卒業研究報告書

JAVA による PRAM コンパイラ作成

指導教員 石水　隆 助手

03-1-47-171

板東　俊助

近畿大学理工学部情報学科

平成 19 年 2 月 5 日提出

概要

より速いコンピュータの必要性はコンピュータが考案された当初からやむことはない。計算速度は常により大きくすることが求められている。例えば、高速計算が必要とされる科学や工学における数値モデリングやシミュレーションの分野では有効な結果を得るために、大量のデータに対して多数回の繰り返し計算が必要なことが多い。計算は適切な時間内に完了しなければならず、工学的計算やシミュレーションは秒あるいは分単位の時間でできなければならない。設計者が効率的に作業するための時間は短いので、解に到達するのに多時間要するシミュレーションは設計環境では受け入れられない。また、システムが複雑になればなるほど、それをシミュレートする時間はますます増加するし、計算に特定の締め切り時間があることもある。一つの例として、天気予報がそれに当てはまる。翌日の天気を正確に予測するのに 2 日もかかる予測は役に立たない。

計算速度を向上させる一つの方法として、一つの問題を解くのに複数のプロセッサを使うことが考えられる。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。

並列性は人間の思考方法および、コンピュータの使い方を確実に変化させる。それは、これまでに解くことのできなかった問題を手の届く範囲にし、また以前には夢にも見なかったような知識の新分野の研究も可能にする。豊富なアーキテクチャは新旧の問題に対して、貴重な解法より効果的な解法の発見の助けとなる。

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、代表的な並列計算モデルである PRAM に対し、PRAM シミュレータの一部である PRAM コンパイラを作成する。PRAM の並列アルゴリズムの実験的な評価・支援をする。

目次

1　序論

1.1 本研究の背景

1.1.1　並列計算機

今日、様々な分野で複雑な問題を高速に解く必要がある。計算速度を向上させるための方法として、性能の良いプロセッサを用いるか、あるいは並列処理を行うかの 2 通りが考えられる。今現在、単一のプロセッサ能力の向上とアルゴリズムの改良には限界があるということを予想されている。一方、並列処理には計算速度の限界は本質的に存在しない。そのため、より高度な高速化を行える計算機として並列計算機（Parallel Computer）が注目されている。並列計算機とは複数のプロセッサを持つ計算機のことで、計算時間の短縮と解ける問題のサイズが大きくなるという二つの利点を持つ。並列計算機は、全てのプロセッサが共有したメモリに対して読み書きを行い、プロセッサ間の通信もメモリを通じて行う共有メモリ型並列計算機（Shared Memory Parallel Computer）と、それぞれのプロセッサが局所メモリを持ち、プロセッサ間の通信は、ネットワークを通じて行う分散メモリ型並列計算機（Distributed Memory Parallel Computer）とに大別される。共有メモリ型計算機はプロセッサ間の通信を高速に行うことができ、プロセッサ間での同期も取りやすいため、通信及び同期にかかるコストを気にせずに高速化を得ることが出来る。一方、プロセッサ数の増加に従い、1 つの共有メモリに全てのプロセッサをつなぐことは困難となる。このため今現在では、プロセッサ数の多い並列計算機では分散メモリ型が主流となっている。また、複数の計算機をネットワークでつなぎ、それ全体を仮想的な計算機として扱うクラスタ処理（Cluster）やグリッド処理（Grid）も幅広く行われている。

高速化が必要とされる分野の具体的な例としては次のようなものがある。

（1）地球環境のシミュレーション分野
（2）天気予報で用いられる数値予報分野
（3）量子力学に基づく分子設計分野
（4）原始物理分野
（5）宇宙物理のシミュレーション分野

これらの分野では、複雑な問題の高速化が求められるが、その際に並列計算機を用いて計算量の大きな問題を短時間で解いている。

1.1.2　並列処理と並列アルゴリズム

現在、並列処理に対して大きな期待が寄せられている。並列処理は、ある問題を解く際、その問題をより小さい複数の部分問題に分割し、その部分問題を複数のプロセッサが協調して解く処理である。並列処理を行うことにより複雑な問題を高速に解くことが出来る。

アルゴリズム（Algorithm）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自信は計算機と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）は並列計算機上で問題を解く手順を記述したものである。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。そのため、一般的には逐次アルゴリズムをそのまま並列アルゴリズムにすることはできず、並列処理に対応させて新たにアルゴリズムを作成する必要がある。

1.1.3 並列計算モデル

以下に逐次アルゴリズムと並列アルゴリズムの違いについて述べる。逐次アルゴリズムでは、図 1. に示す RAM（Random Access Machine）上で実行される。RAM はメモリ、プロセッサ、命令の三つからなり、任意の位置にあるデータを読み書き可能にするものである。この逐次計算モデルによると、プロセッサ一台からメモリに 1 単位時間に 1 命令を実行可能としたものである。

