

# 言語処理 プログラミング

情報工学課程 3回生

担当教員: 水野 修 (@omzn)

(2024 年 12 月 15 日版)



<b>第 1 章 概要: 言語処理プログラミングによるこそ</b>	5
1.1 演習の目的 . . . . .	5
1.2 構成とスケジュール . . . . .	5
1.3 レポートについて . . . . .	6
1.4 プログラム提出について . . . . .	8
1.5 言語処理プログラミング Docker イメージ . . . . .	9
1.6 質問、連絡、資料等の配布について . . . . .	10
1.7 実際の演習について . . . . .	10
1.8 テストについて . . . . .	10
1.9 スケジューリングと進捗の把握 . . . . .	18
<b>第 2 章 課題 1 - 字句解析</b>	24
2.1 課題: 字句の出現頻度表の作成 . . . . .	24
2.2 課題説明 . . . . .	24
2.3 プログラム作成条件 . . . . .	28
2.4 入出力例 . . . . .	29
2.5 拡張仕様 . . . . .	30
2.6 スキヤナ用ヘッダファイル . . . . .	32
2.7 課題 1 メインプログラム例 (サンプル) . . . . .	34
2.8 おまけ . . . . .	35
2.9 スキヤナの作り方のヒント . . . . .	36
2.10 テストケースの意味 . . . . .	38
<b>第 3 章 課題 2 - 構文解析</b>	40
3.1 課題: プリティプリンタの作成 . . . . .	40
3.2 課題説明 . . . . .	40
3.3 LL(1) 構文解析系の作り方 (課題 2 相当) . . . . .	44
3.4 テストケースの意味 . . . . .	49
3.5 課題 3 以降への対応について . . . . .	49

<b>第 4 章 課題 3 - 意味解析</b>	50
4.1 課題: クロスリファレンサの作成 . . . . .	50
4.2 課題説明 . . . . .	50
4.3 クロスリファレンサの作り方 . . . . .	55
4.4 テストケースの意味 . . . . .	58
<b>第 5 章 課題 4 - コンパイラ</b>	59
5.1 課題: コンパイラの作成 . . . . .	59
5.2 課題説明 . . . . .	59
5.3 コード生成の方法 . . . . .	60
5.4 テストケースの意味 . . . . .	70
5.5 COMET II / CASL II の仕様 . . . . .	91
5.6 CASL II アセンブラー・COMET II エミュレータ . . . . .	103
<b>参考文献</b>	106



## 図目次

1.1	簡単な PERT 図 . . . . .	20
1.2	サブタスクレベルの PERT 図 . . . . .	20
1.3	Redmine のチケット一覧 . . . . .	22
1.4	Redmine のチケット . . . . .	23
2.1	MPPL のマイクロ構文 . . . . .	25
3.1	EBNF による MPPL の構文 . . . . .	42
3.2	プリティプリントの例 (1) . . . . .	44
3.3	プリティプリントの例 (2) . . . . .	45
4.1	MPPL の構文に対する制約規則 (1/2) . . . . .	51
4.2	MPPL の構文に対する制約規則 (2/2) . . . . .	52
4.3	課題 3 の実行例 (入力) . . . . .	54
4.4	課題 3 の実行例 (出力) . . . . .	55
4.5	sample11ppmpl に対する記号表のデータ構造の一部 . . . . .	57
5.1	MPPL のセマンティクス (1/3) . . . . .	61
5.2	MPPL のセマンティクス (2/3) . . . . .	62
5.3	MPPL のセマンティクス (3/3) . . . . .	63



## 表目次

1.1	事前作業計画の例（日付は過去の例）	21
1.2	簡単な事前作業計画の例（日付は過去の例）	22

# 1



## 概要: 言語処理プログラミングによるこそ

### 1.1

#### 演習の目的



本科目は、C言語を用いて、比較的簡単なプログラミング言語のコンパイラを作成することにより、コンパイラの基本的な構造とテキスト処理の手法を理解することを主目的とします。また、比較的大きなプログラムを作成する経験を得ることができます。

### 1.2

#### 構成とスケジュール



本科目は、次の4ステップからなります。後の課題はそれ以前の課題で作成したプログラムを再利用もしくは修正することで達成できるように設定されています。なお、以下のプログラム作成課題は、ANSI-Cの範囲内のC言語及びその標準ライブラリのみを用いて作成してください。

C言語の教科書はいつでも必携ですよね。

#### 1.2.1

##### 課題1: 字句の出現頻度表の作成



課題内容関連講義予定日: 2024-09-30(月)

演習期間: 2024-09-30(月) ~ 2024-10-20(日)

レポート・プログラム提出期間: 2024-10-21(月) ~ 2024-10-28(月)

課題の概略: プログラミング言語 MPPL(別紙参照)で書かれたプログラムらしきものを読み込み、字句(トークン)がそれぞれ何個出現したかを数え、出力するプログラムを作成する。

キー技術: テキスト処理、字句解析



らしきもの…

#### 1.2.2

##### 課題2: プリティプリンタの作成



課題内容関連講義予定日: 2024-10-21(月)

演習期間: 2024-10-21(月) ~ 2024-11-24(日)

プリティプリンタって、かわいくない?

**レポート・プログラム提出期間:** 2024-11-25(月) ~ 2024-12-02(月)

**課題の概略:** プログラミング言語 MPPL で書かれたプログラムを読み込み、構文エラーがなければ、入力されたプログラムをプリティプリントした結果を出力し、構文エラーがあれば、そのエラーの情報(エラーの箇所、内容等)を少なくとも一つ出力するプログラムを作成する。

**キー技術:** 再帰呼び出し、構文解析

### 1.2.3

#### 課題 3: クロスリファレンサの作成

**課題内容関連講義予定日:** 2024-11-25(月)

**演習期間:** 2024-11-25(月) ~ 2024-12-15(日)

**レポート・プログラム提出期間:** 2024-12-16(月) ~ 2024-12-23(月)

**課題の概略:** プログラミング言語 MPPL で書かれたプログラムを読み込み、コンパイルエラー、すなわち、構文エラーもしくは制約エラー(型の不一致や未定義な変数の出現等)がなければ、クロスリファレンス表を出力し、エラーがあれば、そのエラーの情報(エラーの箇所、内容等)を少なくとも一つ出力するプログラムを作成する。

**キー技術:** データ構造、ポインタ、記号表

### 1.2.4

#### 課題 4: コンパイラの作成

**課題内容関連講義予定日:** 2024-12-16(月)

**演習期間:** 2024-12-16(月) ~ 2025-01-19(日)

**レポート・プログラム提出期間:** 2025-01-20(月) ~ 2025-01-27(月)

**課題の概略:** プログラミング言語 MPPL で書かれたプログラムを読み込み、コンパイルエラー、すなわち、構文エラーもしくは制約エラー(型の不一致や未定義な変数の出現等)があれば、そのエラーの情報(エラーの箇所、内容等)を少なくとも一つ出力し、エラーがなければ、オブジェクトプログラムとして、CASLII[2](別紙参照)のプログラムを出力するプログラム(すなわちコンパイラ)を作成する。

**キー技術:** コンピューターアーキテクチャ(命令セットレベルアーキテクチャ), コード生成

### 1.3

#### レポートについて



各課題について、レポートを提出すること。1つの課題でもレポートが提出されない場合には、言語処理プログラミング全体が成績評価対象外になるので注意してください。

### 1.3.1

#### レポートの構成

レポートには、次の事項を含めてください。

1. 作成したプログラムの設計情報

- (a) 全体構成：どのようなモジュールがあるか，それらの依存関係(呼び出し関係やデータ参照関係)
- (b) 各モジュールごとの構成：使用されているデータ構造の説明と各変数の意味
- (c) 各関数の外部(入出力)仕様：その関数の機能，引数や戻り値の意味と参照する大域変数，変更する大域変数などを含む

(注) ただし，それ以前のレポートに記載した内容と同じである場合には，その旨のみを記述するだけでよい。たとえば，関数○○が課題1で説明済みであり何も変更されていなければ，

関数○○：課題1 レポートで記載済み  
と書くだけでよい。

なお，モジュールとは，意味的にまとまったプログラムの部分である。小さなプログラムの場合は関数一つ一つをモジュールと考える場合もあるし，いくつかの大域変数を共有して使う関数群を(その大域変数も含めて)モジュールとする場合もある。たとえば，課題1で作成する字句解析系はモジュールである。

## 2. テスト情報

- (a) テストデータ(既に用意されているテストデータについてはファイル名のみでよい)
- (b) テスト結果(テストしたすべてのテストデータについて)
- (c) テストデータの十分性(それだけのテストでどの程度バグがないことが保証できるか)の説明

## 3. この課題を行うための事前計画(スケジュール)と実際の進捗状況(1.9.3節にて述べるRedmineを利用する場合は、この章は省略できる。)

### (a) 事前計画(スケジュール)

当初の計画と、演習中に計画を大きく修正した場合には修正後の計画(最終分だけでよい)を記載すること。計画を立て、〆切までに完成したプログラムとレポートを提出するまでが演習である。

### (b) 事前計画の立て方についての前課題からの改善点(課題1を除く)

### (c) 実際の進捗状況

### (d) 当初の事前計画と実際の進捗との差の原因

事前計画と進捗状況は、開始(予定)日，終了(予定)日，使用(見積もり)時間，作業(予定)内容の4項目をカラムとし行は日付順とする表形式で記述すること(実務では線表で書かれる)。作業(予定)内容は1行程度でよい。詳細な説明は課題1の付録参照。

### 1.3.2

#### レポートの形式

PDF もしくは、Word の形式のファイル。フォーマットは、A4 の用紙サイズで、表紙(1 ページ目)に課題名(「言語処理プログラミング 課題 1」など)、提出日の日付、学籍番号、氏名のみを記載してください。

**提出先** Redmine に作成してある個人用のプロジェクトの「提出」チケット

**提出期間** 各課題提出期間

### 1.4

#### プログラム提出について



各課題で作成したプログラム(ソースプログラム)もレポートと同様の提出期間中に Redmine を経由して提出してください。一つの課題でもプログラムが提出されない場合には、言語処理プログラミング全体が成績評価対象外になります。

提出されるべきものは、ソースプログラムファイルのみ (\*.c, \*.h のファイル) です。それらを同一のフォルダ(ディレクトリ)に置いて、まとめてコンパイルすれば、実行形式ファイルが生成されるものを提出してください。

本演習は、個人演習です。各自がプログラムとレポートを提出します。共同でのプログラム作成や他人のプログラムやレポートをコピーすることは厳禁です。他人のプログラムを見てはいけないですし、見せてもいけません。類似したプログラムが発見された場合には、双方に再提出を求める(再提出する余裕時間がある場合)か、ゼロ評価(再提出する余裕時間がない場合)となります。

なお、みなさんの作成環境とこちらのテスト環境が異なっている場合に(OS、文字コード、コンパイラなどの違いにより)問題(こちらでコンパイルエラーが出るなど)を生ずることがあります(過去の例)。コンパイルエラーなどでこちら側のテストを通らない場合、最低評価となります。特に、コメント内を含めて提出プログラムには、日本語等の複数バイト文字(いわゆる全角文字など)を使わず、1バイト文字(いわゆる半角英数字)のみを用いることをお勧めします。

提出されるプログラムは、どのような環境でもコンパイル・実行できることを期待していますが、みなさんがそれを確認することはかなり困難です。そのため、標準環境として、情報工学課程の演習室(8-205)の Linux 環境にある gcc を指定します(Linux 上の Eclipse 環境に含まれるコンパイラではないことに注意しましょう)。ここで、コンパイル、実行ができるかを確かめてください。すなわち、作成したプログラムファイルが、foo1.c, foo2.c, foo3.c であるとき、

```
gcc -o mpplc foo1.c foo2.c foo3.c
```

のように直接 gcc でコンパイルして、できた実行形式プログラムが思った通りに動くことを確かめてください。

従って、作成中は自分の使いやすい環境で、Eclipse 等の統合環境を用いるのが便利と思ますが、最終的な確認は Linux 上の shell で Makefile を用意して行うのが確実です。



コピー、ダメ、絶対。



macOS や Windows 上の環境で gcc を指定しても実体は clang であったりして、gcc でのコンパイル結果と異なることがあります。特に、Eclipse 上の C コンパイラと演習室環境の gcc では複数ファイルにまたがるグローバル変数の扱いが大きく異なるので、自身の環境で動くからといって、採点者の環境で動くと信じてはいけません。



Eclipse はグローバル変数の二重定義とか平気で無視しよるからね…

なお、docker が利用できる環境であれば、次節に示す Docker イメージを用いることで、演習室とほぼ同じ gcc の環境を再現できます。この Docker イメージも公式環境として取り扱いますので、動作確認に活用してください。

提出プログラムは以下の仕様に従ってコンパイルできるようにしてください。

1. 実行ファイル名は、課題 1 は「tc」、課題 2 は「pp」、課題 3 は「cr」、課題 4 は「mpplc」とします。
2. 以下の方法のいずれかで、上に指定する名前で実行ファイルが生成できるようにしてください。

```
gcc -o 実行ファイル名 *.c
```

または、

```
make
```

make を使う場合は、Makefile も必ず提出ファイルに含めてください。環境依存した外部コマンドなどを make から呼び出す場合は TA・教員環境で動作しないかもしれません。次節で説明する Docker で動作するのであれば問題ありません。

提出プログラムが複数のファイルになる場合には、zip など標準的な形式で 1 つのファイルにまとめて提出してください。zip 以外の形式でまとめられた場合など、こちらで取り出せない場合には評価することができません。

## 1.5

### 言語処理プログラミング Docker イメージ



前節で述べた環境依存問題を解決するために、gcc のバージョンを固定して手軽に利用できる Docker イメージを用意しています。

Docker のインストールは利用環境に依存します。詳細なインストール方法は Moodle に掲載しますので、そちらを確認してください。

Moodle に記載した方法で、Docker の設定が完了したら、

```
lppshell
```

VirtualBox におけるディスク容量が 20GB 程だと、Docker のインストール後にイメージをダウンロードするとディスクあふれが起きる可能性があります。事前に +10GB 程度のディスク容量の拡張を行うか、30GB 以上のディスクを使って新たに仮想マシンを構築してください。

で docker イメージを起動できます。

引き続いで出現するプロンプトでは、普通に gcc を利用できます。例えば次のようにすることで、プログラムをビルドできます。(自身のファイル構成により異なります。もちろん make コマンドなども利用できます。)

```
gcc *.c -o tc
```

Docker 環境から抜けるには、exit コマンドを実行します。

```
exit
```

この Docker 環境は演習実施中を通じて同じ gcc のバージョンを維持しますので、この環境で作成したプログラムであれば、教員、TA の環境と全く同じであることが保証されます。

## 1.6

### 質問、連絡、資料等の配布について



教員から受講生のみなさんへの連絡や資料配付は、各課題の説明会の他、Moodle を用いて行います。

質問は、Redmine にて受け付けます。URL は Moodle にて指示します。また、o-mizuno@kit.ac.jp へメールを送って質問しても良いです。その際は、メールの本文の最初に学生番号と氏名を明記してください。プログラムを見せながら質問したいなど対面での質問を希望する場合には、時間割上の演習時間(月曜日 2 時限)に 8-205 演習室で受け付けます(質問者が途切れるまで)。また、月曜日 2 限以外の時間に、直接、水野教授室(8 号館 3 階 8-320 室)まで質問しに来てもよいです、不在や会議等で質問を受け付けられない場合もあります。事前にアポイントメントをとることをお勧めします。

- メールの場合は名乗ってください。
- プログラムが動かない場合は、動かない状況を詳述してください。
- 動かないプログラムを添付して確認してほしい場合は、教員側がソースコードをコピーして新しいファイルにペーストするのを必要ファイル分繰り返した後、gcc のかけ方に悩んで時間を無駄にすることが無いように、上で説明した通りの手順でビルドできるようにしてください。



質問をするときは、質問を受ける側の気持ちになって有効な質問をしてほしいな。

## 1.7

### 実際の演習について



演習環境としては、月曜日 2 時限に、8-205 演習室が確保してありますが、密になるのを避けるために、同時間帯に CIS 演習室も利用可能です。実際には、最低限、エディタと C コンパイラ等があれば演習可能なので、個人持ちの PC を含めてどこで演習を行っても構いません。確認作業も docker を用いることで、どこでも可能です。週 1 コマの演習だけでは完成しないと思われる所以、十分な時間外演習が必要です。

## 1.8

### テストについて



ソフトウェア開発がモジュール化に基づいて行われていなるならば、テストは単体テストと統合テストに分けられます。

ソフトウェア工学でやったよね。



#### 1.8.1

##### 統合(結合)テスト

テスト(単体テスト)済みのモジュールをすべて接続して行うテスト。ここで問題が起ると、モジュール間の仕様の不整合を意味するので、大きな手戻りが起こる可能性があります。

## 1.8.2

### 単体テスト

モジュールごと、関数や手続きごとに行われるテスト。実際の関数やモジュールは他の関数を呼んでいたり呼ばれていたりするので、単体テストを行うためには、テストされるモジュールだけではなく、次のものも準備しなくてはなりません。

#### テスト用メインプログラム

テストされるモジュールを呼ぶメインプログラム。モジュールの仕様に従って作成される。ドライバ (driver) やテストドライバとも呼ばれます。

#### スタブ

テストされるモジュールが呼び出す関数やモジュールのうち、まだ完成していないものの代わり。通常はどのような引数で呼び出しても同じ値を返す関数であったり、呼び出されるとテスト (人) に返す値を聞くように作られます。埋め草 (stub) とも呼ばれます。

#### テストデータ

入力のパターンをすべて網羅するだけのテストデータセットを用意しなくてはなりません。これらはモジュールのプログラミングと並行して準備されます。他のモジュールができていないからと言って、テストできないということがあってはならないのです。

## 1.8.3

### ブラックボックステスト (black box test)

モジュールの外部仕様のみをみて、その仕様が満足されるかどうかを調べるためのテストデータを用意するテスト。標準的なテストデータの他に、境界値を用いたテストデータ、すなわち、仕様上許されるテストデータぎりぎりの値やそれを越える値を用いたテストデータを用意します。たとえば、データ数が 0 個の場合とか、上限値の場合、それを越える場合などです。境界値のあたりはバグが生じやすいところであり、場合によっては、初期値が境界値であるなどして必ず境界値での実行があることもあり、必ずテストせねばなりません。また、入力として許されないようなデータでもテストするのは、その場合に無限ループに陥ったりシステムダウンしたりしないことや、正しくエラー検出ができることを調べるためにあります。

ブラックボックステストは、システムの開発者でない人が、テストデータを用意して行なうことが望ましいです。なぜなら、システムの内部設計を知っている人が行なうと、その知識に影響され、外部仕様ではなく内部仕様に従ってテストを行いがちなためです。その場合には、システムの開発者が外部仕様を誤って理解していた場合に生ずるバグが検出されなくなります。

ブラックボックステストを系統的に実行するためのフレームワークは多数提供されています。本演習では、Python のテストモジュールである PyTest を利用したテスト実行環境を提供しています。

## PyTest

PyTest は Python のプログラムのユニットテストを支援するツールです。Python のモジュールですが、汎用的に利用できるので、今回は C プログラムの実行結果をテストする用途で利用します。

前節で紹介した Docker イメージにはあらかじめテスト環境も同梱しています。課題 1 の場合、以下のようなコマンドを実行することで、/workspaces/ディレクトリ内で生成された実行ファイルに対して、サンプルファイルに対するブラックボックステストを実行できます。

```
cd /lpp_test/01test && pytest -vv
```

実行結果が仕様の通りであれば、PASSED と表示されますが、そうでない場合は FAILED と表示され、何が仕様と異なるのかが表示されます。

課題 1 に対してテストを実行した結果を以下に示します。このテストは全件 PASSED なので、少なくともこのテストケースの範囲内ではプログラムは正しく動作していると考えられます。

```
# gcc *.c -o tc
# ls
Makefile id-list.c scan.c tc token-list.h token-list_ex.c
# cd /lpp_test/01test/
# pytest -vv
===== test session starts =====
platform linux -- Python 3.9.2, pytest-6.0.2, py-1.10.0, pluggy-0.13.0 -- /usr
    /bin/python3
cachedir: .pytest_cache
rootdir: /lpp_test/01test
plugins: timeout-1.4.1
collected 31 items

00_tc_compile_test.py::test_compile PASSED [ 3%]
00_tc_compile_test.py::test_no_param PASSED [ 6%]
00_tc_compile_test.py::test_not_valid_file PASSED [ 9%]
01_tc_run_test.py::test_run[./input01/sample011.mpl] PASSED [ 12%]
01_tc_run_test.py::test_run[./input01/sample012.mpl] PASSED [ 16%]
01_tc_run_test.py::test_run[./input01/sample012eof.mpl] PASSED [ 19%]
01_tc_run_test.py::test_run[./input01/sample012n.mpl] PASSED [ 22%]
01_tc_run_test.py::test_run[./input01/sample012neof.mpl] PASSED [ 25%]
01_tc_run_test.py::test_run[./input01/sample012s.mpl] PASSED [ 29%]
01_tc_run_test.py::test_run[./input01/sample012seof.mpl] PASSED [ 32%]
01_tc_run_test.py::test_run[./input01/sample013.mpl] PASSED [ 35%]
01_tc_run_test.py::test_run[./input01/sample014.mpl] PASSED [ 38%]
01_tc_run_test.py::test_run[./input01/sample014a.mpl] PASSED [ 41%]
01_tc_run_test.py::test_run[./input01/sample014b.mpl] PASSED [ 45%]
01_tc_run_test.py::test_run[./input01/sample11.mpl] PASSED [ 48%]
01_tc_run_test.py::test_run[./input01/sample11p.mpl] PASSED [ 51%]
01_tc_run_test.py::test_run[./input01/sample11pp.mpl] PASSED [ 54%]
01_tc_run_test.py::test_run[./input01/sample12.mpl] PASSED [ 58%]
01_tc_run_test.py::test_run[./input01/sample12cr.mpl] PASSED [ 61%]
```



