卒業研究報告書

題目

PRAM シミュレータの構築

指 導 教 員 石水隆 助手

報告者

01-1-26-037

橋本 博之

近畿大学理工学部電気工学学科

平成 18 年 2 月 10 日提出

目次

第 1 章　序論

1.1 並列処理と並列アルゴリズム・・・・・・・・・・・・・・・・・・・・・・・1
1.2 並列計算機・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・2
1.3 並列計算モデル・・・・・・・・・・・・・・・・・・・・・・・・・・・・2
1.4 PRAM シミュレータ・・・・・・・・・・・・・・・・・・・・・・・・・・2
1.5 本報告書の構成・・・・・・・・・・・・・・・・・・・・・・・・・・・・3

第 2 章　準備

2.1 PRAM・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・4

2.2 PRAM シミュレータ・・・・・・・・・・・・・・・・・・・・・・・・・・5

2.3 コンパイラ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・6

第 3 章　実験方法

3.1 PRAM シミュレータ・・・・・・・・・・・・・・・・・・・・・・・・・・9

3.1.1 PRAM 用並列言語・・・・・・・・・・・・・・・・・・・・・・・・・・9

3.2 並列アセンブラ・・・・・・・・・・・・・・・・・・・・・・・・・・・・9

3.3 PRAM コンパイラ ・・・・・・・・・・・・・・・・・・・・・・・・・ 10
3.3.1 PRAM コンパイラの構成・・・・・・・・・・・・・・・・・・・・・・10

3.3.2 PRAM コンパイラの仕様・・・・・・・・・・・・・・・・・・・・・・11

第 4 章　結果

4.1 PRAM コンパイラの正当性・・・・・・・・・・・・・・・・・・・・・・12

4.2 PRAM シミュレータの性能・・・・・・・・・・・・・・・・・・・・・・12

第 5 章　考察

5.1 PRAM シミュレータの性能 ・・・・・・・・・・・・・・・・・・・・・・・・14

5.2 PRAM 用並列言語の性能 ・・・・・・・・・・・・・・・・・・・・・・・・・14

第 6 章　結論

6.1 結論 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・15

謝辞 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・16
参考文献 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・17
付録 1 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・18
付録 2・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・20
付録 3・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・116

第 1 章　序論 [1)]

1.1 並列処理と並列アルゴリズム

今日、様々な分野で高速な計算機を用いて計算量の大きな問題を短時間で解くことが求められている。具体的な例としては次のようなものがある。

(1) 量子力学、統計力学、相対論的物理学
(2) 宇宙論と宇宙物理学
(3) 計算流体力学と乱流
(4) 生物学、薬物学、ゲノム配列、遺伝子工学、蛋白質の合成、酸素の働き、細胞のモデル化
(5) 医学、人間の臓器と骨格のモデル化
(6) 大域的な気象と環境のモデル化

高速な計算機が必要となるその他の応用としては、物質探査、経済の計画、暗号解析、地震学、航空機のテスト、原子・核・プラズマ物理学などの数値シミュレーション、ロボテックス、大規模データベースの管理、実時間音声認識、人間と計算機の高度なインターフェースなどがある。

計算速度を向上させるための方法としては、性能の良いプロセッサ(Processoer)を用いるか、あるいは並列処理(Parallel Processing)を行うかの 2 通りが考えられる。近年、プロセッサは高度に集積化され、計算速度の目覚しい向上が成し遂げられてきた。しかし、真空中の高速が 3.0×10^8 m/sec であるという物理的制約により、この傾向が間もなく終わるということを予想している。一方、並列処理にはそのような限界は本質的に存在しない。このため、並列処理に対して大きな期待が寄せられている。

並列処理とは、ある問題を解く際、その問題をより小さい複数の部分問題に分割し、その部分問題を複数のプロセッサが強調して解く処理である。並列処理を行うことにより、単一のプロセッサによる逐次処理よりも高速に問題を解くことができ、またより複雑な問題を解くことが出来る。また、並列処理には次に挙げるような利点もある。

(1) 多くの問題に対して並列の解法の方がごく自然である。
(2) 近年、計算機素子の価格とサイズが急激に下がっており、多数のプロセッサを用いた並列計算機が作り易くなっている。
(3) 並列処理では、問題または問題クラスを解くのに最も適した並列アーキテクチャを運ぶことが可能である。

しかし、並列処理を行うためには、プロセッサ間のデータのやり取りやメモリへのアクセス、プロセッサ間の同期等、並列特有の問題を解決せねばならない。このため、従来の逐次処理で用いられてきた逐次アルゴリズムをそのまま並列処理に用いることはできず、並列処理専用のアルゴリズム、すなわち並列アルゴリズム(Parallel Algorithm)が必要とな

る。

1.2 並列計算機

複数のプロセッサを持ち、並列処理を行える計算機が並列計算機(Parallel Computer)である。並列計算機は、全てのプロセッサが共通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はメモリを通し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)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易いため、通信およ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できる。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繋ぐことは困難となる。このため、現在、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繋ぎ、それ全体を仮想的な計算機として扱うクラスタ(Cluster)処理やグリッド(Grid)処理も幅広く行われている。

1.3 並列計算モデル

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、全ての命令がその移動に関係無く 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、等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史や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。しかし、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あるため、PRAM 自体の実現は困難である。

1.4 PRAM シミュレータ

PRAM 自体の実現は困難であるため、PRAM 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。そのため、PRAM アルゴリズムの実験的な評価を行うために PRAM シミュレータ(PRAM Simulater)が必要となる。

PRAM シミュレータは、高級言語である PRAM 用並列言語を低級言語である並列アセンブラに変換する PRAM コンパイラと、並列アセンブラを実行する VPSM(Virtual Parallel Stack Machine)インタプリタから成る。

本研究では、JAVA 言語を用いて PRAM シミュレータの一部である PRAM コンパイラの設計を行う。本研究で設計した PRAM コンパイラは、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することができる。並列アセンブラプログラムを VPSM インタプリタで実行することにより、PRAM 用並列言語プログラムを PRAM 上で動作させた場合の出力および実行時間を計測でき、PRAM アルゴリズムの計算量を実験的に評価することができるようになる。

1.5 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、PRAM シミュレータおよびコンパイラを定義する。

第 3 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 4 章では、PRAM シミュレータの検証を行う。また、これらを用いて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。また、和計算プログラムを組み、PRAM 用並列言語プログラムを PRAM 上で動作させた場合の実行時間および計算量を計測し、理論値と比較する。

第 5 章では、第 4 章で得た結果を踏まえて PRAM 用並列言語および PRAM シミュレータの性能を考察する。

第 6 章では、本報告書のまとめを述べる。

また、付録 1 に PRAM 用並列言語の文法を、付録 2 に PRAM コンパイラプログラムを、付録 3 に n 個のデータの和計算プログラムを示す。

第 2 章　準備

2.1 PRAM[2)]

図 1 のように、複数のプロセッサがメモリを共有したコンピュータを共有メモリ型並列計算機(Shared Memory Parallel Computer)と呼ぶ。代表的な並列計算モデルであるPRAM(Parallel Random Access Machine)は共有メモリ型並列計算機を抽象化したモデルである。

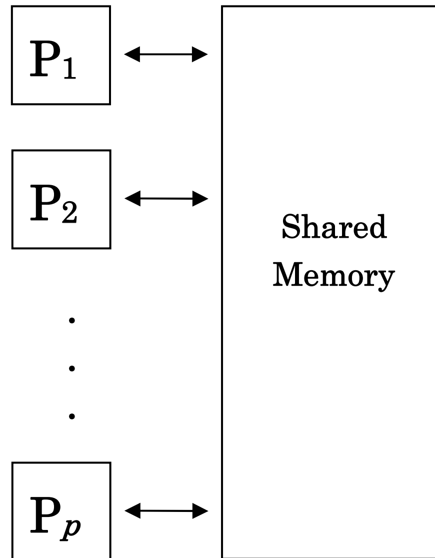

図 1　共有メモリ型並列計算機

PRAM は共有メモリとそれに接続された p 個のプロセッサからなる。並列アルゴリズムの実行中、p 個のプロセッサは入力データの読み出し、中間結果の読みだし、および書き込み、最終結果の書き込みをするために、共有メモリにアクセスする。PRAM 各プロセッサは、共有メモリ上の任意の位置にあるメモリセルに対して 1 単位時間でデータの読み書きができ、また全ての演算は 1 単位時間で行なうことができると仮定されている。また、1 単位時間ごとに全てのプロセッサで同期がとられる完全同期型モデルである。

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 が以下の 4 つに分類される。

(1) EREW(Exclusive-Read Exclusive-Write)PRAM　排他読み出し排他書き込み
　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) CREW(Concurrent-Read Exclusive-Write)PRAM　同時読み出し排他書き込み
　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。

⑶ ERCW(Exclusive-Read Concurrent-Write)PRAM 排他読み出し同時書き込み
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。

⑷ CRCW(Concurrent-Read Concurren-Write)PRAM 同時読み込み同時書き込み
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。

さらに、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。

(i) 優先型(Priority) CRCW PRAM
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。

（ⅱ） 任意型(Arbitrary)CRCW PRAM
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。

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

2.2 PRAM シミュレータ

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および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 要素から成る.

⑴ PRAM 用並列言語
⑵ 並列アセンブラ
⑶ PRAM コンパイラ
⑷ VPSM(Virtual Parallel Stack Machine)インタプリタ

PRAM 用並列言語は、JAVA や C 等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り、並列アセンブラは並列処理を行うための命令を加えたアセンブラである。PRAM コンパイラは、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り、VPSM インタプリタは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。図 2 に PRAM シミュレータの実行の流れを示す。
ユーザは PRAM 用並列言語を用いて PRAM 用並列言語アルゴリズムを記述する。次に PRAM 用並列言語プログラムを PRAM コンパイラを用いて並列アセンブラプログラムに変換し、VPSM 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。

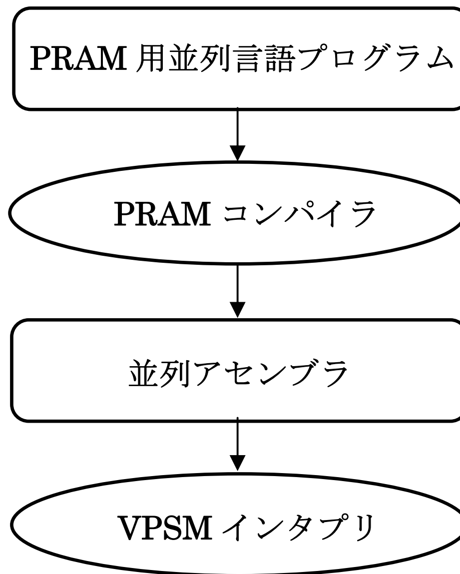

図 2　PRAM シミュレータによる実行の流れ

2.3 コンパイラ [3)]

コンパイラとは、高水準言語(high level programming language)で書かれたプログラムを、計算機が実現可能な形式へ変換する変換系である。この変換をコンパイル(compile)または、翻訳という。本実験で使用した PRAM コンパイラの基本構造を図 3 に示す。コンパイラの内部処理は、図 3 のようなフェーズ(phase)で分けられる。以下にコンパイラの仕様を示す。

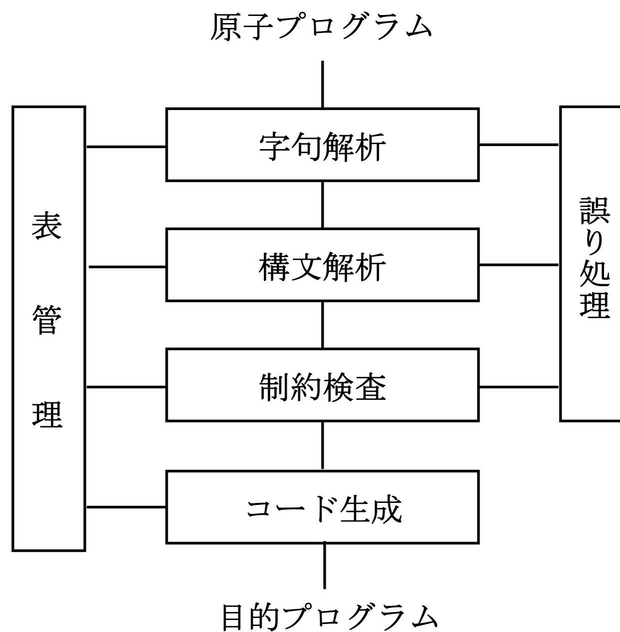

図 3　コンパイラの基本構造

実際にコンパイラを作成しようとすると、原始言語と目的言語の仕様が厳密に定義されている必要がある。まず、原始言語の仕様を説明する。コンパイラへの入力は文字列であるので、コンパイラは、まず、文字列が正しい文であるかを調べ、文の構造を決定しなくてはならない。そのために必要な規則を構文、またはシンタクス(Syntax)という。また構文で記述されない部分は、制約規制(Contextual Constraint)として別に定義される。(制約規則は静的意味論(Static Semantics)と呼ばれることもある。)