並列計算モデルは、並列システム（Parallel System）と分散システム（Distributed System）大きく 2 つに分類することができる。並列システムと分散システムの両方に共有するのは、非常に多くの独立したプロセッサからなっているということである。並列システムと分散システムの最も大きな違いは、協調するかしないかということと、結合度の強さである。逐次計算の場合には計算のみを考えれば十分であったが、並列計算では通信も考える必要がある。並列計算ではプロセッサ間のデータ転送の方法も様々であり、これが並列計算の最も難しい側面である。並列システムにおいては CPU とメモリの結合の仕方に多くの方式が考えられる。この結合がどのようになっているか、またその結合のもとでどのように通信が行われるか、ということを定義しなければならない。したがって、モデルが一つしかないということはない。

本研究においては、並列システムを対象とする。以下の図 2. は、4 台のプロセッサを持つ共有メモリ方並列計算モデルである。すなわち、図 2. のモデルでは各プロセッサがそれぞれ処理を行うことにより、同時に 4 つの処理を行うことができる。

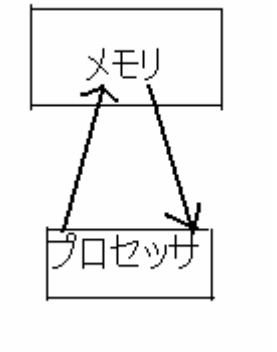

図 1. RAM

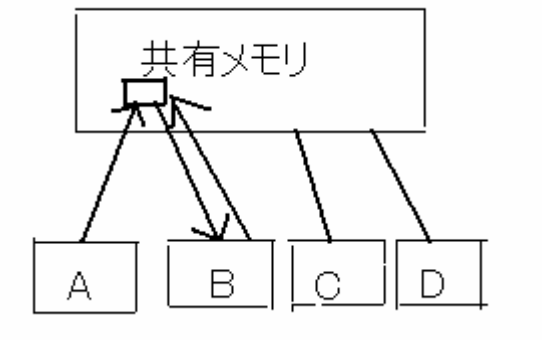

図 2. 共有メモリ型並列計算モデル

1.1.4 PRAM

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

1.1.5 PRAM シミュレータ

1.1.3 でも説明したが、PRAM 自体の実現は非常に困難である。そのため、PRAM アルゴリズムの設計その証明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。従って、PRAM アルゴリズムの実験的な評価を行うために PRAM シミュレータが必要となる。PRAM シミュレータは PRAM 上での並列アルゴリズムの実行をシミュレートし、その実行結果を出力し及びその実行時間を計測する機能を持つプログラムである。

PRAM シミュレータは、以下の 4 つの要素からなる。

① 高級言語である PRAM 用並列言語

JAVA 言語や C 言語などの高級言語に並列処理を行うための命令を加えたものである。

② 低級言語である並列アセンブラ

並列処理を行うための命令を加えたアセンブラである。

③ PRAM コンパイラ

PRAM 用並列言語で記述されたプログラムを並列アセンブラプログラムに変換するコンパイラである。

④ PVSM（Parallel Virtual Stack Machine）

並列アセンブラプログラムを実行できるインタプリタである。

以下の図 3. は PRAM シミュレータの実行の流れを示す。

ユーザは PRAM 用並列言語を用いて PRAM 用並列言語アルゴリズムを記述する。次に PRAM 用並列言語プログラムを PRAM コンパイラを用いて並列アセンブラプログラムに変換し、PVSM を用いて並列アセンブラプログラムを実行する。

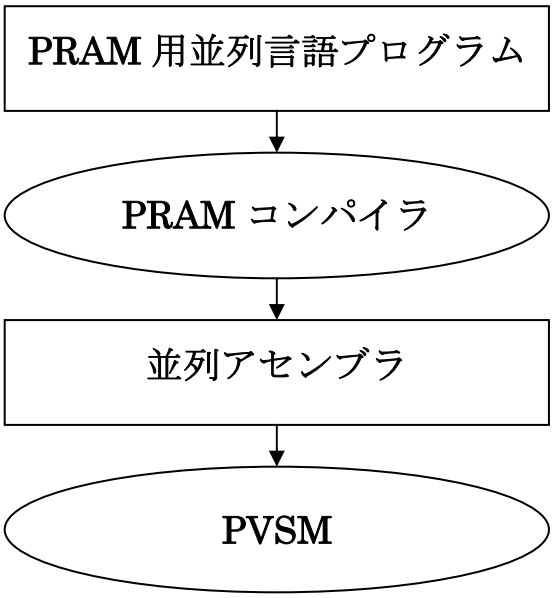

図 3. PRAM シミュレータの実行の流れ

つまり、高級言語である PRAM 用並列言語で書かれたプログラムを並列アセンブラプログラムに変換し、並列アセンブラプログラムを PVSM で実行することにより、ユーザは PRAM 用並列言語プログラムを PRAM 上で動作させた場合の出力および実行時間を計測でき、PRAM アルゴリズムの計算量を実験的に評価することが出来るようになる。

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(Parallel Random Access Machine)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したり、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、本研究で作成した PRAM コンパイラについて述べる。第 3 章では、結果を述べる。PRAM コンパイラの検証を行う。また、いくつかのプログラムを組み実行することにより、作成したコンパイラの正当性を検証する。第 4 章では、3 章の結果を踏まえた上で、結論と今後の課題を示している。