なお、Docker に入らない（ホスト OS 上）でも利用できる lptest というコマンドを準備しているよ。こちらも Moodle に解説があるので、参照してくださいね。



果たして、プログラムの動作の正当性を保証できるテストケースを作成することはできるのだろうか？

```

01_tc_run_test.py::test_run[./input01/sample12crlf.mpl] PASSED [ 64%]
01_tc_run_test.py::test_run[./input01/sample12lf.mpl] PASSED [ 67%]
01_tc_run_test.py::test_run[./input01/sample12lfcf.mpl] PASSED [ 70%]
01_tc_run_test.py::test_run[./input01/sample13.mpl] PASSED [ 74%]
01_tc_run_test.py::test_run[./input01/sample14.mpl] PASSED [ 77%]
01_tc_run_test.py::test_run[./input01/sample14p.mpl] PASSED [ 80%]
01_tc_run_test.py::test_run[./input01/sample15.mpl] PASSED [ 83%]
01_tc_run_test.py::test_run[./input01/sample15a.mpl] PASSED [ 87%]
01_tc_run_test.py::test_run[./input01/sample16.mpl] PASSED [ 90%]
01_tc_run_test.py::test_run[./input01/sample17.mpl] PASSED [ 93%]
01_tc_run_test.py::test_run[./input01/sample18.mpl] PASSED [ 96%]
01_tc_run_test.py::test_run[./input01/sample19p.mpl] PASSED [100%]
=====
===== 31 passed in 0.27s =====

```

次に、課題 2 に対してテストを実行した例を示します。この例では、`sample24.mpl` と `sample25.mpl` が FAILED になっていて、テストが失敗したことを表します。それぞれのテストケースについての詳細が続いている、E で始まる行に検出したエラーが書いてあります。この場合、本来は `writeln ('It''s OK?')` と表示されるべきが、`writeln ('It's OK?')` となっているのが誤りだとされています。

```

# pytest -vv
===== test session starts =====
platform linux -- Python 3.9.2, pytest-6.0.2, py-1.10.0, pluggy-0.13.0 -- /usr
    /bin/python3
cachedir: .pytest_cache
rootdir: /lpp_test/02test
plugins: timeout-1.4.1
collected 33 items

pp_test.py::test_mppl_run[./input01/sample011.mpl] PASSED [ 3%]
(中略)
pp_test.py::test_mppl_run[./input02/sample22.mpl] PASSED [ 69%]
pp_test.py::test_mppl_run[./input02/sample23.mpl] PASSED [ 72%]
pp_test.py::test_mppl_run[./input02/sample24.mpl] FAILED [ 75%]
pp_test.py::test_mppl_run[./input02/sample25.mpl] FAILED [ 78%]
pp_test.py::test_mppl_run[./input02/sample25t.mpl] PASSED [ 81%]
(中略)
pp_test.py::test_mppl_run[./input02/sample2a.mpl] PASSED [100%]

===== FAILURES =====
----- test_run[./input02/sample24.mpl] -----
mpl_file = '../input02/sample24.mpl'

@ pytest.mark.timeout(10)
@ pytest.mark.parametrize(("mpl_file"), test_data)

def test_run(mpl_file):
    """準備したテストケースを全て実行する。"""
    if not Path(TEST_RESULT_DIR).exists():
        os.mkdir(TEST_RESULT_DIR)

```



ちなみにこのエラーは課題 1 の範囲での実装ミスだよ。ただ、課題 1 のテストではこのエラーを発見するのなかなか難しいよ。

```

        out_file = Path(TEST_RESULT_DIR).joinpath(Path(mpl_file).stem + ".out"
    )
    res = common_task(mpl_file, out_file)
    # 正常終了した場合
    if res == 0:
        expect_file = Path(TEST_EXPECT_DIR).joinpath(Path(mpl_file).stem +
".stdout")
        with open(out_file,encoding='utf-8') as ofp, open(expect_file,
encoding='utf-8') as efp:
            out_cont = ofp.readlines()
            est_cont = efp.readlines()
            for i, out_line in enumerate(out_cont):
>                 assert out_line == est_cont[i], "Line does not match."
E                     AssertionError: Line does not match.
E                     assert "    writeln ( 'It's OK?' )" == "    writeln ( 'It
's OK?' )"
E                         -    writeln ( 'It''s OK?' )
E                         ?
E                         -
E                         +    writeln ( 'It's OK?' )

01_pp_run_test.py:88: AssertionError
(後略)

```

pytest に -x オプションを付けると、エラーが出た時点で以後のテストを中止します。

```
pytest -vv -x
```

-k オプションを与えると選択したファイルだけについてテストを実行できます。

```
pytest -vv -k sample24.mpl
```

あるテストケースについての修正部分が別の場所に影響を及ぼして、これまで通っていたテストが通らなくなることはよくありますので、都度全件テストを実行することは品質保証の観点から有益です。これを回帰テスト (regression test) と呼びます。

## 1.8.4 ホワイトボックステスト (white box test)

モジュールの実装に基づいてテストデータを用意するテストです。実装に基づいてテストがどの程度十分かを評価できます。標準的な基準に次の 3 つがあります。

**命令網羅 (C0 カバレッジ)** モジュール中のすべての文を実行するようにテストデータを用意します。もちろん、1 組のテストデータでは無理なので、複数組のテストデータを用意します。

**分岐網羅 (C1 カバレッジ)** モジュール中のあらゆる分岐の真偽を実行するようにテストデータを用意します。たとえば、テストしたい関数が 2 つの if 文が並んでいるものであれば、4 通りのテストパスがあります。

**条件網羅 (C2 カバレッジ)** モジュール中のあらゆる分岐の内部の条件の真偽を実行するようにテストデータを用意します。たとえば、テストしたい関数が 2 つの if 文が

並んでいるものでそれぞれの if の条件が 2 つの論理式を積論理にしたものであれば、8 通りのテストパスがあります。

分岐網羅 (C1 カバレッジ) の方が命令網羅 (C0 カバレッジ) よりも強力なテストであるのは自明ですが、モジュールサイズが大きくなつた場合、コストや時間の制限から C0 カバレッジがやっと、という場合も多くなります。そのような場合には、C0 カバレッジが 100%, C1 カバレッジが 30%，というような表現になります。なお、通常実行されない部分、例えば、バグを早期に検出するための if 文による分岐などはカバレッジ率の分母に入れなくてもよいでしょう。また、while 文などでは本体が 1 回も実行されない場合と複数回実行される場合をテストすれば、C0 でも C1 でもカバーされたとしてよいでしょう。

### gcov

ホワイトボックステストを実行するツールとして、**gcov** が知られています。

gcov は gcc のカバレッジ検出ツールであり、プログラムを gcc でコンパイルする時に以下をオプションで付けることにより、カバレッジ検出を有効にできます。

```
gcc -coverage -c *.c  
gcc -coverage -o tc *.o
```

カバレッジ検出を有効化した自身のプログラム (ここでは **tc**) に対して様々な入力を与えて実行すると、C0・C2 カバレッジの計算に必要な情報が蓄積されていきます。

```
./tc sample11.mpl  
...  
./tc sample19p.mpl
```

この後、カレントディレクトリを見ると、いくつかのファイル (\*.gcda, \*.gcno) が生成されているのが分かります。これがカバレッジ情報になります。複数の C ファイルがあるときは、それぞれに対してカバレッジ情報は生成されています。

```
# ls  
  
scan.c.gcov      tc-scan.gcda        token-list.h  
tc                tc-scan.gcno        token-list_ex.c  
tc-token-list_ex.gcda  token-list_ex.c.gcov  
scan.c          tc-token-list_ex.gcno
```

ここで生成された **gcda** ファイルに対して、**gcov** を実行します。

```
# gcov -b *.gcda
```

```
File 'id-list.c'  
Lines executed:0.00% of 34  
Branches executed:0.00% of 16  
Taken at least once:0.00% of 16  
Calls executed:0.00% of 5  
Creating 'id-list.c.gcov'  
  
File 'scan.c'
```

```
Lines executed:85.21% of 169
Branches executed:97.06% of 136
Taken at least once:84.56% of 136
Calls executed:82.00% of 50
Creating 'scan.c.gcov'

File 'token-list_ex.c'
Lines executed:76.47% of 17
Branches executed:100.00% of 10
Taken at least once:80.00% of 10
Calls executed:75.00% of 8
Creating 'token-list_ex.c.gcov'

Lines executed:71.36% of 220
```

ファイルごとに、以下の情報が示されています。

- Lines executed: 命令網羅 (C0 カバレッジ)
- Branches executed: 分岐命令を通った数
- Taken at least once: 分岐内部の条件の真偽を少なくとも 1 回ずつ通った回数。条件網羅 (C2 カバレッジ)
- Calls executed: 別関数を呼び出した回数

また、最終行には全体の命令網羅が示されています。

この結果からは以下のこと読み取れます。

- id-list.c は実行されていない。
- 全ファイル合計の命令網羅 (lines executed) は 71.36%
- scan.c に関しては、命令網羅は 85.21%，条件網羅は 84.56%。
- token-list\_ex.c に関しては、命令網羅は 76.47%，条件網羅は 80%。

また、gcov を実行した結果生成される\*.gcov ファイルには、ソースプログラムのどの行が実行されたのかを示す詳細な情報が格納されます。テキストエディタで開いて読むことができます。

カバレッジ情報はプログラムの実行ごとに積み重なって行くので、一連のテストが終了したら削除するか待避させる必要があります。\*.gcov, \*.gcda, \*.gcno のファイルをすべて削除すればよいでしょう。

また、カバレッジ情報の収集はプログラムの実行に大きな負荷をかけるので、不要な時まで--coverage オプションを付けてコンパイルすることは控えましょう。

本課題でホワイトボックステストを行うときは、命令網羅 (C0 カバレッジ) を高めることを考えればとりあえず十分です。



gcov の Branches executed の数は、いまいち何を数えているのか分からなくて混乱するから無視して良いよ。

## 1.8.5

### メタモーフィックテスト

メタモーフィックテスト (metamorphic test) とは、メタモーフィック関係 (metamorphic relation) に基づいたテストを実施する手法です。メタモーフィック関係とは、ある現象の中で普遍的に成り立つ関係を抽出したもので、例えば、正弦関数の  $\sin$  を実装したとしましょう。正弦関数においては、 $\sin(\theta) = \sin(\theta + 2n\pi)$ , ( $n$  は 0 以上の自然数) というメタモーフィック関係が成り立つため、如何なる  $n$  に対しても、 $n = 0$  の場合と同じ値になるはずです。仮にテスト結果がそうならない場合、実装もしくは環境にバグがあると考えられます。

もう 1 つ、メタモーフィック関係の例を挙げます。記述統計の値を算出するプログラムにおいて、算術平均 (average), 分散 (variance), 標準偏差 (standard deviation) を算出する関数を実装したとします。これらの関数に対しては、複数の値を与えて目的の値を算出させるのですが、これらの値自体は変えずに、ソートの順を変えて値の順を変えたものを、上記の関数に渡してやることができます。当然、結果が変わらなければなりません。しかし、仮にソートして値の順を変更した結果、実行結果が異なるのであれば、何らかのバグがあると推定できます。

本課題においては、課題 2 のプログラムにおいてメタモーフィック関係を見いだすことができます。課題 2 のプログラムは、渡されたソースコードを内容は変えずに、見やすく段付けしたものを出力します。この出力結果 (A とします) を再度課題 2 のプログラムに渡すと、出力結果 (B とします) を得ます。ここで、A と B は同じ結果になるべきです。もし、A と B が異なるのであれば、それはプログラムに何らかの抜けがあつたりする可能性が高いといえます。こうしたメタモーフィック関係は対等性といいます。

注意しなければならないのは、メタモーフィックテストはブラックボックステストやホワイトボックステストとは異なり、仕様はほとんど見ていないということです。判定できるのは「こうなるべき」という現象そのものが本質的に持つ性質のみです。例えば、上記  $\sin$  の例でも、例えば  $\sin(0) = 1$  と表示してしまうプログラムが、 $\sin(2\pi) = 1, \sin(4\pi) = 1$  と出力するかもしれません。これは出力の内容は間違っていますが、メタモーフィック関係的には正しい結果です。そのため、メタモーフィックテストのみをもって十分なテストができるとは言えません。あくまで、テストの観点を変えたものの 1 つです。



他にも、gcc に対して、あるプログラムをコンパイルしたときに全く利用しなかった部分を削除した gcc を作って、再度同じプログラムをコンパイルすると同じ結果にならなかった、ということから大量の gcc の不具合が見つかったという事例もあるよ。

# 1.9

## スケジューリングと進捗の把握



### 1.9.1

#### 事前計画(スケジュール)

終えるために時間のかかる作業(仕事、タスク)をするときには、スケジュールを立てることが重要です。特に作業結果の質を落とせなくて、〆切も厳密に定められているときに、スケジュールを立てずに作業を行うことは危険過ぎます。

ここでいうスケジュールとは、いつ何を行うかを時刻順に並べて書いたものです。スケジュールがあると次のようなメリットがあります。

- 日々何を行うかが明確なので迷うことがない。
- 何を行うか(作業内容)がわかっているので、その日の作業に必要なものや知識を前もって用意できる。
- 現実がスケジュールよりも遅れ始めたときにすぐ気づけて対処できる。〆切前日に遅れに気づいても時間がないので対処できない(〆切を守れない)。

従って、スケジュールを立てるためには、次の情報が必要になります。

- (a) 作業全体をどのような部分作業(サブタスク)や具体的な行動に分割できるか。
- (b) 各サブタスクの実行の前後関係。例えば、コンパイルが正常に終わらなければテストはできないなど。
- (c) 各サブタスクを行うのに必要と思われる時間(タスクの見積もり時間)。
- (d) この作業に使えない時間帯(睡眠、他の作業を行う時間など)。しかし、これらは、作業の重要度や〆切の切迫度で幾分変動する。

(a)については、作業ができるだけ細かな部分作業に分割するのが望ましいです。なぜなら、部分作業が細かく具体的であればあるほど(c)の見積もり時間が正確になるからです。また、(b)を考えることで、部分作業の抜け(考え方落とし)を防止する効果もあります。

とはいっても、初めて行う作業(今回の演習が該当するかもしれません)の場合は、何をすればよいかがよくわからていないので、(a)で細かく分割できないかもしれません。その結果(c)での見積もりも甘いものにならざるを得なくなります。しかし、作業を始めてしまえば、だんだん何をすればよいかが分かってきます。従って、再スケジュールが必要となります。再スケジュールは、作業の詳細がわかったとき、その結果、見積もり時間が間違っていたことがわかったとき、作業の遅れに気づいたときなどに隨時行うべきです。もちろん、再スケジュールを行っても、最後の〆切を守るように再スケジュールしなくてはなりません。従って、スケジュールは、最後に予備日を置くなど、余裕を持って立てる必要があります。



「進捗は？」あくあたんの出番だ！

### 1.9.2

#### スケジューリングの例(課題1)

作るべきものの詳細がよくわからないので、まず、本日の説明と配付資料からわかる範囲でスケジュールを立てます。次はその例(見積もり時間なし)です。段付けは部分作業

(サブタスク) を表します。

- (a) スケジュールを立てる
- (b) 資料を読む
  - (b-1) 配布された資料を読み直す
  - (b-2) 配布されたプログラムを読む
  - (b-3) コンパイラのテキスト (特にサンプルコンパイラの字句解析系の部分) を読む
- (c) 字句解析系 (スキヤナ) の概略設計 (どのような関数が必要かと関数の外部仕様作成)
- (d) プログラム作成 (コーディング)
  - (d-1) 2.7 節のメインプログラム例のトークンカウント用の配列を初期化部分の作成
  - (d-2) 2.7 節のメインプログラム例のトークンをカウント部分の作成 (スキヤナを利用))
  - (d-3) 2.7 節のメインプログラム例のカウントした結果の出力部分の作成
  - (d-4) スキヤナの作成
- (e) テストプログラムの作成
  - (e-1) ブラックボックステスト用プログラムの作成
    - (e-1-1) ブラックボックステスト用プログラムの作成
    - (e-1-2) バグがない場合の想定テスト結果の準備
  - (e-2) ホワイトボックステスト用プログラムの作成
    - (e-2-1) カバレッジレベルの決定 (何パーセントのカバレッジを目指すか)
    - (e-2-2) ホワイトボックステスト用プログラムの作成
    - (e-2-3) バグがない場合の想定テスト結果の準備
- (f) テストとデバッグを行う
- (g) レポートの作成
  - (g-1) 作成したプログラムの設計情報
  - (g-2) テスト情報
  - (g-3) この課題を行うための事前計画 (スケジュール) と実際の進捗状況
- (h) プログラムとレポートの提出

次にこれらのタスクの関連を図示します。接点がタスクの区切り (サブタスクの開始点や終了点) のポイント、矢印がタスクを表します。従って、グラフの接続関係がタスクの実行の順序制約を表します。このような図の各タスクにタスクを行うのに必要な見積もり時間を付加したものを作成 (PERT) 図といいます。

図 1.1 は書くまでもない簡単な図ですが、サブタスクレベルまで分解して書くと意味のある図になります。例えば、(e-1) は (c) の後直ちに実行可能です。また、早い段階から開始することができますが完全に終えるには他のタスクが終了しなくては駄目なタスクもあり得ます。このようなタスクはさらにサブタスクに分割できことが多いです。

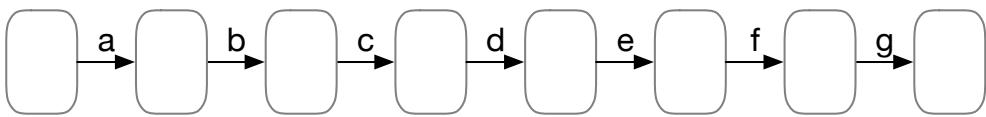


図 1.1 簡単な PERT 図

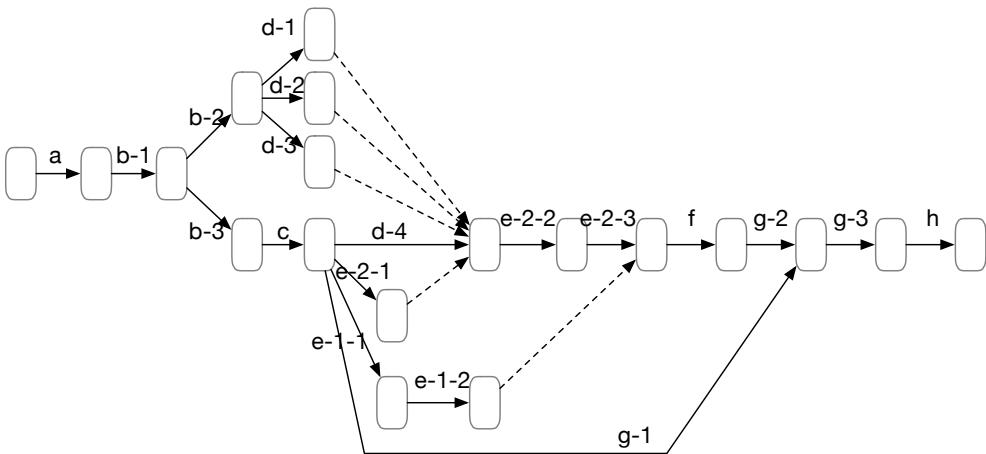


図 1.2 サブタスクレベルの PERT 図

図 1.2 は、サブタスクレベルで書いたパート図の例です。ただし、点線の矢印はダミーのタスクです(2節点間の矢印は高々1本に限るというグラフ理論の制約のため)。パート図はサイクルのない有向グラフになりますので、パート図のタスクを順序制約を守って1列に並べることができます。実際の実行順を決定します。

各タスクの見積もり時間を決定(推定)した後、最後に、各タスクに、各タスクの実行開始日(時刻)と実行終了予定日(時刻)を前から順に決定します(これをレポートに書きます)。このとき、タスク実行のために1日24時間使えると考えてはいけません。また、最後のタスクの実行終了予定日は〆切日よりも前ではならないし、もっと言うなら、十分な予備日が必要です。予備日の日数は、作業見積もりの精度が高ければ少なくともよいですし、低ければそれなりの日数を用意することが必要になります。見積もり時間を多く取ると、(〆切が変わらないとすれば)各タスクで使える時間が少なくなります。言い換えると、作業量がよくわからない時は、早めに仕事を進めよ、ということです。

万一、実際の進捗がスケジュールよりも遅れた場合、対処法は次の通りになります。

- 予備日を減らして未実行タスクの実行日を遅らせ、時間を作る。
- 未実行のサブタスクの見積もり時間を減らして、時間を作る。
- 本来別のことを使う予定であった時間を作業時間に組み込む。例えば、遊びに行くのを止めたり、睡眠時間を減らしたりするなど

いずれにせよ、遅れた理由を究明し、その原因を早急に潰す必要があります。その理由が一般的なものであれば、他の未実行タスクも同様の理由で遅れる可能性もあるので、早めに行動するのが大事です。質問等は、どんな内容でも積極的に行ってください。

表 1.1 に事前作業計画の例を挙げます。また、注意点を以下に挙げます。

- レポートの作成(g)は、対応する作業をするときに記録をちゃんと残しておけば、



睡眠時間は削っちゃダメだよ。あくあたんとの約束だよ。

表 1.1 事前作業計画の例 (日付は過去の例)

開始予定日	終了予定日	見積もり時間	作業内容
10/3	10/3	1	(a) スケジュールを立てる
10/4	10/4	0.5	(b-1) 配布された資料を読み直す
10/4	10/4	0.5	(b-2) 配布されたプログラムを読む
10/4	10/4	1	(b-3) コンパイラのテキスト (プログラム) を読む
10/5	10/7	5	(c) 字句解析系 (スキナ) の概略設計
10/8	10/8	2	(e-1-1) ブラックボックステスト用プログラムの作成
10/9	10/11	5	(d-4) スキナの作成
10/12	10/12	1	(e-1-2) バグがない場合の想定テスト結果の準備
10/13	10/13	0.5	(d-1) トークンカウント用の配列を初期化部分の作成
10/13	10/13	0.5	(d-2) トークンをカウント部分の作成
10/13	10/13	1	(d-3) カウントした結果の出力部分の作成
10/14	10/14	0.5	(e-2-1) カバレッジレベルの決定
10/14	10/14	2	(e-2-2) ホワイトボックステスト用プログラムの作成
10/15	10/15	1	(e-2-3) バグがない場合の想定テスト結果の準備
10/16	10/20	8	(f) テストとデバッグを行う
10/28	10/28	1	(g-1) 作成したプログラムの設計情報を書く
10/29	10/29	1	(g-2) テスト情報を書く
10/30	10/30	1	(g-3) 事前計画と実際の進捗状況を書く
10/31	10/31	-	(h) プログラムとレポートの提出

それをコピーするだけですむはず.