入力が原始言語の正しい文であることがわかれば、次にその文がどのような実行を表しているのかがわからなくてはならない。その与え方をその言語の意味論、または、セマンティクス(Semantics)という。

目的言語の仕様も構文と意味に分けられる。目的言語がアセンブリ言語の場合には、命令のビットパターンやニーモニックは、構文で指定される。命令の意味は、レジスタ転送表現(RTL, Register Transfer Language)などで表されていることが多い。これは機械語命令を実行したときに、レジスタや主記憶間でどのようにデータが移動するかを記述したものである。

なお、原始言語の構文は、コンパイラの効率のために、実際上二つに分けられていることが多い。本報告書では、これらをマクロ構文(Macro Syntax)、マイクロ構文(Micro Syntax)と呼ぶ。マイクロ構文は、プログラムを文字列とみなして、字句、またはトークン(Token)と呼ばれるプログラムを構成する基本的な要素の構文を指定しているものである。以下に表管理および誤り処理の仕様を述べる。また、コンパイラの仕様と各フェーズとの対応を図 4 に示す。

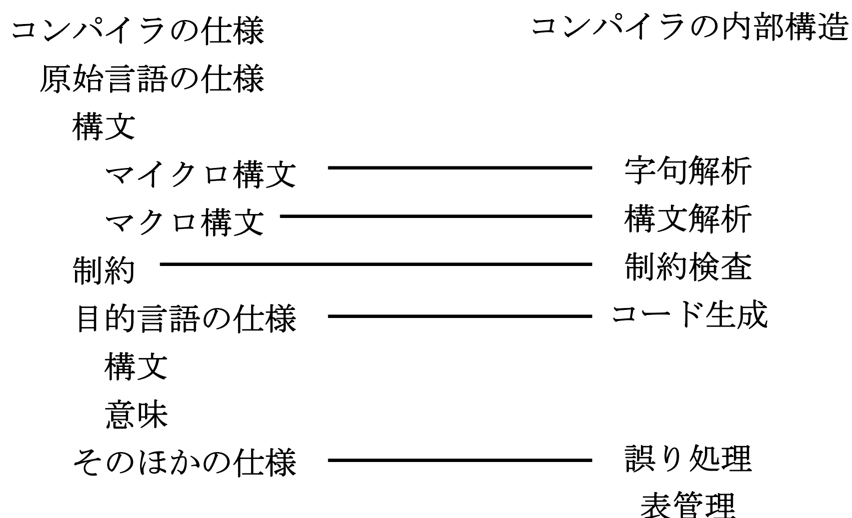

図 4 コンパイラの仕様とフェーズ

表管理(table management, bookkeeping)の仕様は、原始プログラム中で用いられた名前を覚えておき、型情報などのその名前に付随する情報を記録しておくことである。この情

報を管理するためのデータ構造を記号表(symbol table)という。

誤り処理(error handling)の仕様は、原始プログラムが構文を満足しない場合や、型などに制約規則違反がある場合に呼び出されて、プログラマに警告を発する働きをする。

第 3 章　実験方法

3.1 PRAM シミュレータ

3.1.1 PRAM 用並列言語

本研究で使用する PRAM 用並列言語は、C 風の手続き型言語である。この言語は、KC05[4]に並列処理を行う **parallel** 文および実行中のプロセッサ番号を表す特殊変数 **$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 (***exp1*** **,** ***exp2, exp3*** **)** **statement**

exp1、***exp2*** および ***exp3*** は **int** 型の評価値を持つ式であり、**statement** は **parallel** 文を含まない任意の文である。**parallel** 文を実行すると、プロセッサ番号 ***exp1*** から ***exp2*** までの間のプロセッサで ***exp3*** おきのプロセッサ P_{exp1}, $P_{exp1+exp3}$, $P_{exp1+2*exp3}$, …, P_{exp2} が後に続く **statement** を並列に実行する。また、**statement** 中に特殊変数 **$p** を記述すると **$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。

PRAM 用並列言語プログラムは、プロセッサ P_0 のみが命令を実行する逐次モードと、**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モードの 2 つのモードを持つ。プログラム開始時は逐次モードであり、**parallel** 命令中の **statement** は並列モードとして実行され、それ以外の命令は逐次モードとして実行される。

付録[1]に本研究で使用する PRAM 用並列の文法を示す。

3. 2 並列アセンブラ

本研究で使用する並列アセンブラは、命令部オペレータと値部オペランドから成る 2 個組命令で構成される。この言語は、KC05 言語の目的言語として用いられるアセンブラの命令セットに以下の命令を加えたものである。

PUSHP：プロセッサ番号挿入命令。スタックに命令を実行しているプロセッサのプロセッサ番号を挿入する。

PARA　：並列処理開始命令。逐次モードから並列モードに移行する。

逐次モード実行中に PARA 命令を読み込んだ場合、スタックから 2 個のデータ d_1,d_2 が取り出される。この命令以降は、

d_1d_2 プロセッサ **P_{d1}, P_{d1+1}, P_{d1+2}, …, P_{d2}** が並列に命令を実行する。並列モ

ードに PARA 命令を読み込んだ場合、実行エラーとなる。

SYNC　：同期命令。並列モード実行中のプロセッサ間で同期を取り、逐次モードに移行する。逐次モード中に SYNC 命令を読み込んだ場合、並列モード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、それ以降はプロセッサ P_0 のみが命令を実行する。逐次モード中に SYNC 命令が読み込まれると実行時エラーとなる。

3.3 PRAM コンパイラ

本研究では、JAVA 言語を用いて PRAM コンパイラの設計を行った。付録 2 に本研究で作成した PRAM コンパイラプログラムを示す。

3.3.1 PRAM コンパイラの構成

本研究で作成したコンパイラは、以下の構成からなる。

(1) 字句解析部(Lexical　Analyzer)

字句解析部では、原子プログラムを読み、それを字句(Token)の列に変換する。また、同時にプログラムの意味に影響を与えない空白(Tab)、コメント(Coment)等を読み飛ばす。

(2) 構文解析部(Parser)

構文解析部では、字句の列を構文構造にまとめる。例えば、[234]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。式は周りの字句やそのほかの構文構造とまとめられて文(Statement)が形成される。

(3) 制約検査部(Constraint Checker)

制約検査部では、プログラムの宣言部から型情報などを抽出し、各変数や演算子の使用に対して矛盾が無いかを調べる。

(4) コード生成部(Code Generator)

コード生成部では命令部オペレータとアドレス部オペランドの 2 個組コードである並列アセンブラコードの生成を行う。

3.3.2 PRAM コンパイラの仕様

以下に本研究で作成した PRAM コンパイラの仕様を示す。

(PRAM コンパイラの使用方法)

foo.pram を PRAM 用並列アルゴリズム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、foo.pram が並列アセンブラに変換され、outputfile に出力される。outputfile を省略した場合は、OpCode.asm に出力される。

> java Pram [- *option*] foo.pram [outputfile]

foo.asm を並列アセンブラ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、PRAM アルゴリズムの PRAM 上での実行がシミュレートされる。foo.asm を省略した場合は、OpCode.asm を入力ファイルとして実行する。

> java VSM [- *option*] foo.asm

(PRAM コンパイラのオプション)

PRAM コンパイラは実行時に以下のオプションを指定できる。

d： PRAM コンパイラをデバグモードで実行する。

t： PRAM コンパイラをトレスモードで実行する。

n： PRAM 用並列言語プログラムの文法チェックのみを行う。

o： コンパイル後の並列アセンブラプログラムを表示する。

v： コンパイル後に変数表を表示する。

第 4 章　結果

4..1 PRAM コンパイラの正当性

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。以下にプログラム例を示す。

```
main () {
        parallel (0, 10, 1)
                writeint ($p);
}
```

上記のプログラム例は､プロセッサ番号0番から10番の11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。VPSM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と以下のように出力される。

0 1 2 3 4 5 6 7 8 9 10

以上の結果から､本研究で作成した PRAM コンパイラは正しくコンパイルされ､並列処理された結果が出力することを確認した。

4..2 PRAM シミュレータの性能

本研究作成したシミュレータの正当性を和計算プログラムを用いて検証を行う。検証に用いたプログラムを付録 3 に示し、検証結果を表 1 に示す。

表 1　和計算プログラムの実行時間

プロセッサ数(p)	実行時間(t)
2	10497
4	5356
8	2807
16	1554
32	949
64	668
128	549

表 1 は、256 個のデータでプロセッサ数を変えた時の実行時間を計った結果である。本研究で得られた実行結果(表 1 参照)からプロセッサ数と実行時間の関係を示すため、プロセッサ数を x 軸に、実行時間を y 軸にして、その関係を示したグラフを図 5 に示す。また、図 5 に理論値の線を引き本研究での実行時間と比較した。実行時間と理論値の値を比較すると、ほぼ等しい結果であることを確認した。本研究で作成した和計算プログラムは、正しく動作をしている。

本研究で作成した PRAM シミュレータの性能は、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できる。

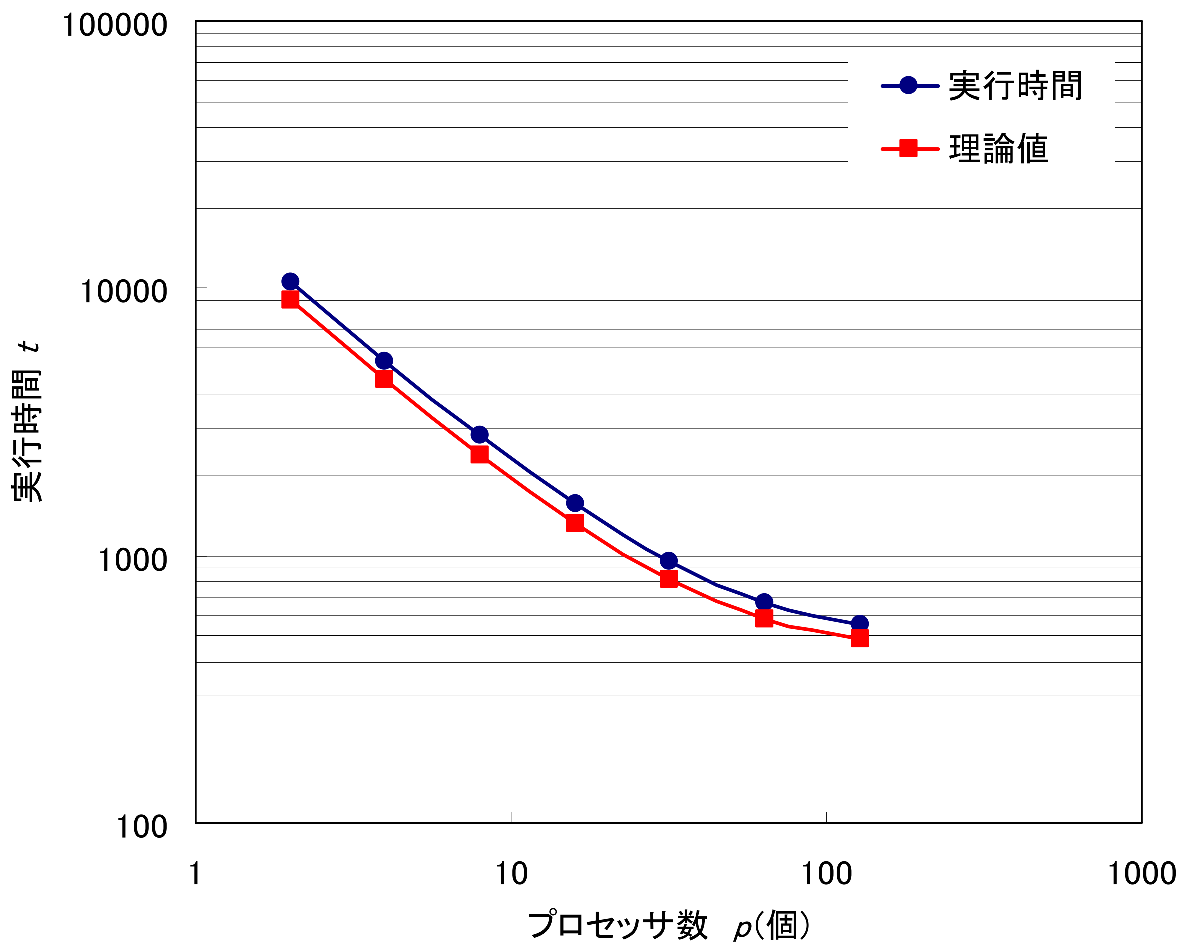

図 5 PRAM シミュレータによる和計算プログラムの実行時間

第 5 章　考察

本研究で作成した PRAM コンパイラを用いることにより、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。しかし、本研究で定義した PRAM 用並列言語は C 言語と比べて関数が扱えない事を含め、文法の制限が多く、適応範囲が狭い言語となっている。