2　研究内容

2.1　PRAM 用並列言語

本研究では、高級言語として付録. 1 に示す k05 言語を拡張した拡張 k05 言語を作成する。
拡張 k05 言語は、高級言語 k05 言語に以下の PRAM 上での並列処理を行う命令“parallel 文”および実行中のプロセッサ番号を表示する特殊記号“$p”を加えたものである。“parallel 文”の文法は以下のように定義される。

parallel (式① , 式②) 文

ここで式①と式②は int 型の評価値を持つ式である。また、ここでの文は“parallel 文”を含まない任意の文である。“parallel 文”は、プロセッサ番号式①から式②までの間のプロセッサを用いて後ろに続く文の並列処理を行う命令である。
また、文中に特殊記号“$p”を記述すると“$p”は実行中のプロセッサ番号の値を持つ変数として処理される。
付録 2 に本研究で作成した拡張 k05 言語の文法を示す。

PRAM 用並列言語プログラムの実行中は、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、逐次状態であり、“parallel 文”を読み並列命令中の文は並列状態として実行され、それ以外の命令は逐次状態として実行される。

2.2　並列アセンブラ

本研究ではアセンブラとして付録 3 に示す VSM アセンブラを作成する。
一般の計算機内部で行われている抽象的なスタックマシンである VSM は Stack、Iseg、Dseg、Pctr、の 4 つの構成要素である。本研究で作成する並列アセンブラは、k05 言語の目的言語として用いられる 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。

○　PARA：並列処理の開始を表す命令である。逐次状態から並列状態に移行する。

逐次状態実行中に PARA 命令を読むことにより、指定した式①と式②のデータがスタックから取り出される。そして、式①から式②へとプロセッサが順に並列に命令を実行していく。また、並列状態に再度 PARA 命令を読みこんだ際、実行エラーとなる。

○　SYNC：並列処理の終了とプロセッサ間の同期を表す命令である。

並列状態実行中にプロセッサ間で同期をとり、逐次状態に移行する。並列状態中に SYNC 命令を読み込んだ際、並列状態実行中の全てのプロセッサが SYNC 命令に到達するまで実行を中断する。全てのプロセッサが SYNC に到達すると、それ以降は逐次状態へと戻り、プロセッサ一台のみが命令を実行する。逐次状態

中に SYNC 命令が読み込まれると実行エラーとなる。

○　PUSHP：プロセッサ番号をスタックに入れる命令である。

スタックに命令を実行しているプロセッサのプロセッサ番号を挿入する。

2.3　PRAM コンパイラ

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。また、コンパイラによる変換工程をコンパイルまたは、翻訳と呼ぶ。

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 用に拡張した拡張 k05 言語を VSM アセンブラに変換する PRAM コンパイラを作成する。

2.3.1　PRAM コンパイラの構成

本研究で作成するコンパイラは、以下の構成から成る。

（1）字句解析部

字句解析部では、入力ファイル中の文字を InputFile クラスを用いてファイルから一文字ずつ読んでいき、k 05 言語のマイクロ構文にしたがって空白文字、コメントなどを読み飛ばしてトークンを最長一致原則に基づいて順次切り出して、構文解析プログラムに渡す役割を持つ。切り出したトークンは int 型のフィールドに格納するものとし、その値とトークンとの対応は Symbol.java に従うものとする。

（2）構文解析部

「abc」という字句の列は、式（Expression）という一つの構文構造にまとめられる。このように構文解析部では、字句の列を構文構造にまとめる。

（3）制約検査部

制約検査部では、整数型とブール型を互いに互換性の無い型として区別するなど型情報を抽出し、演算子や各変数などの一致に対してルールにはまっているか、矛盾が無いかを調べる。

（4）コード生成部

コード生成部では、構文木からオブジェクトコードを生成する。

本研究で作成した PRAM コンパイラは以下のクラスから成る。

・[Lexical Analyzer]：字句解析部であるが、字・記号を読んだ際に判断させるように記憶させた。例として、”p”を読んだ際に“parallel”と判断させるようにした。

・ [Operators]：“PUSHP”“SYNC”などの PRAM 用アセンブラ言語を各々の数字で表し、プログラムを書く際・処理する際の用意化のために拡張した。

・[Symbol]：[Lexical Analyzer]で判断された文字を“symbol”で“parallel 文”“write 文”などを判断させた。

・[Pram]：上記の命令や文の処理のつくりを[Pram]に作成した。

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。

2.3.2 PRAM コンパイラの仕様

以下に本研究で作成した PRAM コンパイラの仕様方法を示す。

[aaa.k]を拡張 k05 言語によって記述された PRAM アルゴリズムとする。以下のコマンドを実行すると、[aaa.k]が VSM アセンブラに変換され、[outputfile]に出力される。[outputfile]を省略した場合は、[OpCode.asm]に出力される。

> **java Pram aaa.k [outputfile]**

PRAM コンパイラにより作成された VSM アセンブラは PVSM インタプリタにより実行される。VSM アセンブラプログラムは以下のコマンドにより実行できる。

