pctr = operand;
                    else pctr++;
                    break;
            default:
                    executeError ("Type mismatch");
                    break;
            }
            break;
    case BLT:     /* <0 ? */
            type1 = stack.getType();
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                    val1 = stack.pop();
                    if (val1 < 0) pctr = operand;
                    else pctr++;
                    break;
            case T_DOUBLE:
                    doubleVal1 = stack.popDouble();
                    if (doubleVal1 < 0.0) pctr = operand;
                    else pctr++;
                    break;
            default:
                    executeError ("Type mismatch");
                    break;
            }
            break;
    case BLE:    /* <=0 ? */
            type1 = stack.getType();
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                    val1 = stack.pop();
                    if (val1 <= 0) pctr = operand;
                    else pctr++;
```

```
                break;
            case T_DOUBLE:
                doubleVal1 = stack.popDouble();
                if (doubleVal1 <= 0.0) pctr = operand;
                else pctr++;
                break;
            default:
                executeError ("Type mismatch");
                break;
            }
            break;
        case BGT:    /* >0 ? */
            type1 = stack.getType();
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                val1 = stack.pop();
                if (val1 > 0) pctr = operand;
                else pctr++;
                break;
            case T_DOUBLE:
                doubleVal1 = stack.popDouble();
                if (doubleVal1 > 0.0) pctr = operand;
                else pctr++;
                break;
            default:
                executeError ("Type mismatch");
                break;
            }
            break;
        case BGE:    /* >=0 ? */
            type1 = stack.getType();
            switch (type1) {
            case T_INT:
            case T_CHAR:
            case T_BOOL:
                val1 = stack.pop();
                if (val1 >= 0) pctr = operand;
```

```
                else pctr++;
                break;
        case T_DOUBLE:
                doubleVal1 = stack.popDouble();
                if (doubleVal1 >= 0.0) pctr = operand;
                else pctr++;
                break;
        default:
                executeError ("Type mismatch");
                break;
        }
        break;
case CALL:    /* call */
        stack.push (pctr);
        pctr = operand;
        callCtr++;
        break;
case RET:     /* return */
        pctr = stack.pop();
        break;
case INPUT:   /* input integer */
        System.out.print ("Integer : ");
        val1 = Console.ReadInteger();
        stack.push (val1);
        pctr++;
        break;
case INPUTC:  /* input character */
        do {
                System.out.print ("Character : ");
                str1 = Console.ReadString();
        } while (str1.length()==0);
        charVal1 = str1.charAt(0);
        stack.push (charVal1);
        pctr++;
        break;
case INPUTD:  /* input double */
        System.out.print ("Double : ");
        doubleVal1 = Console.ReadDouble();
        stack.push (doubleVal1);
```

```
            pctr++;
            break;
        case INPUTS:  /* input string */
            System.out.print ("String : ");
            str1 = Console.ReadString();
            stack.push (str1);
            pctr++;
            break;
        case OUTPUT:  /* output integer */
            val1 = stack.pop();
            System.out.print (val1 + " ");
            pctr++;
            break;
        case OUTPUTD: /* output double */
            doubleVal1 = stack.popDouble();
            System.out.print (doubleVal1 + " ");
            pctr++;
            break;
        case OUTPUTC: /* output character */
            charVal1 = stack.popChar();
            if (traceSW)
                switch(charVal1) {
                case '¥0':
                    System.out.print ("¥¥0");
                    break;
                case '¥b':
                    System.out.print ("¥¥b");
                    break;
                case '¥n':
                    System.out.print ("¥¥n");
                    break;
                case '¥r':
                    System.out.print ("¥¥r");
                    break;
                case '¥t':
                    System.out.print ("¥¥t");
                    break;
                default:
                    System.out.print (charVal1);
```

```
                        break;
                }
            else System.out.print (charVal1);
            pctr++;
            break;
        case OUTPUTB:  /* output boolean */
            boolVal1 = stack.popBool();
            System.out.print (boolVal1 + " ");
            pctr++;
            break;
        case OUTPUTS:
            str1 = stack.popString ();
            if (traceSW)
                for (i=0; i<str1.length(); i++) {
                    char c = str1.charAt(i);
                    switch (c) {
                    case '¥0':
                        System.out.print ("¥¥0");
                        break;
                    case '¥b':
                        System.out.print ("¥¥b");
                        break;
                    case '¥n':
                        System.out.print ("¥¥n");
                        break;
                    case '¥r':
                        System.out.print ("¥¥r");
                        break;
                    case '¥t':
                        System.out.print ("¥¥t");
                        break;
                    default:
                        System.out.print (str1.charAt (i));
                        break;
                    }
                }
            else System.out.print (str1);
            pctr++;
            break;
```