- プログラムとレポートの提出予定日が〆切日より早いのは、予備時間を持っておくためと課題 2 に食い込むと課題 2 が大変になるため。
- テストとデバッグの日程が長いのは、必要なテスト量が現時点では十分に見積もれないとため。
- 講義やその他の予定の都合により、個人個人の計画はこのスケジュール通りにはならない。
- (f) と (g) の間は予備日である。
- 表内の見積もり時間等は、単なる例であり、いい加減な数値である。各自で見積もること。
- これは表形式で記述されているが、実務では(複数人でのプロジェクトのため)線表を書くことが多い。
- 先の作業量が読めないときは、計画が表 1.2 の通り単純になってしまう。しかし、実際に手を付けると必要時間が見えてくるので、その都度計画をより詳細にするのが望ましい。

表 1.2 簡単な事前作業計画の例 (日付は過去の例)

開始予定日	終了予定日	見積もり時間	作業内容
10/3	10/3	1	(a) スケジュールを立てる
10/4	10/7	3	(b) 配布された資料等を読む
10/7	10/10	5	(c) 字句解析系(スキナ)の概略設計
10/10	10/20	10	(d) スキナの作成(コーディング)
10/20	10/22	3	(e) テストデータと想定テスト結果の準備
10/22	10/28	8	(f) テストとデバッグを行う
10/29	10/30	1	(g) レポート作成
10/31	10/31	-	(h) プログラムとレポートの提出

✓ 適用  保存

#	トラッカー	ステータス	題名 ^	開始日	期日	更新日	...
18350	言プロ課題	新規	01 課題1: 字句の出現頻度表の作成	2024/09/30	2024/10/30	2024/09/24 21:49	...
18351	設計	新規	> 01.01 スケジュールを立てる	2024/09/30		2024/09/24 14:57	...
18352	設計	新規	> 01.02 資料を読む			2024/09/24 15:00	...
18353	設計	新規	> 01.03 字句解析系(スキナ)の概略設計			2024/09/24 15:00	...
18354	コーディング	新規	> 01.04 プログラム作成(コーディング)			2024/09/24 15:02	...
18359	テスト	新規	> 01.05 テストプログラムの作成			2024/09/24 15:01	...
21190	提出	新規	> 01.06 プログラムの提出	2024/10/21	2024/10/30	2024/09/25 20:36	...
24054	提出	新規	> 01.07 レポートの提出	2024/10/21	2024/10/30	2024/09/25 20:36	...
21177	言プロ課題	新規	02 課題2: プリティプリンタの作成	2024/10/21	2024/11/25	2024/09/24 21:49	...
21178	設計	新規	> 02.01 スケジュールを立てる	2024/10/21		2024/09/24 15:04	...
21179	設計	新規	> 02.02 資料を読む			2024/09/24 15:05	...
21180	設計	新規	> 02.03 構文解析系(パーサー)の概略設計			2024/09/24 15:05	...
21181	コーディング	新規	> 02.04 プログラム作成(コーディング)			2024/09/24 15:05	...
21186	テスト	新規	> 02.05 テストの実施			2024/09/24 15:06	...
21415	提出	新規	> 02.06 プログラムの提出	2024/11/19	2024/11/25	2024/09/25 20:36	...
24055	提出	新規	> 02.07 レポートの提出	2024/11/19	2024/11/25	2024/09/25 20:36	...
21416	言プロ課題	新規	03 課題3: クロスリファレンサの作成	2024/11/19	2024/12/16	2024/09/24 21:49	...
21417	設計	新規	> 03.01 スケジュールを立てる	2024/11/19		2024/09/24 15:07	...
21418	設計	新規	> 03.02 資料を読む			2024/09/24 15:08	...

図 1.3 Redmine のチケット一覧

### 1.9.3 Redmine によるスケジューリングの記録

各学生ごとに Redmine の「言語処理プログラミング(XXX)」というプロジェクトを作成しています。このプロジェクトには上記のスケジューリングについて、簡単にまとめたものをあらかじめ配置しています。課題で為すべきことは「言プロ課題」という 4 つのチケットの子チケットとして実装しています。言プロ課題は全てのサブチケットが終了すると終わらせることができます。各チケットには「新規」「進行中」「審査待ち」「終了」というステータスが存在します。これを適宜変更していくことで、作業の開始と終了を把握できます。「開始日」や「期日」といった項目は自分で設定してください。

子チケットのうち、「提出」とされるチケットはレポートとプログラムの提出を行う場所ですが、このチケットを終了させることができるのは教員のみです。レポートとプログラムの教員による確認が終わったら各自で「言プロ課題」を終了させてください。また、

## 言プロ課題 #18350

 編集  時間を記録

進捗: 01 課題1: 字句の出現頻度表の作成  
 水野 修 が1年以上前に追加. 約23時間前に更新.

ステータス:	新規	開始日:	2024/09/30
優先度:	通常	期日:	2024/10/30
担当者:	-	進捗率:	<div style="width: 0%; background-color: #ccc; height: 10px;"></div> 0%
予定工数: (合計: 0:00時間)			

ステータス--> [新規] [進行中]

説明  
言語処理プログラミング課題1

子チケット

設計 #18351: 01.01 スケジュールを立てる	新規	<div style="width: 0%; background-color: #ccc; height: 10px;"></div>
設計 #18352: 01.02 資料を読む	新規	<div style="width: 0%; background-color: #ccc; height: 10px;"></div>
設計 #18353: 01.03 字句解析系 (スキナ) の概略設計	新規	<div style="width: 0%; background-color: #ccc; height: 10px;"></div>
コーディング #18354: 01.04 プログラム作成 (コーディング)	新規	<div style="width: 0%; background-color: #ccc; height: 10px;"></div>
テスト #18359: 01.05 テストプログラムの作成	新規	<div style="width: 0%; background-color: #ccc; height: 10px;"></div>
提出 #21190: 01.06 プログラムの提出	新規	<div style="width: 0%; background-color: #ccc; height: 10px;"></div>
提出 #24054: 01.07 レポートの提出	新規	<div style="width: 0%; background-color: #ccc; height: 10px;"></div>

関連するチケット

図 1.4 Redmine のチケット

子チケットに対してさらに詳細な作業を孫チケットとして追加できます。これにより、細かい作業を記録できます。課題1のプロジェクトには、例としてこのテキストで説明した作業の流れが孫チケットとして登録されています。課題2以降は大枠だけ準備していますので、孫チケットによる詳細な流れは自分で作成してください。

Redmine のプロジェクトにおいて、適切に作業や開始・終了日時の記録を行った場合、レポートにおけるスケジューリングに関する記載を免除します。活用してください。





# 2

## 課題 1 - 字句解析

### 2.1

#### 課題: 字句の出現頻度表の作成



課題内容関連講義予定日: 2024-09-30(月)

演習期間: 2024-09-30(月) ~ 2024-10-20(日)

レポート・プログラム提出期間: 2024-10-21(月) ~ 2024-10-28(月)

**課題の概略:** プログラミング言語 MPPL(別紙参照) で書かれたプログラムらしきものを読み込み、字句(トークン)がそれぞれ何個出現したかを数え、出力するプログラムを作成する。

**キー技術:** テキスト処理、字句解析



らしきもの…

### 2.2

#### 課題説明



プログラミング言語 MPPL で書かれたプログラムらしきものを読み込み、字句(トークン)がそれぞれ何個出現したかを数え、出力する C プログラムを作成する。例えば、作成するプログラム名を `tc`, MPPL で書かれたプログラムらしきもののファイル名を `foo mpl` とするとき、コマンドラインからのコマンドを

```
# ./tc foo mpl
```

とすれば (`foo mpl` のみが引数), `foo mpl` 内の字句の出現個数を出力するプログラムを作成する。

**入力** MPPL で書かれたプログラムらしきもののファイル名。コマンドラインから与える。字句(トークン)は、名前、キーワード、符号なし整数、文字列、記号のいずれかである。ただし、キーワードと記号については、それぞれの記号列が別々の字句であるが、名前、符号なし整数、文字列については、その実体が異なっていても同じ「名前」、「符号なし整数」、「文字列」という字句であるとする。字句の定義として、MPPL のプログラムのマイクロ構文を図 2.1 のように与える(左辺の括弧の中は非終端記号の英語表記であり参考のために付加してある)。なお、終端記号は

```

プログラム (program) ::= { 字句 | 分離子 }
字句 (token) ::= 名前 | キーワード | 符号なし整数 | 文字列 | 記号
名前 (name) ::= 英字 { 英字 | 数字 }
キーワード (keyword) ::= "p" "r" "o" "g" "r" "a" "m" | "v" "a" "r" |
    "a" "r" "a" "y" | "o" "f" | "b" "e" "g" "i" "n" | "e" "n" "d" |
    "i" "f" | "t" "h" "e" "n" | "e" "l" "s" "e" |
    "p" "r" "o" "c" "e" "d" "u" "r" "e" | "r" "e" "t" "u" "r" "n" |
    "c" "a" "l" "l" | "w" "h" "i" "l" "e" | "d" "o" | "n" "o" "t" |
    "o" "r" | "d" "i" "v" | "a" "n" "d" | "c" "h" "a" "r" |
    "i" "n" "t" "e" "g" "e" "r" | "b" "o" "o" "l" "e" "a" "n" |
    "r" "e" "a" "d" | "w" "r" "i" "t" "e" | "r" "e" "a" "d" "l" "n" |
    "w" "r" "i" "t" "e" "l" "n" | "t" "r" "u" "e" |
    "f" "a" "l" "s" "e" | "b" "r" "e" "a" "k"
符号なし整数 (unsigned integer) ::= 数字 { 数字 }
文字列 (string) ::= "" { 文字列要素 | "" "" } ""
# ""はアポストロフィ(シングルクオート)である
記号 (symbol) ::= "+" | "-" | "*" | "=" | "<" ">" | "<" | "<" "=" |
    ">" | ">" "=" | "(" | ")" | "[" | "]" | ":" "=" | "." | "," |
    ":" | ";"
英字 (alphabet) ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
    "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" |
    "t" | "u" | "v" | "w" | "x" | "y" | "z" |
    "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" |
    "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" |
    "W" | "X" | "Y" | "Z"
数字 (digit) ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
    "8" | "9"
分離子 (separator) ::= 空白 | タブ | 改行 | 注釈
注釈 (comment) ::= "{" { 注釈要素 } "}" | "/" "*" { 注釈要素 } "*" "/"

```

図 2.1 MPPL のマイクロ構文

ASCII 文字である。

### 図 2.1 のマイクロ構文での表記について

- 文字列要素 (string element) : アポストロフィ "", 改行以外の任意の表示文字を表す。
- 文字列には長さの概念があり、その文字列に含まれるアポストロフィ "", 改行以外の任意の表示文字の数と連続する "" "" の組の数の和である。即ち "" については連続する 2 つを 1 と数える (同様に連続する 4 つを 2 と数える。以下同じ)。言い換えると、文字列のマイクロ構文での { } の繰り返し回数が文字列の長さである。この文字列の長さの概念は、課題 3 の制約規則 (図 4.2)において利用する。
- 注釈要素 (comment element) : 注釈が "{}" で囲まれているときは、注釈要素は閉じ中括弧 "}" 以外の任意の表示文字を表し、 "/" "\*", "\*" "/" で囲まれているときは、連続する注釈要素として "\*" "/" が現れないことを除いて任

文字列の長さについては、課題 1 の時点では特に意識する必要はないよ。



意の表示文字を表す。注釈が開始されたまま EOF を迎えた場合には、EOF までを注釈として処理する。

- 空白 (space), タブ (tab) は、ASCII コードではそれぞれ 20, 09(16 進数表示) であり、C 言語上では、「', '\t」で表現される。
- 改行 (end of line) は OS によって異なるので、「\r', '\n', '\r' '\n', '\n' '\r」の 4 通りの ASCII コード (列) を一つの改行として扱うこと。ただし、「\r', '\n」は ASCII コードではそれぞれ 0D, 0A(16 進数表示) である。
- 表示文字 (graphic character) とは、タブ、改行と通常画面に表示可能な文字 (ASCII では 20 から 7E(16 進数表示) までの文字) を意味する。
- 表示文字ではない文字コード (タブ、改行以外の制御コード) が現れた場合は、存在しないものとして無視してよい。もちろん、エラーとしてもよい。
- 制約規則としては、
  - 最長一致規則：字句として、二つ以上の可能性があるときは最も長い文字の列を字句とする、
  - 英大文字と小文字は区別する、
  - キーワードは予約されている：キーワードは名前ではない、がある。
- なお、このファイルは ASCII コードによるテキストファイルであるとしてよい。あまりにも長い行 (例えば、1 行 1000 文字以上など) はないものとしてよいが、あつたとしても、作成したプログラムが実行時エラーを起こしてはいけない。



「\n' '\r' なんて見たことないけど  
ねえ

**【注意】** 構文において、用いられている記号の意味は次の通りである。

- " " 終端記号であることを示す。
- ::= 左辺の非終端記号が右辺で定義されることを示す
- | 左側と右側のどちらかであることを示す
- { } 内部を 0 回以上繰り返すことを表す
- [ ] 内部を省略してもよい (0 回か 1 回) ことを示す

**出力** 字句とその出現数の表。標準出力へ出力する。出力の仕様は以下の通りである。

- 表の 1 行の行頭に字句をダブルクオーテーションで囲んで表示する。字句には余分なスペースが含まれていても良い。
- 続けて、1 つ以上の空白文字 (空白、タブ) を表示し、字句の個数を表示する。
- 1 行には 1 つの字句についての情報のみを表示する。
- 名前 (NAME), 文字列 (STRING), 符号なし整数 (NUMBER) はその実体が異なっていてもそれぞれを同じ字句として扱う。
- 出現しない字句については出力しない (出力中に個数 0 の行はない)。

以上の仕様に基づくと、出力は以下のようになる。なお、分離子は字句ではないの

で、注釈などについては出力する必要はない。

```
"and      "    10
"array   "     2
.....
"NAME"      38
"NUMBER"    12
"STRING"    2
```

また、字句や分離子を構成しない文字が現れたとき(コンパイラとしてはエラーである)は、その旨(エラーメッセージ)を標準エラー出力(stderr)へ出力し、ファイルの先頭からそこまでの部分について、出現数の表を標準出力(stdout)へ出力せよ。下記の字句解析系がエラーを検出した場合も同様である。

## 2.3

### プログラム作成条件



入力から、字句を切り出す字句解析系(スキヤナ)と、字句を数え、表を作成する主手  
続きとに分割して作成せよ。字句解析系は後の課題で再利用するため、以下のようなモ  
ジュール仕様とせよ。

#### 2.3.1

##### 初期化関数

```
int init_scan(char *filename)
```

filename が表すファイルを入力ファイルとしてオープンする。

**戻り値** 正常な場合 0 を返し、ファイルがオープンできない場合など異常な場合は負の値  
を返す。

#### 2.3.2

##### トークンを一つスキャンする関数

```
int scan()
```

次のトークンのコードを返す。トークンコードは別ファイル(token-list.h)参照のこと。  
End-of-File 等次のトークンをスキャンできないとき、戻り値として負の値を返す。

#### 2.3.3

##### 定数属性

```
int num_attr;
```

scan() の戻り値が「符号なし整数」のとき、その値を格納している。なお、32768 より  
も大きい値の場合は、エラーである。

#### 2.3.4

##### 文字列属性

```
char string_attr[MAXSTRSIZE];
```

scan() の戻り値が「名前」または「文字列」のとき、その実際の文字列を格納している。  
また、それが「符号なし整数」のときは、入力された数字列を格納している。その文字  
列(数字列、名前)は、'\0' で終端されている。例えば、文字列が 'It''s' のときには、  
string\_attr には先頭から順に、'I', 't', '\'', '\'', 's', '\0' が格納される。  
なお、文字列の定義でも述べたとおり、この文字列の長さは 4 とする(2 つの'を 1 つと  
数える)。

もし、string\_attr に格納できないくらい長い文字列(数字列、名前)の場合には、エ  
ラーとする。

プログラムから見た文字列長とは  
異なるので注意してね。本来は、  
格納する時点では' 'を' 'に変換して  
しまえば良いのだけど、課題 4 で  
利用する CASL II でもアポストロ  
フィを表すのに' 'を利用するんだ。  
だから、この時点では変換しない  
で格納した方が良いのだよ。



### 2.3.5

#### 行番号関数

```
int get_lineno()
```

もっとも最近に `scan()` で返されたトークンが存在した行の番号を返す。まだ一度も `scan()` が呼ばれていないときには 0 を返す。

### 2.3.6

#### 終了処理関数

```
void end_scan()
```

`init_scan(filename)` でオープンしたファイルをクローズする。

ただし、必要に応じてこれら以外のインターフェース関数を用意してもかまわない。

### 2.3.7

#### エラー処理

これまでに触れたエラーに限らず、すべてのエラーメッセージは**標準エラー出力**に出力することとする。また、エラーを検出した時点でプログラムは終了させる。

## 2.4

### 入出力例



本課題の入力例は以下の通り。

入力ファイル例 (`sample11pp.mpl`)

```
program sample11pp;
procedure kazuyomikomi(n : integer);
begin
  writeln('input the number of data');
  readln(n)
end;
var sum : integer;
procedure wakakidasi;
begin
  writeln('Sum of data = ', sum)
end;
var data : integer;
procedure goukei(n, s : integer);
  var data : integer;
begin
  s := 0;
  while n > 0 do begin
    readln(data);
    s := s + data;
    n := n - 1
  end
end
```

```

end;
var n : integer;
begin
  call kazuyomikomi(n);
  call goukei(n * 2, sum);
  call wakakidasi
end.

