コンピュータ科学実験３指導書
コンパイラの作成

名古屋大学情報学部コンピュータ科学科

まえがき

実験の目的としては以下のような事柄がある．

(1) 情報工学の基礎について，講義・演習などで得た知識を実験を通して検証・体得する．

(2) 情報工学の基礎となるシステムの構築に関して，実際に計算機ハードウェアおよびソフトウェアのシステムを構築することにより，その構成法を体得する．

(3) 本実験においては，個人ワークとチームワークの二面性を持った課題が与えられている．このような課題の解決に向けての仕事の進め方を体得する．

(4) 分からない問題に対して，文献・資料を調査したり他人に聞くなど工夫をこらし，自らがこれを解決する態度（自立的探求心と問題解決能力）を身につける．

(5) 事象や現象を把握するための基本的な装置・道具（計測機器・ソフトウェア）について学び，その操作方法を会得する．

(6) 実験レポートを記述する訓練を通じてレポートの記述形式を学ぶともに，他人が理解できる文章を書く能力を習得する．

これらの事柄をふまえて，積極的に実験課題に取り込んで欲しい．

レポートの提出期限について

本実験では，大きく分けて 3 つの実験課題レポートを提出する．レポートの書き方についての指導を受けるため，また，実験スケジュールの目安として，各課題レポートの提出期限を下記の通りとする．

- 第 2 週終了の 1 週間後まで: 課題 1–3
- 第 5 週終了の 1 週間後まで: 課題 4–5
- 第 7 週終了の 1 週間後まで: 課題 6–10

実験の心得

実験に対しての心得

(1) 実験中に人身事故などの災害が発生したときは，冷静かつ速やかに応急処置を行い，実験担当者に報告して指示を受けること．

(2) 実験中にふざけたりしない．常に細心の注意を払い，人身事故や災害の無いように心がける．

(3) 欠席や遅刻および無断退席はしない．なお，本人および親族等に緊急やむを得ない事由（病気・事故）が生じたときには，実験担当者に申し出て指示を受けること．無断欠席は不合格となる．

(4) レポートの提出期限を厳守すること．

(5) 指導書をあらかじめよく読み，実験の目的や実験方法・手順などを捉えておくこと．

(6) グラフ用紙や筆記用具などを用意すること．

(7) 予期しない結果が出ることがある．その場合は原因を追求・分析すること．

(8) 難解な問題と思えるものは文献・資料などを調査して解決すること．

(9) 実験が終了したとき，班の代表者は実験担当者に実験の進捗状況や実験装置・機器具の不具合，災害の有無等を報告すること．

実験レポートの書き方

(1) 実験報告書は PDF ファイルとして作成する．

(2) レポートの冒頭に氏名・学籍番号に加えて e-mail アドレスを記載する．レポート再提出などの連絡は，そのアドレスに送るのでメールを見落とさないこと．

(3) レポートの作成には LaTeX を使用することを**強く推奨する**．

- 実験室の環境において使用できるコマンドが情報工学実験の Web ページ[1]に載っている．例えば以下のコマンドがある．

  LaTeX ファイルのコンパイル: `platex main.tex`
  DVI ファイルの閲覧: `pxdvi main.dvi`
  DVI から PDF への変換: `dvipdfmx main.dvi`
  PDF ファイルの閲覧と印刷: `acroread main.pdf`

[1] `http://www.ice.nuie.nagoya-u.ac.jp/jikken/latex.html`

DVI ファイルの印刷: `dvips -t a4 -q -f main.dvi | lpr`

- その他の環境で使用する場合は，TEX Wiki のサイト[2]を参照せよ．特に MS-Windows を使用する場合は，TEX インストーラ 3 のサイト[3]が参考になる．
- この他，LaTeX を取り扱えるウェブサービスを使う方法も推奨される．以下は 2019 年始の推奨されるウェブサービスである．これらでは，実績のある (つまり古い)pLaTeX だけではなく，LuaLaTeX 等が利用可能であり，またディフォルトになっているため注意が必要である．なお，本指導書の一部も Overleaf を用いて執筆されており，コンパイルには LuaLaTeX，ドキュメントクラスとしては bxjsarticle が用いられている．
  - Overleaf: `https://www.overleaf.com/`
  - Cloud LaTeX: `https://cloudlatex.io/`

(4) レポートは覚え書きではない．他人に提出する報告書である．従って，実験の課題名，目的，使用機器具，方法・手順，結果，考察（所見），文献・資料をよく整理し，報告書（実験レポート）を一読してすべての内容が分かるように記述する．

(5) レポートはそれ自体で完結した文書である．よって他の文献（特に実験指導書）を参考しなくとも内容が理解できるように記述する．ただし，道具の使用方法は記述しない．

(6) 各課題ごとに，実験方法，実験結果，考察を記述する．

(7) 実験方法・原理・アルゴリズム

- 分かりやすく記述する．実験で用いた装置の設計仕様や実験回路，原理図などは精確に描き，報告書に付ける．

(8) 実験結果

- 実験に必要なプログラムは，プログラムリストを添付しコメントを付ける．そのプログラム（システム）の設計概念（考え方）を分かりやすく記述する．
- 実験データは必要に応じて表・グラフにする．測定の際に読み取った生データは，有効数字が分かるように正しく書く．生データをもとに何らかの数値処理（情報処理）をしてグラフ化した場合には，その処理方法や理論も説明する．
- グラフには，グラフの目盛，単位，目盛のスケール，パラメータを付け，実験データはグラフの各軸の目盛をもとにプロットする．グラフの曲線上のどこをとってもデータが正しく読み取れるようにする（有効数字が 3 桁なら 3 桁まで正確に読めるようにする）．

[2] `http://oku.edu.mie-u.ac.jp/~okumura/texwiki/`
[3] `http://www.math.sci.hokudai.ac.jp/~abenori/soft/abtexinst.html`

- **表の上**に，表番号と適切なキャプションを付ける．
- **図の下**に，図番号と適切なキャプションを付ける．
- ハードウェア実験の場合は使用機器名，機器のシリアル番号，回路図を付ける．

(9) 考察

- 何に対する考察なのか分かるように書く．
- データの羅列や実験手順の説明を考察と勘違いしない．
- 考察は他者と共有してはならない（他人のものを写してはならない）．

(10) 参考文献

- レポートを書くうえで参考にした文献を記載せよ．
- ウェブ・ページを記載する場合は適切な URL を示せ．Wikipedia のトップページを挙げるなどは論外である．
- 実験指導書を参考文献に挙げてはならない．

レポート提出について

(1) 下記 URL からレポート表紙および自己点検表の PDF ファイルをダウンロードする．
`http://www.ice.nuie.nagoya-u.ac.jp/jikken/index.shtml`

(2) レポート表紙に必要事項を記入し，結果を PDF ファイルに保存する．保存後に再度 PDF ファイルを開き，正しく保存されたか確認すること．

(3) 自己点検表を用いてレポートの内容を確認しチェックを付け，結果を PDF ファイルに保存する．保存後に再度 PDF ファイルを開き，正しく保存されたか確認すること．

(4) ウェブブラウザで NUCT にアクセスし，ログインした後に情報工学実験のページを開く．
`https://ct.nagoya-u.ac.jp/`

(5) 左側メニューから課題を選択し，右側の課題一覧からレポート提出を行う課題を選択する．

(6) レポート表紙，自己点検表，レポート本体の 3 つの PDF ファイルを提出する．ファイルサイズが 50MB を超えるとアップロードできないため注意すること．

(7) レポートの提出日は，実験の終了日から一週間後とする．レポートの提出遅れは減点とする．

(8) 再実験やレポートの再提出を求められた者は実験担当者の指示に従うこと．

その他

(1) 本指導書の誤字・脱字・説明の誤りなどに気づいた場合，誤りの場所と内容を以下のメールアドレスに連絡していただきたい．
jikken-text@nuie.nagoya-u.ac.jp

概要

本実験ではコンパイラの講義を踏まえて Pascal に基づく簡単なプログラミング言語のコンパイラを作成する．本実験により，コンパイラ講義の内容を実現してみることでコンパイラに対する理解を深める．ターゲットマシンとして仮想機械を定義する。本実験では、グループで基本的なコンパイラを作成し，その後に各人がコンパイラの機能を拡張する．

実験スケジュール

第 1 週（協働）： 字句解析・構文解析（**課題 1,2**）

第 2 週（協働）： 記号表の作成（**課題 3**）と式の評価

第 3–5 週（協働）： コード生成（**課題 4,5**）

第 6 週（個人）： 関数・配列の実現（**課題 6,7,8,9**）

第 7 週（個人）： 最適化（**課題 10**）

第 8 週： 予備日

上記のスケジュールは進行下限として想定している．その為，上記のスケジュールで課題を実施した場合，特に課題 7 において時間が足りなくなる事が想定される (課題 7 は 1.5〜2 週を要すると想定されている)．従って，一つの課題が終わったときに実験時間が残っている場合は，次の課題へ進むことが推奨される．また，課題においても前倒しての提出も検討して欲しい．

[3]本資料は，過去の演習資料（作成：結縁祥治先生，小尻智子先生，出口大輔先生）に基づいている．

目 次

第1章　実験概要及びソースプログラムの言語仕様・目的マシン

1.1　ソースプログラムの言語仕様

本実験で開発するコンパイラが扱うソースプログラムは Pascal のサブセットである．実験ではまず基本的な機能を実現するコンパイラを開発し，順次，他の機能を追加課題として加えることにより，より実際的なプログラミング言語に対するコンパイラを開発する．最初の基本的な機能を持つプログラミング言語を PL-0 と定義する．また，段階的にコンパイラの機能を追加していったものをそれぞれ PL-1 ，PL-2 ，PL-3 と定義する．

PL-0 のコンパイラと，PL-1〜PL-3 で追加される機能の言語仕様は以下の通りである[1]．

- PL-0
  - データ型は整数型のみが存在する．
  - 手続きがあり，再帰呼出しが可能である．
  - 名前 (変数名と手続き名) の有効範囲は標準 Pascal と同じである．ただし，手続き定義の入れ子の深さは 1 までとする．すなわち，手続きの中で他の手続きを定義できない．
  - 制御文はつぎの 4 種類が存在する．
    - `if ... then ... else ...` (条件分岐)
    - `while ... do ...` (while ループ)
    - `for ... do ...` (for ループ)
    - `begin ..; ..; .. end` (複合文)
  - 入出力のための命令文，read，write が存在する．
- PL-1
  - 手続き呼び出しのときに引数の値渡しが可能である．引数は複数でもよい．
- PL-2
  - 関数（戻り値の存在する手続き）を処理することが可能である．

[1]追加課題として，手続き同士を相互参照するための forward 文の処理がある．

- PL-3
  - 1 次元の配列を扱うことが可能である．配列の開始番号，添字の範囲は自由に設定できる．

なお，PL-0 の構文の BNF 記法を付録 A に示す．以下は PL-0 のプログラムの例である．

PL0A：**1** から **10** までの加算（その **1**）

```
program PL0A;
var n, sum;
begin
    n := 10;
    sum := 0;
    while n > 0 do
    begin
        sum := sum + n;
        n := n - 1
    end
end.
```

このプログラムを C 言語で書かれた下記のプログラムと同等である．

```
int n, sum;
int main()
{
  n = 10;
  sum = 0;
  while( n > 0 ){
    sum = sum + n;
    n = n - 1;
  }
  return sum;
}
```

他のプログラムの例については付録 B に示す．

本来はここでソースプログラムの意味論を形式的に与えるべきである．しかし，本実験で扱うソースプログラムについては，C 言語などプログラミング言語の知識がある人間にとっては，構文から慣例的な意味を一意に定められる．よって，本指導書では意味論については割愛する．

命令セット

`alloca`:	allocate memory for local variable
`global`:	allocate memory for global variable
`load`:	load
`store`:	store
`add`:	sum of its two operands
`sub`:	difference of its two operands
`mul`:	product of its two operands
`sdiv`:	quotient of its two operands
`icmp`:	comparison
`br`:	branch
`call`:	call a function
`ret`:	return
`phi`:	φ node