```
case OUTPUTL:  /* output lne */
       if (traceSW) System.out.print("¥¥n");
       else System.out.println();
       pctr++;
       break;
case CASTI:    /* cast to int */
       val1 = stack.pop();
       stack.push (val1);
       pctr++;
       break;
case CASTC:    /* cast to char */
       charVal1 = stack.popChar();
       stack.push (charVal1);
       pctr++;
       break;
case CASTD:    /* cast to double */
       doubleVal1 = stack.popDouble();
       stack.push (doubleVal1);
       pctr++;
       break;
case CASTB:    /* cast to boolean */
       boolVal1 = stack.popBool();
       stack.push (boolVal1);
       pctr++;
       break;
case CASTS:    /* cast to string */
       str1 = stack.popString();
       stack.push (str1);
       pctr++;
       break;
case RAND:     /* random */
       stack.push (Math.random());
       pctr++;
       break;
case PARA:   /* parallel */
       if (VSM.parallelCount > VSM.high_processor_i+2
                                          && VSM.inParallel)
              executeError ("Already in parallel mode");
       int h = stack.pop();
```

```
                        int l = stack.pop();
                        VSM.setParallel(l, h);
                        VSM.inParallel = true;
                        VSM.parallelCount++;
                        break;
                case SYNC:   /* syncronous */
                        if(!VSM.inParallel)
                                executeError ("Already in sequential mode");
                        isRunning = false;
                        break;
                case HALT:   /* halt */
                        isRunning = false;
                        break;
                case EOF:    /* end of file */
                        executeError ("Illegal end of file");
                        break;
                default:
                        syntaxError ("Illegal operator");
                }
                if (traceSW)
                        System.out.println (stack);
        }
        return operator;
}

// プログラムカウンタをaddrにセット
public void setProgramCounter (int addr) {
        pctr = addr;
}

//プログラムカウンタを1増やす
public void incProgramCounter() {
        pctr++;
}

//状態を実行中に
public void awake() {
        isRunning = true;
}
```

```
void syntaxError () {              /* 文法エラー */
        System.out.println ("Processor " + processorNumberX+10
                                                    + processorNumberY);
        System.out.println ("Syntax error at line " + pctr);
        iseg.print (pctr);
        System.out.println ();
        System.exit (1);
}

void syntaxError (String err_mes) {   /* 文法エラー */
        System.out.println ("Processor " + processorNumberX*10
                                                    + processorNumberY);
        System.out.println ("Systax error at line " + pctr);
        System.out.println (err_mes);
        iseg.print(pctr);
        System.out.println();
        System.exit (1);
}

void executeError () {              /* 実行時エラー */
        System.out.println ("Processor " + processorNumberX*10
                                                    + processorNumberY);
        System.out.println ("Execute error at line " + pctr);
        iseg.print (pctr);
        System.out.println ();
        System.exit (1);
}

void executeError (String err_mes) {   /* 実行時エラー */
        System.out.println ("Processor " + processorNumberX*10
                                                    + processorNumberY);
        System.out.println ("Execute error at line " + pctr);
        System.out.println (err_mes);
        iseg.print (pctr);
        System.out.println ();
        System.exit (1);
}
```

```
// データの型を返す
int getType (Object data) {
	if (data.getClass() == Integer.class) return T_INT;
	else if (data.getClass() == Double.class) return T_DOUBLE;
	else if (data.getClass() == Character.class) return T_CHAR;
	else if (data.getClass() == Boolean.class) return T_BOOL;
	else if (data.getClass() == String.class) return T_STRING;
	else return T_ERROR;
}

// int型に変換する
int objToInt (Object data) {
	int val = 0;
	int type = getType (data);
	switch (type) {
	case T_INT:
		val = ((Integer) data).intValue();
		break;
	case T_DOUBLE:                          /* double型はint型に変換 */
		val = ((Double) data).intValue();
		break;
	case T_CHAR:                            /* char型はint型に変換 */
		val = (int) ((Character) data).charValue();
		break;
	case T_BOOL:                            /* boolean型はint型に変換 */
		val = (((Boolean) data).booleanValue()) ? 1 : 0;
	case T_STRING:                          /* String型はint型に変換 */
		try {                                 /* 変換できなければエラー */
			val = Integer.parseInt ((String) data);
		} catch (NumberFormatException error_report) {
			executeError ("Data is not integer value : type Mismatched");
		}
		break;
	default:
		executeError ("Data is not integer value : type Mismatched");
		break;
	}
	return val;
}
```