```

プログラム sample11pp.mplに対する出力例

"NAME	"	27
"program	"	1
"var	"	4
"begin	"	5
"end	"	5
"procedure	"	3
"call	"	3
"while	"	1
"do	"	1
"integer	"	6
"readln	"	2
"writeln	"	2
"NUMBER	"	4
"STRING	"	2
"+"	"	1
"-	"	1
"*"	"	1
"	1	
"("	"	8
"	8	
"":=	"	3
"."	"	1
","	"	3
";"	"	6
";"	"	17

## 2.5

### 拡張仕様



もし、余裕があれば、名前についてはその実体ごとにも出現個数を数えて出力するよう拡張してみよ。つまり、n, sum 等の名前毎にも出現個数を数えて出力する。出力の際は、行頭に "Identifier" を付して名前を続けること。sample11pp.mplに対する拡張仕様の出力は以下のようになる。

拡張仕様の出力 (sample11pp.mpl)

"NAME"	27
"Identifier" "s"	4
"Identifier" "goukei"	2
"Identifier" "data"	4

```
"Identifier" "wakakidasi"      2
"Identifier" "sum"            3
"Identifier" "n"              9
"Identifier" "kazuyomikomi"   2
"Identifier" "sample11pp"     1
"program"        1
"var"           4
"begin"         5
"end"          5
"procedure"      3
"call"          3
"while"         1
"do"            1
"integer"       6
"readln"        2
"writeln"       2
"NUMBER"        4
"STRING"        2
"+"
```

1

```
"_"
"@"
">"
"("
)"
":="
"."
","
";"
";"
17
```

## 2.6

### スキヤナ用ヘッダファイル



```
#ifndef SCAN_H
#define SCAN_H

/* scan.h */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAXSTRSIZE 1024

/* Token */
#define TNAME 1 /* Name : Alphabet { Alphabet | Digit } */
#define TPROGRAM 2 /* program : Keyword */
#define TVAR 3 /* var : Keyword */
#define TARRAY 4 /* array : Keyword */
#define TOF 5 /* of : Keyword */
#define TBEGIN 6 /* begin : Keyword */
#define TEND 7 /* end : Keyword */
#define TIF 8 /* if : Keyword */
#define TTHEN 9 /* then : Keyword */
#define TELSE 10 /* else : Keyword */
#define TPROCEDURE 11 /* procedure : Keyword */
#define TRETURN 12 /* return : Keyword */
#define TCALL 13 /* call : Keyword */
#define TWHILE 14 /* while : Keyword */
#define TDO 15 /* do : Keyword */
#define TNOT 16 /* not : Keyword */
#define TOR 17 /* or : Keyword */
#define TDIV 18 /* div : Keyword */
#define TAND 19 /* and : Keyword */
#define TCHAR 20 /* char : Keyword */
#define TINTEGER 21 /* integer : Keyword */
#define TBOOLEAN 22 /* boolean : Keyword */
#define TREADLN 23 /* readln : Keyword */
#define TWRITELN 24 /* writeln : Keyword */
#define TTRUE 25 /* true : Keyword */
#define TFALSE 26 /* false : Keyword */
#define TNUMBER 27 /* unsigned integer */
#define TSTRING 28 /* String */
#define TPLUS 29 /* + : symbol */
#define TMINUS 30 /* - : symbol */
#define TSTAR 31 /* * : symbol */
#define TEQUAL 32 /* = : symbol */
#define TNONEQ 33 /* <> : symbol */
#define TLE 34 /* < : symbol */
#define TLEEQ 35 /* <= : symbol */
#define TGR 36 /* > : symbol */
#define TGREQ 37 /* >= : symbol */
```

```

#define TLPAREN 38 /* ( : symbol */
#define TRPAREN 39 /* ) : symbol */
#define TLSQPAREN 40 /* [ : symbol */
#define TRSQPAREN 41 /* ] : symbol */
#define TASSIGN 42 /* := : symbol */
#define TDOT 43 /* . : symbol */
#define TCOMMA 44 /* , : symbol */
#define TCOLON 45 /* :: symbol */
#define TSEMI 46 /* ; : symbol */
#define TREAD 47 /* read : Keyword */
#define TWRITE 48 /* write : Keyword */
#define TBREAK 49 /* break : Keyword */

#define NUMOFTOKEN 49

#define KEYWORDSIZE 28

#define S_ERROR -1

extern struct KEY {
    char * keyword;
    int keytoken;
} key[KEYWORDSIZE];

extern int error(char *mes);

extern int init_scan(char *filename);
extern int scan(void);
extern int get_linenum(void);
extern void end_scan(void);

extern int num_attr;
extern char string_attr[MAXSTRSIZE];

#endif

```

## 2.7

### 課題 1 メインプログラム例 (サンプル)



```
#include "scan.h"

/* keyword list */
struct KEY key[KEYWORDSIZE] = {
    {"and", TAND}, {"array", TARRAY}, {"begin", TBEGIN},
    {"boolean", TBOOLEAN}, {"break", TBREAK}, {"call", TCALL},
    {"char", TCHAR}, {"div", TDIV}, {"do", TDO},
    {"else", TELSE}, {"end", TEND}, {"false", TFALSE},
    {"if", TIF}, {"integer", TINTEGER}, {"not", TNOT},
    {"of", TOF}, {"or", TOR}, {"procedure", TPROCEDURE},
    {"program", TPROGRAM}, {"read", TREAD}, {"readln", TREADLN},
    {"return", TRETURN}, {"then", TTHEN}, {"true", TTRUE},
    {"var", TVAR}, {"while", TWHILE}, {"write", TWRITE},
    {"writeln", TWRITELN}};

/* Token counter */
int numtoken[NUMOFTOKEN + 1];

/* string of each token */
char *tokenstr[NUMOFTOKEN + 1] = {
    "", "NAME", "program", "var", "array", "of",
    "begin", "end", "if", "then", "else", "procedure",
    "return", "call", "while", "do", "not", "or",
    "div", "and", "char", "integer", "boolean", "readln",
    "writeln", "true", "false", "NUMBER", "STRING", "+",
    "-", "*", "=", "<>", "<", "<=",
    ">", ">=", "(", ")",
    ":=", ".",
    ";", ":", "read",
    "write", "break"};

int main(int nc, char *np[]) {
    int token, i;

    if (nc < 2) {
        error("File name is not given.");
        return 0;
    }
    if (init_scan(np[1]) < 0) {
        error("Cannot open input file.");
        end_scan();
        return 0;
    }
    /* 作成する部分: トークンカウント用の配列? を初期化する */
    while ((token = scan()) >= 0) {
        /* 作成する部分: トークンをカウントする */
    }
    end_scan();
    /* 作成する部分: カウントした結果を出力する */
}
```

```

        return 0;
    }

int error(char *mes) {
    fprintf(stderr, "\nERROR: %s\n", mes);
    return S_ERROR;
}

```

## 2.8

### おまけ



拡張仕様を考える参考にできるコード。

id-list.c

```

#include "scan.h"

struct ID {
    char *name;
    int count;
    struct ID *nextp;
} *idroot;

void init_idtab() { /* Initialise the table */
    idroot = NULL;
}

struct ID *search_idtab(char *np) { /* search the name pointed by np */
    struct ID *p;

    for (p = idroot; p != NULL; p = p->nextp) {
        if (!strcmp(np, p->name))
            return (p);
    }
    return (NULL);
}

void id_countup(char *np) { /* Register and count up the name pointed by np */
    struct ID *p;
    char *cp;

    if ((p = search_idtab(np)) != NULL)
        p->count++;
    else {
        if ((p = (struct ID *)malloc(sizeof(struct ID))) == NULL)
            error("Cannot malloc for p in id_countup");
        return;
    }
    if ((cp = (char *)malloc(strlen(np) + 1)) == NULL)
        error("Cannot malloc for cp in id_countup");

```

```

        return;
    }
    strcpy(cp, np);
    p->name = cp;
    p->count = 1;
    p->nextp = idroot;
    idroot = p;
}
}

void print_idtab() { /* Output the registered data */
    struct ID *p;

    for (p = idroot; p != NULL; p = p->nextp) {
        if (p->count != 0)
            printf("\t\"Identifier\" \"%s\"\t%d\n", p->name, p->count);
    }
}

void release_idtab() { /* Release tha data structure */
    struct ID *p, *q = NULL;

    for (p = idroot; p != NULL; p = q) {
        free(p->name);
        q = p->nextp;
        free(p);
    }
    init_idtab();
}

```

## 2.9

### スキヤナの作り方のヒント



スキヤナの作成は、かなり難しそうに思えますが、次の点を踏まえれば、見通しがよくなります。

- 次に読み込まれる字句は、先頭の文字で、ほぼ何であるかが決まる。

例えば、一文字読み込んで、それが英字であれば、そこから始まる字句は名前かキーワードになります。また、数字であれば、符号なし整数です。 "+"であれば、記号 "+"しかり得ません。他も同様です。分離子があるかもしれないのに、それを考慮すると全体の構成は次のようになります。

先頭の文字が分離子であれば、それを読み飛ばす（注釈の時は注釈全体を読み飛ばす）

分離子以外の文字であれば、次のように場合分けする

英字なら、英数字が続く限り読み込む。それがキーワードのどれかならそのキーワードである  
どのキーワードとも異なれば、名前である。

数字なら、数字が続く限り読み込む。それは符号なし整数である。

.....

- 最長一致原則がある

例えば, "abc"と入力があった場合, aだけでも名前ですが, abもabcも名前です。cの次はスペースなので, 最長の字句はabcとなり, 3文字で1つの字句となります。また, "10abc"と入力があった場合には, 10で1つの字句(符号なし整数)でabc以降は次の字句となります(10aなどは字句ではありません)。つまり, 字句は次の文字まで確認しないと確定しない場合があります(記号 "+" のように確定するものもあります)。従って, 入力は常に1文字先読みしておくと便利です。即ち, 1文字分の文字バッファを持っておき, それに常に次の文字が入っているようにします。以降, この文字バッファを

```
int cbuf;
```

として, 説明します。

## 2.9.1 初期化関数

```
int init_scan(char *filename)
```

filenameが表すファイルを入力ファイルとしてオープンします。オープンに失敗したらエラーで終わります。

成功したら, そのファイルから1文字 cbuf に読んでおきます。もし, このとき, EOF が起こったら cbuf には EOF コードを入れます。

## 2.9.2 トークンを一つスキャンする関数

```
int scan()
```

switch文で処理が分かれます。

```
switch(cbuf) {
    case 空白やタブ: 読み飛ばして, cbuf に次の文字を入れて, 最初に戻る.
    case 英字: 名前かキーワードである.
        英数字が続く限り, cbuf から適当な文字列バッファへ取り出し,
        cbuf へ次の文字を読み込むことを繰り返す.
        cbuf の内容が英数字以外になったら, そこで英数字列が終わるので,
        文字列バッファの中身がキーワードかどうか判別して,
        キーワードならそのトークンコードを,
        それ以外なら名前のトークンコードを返す.
    case 数字: 数字が続く限り読み込んで,
        その数字列が表す値を num_attr に入れ,
        「符号なし整数」のトークンコードを返す.
    case 注釈: 字句と同様に注釈の終わりまで読むが,
        トークンコードを返さずに先頭に戻る
    case 他のトークン: 同様である. (以下略)
```

もちろん, switch文の代わりに, if(...)...else if(...)...を使ってもよいですし, これらを組み合わせても構いません。

ただし、EOF の扱いには、気を付けましょう。特に、文字列やコメント内の処理中に EOF が現れると無限ループに陥りがちです。そのような場合でも無限ループにならないようにしましょう。

### 2.9.3

#### 行番号関数

```
int get_linenumber()
```

行番号を数える変数を用意し、`cbuf` に改行を読み込んだときに、行番号をカウントアップすればよいでしょう。ただし、前述のように改行が 2 つの文字コードの並びである可能性があることに注意しましょう。初期化関数でこの変数を初期化しておきます。この関数が呼ばれた時にその変数の値を返せばよいように思えますが、スキーナの内部で先読みをする場合があり、先読み文字として `newline` を読んだときには行番号がずれることがあります。それを避けるために、トークンの 1 字目を読んだときに、そのトークン用の行番号を確定するようにするとよいでしょう。そして、そのトークンを `scan()` で返してから次の `scan()` が呼ばれるまではその番号を返すようにします。

### 2.10

#### テストケースの意味



以下では配布しているテストプログラム（らしきもの）の一部について、意図している検査内容を示します。

`sample11mpl` ごく一般的なプログラム。1 から入力した数までの和を表示

`sample011mpl` 「NAME」「STRING」という識別子をきちんと処理できるか

`sample12mpl` 本仕様で考え得る一番短いプログラム

`sample012mpl` 異常に長い識別子（字句エラーにすべき）

`sample12crmpl` 改行記号 = CR

`sample12crlfmpl` 改行記号 = CR + LF

`sample12lfmpl` 改行記号 = LF

`sample12lfcrmpl` 改行記号 = LF + CR

`sample012eofmpl` 識別子直後が EOF（字句エラーにはならない）

`sample012nmpl` 異常に長い数値（字句エラーにすべき）

`sample012neofmpl` 数値直後に EOF（字句エラーにはならない）

`sample012smpl` 異常に長い文字列（字句エラーにすべき）

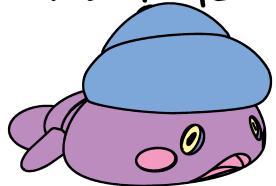
`sample012seofmpl` 文字列開始後に EOF（字句エラーにすべき）

`sample014mpl` 定義されていない字句（字句エラーにすべき）

`sample014ampl` 定義されていない字句（字句エラーにすべき）

`sample014bmpl` 識別子と記号の分離・定義されていない字句との区別（字句エラーにすべき）

つかれた…



# 3



## 課題 2 - 構文解析

### 3.1

#### 課題: プリティプリンタの作成



課題内容関連講義予定日: 2024-10-21(月)

演習期間: 2024-10-21(月) ~ 2024-11-24(日)

レポート・プログラム提出期間: 2024-11-25(月) ~ 2024-12-02(月)

**課題の概略:** プログラミング言語 MPPL で書かれたプログラムを読み込み、構文エラーがなければ、入力されたプログラムをプリティプリントした結果を出力し、構文エラーがあれば、そのエラーの情報(エラーの箇所、内容等)を少なくとも一つ出力するプログラムを作成する。

**キー技術:** 再帰呼び出し、構文解析

### 3.2

#### 課題説明



プログラミング言語 MPPL で書かれたプログラムを読み込み、LL(1) 構文解析法により構文解析を行い、構文エラーがなければ入力されたプログラムをプリティプリントした結果を出力し、構文エラーがあればそのエラーの情報(エラーの箇所、内容等)を少なくとも一つ出力するプログラムを作成する。例えば、作成するプログラム名を pp、MPPL で書かれたプログラムのファイル名を foo.mpl とするとき、コマンドラインからのコマンドを

```
$ ./pp foo.mpl
```

とすれば (foo.mpl のみが引数)、foo.mpl の内容のプリティプリントを出力するプログラムを作成する。

#### 3.2.1

##### 入力

MPPL で書かれたプログラムのファイル名、コマンドラインから与える。MPPL のマイクロ構文は課題 1 の通りである。マクロ構文は図 3.1 の通り(左辺の括弧の中は非終端

記号の英語表記であり参考のために付加してある). 終端記号は字句 (token) である. なお, 「#」以降は構文に関する制約や注意である.

**注意** 図 3.1 の構文において, 用いられている記号の意味は次の通りである.

- " " 終端記号であることを示す
- ::= 左辺の被終端記号が右辺で定義されることを示す
- | 左側と右側のどちらかであることを示す
- { } 内部を 0 回以上繰り返すことを表す
- [ ] 内部を省略してもよい (0 回か 1 回) ことを示す
- ( ) 優先順位を変更する. 通常は「|」がもっとも弱いが, 「( )」により, その内部が優先される
- ε 空語を表す.

「出力指定」の「文字列」に関して補足する. この生成規則

```
出力指定 (output format) ::= 式 [ ":" "符号なし整数" ] | "文字列"  
# この"文字列"の長さ (文字数)は 1以外である.  
# 長さが 1の場合は式から生成される定数の一つである"文字列"とする.
```



これは, writeln('c':8) みたいな出力文を想定しているよ. 出力文では, 変数や定数の桁数を指定できるけど, 長さ 2 以上または長さ 0 の文字列に対しては桁数指定はできないんだ.

では, 出力指定に文字列が出現した場合に, 「式」から導出された「文字列」なのか, 右辺の最後に書いてある「文字列」なのかの判断ができない. そのため, 文字列の長さが 1 以外であれば, ここでの「文字列」を採用し, そうでなければ「式」内の文字列とする.

```
式 (expression) ::= 単純式 { 関係演算子 単純式 }  
# 関係演算子は左に結合的である.  
単純式 (simple expression) ::= [ "+" | "-" ] 項 { 加法演算子 項 }  
# 加法演算子は左に結合的である.  
項 (term) ::= 因子 { 乗法演算子 因子 }  
# 乗法演算子は左に結合的である.  
因子 (factor) ::= 変数 | 定数 | "(" 式 ")" | "not" 因子 | 標準型 "(" 式 ")"  
定数 (constant) ::= "符号なし整数" | "false" | "true" | "文字列"
```

逆に言えば, 「出力指定」においてはまず「式」を評価するが, 「式」が「文字列」のみであった場合は, その長さが 1 であればそのまま「式」を採用し, そうでなければ, 出力指定の「文字列」を採用する.

### 3.2.2

### 出力

入力のプログラム中に構文的な誤りがなければ, そのプログラムのプリティプリント (下記参照) を標準出力へ出力せよ. もし, 構文的な誤りがあれば, それを最初に検出した時点で, 入力ファイルでの検出した行の番号 (つまり, 入力ファイルの何行目に誤りがあったか) と誤りの内容 (演算子が必要なところで演算子がない, など) を標準エラー出力へ出力せよ. エラー検出の前にその部分までのプリティプリントを標準入力に出力してもかまわない.

```

プログラム (program) ::= "program" "名前" ";" ブロック "."
ブロック (block) ::= { 変数宣言部 | 副プログラム宣言 } 複合文
変数宣言部 (variable declaration) ::= 
    "var" 変数名の並び ":" 型 ";" { 変数名の並び ":" 型 ";" }
変数名の並び (variable names) ::= 変数名 { "," 変数名 }
変数名 (variable name) ::= "名前"
型 (type) ::= 標準型 | 配列型
標準型 (standard type) ::= "integer" | "boolean" | "char"
配列型 (array type) ::= "array" "[" 符号なし整数 "]" "of" 標準型
副プログラム宣言 (subprogram declaration) ::=
    "procedure" 手続き名 [ 仮引数部 ] ";" [ 変数宣言部 ] 複合文 ";"
手続き名 (procedure name) ::= "名前"
仮引数部 (formal parameters) ::=
    "(" 変数名の並び ":" 型 { ";" 変数名の並び ":" 型 } ")"
複合文 (compound statement) ::= "begin" 文 { ";" 文 } "end"
文 (statement) ::= 代入文 | 分岐文 | 繰り返し文 | 脱出文 | 手続き呼び出し文 |
    戻り文 | 入力文 | 出力文 | 複合文 | 空文
手続き呼び出し文 (call statement) ::= "call" 手続き名 [ "(" 式の並び ")" ]
式の並び (expressions) ::= 式 { "," 式 }
戻り文 (return statement) ::= "return"
代入文 (assignment statement) ::= 左辺部 ":" 式
左辺部 (left part) ::= 変数
変数 (variable) ::= 変数名 [ "[" 式 "]" ]
式 (expression) ::= 単純式 { 関係演算子 単純式 }
    # 関係演算子は左に結合的である。
単純式 (simple expression) ::= [ "+" | "-" ] 項 { 加法演算子 項 }
    # 加法演算子は左に結合的である。
項 (term) ::= 因子 { 乗法演算子 因子 }
    # 乗法演算子は左に結合的である。
因子 (factor) ::= 変数 | 定数 | "(" 式 ")" | "not" 因子 | 標準型 "(" 式 ")"
定数 (constant) ::= 符号なし整数 | "false" | "true" | "文字列"
乗法演算子 (multiplicative operator) ::= "*" | "div" | "and"
加法演算子 (additive operator) ::= "+" | "-" | "or"
関係演算子 (relational operator) ::= "=" | "<>" | "<" | "<=" | ">" | ">="
入力文 (input statement) ::= ("read" | "readln") [ "(" 変数 { "," 変数 } ")" ]
出力文 (output statement) ::=
    ("write" | "writeln") [ "(" 出力指定 { "," 出力指定 } ")" ]
出力指定 (output format) ::= 式 [ ":" 符号なし整数 ] | "文字列"
    # この"文字列"の長さ(文字数)は1以外である。
    # 長さが1の場合は式から生成される定数の一つである"文字列"とする。
空文 (empty statement) ::= ε

```

図 3.1 EBNF による MPPL の構文

## プリティプリント

プログラムのプリティプリントとは、見やすく段付けして印刷されたプログラムリストである。プログラムの見やすさには主観的な要素が多く、どのようなリストがもっとも見やすいかは一概に言えないが、本課題においては、次の条件を満たすものとする。

- 段付の1段は空白4文字とする。
- 行頭を除いて複数の空白は連続しない。行頭を除いてタブは現れない。行末には空白やタブは現れない。すなわち、特に指定されていない限り、行中の字句と字句の間は1つの空白だけがある。
- 前項の指示に関わらず、";", "."の直前には空白を入れない。
- 字句";"の次は必ず改行される。ただし、仮引数中の";"については改行しない。
- 最初の字句"program"は段付けされない。つまり、1カラム目(行頭)から表示される。
- 変数宣言部"var"は行頭から1段段付けされる。また、変数の並びは改行し、"var"からさらに1段段付けする。
- 副プログラム宣言(キーワード)"procedure"で始まる行は行頭から1段段付けされる。
- 副プログラム宣言内の変数宣言部"var"は、"procedure"からさらに1段段付けされる。また、変数の並びは改行し、"var"からさらに1段段付けする。
- 副プログラム宣言内の一一番外側の"begin", "end"は行頭から1段段付けされる。
- 対応する"begin", "end"が段付けされるときは同じ量だけ段付けされる。直前の字句に引き続いて"end"を印刷すると、対応する"begin"より段付け量が多くなるときは改行して同じにする。
- 対応する"begin", "end"の間の文は、その"begin", "end"より少なくとも1段多く段付けされる。
- 分岐文の"else"の前では必ず改行し、その段付けの量は対応する"if"と同じである。
- 分岐文、繰り返し文中の文が複合文でなく、"then", "else", "do"の次の改行するときは、その改行後の文は"if", "while"よりも1段多く段付けされる。
- 分岐文、繰り返し文中の文が複合文のときは、その先頭の"begin"の前で改行し、段付けする。
- 以上に指定のない点については、見やすさに基づいて字句を配置すること。
- 注釈は削除する。その結果、構文解析結果が変わるとときには空白を一つ入れる。

直感的には、プリティプリントとは、注釈及び無駄な空白やタブがなく(字句と字句の間には一つ空白があってよい)、副プログラム宣言部や複合文、ブロック内部はそのすぐ外よりも一段段付けがされているようなプログラムリストである。図3.2と図3.3にプリティプリントの例を挙げる。

```

/* プリティプリントする前のリスト */
program sample11;var n,sum,data:integer;begin
writeln ('input the number of data') ;readln( n );sum:=0 ;while      n>0do
begin  readln(data);sum:=sum+data;n:=n-1end;writeln('Sum of data = ',sum)end.

/* プリティプリンタの出力 */
program sample11;
  var
    n , sum , data : integer;
begin
  writeln ( 'input the number of data' );
  readln ( n );
  sum := 0;
  while n > 0 do
  begin
    readln ( data );
    sum := sum + data;
    n := n - 1
  end;
  writeln ( 'Sum of data = ' , sum );
end.