図 1.1: 目的マシンの命令セット

1.2 目的プログラム

本実験では，目的プログラムとして LLVM IR と呼ばれる言語非依存の中間表現を対象とする．LLVM IR の命令の多くは，アセンブリに似た 3 番地コードであり，変数はレジスタに保存される．LLVM IR は無限個の仮想レジスタを持つレジスタマシンを対象としたものであり，基本的にすべてのレジスタ変数は SSA 形式で表現される．レジスタマシンとは，計算する値はすべてレジスタを介してやり取りするものであり，値をレジスタにロード，レジスタを指定して演算を実行，演算結果をレジスタに格納，という処理の流れで計算を行なうアーキテクチャである．汎用レジスタを持たず，計算をスタックのみで行なうスタックマシンに対して，最適化しやすいという特徴がある．一方で，スタックマシンと比べると，命令が複雑になっている．代表的なレジスタマシンとして，本実験で扱う LLVM や，Lua の処理系などがある．

本実験で扱うことが想定される LLVM IR の命令を図 1.1 に示す．構文や意味論については付録 C に記す．なお，本実験では属性（`attributes`）を扱わず，属性に関する説明を割愛する．

静的単一代入（SSA）形式とは，基本ブロック内で各変数への代入は一度しか行なわれず，一度代入された変数は，それ以降変更されない．そのため，大域的な定数伝播や共通部分式の除去のような最適化が容易に実現できる．LLVM IR では大域変数は`@`s のように，文字列 s の先頭に`@`を付けた名前で表される．計算に用いるレジスタは`%`n のように，非負整数 n の先頭に`%`を付けた名前で表される．LLVM IR は SSA 形式であるため，一度代入したレジスタには，再度代入することはできない．局所変数は，`alloca` 命令により領域を確保し，その番地を`%`n で保持する．大域変数や局所変数へ値を代入する場合，`store` 命

令を用いて@sや%nが指す番地へ代入することができる．

PL0A をコンパイルした結果として期待する LLVM コードの一つとして図 1.2 がある．変数 n，sum は大域変数として扱われるため，LLVM IR のコードではそれぞれ@n，@sum という変数名となり，それらは大域変数として下記のように宣言される．

```
@n = common global i32 0, align 4
@sum = common global i32 0, align 4
```

i32 は 32 ビットの整数であることを表し，0 で初期化されている．align 4 でアドレスが 4 の倍数であることを保証している．次に命令部分を実行する関数としてメイン関数@main が宣言される．define i32@main() は関数@main が 32 ビットの整数を返し，引数を受け取らないことを宣言している．関数@main の命令については，まず返り値を格納するレジスタとして%1 を alloca 命令で宣言し，store 命令でそのレジスタに値 0 をセットしている．

```
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
```

その後，大域変数@n，@sum にそれぞれ値 10，0 をセットしている．

```
  store i32 10, i32* @n, align 4
  store i32 0, i32* @sum, align 4
```

br label %2 は; <label>:2:へジャンプする．

以下のコードは while 文の条件判定に対応する．

```
; <label>:2:                                             ; preds = %5, %0
  %3 = load i32, i32* @n, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %11
```

n > 0 を判定するために大域変数@n の値をレジスタ%3 に格納し，icmp 命令で値 0 と比較する．sgt は符号付き整数に対する $>$ により比較することを指定し，比較結果は 1 ビット整数としてレジスタ%4 に格納される．br i1 %4, label %5, label %11 はレジスタ%4 が真（すなわち，値 1）のときには; <label>:5:にジャンプし，そうでないときには; <label>:11:にジャンプすることを表す．

以下のコードは while 文の内側の命令列に対応する．

```
; <label>:5:                                             ; preds = %2
  %6 = load i32, i32* @sum, align 4
  %7 = load i32, i32* @n, align 4
  %8 = add nsw i32 %6, %7
  store i32 %8, i32* @sum, align 4
```

```
@n = common global i32 0, align 4
@sum = common global i32 0, align 4

define i32 @main() {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, i32* @n, align 4
  store i32 0, i32* @sum, align 4
  br label %2

; <label>:2:                                      ; preds = %5, %0
  %3 = load i32, i32* @n, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %11

; <label>:5:                                      ; preds = %2
  %6 = load i32, i32* @sum, align 4
  %7 = load i32, i32* @n, align 4
  %8 = add nsw i32 %6, %7
  store i32 %8, i32* @sum, align 4
  %9 = load i32, i32* @n, align 4
  %10 = add nsw i32 %9, -1
  store i32 %10, i32* @n, align 4
  br label %2

; <label>:11:                                     ; preds = %2
  ret i32 0
}
```

図 1.2: PL0A をコンパイルした結果として期待する LLVM コードの例

```
  %9 = load i32, i32* @n, align 4
  %10 = add nsw i32 %9, -1
  store i32 %10, i32* @n, align 4
  br label %2
```

式 sum + n を計算するために，大域変数@sum，@n の値をそれぞれレジスタ%6，%7 に格納し，それらのレジスタが格納する値を add 命令で加算する．nsw は No Signed Wrap（符号あり演算のラップなし）を意味し，加算においてオーバフローが発生する．上記のコードでは sum + n の結果は%9 に格納される．最後の 3 行は n := n - 1 に相当する．

以下のコードは関数の実行の終了（C 言語版での return 0;）に対応する．

```
; <label>:11:                                     ; preds = %2
  ret i32 0
```

上記の ret 命令は 32 ビットの整数として値 0 を関数の戻り値として返している．

1.3 コンパイラの記述言語

実際にコンパイラを開発するためには，コンパイラそのものを記述するプログラミング言語が必要である．本実験のコンパイラを記述するための言語として要求される条件は再帰呼出しが可能なことである[2]．本実験ではプログラミング言語 C を標準のプログラミング言語として用いる．再帰呼出しが自然に記述可能な他の言語，例えば，Python や C++ や Java などを用いて開発してもよい．ただし，C 以外で開発する場合は，個々の言語に対応した字句解析や構文解析ツールを利用する必要がある[3]．

[2] この条件は必須ではないが，一般的な Fortran や BASIC など局所変数を持たない言語ではスタックを自分で管理する必要があり，繁雑になる．

[3] Java の場合は，JavaCC や SableCC などの字句解析・構文解析ツールが存在する．Python の場合は PLY (Python-Lex-Yacc) が存在する．

第2章　字句解析・構文解析の実現

2.1　字句解析

本実験で作成するコンパイラにおける予約語と基本シンボルを以下に挙げる[1]．字句解析の例を図 2.1 に示す．ここでは、左側に示す 5 行からなるプログラムを字句解析して、右側の表にあるような単語に切りわけ、その属性と属性値を出力する。

予約語	`begin div do else end for forward function if`
	`procedure program read then to var while write`
基本シンボル	`+ - * = <> < <= > >= ( ) [ ] , ; :  .  ..  :=`

本実験では，字句解析部の実装に lex を用いる．lex は予約語などトークンとして抽出したい文字の並びを正規表現を用いたパターンで記述することで，字句解析処理を実現可能とするツールである．lex の詳しい使用法は付録 D や関連書籍を参照すること．

課題 1

PL-0 の字句解析を行うプログラム scanner を lex で実現せよ．ソースプログラムはテキストファイルとして入力されるものとする．実行は，

```
% 実行ファイル名 ソースファイル
```

のようにソースファイル名を入力する．結果として，切り出したトークンとその種別を対にして出力するようにせよ．プログラム例をソースプログラムとして実行し，トークンが正しく出力されることを確認せよ．なお、付録 E にサンプルプログラムがあるので，参考にすること．

2.2　構文解析

本実験では，構文解析部の実装に yacc を用いる．yacc は実現したいプログラミング言語の文法を構文規則として記述することで，構文解析処理を実現可能とするツールである．yacc で用いる構文規則は BNF 記法を参考に構成されており，文法が BNF 記法で与えられるプログラミング言語を実装するのに都合がよい．yacc の詳しい使用法は関連書籍を参照すること．

[1] PL-0 の仕様には入っていないが，後に必要となる語も予約語として登録してある．

```
program EX1;
var a;
begin
    a:=100
end.
```

単語	属性	属性値
program	予約語	program
EX1	識別子	“EX1”
;	シンボル	;
var	予約語	var
a	識別子	“a”
;	シンボル	;
begin	予約語	begin
a	識別子	“a”
:=	シンボル	:=
100	数値	100
end	予約語	end
.	シンボル	.

図 2.1: 字句解析の例

課題 2

構文解析を行うプログラム parser を yacc で実現せよ．課題 1 で作成した lex プログラム scanner と組み合わせることにより実現する．構文が正しければ何も出力せず，正しくなければ誤りが検出された行番号とその時点のトークンを出力するようにせよ．なお，付録 E にサンプルプログラムがあるので，参考にすること．parser が完成したら，プログラム例の pl0a.p，pl0b.p，pl0c.p, pl0d.p をソースプログラムとして入力し動作を確認せよ．また，pl1a.p や pl2a.p を入力し，誤りが検出出来るか確認せよ．さらに，scanner と parser を実行するための Makefile を作成し，make でコンパイルせよ．

第3章　記号表の管理

一般に 3 番地コードで構成される目的プログラムでは，ソースプログラム中に現れる名前（変数名，手続き名，関数名）を 2 進数（非負整数）で表さなければならない．この変換は**記号表**によって以下のように行われる．ソースプログラムを構文解析するとき，変数宣言，手続き宣言，および関数宣言があると，その宣言内容を記号表に登録する．変数宣言の場合は，その変数に割り当てられるスタックの位置が書き込まれる．手続き宣言や関数宣言の場合には，目的プログラム中でのその手続きのコードの先頭番地が書き込まれる[1]．コンパイラが目的プログラムを出力するときに，変数の参照，関数呼出し，手続き呼出しがあった場合，記号表を参照して，スタックでの位置あるいは目的プログラム中でのジャンプ先の番地を得ることができる．

本実験では LLVM IR を目的プログラムとするために，手続き名や関数名を記号表で管理する必要はなく，それぞれの変数の種別とレジスタ番号を管理すればよい．本章では本実験のために必要な記号表の管理について説明する．

3.1　名前の有効範囲

PL-0 には，**大域変数**（global variable）と**局所変数**（local variable）が存在する．大域変数はプログラムの全ての箇所で有効であり，局所変数はその手続きあるいは関数の中でのみ有効である．もし，同じ名前を持つ大域変数と局所変数がある場合は局所変数が参照される．

3.2　記号表への登録と参照

以降では，図 3.1 (a) のプログラムをコンパイルすることを例に，記号表（図 3.1 (b)）の役割を説明する．なお，コンパイルして得たい LLVM プログラムは図 3.2 のものとする．

図 3.1 (a) のプログラムの 2 行目をコンパイルしたとき，変数 `x`，`y`，`z` の値を保持するための領域として変数名`@x`，`@y`，`@z` がそれぞれ確保されなければならない．したがって，このプログラムに対して，コンパイラは

```
@x = common global i32 0, align 4
```

[1] Pascal 系のプログラミング言語が変数，手続き，関数に対して，あらかじめ宣言を要求するのは記号表の実現をこのように容易にするためである．宣言がないと記号表の作成が容易でなくなることに注意しよう．

```
program EX1;
var x, y, z;
begin
   x := 12;
   y := 20;
   z := x + y
end.
```

x	–	glob
y	–	glob
z	–	glob

(a) ソースプログラム　　(b) 記号表

図 3.1: 記号表の用い方 (1)

```
@y = common global i32 0, align 4
@z = common global i32 0, align 4
```

という命令を生成して，図 3.1 (b) のような記号表を生成する．記号表は，記号，その記号に割り当てられた番号，大域変数か局所変数かの別，の 3 カラムのデータで構成されている．このコードでは，@で始まる識別子が大域変数を表しており，int 型（i32）の領域を確保しており，アドレスのアラインメントが 4 であることを示している．4 行目以降のコード生成では，記号表を参照しながら x，y，z をそれぞれ@x，@y，@z に置き換えればよい．例えば，5 行目の変数 y への代入のとき，記号表により変数 y のレジスタは@y であることが分かるので，store i32 20, i32* @y, align 4 というコードが出力されていることが分かる（目的プログラム 8 行目）．逆に，記号表に登録されていない記号が出現したときは宣言されていない変数を参照しているのでエラーを出力すればよい．