5.1 PRAM シミュレータの性能

また、本研究で作成したシミュレータは入力ファイルとしては同じディレクトリにあるファイル名のみ、出力ファイルも同じディレクトリにあるファイル名のみ認めるため、使用者が好むファイル名を自ら指定したり、入力や出力の際に他のディレクトリから参照したりすることはできない。

5.2 PRAM 用並列言語の性能

本研究で作成した PRAM 用並列言語とは C 風の手続き型言語に並列処理をするための命令 parallel を加えたものであり、基本的な命令はほぼ C 言語と同一のものである。しかし、本研究で作成した PRAM 用並列言語は適応範囲が C 言語と比べて低いため、関数に対応していないなどの使い勝手の悪さが残る。この問題を解決するためには本研究で作成した PRAM 用並列言語の適応範囲を広めるために、仕様をさらに改善していく必要がある。

第 6 章　結論

6.1 結論

本研究では、PRAM アルゴリズムを記述できる PRAM 用並列言語の提案、及び PRAM 用並列言語プログラムの PRAM 上での動作をシミュレートできるシミュレータを作成した。

本研究で作成したシミュレータは PRAM 用並列言語を並列アセンブラに変換して、さらに PRAM 用並列言語の PRAM 上の実行時間を計測することができ、PRAM シミュレータとしての最低限の能力は備えている。しかし、本研究で作成した PRAM 用並列言語は C 言語と比べて関数が扱えない事を含め、文法の制限が多く、適応範囲が狭い言語となっている。また、本研究で作成したシミュレータは入力ファイルとしては同じディレクトリにあるファイル名のみ、出力ファイルとしても同じディレクトリにあるファイル名のみ認めるため、使用者が好むファイル名を自ら指定したり、入力や出力の際に他のディレクトリから参照したりすることはできない。

今後の課題としては PRAM 用並列言語の適応範囲をさらに広げるために、より拡張し、その拡張した PRAM 用並列言語に対応できるシミュレータを作成することが挙げられる。

さらに、入力や出力の際に別のディレクトリから参照できるようにすることも挙げられる。

謝辞

研究をするにあたり、アルゴリズムの基礎から並列処理についてまで、数え切れないほどの、御指導や御助言など大変尽力を尽くしていただき、石水隆助手には、誠に感謝申し上げます。また、同じ研究室の皆には色々とお世話になり、感謝申し上げます。

参考文献

1) 渋沢進：“並列分散処理入門”、培風館、東京、1~5、(1998)
2) 渋沢進：“並列分散処理入門”、培風館、東京、39、46、48~50、(1998)
3) 辻野嘉宏：“コンパイラ”、昭晃堂、東京、(2000)
4) 加藤暢：“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、(2005)
5) 疋田輝雄・石畑清：“コンパイラの理論と実現”、共立出版、東京、(1992)
6) Joseph JáJá：”An Introduction to Parallel Algorithms”, Addison-Wesley(1992)

付録 1 PRAM 用並列言語の文法

本研究で作成した PRAM 用並列言語の文法を以下に示す。

<Program>：：= <Main_function>

<Main_function>：：= “main” “(“ “)” <Block>

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

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

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

<If_statement> ：：= “if” “(“ <Expression> “)” <Statement>

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

<Assignment>：：= <Lefthand_side> “=” <Expression> “;”

<Writeint_statement>：：= “writeint” “(“ <Expression> “)” “;”

<Writechar_statement>：：= “writechar” “(“ <Expression> “)” “;”

<Parallel_statement> ：：= “parallel” “(“ <Expression>, <Expression>,
<Expression> “)” <Statement>

<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”

付録 2 PRAM コンパイラプログラム

本研究で用いた PRAM コンパイラプログラムを以下に示す。本研究で作成したプログラムは以下の構成から成る。

(1) inputFile.java

(2) Instraction.java

(3) InstractionSegment.java

(4) LexicalAnalyzer.java

(5) Operators.java

(6) Symbol.java

(7) Type.java

(8) Var.java

(9) VarTable.java

(10) Kc.java

```
/* 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;
            }
        }
    }
```

/* InputFile.java ここまで　*/

/* Instraction.java */

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

     public Instraction(int i, int j) {
         operator = i;
         operand = j;
         register = false;
     }

     public Instraction(int i) {
         operator = i;
         operand = -1;
         register = false;
     }
```

```
public Instraction(int i, int j, boolean r) {
        operator = i;
        operand = j;
        register = r;
}

public Instraction(int i, boolean r) {
        operator = i;
        operand = -1;
        register = r;
}

public String toString() {
        switch(operator) {
        case PUSH:
        case PUSHI:
        case POP:
        case BEQ:
        case BNE:
        case BLE:
        case BLT:
        case BGE:
        case BGT:
        case JUMP:
        case CALL:
                return opName() + "¥t" + operand + "¥t";
```

```
                        /* オペランドを持つ場合 */
        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 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 INPUTS:   return "INPUTS ";      // input string
            case OUTPUT:   return "OUTPUT ";      // output integer
            case OUTPUTC:  return "OUTPUTC";      // output character
            case OUTPUTL:  return "OUTPUTL";      // output line
            case OUTPUTS:  return "OUTPUTS";      // output string
            case RAND:     return "RAND   ";      // random
            case HALT:     return "HALT   ";      // halt
            case ASSGNS:   return "ASSGNS ";      // assign string
            case LOADS:    return "LOADS  ";      // load stirng
            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
            }
        }
}
/* Instraction.java ここまで */


/* 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.Vector; //ベクトル用
```

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

   boolean debugSW;

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

    // Iseg に命令を加える
   public int appendCode(Instraction inst) {
          if (size == isegPtr) {
                 if (debugSW) System.out.println(isegPtr+": "+inst);
                 iseg.addElement(inst);
                 size++;
          } else {
                 Instraction oldInst = ((Instraction) iseg.get(isegPtr));
                 if (debugSW) System.out.print(isegPtr+": "+oldInst);
                 iseg.removeElementAt(isegPtr);
                 iseg.insertElementAt(inst, isegPtr);
                 if (debugSW) System.out.println("-> "+inst);
```

```
        }
        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.addElement(inst);
            size++;
        } else {
            Instraction inst = ((Instraction) iseg.get(isegPtr));
            if (debugSW) System.out.print(isegPtr+": "+inst);
            inst.operator = operator;
            inst.operand = operand;
            inst.register = register;
            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.addElement(inst);
            size++;
        } else {
            Instraction inst = ((Instraction) iseg.get(isegPtr));
            if (debugSW) System.out.print(isegPtr+": "+inst);
            inst.operator = operator;
            inst.operand = -1;
            inst.register = register;
            if (debugSW) System.out.println("-> "+inst);
        }
        isegPtr++;
        return isegPtr-1;
    }

    public int appendCode(int operator, int operand) {
        if (size == isegPtr) {
            Instraction inst = new Instraction(operator, operand);
            if (debugSW) System.out.println(isegPtr+": "+inst);
            iseg.addElement(inst);
            size++;
        } else {
            Instraction inst = ((Instraction) iseg.get(isegPtr));
            if (debugSW) System.out.print(isegPtr+": "+inst);
            inst.operator = operator;
            inst.operand = operand;
```

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

    public int appendCode(int operator) {
        if (size == isegPtr) {
            Instraction inst = new Instraction(operator);
            if (debugSW) System.out.println(isegPtr+": "+inst);
            iseg.addElement(inst);
            size++;
        } else {
            Instraction inst = ((Instraction) iseg.get(isegPtr));
            if (debugSW) System.out.print(isegPtr+": "+inst);
            inst.operator = operator;
            inst.operand = -1;
            inst.register = false;
            if (debugSW) System.out.println("-> "+inst);
        }
        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 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();
	}

	// 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));
		if (debugSW)
			System.out.print(addr + ": " + inst);
		inst.operator = operator;
		inst.operand = operand;
		if (debugSW)
			System.out.println("-> " + inst);
	}

	// 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));
```

```
        if (debugSW)
              System.out.print(addr + ": " + inst);
        inst.operand = operand;
        if (debugSW)
              System.out.println("-> " + inst);
}

// 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.removeElementAt(addr);
        iseg.insertElementAt(inst, addr);
        if (debugSW) System.out.println("-> "+inst);
}

public 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);
}

public 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.exit(1);
            }
        }
        /* InstoractionSegment.java ここまで */


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

                //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, boolean tsw) {
                    //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;
                    traceSW = tsw;
                }
```

```
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') {
        if ('0' <= inFile.nextc && inFile.nextc <= '7') {  /* 符号無
            し 8 進整数 */
            c = inFile.nextChar();
            value = extractOctIntValue(c);
        } else if (inFile.nextc == 'x') {     /* 符号無し 16 進整数 */
            inFile.nextChar();
            c = inFile.nextChar();
            value = extractHexIntValue(c);
        } else value = 0;                 /* 符号無し整数(0) */
        ttype = S_INTEGER;
    } else if (Character.isDigit(c)) {       /* 符号無し 10 進整数 */
        value = extractIntValue(c);
        ttype = S_INTEGER;
    } else if (Character.isLowerCase(c) || Character.isUpperCase(c)
        || c=='_') {
        String str = extractWord(c);
        switch (c) {
        case 'p':
```

```
if (str.equals("parallel")) ttype = S_PARALLEL;/* parallel */
        else ttype = S_NAME;
        break;
case 'b':
if (str.equals("boolean")) ttype = S_BOOLEAN;/* boolean */
        else ttype = S_NAME;
        break;
case 'c':
        if (str.equals("char")) ttype = S_CHAR;     /* char */
        else ttype = S_NAME;
        break;
case 'd':
        if (str.equals("do")) ttype = S_DO;           /* do */
        else ttype = S_NAME;
        break;
case 'e':
        if (str.equals("else")) ttype = S_ELSE;     /* else */
        else ttype = S_NAME;
        break;
case 'f':
        if (str.equals("false")) ttype = S_FALSE;  /* false */
        else if (str.equals("for")) ttype = S_FOR;   /* for */
        else ttype = S_NAME;
        break;
case 'i':
        if (str.equals("if")) ttype = S_IF;           /* if */
```

```
        else if (str.equals("int")) ttype = S_INT;    /* int */
        else ttype = S_NAME;                          /* 变数名 */
        break;
    case 'm':
        if (str.equals("main")) ttype = S_MAIN;      /* main */
        else ttype = S_NAME;                          /* 变数名 */
        break;
    case 'r':
        if (str.equals("rand")) ttype = S_RAND;      /* rand */
        else if (str.equals("readint"))
            ttype = S_READINT;                        /* readint */
        else if (str.equals("readchar"))
            ttype = S_READCHAR;                       /* readchar */
        else if (str.equals("readstr"))
            ttype = S_READSTR;                        /* readstr */
        else ttype = S_NAME;                          /* 变数名 */
        break;
    case 't':
        if (str.equals("true")) ttype = S_TRUE;      /* true */
        else ttype = S_NAME;
        break;
    case 'w':
        if (str.equals("while")) ttype = S_WHILE;   /* while */
        else if (str.equals("write"))
            ttype = S_WRITE;                          /* write */
        else if (str.equals("writeint"))
```

```
                ttype = S_WRITEINT;                    /* writeint */
            else if (str.equals("writechar"))
                ttype = S_WRITECHAR;                   /* writechar */
            else if (str.equals("writeln"))
                ttype = S_WRITELN;                     /* writeln */
            else if (str.equals("writestr"))
                ttype = S_WRITESTR;                    /* writestr */
            else ttype = S_NAME;                       /* 变数名 */
            break;
        default:
            ttype = S_NAME;                            /* 变数名 */
            break;
        }
        name = str;
    } else {
        switch(c) {
        case '(':
            ttype = S_LPAREN;                          /* ( */
            break;
        case ')':
            ttype = S_RPAREN;                          /* ) */
            break;
        case '{':
            ttype = S_LBRACE;                          /* { */
            break;
        case '}':
```