```

図 3.2 プリティプリントの例 (1)

## 3.3

### LL(1) 構文解析系の作り方 (課題 2 相当)



前期のコンパイラの講義で、説明したとおり、次のような手順で作れます。コンパイラの教科書 [1] も適宜参照のこと。

#### 3.3.1

##### EBNF 記法で書かれた文法を用意する

これは課題 (図 3.1 に) にあります。

#### 3.3.2

##### EBNF の規則の左辺がすべて異なるようにする

もし、同じ左辺を持つ規則があれば、右辺を「|」で結んだ規則に置き換えて、1つにします。たとえば、

$$\begin{aligned} A &::= \alpha \\ A &::= \beta \end{aligned}$$

という規則があれば、

$$A ::= (\alpha)|(\beta)$$

```

/* プリティプリントする前のリスト */
program sample11pp;
procedure kazuyomikomi(n : integer); begin writeln('input the number of data'
    ); readln(n) end;
var sum : integer; procedure wakakidasi; begin writeln('Sum of data = ', sum)
    end; var data : integer;
procedure goukei(n, s : integer); var data : integer; begin
s := 0; while n > 0 do begin readln(data); s := s + data; n := n - 1 end
end; var n : integer; begin call kazuyomikomi(n); call goukei(n * 2, sum);
    call wakakidasi

end.

/* プリティプリンタの出力 */
program sample11pp;
    procedure kazuyomikomi ( n : integer );
begin
    writeln ( 'input the number of data' );
    readln ( n )
end;
var
    sum : integer;
procedure wakakidasi;
begin
    writeln ( 'Sum of data = ' , sum )
end;
var
    data : integer;
procedure goukei ( n , s : integer );
    var
        data : integer;
begin
    s := 0;
    while n > 0 do
        begin
            readln ( data );
            s := s + data;
            n := n - 1
        end
    end;
    var
        n : integer;
begin
    call kazuyomikomi ( n );
    call goukei ( n * 2 , sum );
    call wakakidasi
end.

```

図 3.3 プリティプリントの例 (2)

に置き換え、各非終端記号に対して、それを左辺に持つ規則を1つにします。

図3.1に与えられた構文規則には左辺が同じものはないはずです。

### 3.3.3 構文解析処理関数

各規則に対して(つまり、各非終端記号に対して)、構文解析処理関数を1つずつ作ります。そのとき、

- 終端記号に対しては、それが次の字句であることを確認する
- 非終端記号に対しては、その非終端記号に対応する関数を呼ぶ
- 「… | …」はswitch文かif文に置き換える
- 「{ … }」はwhile文に置き換える
- 「[ … ]」はif文になる

ここでループの中に入るかどうか、ifなどの選択肢のどこへ行くのかは、それぞれの選択肢のFIRST集合を計算して、次の字句がそれに属している方へ処理が進むようにプログラムします。ここで、それぞれのFIRST集合が共通部分を持てば、どちらへ行けばいいのか決定できないので、LL(1)構文解析ができることになります。たとえば、

```
プログラム (program) ::= "program" "名前" ";" ブロック ". "
```

に対しては、

```
int parse_program() {
    if (token != TPROGRAM) return(error("Keyword 'program' is not found"));
    token = scan();
    if (token != TNAME) return(error("Program name is not found"));
    token = scan();
    if (token != TSEMI) return(error("Semicolon is not found"));
    token = scan();
    if (parse_block() == ERROR) return(ERROR);
    if (token != TDOT) return(error("Period is not found at the end of program"));
    token = scan();
    return(NORMAL);
}
```

とすれば良いし、

```
項 ::= 因子 { 乗法演算子 因子 }
```

に対しては、

```
int parse_term() {
    if(parse_factor() == ERROR) return(ERROR);
    while(token == TSTAR || token == TAND || token == TDIV) {
        /* 「乗法演算子 因子」のFIRST集合の要素が、TSTAR, TAND,
        TDIVなので、この条件となる */
```

```

    if(parse_multiple_opr() == ERROR) return(ERROR);
    if(parse_factor() == ERROR) return(ERROR);
}
return(NORMAL);
}

```

とすればよいわけです。このように関数をすべての非終端記号について（生成規則について）作っていくことになります。

もちろん、最初に、

```

#define NORMAL 0
#define ERROR 1
int token;

```

などと宣言しておく必要があります。また、ここで使っている関数 error() はエラーメッセージを標準エラー出力に出力して、戻り値として、ERROR を返す関数であるとしています。たとえば、次のように定めます。

```

int error(char *mes) {
    fprintf(stderr, "Line: %d ERROR: %s.\n", get_linenumber(), mes);
    return ERROR;
}

```

構文解析系を最初に呼び出すためには、スキャナを初期化した後、

```

token = scan();
parse_program();

```

とします。

なお、課題に与えられている EBNF 記法で書かれた構文は、意味をわかりやすくするために冗長な非終端記号が数多く導入されています。たとえば、規則、

```

項 ::= 因子 { 乗法演算子 因子 }

```

には、非終端記号「乗法演算子」がありますが、この非終端記号はここにしか（つまり、1カ所しか）ありません。「乗法演算子」の規則は、

```

乗法演算子 ::= "*" | "div" | "and"

```

ですので、この二つをまとめて、

```

項 ::= 因子 { ("*" | "div" | "and") 因子 }

```

としても、何の問題もありません。このように、右辺の 1 カ所にしか現れない非終端記号は、規則の代入によってなくすることができます。なくしてしまえば、作成する関数の数が減り、関数を超えた情報のやりとりも減るので、一つの規則が大きくなりすぎない限りにおいて、プログラムが簡単になります。以上は構文解析をして、構文エラーのチェックだけをする方法です。

最後に文の処理について簡単にコメントします。文が代入文か○○文か△△文かの判定は、最初のトークンでほとんどわかります。最初のトークンが名前であれば代入文ですし、"if"であれば分岐文です。このように、空文を除いてすべての文の最初のトークンは決まっています。従って、それら以外のトークンが文の先頭にあるときには、そこに空文があるものとすれば良いでしょう。理論的には、FOLLOW集合を計算して、空文がある場合とエラーの場合を分ける必要がありますが、実際には、エラーの場合には後でエラーを検出できますので、これで問題ありません。(ただし、エラーの出るタイミングがずれます。3.4節を参照のこと。)

### 3.3.4 プリティプリンタについて

上記の LL(1) 構文解析系をプリティプリンタにするには、適切な位置に `printf` 文を加えるだけです。スキヤナを呼び出す毎にスキャンした字句(トークン)(トークン番号ではなく、プログラムリスト上の文字列)を `printf` 文で出力すれば、余計な空白やコメントを削除することができます。このとき、次のような字句の配列があれば便利かもしれませんね。

```
char *token_str[] = {
    "", /* 対応なし */
    "", /* TNAME */
    "program",
    "var",
    "array",
    /* 以下、略 */
}
```

もちろん、名前、文字列、符号なし整数のときはその実体を出力する必要があります。ただ、これだけでは字句が隙間なく詰まって出力されてしましますので、その字句が行頭でなければ、その字句を出力する前に空白文字を一つ出力する、行頭であれば、段付け量だけ出力する、適切なタイミングで改行('\'n')を出力することを加える必要があります。

### 3.3.5 テストについて

プリティプリンタは自由度が高いので、どのようにテストを行うかは悩ましいところです。もちろん、段付けが正しくできていることは必要ですが、1.8.5節でも述べたように、メタモーフィックテストを行うこともできます。やり方は、作成したプログラムが出力した整形済みプログラムリストを、再度同じプログラムに対して入力します。これでエラーが出るようでしたら、どこかにバグがあります。また、エラーは出なくても、1回目の出力と2回目の出力が異なっている場合は、どこかにバグがあります。こうしたテストを行うことで、プログラムの信頼性を高めることができます。

課題2に関しては、Dockerで提供しているテストケースは非常に狭いテストとなっているため、PASSEDにならなくても仕様は満たしている可能性はあります。

## 3.4

### テストケースの意味



以下では配布しているテストプログラム(らしきもの)の一部について、意図している検査内容を示します。以下の正しくない想定のプログラムではエラー行の行番号を表示する必要がありますが、どうしても実際にエラーが存在した場所とエラー検知時点での行番号がズれることがあります(以下で「本当は」と書いているものがそうです。)。その場合は気にせずにエラーを検知した時点の行番号を表示してください。

sample02ampl 11 行目 break 文はループの中に有るべき。(構文解析自体は通るが、エラーとする。)

sample021mpl 2 行目(本当は 1 行目) program 文が; で終わっていない。

sample022mpl 2 行目 変数宣言での、忘れ。ただし、エラーとしては: が無い、となる。

sample023mpl 2 行目 変数宣言後の; が多い。エラーとしては複合文のエラーになる。

sample024mpl 3 行目 文字列内の'の表記は''とするべき。エラーとしては、)が無い、となるか。

sample025mpl 6 行目(本当は 5 行目) 「文」の末尾に; は不要。; が必要なのは「複文」の区切り。



gcc とかのコンパイラがなんでエラーそのものの場所を教えてくれないのかを、身を以て体験できるね。

## 3.5

### 課題 3 以降への対応について



型のチェックまでしよう(課題 3 相当)とすれば、各関数の戻り値が ERROR や NORMAL だけではすみません。たとえば、上の関数 `parse_term()` で型のチェックをしようとすれば、被演算子を担当する `parse_factor()` からその被演算子の型を返してもらう必要があります。また、演算子が具体的に何かを知る必要があります。下位の関数から必要な情報をもらって `parse_term()` は型のチェックを行うわけです。コード生成(課題 4 相当)は、以上の関数群にコード生成を行う出力命令を加えていくだけでできるはずです。以上その他に、型チェック、コード生成に共通して必要なものは、記号表です。注意して記号表を設計しましょう。



# 4



## 課題 3 - 意味解析

### 4.1

#### 課題: クロスリファレンサの作成



課題内容関連講義予定日: 2024-11-25(月)

演習期間: 2024-11-25(月) ~ 2024-12-15(日)

レポート・プログラム提出期間: 2024-12-16(月) ~ 2024-12-23(月)

**課題の概略:** プログラミング言語 MPPL で書かれたプログラムを読み込み、コンパイルエラー、すなわち、構文エラーもしくは制約エラー（型の不一致や未定義な変数の出現等）がなければ、クロスリファレンス表を出力し、エラーがあれば、そのエラーの情報（エラーの箇所、内容等）を少なくとも一つ出力するプログラムを作成する。

**キー技術:** データ構造、ポインタ、記号表

### 4.2

#### 課題説明



プログラミング言語 MPPL で書かれたプログラムを読み込み、コンパイルエラー、すなわち、構文エラーもしくは制約エラー（型の不一致や未定義な変数の出現等）がなければ、クロスリファレンス表を標準出力に出力し、エラーがあれば、そのエラーの情報（エラーの箇所、内容等）を少なくとも 1 つ標準エラー出力に出力するプログラムを作成する。例えば、作成するプログラム名を `cr`、MPPL で書かれたプログラムのファイル名を `foo.mppl` とするとき、コマンドラインからのコマンドを

```
$ ./cr foo.mppl
```

とすれば (`foo.mppl` のみが引数)、`foo.mppl` の内容のプログラムのクロスリファレンス表を出力するプログラムを作成する。

#### 4.2.1

##### 入力

MPPL で書かれたプログラムのファイル名、コマンドラインから与える。MPPL のマイクロ構文、マクロ構文はそれぞれ課題 1、2 の通りである。図 4.1、図 4.2 に MPPL の

```

プログラム ::= "program" "名前" ";" ブロック "."
# この"名前"はプログラムの名前なので、ブロック中の名前と同じでもかまわない
ブロック ::= { 変数宣言部 | 副プログラム宣言 } 複合文
# ブロックの複合文中で使われる変数名はブロックの変数宣言部で宣言されていなければならぬ
# ブロックの複合文中で使われる手続き名は副プログラム宣言で宣言されていなければならない
# ブロックの変数宣言部で宣言されている変数名のスコープはプログラム全体である
# すべての名前は、プログラムテキスト中で使用される前に宣言されていなければならない
変数宣言部 ::= "var" 変数名の並び ":" 型 ";" { 変数名の並び ":" 型 ";" }
# 変数名の並びにある変数名の型をその次の":"の次の型とする
変数名の並び ::= 変数名 { "," 変数名 }
# 同じスコープを持つ同じ名前が複数回宣言されていてはならない
変数名 ::= "名前"
型 ::= 標準型 | 配列型
標準型 ::= "integer" | "boolean" | "char"
# それぞれinteger型、boolean型、char型を表す
配列型 ::= "array" "[" "符号なし整数" "]" "of" 標準型
# 標準型を要素型とする配列型を表す。"符号なし整数"は配列の大きさ(要素数)を表す
# 従って、"符号なし整数"の値は1以上でなくてはならない
# 添字の下限は0であり、上限は配列の大きさより1小さな値である(Cと同じ)
副プログラム宣言 ::= "procedure" 手続き名 [ 仮引数部 ] ";" [ 変数宣言部 ] 複合文 ;;
# この手続き名のスコープはプログラム全体である
# この手続き名をこの複合文内で使用することはできない。再帰呼び出しはできない。
# この仮引数部と変数宣言部で宣言されている名前のスコープはこの複合文である。
# すべての名前は、プログラムテキスト中で使用される前に宣言されていなければならない
# 静的スコープルールを採用する。手続き名、変数名を問わず、同じスコープを持つ複数の
# 同じ名前の宣言があった場合には二重定義エラーである
手続き名 ::= "名前"
仮引数部 ::= "(" 変数名の並び ":" 型 { ";" 変数名の並び ":" 型 } ")"
# 仮引数部に現れる型は標準型でなくてはならない
# 変数名の並びにある変数名の型をその次の":"の次の型とする
複合文 ::= "begin" 文 { ";" 文 } "end"
文 ::= 代入文 | 分岐文 | 繰り返し文 | 脱出文 | 手続き呼び出し文 | 戻り文 | 入力文 |
    出力文 | 複合文 | 空文
分岐文 ::= "if" 式 "then" 文 [ "else" 文 ]
# 式の型はboolean型でなくてはならない
繰り返し文 ::= "while" 式 "do" 文
# 式の型はboolean型でなくてはならない
脱出文 ::= "break"
手続き呼び出し文 ::= "call" 手続き名 [ "(" 式の並び ")" ]
# 手続き名は副プログラム宣言で手続き名として宣言されていなくてはならない
式の並び ::= 式 { "," 式 }
# 式の数はその手続き名の宣言の仮引数部の変数の数と一致していなくてはならない
# 式の並びがない場合や、仮引数部がないという場合はそれぞれの数は0とする
# 式と仮引数部の変数は順序で対応し、対応する式と変数は同じ標準型でなくてはならない
戻り文 ::= "return"
代入文 ::= 左辺部 ":"= 式
# 左辺部と式の型は同じ型で標準型でなくてはならない

```

図 4.1 MPPL の構文に対する制約規則 (1/2)

```

左辺部 ::= 変数
# 左辺部の型は変数の型である

変数 ::= 変数名 [ "式" ]
# 式が付いているときの変数名の型はarray型, 式の型はinteger型でなくてはならない
# そのときの変数の型はarray型の要素型である。
# 式が付いていない時の変数の型は変数名の型である

式 ::= 単純式 { 関係演算子 単純式 }
# 一つの単純式だけの式の型はその単純式の型である

関係演算子 ::= "=" | "<>" | "<" | "<=" | ">" | ">="
# 関係演算子の被演算子の型は同じ標準型でなくてはならない。
# 結果の型はboolean型である

単純式 ::= [ "+" | "-" ] 項 { 加法演算子 項 }
# 一つの項だけの単純式の型はその項の型である
# "+"か"-"があるとき左の項の型はinteger型でなくてはならない。
# 結果の型はinteger型である

加法演算子 ::= "+" | "-" | "or"
# "+"と"-"の被演算子の型はinteger型でなくてはならない。
# 結果の型はinteger型である
# "or"の被演算子の型はboolean型でなくてはならない。
# 結果の型はboolean型である

項 ::= 因子 { 乗法演算子 因子 }
# 一つの因子だけの項の型はその因子の型である

乗法演算子 ::= "*" | "div" | "and"
# "*"と"div"の被演算子の型はinteger型でなくてはならない。
# 結果の型はinteger型である
# "and"の被演算子の型はboolean型でなくてはならない。
# 結果の型はboolean型である

因子 ::= 変数 | 定数 | "(" 式 ")" | "not" 因子 | 標準型 "(" 式 ")"
# 変数や定数のとき, 結果の因子の型はそれぞれの型である
# "(" 式 ")" のとき, 結果の因子の型は式の型である。
# "not"の被演算子の型はboolean型でなくてはならない。
# 結果の因子の型はboolean型である
# 標準型 "(" 式 ")" のとき, 結果の因子の型はその標準型である。
# 標準型 "(" 式 ")" の式の型は標準型でなくてはならない

定数 ::= "符号なし整数" | "false" | "true" | "文字列"
# "符号なし整数"のとき, 定数の型はinteger型,
# "false"と"true"のときはboolean型である
# 文字列は長さが1でなくてはならず, そのときの定数の型はchar型である
# 文字列の長さについては, 2.2節の文字列の長さについての記述を参照

入力文 ::= ("read" | "readln") [ "(" 変数 { "," 変数 } ")" ]
# 変数の型はinteger型かchar型でなくてはならない

出力文 ::= ("write" | "writeln") [ "(" 出力指定 { "," 出力指定 } ")" ]
出力指定 ::= 式 [ ":" "符号なし整数" ] | "文字列"
# 式の型は標準型でなくてはならない
# 文字列は長さが0か2以上である。
# 文字列の長さについては, 2.2節の文字列の長さについての記述を参照

空文 ::= ε

```

図 4.2 MPPL の構文に対する制約規則 (2/2)

制約規則を、マクロ構文と共に示す。「#」以降の部分が制約規則またはそれに関連する意味である。ただし、長さが 16 以上の名前は、最初の 16 文字が一致していれば同一の名前としてもよい（もちろん、16 文字より長い先頭部分が一致していれば同一の名前とするようにもよい）。さらに、全文字列が一致している場合にのみ同一の名前としてもよい。どれにしたか、レポートに明記すること）。また、型の一致は構造一致で判定するものとする（型名の宣言がないので、この指定は特に意味はない）。なお、配列型の型が一致するとは、要素型が一致し要素数が同じであることを意味する。

## 4.2.2 出力

入力のプログラム中に構文的な誤りや制約規則違反がなければ、そのプログラムのクロスリファレンス表（4.2.3 節参照）を標準出力へ出力せよ。もし構文的な誤りや制約規則違反があれば、それを最初に検出した時点で、検出した行の番号と、誤りまたは違反の内容（配列の添字の型が integer ではない、など）を標準エラー出力へ出力せよ。

## 4.2.3 クロスリファレンス表

プログラム中に現れた名前の型、その名前が定義された行の番号、参照された（定義以外で現れた）行の番号を表形式で表したもの。定義の場合でも参照の場合でも、その名前が出現している行の番号を表示する。名前でソートされていることが望ましい（名前でソートすることは拡張仕様である）。たとえば、次のプログラム、

```

1 program sample11;
2 var n, sum, data : integer;
3 begin
4   writeln('input the number of data');
5   readln(n);
6   sum := 0;
7   while n > 0 do begin
8     readln(data);
9     sum := sum + data;
10    n := n - 1
11  end;
12  writeln('Sum of data = ', sum)
13 end.
```

のクロスリファレンス表は、

Name	Type	Define	References
data	integer	2	8,9
n	integer	2	5,7,10,10
sum	integer	2	6,9,9,12

が期待されている。仕様をまとめると、

- Name, Type, Define, Reference はそれぞれ、名前、型、定義行、参照行のカラムを示しているヘッダ行である。必ず出力する。

```

1 program sample11pp;
2 procedure kazuyomikomi(n : integer);
3 begin
4   writeln('input the number of data');
5   readln(n)
6 end;
7 var sum : integer;
8 procedure wakakidasi;
9 begin
10  writeln('Sum of data = ', sum)
11 end;
12 var data : integer;
13 procedure goukei(n, s : integer);
14  var data : integer;
15 begin
16  s := 0;
17  while n > 0 do begin
18    readln(data);
19    s := s + data;
20    n := n - 1
21  end
22 end;
23 var n : integer;
24 begin
25  call kazuyomikomi(n);
26  call goukei(n * 2, sum);
27  call wakakidasi
28 end.