大域変数の値を参照する場合は，一旦レジスタにロードしてからそのレジスタを参照して使用する．そのために，記号表では新たに割り当てたレジスタの番号を記録しておく．新たに割り当てるレジスタの番号を管理するには，値が 0 で初期化されたカウンタ用の変数（int 型）を用意しておけばよい．メイン関数に相当する@main に対して，最初に返り値を格納するレジスタを%1 として用意（%1 = alloca i32, align 4）するので，カウンタの値はその時点でインクリメントされる．6 行目の z := x + y を実行するコードを生成するために式 x + y を解析し，記号表を下記のように更新し，さらに LLVM コードを生成する．

x	2	glob
y	3	glob
z	–	glob

```
%2 = load i32, i32* @x, align 4
%3 = load i32, i32* @y, align 4
%4 = add nsw i32 %2, %3, align 4
```

```
@x = common global i32 0, align 4
@y = common global i32 0, align 4
@z = common global i32 0, align 4
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 12, i32* @x, align 4
  store i32 20, i32* @y, align 4
  %2 = load i32, i32* @x, align 4
  %3 = load i32, i32* @y, align 4
  %4 = add nsw i32 %2, %3, align 4
  store i32 %4, i32* @z, align 4
  ret i32 0
}
```

図 3.2: 図 3.1 (a) のプログラムをコンパイルして得たい LLVM プログラム

レジスタ%4 は x + y の結果を格納している．変数 z に x + y の結果を格納するので，次に，以下のコードを生成する．

```
  store i32 %4, i32* @z, align 4
```

これで実行される命令がなくなりプログラムの処理が終了したので以下のコードを生成してコンパイル作業を終える．

```
  ret i32 0
}
```

以上は大域変数のみが出現する場合であった．次に大域変数と局所変数が出現する場合として下記のソースプログラムを考える．

```
program M;
var x, y, z;
procedure A;
var x;
begin
   x := 1
end;
procedure B;
var y;
begin
```

```
   y := 2;
   x := 5;
end;
begin
   z := 3;
end.
```

このプログラムをコンパイルして得たい LLVM プログラムは下記のものとする．

```
@x = common global i32 0, align 4
@z = common global i32 0, align 4
@y = common global i32 0, align 4

define void @A() #0 {
  %1 = alloca i32, align 4
  store i32 1, i32* %1, align 4
  ret void
}

define void @B() #0 {
  %1 = alloca i32, align 4
  store i32 2, i32* %1, align 4
  store i32 5, i32* @x, align 4
  ret void
}

define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 3, i32* @z, align 4
  ret i32 0
}
```

このプログラムには手続きと局所変数が存在する．ソースプログラムを 3 行目まで解析したとき，記号表には大域変数と手続き A が登録されている．4 行目まで解析すると局所変数 x が登録される．ここで記号表の x の欄に 2 を入れ，局所変数 x は%2 に割り当てる．次に，8 行目まで解析すると，8 行目は局所変数 x の有効範囲外なので，局所変数 x は記号表から削除され，手続き B が登録される．このとき，レジスタの番号はリセットされる．9 行目まで解析すると，手続き B 中の局所変数 y が登録される．ここで記号表の y の欄に 2 を入れ，局所変数 y は%2 に割り当てる．11 行目の y は局所変数の y であるため，@x へ

x	–	glob
y	–	glob
z	–	glob
A	–	proc

(a) 3 行目

x	–	glob
y	–	glob
z	–	glob
A	–	proc
x	2	local

(b) 4 行目

x	–	glob
y	–	glob
z	–	glob
A	–	proc
B	–	proc

(c) 8 行目

x	–	glob
y	–	glob
z	–	glob
A	–	proc
B	–	proc
y	2	local

(d) 9 行目

x	–	glob
y	–	glob
z	–	glob
A	–	proc
B	–	proc

(e) 15 行目

図 3.3: 記号表の用い方 (2)：プログラム M の i 行目まで解析した際の記号表

値を代入する．一方，12 行目の x は大域変数の x であるため，13 行目まで解析すると局所変数 y は記号表から削除される．そして，再びレジスタの番号はリセットされる．15 行目の z は大域変数であるため，12 行目同様，@z への代入を行なう．

それぞれの手続きのコード生成を行なうときは記号表を下から参照する．例えば，手続き A での x は局所変数であるが，これは 4 行目まで解析したときの記号表を下から参照することで分かる（図 3.3 (b)）．また，手続き B での x が大域変数であることは，図 3.3 (d) の記号表で分かる．このとき，手続き A の処理が終った時点で記号表から x を削除していないと，局所変数として誤った番地が割り当てられてしまうことに注意しよう．

以上に述べたように，記号表の実現には，各記号のスタックでの位置，大域変数か局所変数かの別を記録するための表を実現するためのデータ構造が必要である．また，表に記号を登録する操作，表から記号を検索する操作，局所変数であるエントリを削除する操作，などを実現することが必要である．この記号表により変数の有効範囲の機構を実現できる．

局所変数は大域変数と異なり，参照する際，代入して値を更新する際に，毎回，新しいレジスタに割り当てる．よって，代入して更新される際にも記号表を更新する．ただし，基本ブロックないで初めて変数を参照する際には，その基本ブロックにジャンプしてきた基本ブロッック内でのレジスタ番号を参照する必要がある．そのようなジャンプしてくる基本ブロックが複数ある場合には ϕ 関数を使用するなど記号表の管理を工夫する必要がある．

課題 3

図 3.3 のように，変数名，変数の最新のレジスタ番号，大域変数または局所変数の 3 種類のデータで構成される記号表を作成せよ．また，記号表に対する以下の操作関数を実現せよ．

- 記号表への変数・手続きの登録（insert）
- 変数・手続きの検索（lookup）
- 記号表から局所変数の削除（delete）

最も単純な実現方法は，記号表を構造体を利用した線形リストで表現し，そのリストをスタックとみなして登録をスタックへのプッシュ，削除をポップとして実現することである．なお，以降の説明のため本指導書では以下の列挙体を宣言して記号表を作成したとする．

```
/* 記号表の管理 + 変数・定数の区別用 */
typedef enum {
  GLOBAL_VAR, /* 大域変数 */
  LOCAL_VAR,  /* 局所変数 */
  PROC_NAME,  /* 手続き   */
  CONSTANT    /* 定数     */
} Scope;
```

手続き名を扱う理由は，手続き・関数を扱えるように拡張する際に，手続き・関数の引数個数や返り値の参照に手続き名・関数名を記録しておくことが必要になるためである．

ソースプログラムで変数が宣言された時点で記号表に要素を追加する関数，変数が呼び出された時点で変数を検索する関数，手続きの最後でその手続きの局所変数すべてを削除する関数を呼び出すコードを，課題 2 で作成された parser.y に追加せよ．各関数内で関数が呼び出されたことを示すコメントを出力させて，記号表が適切に操作されたことがわかるようにせよ．すなわち，要素を追加する関数では記号表の全ての要素の全てのデータ（変数・手続き名と大域・局所変数もしくは手続きの区別）を，要素を検索する関数では検索された変数の種別（大域・局所変数もしくは手続きの区別）を，そして要素を削除する関数では削除後の全ての要素の全てのデータを出力する．なお，本課題では，記号表に操作するデータは変数名・手続き名，大域変数か局所変数か関数の区別とする．

サンプルプログラム PL0A，PL0B，PL0C, PL0D をソースプログラムとして入力し動作を確認せよ．作成したプログラムにおける記号表の構造と，各操作関数を挿入する位置について解説し，レポートとして提出せよ．

第4章　コード生成

4.1　コード生成の方針

本実験におけるコンパイラでは構文解析をしながらコードを生成する．したがって，構文規則のアクション部を変更してコード生成の機能を付加すればよい．

コード生成では最終的にテキストファイル（拡張子は.ll）を生成する．後述するように後から決定するアドレスが存在するため，生成したコードを後から生成できるようにしておく必要がある．命令によって文字数などが異なるために，命令を生成するたびにファイルにコードを出力してしまうと後から書き換えることが困難になる．そこで，生成するコードを表現するデータ構造を用意し，その構造に沿ってコード生成し，最後にその構造を LLVM-IR の構文に従って書き出すこととする．

4.2　LLVM IR コードの内部表現

まず，生成する LLVM-IR の命令を表現するために図 4.1 のように命令名やその引数が取る情報を列挙定数として宣言する．列挙体 `LLVMcommand` は命令名を定義しており，列挙体 `Cmptype` は比較演算命令 `icmp` の第 1 引数として与えられる比較の種類（`eq`，`ne`，`sgt` など）を表す．

関数定義の中身は命令の列であるので，図 4.2 のように命令の列を線形リストで表現する．構造体 `llvmcode` のメンバ `command` は命令名（すなわち，命令の種類）を表し，メンバ `args` は `command` が保持する命令に対する引数を表す．命令ごとに引数の数や種類が異なるために共用体により，命令に応じた引数を保持するようにしている．メンバ `next` は次の命令へのポインタであり，このポインタで命令の列（線形リスト）を構成する．コード生成時には大域変数 `codehd` と `codetl`（それぞれ線形リストの先頭と末尾の命令へのポインタ）を利用する．

```
/* LLVM 命令名の定義 */
typedef enum {
  Alloca,   /* alloca */
  Store,    /* store  */
  Load,     /* load   */
  BrUncond, /* br     */
  BrCond,   /* brc    */
  Label,    /* label  */
  Add,      /* add    */
  Sub,      /* sub    */
  Icmp,     /* icmp   */
  Ret       /* ret    */
} LLVMcommand;

/* 比較演算子の種類 */
typedef enum {
  EQUAL, /* eq （==）*/
  NE,    /* ne （!=）*/
  SGT,   /* sgt （>, 符号付き） */
  SGE,   /* sge （>=, 符号付き）*/
  SLT,   /* slt （<, 符号付き） */
  SLE    /* sle （<=, 符号付き）*/
} Cmptype;
```

図 4.1: LLVM-IR の命令に対する列挙定数の宣言

```
typedef struct llvmcode {
  LLVMcommand command; /* 命令名 */
  union { /* 命令の引数 */
    struct { /* alloca */
      Factor retval;
    } alloca;
    struct { /* store  */
      Factor arg1;  Factor arg2;
    } store;
    struct { /* load   */
      Factor arg1;  Factor retval;
    } load;
    struct { /* br     */
      int arg1;
    } bruncond;
    struct { /* brc    */
      Factor arg1;  int arg2;  int arg3;
    } brcond;
    struct { /* label  */
      int l;
    } label;
    struct { /* add    */
      Factor arg1;  Factor arg2;  Factor retval;
    } add;
    struct { /* sub    */
      Factor arg1;  Factor arg2;  Factor retval;
    } sub;
    struct { /* icmp   */
      Cmptype type;  Factor arg1;  Factor arg2;  Factor retval;
    } icmp;
    struct { /* ret    */
      Factor arg1;
    } ret;
  } args;
  struct llvmcode *next; /* 次の命令へのポインタ */
} LLVMcode;

LLVMcode *codehd = NULL; /* 命令列の先頭のアドレスを保持するポインタ */
LLVMcode *codetl = NULL; /* 命令列の末尾のアドレスを保持するポインタ */
```

図 4.2: LLVM-IR の命令を表現する構造体

ヒント

式からコード生成する際には式を逆ポーランド記法における評価のように解釈するとよい．例えば，x+y に対してコードを生成する際には x の値をロードし，次に y の値をロードし，最後に，加算を行う．よって，式を逆ポーランド記法 x y +と読み替えてスタックにプッシュし，ポップしながらコードを生成する．なお，生成の際には演算子の引数だけがわかればよいので整数，変数をロードして得たレジスタ番号さえ分かれば十分である．そこで，整数もしくはレジスタ番号を保持するスタックを図 4.3 のように準備する．

生成するコードは関数定義の列とみなせる．そこで，図 4.4 のように関数定義の列を線形リストで表現する．構造体 fundecl のメンバ fname は関数名を表し，arity はその関数の引数個数を表す．メンバ args は仮引数の名称を保持し，codes はその関数の定義となる命令列（線形リスト）の先頭のアドレスを保持するポインタである．メンバ next は次の関数定義へのポインタであり，このポインタで関数定義の列（線形リスト）を構成する．コード生成時には大域変数 declhd と decltl（それぞれ線形リストの先頭と末尾の命令へのポインタ）を利用する．

4.3 コード生成の基本

図 4.5 に expression 部の加算式に対してコード生成の機能を付加した例を示す．いくつかの処理は他の命令の処理でも同様であるので，必要に応じて関数を作成し，その関数を呼び出すとよい．例えば，命令のためにメモリを確保する処理は繰り返し利用される．

```
tmp = (LLVMcode *)malloc(sizeof(LLVMcode)); /*メモリ確保 */
tmp->next = NULL;           /* 次の命令へのポインタを初期化 */
```