```
        ttype = S_RBRACE;                   /* } */
        break;
    case '[':
        ttype = S_LBRACKET;                 /* [ */
        break;
    case ']':
        ttype = S_RBRACKET;                 /* ] */
        break;
    case ',':
        ttype = S_COMMA;                    /* , */
        break;
    case ';':
        ttype = S_SEMICOLON;                /* ; */
        break;
    case '+':
        if (inFile.nextc == '=') {          /* += */
            inFile.nextChar();
            ttype = S_ADDLHS;
        } else if (inFile.nextc == '+') {   /* ++ */
            inFile.nextChar();
            ttype = S_INC;
        } else ttype = S_ADD;               /* + */
        break;
    case '-':
        if (inFile.nextc == '=') {          /* -= */
            inFile.nextChar();
```

```
                        ttype = S_SUBLHS;
                } else if (inFile.nextc == '-') {       /* -- */
                        inFile.nextChar();
                        ttype = S_DEC;
                } else ttype = S_SUB;                   /* - */
                break;
        case '*':
                if (inFile.nextc == '=') {              /* *= */
                        inFile.nextChar();
                        ttype = S_MULLHS;
                } else ttype = S_MUL;                   /* * */
                break;
        case '/':
                if (inFile.nextc == '=') {               /* /= */
                        inFile.nextChar();
                        ttype = S_DIVLHS;
                } else if (inFile.nextc == '*') {  * 注釈はスキップ */
                        inFile.nextChar();
                        c = inFile.nextChar();
                        do {
                                c = inFile.nextChar();
                                if (c == '¥0') syntaxError("Illegal end
of file");
                        } while (!(c == '*' && inFile.nextc == '/'));
                        inFile.nextChar();
                        ttype = nextToken();
                } else if (inFile.nextc == '/') { /* 注釈はスキップ */
```

```
                inFile.nextChar();
                c = inFile.nextChar();
                do {
                        c = inFile.nextChar();
                } while (c != '¥n' && c != '¥0');
                ttype = nextToken();
        } else ttype = S_DIV;                                   /* / */
        break;
case '%':
        if (inFile.nextc == '=') {                              /* /= */
                inFile.nextChar();
                ttype = S_MODLHS;
        } else 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("Illegal character : &");
        break;
case '|':
        if (inFile.nextc == '|') {                          /* || */
                inFile.nextChar();
                ttype = S_OR;
        } else syntaxError("Illegal character : |");
        break;
case '¥'':                                                  /* 文字 */
        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");

        if (inFile.nextc == '¥'') {
            inFile.nextChar();
            ttype = S_CHARACTER;
            value = (int) c;
        } else syntaxError("Illegal character : " + c);
        break;
    case '"' :
        String str = "";
        boolean valid = true;
        c = inFile.nextChar();
        while (c != '"') {
            if (valid) {
                if (c == '¥¥' ) {
                    c = inFile.nextChar();
                    switch(c) {
                    case '0':
                        c = '¥0';
                        valid = false;
                        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    ==    '¥0')
syntaxError("Illegal end of file");
                            if(valid) str += c;
                        }
                        c = inFile.nextChar();
                    }
                    ttype = S_STRING;
                    string = str;
```

```
                        break;
                case '$':
                        if (inFile.nextc == 'p') {                      /* $p */
                                inFile.nextChar();
                                ttype = S_PROCESSOR;
                        } else syntaxError("Illegal character : |");
                        break;
                default:
                        syntaxError("Illegal character : " + c);
                        break;
                }
        }
        if (traceSW) System.out.println(token());
        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 int extractOctIntValue(char c) {     /* c で始まる８進整数を得る */
```

```
		int v = Character.digit(c, 8);		/* 文字を整数に変換 */
		while (('0' <= inFile.nextc && inFile.nextc <= '7')) {
			c = inFile.nextChar();
			v = v * 8 + Character.digit(c, 8);
		}
		return v;
	}

	public int extractHexIntValue(char c) {	/* c で始まる 16 進整数を得る */
		int v = Character.digit(c, 16);		/* 文字を整数に変換 */
		while (('0' <= inFile.nextc && inFile.nextc <= '9') || ('a' <=
inFile.nextc && inFile.nextc <= 'f')
			|| ('A' <= inFile.nextc && inFile.nextc <= 'F')) {
			c = inFile.nextChar();
			v = v * 16 + Character.digit(c, 16);
		}
		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 String token() {
        switch(ttype) {
        case S_NULL:
                return "Null";
        case S_MAIN:
                return "main";
        case S_IF:
                return "if";
        case S_ELSE:
                return "else";
        case S_WHILE:
                return "while";
        case S_FOR:
                return "for";
        case S_DO:
                return "do";
        case S_READINT:
                return "readint";
        case S_READCHAR:
                return "readchar";
        case S_READSTR:
                return "readstr";
```

```
case S_WRITE:
        return "write";
case S_WRITEINT:
        return "writeint";
case S_WRITECHAR:
        return "writechar";
case S_WRITELN:
        return "writeln";
case S_WRITESTR:
        return "writestr";
case S_RAND:
        return "rand";
case S_INT:
        return "int";
case S_CHAR:
        return "char";
case S_BOOLEAN:
        return "boolean";
case S_EQUAL:
        return "==";
case S_NOTEQ:
        return "!=";
case S_LESS:
        return "<";
case S_GREAT:
        return ">";
```

```
case S_LESSEQ:
        return "<=";
case S_GREATEQ:
        return ">=";
case S_AND:
        return "&&";
case S_OR:
        return "||";
case S_NOT:
        return "!";
case S_ADD:
        return "+";
case S_SUB:
        return "-";
case S_MUL:
        return "*";
case S_DIV:
        return "/";
case S_MOD:
        return "%";
case S_ASSIGN:
        return "=";
case S_ADDLHS:
        return "+=";
case S_SUBLHS:
        return "-=";
```

```
case S_MULLHS:
        return "*=";
case S_DIVLHS:
        return "/=";
case S_MODLHS:
        return "%=";
case S_INC:
        return "++";
case S_DEC:
        return "--";
case S_SEMICOLON:
        return ";";
case S_LPAREN:
        return "(";
case S_RPAREN:
        return ")";
case S_LBRACE:
        return "{";
case S_RBRACE:
        return "}";
case S_LBRACKET:
        return "[";
case S_RBRACKET:
        return "]";
case S_COMMA:
        return ",";
```

```
    }
        case S_INTEGER:
            return "" + value;
        case S_CHARACTER:
            return "'" + (char) value + "'";
        case S_NAME:
            return name;
        case S_TRUE:
            return "true";
        case S_FALSE:
            return "false";
        case S_STRING:
            return "¥"" + string + "¥"";
        case S_EOF:
            return "EndOfFile";
        default:
            return "Error";
        }
    }

    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);
        }
    }
/* LexicalAnalyzer.java ここまで */


/* 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 REMOVE  = 21;  // remove
static final int LOAD    = 22;  // load
static final int POP     = 23;  // pop
static final int INC     = 24;  // ++
static final int DEC     = 25;  // --
static final int PREINC  = 26;  // 前置++
static final int PREDEC  = 27;  // 前置--
static final int POSTINC = 28;  // 後置++
static final int POSTDEC = 29;  // 後置--
static final int SETFR   = 30;  // set frame register
static final int INCFR   = 31;  // inc frame register
static final int DECFR   = 32;  // dec frame register
static final int JUMP    = 33;  // jump
static final int BLT     = 34;  // < ?
static final int BLE     = 35;  // <= ?
static final int BEQ     = 36;  // == ?
static final int BNE     = 37;  // != ?
static final int BGE     = 38;  // > ?
static final int BGT     = 39;  // >= ?
```

```
        static final int CALL    = 40;  // call
        static final int RET     = 41;  // return
        static final int INPUT   = 42;  // input integer
        static final int INPUTC  = 43;  // input character
        static final int OUTPUT  = 44;  // output integer
        static final int OUTPUTC = 45;  // output character
        static final int OUTPUTL = 46;  // output line
        static final int RAND    = 47;  // random
        static final int HALT    = 48;  // halt
        static final int PUSHS   = 49;  // push string
        static final int POPS    = 50;  // pop string
        static final int ASSGNS  = 51;  // assign string
        static final int LOADS   = 52;  // load string
        static final int INPUTS  = 53;  // input string
        static final int OUTPUTS = 54;  // output string
        static final int PARA    = 55;  // parallel
        static final int SYNC    = 56;  // synchronous
        static final int PUSHP   = 57;  // push processor number
        static final int EOF     =255;  // end of file
        static final int ERROR   = -1;  // error
}
/* Operators.java ここまで */


/* 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_WRITE = 9;       /* write */
static final int S_WRITEINT = 10;   /* writeint */
static final int S_WRITECHAR = 11;  /* writechar */
static final int S_WRITELN = 12;    /* writeln */
static final int S_RAND = 13;       /* rand */
static final int S_INT = 14;        /* int */
static final int S_CHAR = 15;       /* char */
static final int S_BOOLEAN = 16;    /* boolean */
static final int S_EQUAL = 17;      /* == */
static final int S_NOTEQ = 18;      /* != */
static final int S_LESS = 19;       /* < */
static final int S_GREAT = 20;      /* > */
static final int S_LESSEQ = 21;     /* <= */
static final int S_GREATEQ = 22;    /* >= */
static final int S_AND = 23;        /* && */
static final int S_OR = 24;         /* || */
static final int S_NOT = 25;        /* ! */
```

```
static final int S_ADD =26;         /* + */
static final int S_SUB = 27;        /* - */
static final int S_MUL = 28;        /* * */
static final int S_DIV = 29;        /* / */
static final int S_MOD = 30;        /* % */
static final int S_ASSIGN = 31;     /* = */
static final int S_ADDLHS = 32;     /* += */
static final int S_SUBLHS = 33;     /* -= */
static final int S_MULLHS = 34;     /* *= */
static final int S_DIVLHS = 35;     /* /= */
static final int S_MODLHS = 36;     /* %= */
static final int S_INC = 37;        /* ++ */
static final int S_DEC = 38;        /* -- */
static final int S_SEMICOLON = 39;  /* ; */
static final int S_LPAREN = 40;     /* ( */
static final int S_RPAREN = 41;     /* ) */
static final int S_LBRACE = 42;     /* { */
static final int S_RBRACE = 43;     /* } */
static final int S_LBRACKET = 44;   /* [ */
static final int S_RBRACKET = 45;   /* ] */
static final int S_COMMA = 46;      /* , */
static final int S_INTEGER =47;     /* 整数 */
static final int S_CHARACTER =48;   /* 文字 */
static final int S_NAME = 49;       /* 変数名 */
static final int S_TRUE = 50;       /* true */
static final int S_FALSE = 51;      /* false */
```

```
        static final int S_STRING = 52;     /* 文字列 */
        static final int S_READSTR = 53;    /* readstr */
        static final int S_WRITESTR = 54;   /* writestr */
        static final int S_PARALLEL = 55;   /* parallel */
        static final int S_PROCESSOR = 56;  /* $p */
        static final int S_EOF = 255;       /* end of file */
}
/* Symbol.java ここまで */


/* Type.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_WRITE = 9;       /* write */
        static final int S_WRITEINT = 10;   /* writeint */
        static final int S_WRITECHAR = 11;  /* writechar */
        static final int S_WRITELN = 12;    /* writeln */
        static final int S_RAND = 13;       /* rand */
```

```
static final int S_INT = 14;        /* int */
static final int S_CHAR = 15;       /* char */
static final int S_BOOLEAN = 16;    /* boolean */
static final int S_EQUAL = 17;      /* == */
static final int S_NOTEQ = 18;      /* != */
static final int S_LESS = 19;       /* < */
static final int S_GREAT = 20;      /* > */
static final int S_LESSEQ = 21;     /* <= */
static final int S_GREATEQ = 22;    /* >= */
static final int S_AND = 23;        /* && */
static final int S_OR = 24;         /* || */
static final int S_NOT = 25;        /* ! */
static final int S_ADD =26;         /* + */
static final int S_SUB = 27;        /* - */
static final int S_MUL = 28;        /* * */
static final int S_DIV = 29;        /* / */
static final int S_MOD = 30;        /* % */
static final int S_ASSIGN = 31;     /* = */
static final int S_ADDLHS = 32;     /* += */
static final int S_SUBLHS = 33;     /* -= */
static final int S_MULLHS = 34;     /* *= */
static final int S_DIVLHS = 35;     /* /= */
static final int S_MODLHS = 36;     /* %= */
static final int S_INC = 37;        /* ++ */
static final int S_DEC = 38;        /* -- */
static final int S_SEMICOLON = 39;  /* ; */
```