```

図 4.3 課題 3 の実行例 (入力)

- 各カラム間には縦棒"|"を入れる。縦棒の前後には任意の数の空白が入っても良い。
- 名前にはプログラム名は含めない。
- 型が配列型の時の型の表示は、「array[100] of integer」のようにせよ。
- 手続き名の型の表示は引数がなければ「procedure」とし、例えば integer 型の引数が 2 つあれば「procedure(integer, integer)」のようにせよ。
- もし、名前が副プログラム宣言内で宣言されている変数名であれば、Name の欄は、

data:proc # 手続きproc の変数 data の意味
----------------------------------

のように区別すること。

- 仮引数名はその副プログラム宣言内で宣言されている変数名と同様に扱ってよい。
- 参照行は行番号をカンマで区切って表示する。

図 4.3, 図 4.4 に課題 3 の入出力の例を示す。

Name	Type	Define	References
data	integer	12	
data:goukei	integer	14	18, 19
goukei	procedure(integer, integer)	13	26
kazuyomikomi	procedure(integer)	2	25
n	integer	23	25, 26
n:goukei	integer	13	17, 20, 20
n:kazuyomikomi	integer	2	5
s:goukei	integer	13	16, 19, 19
sum	integer	7	10, 26
wakakidasi	procedure	8	27

図 4.4 課題 3 の実行例 (出力)

## 4.3

### クロスリファレンサの作り方



課題 2 の結果を利用して作ります。コンパイルエラーとして新たにチェックする点は次の通りです。

- (a) 名前の二重定義（同じスコープを持つ同じ名前が二度以上宣言されている）があれば、エラーとする。
- (b) 名前の未定義（宣言されていない名前が式中などで使われている）があれば、エラーとする。
- (c) 演算子には被演算子の型として許されるものが指定されている。それ以外の型であればエラーとする。
- (d) 同様に if 文, while 文の条件式の型は boolean しか認められていないので、それ以外の型であれば、エラーとする。
- (e) そのほか、型が指定されている部分の型が、それ以外の型であれば、エラーとする。

以上のチェックでエラーがなければ、クロスリファレンス表を出力します。

以上の処理は、以下のような手順で作ることができます。

#### 4.3.1

##### プリティプリントの削除

課題 2 のプリティプリント機能は不要なので、出力部分はすべて削除またはコメントアウトします（コメントアウトを推奨します）。これは最後に行ったほうがよいかもしれません。

#### 4.3.2

##### 記号表の作成

宣言された名前の型を記憶しておく必要があります。つまり、記号表が必要です。そのためのモジュールとして、id-list.c が参考になるかもしれません（ただし、id-list.c は線形リストで記録する記号表なので、動くアルゴリズムではありますが、実用的なコン



課題 4 でアセンブリコード中に対応するコメント行を挿入したい時は、このプリティプリント用の関数を利用できるよ。

パイラに対して推奨できるアルゴリズムではありません). `struct ID` に型情報を記憶するメンバを追加すればいいわけです。また、クロスリファレンス表を出力するために、宣言された行番号や、出現した行番号を記録するメンバも必要です。課題 1 で必要だった出現個数は不要なので、関数 `id_countup` は不要ですが、代わりに、型情報を記憶するための関数、行番号を記憶する関数などが必要かもしれません。それらは関数 `id_countup` を参考にして作成することができます。なお、記号表には、名前の文字列と型情報以外に、種類（特に仮引数名か否か）も記録しておくとよいでしょう。

また、記号表は大域名用と副プログラム宣言内用の最大二つが同時に必要です。後者は副プログラム宣言が終わるごとに空にする（もしくは切り替える）必要があります。

以上を考慮すると、データ構造の宣言は次のようにすると良いでしょう。

```
struct TYPE {
    int ttype;           /* TPINT TPCHAR TPBOOL TPARRAYINT TPARRAYCHAR TPARRAYBOOL
    TPPROC */
    int arraysize;       /* 配列型の場合の配列サイズ */
    struct TYPE *etp;   /* 配列型の場合の要素の型 */
    struct TYPE *paratp; /* 手続き型の場合の、仮引数の型リスト */
};

struct LINE {
    int reflinenum;
    struct LINE *nextlinep;
};

struct ID {
    char *name;
    char *procname;      /* 手続き宣言内で定義されている時の手続き名。それ以外はNULL
    */
    struct TYPE *itp;    /* 型 */
    int ispara;          /* 1:仮引数、0:通常の変数 */
    int deflinenum;      /* 定義行 */
    struct LINE *irefp; /* 参照行のリスト */
    struct ID *nextp;
} *globalidroot, *localidroot; /* Pointers to root of global & local symbol
tables */
```

これを使って、先ほどの例の `sample11pp` を解析すると、データ構造は図 4.5 のようになるでしょう。

### 4.3.3 返値

構文解析用に作った関数で、値を持つもの（単純式とか項などに対応するもの）は、`ERROR` や `NORMAL` だけを返すだけではすみません。その型も返す必要があります。これは、関数値として返してもいいし、引数で返してもいいし、大域変数を経由して返してもかまいません。

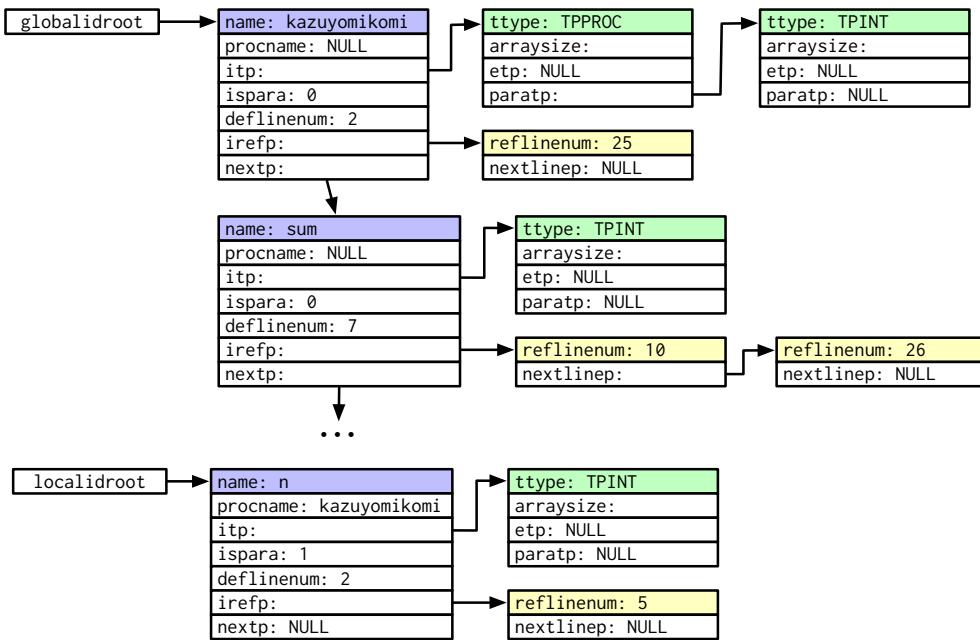


図 4.5 sample11pp.mpl に対する記号表のデータ構造の一部

#### 4.3.4 型のチェック

以上の型情報を利用して、型のチェックルーチンを必要な関数に組み込みます。

#### 4.3.5 出力

最後に、記号表の中身を出力例のように出力するルーチンを書きます。副プログラム宣言用は、その宣言が終わるたびに出力すると楽ですが、それでは、名前の辞書式順に出力できません（副プログラムごとに出力されます）。全体を辞書式順に出力しようと工夫が必要です。

#### 4.3.6 実装上の注意

リスト構造を使うようになるので、ポインタの参照違反で不具合が出ることが増えると思います。次の2点を心がけると、不具合を劇的に減らすことができます。

**ポインタ変数は必ず NULL で初期化する。** ポインタ変数は必ず NULL(または意味のあるポインタの値)で初期化するようにしましょう。

**NULL が入っているポインタ変数に対して->による参照が行われないようにする。**

NULL に対して参照を行うと無条件でエラー終了します。場合分けなどで、初期値 NULL のまま、参照先の値などを得るために->を書くことが多いです。

## 4.4

### テストケースの意味



以下では配布しているテストプログラム（らしきもの）の一部について、意図している検査内容を示します。

sample31p.mpl いろいろな手続き呼び出し.

sample032p.mpl 再帰呼び出しを試みる（エラーにすべき）

sample33p.mpl 31p をさらにややこしく.

sample34.mpl 入出力.

sample35.mpl 記号表の出力. 型変換演算子.



# 5



## 課題 4 - コンパイラ

### 5.1

#### 課題: コンパイラの作成



課題内容関連講義予定日: 2024-12-16(月))

演習期間: 2024-12-16(月)) ~ 2025-01-19(日)

レポート・プログラム提出期間: 2025-01-20(月) ~ 2025-01-27(月)

**課題の概略:** プログラミング言語 MPPL で書かれたプログラムを読み込み、コンパイルエラー、すなわち、構文エラーもしくは制約エラー（型の不一致や未定義な変数の出現等）があれば、そのエラーの情報（エラーの箇所、内容等）を少なくとも一つ出力し、エラーがなければ、オブジェクトプログラムとして、CASL II[2](5.5 節参照) のプログラムを出力するプログラム（すなわちコンパイラ）を作成する。

**キー技術:** コンピューターアーキテクチャ（命令セットレベルアーキテクチャ）、コード生成



進捗状況は?  
そういうば、2022 年から CASL II は情報処理技術者試験に含まれなくなっちゃったんだね。



いよいよコンパイラの作成だ！

### 5.2

#### 課題説明



プログラミング言語 MPPL で書かれたプログラムを読み込み、コンパイルエラー、すなわち、構文エラーもしくは制約エラー（型の不一致や未定義な変数の出現等）があれば、そのエラーの情報（エラーの箇所、内容等）を少なくとも一つ出力するプログラムを作成する。エラーがなければ、オブジェクトプログラムとして、CASL II（別紙参照）のプログラムを出力する。例えば、作成するプログラム名を mpplc、MPPL で書かれたプログラムのファイル名を foompl とするとき、コマンドラインからのコマンドを

```
$ ./mpplc foo.mpl
```

とすれば (foo.mpl のみが引数)、foo.mpl の内容のプログラムをコンパイルした結果 (CASL II のプログラム) を持つファイル foo.csl を生成するプログラムを作成する。

## 5.2.1

### 入力

MPPL で書かれたプログラムのファイル名、コマンドラインから与える。MPPL のマイクロ構文、マクロ構文、制約規則はそれぞれ課題 1, 2, 3 の通りである。MPPL のセマンティクスを、マクロ構文と共に示す。「#」以降の部分がセマンティクスである。

なお、ここに特に定められていないことは処理系依存であるとする。

## 5.2.2

### 出力

入力のプログラム中に構文的な誤りや制約規則違反がなければ、オブジェクトプログラムとして、CASL II（別紙参照）のプログラムを出力せよ。出力ファイル名は入力ファイル名の拡張子を“.csl”へ変更したものとし、カレントディレクトリに出力せよ。出力ファイル名を生成する際には、ファイルが絶対パス、相対パスのいずれで渡されても正しく処理できるようにすべきである。もし構文的な誤りや制約規則違反があれば、それを最初に検出した時点で、誤りまたは違反の内容を標準エラー出力へ出力せよ。誤りを検出しなかった場合は、標準エラー出力には何も出力しないこと。また、本課題では標準出力へ出力するものは無い。

生成した.csl ファイルは、5.6 節で紹介する CASL II アセンブラー・COMET II エミュレータにて実行可能である。

## 5.3

### コード生成の方法



コード生成を作成しようとするとき、まず、実行時環境を決めてから、どのようなコードを生成するかを考えます。

## 5.3.1

### 実行時環境について

#### 変数の割付

MPPL の仕様では、再帰呼び出しがないので、すべての変数を静的に割り付けることができます。また、integer の大きさを 1 語 (2 バイト)、boolean と char の大きさを 1 バイトというように決めることができます。しかし、ここでは、簡単のために、integer, boolean, char とも大きさを 1 語とすることにします。従って、たとえば、

```
var aaa : integer; bbb : boolean;
```

のように変数が宣言されていたとすれば、目的プログラムには、

```
$aaa      DS  1
$bbb      DS  1
```

のように、宣言があればよいことになります。ここで、ラベルは変数名の頭に\$を付けたものとしています。このようにすると、新しくラベル名を考え出す必要がないこと、頭に\$を付けてあるので予約されている文字列と重なることがないことなどの利点があります。

具体的には

```
sample11.mpl
../input01/sample11.mpl
/lpptest/sample11.mpl
とかが全部処理できないとダメだよ。最後の。を探して.csl を附加する方針で作ると間違が少ないよ。
```

進捗は?

```

プログラム ::= "program" "名前" ";" ブロック "."
  # ブロックが実行されるべきプログラムの本体である
ブロック ::= { 変数宣言部 | 副プログラム宣言 } 複合文
  # 複合文が実行されるべき命令である
  # この変数宣言部に宣言されている変数(大域変数)の初期値は不定である
変数宣言部 ::= "var" 変数名の並び ":" 型 ";" { 変数名の並び ":" 型 ";" }
  # 変数の並びにある変数名がその直後の":"の次の型と関連付けられる
変数名の並び ::= 変数名 { "," 変数名 }
変数名 ::= "名前"
型 ::= 標準型 | 配列型
標準型 ::= "integer" | "boolean" | "char"
  # "integer", "boolean", "char"はそれぞれ整数型, 論理型, 文字型を表し,
  # それぞれ, 整数值(16bit), 論理値(true, false), 文字(1バイト文字)を
  # 保持できることを表す
配列型 ::= "array" "[" "符号なし整数" "]" "of" 標準型
副プログラム宣言 ::= "procedure" 手続き名 [ 仮引数部 ] ";" [ 変数宣言部 ] 複合文 ;;
  # この手続き名が呼ばれた(callされた)時は, この複合文が実行される.
  # 引数の渡し方は参照渡しである
  # この変数宣言部に宣言されている変数(局所変数)の初期値は不定である
  # 実行終了後は, この副プログラム(手続き)を呼び出した呼び出し文の次に戻る
  # 再帰呼び出しは許さない
手続き名 ::= "名前"
仮引数部 ::= "(" 変数名の並び ":" 型 { ";" 変数名の並び ":" 型 } ")"
複合文 ::= "begin" 文 { ";" 文 } "end"
  # 文の並びを前から順に実行する
文 ::= 代入文 | 分岐文 | 繰り返し文 | 脱出文 | 手続き呼び出し文 | 戻り文 | 入力文 |
  出力文 |
  複合文 | 空文
分岐文 ::= "if" 式 "then" 文 [ "else" 文 ]
  # 式を評価し, その式の値がtrueであれば左の文を, falseであれば右の文を実行する
  # 式の値がfalseであって, else部がなければ何もしない
繰り返し文 ::= "while" 式 "do" 文
  # [式を評価し, その式の値が true であれば文を実行する] ことを繰り返す
  # 式の値がfalseになればこの文の実行を終える
脱出文 ::= "break"
  # この脱出文を含む繰り返し文のうち, 最も内側の繰り返し文の実行を終える
手続き呼び出し文 ::= "call" 手続き名 [ "(" 式の並び ")" ]
  # 手続き名で指定された手続きを呼び出す
  # 引数の渡し方は参照渡しである.
式の並び ::= 式 { "," 式 }
  # 実引数である式と仮引数である仮引数部の変数は並んでいる順番で対応する
  # 実引数である式が左辺値を持たないときには, 主記憶中に場所を取り, そこに式の値を
  # 置いてその場所の番地を渡す. 手続きの呼び出しが終わるとその場所は再利用してよい.
戻り文 ::= "return"
  # 副プログラム宣言の複合文中で実行されると, その複合文の実行が終了したときと同様に,
  # その手続きの実行を終了する.
  # それ以外の場所, 即ちブロックの複合文内で実行されると, プログラムの実行を終了する

```

図 5.1 MPPL のセマンティクス (1/3)

```

代入文 ::= 左辺部 ":"= 式
  # 式を評価し, その式の値を左辺部が表す変数に代入する
左辺部 ::= 変数
変数 ::= 変数名 [ "[" 式 "]"
  # 式があれば評価する
  # 式の値が配列の添字の範囲外のときは, その旨を表示して実行を停止する
  # 変数名が表す配列の, その式の値を添字の値とする配列要素が変数である
  # 式がなければ, 変数名が表すものが変数である
式 ::= 単純式 { 関係演算子 単純式 }
  # 左の単純式の値にその右隣の単純式の値で関係演算子が表す演算を施す
  # これを左から順に関係演算子と単純式の組がある限り行う
  # その結果の値が, 式の値である
  # 関係演算子がなければ, 一つある単純式の値が式の値である
単純式 ::= [ "+" | "-" ] 項 { 加法演算子 項 }
  # 先頭に"-"があるとき, 先頭の項の値に-1を乗じたものをその項の値とする
  # 左の項の値にその右隣の項の値で加法演算子が表す演算を施す
  # これを左から順に加法演算子と項の組がある限り行う
  # その結果の値が, 単純式の値である
  # 加法演算子がなければ, 一つある項の値が単純式の値である
項 ::= 因子 { 乗法演算子 因子 }
  # 左の因子の値にその右隣の因子の値で乗法演算子が表す演算を施す
  # これを左から順に乗法演算子と因子の組がある限り行う
  # その結果の値が, 項の値である
  # 乗法演算子がなければ, 一つある因子の値が項の値である
因子 ::= 変数 | 定数 | "(" 式 ")" | "not" 因子 | 標準型 "(" 式 ")"
  # 変数については, 変数が保持する値が全体の因子の値である
  # 定数については, 定数自身の値が全体の因子の値である
  # "(" 式 ")"については, 式の値が全体の因子の値である
  # "not" 因子については, この因子の値を論理否定した値が全体の因子の値である
  # 標準型 "(" 式 ")"については, 次の規則に従う
  # 式の型がinteger 型のとき,
  # 標準型が"integer"であれば, 式の値が,
  # 標準型が"boolean"であれば, 式の値が 0 であれば false が, 0 でなければ true が,
  # 標準型が"char"であれば, 式の値の 2 の補数表現の下位 7 ビットを文字コードとして
  # 持つ文字が,
  # 全体の因子の値である
  # 式の型がboolean 型のとき,
  # 標準型が"integer"であれば, 式の値が false であれば 0 を, true であれば 1 が,
  # 標準型が"boolean"であれば, 式の値が,
  # 標準型が"char"であれば, 式の値が false であれば文字コードが 0, true であれば
  # 文字コードが 1 である文字が,
  # 全体の因子の値である
  # 式の型がchar 型のとき,
  # 標準型が"integer"であれば, 式の値の文字コードの整数値が,
  # 標準型が"boolean"であれば, 式の値の文字コードの整数値が 0 であれば false が,
  # 0 でなければ true が,
  # 標準型が"char"であれば, 式の値が,
  # 全体の因子の値である

```

図 5.2 MPPL のセマンティクス (2/3)

```

定数 ::= "符号なし整数" | "false" | "true" | "文字列"
# "符号なし整数"の値はそれを10進数表現として解釈した値が定数の値である
# "false"と"true"のときはそれぞれfalse, trueの論理値が定数の値である
# "文字列"のときは、その文字自体が定数の値である。文字列が「」のときは「」と解釈する
# (注：ここでの文字列は図4.2の制約規則により1文字に限定されている)

乗法演算子 ::= "*" | "div" | "and"
# それぞれ乗算、除算、論理積の演算を表す
# オーバーフロー、ゼロ除算が発生すると、その旨を表示して実行を停止する

加法演算子 ::= "+" | "-" | "or"
# それぞれ加算、減算、論理和の演算を表す
# オーバーフローが発生すると、その旨を表示して実行を停止する

関係演算子 ::= "=" | "<>" | "<" | "<=" | ">" | ">="
# 左の被演算子が右の被演算子に対して、それぞれ、等しい、等しくない、小さい、
# 以下である、大きい、以上である、という命題の真偽を判定した結果の論理値を表す
# ただし、文字型については文字コードの大小で結果の真偽が決定する
# 論理型については、false < trueとする

入力文 ::= ("read" | "readln") [ "(" 変数 { "," 変数 } ")" ]
# 変数の型はinteger型かchar型でなくてはならない
# 標準入力からデータを読み込み、次のように左の変数から順に代入する
# 変数がinteger型のとき、(1)空白、タブ、改行コードがあればそれらをすべて読み飛ばす、
# (2)数字列があればそれを10進数字列とみなして整数に変換して格納する。
# 数字列がなければ0を読み込む
# 変数がchar型のとき、行末でなければ1文字を読み込んで変数に格納する
# 行末であれば、変数には改行コード('\'n')が格納される
# "readln"の場合は最後にその行の改行コードまで読み飛ばす。
# "read"の場合は読み飛ばさない

出力文 ::= ("write" | "writeln") [ "(" 出力指定 { "," 出力指定 } ")" ]
# 出力指定に従って順にデータを標準出力へ出力する
# "writeln"の場合は最後に改行する。"write"の場合はしない

出力指定 ::= 式 [ ":" "符号なし整数" ] | "文字列"
# 式であれば、式の値を標準出力へ出力する。
# 式の型がinteger型の時は10進数表現で、char型の時はその文字を、
# boolean型の時はその値の真偽に従って「true」か「false」を表示する
# "符号なし整数"があればその値の文字数で表示する
# 式の値を表示するのに文字数が余れば、左に空白を補う(右詰めで表示する)
# 式の値を表示するのに文字数が不足すれば、表示される内容は処理系依存である
# "符号なし整数"がなければ、その値を表示できる最小の文字数で表示される
# "文字列"であれば、その文字列自体を標準出力へ表示する
# ただし、その中で「」が偶数個(2n個)連続していればn個の「」とみなす

空文 ::= ε
# 何もしない

```

図5.3 MPPLのセマンティクス(3/3)

す。もちろん、他の付け方でもかまわないです。例えば、ジャンプ命令用のラベルと同じように各変数のラベル（例えば、`L0110` のように）を与える方法もあります。この場合、記号表のその変数のノードにそのラベル番号を覚えておく必要があります。

また、手続き `p` に局所的な変数 `x` の場合は、同じ名前の変数が他の手続きにあつたり、大域変数であつたりするかも知れないで、`$x` とするのは問題があります（アセンブラーで二重定義エラーが出る）。そこで、例えば、`$x%p` のように、手続き名と変数名を合成したラベルを使うといいかもしれません。変数にラベル番号を与える場合には、全体で別々の番号を与えることにすれば、この問題はありません。

配列の場合は、たとえば、

```
var abc : array[100] of integer;
```

ならば、

```
$abc DS 100
```

を目的プログラムに生成すればよいでしょう。

### 実行時スタックについて

上記のように再帰呼び出しがないので、実行時スタックは必要ないのですが、戻り番地を保持するのには、スタックを用いるのが便利です。また、どんな複雑な式でも処理できる必要がある、とすると、式の途中結果を記憶するスタックが必要になります。この場合は、全ての部分式は計算結果をスタックに PUSH すると決めて実現できます。もちろん、この決定は、部分式を計算するたびに主記憶への参照が発生するので効率は良くありません。