> **java PVSM [- c] [aaa.asm]**

[aaa.asm]を PRAM コンパイラが生成した VSM アセンブラプログラムとする。ファイル名を省略した際は OpCode.asm が実行される。また、「オプション –c 」を付けて実行することにより、拡張 k05 言語で書かれた PRAM プログラムを PRAM 上で実行した際の実行次官を測定できる

3　結果及び検証

本研究では、作成したコンパイラの正当性を検証するため、いくつかの PRAM プログラムを作成し、それが正しくコンパイルされるかを検証した。
以下の図 4 は、拡張 k05 言語によるプログラム例を示す。

```
Main () {
        Parallel ( 0 , 15)
              Write( $p ) ;
      }
```

図 4　拡張 k05 言語プログラム

図 4 のプログラムは、プロセッサ番号 0 番から 15 番の 16 台のプロセッサが実行中のプロセッサ番号の値を持つ変数として処理され、並列に実行するというプログラムである。本研究で作成した PRAM コンパイラを用いることにより、図 4 のプログラムは**図 5** に示す VSM アセンブラに変換される。

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

図 5　VSM アセンブラ

図 5 の VSM アセンブラプログラムを PVSM で実行することにより、図 6 の実行結果が得られる。

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

図 6　PVSM の実行結果

図 6 の実行結果より、実行にかかるステップ数が 7 であると計測される。すなわち、図 4 の拡張 k05 言語プログラムを PRAM 上で実行させたときの実行時間が 7 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 コンパイラは PRAM プログラムを正しくコンパイルし、PVSM により並列処理された結果が出力することを確認できた。

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 コンパイラ作成した。本研究で作成した PRAM コンパイラを用いることにより、PRAM アルゴリズムの実行にかかる時間を計測できる。PRAM シミュレータとしての最低限の能力は備えたシミュレータを実現できる。

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は可能にしたが、今後の課題として“parallel 命令”中に“parallel 命令”を実行できるという機能を拡張することが考えられる。に対応させ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、プロセッサ番号を 2 つ指定して、その間全てを順に並列化させているが、指定したプロセッサ番号の間を 1 つとばし、2 つとばしなどで並列化できるような“parallel 文”に対応させることが考えられる。

謝辞

本研究をするにあたり、JAVA 言語、アルゴリズムの基礎から並列処理まで数え切れないほどのご指導、御助言を頂いた石水隆先生には感謝の気持ちで一杯です。また、ご迷惑もたくさんおかけしたと思いますが、この一年間本当にありがとうございました。

そして、同じ情報論理工学研究室の方々には常日頃から助言を賜り、様々な相談にものって頂き、深い感謝、敬愛の気持ちを表します。

参考文献

[1] “情報・コンピュータシステム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 17 年)

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

付録

1　k05 言語の文法

以下に本研究で用いた k05 言語の文法を示す。

<Program>::=<Main_fanction>

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

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

<Statement>::= <If_statement> | <While_statement> | <Assignment>
　　| <Write_statement> | <Writechar_statement>
　　| “{” { <Statement> } “}” | “;”

<If_statement>::= “(” <Expression> “)” <Statement>
<While_statement> :: = “while” “(” <Expression> “)” <Statement>

<Assignment>::= <Lefthand_side> “=” <Expression> “;”
<Writeint_statement>::= “writeint” “(” <Expression> “)” “;”
<Writechar_statement>::= “writechar””(” <Expression> “)” “;”

<Lefthand_side>::= NAME | NAME “[” <Expression> “]”

<Expression> ::= <Logical_term> { “| |“ <Logical_term> }
<Logical_term> ::= <Logical_factor> { “&&“ <Logical_factor> }
<Logical_factor> ::= <Arithmetic_expression>
　| <Arithmetic_expression> “==“ <Arithmetic_expression>
　| <Arithmetic_expression> “!=“ <Arithmetic_expression>
　| <Arithmetic_expression> “<“ <Arithmetic_expression>
　| <Arithmetic_expression> “<=“ <Arithmetic_expression>
　| <Arithmetic_expression> “>“ <Arithmetic_expression>
　| <Arithmetic_expression> “>=“ <Arithmetic_expression>
<Arithmetic_expression> ::= <Arithmetic_term> { (“+“ | “-“) <Arithmetic_term> }
<Arithmetic_term> ::= <Arithmetic_factor> { (“*“ | “/“ | “%“) <Arithmetic_factor> }
<Arithmetic_factor> ::= <Unsigned_factor> | “!“ <Arithmetic_factor>
　| “-“ <Arithmetic_factor>

<Unsigned_factor> ::= NAME | NAME “[” <Expression> “]“ | INT | CHAR
| “(“ <Expression> “)“ | “readint“ | “readchar“ | “true“ | “false“

2 拡張 k05 言語の文法

以下に本研究で作成した拡張 k05 言語の文法を示す。

<Program>::=<Main_fanction>

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

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

<Statement>::= <Parallel_statement> | <If_statement> | <While_statement>
|<Assignment>| <Write_statement>
| <Writechar_statement> | “{” { <Statement> } “}” | “;”

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

<Lefthand_side>::= NAME | NAME “[” <Expression> “]”