また，生成した命令をリストに追加する処理も繰り返し利用される．

```
if( codetl == NULL ){ /* 解析中の関数の最初の命令の場合 */
  if( declhd == NULL ){   /* 解析中の関数がない場合 */
    /* 関数宣言を処理する段階でリストが作られているので，ありえない */
    fprintf(stderr,"unexpected error\n");
  }
  declhd->codes = tmp;   /* 関数定義の命令列の先頭の命令に設定 */
  codehd = codetl = tmp; /* 生成中の命令列の末尾の命令として記憶 */
} else { /* 解析中の関数の命令列に 1 つ以上命令が存在する場合 */
  codetl->next = tmp; /* 命令列の末尾に追加 */
  codetl = tmp;       /* 生成中の命令列の末尾の命令として記憶 */
}
```

```
/* 変数もしくは定数の型 */
typedef struct {
  Scope type;       /* 変数（のレジスタ）か整数の区別 */
  char vname[256]; /* 変数の場合の変数名 */
  int val;          /* 整数の場合はその値，変数の場合は割り当てたレジスタ番号 */
} Factor;

/* 変数もしくは定数のためのスタック */
typedef struct {
  Factor element[100];  /* スタック（最大要素数は 100 まで） */
  unsigned int top;     /* スタックのトップの位置          */
} Factorstack;

Factorstack fstack; /* 整数もしくはレジスタ番号を保持するスタック */

void init_fstack(){ /* fstack の初期化 */
  fstack.top = 0;
  return;
}

Factor factorpop() {
  Factor tmp;
  tmp = fstack.element[fstack.top];
  fstack.top --;
  return tmp;
}

void factorpush(Factor x) {
  fstack.top ++;
  fstack.element[fstack.top] = x;
  return;
}
```

図 4.3: 整数もしくはレジスタ番号を表す構造体とそれを保持するスタック

```
/* LLVM の関数定義 */
typedef struct fundecl {
  char fname[256];      /* 関数名                          */
  unsigned arity;       /* 引数個数                        */
  Factor args[10];      /* 引数名                          */
  LLVMcode *codes;      /* 命令列の線形リストへのポインタ */
  struct fundecl *next; /* 次の関数定義へのポインタ        */
} Fundecl;

/* 関数定義の線形リストの先頭の要素のアドレスを保持するポインタ */
Fundecl *declhd = NULL;
/* 関数定義の線形リストの末尾の要素のアドレスを保持するポインタ */
Fundecl *decltl = NULL;
```

図 4.4: 関数定義を表す構造体

expression 部では，現在解析している式を評価し，その結果をスタックの先頭に積むようなコードを生成する．例えば，変数 x の相対番地が 1 であるとき，文 x:=2+4*5-7; に対しては，図 4.6 のような目的プログラムが生成される．ここで，スタックの先頭に常に expression の評価結果が格納されることに注意しよう．

```
expression
        : term
        | PLUS term
        | MINUS term
        | expression PLUS term
        {
          LLVMcode *tmp;              /* 生成した命令へのポインタ */
          Factor arg1, arg2, retval; /* 加算の引数・結果 */
          tmp = (LLVMcode *)malloc(sizeof(LLVMcode)); /*メモリ確保 */
          tmp->next = NULL;           /* 次の命令へのポインタを初期化 */
          tmp->command = Add;         /* 命令の種類を加算に設定 */
          arg2 = factorpop();         /* スタックから第 2 引数をポップ */
          arg1 = factorpop();         /* スタックから第 1 引数をポップ */
          retval.type = LOCAL_VAR;    /* 結果を格納するレジスタは局所 */
          retval.val = cntr;          /* 新規のレジスタ番号を取得 */
          cntr ++;                    /* カウンタをインクリメント */
          (tmp->args).add.arg1 = arg1; /* 命令の第 1 引数を指定 */
          (tmp->args).add.arg2 = arg2; /* 命令の第 2 引数を指定 */
          (tmp->args).add.retval = retval; /* 結果のレジスタを指定 */
          if( codetl == NULL ){ /* 解析中の関数の最初の命令の場合 */
            if( declhd == NULL ){   /* 解析中の関数がない場合 */
              /* 関数宣言を処理する段階でリストが作られているので, ありえない */
              fprintf(stderr,"unexpected error\n");
            }
            declhd->codes = tmp;   /* 関数定義の命令列の先頭の命令に設定 */
            codehd = codetl = tmp; /* 生成中の命令列の末尾の命令として記憶 */
          } else { /* 解析中の関数の命令列に 1 つ以上命令が存在する場合 */
            codetl->next = tmp; /* 命令列の末尾に追加 */
            codetl = tmp;       /* 命令列の末尾の命令として記憶 */
          }
          factorpush( retval ); /* 加算の結果をスタックにプッシュ */
        }
        | expression MINUS term
        ;
```

図 4.5: expression 部

```
%1 = mul nsw i32 4, 5  ; 4 * 5
%2 = add nsw i32 2, %1 ; 2 + 4 * 5
%3 = sub nsw i32 %2, 7 ; 2 + 4 * 5 - 7
```

図 4.6: x := 2 + 4 ∗ 5 − 7; に対する目的プログラム（x の番地が 1 の場合）

```
program
        : PROGRAM IDENT SEMICOLON outblock PERIOD
          {
            ...
            displayLlvmfundecl( declhd );
          }
        ;
```

図 4.7: 生成したコードをファイルに書き出す処理

課題 4

PL-0 の任意のプログラムを生成する前に，図 4.6 のような四則演算式をコンパイルする簡単なコンパイラを作成せよ．課題 3 で作成したプログラムの適切な箇所に，コードを生成する処理を追加せよ．

コード生成は構文解析をしながら内部表現としてコードを生成し，構文解析が成功終了した時点で内部表現のコードを目的プログラムの構文としてファイルに書き出せばよい．具体的には，付録 E.3 の program の生成規則に対する処理の最後でファイルに書き出す（図 4.7,4.8）．なお，図 4.8 の関数は標準出力に生成コードを生成する関数となっているので，ファイルに書き出すように変更すること．

コンパイラが完成したら，課題 3 の図 3.1 (a) のプログラム（「リソース」のサンプルプログラム ex1.p）をソースプログラムとしてコンパイルし，図 3.2 のようなコードが生成されるか確認せよ．コードを書き出すファイルの名前は result.ll とすることとする．

また、目的プログラムを実行できる仮想機械（LLVM のインタプリタ lli コマンド）（付録 G 参照）が用意されているので，出力された目的プログラムを仮想機械に入力し実行せよ．コンパイラが正常に動作していることを確認すること．

生成したコードの種類と，構文中でのコードの生成箇所を解説し，レポートとして提出せよ．

```
void displayFactor( Factor factor ){
  switch( factor.type ){
  case GLOBAL_VAR:
    printf("@%s", factor.vname );
    break;
  case LOCAL_VAR:
    printf("%%%d", factor.val );
    break;
  case CONSTANT:
    printf("%d", factor.val );
    break;
  default:
    break;
  }
  return;
}

void displayLlvmcodes( LLVMcode *code ){
  if( code == NULL ) return;
  printf("  ");
  switch( code->command ){
  case Alloca:
    displayFactor( (code->args).alloca.retval );
    printf(" = alloca i32, align 4\n");
    break;
  case ...:
    ...
  default:
    break;
  }
  displayLlvmcodes( code->next );
}

void displayLlvmfundecl( Fundecl *decl ){
  if( decl == NULL ) return;
  printf("define i32 @%s() {\n", decl->fname );
  displayLlvmcodes( decl->codes );
  printf("}\n");
  if( decl->next != NULL ) {
    printf("\n");
    displayLlvmfundecl( decl->next );
  }
  return;
}
```


図 4.8: 生成コードを印字する関数

4.4 後から決定するアドレスについて（バックパッチ）

if 文，while 文では，条件付ジャンプ命令と無条件ジャンプ命令を適宜組合せてコードを生成する．このとき問題となるのは，これらの飛び先アドレスがコード生成の後で決定されることである．このような場合，厳密な意味では 1 パス・コンパイラとしては処理できない．この解決策として，br 命令へのポインタを記憶しておけばよい．ソースプログラムにはジャンプ命令がないので，ジャンプが発生するのは if 文と while 文に限られ，これらの構造は階層的である．よって，br 命令のアドレスを格納するスタックを用意しておけば，あたかも 1 パス・コンパイラのようにジャンプ先を指定することができる．

4.5 エラー処理について

最も簡単なエラー処理は，構文グラフに従わないプログラムが入力されたときに，間違っていることを表示してコンパイルを中止することである．しかし，これではユーザにとっては非常に使いづらいコンパイラとなってしまう．最低限，ユーザにとって欲しい情報は，プログラムのどこでどういう種類の誤りをしたのかということである．さらに，その場合にどういう誤りかが完全に分かれば，一時的に訂正を試みて，さらにコンパイルを続け，他の誤りも指摘できる場合もある．しかし，一般に誤りの意味的な同定は非常に困難である．

エラー処理のために，字句解析プログラム，構文解析プログラム，記号表管理プログラムなどが複雑になる．例えば，字句解析プログラムでは，今プログラムのどこを解釈しているか，ということを常に保持していなければならない．また，その他のエラーに対して，適切なエラーメッセージを出すことも容易な作業とは言えない．

課題 5

PL-0 コンパイラを完成させよ．課題 4 で作成したプログラムに対し，手続き，制御文，入出力が処理できるようなコードを追加せよ．コンパイラが完成したら，サンプルプログラムの PL0A，PL0B，PL0C, PL0D をソースプログラムとして入力しコンパイルせよ．

read，write については C 言語で scanf，printf 関数を用いた際に生成される LLVM IR コードを生成することを目標とし，レポートではその目標コードを例を用いて説明せよ．

for ループと手続き呼び出しの実現方法を解説し，レポートとして提出すること．各処理について，生成したコードと構文中でのコードの生成箇所について記述すること．

4.6 引数の値渡しが可能な手続きへの拡張と関数呼び出しの実現

手続きの引数を実現するために，言語仕様を以下のように変更する．

$$\begin{aligned}\langle proc\_call\_statement\rangle \;::=\; & \langle proc\_call\_name\rangle \\ \mid\; & \langle proc\_call\_name\rangle \text{ `(' } \langle arg\_list\rangle \text{ `)'}\end{aligned}$$

$$\begin{aligned}\langle proc\_decl\rangle \;::=\; & \text{`}\texttt{procedure}\text{'} \langle proc\_name\rangle \text{ `;' } \langle inblock\rangle \\ \mid\; & \text{`}\texttt{procedure}\text{'} \langle proc\_name\rangle \text{ `(' } \langle id\_list\rangle \text{ `)' `;' } \langle inblock\rangle\end{aligned}$$

手続きを呼び出すときに用いられる引数を**実引数**，手続き宣言において呼び出される側の引数を**仮引数**という．値渡し（call by value）[1]では，手続きが呼び出されると，実引数の式が評価され，その値が仮引数に代入されて，制御が手続きのプログラムに移る．したがって，仮引数は通常の局所変数と同様に扱えばよい．ただし，その初期値は実引数によって決定される．

手続き呼出しにおける引数の値受渡しの実現について説明する．図 4.9 (a) のようなソースプログラムに対しては，図 4.9 (b) のような記号表と図 4.9 (c) のような目的プログラムが生成される．

LLVM IR では，関数呼び出しは `call` 命令によって実現される．`call` 命令は，`@`で始まる識別子で表現される関数に対し，`()` で引数の型と値を指定する．引数を計算し，例えば`%2` に計算結果が格納されているとき，

```
%2 = load i32, i32* @i, align 4
call i32 @twice(i32 %2)
```

で，引数を渡して `twice` を呼び出すことができる．一方，呼び出された関数内では，`ret` 命令を実行することで，呼び出し元へと計算結果を返すことができる．

課題 6

手続き呼出しのときに引数を値渡しすることができるよう言語仕様を PL-1 に拡張し，PL-1 のコンパイラを実現せよ．そして，サンプルプログラム `PL1A` を入力して実行せよ．なお，引数は複数でもよいこととする．

実現した手続きの引数の値渡し方法を説明し，レポートとして提出せよ．生成したコードとその生成箇所について記述すること．

関数呼出しの場合は，引数の受渡しのメカニズムは手続き呼出しと同様である．しかし，戻り値のための領域が必要になる．LLVM IR では，`i32` 型の関数の先頭で，

```
%1 = alloca i32, align 4
```