```
        static final int S_LPAREN = 40;     /* ( */
        static final int S_RPAREN = 41;     /* ) */
        static final int S_LBRACE = 42;     /* { */
        static final int S_RBRACE = 43;     /* } */
        static final int S_LBRACKET = 44;   /* [ */
        static final int S_RBRACKET = 45;   /* ] */
        static final int S_COMMA = 46;      /* , */
        static final int S_INTEGER =47;     /* 整数 */
        static final int S_CHARACTER =48;   /* 文字 */
        static final int S_NAME = 49;       /* 変数名 */
        static final int S_TRUE = 50;       /* true */
        static final int S_FALSE = 51;      /* false */
        static final int S_STRING = 52;     /* 文字列 */
        static final int S_READSTR = 53;    /* readstr */
        static final int S_WRITESTR = 54;   /* writestr */
        static final int S_PARALLEL = 55;   /* parallel */
        static final int S_PROCESSOR = 56;  /* $p */
        static final int S_EOF = 255;       /* end of file */
}
/* Type.java ここまで */


/* Var.java */
    public class Var implements Type {
        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) {
        type = t;
        name = n;
        address = a;
        size = 1;
}

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

public String toString() {
        return typeName()+"¥t"+name +"¥t"+address;
}

public String typeName() {
        switch (type) {
        case T_INT:           return "int";
        case T_ARRAYOFINT:    return "int["+size+"]";
        case T_CHAR:          return "char";
        case T_ARRAYOFCHAR:   return "char["+size+"]";
```

```
            case T_BOOL:        return ″boolean″;
            case T_ARRAYOFBOOL: return ″boolean[″+size+″]″;
            default:            return ″unknown″;
            }
        }
}
```

/* Var.java */

/* VarTable.java */

```
    import java.util.Vector;

    public class VarTable {
        Vector vt;
        int nextAddress;
        boolean debugSW;