<Expression> ::= <Logical_term> { “| |“ <Logical_term> }
<Logical_term> ::= <Logical_factor> { “&&“ <Logical_factor> }
<Logical_factor> ::= <Arithmetic_expression>
| <Arithmetic_expression> “==“ <Arithmetic_expression>
| <Arithmetic_expression> “!=“ <Arithmetic_expression>
| <Arithmetic_expression> “<“ <Arithmetic_expression>
| <Arithmetic_expression> “<=“ <Arithmetic_expression>
| <Arithmetic_expression> “>“ <Arithmetic_expression>
| <Arithmetic_expression> “>=“ <Arithmetic_expression>
<Arithmetic_expression> ::= <Arithmetic_term> { (“+“ | “-“) <Arithmetic_term> }
<Arithmetic_term> ::= <Arithmetic_factor> { (“*“ | “/“ | “%“) <Arithmetic_factor> }
<Arithmetic_factor> ::= <Unsigned_factor> | “!“ <Arithmetic_factor>

| “-“ <Arithmetic_factor>

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

3 VSM アセンブラの文法

以下に本研究用いた VSM アセンブラの文法を示す。

```
ASSGN:
        addr = Stack [--SP];
        Dseg[addr] = Stack[SP] = Stack [SP+1];
ADD:    BINOP(+);
SUM:    BINOP(-);
MUL:    BINOP(*);
DIV:
        If (Stack[SP] == 0)
        {
                printf(“Zero divider detected¥n”);
                return -2;
        }
        BINOP(¥);
MOD:
        If (Stack[SP] == 0)
        {
                printf(“Zero divider detected¥n”);
                return -2;
        }
        BINOP(%);
CSIGN: Stack [SP] = -Stack[SP];
AND:    BINOP(&&);
OR:     BINOP(||);
NOT:    Stack [SP] =   !Stack [SP];
COPY:   ++SP;   Stack [SP] = Stack [SP-1];
PUSH:   Stack [++SP] = Dseg[addr];
PUSHI: Stack [++SP] = addr;
REMOVE: --SP;
POP:    Dseg[addr] = Stack[SP--];
INC:    Stack [SP] = ++ Stack [SP];
DEC:    Stack [SP] = -- Stack [SP];
COMP:
```

```
            Stack [SP-1] = Stack [SP-1] > Stack [SP] ? 1:
            Stack [SP-1] < Stack [SP] ? -1 0;
            SP--;
    BLT:    if (Stack [SP--] <  0) Pctr = addr;
    BLE:    if (Stack [SP--] <= 0) Pctr = addr;
    BEQ:    if (Stack [SP--] == 0) Pctr = addr;
    BNE:    if (Stack [SP--] !=  0) Pctr = addr;
    BGE:    if (Stack [SP--] >= 0) Pctr = addr;
    BGT:    if (Stack [SP--] >  0) Pctr = addr;
    JUMP:   Pctr = addr;
    HALT:   return 0;
    INPUT:  scanf("%d%*c", &Stack[++SP]);
    INPUTC:scanf("%c%*c", &Stack[++SP]);
    OUTPUT:printf("%15d¥n", Stack [SP--]
    OUTPUTC:printf("%15c¥n", Stack [SP--]
    LOAD:   Stack [SP] = Dseg[Stack [SP]];
```

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。本研究で作成したプログラムは以下の構成から成る。

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

（１）Pram.java

```
import ioTools.*;

public class Pram implements Operators, Symbol, Type, TruthValue
 {
    static LexicalAnalyzer lexer;
```

```
static VarTable vt;
static InstractionSegment iseg;

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

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

        lexer.nextToken();
        parseProgram();

        lexer.inFile.closeFile();

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

// プログラム文
public static void parseProgram()
{
        parseMain_function();
        iseg.appendCode(HALT);
}

// メイン文
public static void parseMain_function()
```

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

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

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

        parseBlock();
}

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

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

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

        lexer.nextToken();
}

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

```
{
        type = lexer.ttype;
        if(type==S_INT)type=T_INT;
        if(type==S_BOOLEAN)type=T_BOOL;
        lexer.nextToken();
}
else syntaxError();

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

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

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

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

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

```
            if(type==T_ARRAYOFBOOL)type=T_BOOL;

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

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

                    if(type==T_BOOL)type=T_ARRAYOFBOOL;

            lexer.nextToken();

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

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

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

```
//Statement 文の作成
public static void parseStatement()
{
switch (lexer.ttype)
    {

        case S_IF:
                    parseIf_statement();
                    break;

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

       case S_NAME:
                    parseAssignment();
                    break;

       case S_PARALLEL:
                    parseParallel_statement();
                    break;

      case S_PROCESSOR:
            parseExpression();
            lexer.nextToken();
            break;

        case S_SEMICOLON:
            lexer.nextToken();
            break;

      case S_WHILE:      /* while 文  */
                    parseWhile_statement();
```

```
                    break;

        case S_WRITE:       /* write 文 */
                parseWrite_statement();
                break;

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

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

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

            default:
                syntaxError();
                break;
    }
}

//parallel 文の作成
        public static void parseParallel_statement()
{
        if (lexer.ttype==S_PARALLEL) lexer.nextToken();
                else syntaxError();

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

        parseExpression();

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

        parseExpression();
```