```
// double型に変換する
double objToDouble (Object data) {
        double doubleVal = 0.0;
        int type = getType (data);
        switch (type) {
        case T_DOUBLE:
                doubleVal = ((Double) data).doubleValue();
                break;
        case T_INT:                                  /* int型はdouble型に変換 */
                doubleVal = ((Integer) data).doubleValue();
                break;
        case T_CHAR:                                 /* char型はdouble型に変換 */
                doubleVal = (double) ((Character) data).charValue();
                break;
        case T_BOOL:                                 /* boolean型はdouble型に変換 */
                doubleVal = (((Boolean) data).booleanValue()) ? 1.0 : 0.0;
                break;
        case T_STRING:                               /* String型はdouble型に変換 */
                try {                                  /* 変換できなければエラー */
                        doubleVal = Double.parseDouble ((String) data);
                } catch (NumberFormatException error_report) {
                        executeError ("Data is not double value : type Mismatched");
                }
                break;
        default:
                executeError ("Data is not double value : type Mismatched");
                break;
        }
        return doubleVal;
}

// char型に変換する
char objToChar (Object data) {
        char charVal = '¥0';
        int type = getType (data);
        switch (type) {
        case T_CHAR:
                charVal = ((Character) data).charValue();
                break;
```

```
        case T_INT:                                 /* int型はchar型に変換 */
                charVal = (char) ((Integer) data).intValue();
                break;
        case T_DOUBLE:                              /* double型はchar型に変換 */
                charVal = (char) ((Double) data).doubleValue();
                break;
        case T_BOOL:                                /* boolean型はchar型に変換 */
                charVal = (((Boolean) data).booleanValue()) ? '¥1' : '¥0';
                break;
        case T_STRING:                      /* String型は長さ1のときはchar型に変換 */
                String str = (String) data;             /* それ以外はエラー */
                if (str.length() == 1)
                        charVal = str.charAt(0);
                else executeError ("Data is not character : type Mismatched");
                break;
        default:
                executeError ("Data is not character : type Mismatched");
                break;
        }
        return charVal;
}

// boolean型に変換する
boolean objToBool (Object data) {
        boolean boolVal = false;
        int type = getType (data);
        switch (type) {
        case T_BOOL:
                boolVal = ((Boolean) data).booleanValue();
                break;
        case T_INT:                   /* int型は0ならばfalse, それ以外はtrueを返す */
                boolVal = ((Integer) data).intValue() != 0;
                break;
        case T_DOUBLE:           /* double型は0.0ならばfalse, それ以外はtrueを返す */
                boolVal = ((Double) data).doubleValue() != 0.0;
                break;
        case T_CHAR:              /* char型は'¥0'ならばfalse, それ以外はtrueを返す */
                boolVal = ((Character) data).charValue() != '¥0';
                break;
```

```
            case T_STRING:        /* String型は"true"ならばtrue, それ以外はfalseを返す */
                    boolVal = ((String) data).equalsIgnoreCase ("true");
                    break;
            default:
                    executeError ("Data is not boolean : type Mismatched");
                    break;
            }
            return boolVal;
    }

    // String型に変換する
    String objToString (Object data) {
            String str = "";
            int type = getType (data);
            switch (type) {
            case T_STRING:
                    str = (String) data;
                    break;
            case T_INT:                                /* int型はString型に変換 */
                    str = ((Integer) data).toString();
                    break;
            case T_DOUBLE:                             /* double型はString型に変換 */
                    str = ((Double) data).toString();
                    break;
            case T_CHAR:                               /* char型はString型に変換 */
                    str = ((Character) data).toString();
                    break;
            case T_BOOL:                               /* boolean型はString型に変換 */
                    str = ((Boolean) data).toString();
                    break;
            default:                                   /* Object型はString型に変換 */
                    str = data.toString();
                    break;
            }
            return str;
    }
}
```

（１１）　VSM.java

```
import ioTools.*;
public class VSM implements Operators, PramMode {
        static final int PROCESSORMAX = 10;
        static final int MODE = M_EREW;        // 同時読み書きモード
        static final boolean FREGP = false;    // フレームレジスタ修飾(本VSMでは未使用)
        static VSMLexer lexer;                 // 字句解析機
        static VirtualStackMachine[][] vsm;    // Virtual Stack Machine
        static InstractionSegment iseg;        // Instraction Segment
        static DataSegment dseg;               // Data Segment

        static boolean inParallel;             // 並列モード
        static int parallelCount;
        static int low_processor_i;            // プロセッサ番号の下限
        static int high_processor_i;           // プロセッサ番号の上限
        static int low_processor_j;
        static int high_processor_j;
        static int cnt1,cnt2;