[1] この他に**参照受渡し**（call by reference），**名前受渡し**（call by name）などがある．

```
program test;
var i, r;
procedure twice(n)
begin
   r := n + n;
end;
begin
   i := 2;
   twice(i);
   write(r)
end.
```

i	–	glob
r	–	glob
twice	–	proc
n	1	loc

(a) ソースプログラム　(b) 6 行目まで解析したときの記号表

```
@r = common global i32 0, align 4
@i = common global i32 0, align 4

define void @twice(i32) #0 {
 %2 = alloca i32, align 4
 store i32 %0, i32* %2, align 4
 %3 = load i32, i32* %2, align 4
 %4 = load i32, i32* %2, align 4
 %5 = add nsw i32 %3, %4
 store i32 %5, i32* @r, align 4
 ret void
}

define i32 @main() #0 {
 %1 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 2, i32* @i, align 4
 %2 = load i32, i32* @i, align 4
 call void @twice(i32 %2)
 %3 = load i32, i32* @r, align 4
 call void @write(i32 %3)
 ret i32 0
}
```

(c) 目的プログラム

図 4.9: 手続き呼出しのメカニズム

のように領域を確保している．

C などでは，戻り値のための return 文が存在するが，PL-2 では，関数と同じ名前の変数に値を代入することで，戻り値を実現している．そのような関数名と同じ名前の変数は戻り値を格納するためだけに使用される，すなわち，その変数を含むような式を扱わないこととする．一方，LLVM IR では，上記で確保した領域から値を読み出し，ret 命令を用いて戻り値を返すようになっている．

課題 7

関数を含むプログラムにも対応できるように言語仕様を PL-2 に拡張せよ．そして，PL-2 のコンパイラを実現せよ．言語仕様は，ソースプログラム PL2A，PL2B を参照のこと．なお，引数は複数でもよいこととする．ソースプログラム PL2A，PL2B を入力し実行せよ．

拡張した言語仕様および関数の実現方法を，レポートとして提出せよ．関数を実現するために生成したコードに関して説明すること．また，記号表を拡張した場合，拡張した構造についても記述すること．

4.7 配列の実現

1 次元の配列を扱うことができるよう言語仕様を（PL-3）に拡張する．

PL-3 における配列の定義，記述法は以下のようである．

```
var a[5..10];          ← a[5] から a[10] までの配列宣言
a[5] := 5;
b := a[i] + 10;
```

配列が通常の変数と違う点は，添字があるために，宣言されたときに領域を複数確保しなければならない点と，参照のときに相対修飾しなければならない点である．

LLVM IR では，getelementptr 命令を用いて，配列の指定した番号の要素のアドレスを得る．アドレスが得られれば，通常の変数と同様に扱うことができる[2]．

課題 8

PL-3 のコンパイラ pl3 を実現せよ．上記の記述法を満足するよう構文規則を拡張すること．付録にある「プログラム例」の PL3A と PL3B を入力して実行せよ．

配列の実現方法に関して説明し，レポートとして提出せよ．配列を実現するために拡張した記号表の構造，生成したコードに関して記述すること．

[2] プログラムで，配列の添字が 1 から始まっているので，配列要素の相対番地を計算するときに 1 を引いている．

```
program ARRAY;
var a[1..10], i, s;
begin
   i := 1
   while i <= 10 do
   begin
      a[i] := i; ← (i)
      i := i + 1
   end;
   i := 1;
   while i <= 10 do
   begin
      s := a[i] + s; ← (i)
      i := i + 1
   end
end.
```

(a) ソースプログラム

```
%6 = load i32, i32* @i, align 4
%7 = sub nsw i32 %6, 1
%8 = load i32, i32* @i, align 4
%9 = sub nsw i32 %8, 1
%10 = sext i32 %9 to i64
%11 = getelementptr inbounds [10 x i32], [10 x i32]* @a, i64 0, i64 %10
store i32 %7, i32* %11, align 4
```

(b) (i) の部分のコード

```
%19 = load i32, i32* @i, align 4
%20 = sub nsw i32 %19, 1
%21 = sext i32 %20 to i64
%22 = getelementptr inbounds [10 x i32], [10 x i32]* @a, i64 0, i64 %21
%23 = load i32, i32* %22, align 4
%24 = load i32, i32* @s, align 4
%25 = add nsw i32 %23, %24
store i32 %25, i32* @s, align 4
```

(c) (ii) の部分のコード

図 4.10: 配列の実現

課題 9（選択）

forward 文を含むプログラムにも対応できるように言語使用を拡張せよ (PL-4)．そして，PL-4 のコンパイラ `pl4` を実現せよ．

forward 文は，手続き同士で相互参照をするための機構である．本実験で製作するコンパイラは 1 パス・コンパイラであるため，呼び出される手続きや関数は，呼び出す手続きや関数よりも，プログラム内で前に書かれている必要がある．forward 文は，C 言語のプロトタイプ宣言のように，呼び出す手続きや関数よりも後ろに記述されている呼び出される手続きや関数の宣言部分を，あらかじめ宣言しておくことで，その制約を緩和する機能を持った文である．

forward 文の構文を決定する際，procedure と function の違い，引数の有無，などを考慮する必要がある．また，呼び出そうとしている手続きや関数が forward 宣言されているだけか，あるいは実際にコード生成済か，などを管理するよう記号表を拡張する必要がある．

第5章　コード最適化

コード最適化とは，

- 処理速度の向上
- 目的コードの容量削減

を目的とした処理である．以下に最適化の例を挙げる．

5.1　命令の置き換え

一般的に，数値演算よりもビット演算のほうが効率が良いことが多い．そのため，例えば 2^n 倍する演算を n 桁シフトに置き換えることにより，計算を高速化できることが期待できる．図 5.1 に，この最適化により生成されるコードの例を示す．

5.2　定数伝播（Constant propagation）

定数伝播とは，基本ブロック内で行われる局所最適化の一つであり，定数式をコンパイル時に事前に計算しておき，出力するコードを単純化する最適化方法である．定数畳み込み（Constant folding）とも呼ぶ．図 5.2 に，この最適化により生成されるコードの例を示す．

5.3　デッドコード削除（Dead-code elimination）

デッドコード削除とは，不要なコード（冗長なコード，到達不可能なコード）を削除する最適化方法である．図 5.3 に，この最適化により生成されるコードの例を示す．

```
program OPT1;
var x;
begin
   x = 10;
   x = x * 8;
   x = x / 4;
   x = x * 32;
end.
```

(i) ソースプログラム

```
define i32 @main() #0 {
  store i32 10, i32* @x, align 4
  %1 = load i32, i32* @x, align 4
  %2 = shl i32 %1, 3
  store i32 %2, i32* @x, align 4
  %3 = load i32, i32* @x, align 4
  %4 = ashr i32 %3, 2
  store i32 %4, i32* @x, align 4
  %5 = load i32, i32* @x, align 4
  %6 = shl i32 %5, 4
  store i32 %6, i32* @x, align 4
  ret i32 0 }
```

(ii) LLVM コード

図 5.1: 命令置き換えの例

```
program OPT1;
var x;
begin
   x = 4 + 3 * 2 - 1
end.
```

(i) ソースプログラム

```
define i32 @main() #0 {
  store i32 9, i32* @x, align 4
  ret i32 0
}
```

(ii) LLVM コード

図 5.2: 定数伝播（定数畳み込み）の例

```
program OPT2;
var x, y;
function afunc(n);
begin
    x = n; <- 無駄なコード
    x = n + 1;
    fact := x * 2
    x = n * 10; <- 無駄なコード
end;
begin
    call(2)
end.
```

(i) ソースプログラム

```
define i32 @afunc(i32) #0 {
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    store i32 %0, i32* %2, align 4
    %4 = load i32, i32* %2, align 4
    %5 = add nsw i32 %4, 1
    store i32 %5, i32* %3, align 4
    %6 = load i32, i32* %3, align 4
    ret i32 %6
}

define i32 @main() #0 {
  %1 = call i32 @afunc(i32 2)
  ret i32 0
}
```

(ii) LLVM コード

図 5.3: デッドコード削除の例

課題 10（コード最適化）

前述の最適化のうち 1 つ以上を選び，実装せよ．また，その最適化方法について調べ，詳細を説明せよ．上記以外の最適化を実施しても良いが，その場合はその旨を明記すること．

最適化以前のコードと最適化後のコードを比較し，考察せよ．特に，生成したコードのどの部分が最適化されたのかについて説明し，実行結果が変わらないことを確認せよ．

関連図書

[1] カーニハン，リッチー／石田晴久訳:“プログラミング言語 C 第 2 版”，共立出版（1989）.

[2] 五月女 健治：“yacc/lex：プログラムジェネレータ on UNIX”，啓学出版（1996）.

[3] 柏木 餅子，風薬：“きつねさんでもわかる LLVM：コンパイラを自作するためのガイドブック”，インプレスジャパン（2013）.

[4] 出村 成和：“LLVM/Clang 実践活用ハンドブック”，ソシム（2014）.

付 録 A　PL-0 の構文規則（BNF 記法）

⟨*program*⟩ ::= ‘`program`’ ‘`IDENT`’ ‘`;`’ ⟨*outblock*⟩ ‘`.`’

⟨*outblock*⟩ ::= ⟨*var_decl_part*⟩ ⟨*subprog_decl_part*⟩ ⟨*statement*⟩

⟨*var_decl_part*⟩ ::= ⟨*var_decl_list*⟩ ‘`;`’
| /* empty */

⟨*var_decl_list*⟩ ::= ⟨*var_decl_list*⟩ ‘`;`’ ⟨*var_decl*⟩
| ⟨*var_decl*⟩

⟨*var_decl*⟩ ::= ‘`var`’ ⟨*id_list*⟩

⟨*subprog_decl_part*⟩ ::= ⟨*subprog_decl_list*⟩ ‘`;`’
| /* empty */

⟨*subprog_decl_list*⟩ ::= ⟨*subprog_decl_list*⟩ ‘`;`’ ⟨*subprog_decl*⟩
| ⟨*subprog_decl*⟩

⟨*subprog_decl*⟩ ::= ⟨*proc_decl*⟩

⟨*proc_decl*⟩ ::= ‘`procedure`’ ⟨*proc_name*⟩ ‘`;`’ ⟨*inblock*⟩

⟨*proc_name*⟩ ::= ‘`IDENT`’

⟨*inblock*⟩ ::= ⟨*var_decl_part*⟩ ⟨*statement*⟩

⟨*statement_list*⟩ ::= ⟨*statement_list*⟩ ‘`;`’ ⟨*statement*⟩
| ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*assignment_statement*⟩
| ⟨*if_statement*⟩
| ⟨*while_statement*⟩
| ⟨*for_statement*⟩
| ⟨*proc_call_statement*⟩
| ⟨*null_statement*⟩
| ⟨*block_statement*⟩
| ⟨*read_statement*⟩
| ⟨*write_statement*⟩

⟨*assignment_statement*⟩ ::= 'IDENT' ':=' ⟨*expression*⟩

⟨*if_statement*⟩ ::='if' ⟨*condition*⟩ 'then' ⟨*statement*⟩ ⟨*else_statement*⟩

⟨*else_statement*⟩ ::= 'else' ⟨*statement*⟩
| /* empty */

⟨*while_statement*⟩ ::= 'while' ⟨*condition*⟩ 'do' ⟨*statement*⟩

⟨*for_statement*⟩ ::= 'for' 'IDENT' ':=' ⟨*expression*⟩
'to' ⟨*expression*⟩ 'do' ⟨*statement*⟩

⟨*proc_call_statement*⟩ ::= ⟨*proc_call_name*⟩

⟨*proc_call_name*⟩ ::= 'IDENT'

⟨*block_statement*⟩ ::= 'begin' ⟨*statement_list*⟩ 'end'

⟨*read_statement*⟩ ::= 'read' '(' 'IDENT' ')'

⟨*write_statement*⟩ ::= 'write' '(' ⟨*expression*⟩ ')'

⟨*null_statement*⟩ ::= /* empty */

⟨*condition*⟩ ::= ⟨*expression*⟩ '=' ⟨*expression*⟩
| ⟨*expression*⟩ '<>' ⟨*expression*⟩
| ⟨*expression*⟩ '<' ⟨*expression*⟩
| ⟨*expression*⟩ '<=' ⟨*expression*⟩
| ⟨*expression*⟩ '>' ⟨*expression*⟩
| ⟨*expression*⟩ '>=' ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*term*⟩
| '+' ⟨*term*⟩
| '-' ⟨*term*⟩
| ⟨*expression*⟩ '+' ⟨*term*⟩
| ⟨*expression*⟩ '-' ⟨*term*⟩

⟨*term*⟩ ::= ⟨*factor*⟩
| ⟨*term*⟩ '*' ⟨*factor*⟩
| ⟨*term*⟩ 'div' ⟨*factor*⟩

⟨*factor*⟩ ::= ⟨*var_name*⟩
| 'NUMBER'
| '(' ⟨*expression*⟩ ')'

⟨*var_name*⟩ ::= 'IDENT'

⟨*arg_list*⟩ ::= ⟨*expression*⟩
| ⟨*arg_list*⟩ ',' ⟨*expression*⟩

⟨*id_list*⟩ ::= ‘IDENT’
| ⟨*id_list*⟩ ‘,’ ‘IDENT’

“⟨” と “⟩” で囲まれたものを**構文要素**と呼ぶ．BNF 記法では，::= の左辺にある構文要素が右辺によって定義されていることを表している．“|” は「または」という意味である．例えば，⟨*id_list*⟩ は，一つの IDENT か，または ⟨*id_list*⟩ と IDENT を ‘,’ で繋げたものであると定義されている．したがって，⟨*id_list*⟩ は，

IDENT
IDENT, IDENT
IDENT, IDENT, IDENT
...