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

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

//If 文の作成
public static void parseIf_statement()
{
        int ktest=-1;

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

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

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

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

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

```
//While 文の作成
public static void parseWhile_statement()
{
        int ktest=-1;

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

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

        int ad1=iseg.isegPtr;

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

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

//Assignment 文の作成
public static void parseAssignment()
{
        int ktest1=-1;
        int ktest2=-1;
        ktest1=parseLefthand_side();

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

```
        ktest2=parseExpression();

        if((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
        ( ktest2==T_INT || ktest2==T_ARRAYOFINT)){}

        else if((ktest1==T_BOOL || ktest1==T_ARRAYOFBOOL) &&
        (ktest2==T_BOOL || ktest2==T_ARRAYOFBOOL)){}

        else
    {
                syntaxError();
    }

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

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



//write 文の作成
public static void parseWrite_statement()
{
        int ktest=-1;
        if (lexer.ttype==S_WRITE) lexer.nextToken();
        else syntaxError();

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

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

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

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

        switch (ktest) {
        case T_INT:

        case T_CHAR:
                iseg.appendCode (OUTPUTC);
                break;

                case T_DOUBLE:
                iseg.appendCode (OUTPUTD);
                break;

                case T_BOOL:
                iseg.appendCode (OUTPUTB);
                break;

                case T_ARRAYOFINT:
                iseg.appendCode (OUTPUT);
                break;

        case T_STRING:
                iseg.appendCode (OUTPUTS);
                break;

        default:
                syntaxError ();
                break;
        }
}

//Writechar 文の作成
public static void parseWritechar_statement()
{
        int ktest=-1;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        if (lexer.ttype==S_NAME)
    {
                if(vt.exist(lexer.name)==false)
        {
                syntaxError();
                }
                ktest = vt.getType(lexer.name);
```

```
                iseg.appendCode(PUSHI,vt.getAddress(lexer.name));
        lexer.nextToken();
    }
        else syntaxError();

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

public static void parseLefthand_side2(int ktest)
{
        if(lexer.ttype==S_LBRACKET)
        {
        if(ktest==T_INT||ktest==T_BOOL)
                {
                        syntaxError();
                }
        lexer.nextToken();
        }
        else syntaxError();
        parseExpression();
        iseg.appendCode(ADD);
        if(lexer.ttype==S_RBRACKET)
    {
                lexer.nextToken();
    }
        else syntaxError();
}
```

```
public static int parseExpression()
{
        int ktest1=-1;
        int ktest2=-1;
        ktest1=parseLogical_term();

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

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

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

                ktest2=parseLogical_factor();

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

```
    }
        return ktest1;
}

public static int parseLogical_factor()
{
        int betype =-1;
        int pctr =0;
        int ktest1 = -1;
        int ktest2 = -1;

        ktest1=parseArithmetic_expression();
        betype=lexer.ttype;

        if(betype==S_EQUAL||betype==S_NOTEQ||betype==S_LESS||
        betype==S_LESSEQ||betype==S_GREAT||betype==S_GREATEQ){

        switch (betype)
    {
        case S_EQUAL:
                lexer.nextToken();

                ktest2=parseArithmetic_expression();

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

```
    case S_NOTEQ:
            lexer.nextToken();

            ktest2=parseArithmetic_expression();

    if((((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
        ( ktest2==T_INT || ktest2==T_ARRAYOFINT))||
        ((ktest1==T_BOOL || ktest1==T_ARRAYOFBOOL) &&
        (ktest2==T_BOOL || ktest2==T_ARRAYOFBOOL)))==false)

        {
            syntaxError();
        }
    else
        {
                ktest1=T_BOOL;
        }
    pctr =iseg.appendCode(COMP);
    iseg.appendCode(BNE,pctr+4);
            break;

case S_LESS:
    lexer.nextToken();


    ktest2=parseArithmetic_expression();

    if((((ktest1==T_INT || ktest1==T_ARRAYOFINT) &&
        ( ktest2==T_INT || ktest2==T_ARRAYOFINT))||
       ((ktest1==T_BOOL || ktest1==T_ARRAYOFBOOL) &&
        (ktest2==T_BOOL || ktest2==T_ARRAYOFBOOL)))==false)
        {
            syntaxError();
        }
    else
        {
                ktest1=T_BOOL;
        }
    pctr =iseg.appendCode(COMP);
```

```
        iseg.appendCode(BLT,pctr+4);
        break;

    case S_LESSEQ:
        lexer.nextToken();


        ktest2=parseArithmetic_expression();

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

    case S_GREAT:
        lexer.nextToken();

        ktest2=parseArithmetic_expression();

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

```
                }
            pctr =iseg.appendCode(COMP);
            iseg.appendCode(BGT,pctr+4);
            break;

        case S_GREATEQ:
            lexer.nextToken();

            ktest2=parseArithmetic_expression();

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

        default:
            break;
        }
iseg.appendCode(PUSHI, 0);
iseg.appendCode(JUMP,pctr+5);
iseg.appendCode(PUSHI, 1);
}
return ktest1;
}

public static int parseArithmetic_expression()
{
            int betype=-1;
            int ktest1 = -1;
```