        static boolean execSW = true;          /* コンパイルだけ */
        static boolean objOutSW = false;       /* 目的コードの表示 */
        static boolean objPrtSW = false;       /* 目的コードの表示 */
        static boolean traceSW = false;        /* トレースモード */
        static boolean stateSW = false;        /* 実行データの表示 */
        static boolean debugSW = false;        /* デバッグモード */
        static boolean waitSW = false;         /* 1ステップ毎の実行モード */

        static int execTime = 0;               /* 実行時間 */
        static int execSequentialTime = 0;     /* 逐次処理部実行時間 */
        static int execParallelTime = 0;       /* 並列処理部実行時間 */
        static int execProcessors = 1;         /* 実行プロセッサ数 */
        static int execWork = 0;               /* 実行仕事量 */

        public static void main (String[] args) {
                String sourceFile = setUpOption (args);
                dseg = new DataSegment (MODE);          /* data segment */
                lexer = new VSMLexer (sourceFile);      /* 字句解析器 */
                iseg = lexer.makeIseg (debugSW);        /* instraction segment */
                lexer.inFile.closeFile();      /* 解析が終了したらファイルを閉じる */
```

```
        if (objOutSW) iseg.dump();
        if (objPrtSW) iseg.dumpToFile();

        if (execSW) {
                vsm = new VirtualStackMachine[PROCESSORMAX][PROCESSORMAX];
                for (int i=0; i<PROCESSORMAX; i++){
                        for (int j=0; j<PROCESSORMAX; j++){
                                vsm[i][j] = new VirtualStackMachine
                                                (iseg, dseg, i, j, FREGP);
                        }
                }
                startVSM (0);                        /* VSM開始 */
                if (stateSW) execReport();
                else System.out.println ("Execution finishied");
        }
}

public static void executeVSM (InstractionSegment is, boolean tsw,
                                boolean dsw, boolean ssw, boolean wsw) {
        dseg = new DataSegment (MODE);         /* data segment */
        iseg = is;
        traceSW = tsw;
        debugSW = dsw;
        stateSW = ssw;
        waitSW = wsw;

        if (objOutSW) iseg.dump();
        if (objPrtSW) iseg.dumpToFile();

        if (execSW) {
                vsm = new VirtualStackMachine[PROCESSORMAX][PROCESSORMAX];
                for (int i=0; i<PROCESSORMAX; i++){
                        for (int j=0; j<PROCESSORMAX; j++){
                                vsm[i][j] = new VirtualStackMachine
                                                (iseg, dseg, i, j, FREGP);
                        }
                }
                startVSM (0);                        /* VSM開始 */
                if (stateSW) execReport();
```

```
                else System.out.println ("Execution finishied");
        }
}

static void startVSM (int startAddr) {
        int operator;
        vsm[0][0].setProgramCounter (startAddr);
        vsm[0][0].awake();
        inParallel = false;
        low_processor_i = 0;
        high_processor_i = 0;
        low_processor_j = 0;
        high_processor_j = 0;

        while (true) {
                if (waitSW) Console.ReadString();
                execTime++;
                if (inParallel) {
                        execParallelTime++;
                        execWork += high_processor_i-low_processor_i+1;
                        execWork += high_processor_j-low_processor_j+1;
                        boolean reachSyncronous = true;
                        for (int i=low_processor_i; i<=high_processor_i; i++) {
                                for (int j=low_processor_j;
                                                j<=high_processor_j; j++){
                                        operator = vsm[i][j].execute(traceSW);
                                        if ((parallelCount > high_processor_i+2
                                        && operator == PARA) || operator == HALT)
                                                executeError ("Illegal operator
                                                on processor "+i, vsm[i][j].pctr);
                                        if (operator != SYNC) reachSyncronous = false;
                                }
                        }
                        if(parallelCount == high_processor_i+2) {
                                if(cnt1==0) {
                                        int addr =
                                vsm[low_processor_i][low_processor_j].pctr+1;
                                        for(int i=low_processor_i;
                                                        i<=high_processor_i; i++){
```

```
                        for(int j=low_processor_j;
                                j<=high_processor_j; j++){
                        vsm[i][j].setProgramCounter(addr);
                                vsm[i][j].awake();
                        }
                }
        }else {
                int addr =
                vsm[low_processor_i][low_processor_j].pctr;
                for(int i=low_processor_i;
                                i<=high_processor_i; i++){
                        for(int j=low_processor_j;
                                j<=high_processor_j; j++){
                        vsm[i][j].setProgramCounter(addr);
                                vsm[i][j].awake();
                        }
                }
        }
        cnt1++;
}
if (reachSyncronous) {
        int addr =
        vsm[low_processor_i][low_processor_j].pctr+1;
        if(cnt2==0) {
                for (int i=low_processor_i;
                                i<=high_processor_i; i++) {
                        for (int j=low_processor_j;
                                j<=high_processor_j; j++){
                        vsm[i][j].setProgramCounter(addr);
                                vsm[i][j].awake();
                        }
                }
        }else {
                inParallel = false;
                high_processor_i= 0;
                low_processor_i = 0;
                high_processor_j = 0;
                low_processor_j = 0;
                vsm[0][0].setProgramCounter(addr);
```