となり，識別子のリストを表すことになる．

付 録B　プログラム例

B.1　PL-0 プログラム

PL0B：素数の生成（その 1）

```
program PL0B;
var n, x;
procedure prime;
var m;
begin
    m := x div 2;
    while x <> (x div m) * m do
        m := m - 1;
    if m = 1 then
        write(x)
end;
begin
    read(n);
    while 1 < n do
    begin
        x := n;
        prime;
        n := n - 1
    end
end.
```

PL0C：素数の生成（その 2）

```
program PL0C;
var n, x, i;
procedure prime;
var m;
begin
```

```
    m := x div 2;
    while x <> (x div m) * m do
        m := m - 1;
    if m = 1 then
        write(x)
end;
begin
    read(n);
    for i := 2 to n do
    begin
        x := i;
        prime;
    end
end.
```

PL0D：階乗の計算（その 1）

```
program PL0D;
var n, temp;
procedure fact;
var m;
begin
    if n <= 1 then
        temp:=1
    else
    begin
        m:=n;
        n:=n-1;
        fact;
        temp:=temp*m
    end
end;
begin
    read(n);
    fact;
    write(temp);
end.
```

B.2　PL-1 プログラム

PL1A：階乗の計算（その 2）

```
program PL1A;
var n, temp;
procedure fact(n);
begin
    if n <= 1 then
        temp:=1
    else
    begin
        fact(n - 1);
        temp := temp * n
    end
end;
begin
    read(n);
    fact(n);
    write(temp)
end.
```

B.3　PL-2 プログラム

PL2A：階乗の計算（その 3）

```
program PL2A;
var n;
function fact(n);
    if n <= 0 then
        fact := 1
    else
        fact := fact(n - 1) * n;
begin
    read(n);
    write(fact(n))
end.
```

PL2B：**n** 乗の計算

```
program PL2B;
var m, n;
function power(m,n);
    if n <= 0 then
        power := 1
    else
        power := power(m,n - 1) * m;
begin
    read(m);
    read(n);
    write(power(m,n))
end.
```

B.4　PL-3 プログラム

PL3A：バブルソート

```
program PL3A;
var i, j, n, a[1..100];
procedure initialize(n);
var i;
begin
    for i := 1 to n do
        read(a[i])
end;
procedure swap(j);
var temp;
begin
    temp := a[j];
    a[j] := a[j+1];
    a[j+1] := temp
end;
begin
    read(n);
    if n <= 100 then
    begin
        initialize(n);
```

```
        i := n;
        while 1 <= i do
        begin
            j:=1;
            while j < i do
            begin
                if a[j] > a[j+1] then
                    swap(j);
                j := j + 1
            end;
            write(a[i]);
            i := i - 1
        end
    end
end.
```

PL3B：素数の生成（その **3**）

```
program PL3B;
var a[2..100],i,n;
procedure initialize;
var i;
    for i:= 2 to 100 do
        a[i] := 0;
procedure check(p);
var i;
begin
    i := p;
    while i <= 100 do
    begin
        a[i] := 1;
        i := i + p;
    end
end;
begin
    initialize;
    read(n);
    if n <= 100 then
    for i := 2 to n do
```

```
          if a[i] = 0 then
           begin
               write(i);
               check(i);
           end
  end.
```

PL4A：相互再帰

1 から n までの加算

```
program PL7A;
var sum, n;
forward procedure proc2;
procedure proc1;
begin
   sum := sum + n;
   proc2
end;
procedure proc2;
begin
   n := n - 1;
   if n > 0 then
       proc1
   else
     write(sum);
end;
begin
  sum :=0;
  read(n);
  proc1
end.
```

付 録C　目的プログラムの仕様

LLVM IR では，識別子は大域識別子（global identifiers）と局所識別子（local identifiers）の 2 種類からなる．大域識別子（関数名，大域変数名）は,「@」から始まり，局所識別子（レジスタ名）は「%」から始まる．さらに，識別子には 3 つの形式が存在する．

- 名前付き値（named values）: 例えば %foo，%FooBar．下記の正規表現で定義される文字列に相当する．

```
[%@][-a-zA-Z$._][-a-zA-Z$._0-9]*
```

- 名前無し値（unnamed values）: 非負整数で指定する． %12，@2，%44 など．
- 定数（constants）: 整数など．

以下に本実験で使用する各命令について，簡単に説明する．なお，説明では扱うデータの方は 32 ビットの整数（i32 型）に限定する．

alloca: 局所変数

```
%i = alloca i32, align 4
```

現在実行中の関数のスタックフレーム上に int 型の変数を確保し，そのメモリ番地を返す．i32 が型を表し，align 4 は，アラインメントを表す．上記の例では%iでメモリ番地を受け取っている．

global: 大域変数

```
@n = common global i32 0, align 4
```

i32 型の大域変数を確保し（0 で初期化），そのメモリ番地を返す．

load: ロード命令

r = load i32, i32* p, align 4

メモリ p が指すメモリ番地から値を読み出し，レジスタ r に格納する．load 直後の i32 は読み込む値の型，続く i32*はメモリ番地を表す型を示す．以下の例では，i32 型の大域変数@n から値を読み出し，レジスタ%2 へと格納している．

```
%2 = load i32, i32* @n, align 4
```

store: ストア命令

store i32 v, i32* p, align 4

i32 型の値 v を，p が指しているメモリ番地に格納している．以下の例では，i32 型の値 0 を@x のメモリ番地へストアしている．

```
store i32* 0, i32* @x, align 4
```

以下の例では，%2 に格納されている値を@y のメモリ番地へストアしている．

```
store i32* %2, i32* @y, align 4
```

算術演算

r = opr [nsw] i32 op_1, op_2

opr で指定された算術演算を引数 op_1，op_2 に対して行い，演算結果をレジスタ r に格納する．本実験で使用する算術演算は add（加算），sub（減算），mul（乗算），sdiv（除算）である．nsw は，No Signed Wrap の略で，オーバーフローが発生したときに，結果が Poison Value として扱われることを意味する．Poison Value とは，未定義の演算が行なわれた結果であることを示すラベルであり，それ以降の計算でその計算結果を使ったものにも Poison Value が伝播する．例えば，以下のように用いる．

```
%3 = add nsw i32 %2, 10

%4 = sub nsw i32 100, 1%3

%5 = mul nsw i32 %4, 2

%6 = sdiv i32 %5, %2
```

icmp: 整数比較命令

r = icmp $cond$ i32 op_1, op_2

$cond$ で指定された比較を行い，true/false の 2 値（型は i1）を返す．比較演算には以下の 10 種類がある．

$cond$	意味
eq	equal
ne	not equal
ugt	unsigned greater than
uge	unsigned greater or equal
ult	unsigned less than
ule	unsigned less or equal
sgt	signed greater than
sge	signed greater or equal
slt	signed less than
sle	signed less or equal

以下の例では，%6 と 10 が等しいか調べ，結果を%7 に格納している．

```
%7 = icmp eq i32 %6, 10
```

br: ジャンプ命令

br label $dest$

無条件でラベル $dest$ へとジャンプする．

br i1 $cond$, label l_1, label l_2

icmp を実行した結果が $cond$ に格納されているとき，その値が true のとき，ラベル l_1 へ，false のときラベル l_2 へとジャンプする．

call: 関数呼び出し

r = call i32 f([i32 v]*)

戻り値の型が i32 である関数 f を読み出し，結果をレジスタ r に格納する．引数を指定する場合は，i32 v という型と値の組をカンマ区切りで並べる．戻り値がない関数の場合は，型の指定を void にする．

call void f([i32 v]*)

ret: 復帰命令

```
ret i32 v
```

i32 型の値 v を返し，関数呼び出しを抜け，呼び出し元へ戻る．戻り値がない場合は，

```
ret void
```

と指定する．

label: ラベル

```
; <label>:l:
```

この命令の場所を l とラベル付けする．以下の例では%10 というラベルを定義している．

```
; <label>:10:
```

配列の宣言

以下の例では，i32 型の要素数 100 の配列をゼロ初期化した上で大域オブジェクトとしてメモリを確保し，メモリ番地を@a に格納している．

```
@a = common global [100 x i32] zeroinitializer, align 16
```

関数定義内で局所的に確保する場合は，common global ではなく alloca を用いる．

```
@a = alloca [100 x i32] zeroinitializer, align 16
```

getelementptr: 配列要素の参照

```
r = getelementptr inbounds [n x i32], [n x i32]* @a, i64 0, i64 i
```

要素数 n の配列 a の i 番目の要素を格納するメモリ番地をレジスタ r に格納する．例えば以下の例では，配列長 100 の大域変数@a の要素のうち，%9 番目の値のメモリ番地を得て，その番地（すわなち，a の%9 番目の要素）に 100 という値を格納している．

```
%10 = getelementptr inbounds [100 x i32], [100 x i32]* @a, i64 0, i64 %9
store i32 100, i %10, align 4
```

このとき，配列のインデックスの指定は i64 型であることに注意．

sext: 型の変換

r = sext ty_1 v to ty_2

ty_1 型の値 v を ty_2 型へと変換する（signed extend）．例えば，以下の例ではレジスタ%7 の値を i32 型から i64 型等の型変換し，レジスタ%8 に格納している．

```
%8 = sext i32 %7 to i64
```

配列要素の参照の例

配列の値を参照するには参照した配列要素のインデックスを i64 型に型変換した上で，配列要素の参照に用いる．

```
%8 = sext i32 %9 to i64
%9 = getelementptr inbounds [100 x i32], [100 x i32]* @a, i64 0, i64 %8
%10 = load i32, i32* %9, align 4
```

phi 関数

phi 関数を用いて，どこからジャンプしてきたかに応じて値を切り替える．以下の例では，ラベル%1 からジャンプしてきた場合は 0，ラベル%7 からジャンプしてきた場合は%9 の値を%11 へ格納する．

```
%11 = phi i32 [0, %1], [%9, %7]
```

関数

実行可能な LLVM IR コードは main 関数を 1 つ含む．main 関数は以下の様に定義できる．

```
define i32 @main(){

  ...

}
```

同様に，今回のソースプログラムに含まれる手続きや関数も関数（手続きは戻り値なし，関数は戻り値あり）として定義できる．戻り値なしの関数@procname は以下のように void 型として定義する．

```
define void @procname(){
```

```
  ...
}
```

これに対応する C 言語コードは以下の通りである．

```
void procname(){

  ...

}
```

i32 型の戻り値ありの関数@funcname は以下のように i32 型として定義する．

```
define i32 @funcname(...){
%1 = alloca i32, align 4

  ...

}
```

返り値用のレジスタが%1 として確保される．これに対応する C 言語コードは以下の通りである．

```
int funcname(...){

  ...

}
```

手続きや関数が引数を持つ場合，関数名に続く (...) の中に仮引数のリストを書く．仮引数のリストは，型と変数名からなる．仮引数は順に%0, %1, ... となる．その後，返り値用（以下では%3）及び引数用（以下では%4，%5）のレジスタが確保される．そして，store 命令を用いて値をコピーする．

```
define i32 @funcname(i32, i32){
   %3 = alloca i32, align 4
   %4 = alloca i32, align 4
   %5 = alloca i32, align 4
   store i32 %0, i32 %4, align 4
   store i32 %1, i32 %5, align 4

   ...

}
```