部分式の結果の格納場所として、特定のレジスタを利用すると、効率を高めることができます。以降の説明・例示したプログラムでは、部分式の計算結果は必ず GR1 に置くことにします。二項演算を行う際には、はじめの項の結果が GR1 に入っているので、これをスタックに待避して次の項の結果を取得し、最初の項を GR2 に POP することで演算を行うことができます。

もっと効率を上げようすると、レジスタを有効に利用する必要があります。これは、各自で考えてください。コンパイラの講義で講義をしませんでしたが、テキストの「8 目的コード生成」の章 [1] が参考になります。また、テキストのサンプルコンパイラのようにレジスタが無数にあるとして、すべてのレジスタをスタックのように利用してコードを生成してしまうという方法もあります。そうすると COMET II には汎用レジスタが 8 個しかありませんので、9 個目が必要になった時点で困ったことになります。その場合には、そのときにスタックへ待避するようにすれば良いのかもしれません。普通は、8 個もの部分式の値を覚えておく必要のあるような式は滅多に出てきません。手で無理矢理作ろうとしても、とても長い式になってしまいます。その意味では、9 個以上必要な式の場合にはレジスタを待避するので、効率の悪いコードになるとしても実用上は何の問題もないかもしれません。ただ、9 個以上必要な式は出てこない、と仮定するのはバグの元になりますので、止めてください。

### 5.3.2

#### コード生成

構文解析ルーチンにコード生成用の命令を挟み込んでいけばできます。コード生成用の関数は例えば次のようなものを定義できるでしょう。

```
int get_label_num(void) {
    static int labelcounter = 1;
    return labelcounter++;
}

void gen_label(int labelnum) {
    fprintf(caslfp, "L%04d\n", labelnum);
}

void gen_code(char *opc, char *opr) {
    fprintf(caslfp, "\t%s\t%s\n", opc, opr);
}

void gen_code_label(char *opc, char *opr, int label) {
    fprintf(caslfp, "\t%s\t%sL%04d\n", opc, opr, label);
}
```

これらはあくまで一例なので、自分で使いやすいコード生成関数を作ってください。

#### 文のコード生成

たとえば、次のプログラムは `if` 文の構文解析ルーチンですが、コメントで挟み込んで書いているようにコードを生成するプログラムを書けば、`if` 文の目的プログラムが生成できます。

```
/* 分岐文 ::= "if" 式 "then" 文 [ "else" 文 ] */
/* どの"if"に対応するか曖昧な"else"は候補の内で最も近い"if"に対応するとする.*/
int p_ifst(void) {
    int label1, label2;
    if(token != TIF) return(error("Keyword 'if' is not found"));
    token = scan();
    if(p_exp() == ERROR) return ERROR;
    /* p_exp()が条件式のコードを生成してくれていると仮定できるので、 */
    /* 新たなラベルL0001を確保して、          */
    label1 = get_label_num();
    /*      CPA GR1,GR0          */
    gen_code("CPA", "GR1,GR0");
    /*      JZE L0001          */
    gen_code_label("JZE", label1);
    /* を生成する。          */
    /* GR0には常に定数0が入っているものとする。最初に0をセットして */
    /* それ以後値を変えなければよい。          */
    if(token != TTHEN) return(error("Keyword 'then' is not found"));
    token = scan();
    if(p_st() == ERROR) return ERROR;
    /* 同様にp_st()が真の場合の文のコードを生成してくれると仮定できる */
```

```

if(token == TELSE) {
    /* 新たなラベルL0002を確保して, */
    label2 = get_label_num();
    /*      JUMP L0002           */
    gen_code_label("JUMP",label2);
    /* L0001                   */
    gen_label(label1);
    /* を生成する.           */
    token = scan();
    if(p_st() == ERROR) return(ERROR);
    /* 同様にp_st()が偽の場合の文のコードを生成してくれると仮定できる */
    /* L0002                   */
    gen_label(label2);
    /* を生成する(ラベルを立てる). */
} else {
    /* elseがないときは, ラベル          */
    /* L0001                   */
    gen_label(label1);
    /* を生成するだけでよい.           */
}
return NORMAL;
}

```

この結果、生成される `if` 文の目的プログラムは以下のようになります。

```

; 分岐文 "if" 式 "then" 文1 "else" 文2 の場合
(式のコード)
CPA      GR1,   GR0
JZE L0001
(文2のコード)
JUMP L0002
L0001
(文1のコード)
L0002

```

このように文のコード生成は、条件分岐、無条件分岐、ラベル生成がほとんどです。

## 式のコード生成

同様に式の場合も、部分式は部分式の構文解析ルーチンが結果を `GR1` に置くコードを生成してくれると仮定してコード生成ができます。たとえば、次は項に対する構文解析ルーチンですが、コメントのようにコードを生成するプログラムを書けば良いわけです。

```

/* 項 ::= 因子 { 乗法演算子 因子 } */
/* 乗法演算子 ::= "*" | "div" | "and" */
int p_term(void) {
    /* ここに演算子を覚える作業用変数を宣言する。たとえば, */
    int opr;
    if (p_factor() == ERROR) return ERROR;
    /* p_factor()が部分式のコードを生成してくれる(式の結果はGR1にある) */
    /* 部分式の結果をスタックに待避 */
    /* PUSH 0,GR1 */
}

```

```

gen_code("PUSH", "0,GR1");
while(token == TSTAR || token == TDIV || token == TAND) {
    /* ここで、演算子を覚えておく。つまり */
    opr = token;
    /* を加える。 */
    token = scan();
    if (p_factor() == ERROR) return ERROR;
    /* p_factor()が部分式のコードを生成してくれる。（式の結果はGR1 にある） */
    /* ここで被演算子が二つ揃うので、次のようにコード生成する。 */
    /* POP GR2      待避した結果（左の因子）をGR2 に戻す */
    gen_code("POP", "GR2");
    /* もしopr が TSTAR なら、 MULA GR1,GR2 */
    if (opr == TSTAR) {
        gen_code("MULA", "GR1,GR2");
    }
    /* もしopr が TDIV なら、 DIVA GR2,GR1  （計算順に注意） */
    /* LD      GR1,GR2 */
    else if (opr == TDIV) {
        gen_code("DIVA", "GR2,GR1");
        gen_code("LD", "GR1,GR2");
    } else
    /* もしopr が TAND なら、 AND  GR1,GR2 */
    else if (opr == TAND) {
        gen_code("AND", "GR1,GR2");
    }
    /* を生成する。（GR1 に計算結果が入っている） */
}
return NORMAL;
}

```

この例では、型のチェックは省略してありますが、本来は型のチェックを行うコードも入っている必要があります。

### 5.3.3 参照渡しについて

「参照渡し」とは、手続きを呼んだ側から呼ばれた手続きへ実引数を渡す渡し方の一つです。プログラミング言語によっては実引数が変数（l-value を持っている）のときのみ参照渡しが可能ですが、本演習では次のように定義して、実引数が式の場合にでも参照渡しができるようにします。

#### (1) 実引数の渡し方（変数の場合）

実引数は主記憶中に値を置く場所を持っているので、その番地を呼ばれた手続きへ渡す。

```

LAD    GR1,    $n
PUSH   0,GR1
CALL   $kazuyomikomi
; call kazuyomikomi ( n );

```

## (2) 実引数の渡し方 (変数以外の式や定数の場合)

実引数は値のみであり、場所を持っていないので、作業用の領域を確保してその場所にその値を格納し、その場所の番地を呼ばれた手続きへ渡す。再帰呼び出しが許されていないので、その場所は各仮引数毎に一つずつでよい。

```
LAD    GR1,    $n
LD    GR1,  0,GR1
PUSH  0,GR1
LAD    GR1,    2
POP   GR2
MULA  GR1,GR2
JOV   EOF

; ここまでが n * 2 のコード：結果がGR1に入っている
LAD    GR2,    =0 ; GR2 ← リテラル (=0)の番地
ST    GR1,  0,GR2 ; (0+GR2)番地 ← GR1
PUSH  0,GR2 ; リテラルの番地 (GR2)を引数としてプッシュ
; n * 2 の結果をリテラル (=0)の番地に格納し、その番地をスタックにプッシュ
LAD    GR1,    $sum
PUSH  0,GR1
; $sum のアドレスをスタックにプッシュ
CALL   $goukei
; call goukei ( n * 2 , sum );
```

[補足] 上のコードでは、リテラルを一時的な計算結果の格納場所として利用している。この演習で用いる CASL II 実装では、すべての出現したリテラルを独立した領域として確保し、プログラム末尾に自動的に追加する。

リテラルを用いたプログラム

```
LAD    GR2,    =0
```

は、

```
LAD    GR2,    L9999
...
L9999  DC  0
```

とほぼ等価である。本来、下のように書かねばならない CASL II のコードを、上のようにラベルを省略した形で記述できる。

## (3) 仮引数の参照法

呼ばれた手続き内で仮引数が参照されたとき、その参照が値を使用する場合でも代入される場合でも、渡された番地が示す場所を参照する。以下に参考コードを載せる。

```
$$n%kazuyomikomi    DC  0      ; 仮引数
$data%kazuyomikomi  DC  0      ; 局所変数
$kazuyomikomi
    POP   GR2          ; CALL 命令がスタックに積んだ返り番地
    POP   GR1          ; 仮引数の値 (実引数の番地)
```

ただし、ラベルを持たないので後から同じ場所を参照することはできないよ。



```

ST      GR1, $$n%kazuyomikomi
PUSH    0,GR2          ; 返り番地をスタックに戻す
...
; GR1 に仮引数の値を代入する
LD      GR1, $$n%kazuyomikomi ; GR1 には実引数の番地が入る
LD      GR1,0,GR1           ; GR1 番地の値を GR1 に格納
...
; GR1 に局所変数の値を代入する
LAD    GR1, $data%kazuyomikomi ; GR1 には局所変数の番地が入る
LD      GR1,0,GR1           ; GR1 番地の値を GR1 に格納
...
RET

```

上の例のように、仮引数 (\$\$で始まる) と局所変数 (\$で始まる) の識別子生成ルールを変えると後から見たときに判別がつきやすい。(コーディングの手間は増える。) 以降の章で示す出力例は、仮引数と局所変数を区別した識別子を利用している。

### 5.3.4 【コラム】 DC, DS 命令について

CASL II の DC, DS 疑似命令はメモリ上に領域を確保するだけなので、命令の流れから独立したところに確保すべきです。例えば、以下のような書き方はアセンブラー的にはダメです。

```

PRG000 START
LD      GR1,DATA
DATA   DC    'ABC'
ST      GR1,RESULT
RET
RESULT DS    1
END

```

```

CASL LISTING
2 0000 1010      LD  GR1,DATA
2      0002
3 0002 0041  DATA   DC    'ABC'
3      0042
3      0043
3      0000
4 0006 1110      ST  GR1,RESULT
4      0009
5 0008 8100      RET
6 0009 0000  RESULT DS  1

```

ただし、現在の CASL II, COMET II 実装では、こう書いても DC のところがスキップされているかのように見えます。この理由は、機械語表現にあります。LD 命令の次に来る命令語らしきもの 0041 の上位 2 バイトが 00 であれば、NOP 命令と判断して(5.5 節を参照)先に進む動作をしているからです。確保した領域が数値であれば 0~255、文字列であればほぼ全ての文字はこの上位 2 バイトが 00 になりますので、COMET II は NOP だな

と思ってスキップします。そのため、多くの場合、上で示したような実装でもなんとなく動いているように見えてしまうのです。（あまり良いことではありません）

もちろん、DC 命令で確保した数値が-1 (#FFFF) などであれば、NOP とは見なされず、しっかりと想定外の動作をします。以下のコードでは DATA2 は 4368 という値の DC ですが、直前の ST 命令と全く同じ機械語コード#1110 になっています。

#### CASL LISTING

```
2 0000 1010      LD  GR1,DATA
2      0002
3 0002 0041  DATA    DC  'ABC'
3      0042
3      0043
3      0000
4 0006 1110      ST  GR1,RESULT
4      000a
5 0008 1110  DATA2   DC  4368
6 0009 8100      RET
7 000a 0000  RESULT  DS  1
```

これを COMET II で読み込むと、ST GR1, #8100 というコードに見えてしまい、RET 命令が無いので暴走を始めます。

```
comet2> di 0
#0000  LD  GR1,    #0002
#0002  NOP
#0003  NOP
#0004  NOP
#0005  NOP
#0006  ST  GR1,    #000a
#0008  ST  GR1,    #8100
#000a  NOP
```

一時的にしか使わない数値や文字列の場合、リテラルを利用すると上のような問題は発生しなくなります。リテラルの領域は CASL II アセンブラーがアセンブル時に安全なところに確保して参照する実装になっているからです。

## 5.4

### テストケースの意味



以下では配布しているテストプログラムについて、意図している検査内容を示します。

sample41.mpl 配列の添字が範囲外の場合。コンパイラでは CASL II のコードを問題無く生成するが、CASL II の実行時に添字の判定を行って実行時エラーを出す。



## 5.4.1

### 出力例

コメント行には、対応する MPPL プログラムの行を表示しています。（コメント行の出力は課題として要求されていません）ただし、あるコメント行はその行の直前までのアセンブリコードの内容を表していることに注意してください。

また、

```
; -----  
; Utility functions  
; -----
```

以降は、出題時に配布するユーティリティサブルーチン群である。END 命令を出力する直前にこのコードを出力することで利用できる。



課題 2 のプリティプリントのコードを流用すると、割と簡単にアセンブリコードに対応する MPPL コードのコメントが挿入できるよ。プリティプリントの内容を文字バッファに貯めていって、プリティプリントの改行のタイミングでコメントを生成すれば良いよ。

sample11pp.mppl

```
program sample11pp;  
procedure kazuyomikomi(n : integer);  
begin  
    writeln('input the number of data');  
    readln(n)  
end;  
var sum : integer;  
procedure wakakidasi;  
begin  
    writeln('Sum of data = ', sum)  
end;  
var data : integer;  
procedure goukei(n, s : integer);  
    var data : integer;  
begin  
    s := 0;  
    while n > 0 do begin  
        readln(data);  
        s := s + data;  
        n := n - 1  
    end  
end;  
var n : integer;  
begin  
    call kazuyomikomi(n);  
    call goukei(n * 2, sum);  
    call wakakidasi  
end.
```

sample11pp.cs1

```
1 %%sample11pp      START    L0001  
2 ;  program sample11pp;  
3 $$n%kazuyomikomi
```

```

4      DC      0
5 ;  procedure kazuyomikomi ( n : integer );
6 $kazuyomikomi
7      POP    GR2
8      POP    GR1
9      ST     GR1,$$n%kazuyomikomi
10     PUSH   0,GR2
11 ;  begin
12     LAD    GR1,'input the number of data'
13     LAD    GR2,0
14     CALL   WRITESTR
15     CALL   WRITELINE
16 ;  writeln ( 'input the number of data' );
17     LD     GR1,$$n%kazuyomikomi
18     CALL   READINT
19     CALL   READLINE
20 ;  readln ( n )
21     RET
22 ;  end;
23 ;  var
24 $sum   DC      0
25 ;  sum : integer;
26 ;  procedure wakakidasi;
27 $wakakidasi
28 ;  begin
29     LAD    GR1,'Sum of data = '
30     LAD    GR2,0
31     CALL   WRITESTR
32     LD     GR1,$sum
33     LAD    GR2,0
34     CALL   WRITEINT
35     CALL   WRITELINE
36 ;  writeln ( 'Sum of data = ' , sum )
37     RET
38 ;  end;
39 ;  var
40 $data  DC      0
41 ;  data : integer;
42 $$s%goukei
43      DC      0
44 $$n%goukei
45      DC      0
46 ;  procedure goukei ( n , s : integer );
47 ;  var
48 $data%goukei
49      DC      0
50 ;  data : integer;
51 $goukei
52      POP    GR2
53      POP    GR1
54      ST     GR1,$$s%goukei
55      POP    GR1

```

```

56      ST      GR1,$$n%goukei
57      PUSH    0,GR2
58; begin
59      LD      GR1,$$s%goukei
60      PUSH    0,GR1
61      LAD     GR1,0
62      POP     GR2
63      ST      GR1,0,GR2
64; s := 0;
65 L0002
66      LD      GR1,$$n%goukei
67      LD      GR1,0,GR1
68      PUSH    0,GR1
69      LAD     GR1,0
70      POP     GR2
71      CPA     GR2,GR1
72      JPL     L0004
73      LAD     GR1,0
74      JUMP   L0005
75 L0004
76      LAD     GR1,1
77 L0005
78      CPA     GR1,GR0
79      JZE    L0003
80; while n > 0 do
81; begin
82      LAD     GR1,$data%goukei
83      CALL   READINT
84      CALL   READLINE
85; readln ( data );
86      LD      GR1,$$s%goukei
87      PUSH    0,GR1
88      LD      GR1,$$s%goukei
89      LD      GR1,0,GR1
90      PUSH    0,GR1
91      LD      GR1,$data%goukei
92      POP     GR2
93      ADDA   GR1,GR2
94      JOV    EOFV
95      POP     GR2
96      ST      GR1,0,GR2
97; s := s + data;
98      LD      GR1,$$n%goukei
99      PUSH    0,GR1
100     LD     GR1,$$n%goukei
101     LD     GR1,0,GR1
102     PUSH    0,GR1
103     LAD     GR1,1
104     POP     GR2
105     SUBA   GR2,GR1
106     JOV    EOFV
107     LD      GR1,GR2

```

```

108      POP    GR2
109      ST     GR1,0,GR2
110 ;  n := n - 1
111 ;  end
112      JUMP   L0002
113 L0003
114      RET
115 ;  end;
116 ;  var
117 $n    DC    0
118 ;  n : integer;
119 L0001
120      LAD    GR0,0
121 ;  begin
122      LAD    GR1,$n
123      PUSH   0,GR1
124      CALL   $kazuyomikomi
125 ;  call kazuyomikomi ( n );
126      LAD    GR1,$n
127      LD     GR1,0,GR1
128      PUSH   0,GR1
129      LAD    GR1,2
130      POP    GR2
131      MULA  GR1,GR2
132      JOV   EOFV
133      LAD    GR2,=0
134      ST     GR1,0,GR2
135      PUSH   0,GR2
136      LAD    GR1,$sum
137      PUSH   0,GR1
138      CALL   $goukei
139 ;  call goukei ( n * 2 , sum );
140      CALL   $wakakidasi
141 ;  call wakakidasi
142      CALL   FLUSH
143      RET
144 ;  end.
145 ; -----
146 ; Utility functions
147 ; -----
148 ; オーバーフロー発生時にプログラム終了
149 EOFV   CALL   WRITELINE
150      LAD    GR1,EOFV1
151      LD     GR2,GR0
152      CALL   WRITESTR
153      CALL   WRITELINE
154      SVC    1      ; overflow error stop
155 EOFV1  DC    '***** Run-Time Error : Overflow *****'
156 ; 0除算発生時にプログラム終了
157 E0DIV   JNZ   EOFV
158      CALL   WRITELINE
159      LAD    GR1,E0DIV1

```

```

160      LD      GR2,GR0
161      CALL    WRITESTR
162      CALL    WRITELINE
163      SVC    2      ; 0-divide error stop
164 E0DIV1 DC     '***** Run-Time Error : Zero-Divide *****'
165 ; 配列の添字あふれ発生時にプログラム終了
166 EROV   CALL    WRITELINE
167      LAD    GR1,EROV1
168      LD     GR2,GR0
169      CALL    WRITESTR
170      CALL    WRITELINE
171      SVC    3      ; range-over error stop
172 EROV1 DC     '***** Run-Time Error : Range-Over in Array Index *****'
173 ; ##### 表示サブルーチン #####
174 ; # WRITE(CHAR,STR,INT,BOOL)は出力バッファに
175 ; # 文字を格納していくだけであり、出力は
176 ; # FLUSH または WRITELINE を CALL 時に初めて行われる
177 ; WRITECHAR
178 ; GR1 の値(文字)をGR2 のけた数で出力する
179 ; GR2 が 0 なら必要最小限の桁数で出力する
180 WRITECHAR
181      RPUSH
182      LD     GR6,SPACE
183      LD     GR7, OBUFSIZE
184 WC1   SUBA  GR2,ONE      ; while(--c > 0) {
185      JZE   WC2
186      JMI   WC2
187      ST    GR6,OBUF,GR7      ; *p++ = ' ';
188      CALL  BOVFCHECK
189      JUMP  WC1      ; }
190 WC2   ST    GR1,OBUF,GR7      ; *p++ = GR1;
191      CALL  BOVFCHECK
192      ST    GR7, OBUFSIZE
193      RPOP
194      RET
195 ; WRITESTR
196 ; GR1 が指す文字列を GR2 のけた数で出力する
197 ; GR2 が 0 なら必要最小限の桁数で出力する
198 WRITESTR
199      RPUSH
200      LD     GR6,GR1      ; p = GR1;
201 WS1   LD     GR4,0,GR6      ; while(*p != 0) {
202      JZE   WS2
203      ADDA GR6,ONE      ; p++;
204      SUBA GR2,ONE      ; c--;
205      JUMP  WS1      ; }
206 WS2   LD     GR7,OBUFSIZE      ; q = OBUFSIZE;
207      LD     GR5,SPACE
208 WS3   SUBA GR2,ONE      ; while(--c >= 0) {
209      JMI   WS4
210      ST    GR5,OBUF,GR7      ; *q++ = ' ';
211      CALL  BOVFCHECK