```
                        vsm[0][0].awake();
                    }
                    cnt2++;
                }
                if (high_processor_i-low_processor_i+1 > execProcessors){
                    if(high_processor_j-low_processor_j+1
                                                > execProcessors){
                        execProcessors =
                        (high_processor_i-low_processor_i+1)
                        *(high_processor_j-low_processor_j+1);
                    }
                }
            } else {
                execSequentialTime++;
                execWork++;
                operator = vsm[0][0].execute(traceSW);
                if (operator == HALT) break;
                else if (operator == PARA) {
                    int addr =
                    vsm[low_processor_i][low_processor_j].pctr+1;
                    for (int i=low_processor_i;
                                i<=high_processor_i; i++) {
                        for (int j=low_processor_j;
                                    j<=high_processor_j; j++){
                            vsm[i][j].setProgramCounter(addr);
                            vsm[i][j].awake();
                        }
                    }
                    if (high_processor_i-low_processor_i+1
                                                > execProcessors){
                        if(high_processor_j-low_processor_j+1
                                                > execProcessors){
                            System.out.println("a");
                            execProcessors =
                        (high_processor_i-low_processor_i+1);
                        }
                    }
                } else if (operator == SYNC)
            executeError ("Illegal operator on processor 0", vsm[0][0].pctr);
```

```
                }
                dseg.set();
        }
}

static String setUpOption (String[] args) {
        int i;
        for (i=0; i<args.length; i++) {
                if (args[i].charAt(0) == '-') {
                        if (args[i].indexOf('c') != -1) stateSW = true;
                                                        /* 実行データの表示 */
                        if (args[i].indexOf('d') != -1) debugSW = true;
                                                        /* デバッグモード */
                        if (args[i].indexOf('n') != -1) execSW = false;
                                                        /* コンパイルだけ */
                        if (args[i].indexOf('o') != -1) objOutSW = true;
                                                        /* 目的コードの表示 */
                        if (args[i].indexOf('p') != -1) objPrtSW = true;
                                                        /* 目的コードの表示 */
                        if (args[i].indexOf('t') != -1) traceSW = true;
                                                        /* トレースモード */
                        if (args[i].indexOf('w') != -1) waitSW = true;
                                                    /* 1ステップ毎の実行モード */
                } else break;
        }
        if (i<args.length) return args[i];
        else return "OpCode.asm";
}

public static void setParallel (int l, int h) {
        if (parallelCount==0){
                low_processor_i = l;
                high_processor_i = h;
        }else{
                low_processor_j = l;
                high_processor_j = h;
        }
        inParallel = true;
}
```

```
static void execReport() {
        System.out.println();
        System.out.println ("Execution time     : " + execTime);
        System.out.println ("Sequential time    : " + execSequentialTime);
        System.out.println ("Parallel time      : " + execParallelTime);
        System.out.println ("Number of procesosrs: " + execProcessors);
        int execCost = execTime * execProcessors;
        System.out.println ("Execution cost     : " + execCost);
        System.out.println ("Execution work time : " + execWork);
        System.out.println ("Object code size   : " + iseg.size);
        int maxStackDepth = vsm[0][0].stack.maxSp;
        for (int i=1; i<PROCESSORMAX; i++){
                for (int j=1; j<PROCESSORMAX; j++)
                        if (maxStackDepth<vsm[i][j].stack.maxSp)
                                        maxStackDepth = vsm[i][j].stack.maxSp;
        }
        System.out.println ("Max stack depth     : " + maxStackDepth);
        int maxFrameSize = dseg.size - vsm[0][0].minFreg;
        System.out.println ("Max frame size      : " + maxFrameSize);
        int FunctionCalls = 0;
        for (int i=0; i<PROCESSORMAX; i++){
                for (int j=0; j<PROCESSORMAX; j++)
                                FunctionCalls += vsm[i][j].callCtr;
        }
        System.out.println ("Function calls      : " + FunctionCalls);
}

static void syntaxError (int addr) {                /* 文法エラー */
        System.out.println ("Syntax error at line " + addr);
        iseg.print (addr);
        System.out.println();
        System.exit (1);
}

static void syntaxError (String err_mes, int addr) {  /* 文法エラー */
        System.out.println ("Systax error at line " + addr);
        System.out.println (err_mes);
        iseg.print (addr);
        System.out.println();
```