付 録D　Lex

本章では情報工学実験第 1 の指導書の付録 B に基づいて Lex について説明する．

D.1　LEX とは

ファイルの内容を読み込んで処理するプログラムは，一文字ずつ読み込みながら処理することが多い．しかし，C のソースプログラムのように文法にしたがって書かれたファイルを処理する場合には，連続した文字を一つの単語として認識してから，予約語の判定や識別子の判定などを行なう必要がある．LEX は，このような入力を単語単位で扱うプログラムの作成を支援する．

LEX は字句解析プログラムを生成するツールである．LEX によって生成されたプログラムは，ファイルからの入力を解析し，**トークン**と呼ばれる字句に分解する．LEX を用いて，トークンを入力単位として扱うプログラムを記述することで，記述が簡潔になり見通しの良いプログラムを作成することができる．

LEX の記述は三つの部分から構成される．それぞれ，定義部（definition section）, ルール部（rule section）, サブルーチン部（subroutine section）と呼ばれ，“%%” によって区切られる．この構成を図 D.1 に示す．定義部ではモードの宣言，パターンの名前の定義，C のプログラムの記述を行う．ルール部では，状態を表すモード, 入力パターン, アクションの三つを記述する．サブルーチン部には必要な関数を C で記述する．定義部とサブルーチン部は省略が可能である．サブルーチン部を省略するときは二つ目の “%%” も省略可能である．

D.1.1　記述方法

LEX を使った例題として次のようなプログラムを考える．

> Pascal で記述されたソースプログラムの単語の数を数える．ただし，コメントとコード内の単語はそれぞれ別々に数える．

なお，“:=” のような複数の記号からなる演算子も一つの単語とする．また，シングルクウォートで囲まれた文字列は単語に分解しないで，その文字列全体を一つの単語として数える．スペース，タブ，改行や Pascal の中で意味を持たない記号は単語として扱わない．

```
%{

#include <stdio.h>

%}

JAPANESE_ESC  (\033\044\102)
ASCII_ESC     (\033\050\102)

%Start      ASCII JAPANESE
```

定義部

```
%%
<ASCII>{JAPANESE_ESC} {  /* Search japanese_esc[] in ASCII mode. */
    BEGIN(JAPANESE);  /* Move to JAPANESE mode. */
}
<ASCII>. {
    putchar(yytext[0]);
}

<JAPANESE>{ASCII_ESC} {  /* Search ascii_esc[] in JAPANESE mode. */
    BEGIN(ASCII);  /* Move to ASCII mode. */
}
<JAPANESE>. {
    putchar(0x80 | yytext[0]);
}

<ASCII>\n {
    putchar(yytext[0]);
}
```

ルール部

```
%%
int main()
{
    BEGIN(ASCII);             /* Start in ASCII mode. */
    yylex();

    return 0;
}
```

サブルーチン部

図 D.1: LEX の記述構成

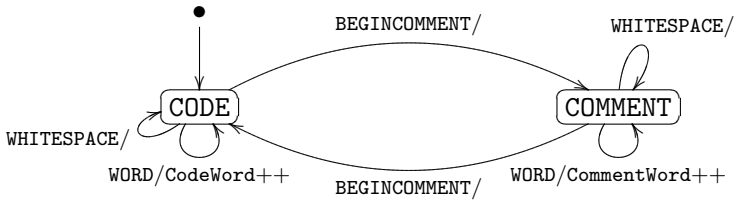

図 D.2: 状態遷移図

状態

このプログラムには二つの状態が存在する．一つは「コードの処理（CODE)」，もう一つは「コメントの処理（COMMENT)」である．LEX では状態を**モード**と呼び，その記述で用いるモードを定義部の中で次のように記述する．

```
%Start CODE COMMENT
```

また，入力のトークンを次の 4 種類に分類して扱う．

```
WORD, WHITESPACE, BEGINCOMMENT, ENDCOMMENT
```

WHITESPACE はスペース，タブ，改行を表し，BEGINCOMMENT, ENDCOMMENT はそれぞれコメントの開始と終了を表す．それ以外の単語は WORD に含まれる．

作成するプログラムの状態遷移図を図 D.2 に示す．図中の CodeWord, CommentWord は単語数を格納する変数を意味し，各状態で WORD が入力されるたびに値が 1 増やされる．また，BEGINCOMMENT または ENDCOMMENT が入力されると状態が変化する．

パターン

LEX では，ルール部において状態遷移を次のように記述する．

```
<モード名> パターン アクション
```

パターンはトークンの形式を表し，**アクション**は C による文である．モード名で指定された状態であるときに，パターンに適合したトークンが入力されれば，そのアクションが実行される．

パターンは正規表現で記述する．以下では，正規表現の基本的な書き方について説明する．

最も簡単なパターンは特定の文字列の記述である．例えば，コメントの開始記号や終了記号は，それぞれ “(*” と “*)” であり[1]，これらを直接記述する．なお，正規表現では特別な意味を持つ記号があるため，実際に記述する際には "(*" とダブルクウォートで囲う

[1] コメントの開始記号と終了記号に “{”,“}” を使用する場合もあるが，ここでは考えないことにする．

ようにする．あるいは，直前にバックスラッシュ “\” をつけて，\(* と記述する．また，タブ，改行，バックスラッシュはそれぞれ次のように記述する．

```
\t, \n, \\
```

また，“\133” のようにバックスラッシュの後に 8 進数の数字を記述すると，ASCII コードでその数字に該当する文字として扱われる．

アルファベットの中の一文字などのように，ある特定の文字集合の中の一文字を指定することができる．例えば，WHITESPACE は次のように記述される．

```
[ \t\n]
```

また，アルファベットや一桁の数字の場合は範囲を指定することができる．例えば，小文字のアルファベットの集合の中の一文字を次のように記述する．

```
[a-z]
```

複数の範囲や文字を組み合わせて指定することも可能である．例えば，アルファベットと数字の集合の中の一文字は次のように記述する．

```
[0-9A-Za-z]
```

ある特定の文字以外の文字を指定するには，“[” の直後に補集合を表す “^” を付加する．例えば，アルファベット以外の一文字を表すには次のように記述する．

```
[^A-Za-z]
```

なお，任意の一文字を表す場合にはこのような表現を用いず，ピリオド “.” を記述する．ただしピリオドは改行文字にはマッチしない．

パターンは複数のパターンを連結して記述することができる．例えば，“[a-zA-Z][a-zA-Z]” と記述すると，2 文字のアルファベットからなる文字列を意味する．しかし，これではパターンの任意回の繰り返しなどを表現できない．正規表現には次の 3 つの演算子がある．

- ? 0 回または 1 回だけ出現する．
- * 0 回以上出現する．
- + 1 回以上出現する．

例えば，Pascal における識別子は，先頭がアルファベットで，その後に 0 個以上のアルファベットまたは数字がつながったトークンである．これを正規表現で記述すると次のようになる．

```
[A-Za-z][0-9A-Za-z]*
```

複数のパターンを選択したい場合には，演算子 “|” を用いる．例えば，Pascal の演算子のうち 2 つの記号から構成されるものは次のように記述する．

```
"<>"|":="|"<="|">="|".."
```

正規表現で用いられる演算子には優先順位が存在する．よって，複数のパターンを結合する際には優先順位を考慮する必要がある．ここで述べた演算子の優先順位は次の表の通りである．表の上の方が優先度が高い．

演算子	優先度
"a" \b \133 [c] .(ピリオド)	高い
(a)	
a? b* c+	
ab	
a|b	低い

優先順位を変更するには括弧を用いる．例えば，“ab*”と“(ab)*”では優先度の違いから，前者は“abbbbbb...”を，後者は“abababab...”を表す．

LEX では入力が複数の正規表現にマッチする場合は，もっとも長いパターンにマッチする．同じ長さの場合は最初に現れたルールにマッチする．定義部に書かれたパターン定義の順番ではなく，ルール部の記述の順に依存する．

パターン定義

パターンをルール部の中に直接記述することができるが，パターンに名前をつけてその名前を参照するようにすることができる．名前をつけることで，記述を見やすくし，また意味がわかりやすくなる．パターンの名前は定義部で次のように定義する．

```
パターン名　パターン
```

例えば，WHITESPACE は次のように記述される．

```
WHITESPACE  [ \t\n]
```

定義したパターン名は“{”, “}”で囲うことで参照することができる．例えば，Pascal の演算子を表すパターン OPERATOR を次のように記述する．

```
OPERATOR     ({COMPOUNDOP}|{SINGLEOP})
COMPOUNDOP   ("<>"|[<>:]=|"..")
SINGLEOP     ([\+\-\*/\.,:;=<>^\(\)\[\]])
```

ここでは，COMPOUNDOP, SINGLEOP が参照されている．なお，各パターンが括弧で囲まれているが，これは参照された際に演算子の優先順位がパターンに影響することを防ぐためである．影響がない場合でも囲うようにすることで，未然に間違いを防ぐことができる．

アクション

アクションにはCの文を記述する．基本的に，記述できる文は一つであり，複数記述する場合には “{”, “}” で囲う．また，アクションが必要ない場合は空文であるセミコロン “;” だけを書く．例えば，Pascalのプログラムには現れない文字が入力される場合にそれを無視するようにするには，次のように記述する．

```
<CODE>.  ;
```

ただし，このルールは他のルールより後に記述する．後に記述することで，他のパターンに適合しなかった場合だけこのパターンに適合するようになる．

アクションの中ではモードを変更することができる．変更するにはマクロ `BEGIN()` を用いる．引数には遷移したいモードを書く．例えば，`CODE` から `COMMENT` に遷移したい場合は次のように記述する．

```
BEGIN(COMMENT);
```

サブルーチン部

サブルーチン部では，プログラムに必要な関数を記述する．この記述の中で，入力ファイルの設定や初期状態の設定を行なう．入力ファイルを設定するには，入力ファイルをオープンし，変数 `yyin` にそのファイルポインタを代入する．`yyin` を設定しない場合は，標準入力が入力ファイルになる．また，初期状態の設定はアクションと同様に `BEGIN()` で指定する．字句解析を開始するには関数 `yylex()` を呼び出す．

D.1.2 記述例

例題の記述例を図D.3に示す．この記述例では，二つの関数 `ppwc()`, `main()` が定義されている．関数 `ppwc()` は引数で指定されたファイルを入力ファイルに設定して字句解析を行ない，結果を引数で指定された変数に代入する．関数 `main()` はコマンドの引数を調べ，引数で指定されたファイルをオープンして `ppwc()` を呼び出す．引数にファイルが指定されない場合は標準入力を入力ファイルに設定する．

この二つの関数は一つの関数で記述することが可能だが，関数 `ppwc()` にLEXの記述に依存した部分をまとめてあるため，関数 `main()` を別のファイルに記述して分割コンパイルしたり，他のファイルで定義された関数から関数 `ppwc()` を呼び出すことが可能である．

D.1.3 使用方法

LEXプログラムに対してコマンド `lex` を実行するとCのプログラムが生成される．“`-t`” オプションを指定することで標準出力に出力することができ，リダイレクトを利用することで任意の名前のファイルに生成されたCプログラムを保存できる．生成されたプログラ

```
 1 %{
 2 /*
 3  * ppwc: Pascal Program Word Counter,
 4  *           prduced by Appon Software Labo.
 5  */
 6
 7 #include <stdio.h>
 8
 9 int     CodeWord = 0;           /* for word count in CODE */
10 int     CommWord = 0;           /* for word count in COMMENT */
11
12 %}
13
14 WORD            ({IDENTIFIER}|{NUMBER}|{STRING}|{OPERATOR})
15 IDENTIFIER      ([a-zA-Z][0-9a-zA-Z]*)
16 NUMBER          ({INTEGER}(\.{INTEGER})?([Ee][\+\-]?{INTEGER})?)
17 INTEGER         ([0-9]+)
18 STRING          ('([^']|'')*')
19 OPERATOR        ({COMPOUNDOP}|{SINGLEOP})
20 COMPOUNDOP      ("<>"|[<>:]=|"..")
21 SINGLEOP        [\+\-\*/\.,:;=<>^\(\)\[\]]
22 BEGINCOMMENT    ("(*")
23 ENDCOMMENT      ("*)")
24 WHITESPACE      ([ \t\n]+)
25
26 %Start  CODE COMMENT
27
28 %%
29
30 <CODE>{WORD} {                 /* Count words. */
31     CodeWord++;
32 }
33
34 <CODE>{BEGINCOMMENT} {          /* Enter to COMMENT mode. */
35     BEGIN(COMMENT);
36 }
37
38 <CODE>{WHITESPACE}      ;       /* Skip white spaces. */
39 <CODE>.                 ;       /* Skip unexpected symbols. */
40
41
42 <COMMENT>{WORD} {               /* Count words. */
43     CommWord++;
44 }
45
46 <COMMENT>{ENDCOMMENT} {         /* Enter to CODE mode. */
47     BEGIN(CODE);
48 }
49
50 <COMMENT>{WHITESPACE}   ;       /* Skip white spaces. */
51 <COMMENT>.              ;       /* Skip unexpected symbols. */
52
53 %%
54
55 void    ppwc(FILE *file, int *code_words, int *comm_words)
56 {
57     yyin = file;                /* Set input file */
58     BEGIN(CODE);                /* Initial mode. */
59     yylex();                    /* Call lexical analyzer. */
60
61     *code_words = CodeWord;     /* Return number of words in code. */
62     *comm_words = CommWord;     /* Return number of words in comment. */
63 }
64
65 void    main(int argc, char **argv)
66 {
67     FILE        *input_file = stdin;    /* Input file */
68     int         code_words;             /* Number of words in code */
69     int         comm_words;             /* Number of words in comments */
70
71     if (argc > 1 && (input_file = fopen(*++argv, "r")) == NULL) {
72         fprintf(stderr, "Can't open %s.\n", *argv);
73         exit(1);
74     }
75
76     ppwc(input_file, &code_words, &comm_words);
77
78     printf("words in code    = %d\n", code_words);
79     printf("words in comment = %d\n", comm_words);
80 }
```