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

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

```
		vt.addElement (var);
		if (debugSW) System.out.println(var);
		nextAddress ++;
		return true;
	}

	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 */
		Var var = new Var(t, n, nextAddress, s);
		vt.addElement (var);
		if (debugSW) System.out.println(var);
		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 void dump() {			/* 表を表示 */
		for (int i=0; i<vt.size(); i++)
			System.out.println((Var) vt.get(i));
	}
}
/* VarTable.java ここまで */
```

```
/* Kc.java */
    import ioTools.*;

    public class Kc implements Operators, Symbol, Type {
        static LexicalAnalyzer lexer;                 /* 字句解析器 */
        static VarTable vt;                           /* 変数の名前表 */
        static InstractionSegment iseg;               /* 命令列 */
        static int expType;                           /* 式の型 */
        static boolean leftValue;                     /* 左辺値 */
        static int arraySize;                         /* 配列のサイズ */
        static boolean parallel;                      /* parallel の中ですか */

        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 statSW = false;           /* 実行データの表示 */
        static boolean debugSW = false;          /* デバッグモード */
        static boolean symPrtSW = false;         /* 記号表の表示 */
        static boolean varOutSW = false;         /* 変数表の表示 */

        static String sourceFile;
        static String objectFile;

        public static void main(String[] args) {
            setUpOption(args);
```

```
        lexer = new LexicalAnalyzer(sourceFile, traceSW);
                                             /* 字句解析器 */
        vt = new VarTable(debugSW);              /* 変数の名前表 */
        iseg = new InstractionSegment(debugSW); /* 命令列 */
        parallel = false;
        parseProgram();                          /* プログラム解析へ */

        lexer.inFile.closeFile();  /* 解析が終了したらファイルを閉じる */

        if (execSW) {
            if(objectFile == null)
                iseg.dumpToFile();               /* オブジェクトファ
イル出力 デフォルト名 */
                else iseg.dumpToFile(objectFile);/* オブジェクトファ
イル出力 名前指定 */
        }
        if (objOutSW) iseg.dump();               /* 目的コード表示 */
        if (varOutSW) vt.dump();                 /* 変数表 */
        System.out.println("Compile finished");
    }

    public static void parseProgram() {          /* プログラム解析 */
        if (lexer.nextToken() != S_MAIN) syntaxError("Need main");
        parseMain();                             /* main 関数解析へ */
        if (lexer.ttype != S_EOF) syntaxError("Illegal token");
    }
```

```
public static int parseMain() {            /* main 関数 */
	if (lexer.nextToken() != S_LPAREN) syntaxError("Need (");
	if (lexer.nextToken() != S_RPAREN) syntaxError("Need )");
	if (lexer.nextToken() != S_LBRACE) syntaxError("Need {");
	parseBlock();
	iseg.appendCode(HALT);                  /* 終了(停止)命令 */
	return lexer.ttype;
}

public static int parseBlock() {           /* ブロック */
	if (lexer.nextToken() != S_RBRACE) {
		if (lexer.ttype == S_INT || lexer.ttype == S_CHAR
			|| lexer.ttype == S_BOOLEAN)
			parseVarDeclList();
		if (lexer.ttype != S_RBRACE) parseStatementList();
		else lexer.nextToken();
	} else lexer.nextToken();
	return lexer.ttype;
}

public static int parseVarDeclList() {     /* 変数宣言の並び */
	while (lexer.ttype == S_INT || lexer.ttype == S_CHAR
		|| lexer.ttype == S_BOOLEAN) {
		int type = lexer.ttype;
```

```
                parseVarDecl(type);
                while (lexer.ttype == S_COMMA) parseVarDecl(type);

                if (lexer.ttype != S_SEMICOLON) syntaxError("Need ;");
                lexer.nextToken();
            }
            return lexer.ttype;
        }

    public static int parseVarDecl(int type) {  /* 変数宣言 */
            String name;
            if (lexer.nextToken() != S_NAME) syntaxError("Illegal token");

            if (vt.exist(lexer.name)) syntaxError(lexer.name + " is already
defined");
                                                  /* 名前の重複チェック */

            name = lexer.name;
            if (lexer.nextToken() == S_LBRACKET) {  /* 配列型の場合 */
                if (lexer.nextToken() != S_INTEGER) syntaxError("Illegal
Token");
                int size = lexer.value;
                if (lexer.nextToken() != S_RBRACKET) syntaxError("Need [");
                lexer.nextToken();
                if (type == S_INT) type = T_ARRAYOFINT;
                else if (type == S_CHAR) type = T_ARRAYOFCHAR;
                else type = T_ARRAYOFBOOL;
```

```
vt.addElement(type, name, size);    /* 名前表に登録 */
if (lexer.ttype == S_ASSIGN) {
    int addr = vt.getAddress(name);
    lexer.nextToken();
    if (type == T_ARRAYOFINT || type == T_ARRAYOFBOOL)
        parseInitialAssignArray(type, addr, size);
                                    /* 整数型論理型は配列代
入へ */
    else if (lexer.ttype == S_LBRACE)   /* 文字型配列の場
合 */
        parseInitialAssignArray(type, addr, size);
                                    /* { があれば
配列代入へ */
    else if (lexer.ttype == S_STRING) { /* 文字列ならば文
字列代入 */
        String str = lexer.string;
        if  (str.length()>size-1)  syntaxError("Array
index is out of bound");
        for(int i=0; i<str.length(); i++) {
            iseg.appendCode(PUSHI, addr+i);
            iseg.appendCode(PUSHI, str.charAt(i));
            iseg.appendCode(ASSGN);
            iseg.appendCode(REMOVE);
        }
        iseg.appendCode(PUSHI, addr+str.length());
        iseg.appendCode(PUSHI, '¥0');
        iseg.appendCode(ASSGN);
        iseg.appendCode(REMOVE);
```

```
                    lexer.nextToken();
                } else syntaxError("Need { or String");
            }
        } else {                                /* スカラー変数の場合 */
          if (type == S_INT) type = T_INT;
            else if (type == S_CHAR) type = T_CHAR;
            else type = T_BOOL;
            vt.addElement(type, name);          /* 名前表に登録 */
            if  (lexer.ttype == S_ASSIGN) {     /* 初期値 */
                lexer.nextToken();
                iseg.appendCode(PUSHI, vt.getAddress(name));
                parseExpression();
                switch (type) {
                case T_INT:
                case T_CHAR:
                    if (expType != T_INT && expType != T_CHAR)
                        syntaxError("Type mismatched");/* 両辺
の型が一致しなければエラー */
                    break;
                case T_BOOL:
                    if (expType != T_BOOL)
                        syntaxError("Type mismatched");/* 両辺
の型が一致しなければエラー */
                    break;
                default:
                    syntaxError("Type mismatched");
                }
```

```
                    leftToRight();                  /* 左辺値ならばその
アドレスの値をロード */
                    iseg.appendCode(ASSGN);         /* 代入 */
                    iseg.appendCode(REMOVE);
                }
            }
        return lexer.ttype;
    }

    public static int parseInitialAssignArray(int type, int addr, int size) {
                                                    /* 配列への初期値代入 */
        int index = 0;
        if (lexer.ttype != S_LBRACE) syntaxError("Need {");
        lexer.nextToken();
        if (type == T_ARRAYOFINT || type == T_ARRAYOFCHAR) {
            if (lexer.ttype != S_INTEGER && lexer.ttype != S_CHARACTER)
                syntaxError("Need Integer or Character");
        } else if (lexer.ttype != S_TRUE && lexer.ttype != S_FALSE)
            syntaxError("Need Integer or Character");
        lexer.nextToken();
        iseg.appendCode(PUSHI, addr);
        iseg.appendCode(PUSHI, lexer.value);
        iseg.appendCode(ASSGN);
        iseg.appendCode(REMOVE);
        while (lexer.ttype == S_COMMA) {
            index++;
            if (index>=size) syntaxError("Array index is out of bound");
```

```
				lexer.nextToken();
				if (type == T_ARRAYOFINT || type == T_ARRAYOFCHAR)	{
					if (lexer.ttype != S_INTEGER && lexer.ttype !=
S_CHARACTER)
						syntaxError("Need Integer or Character");
				} else if (lexer.ttype != S_TRUE && lexer.ttype != S_FALSE)
					syntaxError("Need Integer or Character");
				lexer.nextToken();
				iseg.appendCode(PUSHI, addr+index);
				iseg.appendCode(PUSHI, lexer.value);
				iseg.appendCode(ASSGN);
				iseg.appendCode(REMOVE);
			}
			if (lexer.ttype != S_RBRACE) syntaxError("Need }");
			return lexer.nextToken();
		}

		public static int parseStatementList() {	/* 文の並び */
			while (lexer.ttype != S_RBRACE) parseStatement();
			return lexer.nextToken();
		}

		public static int parseStatement() {		/* 文 */
			switch (lexer.ttype) {
			case S_PARALLEL:				/* parallel 文*/
				parsePARALLEL();
				break;
```

```
case S_IF:                               /* if 文 */
        parseIf();
        break;
case S_WHILE:                            /* while 文 */
        parseWhile();
        break;
case S_FOR:                              /* for 文 */
        parseFor();
        break;
case S_DO:                               /* do-while 文 */
        parseDoWhile();
        break;
case S_WRITE:                            /* writeint 文 */
        parseWrite();
        break;
case S_WRITEINT:                         /* writeint 文 */
        parseWriteint();
        break;
case S_WRITECHAR:                        /* writechar 文 */
        parseWritechar();
        break;
case S_WRITELN:                          /* writeln 文 */
        parseWriteln();
        break;
case S_WRITESTR:                         /* writestr 文 */
        parseWritestr();
```

```
        break;
    case S_NAME:                                /* 式文 */
    case S_INTEGER:
    case S_CHARACTER:
    case S_TRUE:
    case S_FALSE:
    case S_READINT:
    case S_READCHAR:
    case S_READSTR:
    case S_RAND:
    case S_LPAREN:
    case S_NOT:
    case S_SUB:
    case S_INC:
    case S_DEC:
    case S_STRING:
        parseExpression();
        removeStackTop(); // スタックトップの除去
        if (lexer.ttype != S_SEMICOLON) syntaxError("Need ;");
        lexer.nextToken();
        break;
    case S_SEMICOLON:                           /* 空文 */
        lexer.nextToken();
        break;
    case S_LBRACE:                              /* { 文の並び } */
        lexer.nextToken();
```

```
                parseStatementList();
                break;
            default:
                syntaxError("Illegal Token");
                break;
            }
            return lexer.ttype;
    }

    public static int parsePARALLEL() {          /* parallel 文 */
            if (parallel ==  true) syntaxError              ("prarellel 文の中に
parallel 文はだめです");
            parallel = true;
            if (lexer.nextToken() != S_LPAREN) syntaxError("Need(");
            lexer.nextToken();
            parseExpression();
            if (lexer.ttype != S_COMMA) syntaxError("Need,");
            lexer.nextToken();
            parseExpression();
            iseg.appendCode(PARA);
            if (lexer.ttype != S_COMMA) syntaxError("Need,");
            lexer.nextToken();
            parseExpression();
            if (lexer.ttype != S_RPAREN) syntaxError("Need)");
            lexer.nextToken();
            parseStatement();
            iseg.appendCode(SYNC);
```

```
        parallel = false;
        return lexer.ttype;
        }

    public static int parseIf() {                /* if 文 */
        if (lexer.nextToken() != S_LPAREN) syntaxError("Need (");
        lexer.nextToken();
        parseExpression();                       /* 条件式 */
        if (expType != T_BOOL) syntaxError("Type mismatched");
                                                 /* 式が論理型でなければエラ
ー */
        leftToRight();                 /* 左辺値ならばそのアドレスの値を
ロード */
        if (lexer.ttype != S_RPAREN) syntaxError("Need )");
        lexer.nextToken();

        int ln_BEQ;
        switch (iseg.operator(iseg.isegPtr-1)) { /* 条件式が偽なら分岐 */
                                                 /* 分岐先は未定 */
        case NOT:
            iseg.isegPtr --;
            if (iseg.operator(iseg.isegPtr-1) == INC) {
                iseg.isegPtr --;
                ln_BEQ = iseg.appendCode(BGE);    /* x<y のとき */
            } else if (iseg.operator(iseg.isegPtr-1) == DEC) {
                iseg.isegPtr --;
                ln_BEQ = iseg.appendCode(BLE);    /* x>y のとき */
```

```
        } else ln_BEQ = iseg.appendCode(BNE); /* x==y のとき */
        break;
case INC:
        iseg.isegPtr --;
        ln_BEQ = iseg.appendCode(BLT);        /* x>=y のとき */
        break;
case DEC:
        iseg.isegPtr --;
        ln_BEQ = iseg.appendCode(BGT);        /* x<=y のとき */
        break;
default:
        ln_BEQ = iseg.appendCode(BEQ);
        break;
}

parseStatement();

if (lexer.ttype == S_ELSE) {
        lexer.nextToken();
        int ln_JUMP = iseg.appendCode(JUMP);  /* 次の文へ JUMP
                                                JUMP 先は未定 */
        iseg.replaceCode(ln_BEQ, iseg.isegPtr);/* 条件分岐の
                                                 分岐先決定 */
        parseStatement();

        iseg.replaceCode(ln_JUMP, iseg.isegPtr);/* JUMP 先決定 */
```

```
        } else iseg.replaceCode(ln_BEQ, iseg.isegPtr);/* 条件分岐の
                                                    分岐先決定 */
        return lexer.ttype;
    }

    public static int parseWhile() {            /* while 文 */
        int ln_entry = iseg.isegPtr;            /* while 文の入口 */

        if (lexer.nextToken() != S_LPAREN) syntaxError("Need (");
        lexer.nextToken();
        parseExpression();                       /* 条件式 */
        if (expType != T_BOOL) syntaxError("Type mismatched");
                                          /* 式が論理型でなければエラ
ー */
        leftToRight();                  /* 左辺値ならばそのアドレスの値を
ロード */
        if (lexer.ttype != S_RPAREN) syntaxError("Need )");
        lexer.nextToken();

        int ln_BEQ;
        switch (iseg.operator(iseg.isegPtr-1)) { /* 条件式が偽なら分岐 */
                                          /* 分岐先は未定 */
        case NOT:
            iseg.isegPtr --;
            if (iseg.operator(iseg.isegPtr-1) == INC) {
```

```
                iseg.isegPtr --;
                ln_BEQ = iseg.appendCode(BGE);    /* x<y のとき */
        } else if (iseg.operator(iseg.isegPtr-1) == DEC) {
                iseg.isegPtr --;
                ln_BEQ = iseg.appendCode(BLE);    /* x>y のとき */
        } else ln_BEQ = iseg.appendCode(BNE); /* x==y のとき */
        break;
case INC:
        iseg.isegPtr --;
        ln_BEQ = iseg.appendCode(BLT);        /* x>=y のとき */
        break;
case DEC:
        iseg.isegPtr --;
        ln_BEQ = iseg.appendCode(BGT);        /* x<=y のとき */
        break;
default:
        ln_BEQ = iseg.appendCode(BEQ);
        break;
}

parseStatement();

iseg.appendCode(JUMP, ln_entry);        /* 入口へ JUMP */
iseg.replaceCode(ln_BEQ, iseg.isegPtr); /* 条件分岐の
                                           分岐先決定 */
return lexer.ttype;
```

```
    }


    public static int parseFor() {              /* for 文 */
            if (lexer.nextToken() != S_LPAREN) syntaxError("Need (");
            if (lexer.nextToken() != S_SEMICOLON) {
                    parseExpression();                  /* 初期式 */
                    removeStackTop(); // スタックトップの除去
                    while (lexer.ttype == S_COMMA) {
                            lexer.nextToken();
                            parseExpression();
                            removeStackTop(); // スタックトップの除去
                    }
                    if (lexer.ttype != S_SEMICOLON) syntaxError("Need ;");
            }
            lexer.nextToken();
            int ln_entry = iseg.isegPtr;            /* for 文の入口 */
            parseExpression();                          /* 条件式 */
            if (expType != T_BOOL) syntaxError("Type mismatched");
                                                    /* 式が論理型でなければエラ
ー */
            leftToRight();                      /* 左辺値ならばそのアドレスの値を
ロード */
            if (lexer.ttype != S_SEMICOLON) syntaxError("Need ;");
            lexer.nextToken();
            int ln_BEQ;
            switch (iseg.operator(iseg.isegPtr-1)) { /* 条件式が偽なら分岐 */
                                                    /* 分岐先は未定 */
```

```
case NOT:
        iseg.isegPtr --;
        if (iseg.operator(iseg.isegPtr-1) == INC) {
                iseg.isegPtr --;
                ln_BEQ = iseg.appendCode(BGE);    /* x<y のとき */
        } else if (iseg.operator(iseg.isegPtr-1) == DEC) {
                iseg.isegPtr --;
                ln_BEQ = iseg.appendCode(BLE);    /* x>y のとき */
        } else ln_BEQ = iseg.appendCode(BNE); /* x==y のとき */