```
            System.exit (1);
        }

        static void executeError (int addr) {              /* 実行時エラー */
            System.out.println ("Execute error at line " + addr);
            iseg.print (addr);
            System.out.println();
            System.exit (1);
        }

        static 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);
            iseg.print (addr);
            System.out.println();
            System.exit (1);
        }
}
```

（１２）　VSMLexer.java

```
import java.util.Vector; //ベクトル用

public class VSMLexer implements Operators {
        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 VSMLexer (String fname) {
            //入力ファイルを開く
            inFile = new InputFile(fname);
        }

        public InstractionSegment makeIseg (boolean debugSW) {
            InstractionSegment iseg = new InstractionSegment (debugSW);
            while (true) {
                Instraction inst = nextOperator();
                if (inst.operator == EOF) break;
                iseg.appendCode (inst);
            }
            for (int i=0; i<iseg.size; i++) iseg.checkAddress(i);
```

```
        return iseg;
}

public Instraction nextOperator() {              /* 字句解析 次のオペレータを得る */
        int operator = NOP;
        int operand = Integer.MAX_VALUE;
        double doubleOperand = Double.NaN;
        String stringOperand = "";
        String str = "";
        char c;

        // 空白をスキップする
        do { c = skipSpace();
        } while (c == '¥n');

        if (c == '¥0') operator = EOF;                /* End of file */
        else if (Character.isUpperCase (c)) { str = extractWord (c);
                switch (c) {
                case 'A':
                        if (str.equals ("ADD")) operator = ADD;             /* ADD */
                        else if (str.equals("ADDLHS"))operator=ADDLHS;/* ADDLHS */
                        else if (str.equals ("AND")) operator = AND;        /* AND */
                        else if (str.equals ("ASSGN")) operator = ASSGN; /* ASSGN */
                        else syntaxError ("Unknown operator : "+str);
                        break;
                case 'B':
                        if (str.equals ("BEQ")) operator = BEQ;             /* BEQ */
                        else if (str.equals ("BGE")) operator = BGE;        /* BGE */
                        else if (str.equals ("BGT")) operator = BGT;        /* BGT */
                        else if (str.equals ("BLE")) operator = BLE;        /* BLE */
                        else if (str.equals ("BLT")) operator = BLT;        /* BLT */
                        else if (str.equals ("BNE")) operator = BNE;        /* BNE */
                        else syntaxError ("Unknown operator : "+str);
                        break;
                case 'C':
                        if (str.equals ("CALL")) operator = CALL;          /* CALL */
                        else if (str.equals ("CASTB")) operator = CASTB; /* CASTB */
                        else if (str.equals ("CASTC")) operator = CASTC; /* CASTC */
                        else if (str.equals ("CASTD")) operator = CASTD; /* CASTD */
```

```
        else if (str.equals ("CASTI")) operator = CASTI; /* CASTI */
        else if (str.equals ("CASTS")) operator = CASTS; /* CASTS */
        else if (str.equals ("COMP")) operator = COMP;    /* COMP */
        else if (str.equals ("COPY")) operator = COPY;    /* COPY */
        else if (str.equals ("CSIGN")) operator = CSIGN; /* CSIGN */
        else syntaxError ("Unknown operator : "+str);
        break;
case 'D':
        if (str.equals ("DEC")) operator = DEC;          /* DECFR */
        else if (str.equals ("DECDR")) operator = DECFR; /* DECFR */
        else if (str.equals ("DIV")) operator = DIV;       /* DIV */
        else if (str.equals ("DIVLHS")) operator=DIVLHS;/* DIVLHS */
        else syntaxError ("Unknown operator : "+str);
        break;
case 'E':
        if (str.equals ("ERROR")) operator = ERROR;      /* ERROR */
        else syntaxError ("Unknown operator : "+str);
        break;
case 'H':
        if (str.equals ("HALT")) operator = HALT;         /* HALT */
        else syntaxError ("Unknown operator : "+str);
        break;
case 'I':
        if (str.equals ("INC")) operator = INC;            /* INC */
        else if (str.equals ("INCFR")) operator = INCFR; /* INCFR */
        else if (str.equals ("INPUT")) operator = INPUT; /* INPUT */
        else if (str.equals ("INPUTC")) operator=INPUTC;/* INPUTC */
        else if (str.equals ("INPUTD")) operator=INPUTD;/* INPUTD */
        else if (str.equals ("INPUTS")) operator=INPUTS;/* INPUTS */
        else syntaxError ("Unknown operator : "+str);
        break;
case 'J':
        if (str.equals ("JUMP")) operator = JUMP;         /* JUMP */
        else syntaxError ("Unknown operator : "+str);
        break;
case 'L':
        if (str.equals ("LOAD")) operator = LOAD;         /* LOAD */
        else syntaxError ("Unknown operator : "+str);
        break;
```