図 D.3: プログラム記述例

ムは通常の C のプログラムと同じようにコンパイルする．ただし，LEX 用のライブラリをリンクするためにコンパイラの引数に “-ll” オプションをつける必要がある．

ppwc.l というファイル名の LEX プログラムから，実行形式ファイル ppwc を作成する例を以下に示す．

```
% lex -t ppwc.l > ppwc.c
% cc -o ppwc ppwc.c -ll
```

なお，LEX の記述が間違っていても lex の実行時にエラーにならず，生成された C プログラムをコンパイルしたときにエラーになることがあるので注意すること．

D.1.4 複数の LEX 記述

LEX で生成される C プログラムには yytext，yylex() のように yy というプレフィックスの付いた変数名や関数名が使用されている．一つのプログラムに複数の LEX 記述から生成されたプログラムが含まれる場合はこれらの関数が二重定義にならないようにプレフィックスを変更する必要がある．このプレフィックスは -P オプションで指定することができる．

```
% lex -Pyy2 -t sample.l > sample.c
```

この例で生成されるプログラムでは変数名，関数名に yy2 というプレフィックスが付けられる．

デフォルトのプレフィックスで生成されたプログラムの中で呼び出されている関数 yywrap() は LEX で生成されたプログラムではなく，LEX のライブラリ（libl）で定義されている．そのため，上の例の場合 yy2wrap() の定義を別途記述する必要がある．ライブラリでマクロとして定義されている関数 yywrap() は常に 1 を返す関数であり，これと同じ動作をする関数 yy2wrap() をサブルーチン部で定義してやればよい．

付 録E　サンプル・プログラム

E.1　scanner.l

```
%{
/*
 * scanner: scanner for PL-*
 *
 */

#include <stdio.h>
#include <string.h>
#include "symbols.h"

#define MAXLENGTH 16

typedef union {
    int num;
    char ident[MAXLENGTH+1];
} token;

/*
 * yylval という変数名にするのは，yacc との融合時にプログラムの変更を
 * 最小限にするためである.
 */
token yylval;

%}
%option yylineno                    /*行番号を保持する変数*/
%%

begin           return SBEGIN;
div             return DIV;
```

```
      ...

while           return WHILE;
write           return WRITE;

"+"             return PLUS;
"-"             return MINUS;

      ...

"."             return PERIOD;
":="            return ASSIGN;

[0-9]|[1-9][0-9]* {
    yylval.num = atoi(yytext);
    return NUMBER;
}

[a-zA-Z][0-9a-zA-Z]* {
    strcpy(yylval.ident, yytext);
    return IDENT;
}

[ \t\n]         ;

. {
    fprintf(stderr, "cannot handle such characters: %s\n", yytext);
}

%%

main(int argc, char *argv[]) {
    FILE *fp;
    int tok;

    if (argc != 2) {
        fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(1);
    }
```

```
    if ((fp = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "cannot open file: %s\n", argv[1]);
        exit(1);
    }

    /*
     * yyin は lex の内部変数であり，入力のファイルポインタを表す.
     */
    yyin = fp;

    /*
     * yylex() を呼び出すことにより，トークンが一つ切り出される.
     * yylex() の戻り値は，上のアクション部で定義した戻り値である.
     * yytext には，切り出されたトークンが文字列として格納されている.
     */
    while (tok = yylex()) {
        switch (tok) {
        case NUMBER:
            printf("\"%s\":\t%d\t%d\n", yytext, tok, yylval.num);
            break;

        case IDENT:
            printf("\"%s\":\t%d\t%s\n", yytext, tok, yylval.ident);
            break;

        default:
            printf("\"%s\":\t%d\tRESERVE\n", yytext, tok);
            break;
        }
    }
}
```

E.2 symbols.h

```
/*
 * symbols.h
 *
```

```
 * 注意: このファイルは課題 1 でのみ使用する
 */

enum {
    SBEGIN = 1,                  /* begin */
    DIV,
    DO,
    ELSE,
    SEND,                       /* end */
    FOR,
    FORWARD,
    FUNCTION,
    IF,
    PROCEDURE,
    PROGRAM,
    READ,
    THEN,
    TO,
    VAR,
    WHILE,
    WRITE,

    PLUS,
    MINUS,
    MULT,
    EQ,                         /* = */
    NEQ,                        /* <> */
    LE,                         /* <= */
    LT,                         /* < */
    GE,                         /* >= */
    GT,                         /* > */
    LPAREN,                     /* ( */
    RPAREN,                     /* ) */
    LBRACKET,                   /* [ */
    RBRACKET,                   /* ] */
    COMMA,
    SEMICOLON,
    COLON,
    INTERVAL,                   /* .. */
```

```
    PERIOD,
    ASSIGN,                     /* := */
    NUMBER,
    IDENT,
};
```

E.3 parser.y

```
%{
/*
 * parser; Parser for PL-*
 */

#define MAXLENGTH 16

#include <stdio.h>

%}

%union {
    int num;
    char ident[MAXLENGTH+1];
}

%token SBEGIN DO ELSE SEND
%token FOR FORWARD FUNCTION IF PROCEDURE
%token PROGRAM READ THEN TO VAR
%token WHILE WRITE

%left PLUS MINUS                        ← 注意
%left MULT DIV                          ← 注意

%token EQ NEQ LE LT GE GT
%token LPAREN RPAREN LBRACKET RBRACKET
%token COMMA SEMICOLON COLON INTERVAL
%token PERIOD ASSIGN
%token <num> NUMBER                     ← yylval の型を指定
%token <ident> IDENT                    ← yylval の型を指定
```

```
%%

program
        : PROGRAM IDENT SEMICOLON outblock PERIOD
        ;

outblock
        : var_decl_part subprog_decl_part statement
        ;

var_decl_part
        : /* empty */
        | var_decl_list SEMICOLON
        ;

         ...

%%
yyerror(char *s)
{
  fprintf(stderr, "%s\n", s);
}
```

付 録F　プログラムのコンパイル方法

ここでは，C 言語で作成したコンパイラのコードをコンパイルする方法について説明する．使用するコマンドは以下の通りである．

- cc：C コンパイラ．C 言語のコードをコンパイルする．
- lex：lex コマンド．lex のファイル（.l ファイル）をコンパイルし，C 言語のコード（lex.yy.c）を生成する．
- yacc： yacc コマンド．yacc のファイル（.y ファイル）をコンパイルし，C 言語のコード（y.tab.c）を生成する．

F.1　lex を使う場合

```
lex scanner.l
cc lex.yy.c -ll -o parser
```

lex scanner.l により，lex.yy.c が生成される．続いて cc コマンドにより，実行可能なファイルを生成する．このとき，-ll オプションによって，実行に必要な lex のライブラリ（liblex）をリンクする．ここでは-o オプションにより，生成される実行ファイル名を指定している．

F.2　yacc と lex を使う場合

```
yacc -d parser.y
lex scanner.l
cc y.tab.c lex.yy.c -ll -o parser
```

yacc -d paraser.y により，y.tab.c が生成される．また，-d オプションにより，トークンなどが定義された y.tab.h ファイルも生成される．最後に cc コマンドにより，実行可能なファイルを生成する．

F.3 Makefile を用いたコンパイル

F.3.1 Makefile とは

上記の様に，コンパイル手順が複雑になると，どのファイルを更新したのか，どのコマンドから実行しなければならないのかが分からなくなり，ミスが増える．そこで，コンパイル手順を自動化する．

make とはプログラムのコンパイル時の作業を自動化するツールである．ファイルの更新時間をチェックし，更新されているもののみを再コンパイルする．自動的なコンパイルを実現するために，どのファイルをどうやってコンパイルするのか，ターゲットファイルを生成するためのコンパイル情報を Makefile に記述しておく．

一旦 Makefile を作成しておけば，

```
% make
```

というコマンドを実行するだけで，自動的にコンパイルが行われる．

F.3.2 Makefile の書き方

詳細は，Web 上の記事等を参考にするとよいが，ここでは簡単に説明する．

Makefile はマクロ定義部とルール記述部からなり，ルール記述部では，ファイル間の依存関係とターゲットファイルの生成手順を記述する．ルールの書き方は，以下の形式である．

ターゲットファイル：依存ファイル群
＜タブ＞生成規則

これにより，依存ファイル群のうちのどれかが更新された場合，生成規則が実行され，ターゲットファイルが更新される．

例えば，test.c をコンパイルして test という実行プログラムを生成する場合の例は以下の通りである．

```
test: test.c
    cc -o test test.c
```

F.4 作成したコンパイラの実行方法

上記コンパイル処理により，parser という実行可能ファイルが生成される．これを用いることで，PL-0 などのソースプログラムをコンパイルすることができる．

```
./parser pl0a.p
```

これにより，課題 5 完了時点では，pl0a.ll が生成されるはずである．次節では，この pl0a.ll を実行する方法について説明する．

付 録G　LLVM IRの実行方法

例えば，pl0a.p をコンパイルして pl0a.ll を生成したとする．生成した LLVM IR のコードを実行するには，まず，pl0a.ll を，lli コマンドを用いて実行する．

```
lli pl0a.ll
```

サンプルコードの挙動を確認したい場合は，サンプルコードと同等の C 言語コードを作成し，clang コマンドでコンパイルして LLVM IR コードを生成し，その LLVM IR コードを実行して比較してもよい．例えば，pl0a.p と同等の pl0a.c を作成した後，下記のコマンドを実行すると，LLVM IR のコマンドが生成できる．

```
clang -S -O0 -emit-llvm pl0a.c
```

これにより，pl0a.ll が生成され，lli コマンドで実行できる．このとき，生成されるファイル名には十分注意すること．C 言語コードのファイル名が拡張子を除いて等しい場合，作成したコンパイラで生成したファイルが上書きされてしまう．これを避けるために clang の実行時に-o オプションを使用するなど工夫すること．

なお，その LLVM IR コードと作成したコンパイラで生成したコードを比較することで適切なコード生成ができたか視認することも一つの方法である．

通常，main 関数は終了ステータス（exit status）を返す．C 言語の場合，EXIT_SUCCESS（値は 0）で正常終了，EXIT_FAILURE（値は 1）でエラーを示す．Linux では，ひとつ前に実行したコマンドの終了ステータスが何であったかを，$?を使って得ることができる．例えば，以下のようにコマンドを実行することで終了ステータス（すなわち，sample.ll 内の main 関数が返した値）が表示される．

```
> lli sample.ll

> echo $?
```

これを利用し，計算結果を確認することができる．一方で，この方法ではプログラムが行なった計算結果を，標準（エラー）出力に表示しない限り，確認することができない．そこで，main 関数中の，例えば，

```
  ret i32 0
}
```

となっている部分を

```
  %12 = load i32, i32* @sum, align 4
  ret i32 %12
}
```

のように変更することにより，計算結果が格納された@sum の値を返すことができる．これを echo $?コマンドにより出力することで，@sum の計算結果を確認することができる．