        break;
case INC:
        iseg.isegPtr --;
        ln_BEQ = iseg.appendCode(BLT);        /* x>=y のとき */
        break;
case DEC:
        iseg.isegPtr --;
        ln_BEQ = iseg.appendCode(BGT);        /* x<=y のとき */
        break;
default:
        ln_BEQ = iseg.appendCode(BEQ);
        break;
}

int ln_JUMP = iseg.appendCode(JUMP);
int ln_cont = iseg.isegPtr;
if (lexer.ttype != S_RPAREN) {
```

```
                parseExpression();                  /* 継続式 */
                removeStackTop(); // スタックトップの除去
                while (lexer.ttype == S_COMMA) {
                        lexer.nextToken();
                        parseExpression();
                        removeStackTop(); // スタックトップの除去
                }
        }
        iseg.appendCode(JUMP, ln_entry);        /* 入口へ JUMP */
        if (lexer.ttype != S_RPAREN) syntaxError("Need )");
        lexer.nextToken();

        iseg.replaceCode(ln_JUMP, iseg.isegPtr);/* JUMP 先決定 */

        parseStatement();

        iseg.appendCode(JUMP, ln_cont);          /* 継続式へ JUMP */
        iseg.replaceCode(ln_BEQ, iseg.isegPtr); /* 条件分岐の
                                                  分岐先決定 */
        return lexer.ttype;
}

public static int parseDoWhile() {          /* do-while 文 */
        int ln_entry = iseg.isegPtr;            /* do-while 文の入口 */
        lexer.nextToken();
```

```
parseStatement();

if (lexer.ttype != S_WHILE) syntaxError("Need while");
if (lexer.nextToken() != S_LPAREN) syntaxError("Need (");
lexer.nextToken();
parseExpression();                    /* 条件式 */
if (expType != T_BOOL) syntaxError("Type mismatched");
                                        /* 式が論理型でなければエラ
ー */
leftToRight();                  /* 左辺値ならばそのアドレスの値を
ロード */
if (lexer.ttype != S_RPAREN) syntaxError("Need )");
if (lexer.nextToken() != S_SEMICOLON) syntaxError("Need ;");

switch (iseg.operator(iseg.isegPtr-1)) { /* 条件式が真なら入口へ分
岐 */
case NOT:
       iseg.isegPtr --;
       if (iseg.operator(iseg.isegPtr-1) == INC) {
              iseg.isegPtr --;
              iseg.appendCode(BLT, ln_entry);   /* x<y のとき */
       } else if (iseg.operator(iseg.isegPtr-1) == DEC) {
              iseg.isegPtr --;
              iseg.appendCode(BGT, ln_entry);   /* x>y のとき */
       } else iseg.appendCode(BEQ, ln_entry);/* x==y のとき */
       break;
case INC:
```

```
            iseg.isegPtr --;
            iseg.appendCode(BGE, ln_entry);       /* x>=y のとき */
            break;
        case DEC:
            iseg.isegPtr --;
            iseg.appendCode(BLE, ln_entry);       /* x<=y のとき */
            break;
        default:
            iseg.appendCode(BNE, ln_entry);
            break;
        }

        return lexer.ttype;
    }

    public static int parseWrite() {            /* write 文 */
        if (lexer.nextToken() != S_LPAREN) syntaxError("Need (");
        lexer.nextToken();
        parseExpression();
        if (expType != T_INT && expType != T_CHAR)
            syntaxError("Type mismatched");     /* 式が整数型でなければ
エラー */
        leftToRight();                    /* 左辺値ならばそのアドレスの値を
ロード */
        if (lexer.ttype != S_RPAREN) syntaxError("Need )");
        if (lexer.nextToken() != S_SEMICOLON) syntaxError("Need ;");
        if (expType == T_INT) iseg.appendCode(OUTPUT);
```

```
            else iseg.appendCode(OUTPUTC);
            return lexer.nextToken();
    }

    public static int parseWriteint() {          /* writeint 文 */
            if (lexer.nextToken() != S_LPAREN) syntaxError("Need (");
            lexer.nextToken();
            parseExpression();
            if (expType != T_INT && expType != T_CHAR)
                    syntaxError("Type mismatched");     /* 式が整数型でなければ
エラー */
            leftToRight();                   /* 左辺値ならばそのアドレスの値を
ロード */
            if (lexer.ttype != S_RPAREN) syntaxError("Need )");
            if (lexer.nextToken() != S_SEMICOLON) syntaxError("Need ;");
            iseg.appendCode(OUTPUT);
            return lexer.nextToken();
    }

    public static int parseWritechar() {         /* writechar 文 */
            if (lexer.nextToken() != S_LPAREN) syntaxError("Need (");
            lexer.nextToken();
            parseExpression();
            if (expType != T_INT && expType != T_CHAR)
                    syntaxError("Type mismatched");     /* 式が整数型でなければ
エラー */
            leftToRight();                   /* 左辺値ならばそのアドレスの値を
ロード */
```

```
        if (lexer.ttype != S_RPAREN) syntaxError("Need )");
        if (lexer.nextToken() != S_SEMICOLON) syntaxError("Need ;");
        iseg.appendCode(OUTPUTC);
        return lexer.nextToken();
    }

    public static int parseWriteln() {          /* writeln 文 */
        if (lexer.nextToken() != S_SEMICOLON) syntaxError("Need ;");
        iseg.appendCode(OUTPUTL);
        return lexer.nextToken();
    }

    public static int parseWritestr() {         /* writestr 文 */
        if (lexer.nextToken() != S_LPAREN) syntaxError("Need (");
        lexer.nextToken();
        if (lexer.ttype == S_STRING) parseString();
        else if(lexer.ttype == S_NAME) parseVar();
        else syntaxError("Need String or Var");
        if (expType != T_ARRAYOFCHAR)
                syntaxError("Type mismatched");     /* 式が文字型配列でなければエラー */
        if (leftValue) iseg.appendCode(LOADS);   /* 左辺値ならばそのアドレスの値をロード */
        if (lexer.ttype != S_RPAREN) syntaxError("Need )");
        if (lexer.nextToken() != S_SEMICOLON) syntaxError("Need ;");
        iseg.appendCode(OUTPUTS);
        return lexer.nextToken();
```

```
    }

    public static int parseExpression() {
            parseLogicalExpression();
            if (lexer.ttype == S_ASSIGN) {
                    if (!leftValue) syntaxError("No left Value");/* 左辺値を持た
なければエラー */
                    int leftExpType = expType;
                    lexer.nextToken();
                    parseExpression();
                    switch (leftExpType) {
                    case T_INT:
                    case T_CHAR:
                            if (expType != T_INT && expType != T_CHAR)
                                    syntaxError("Type mismatched");
                                                    /* 両辺の型が一致しなければエラー
*/
                            leftToRight();                      /* 左辺値ならばその
アドレスの値をロード */
                            iseg.appendCode(ASSGN);
/* 代入 */
                            expType = leftExpType;
                    break;
                    case T_BOOL:
                            if (expType != T_BOOL)
                                    syntaxError("Type mismatched");
                            leftToRight();                      /* 左辺値ならばその
アドレスの値をロード */
```

```
                            iseg.appendCode(ASSGN);
/* 代入 */
                            break;
                    case T_ARRAYOFCHAR:
                            if (expType != T_ARRAYOFCHAR)
                                    syntaxError("Type mismatched");
                                                    /* 両辺の型が一致しなければエラー
*/
                            if (leftValue) iseg.appendCode(LOADS);
                                                    /* 左辺値ならばそのアドレスの値をロ
ード */
                            iseg.appendCode(ASSGNS);
/* 文字列代入 */
                            expType = T_INT;
                            break;
                    default:
                            syntaxError("Type mismatched");
                    }
                    leftValue = false;
            } else if (lexer.ttype == S_ADDLHS || lexer.ttype == S_SUBLHS ||
lexer.ttype == S_MULLHS
                    || lexer.ttype == S_DIVLHS || lexer.ttype == S_MODLHS ) {
                    if (!leftValue) syntaxError("No left Value");      /* 左辺値
を持たなければエラー */
                    if (expType != T_INT && expType != T_CHAR)
                            syntaxError("Type mismatched");
                    int operator = lexer.ttype;
                    int LeftExpType = expType;
```

```
			lexer.nextToken();
			iseg.appendCode(COPY);
			iseg.appendCode(LOAD);
			parseExpression();
			if (expType != T_INT && expType != T_CHAR)
				syntaxError("Type mismatched");
			leftToRight();			/* 左辺値ならばそのアドレス
の値をロード */
			switch(operator) {
			case S_ADDLHS:
				iseg.appendCode(ADD);				/*
和代入 */
				break;
			case S_SUBLHS:
				iseg.appendCode(SUB);				/*
差代入 */
				break;
			case S_MULLHS:
				iseg.appendCode(MUL);				/*
積代入 */
				break;
			case S_DIVLHS:
				iseg.appendCode(DIV);				/*
商代入 */
				break;
			case S_MODLHS:
				iseg.appendCode(MOD);				/*
剰余代入 */
```

```
                    break;
                }
                iseg.appendCode(ASSGN);
                expType = LeftExpType;
                leftValue = false;
            }
            return lexer.ttype;
        }


    public static int parseLogicalExpression() {        /* 式 (論理項
                                                        またはその論理和)*/
            parseLogicalTerm();
            if (lexer.ttype == S_OR) {
                if (expType != T_BOOL) syntaxError("Type mismatched");
                leftToRight();                      /* 左辺値ならばそのアドレス
の値をロード */
                do {
                    lexer.nextToken();
                    parseLogicalTerm();
                    if (expType != T_BOOL) syntaxError("Type mismatched");
                    leftToRight();                      /* 左辺値ならばその
アドレスの値をロード */
                    if (iseg.operator(iseg.isegPtr-1) == PUSHI
                        && iseg.operator(iseg.isegPtr-2) == PUSHI) {
                        /* 定数||定数 はコンパイル時に計算 */
                        iseg.isegPtr -=2;
                        if (iseg.operand(iseg.isegPtr)!=0
```

```
                                || iseg.operand(iseg.isegPtr+1)!=0)
                                iseg.appendCode(PUSHI, 1);
                        else iseg.appendCode(PUSHI, 0);
                } else iseg.appendCode(OR);
            } while (lexer.ttype == S_OR);
            leftValue = false;
        }
        return lexer.ttype;
    }

    public static int parseLogicalTerm() {      /* 論理項 (論理因子
                                                    またはその論理積)*/
        parseLogicalFactor();
        if (lexer.ttype == S_AND) {
            if (expType != T_BOOL) syntaxError("Type mismatched");
            leftToRight();                  /* 左辺値ならばそのアドレス
の値をロード */
            do {
                lexer.nextToken();
                parseLogicalFactor();
                if (expType != T_BOOL) syntaxError("Type mismatched");
                leftToRight();                  /* 左辺値ならばその
アドレスの値をロード */
                if (iseg.operator(iseg.isegPtr-1) == PUSHI
                    && iseg.operator(iseg.isegPtr-2) == PUSHI) {
                    /* 定数&&定数 はコンパイル時に計算 */
                    iseg.isegPtr -=2;
```

```
                    if (iseg.operand(iseg.isegPtr)!=0
                            && iseg.operand(iseg.isegPtr+1)!=0)
                            iseg.appendCode(PUSHI, 1);
                    else iseg.appendCode(PUSHI, 0);
                } else iseg.appendCode(AND);
            } while (lexer.ttype == S_AND);
            leftValue = false;
        }
        return lexer.ttype;
    }

    public static int parseLogicalFactor() {    /* 論理因子 (算術式 または
                                                    二つの算術式の大小比較*/
        parseArithmeticExpression();
        if (lexer.ttype == S_EQUAL || lexer.ttype == S_NOTEQ) {
            if (expType != T_INT && expType != T_CHAR && expType != T_BOOL)
                syntaxError("Type mismatched");
            int leftExpType = expType;
            leftToRight();                      /* 左辺値ならばそのアドレス
の値をロード */
            int operator = lexer.ttype;
            lexer.nextToken();
            parseArithmeticExpression();
            switch (leftExpType) {
            case T_INT:
            case T_CHAR:
                if (expType != T_INT && expType != T_CHAR)
```

```
                syntaxError("Type mismatched");
                        /* 両辺の型が一致しなければエラー */
            break;
        case T_BOOL:
            if (expType != T_BOOL)
                syntaxError("Type mismatched");
                        /* 両辺の型が一致しなければエラー */
            break;
        default:
            syntaxError("Type mismatched");
        }
        leftToRight();                  /* 左辺値ならばそのアドレス
の値をロード */
        int pctr = iseg.appendCode(COMP);
        if (operator == S_EQUAL)
            iseg.appendCode(NOT);
        expType = T_BOOL;
        leftValue = false;
    } else if (lexer.ttype == S_LESS || lexer.ttype == S_GREAT
        || lexer.ttype == S_LESSEQ || lexer.ttype == S_GREATEQ) {
        if (expType != T_INT && expType != T_CHAR)
            syntaxError("Type mismatched");
        int leftExpType = expType;
        leftToRight();                  /* 左辺値ならばそのアドレス
の値をロード */
        int operator = lexer.ttype;
        lexer.nextToken();
```

```
                parseArithmeticExpression();
                if (expType != T_INT && expType != T_CHAR)
                        syntaxError("Type mismatched");
                leftToRight();                  /* 左辺値ならばそのアドレス
の値をロード */
                int pctr = iseg.appendCode(COMP);
                switch (operator) {
                case S_LESS:
                        iseg.appendCode(INC);
                        iseg.appendCode(NOT);
                        break;
                case S_GREAT:
                        iseg.appendCode(DEC);
                        iseg.appendCode(NOT);
                        break;
                case S_LESSEQ:
                        iseg.appendCode(DEC);
                        break;
                case S_GREATEQ:
                        iseg.appendCode(INC);
                        break;
                }
                expType = T_BOOL;
                leftValue = false;
        }
        return lexer.ttype;
    }
```

```
    public static int parseArithmeticExpression() { /* 算術式 (算術項
                                                         またはその和/差)*/
        parseArithmeticTerm();
        if (lexer.ttype == S_ADD || lexer.ttype == S_SUB) {
            if (expType != T_INT && expType != T_CHAR)
                syntaxError("Type mismatched");
            leftToRight();                  /* 左辺値ならばそのアドレス
の値をロード */
            do {
                int operator = lexer.ttype;
                lexer.nextToken();
                parseArithmeticTerm();
                if (expType != T_INT && expType != T_CHAR)
                    syntaxError("Type mismatched");
                leftToRight();                  /* 左辺値ならばその
アドレスの値をロード */
                switch (operator) {
                case S_ADD:
                    if (iseg.operator(iseg.isegPtr-1) == PUSHI
                        &&  iseg.operator(iseg.isegPtr-2)  ==
PUSHI) {
                     /* 定数+定数 はコンパイル時に計算 */
                        iseg.isegPtr -=2;
                        int val = iseg.operand(iseg.isegPtr)
                            + iseg.operand(iseg.isegPtr+1);
                        iseg.appendCode(PUSHI, val);
```

```
                        } else iseg.appendCode(ADD);
                        break;
                    case S_SUB:
                        if (iseg.operator(iseg.isegPtr-1) == PUSHI
                            &&  iseg.operator(iseg.isegPtr-2)  ==
PUSHI) {
                         /* 定数-定数 はコンパイル時に計算 */
                            iseg.isegPtr -=2;
                            int val = iseg.operand(iseg.isegPtr)
                                - iseg.operand(iseg.isegPtr+1);
                            iseg.appendCode(PUSHI, val);
                        } else iseg.appendCode(SUB);
                        break;
                    }
                } while (lexer.ttype == S_ADD || lexer.ttype == S_SUB);
                expType = T_INT;
                leftValue = false;
            }
    return lexer.ttype;
    }

    public static int parseArithmeticTerm() {   /* 算術項 (算術因子
                                                またはその積/商/剰余)*/
        parseArithmeticFactor();
        if (lexer.ttype == S_MUL || lexer.ttype == S_DIV
            || lexer.ttype == S_MOD) {
            if (expType != T_INT && expType != T_CHAR)
```

```
                syntaxError("Type mismatched");
            leftToRight();                    /* 左辺値ならばそのアドレス
の値をロード */
            do {
                int operator = lexer.ttype;
                lexer.nextToken();
                parseArithmeticFactor();
                if (expType != T_INT && expType != T_CHAR)