```
case 'M':
       if (str.equals ("MOD")) operator = MOD;             /* MOD */
       else if (str.equals ("MODLHS")) operator=MODLHS;/* MODLHS */
       else if (str.equals ("MUL")) operator = MUL;        /* MUL */
       else if (str.equals ("MULLHS")) operator=MULLHS;/* MULLHS */
       else syntaxError ("Unknown operator : "+str);
       break;
case 'N':
       if (str.equals ("NOP")) operator = NOP;             /* NOP */
       else if (str.equals ("NOT")) operator = NOT;        /* NOT */
       else syntaxError ("Unknown operator : "+str);
       break;
case 'O':
       if (str.equals ("OR")) operator = OR;               /* OR */
       else if (str.equals ("OUTPUT")) operator=OUTPUT;/* OUTPUT */
      else if (str.equals("OUTPUTB"))operator=OUTPUT;/* OUTPUTB */
     else if (str.equals("OUTPUTC"))operator=OUTPUTC;/* OUTPUTC */
     else if (str.equals("OUTPUTD"))operator=OUTPUTD;/* OUTPUTD */
     else if (str.equals("OUTPUTL"))operator=OUTPUTL;/* OUTPUTL */
     else if (str.equals("OUTPUTS"))operator=OUTPUTS;/* OUTPUTS */
       else syntaxError ("Unknown operator : "+str);
       break;
case 'P':
       if (str.equals ("PARA")) operator = PARA;          /* PARA */
       else if (str.equals ("POP")) operator = POP;        /* POP */
     else if (str.equals("POSTDEC"))operator=POSTDEC;/* POSTDEC */
     else if (str.equals("POSTINC"))operator=POSTINC;/* POSTINC */
       else if (str.equals ("PREDEC")) operator=PREDEC;/* PREDEC */
       else if (str.equals ("PREINC")) operator=PREINC;/* PREINC */
       else if (str.equals ("PUSH")) operator = PUSH;    /* PUSH */
       else if (str.equals ("PUSHB")) operator = PUSHB; /* PUSHB */
       else if (str.equals ("PUSHC")) operator = PUSHC; /* PUSHC */
       else if (str.equals ("PUSHD")) operator = PUSHD; /* PUSHD */
       else if (str.equals ("PUSHI")) operator = PUSHI; /* PUSHI */
       else if (str.equals ("PUSHP")) operator = PUSHP; /* PUSHP */
       else if (str.equals ("PUSHS")) operator = PUSHS; /* PUSHS */
       else syntaxError ("Unknown operator : "+str);
       break;
```

```
        case 'R':
                if (str.equals ("RAND")) operator = RAND;          /* RAND */
                else if (str.equals ("RET")) operator = RET;        /* RET */
                else if (str.equals ("REMOVE")) operator=REMOVE;/* REMOVE */
                else syntaxError ("Unknown operator : "+str);
                break;
        case 'S':
                if (str.equals ("SETFR")) operator = SETFR;       /* SETFR */
                else if (str.equals ("SUB")) operator = SUB;        /* SUB */
                else if (str.equals ("SUBLHS")) operator=SUBLHS;/* SUBLHS */
                else if (str.equals ("SYNC")) operator = SYNC;    /* SYNC */
                else syntaxError ("Unknown operator : "+str);
                break;
        case 'X':
                if (str.equals ("XOR")) operator = XOR;             /* XOR */
                else syntaxError ("Unknown operator : "+str);
                break;
default:
                syntaxError ("Unknown operator : "+str);
                break;
        }
} else syntaxError ("Illegal charactor : "+c);

c = skipSpace();                              /* 空白をスキップ */

switch (operator) {
case PUSH:
case PUSHI:
case POP:
case BEQ:
case BNE:
case BLE:
case BLT:
case BGE:
case BGT:
case JUMP:
case CALL:
        operand = extractIntValue (c);
        c = skipSpace();
```