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

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

                default:
                        break;
            }
    }
        return ktest1;
}

public static int parseArithmetic_term()
{
        int betype=-1;
```

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

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

        case S_DIV:
            lexer.nextToken();

            ktest2=parseArithmetic_factor();

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

        case S_MOD:
            lexer.nextToken();

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

```
                                {
                                    syntaxError();
                                }
                        iseg.appendCode(MOD);
                        break;

                default:
                        break;
                }
        }
return ktest1;
}

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

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

        case S_SUB:
                lexer.nextToken();

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

                break;

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

```
public static int parseUnsigned_factor()
{
        String name = "";
        int ktest = -1;

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

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

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

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

    case S_LPAREN:
        lexer.nextToken();
```

```
        ktest = parseExpression();

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

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

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

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

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

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

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

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

    parseExpression();

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

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

（２）LexicalAnalyzer.java

```
    public class LexicalAnalyzer implements Symbol {
```

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


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

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


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

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

    if (c == '¥0') ttype = S_EOF;                   /* End of file */
    else if (c == '0') {                            /*  符号無し整数(0) */
        value = 0;
        ttype = S_INTEGER;
    } else if (Character.isDigit(c)) {              /*  符号無し整数  */
        value = extractIntValue(c);
        ttype = S_INTEGER;
    } else if (Character.isLowerCase(c) || Character.isUpperCase(c)
                || c=='_') {
        String str = extractWord(c);
        switch (c) {
        case 'b':
```

```
        if (str.equals("boolean")) ttype = S_BOOLEAN;/* boolean */
        else ttype = S_NAME;                                    /*  変数名  */
        break;

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

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

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

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

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

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

case 't':
        if (str.equals("true")) ttype = S_TRUE;          /* true */
```

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

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

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

        default:
            ttype = S_NAME;                                    /*  変数名  */
            break;
        }
        name = str;
    } else {
        switch(c) {
        case '(':
            ttype = S_LPAREN;                                    /* ( */
            break;
        case ')':
            ttype = S_RPAREN;                                    /* ) */
            break;
        case '{':
            ttype = S_LBRACE;                                    /* { */
            break;
        case '}':
            ttype = S_RBRACE;                                    /* } */
            break;
        case '[':
            ttype = S_LBRACKET;                                    /* [ */
            break;
        case ']':
            ttype = S_RBRACKET;                                    /* ] */
```

```
        break;
    case ',':
        ttype = S_COMMA;                    /* , */
        break;
    case ';':
        ttype = S_SEMICOLON;                /* ; */
        break;
    case '+':
        ttype = S_ADD;                      /* + */
        break;
    case '-':
        ttype = S_SUB;                      /* - */
        break;
    case '*':
        ttype = S_MUL;                      /* * */
        break;
    case '/':
        ttype = S_DIV;                      /* / */
        break;
    case '%':
        ttype = S_MOD;                      /* % */
        break;
    case '=':
        if (inFile.nextc == '=') {          /* == */
            inFile.nextChar();
            ttype = S_EQUAL;
        } else ttype = S_ASSIGN;            /* = */
        break;
    case '<':
        if (inFile.nextc == '=') {          /* <= */
            inFile.nextChar();
            ttype = S_LESSEQ;
        } else ttype = S_LESS;              /* < */
        break;
    case '>':
        if (inFile.nextc == '=') {          /* >= */
            inFile.nextChar();
            ttype = S_GREATEQ;
        } else ttype = S_GREAT;             /* < */
```

```
        break;
    case '!':
        if (inFile.nextc == '=') {                          /* != */
            inFile.nextChar();
            ttype = S_NOTEQ;
        } else ttype = S_NOT;                                   /* ! */
        break;
    case '&':
        if(inFile.nextc == '&') {                           /* && */
            inFile.nextChar();
            ttype = S_AND;
        } else syntaxError();
        break;
    case '|':
        if (inFile.nextc == '|') {                          /* || */
            inFile.nextChar();
            ttype = S_OR;
        } else syntaxError();
        break;
    case '¥':                                                   /* 文字 */
        c = inFile.nextChar();
        if (c != '¥' && inFile.nextc == '¥') {
            inFile.nextChar();
            ttype = S_CHARACTER;
            value = (int) c;
        } else syntaxError();
        break;

    case '$':
        if (inFile.nextc == 'p') {                          /* $p */
                inFile.nextChar();
                ttype = S_PROCESSOR;
        } else syntaxError();
        break;

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

```
            }
            return ttype;
        }

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

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

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

（３）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
}
```