                    syntaxError("Type mismatched");
                leftToRight();                    /* 左辺値ならばその
アドレスの値をロード */
                switch (operator) {
                case S_MUL:
                    if (iseg.operator(iseg.isegPtr-1) == PUSHI) {
                        if  (iseg.operator(iseg.isegPtr-2)  ==
PUSHI) {
                    /* 定数*定数 はコンパイル時に計算 */
                            iseg.isegPtr -=2;
                            int            val            =
iseg.operand(iseg.isegPtr)
                                        *
iseg.operand(iseg.isegPtr+1);
                            iseg.appendCode(PUSHI, val);
                        } else if (iseg.operand(iseg.isegPtr-1)
== 0) {
                    /* x*0 は 0 に変換 */
                            iseg.isegPtr --;
                            removeStackTop(); // スタックト
ップの除去
```

```
                                                    iseg.appendCode(PUSHI, 0);
                                              } else if (iseg.operand(iseg.isegPtr-1)
== 1)
                                         /* *1 は削除 */
                                                    iseg.isegPtr--;
                                              else  if  (iseg.operand(iseg.isegPtr-1)
== -1) {
                                         /* x/-1 は -x に変換 */
                                                    iseg.isegPtr--;
                                                    if
(iseg.operator(iseg.isegPtr-1) == CSIGN)
                                                          iseg.isegPtr--;
                                                    else iseg.appendCode(CSIGN);
                                              } else iseg.appendCode(MUL);
                                        } else iseg.appendCode(MUL);
                                        break;
                                  case S_DIV:
                                        if (iseg.operator(iseg.isegPtr-1) == PUSHI) {
                                              if (iseg.operand(iseg.isegPtr-1) == 0)
                                                    syntaxError("Zero        divider
detected");
                                              else if (iseg.operator(iseg.isegPtr-2)
== PUSHI) {
                                         /* 定数/定数 はコンパイル時に計算 */
                                                    iseg.isegPtr -=2;
                                                    int            val             =
iseg.operand(iseg.isegPtr)
                                                          /
iseg.operand(iseg.isegPtr+1);
```

```
                                        iseg.appendCode(PUSHI, val);
                                    } else if (iseg.operand(iseg.isegPtr-1)
== 1)
                                /* /1 は 削除 */
                                        iseg.isegPtr--;
                                    else  if  (iseg.operand(iseg.isegPtr-1)
== -1) {
                                /* x/-1 は -x に変換 */
                                        iseg.isegPtr--;
                                        if
(iseg.operator(iseg.isegPtr-1) == CSIGN)
                                            iseg.isegPtr--;
                                        else iseg.appendCode(CSIGN);
                                    } else iseg.appendCode(DIV);
                                } else iseg.appendCode(DIV);
                                break;
                            case S_MOD:
                                if (iseg.operator(iseg.isegPtr-1) == PUSHI) {
                                    if (iseg.operand(iseg.isegPtr-1) == 0)
                                        syntaxError("Zero        divider
detected");
                                    else if (iseg.operator(iseg.isegPtr-2)
== PUSHI) {
                                /* 定数%定数 はコンパイル時に計算 */
                                        iseg.isegPtr -=2;
                                        int            val             =
iseg.operand(iseg.isegPtr)
                                            %
iseg.operand(iseg.isegPtr+1);
```

```
                                        iseg.appendCode(PUSHI, val);
                                } else iseg.appendCode(MOD);
                        } else iseg.appendCode(MOD);
                        break;
                }
        } while (lexer.ttype == S_MUL || lexer.ttype == S_DIV
                || lexer.ttype == S_MOD);
        expType = T_INT;
        leftValue = false;
    }
return lexer.ttype;
}

public static int parseArithmeticFactor() { /* 算術因子 */
    if (lexer.ttype == S_SUB) {
        /* 単項演算子つきの場合 */
        lexer.nextToken();
        parseArithmeticFactor();
      if (expType != T_INT && expType != T_CHAR)
            syntaxError("Type mismatched");
        leftToRight();                  /* 左辺値ならばそのアドレス
の値をロード */
        expType = T_INT;
        switch (iseg.operator(iseg.isegPtr-1)) {
            case CSIGN:   /* - - は除去 */
            iseg.isegPtr--;
            break;
```

```
case PUSHI:        /* -定数 はコンパイル時に計算 */
        iseg.isegPtr--;
        int val = iseg.operand(iseg.isegPtr);
        iseg.appendCode(PUSHI, -val);
        break;
default:
        iseg.appendCode(CSIGN);
        break;
}
leftValue = false;
} else if (lexer.ttype == S_NOT) {
lexer.nextToken();
parseArithmeticFactor();
if (expType != T_BOOL) syntaxError("Type mismatched");
leftToRight();                  /* 左辺値ならばそのアドレス
の値をロード */
switch (iseg.operator(iseg.isegPtr-1)) {
case NOT:          /* ！！ は除去 */
        iseg.isegPtr--;
        break;
case PUSHI:        /* !定数 はコンパイル時に計算 */
        iseg.isegPtr--;
        if (iseg.operand(iseg.isegPtr)==0)
                iseg.appendCode(PUSHI, 1);
        else iseg.appendCode(PUSHI, 0);
        break;
default:
```

```
                    iseg.appendCode(NOT);
                    break;
                }
                leftValue = false;
        } else if (lexer.ttype == S_INC || lexer.ttype == S_DEC) {
                int operator = lexer.ttype;
                if (lexer.nextToken() != S_NAME) syntaxError("Need Var");
                parseUnsignedFactor();
                if (!leftValue) syntaxError("No left Value");
                if (expType != T_INT && expType != T_CHAR)
                        syntaxError("Type mismatched");
                iseg.appendCode(COPY);
                iseg.appendCode(LOAD);
                if (operator == S_INC) iseg.appendCode(INC);
                else iseg.appendCode(DEC);
                iseg.appendCode(ASSGN);
                leftValue = false;
        } else parseUnsignedFactor();
        return lexer.ttype;
}

public static int parseUnsignedFactor() { /* 単項演算子無し因子 */
        switch (lexer.ttype) {
        case S_NAME:                            /* 変数 */
                parseVar();
                if (lexer.ttype == S_INC || lexer.ttype == S_DEC) {
```

```
            if (expType != T_INT && expType != T_CHAR)
                syntaxError("Type mismatched");
            int operator = lexer.ttype;
            lexer.nextToken();
            iseg.appendCode(COPY);
            iseg.appendCode(LOAD);
            if (operator == S_INC) iseg.appendCode(INC);
            else iseg.appendCode(DEC);
            iseg.appendCode(ASSGN);
            if (operator == S_DEC) iseg.appendCode(INC);
            else iseg.appendCode(DEC);
            leftValue = false;
        }
        break;
    case S_PROCESSOR:                       /* $p */
        parseProcessor();
        break;
    case S_INTEGER:                         /* 定数 */
        parseConst();
        break;
    case S_CHARACTER:                       /* 文字型定数 */
        parseCharConst();
        break;
    case S_TRUE:                            /* 論理型定数 */
    case S_FALSE:
        parseLogicalConst();
```

```
        break;
    case S_READINT:                         /* readint */
        parseReadint();
        break;
    case S_READCHAR:                        /* readchar */
        parseReadchar();
        break;
    case S_READSTR:                         /* readstr */
        parseReadstr();
        break;
    case S_RAND:                            /* rand */
        parseRand();
        break;
    case S_LPAREN:                          /* ( 式 ) */
        lexer.nextToken();
        parseExpression();
        if (lexer.ttype != S_RPAREN) syntaxError("Need )");
        lexer.nextToken();
        leftValue = false;
        break;
    case S_STRING:                          /* 文字列 */
        parseString();
        break;
    default:
        syntaxError();
        break;
```

```
        }
        return lexer.ttype;
    }


    public static int parseVar() {            /* 変数 */
        if (!vt.exist(lexer.name)) syntaxError(lexer.name + " : Unknown");
        /* 宣言済み変数かチェック */
        int type = vt.getType(lexer.name);
        arraySize = vt.getSize(lexer.name);
        iseg.appendCode(PUSHI, vt.getAddress(lexer.name));
        lexer.nextToken();
        if (type == T_ARRAYOFINT || type == T_ARRAYOFCHAR
                || type == T_ARRAYOFBOOL) {
                /* 配列型の場合*/
                if (lexer.ttype == S_LBRACKET) {
                        lexer.nextToken();
                        parseArithmeticExpression();      /* 配列の添字 */
                        leftToRight();                    /* 左辺値ならばその
アドレスの値をロード */
                        if(iseg.operator(iseg.isegPtr-1) == PUSHI) {
                                /* 定数添字のアドレスはコンパイル時に計算 */
                                iseg.isegPtr-=2;
                                int addr = iseg.operand(iseg.isegPtr)
                                        + iseg.operand(iseg.isegPtr+1);
                                iseg.appendCode(PUSHI, addr);
                        } else iseg.appendCode(ADD);      /* 添字の値の分だけ
                                                    アドレスをずらす*/
```

```
                if (lexer.ttype != S_RBRACKET) syntaxError("Need ]");
                lexer.nextToken();
                if (type == T_ARRAYOFINT) expType = T_INT;
                else if (type == T_ARRAYOFCHAR) expType = T_CHAR;
                else expType = T_BOOL;
            } else expType = type;
        } else expType = type;
        leftValue = true;
        return lexer.ttype;
    }

    public static int parseProcessor() {      /* $p */
        iseg.appendCode(PUSHP);
        expType = T_INT;
        leftValue = false;
        return lexer.nextToken();
    }

    public static int parseConst() {          /* 定数 */
        iseg.appendCode(PUSHI, lexer.value);
        expType = T_INT;
        leftValue = false;
        return lexer.nextToken();
    }

    public static int parseCharConst() {      /* 文字型定数 */
```

```
        iseg.appendCode(PUSHI, lexer.value);
        expType = T_CHAR;
        leftValue = false;
        return lexer.nextToken();
}

public static int parseLogicalConst() {   /* 論理型定数 */
        if (lexer.ttype == S_TRUE) iseg.appendCode(PUSHI, 1);
        else iseg.appendCode(PUSHI, 0);
        expType = T_BOOL;
        leftValue = false;
        return lexer.nextToken();
}

public static int parseReadint() {        /* readint */
        iseg.appendCode(INPUT);
        expType = T_INT;
        leftValue = false;
        return lexer.nextToken();
}

public static int parseReadchar() {       /* readchar */
        iseg.appendCode(INPUTC);
        expType = T_CHAR;
        leftValue = false;
        return lexer.nextToken();
```

```
    }

    public static int parseReadstr() {          /* readstr */
            iseg.appendCode(INPUTS);
            expType = T_ARRAYOFCHAR;
            leftValue = false;
            return lexer.nextToken();
    }

    public static int parseRand() {             /* rand */
            if (lexer.nextToken() != S_LPAREN) syntaxError("Need (");
            lexer.nextToken();
            parseExpression();
            if (expType != T_INT && expType != T_CHAR)
                    syntaxError("Type mismatched");      /* 式が整数型でなければ
エラー */
            leftToRight();                      /* 左辺値ならばそのアドレスの値を
ロード */
            if (lexer.ttype != S_RPAREN) syntaxError("Need )");
            iseg.appendCode(RAND);
            return lexer.nextToken();
    }

    public static int parseString() {
            iseg.appendCode(PUSHI, '¥0');
            for (int i=lexer.string.length()-1; i>=0; i--)
                    iseg.appendCode(PUSHI, lexer.string.charAt(i));
```

```
        expType = T_ARRAYOFCHAR;
        leftValue = false;
        return lexer.nextToken();
    }


    public static void setUpOption (String[] args) {
        int i;
        for (i=0; i<args.length; i++) {
            if (args[i].charAt(0) == '-') {
                if (args[i].indexOf('c') != -1) statSW = 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('s') != -1) symPrtSW = true;    /*
記号表の表示 */
                if (args[i].indexOf('t') != -1) traceSW = true;     /*
トレースモード */
                if (args[i].indexOf('v') != -1) varOutSW = true;    /*
変数表の表示 */
            } else break;
        }
        if (i<args.length) {
            sourceFile = args[i];
```

```
            i++;
            if (i<args.length) objectFile = args[i];
            else objectFile = null;
        } else {
            System.out.print("Source file : ");
            sourceFile = Console.ReadString();
            objectFile = null;
        }
    }

    // スタックトップのデータを除去する
    // それに従い無効となるデータも全て除去する
    public static void removeStackTop() {
                                    /* スタックトップの除去(文字列の場合) */
        if (!leftValue && expType == T_ARRAYOFCHAR) {
            int operator, operand;
            do {
                operator = iseg.operator(iseg.isegPtr-1);
                operand = iseg.operand(iseg.isegPtr-1);
                iseg.isegPtr--;
                iseg.print(iseg.isegPtr);
            } while (operator == PUSHI && operand !=0);
                                    /* PUSHI 0 が現れるまでスタックをポップ */
        } else {
            int depth = 1;
            while (depth > 0 ) {
```

```
                        int prevOperator = iseg.operator(iseg.isegPtr-1);
                        if (prevOperator == INC || prevOperator == DEC
                                || prevOperator == CSIGN || prevOperator == NOT
                                || prevOperator == LOAD || prevOperator == RAND)
{
                                iseg.isegPtr--;
                                prevOperator = iseg.operator(iseg.isegPtr-1);
                        } else if (prevOperator == PUSH || prevOperator ==
PUSHI) {
                                iseg.isegPtr--;
                                depth--;
                        } else if (prevOperator == ADD || prevOperator == SUB
                                || prevOperator == MUL || prevOperator == DIV
                                || prevOperator == MOD || prevOperator == AND
                                || prevOperator == OR || prevOperator == XOR
                                || prevOperator == COMP) {
                                iseg.isegPtr--;
                                depth++;
                        } else break;
                }
                for (int i=depth; i>0; i--)
                        iseg.appendCode(REMOVE);          /* スタックトップの
除去 */
        }
    }

    // スタックトップの値が左辺値であればロードして右辺値にする
```

```
    public static void leftToRight() {
        if (leftValue) {
            if (iseg.operator(iseg.isegPtr-1) == PUSHI) {
                /* PUSHI addr, LOAD は PUSH addr に変換 */
                iseg.isegPtr --;
                int addr = iseg.operand(iseg.isegPtr);
                iseg.appendCode(PUSH, addr);
            } else iseg.appendCode(LOAD);
        }
    }

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

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

```
}

/* Kc.java ここまで */
```

付録3 和計算プログラム

```
// nデータの和計算プログラム
// n個のデータの和をpプロセッサを使って求める
// 理論的には計算量は O (n/p + log p)

main() {
	int n = 16;
	int p = 4;
	int i;
	int j[4];
	int a[16];
	int b[4];

	// 初期値代入
	parallel (0, p-1,1) {
		for (j[$p]=n*$p/p; j[$p]<n*($p+1)/p; j[$p]++) {
			// a[j[$p]] に0～9の乱数を入れる
			a[j[$p]] = rand (10);
			// 初期値表示
			write (a[j[$p]]);
		}
	}
	writeln;

	// 各プロセッサがaのn/p個のデータの和を求め配列bに代入
	parallel (0, p-1,1) {
		b[$p] = 0;
		for (j[$p]=n*$p/p; j[$p]<n*($p+1)/p; j[$p]++) {
			b[$p] += a[j[$p]];
		}
	}

	// 2分木を用いて配列bの和計算
	for (i=1; i<p; i*=2) {
		parallel (0, p/(2*i)-1,1) {
			b[$p] += b[$p + p/(2*i)];
		}
```

```
        }

        // 和出力
        write (b[0]);
}
```