```
                if (c != '¥n' && c != '¥0' )
                        syntaxError ("Illegal character : "+c);
                return new Instraction (operator, operand);
        case PUSHD:
                doubleOperand = extractDoubleValue (c);
                c = skipSpace();
                if (c != '¥n' && c != '¥0' )
                        syntaxError ("Illegal character : "+c);
                return new Instraction (operator, doubleOperand);
        case PUSHC:
                operand = (int) extractChar (c);
                c = skipSpace();
                if (c != '¥n' && c != '¥0' )
                        syntaxError ("Illegal character : "+c);
                return new Instraction (operator, operand);
        case PUSHS:
                if (c != '¥"') syntaxError ("Need string operand");
                stringOperand = extractString (c);
                c = skipSpace();
                if (c != '¥n' && c != '¥0' )
                        syntaxError ("Illegal charactor : "+c);
                return new Instraction (operator, stringOperand);
        default:
                if (c != '¥n' && c != '¥0' )
                        syntaxError ("Illegal charactor : "+c);
                return new Instraction (operator);
        }
}

    // cで始まる命令語を得る
    public String extractWord (char c) {
            String str = String.valueOf(c);
            while (Character.isUpperCase(inFile.nextc)) {
                                                    // アセンブリ命令の文字は大文字のみ
                    c = inFile.nextChar();
                    str += c;
            }
            return str;
}
```

```
// cで始まる整数を得る
public int extractIntValue (char c) {
        if (!Character.isDigit (c) && c != '-') syntaxError ("Need int operand");
        String str = String.valueOf (c);
        while (Character.isDigit (inFile.nextc)) {
                c = inFile.nextChar();
                str += c;
        }
        return Integer.parseInt (str);
}

// cで始まる実数を得る
public double extractDoubleValue (char c) {
        if (!Character.isDigit (c) && c != '.' && c != '-')
                                        syntaxError ("Need double operand");
        String str = String.valueOf (c);
        while (Character.isDigit (inFile.nextc) || inFile.nextc == '.') {
                c = inFile.nextChar();
                str += c;
        }
        return Double.parseDouble (str);
}

// cで始まるbool値を得る
public boolean extractBool (char c) {
        String s = String.valueOf(c);
        while (Character.isLowerCase(inFile.nextc)
                || Character.isUpperCase(inFile.nextc)) {
                c = inFile.nextChar();
                s = s + c;
        }
        if (s.equalsIgnoreCase ("true")) return true;
        else if (s.equalsIgnoreCase ("false")) return false;
        else syntaxError ("Need boolean operand");
        return false;
}
```

```
// 文字を得る (文字列の前後は'¥''で囲まれている)
public char extractChar (char c) {
        if (c != '¥'') syntaxError ("Need char operand");
        c = inFile.nextChar();
        if (c == '¥¥') {
                c = inFile.nextChar();
                switch(c) {
                case '0':
                        c = '¥0';
                        break;
                case 'b':
                        c = '¥b';
                        break;
                case 'n':
                        c = '¥n';
                        break;
                case 'r':
                        c = '¥r';
                        break;
                case 't':
                        c = '¥t';
                        break;
                case '¥'':
                        c = '¥'';
                        break;
                case '¥¥':
                        c = '¥¥';
                        break;
                default:
                        syntaxError ("Illegal character : ¥¥" + c);
                        break;
                }
        } else if (c == '¥'') syntaxError ("No character");
        inFile.nextChar();
        if (inFile.currentc != '¥'') syntaxError ("Need ¥'");
        inFile.nextChar();
        return c;
}
```

```
// 文字列を得る (文字列の前後は'¥"'で囲まれている)
public String extractString (char c) {
        if (c != '¥"') syntaxError ("Need string operand");
        String str = "";
        while (inFile.nextc != '¥"') {
                c = inFile.nextChar ();
                if (c == '¥n') syntaxError ("Illegal end of line in string");
                if (c == '¥0') syntaxError ("Illegal end of file in string");
                if (c == '¥¥') {
                        c = inFile.nextChar();
                        switch (c) {
                        case '0':
                                c = '¥0';
                                break;
                        case 'b':
                                c = '¥b';
                                break;
                        case 'n':
                                c = '¥n';
                                break;
                        case 'r':
                                c = '¥r';
                                break;
                        case 't':
                                c = '¥t';
                                break;
                        case '¥'':
                                c = '¥'';
                                break;
                        case '¥¥':
                                c = '¥¥';
                                break;
                        default:
                                syntaxError ("Illegal character : ¥¥" + c);
                                break;
                        }
                }
                str += c;
        }
```

```
            inFile.nextChar ();
            return str;
        }

        public char skipSpace() {              /* 空白をスキップ */
            char c;
            do {
                c = inFile.nextChar();
                if (c == '#')
                    do {
                        c = inFile.nextChar();
                    } while (c != '¥n' && c != '¥0');
                if (c == '¥n' || c == '¥0') break;
            } while (c == ' ' || c == '¥t');
            return c;
        }

        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);
        }

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