（4）Symbol.java

```
interface Symbol {
    static final int S_ERROR = -1;
    static final int S_NULL = 0;
    static final int S_MAIN = 1;            /* main */
    static final int S_IF = 2;              /* if */
    static final int S_ELSE = 3;            /* else */
    static final int S_WHILE = 4;           /* while */
    static final int S_FOR = 5;             /* for */
    static final int S_DO = 6;              /* do */
    static final int S_READINT = 7;         /* readint */
    static final int S_READCHAR = 8;        /* readchar */
    static final int S_READDOUBLE = 9;    /* readdouble */
    static final int S_READSTR = 10;        /* readstr */
```

```
static final int S_WRITE = 11;          /* write */
static final int S_WRITEINT = 12;     /* writeint */
static final int S_WRITECHAR = 13;    /* writechar */
static final int S_WRITEDOUBLE = 14;/* writedouble */
static final int S_WRITESTR = 15;     /* writestr */
static final int S_WRITELN = 16;      /* writeln */
static final int S_RAND = 17;          /* rand */
static final int S_INT = 18;           /* int */
static final int S_CHAR = 19;          /* char */
static final int S_BOOLEAN = 20;       /* boolean */
static final int S_DOUBLE = 21;        /* double */
static final int S_STRING = 22;        /* String */
static final int S_EQUAL = 23;         /* == */
static final int S_NOTEQ = 24;         /* != */
static final int S_LESS = 25;          /* < */
static final int S_GREAT = 26;         /* > */
static final int S_LESSEQ = 27;        /* <= */
static final int S_GREATEQ = 28;       /* >= */
static final int S_AND = 29;           /* && */
static final int S_OR = 30;            /* || */
static final int S_NOT = 31;           /* ! */
static final int S_ADD =32;            /* + */
static final int S_SUB = 33;           /* - */
static final int S_MUL = 34;           /* * */
static final int S_DIV = 35;           /* / */
static final int S_MOD = 36;           /* % */
static final int S_ASSIGN = 37;        /* = */
static final int S_ADDLHS = 38;        /* += */
static final int S_SUBLHS = 39;        /* -= */
static final int S_MULLHS = 40;        /* *= */
static final int S_DIVLHS = 41;        /* /= */
static final int S_MODLHS = 42;        /* %= */
static final int S_INC = 43;           /* ++ */
static final int S_DEC = 44;           /* -- */
static final int S_SEMICOLON = 45;    /* ; */
static final int S_LPAREN = 46;        /* ( */
static final int S_RPAREN = 47;        /* ) */
static final int S_LBRACE = 48;        /* { */
static final int S_RBRACE = 49;        /* } */
```

```
    static final int S_LBRACKET = 50;     /* [ */
    static final int S_RBRACKET = 51;     /* ] */
    static final int S_COMMA = 52;        /* , */
    static final int S_INTEGER = 53;      /*  整数  */
    static final int S_CHARACTER =54;     /*  文字  */
    static final int S_NAME = 55;         /*  変数名  */
    static final int S_TRUE = 56;         /* true */
    static final int S_FALSE = 57;        /* false */
    static final int S_DOUBLEVAL = 58;    /*  実数  */
    static final int S_STR= 59;           /*  文字列  */
    static final int S_PARALLEL = 60;     /* parallel */
    static final int S_PROCESSOR = 61;    /* $p */
    static final int S_EOF = 255;         /* end of file */
}
```

（５）Var.java

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

    public Var(int t, String n, int a, int s) {
            type = t;
        name = n;
        address = a;
        size = s;
    }
}
```

（６）VarTable.java

```
import java.util.Vector;

public class VarTable {
    Vector vt;
    int nextAddress;

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

```
}

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

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

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

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

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

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

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

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

（7）inputFile.java

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

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

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

```
nextChar();
}

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

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

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

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

    public static void main(String[] args) {
    InputFile inFile = new InputFile("bsort.k");
    do {
        do {
            inFile.nextChar();
            System.out.print(inFile.currentc);
        } while(inFile.currentc != '¥n' && inFile.currentc != '¥0');

        */
        System.out.println(inFile.line);
        inFile.readInputFile();
        */
    } while(inFile.line != null);
    }
}
```

（８）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
            }
    }
}
```

（９）InstractionSegment.java

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

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

```
boolean debugSW;

public InstractionSegment(boolean dsw) {
        iseg = new ArrayList();
        isegPtr = 0;
        size = 0;
        debugSW = dsw;
}

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

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

```
                inst.stringOperand = "";
                inst.register = false;
                if (debugSW) System.out.println ("-> "+inst);
        }
        isegPtr++;
        return isegPtr-1;
}

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

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

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

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

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

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

```
		} else {
			Instraction inst = ((Instraction) iseg.get (isegPtr));
			replace (isegPtr, inst, operator, Integer.MAX_VALUE, doubleOperand, "", register);
		}
		isegPtr++;
		return isegPtr-1;
}

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

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

```
public int operator (int addr) {          /*  オペレータを返す  */
        if (addr < 0 || addr >= size)
                executeError("Illegal iseg address ", addr);
        return ((Instraction) iseg.get (addr)).operator;
}

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

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

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

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

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

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

```
        return ((Instraction) iseg.get (addr)).register;
}

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

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

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

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

```
        outputFile.close();
}

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

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

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

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

// addr 番目の命令の オペレータ，オペランド を operator, doubleOperand に変更する
public void replaceCode (int addr, int operator, double doubleOperand) {
```

```
        Instraction inst = ((Instraction) iseg.get (addr));
        replace (addr, inst, operator, Integer.MAX_VALUE, doubleOperand, "", inst.register);
}

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

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

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

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

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

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

（１０）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;
}
```

（１１）TruthValue.java

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