```

```

212      JUMP   WS3          ; }
213 WS4    LD     GR4,0,GR1       ; while(*GR1 != 0) {
214      JZE   WS5
215      ST    GR4,OBUF,GR7    ; *q++ = *GR1++;
216      ADDA  GR1,ONE
217      CALL  BOVFCHECK
218      JUMP  WS4          ; }
219 WS5    ST    GR7,OBUFSIZE  ; OBUFSIZE = q;
220      RPOP
221      RET
222 BOVFCHECK
223      ADDA  GR7,ONE
224      CPA   GR7,BOVFLEVEL
225      JMI   BOVF1
226      CALL  WRITELINE
227      LD    GR7,OBUFSIZE
228 BOVF1  RET
229 BOVFLEVEL
230      DC    256
231 ; WRITEINT
232 ; GR1 の値(整数)をGR2 のけた数で出力する
233 ; GR2 が 0 なら必要最小限の桁数で出力する
234 WRITEINT
235      RPUSH
236      LD    GR7,GR0          ; flag = 0;
237      CPA  GR1,GR0          ; if(GR1>=0) goto WI1;
238      JPL  WI1
239      JZE  WI1
240      LD    GR4,GR0          ; GR1= - GR1;
241      SUBA GR4,GR1
242      CPA   GR4,GR1
243      JZE  WI6
244      LD    GR1,GR4
245      LD    GR7,ONE          ; flag = 1;
246 WI1   LD    GR6,SIX         ; p = INTBUF+6;
247      ST    GR0,INTBUF,GR6  ; *p = 0;
248      SUBA GR6,ONE          ; p--;
249      CPA   GR1,GR0          ; if(GR1 == 0)
250      JNZ  WI2
251      LD    GR4,ZERO         ; *p = '0';
252      ST    GR4,INTBUF,GR6
253      JUMP WI5          ; }
254      ; else {
255 WI2   CPA   GR1,GR0          ; while(GR1 != 0) {
256      JZE  WI3
257      LD    GR5,GR1          ; GR5 = GR1 - (GR1 / 10) * 10;
258      DIVA GR1,TEN           ; GR1 /= 10;
259      LD    GR4,GR1
260      MULA GR4,TEN
261      SUBA GR5,GR4
262      ADDA GR5,ZERO          ; GR5 += '0';
263      ST    GR5,INTBUF,GR6  ; *p = GR5;

```

```

264      SUBA    GR6,ONE           ;  p--;
265      JUMP    WI2              ;  }
266 WI3      CPA    GR7,GR0           ;  if(flag != 0) {
267      JZE    WI4              ;
268      LD     GR4,MINUS          ;  *p = '-';
269      ST     GR4,INTBUF,GR6       ;
270      JUMP    WI5              ;  }
271 WI4      ADDA   GR6,ONE           ;  else p++;
272      ;  }
273 WI5      LAD    GR1,INTBUF,GR6       ;  GR1 = p;
274      CALL   WRITESTR          ;  WRITESTR();
275      RPOP
276      RET
277 WI6      LAD    GR1,MMINT          ;
278      CALL   WRITESTR          ;  WRITESTR();
279      RPOP
280      RET
281 MMINT   DC    '-32768'
282 ;  WRITEBOOL
283 ;  GR1 の値(真理値)が 0なら'FALSE'を
284 ;  0以外なら'TRUE'をGR2 のけた数で出力する
285 ;  GR2 が 0 なら必要最小限の桁数で出力する
286 WRITEBOOL
287      RPUSH
288      CPA    GR1,GR0           ;  if(GR1 != 0)
289      JZE    WB1              ;
290      LAD    GR1,WBTRUE          ;  GR1 = TRUE;
291      JUMP    WB2              ;
292      ;  else
293 WB1      LAD    GR1,WBFALSE         ;  GR1 = FALSE;
294 WB2      CALL   WRITESTR          ;  WRITESTR();
295      RPOP
296      RET
297 WBTRUE   DC    'TRUE'
298 WBFALSE  DC    'FALSE'
299 ;  WRITELINE
300 ;  改行を出力する
301 WRITELINE
302      RPUSH
303      LD     GR7,OBUFSIZE
304      LD     GR6,NEWLINE
305      ST     GR6,OBUF,GR7
306      ADDA   GR7,ONE
307      ST     GR7,OBUFSIZE
308      OUT    OBUF,OBUFSIZE
309      ST     GR0,OBUFSIZE
310      RPOP
311      RET
312 ;  FLUSH
313 ;  出力バッファをすべて表示する
314 FLUSH   RPUSH
315      LD     GR7,OBUFSIZE

```

```

316      JZE    FL1
317      CALL   WRITELINE
318 FL1    RPOP
319      RET
320 ; ##### 入力サブルーチン #####
321 ; READCHAR
322 ; GR1 が指す番地に文字一つを読み込む
323 READCHAR
324     RPUSH
325     LD     GR5,RPBBUF      ; if(RPBBUF != 0) {
326     JZE   RC0
327     ST     GR5,0,GR1       ; *GR1 = RPBBUF;
328     ST     GR0,RPBBUF      ; RPBBUF = 0
329     JUMP  RC3            ; return; }
330 RC0   LD     GR7,INP        ; inp = INP;
331     LD     GR6,IBUFSIZE    ; if(IBUFSIZE == 0) {
332     JNZ   RC1
333     IN     IBUF,IBUFSIZE   ; IN();
334     LD     GR7,GR0          ; inp = 0;
335             ; }
336 RC1   CPA   GR7,IBUFSIZE   ; if(inp == IBUFSIZE) {
337     JNZ   RC2
338     LD     GR5,NEWLINE     ; *GR1 = '\n';
339     ST     GR5,0,GR1       ;
340     ST     GR0,IBUFSIZE    ; IBUFSIZE = INP = 0;
341     ST     GR0,INP          ;
342     JUMP  RC3            ; }
343             ; else {
344 RC2   LD     GR5,IBUF,GR7  ; *GR1 = *inp++;
345     ADDA  GR7,ONE
346     ST     GR5,0,GR1       ;
347     ST     GR7,INP          ; INP = inp;
348             ; }
349 RC3   RPOP
350     RET
351 ; READINT
352 ; GR1 が指す番地に整数値一つを読み込む
353 READINT RPUSH
354             ; do {
355 RI1   CALL   READCHAR      ; ch = READCHAR();
356     LD     GR7,0,GR1       ;
357     CPA   GR7,SPACE        ; } while(ch==' ' || ch=='\t' || ch=='\n');
358     JZE   RI1
359     CPA   GR7,TAB          ;
360     JZE   RI1
361     CPA   GR7,NEWLINE      ;
362     JZE   RI1
363     LD     GR5,ONE         ; flag = 1
364     CPA   GR7,MINUS        ; if(ch == '-') {
365     JNZ   RI4
366     LD     GR5,GR0          ; flag = 0;
367     CALL  READCHAR        ; ch = READCHAR();

```

```

368      LD      GR7,0,GR1      ; }
369 RI4     LD      GR6,GR0      ; v = 0;
370 RI2     CPA    GR7,ZERO     ; while('0' <= ch && ch <= '9') {
371       JMI    RI3
372       CPA    GR7,NINE
373       JPL    RI3
374       MULA   GR6,TEN       ; v = v*10+ch-'0';
375       ADDA   GR6,GR7
376       SUBA   GR6,ZERO
377       CALL    READCHAR     ; ch = READSCHAR();
378       LD      GR7,0,GR1
379       JUMP   RI2      ; }
380 RI3     ST      GR7,RPBBUF   ; ReadPushBack();
381       ST      GR6,0,GR1      ; *GR1 = v;
382       CPA    GR5,GR0      ; if(flag == 0) {
383       JNZ    RI5
384       SUBA   GR5,GR6      ; *GR1 = -v;
385       ST      GR5,0,GR1
386           ; }
387 RI5     RPOP
388       RET
389 ; READLINE
390 ; 入力を改行コードまで(改行コードも含む)読み飛ばす
391 READLINE
392       ST      GR0,IBUFSIZE
393       ST      GR0,INP
394       ST      GR0,RPBBUF
395       RET
396 ONE    DC      1
397 SIX    DC      6
398 TEN    DC      10
399 SPACE  DC      #0020      ; '
400 MINUS  DC      #002D      ; '-'
401 TAB    DC      #0009      ; '\t'
402 ZERO   DC      #0030      ; '0'
403 NINE   DC      #0039      ; '9'
404 NEWLINE DC      #000A      ; '\n'
405 INTBUF DS      8
406 OBUFSIZE
407       DC      0
408 IBUFSIZE
409       DC      0
410 INP    DC      0
411 OBUF   DS      257
412 IBUF   DS      257
413 RPBBUF DC      0
414       END

```

### sample11pp.cs1 の実行結果

```
comet2> run
```

```
OUT> input the number of data
IN> 3
IN> 1
IN> 2
IN> 3
IN> 4
IN> 5
IN> 6
OUT> Sum of data = 21
Program finished (RET)
```

### sample16.mpl

```
program sample16; /* prime numbers */
var furui : array[2000] of boolean;
    i, j : integer;
begin
    i := 2;
    while i < 2000 do begin
        furui[i] := true;
        i := i + 1
    end;
    furui[0] := false;
    furui[1] := false;
    i := 2;
    while i < 2000 do begin
        if furui[i] then begin
            writeln(i, ' is a prime number');
            j := i;
            if i < 1024 then
                while j < 2000 do begin
                    furui[j] := false;
                    j := j + i
                end
            end;
            i := i + 1
        end
    end
end.
```

### sample16.csl

```
1 %%sample16      START  L0001
2 ;  program sample16;
3 ;  var
4 $furui  DS      2000
5 ;  furui : array [ 2000 ] of boolean;
6 $j      DC      0
7 $i      DC      0
8 ;  i , j : integer;
9 L0001
10      LAD      GR0,0
11 ;  begin
12      LAD      GR1,$i
13      PUSH     0,GR1
14      LAD      GR1,2
15      POP      GR2
16      ST       GR1,0,GR2
17 ;  i := 2;
18 L0002
19      LD       GR1,$i
20      PUSH     0,GR1
21      LAD      GR1,2000
```

```

22      POP    GR2
23      CPA    GR2,GR1
24      JMI    L0004
25      LAD    GR1,0
26      JUMP   L0005
27 L0004
28      LAD    GR1,1
29 L0005
30      CPA    GR1,GR0
31      JZE    L0003
32 ;  while i < 2000 do
33 ;  begin
34      LD     GR1,$i
35      CPA    GR1,GR0
36      JMI    EROV
37      LAD    GR2,1999
38      CPA    GR1,GR2
39      JPL    EROV
40      LAD    GR1,$furui,GR1
41      PUSH   0,GR1
42      LAD    GR1,1
43      POP    GR2
44      ST     GR1,0,GR2
45 ;  furui [ i ] := true;
46      LAD    GR1,$i
47      PUSH   0,GR1
48      LD     GR1,$i
49      PUSH   0,GR1
50      LAD    GR1,1
51      POP    GR2
52      ADDA   GR1,GR2
53      JOV    EOFV
54      POP    GR2
55      ST     GR1,0,GR2
56 ;  i := i + 1
57      JUMP   L0002
58 L0003
59 ;  end;
60      LAD    GR1,0
61      CPA    GR1,GR0
62      JMI    EROV
63      LAD    GR2,1999
64      CPA    GR1,GR2
65      JPL    EROV
66      LAD    GR1,$furui,GR1
67      PUSH   0,GR1
68      LAD    GR1,0
69      POP    GR2
70      ST     GR1,0,GR2
71 ;  furui [ 0 ] := false;
72      LAD    GR1,1
73      CPA    GR1,GR0

```

```

74      JMI    EROV
75      LAD    GR2,1999
76      CPA    GR1,GR2
77      JPL    EROV
78      LAD    GR1,$furui,GR1
79      PUSH   0,GR1
80      LAD    GR1,0
81      POP    GR2
82      ST     GR1,0,GR2
83 ; furui [ 1 ] := false;
84      LAD    GR1,$i
85      PUSH   0,GR1
86      LAD    GR1,2
87      POP    GR2
88      ST     GR1,0,GR2
89 ; i := 2;
90 L0006
91      LD     GR1,$i
92      PUSH   0,GR1
93      LAD    GR1,2000
94      POP    GR2
95      CPA    GR2,GR1
96      JMI    L0008
97      LAD    GR1,0
98      JUMP   L0009
99 L0008
100     LAD   GR1,1
101 L0009
102     CPA   GR1,GR0
103     JZE   L0007
104 ; while i < 2000 do
105 ; begin
106     LD    GR1,$i
107     CPA   GR1,GR0
108     JMI   EROV
109     LAD   GR2,1999
110     CPA   GR1,GR2
111     JPL   EROV
112     LD    GR1,$furui,GR1
113     CPA   GR1,GR0
114     JZE   L0010
115 ; if furui [ i ] then
116 ; begin
117     LD    GR1,$i
118     LAD   GR2,0
119     CALL  WRITEINT
120     LAD   GR1,' is a prime number'
121     LAD   GR2,0
122     CALL  WRITESTR
123     CALL  WRITELINE
124 ; writeln ( i , ' is a prime number' );
125     LAD   GR1,$j

```

```

126      PUSH    0,GR1
127      LD      GR1,$i
128      POP    GR2
129      ST      GR1,0,GR2
130 ;   j := i;
131      LD      GR1,$i
132      PUSH    0,GR1
133      LAD     GR1,1024
134      POP    GR2
135      CPA     GR2,GR1
136      JMI     L0014
137      LAD     GR1,0
138      JUMP   L0015
139 L0014
140      LAD     GR1,1
141 L0015
142      CPA     GR1,GR0
143      JZE     L0012
144 ;   if i < 1024 then
145 L0016
146      LD      GR1,$j
147      PUSH    0,GR1
148      LAD     GR1,2000
149      POP    GR2
150      CPA     GR2,GR1
151      JMI     L0018
152      LAD     GR1,0
153      JUMP   L0019
154 L0018
155      LAD     GR1,1
156 L0019
157      CPA     GR1,GR0
158      JZE     L0017
159 ;   while j < 2000 do
160 ;   begin
161      LD      GR1,$j
162      CPA     GR1,GR0
163      JMI     EROV
164      LAD     GR2,1999
165      CPA     GR1,GR2
166      JPL     EROV
167      LAD     GR1,$furui,GR1
168      PUSH    0,GR1
169      LAD     GR1,0
170      POP    GR2
171      ST      GR1,0,GR2
172 ;   furui [ j ] := false;
173      LAD     GR1,$j
174      PUSH    0,GR1
175      LD      GR1,$j
176      PUSH    0,GR1
177      LD      GR1,$i

```

```

178      POP    GR2
179      ADDA   GR1,GR2
180      JOV    EOF
181      POP    GR2
182      ST     GR1,0,GR2
183 ;   j := j + i
184 ;   end
185      JUMP   L0016
186 L0017
187 L0012
188 L0010
189 ;   end;
190      LAD    GR1,$i
191      PUSH   0,GR1
192      LD     GR1,$i
193      PUSH   0,GR1
194      LAD    GR1,1
195      POP    GR2
196      ADDA   GR1,GR2
197      JOV    EOF
198      POP    GR2
199      ST     GR1,0,GR2
200 ;   i := i + 1
201 ;   end
202      JUMP   L0006
203 L0007
204      CALL   FLUSH
205      RET
206 ;   end.
207 ; -----
208 ; Utility functions
209 ; -----
210 ; (sample11pp と同一のため省略)
211      END

```

### sample16.cs1 の実行結果

```

comet2> run
OUT> 2 is a prime number
OUT> 3 is a prime number
OUT> 5 is a prime number
OUT> 7 is a prime number
OUT> 11 is a prime number
(..中略..)
OUT> 1997 is a prime number
OUT> 1999 is a prime number
Program finished (RET)

```

### sample35.mpl

```
program typeconv;
var i : integer; b : boolean; c : char;
begin
  i := integer(false);
  while i <= integer(true) do begin
    writeln( boolean(i), ' : ', i);
    i := i + 1;
  end;
  writeln;
  i := integer(' ');
  while i < 127 do begin
    if i div 16 * 16 = i then writeln;
    write(char(i), ' ');
    i := i + 1;
  end;
  writeln
end.
```

### sample35.cs1

```
1 %%typeconv      START  L0001
2 ;  program typeconv;
3 ;  var
4 $i      DC      0
5 ;  i : integer;
6 $b      DC      0
7 ;  b : boolean;
8 $c      DC      0
9 ;  c : char;
10 L0001
11       LAD      GR0,0
12 ;  begin
13       LAD      GR1,$i
14       PUSH     0,GR1
15       LAD      GR1,0
16       POP      GR2
17       ST       GR1,0,GR2
18 ;  i := integer ( false );
19 L0002
20       LD       GR1,$i
21       PUSH     0,GR1
22       LAD      GR1,1
23       POP      GR2
24       CPA      GR2,GR1
25       JMI      L0004
26       JZE      L0004
27       LAD      GR1,0
28       JUMP    L0005
29 L0004
```

```

30      LAD    GR1,1
31 L0005      CPA    GR1,GR0
32      JZE    L0003
33
34 ; while i <= integer ( true ) do
35 ; begin
36      LD     GR1,$i
37      CPA    GR1,GR0
38      JZE    L0006
39      LAD    GR1,1
40 L0006      LAD    GR2,0
41      CALL   WRITEBOOL
42      LAD    GR1,=:'
43      LAD    GR2,0
44      CALL   WRITESTR
45      LD     GR1,$i
46      LAD    GR2,0
47      CALL   WRITEINT
48      CALL   WRITELINE
49
50 ; writeln ( boolean ( i ) , ' : ' , i );
51      LAD    GR1,$i
52      PUSH   0,GR1
53      LD     GR1,$i
54      PUSH   0,GR1
55      LAD    GR1,1
56      POP    GR2
57      ADDA   GR1,GR2
58      JOV    EOF
59      POP    GR2
60      ST     GR1,0,GR2
61 ; i := i + 1;
62      JUMP   L0002
63 L0003
64 ; end;
65      CALL   WRITELINE
66 ; writeln;
67      LAD    GR1,$i
68      PUSH   0,GR1
69      LAD    GR1,32
70      POP    GR2
71      ST     GR1,0,GR2
72 ; i := integer ( ' ' );
73 L0007      LD     GR1,$i
74      PUSH   0,GR1
75      LAD    GR1,127
76      POP    GR2
77      CPA    GR2,GR1
78      JMI    L0009
79      LAD    GR1,0
80      JUMP   L0010

```

```

82 L0009
83     LAD    GR1,1
84 L0010
85     CPA    GR1,GR0
86     JZE    L0008
87 ;  while i < 127 do
88 ;  begin
89     LD     GR1,$i
90     PUSH   0,GR1
91     LAD    GR1,16
92     POP    GR2
93     DIVA   GR2,GR1
94     JOV    EOFV
95     LD     GR1,GR2
96     PUSH   0,GR1
97     LAD    GR1,16
98     POP    GR2
99     MULA   GR1,GR2
100    JOV    EOFV
101    PUSH   0,GR1
102    LD     GR1,$i
103    POP    GR2
104    CPA    GR2,GR1
105    JZE    L0013
106    LAD    GR1,0
107    JUMP   L0014
108 L0013
109    LAD    GR1,1
110 L0014
111    CPA    GR1,GR0
112    JZE    L0011
113 ;  if i div 16 * 16 = i then
114     CALL   WRITELINE
115 L0011
116 ;  writeln;
117     LD     GR1,$i
118     LAD    GR2,#007F
119     AND    GR1,GR2
120     LAD    GR2,0
121     CALL   WRITECHAR
122     LAD    GR1,32
123     LAD    GR2,0
124     CALL   WRITECHAR
125 ;  write ( char ( i ) , ' ' );
126     LAD    GR1,$i
127     PUSH   0,GR1
128     LD     GR1,$i
129     PUSH   0,GR1
130     LAD    GR1,1
131     POP    GR2
132     ADDA   GR1,GR2
133     JOV    EOFV

```

```
134      POP      GR2
135      ST       GR1,0,GR2
136 ;   i := i + 1;
137      JUMP    L0007
138 L0008
139 ;   end;
140      CALL    WRITELINE
141 ;   writeln
142      CALL    FLUSH
143      RET
144 ;   end.
145 ; -----
146 ; Utility functions
147 ; -----
148 ; (sample11pp と同一のため省略)
149      END
```

### sample35.csl の実行結果

```
comet2> run
OUT> FALSE : 0
OUT> TRUE : 1
OUT>
OUT>
OUT> ! " # $ % & ' ( ) * + , - . /
OUT> 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
OUT> @ A B C D E F G H I J K L M N O
OUT> P Q R S T U V W X Y Z [ \ ] ^ _
OUT> ' a b c d e f g h i j k l m n o
OUT> p q r s t u v w x y z { | } ^
Program finished (RET)
```

## 5.5

### COMET II / CASL II の仕様



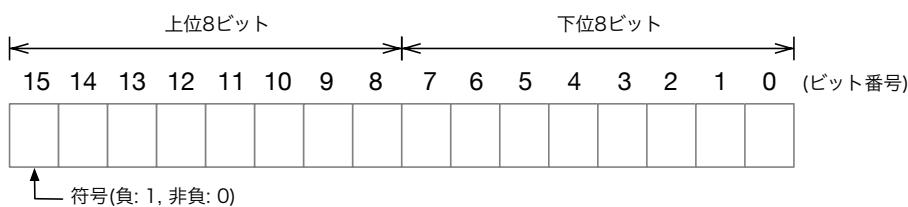
本課題で用いる COMET II エミュレータ / CASL II アセンブラーにおいては、基本情報技術者試験で定義された仕様を独自に拡張した仕様を用いている。オリジナルの仕様は別紙「アセンブラー言語の仕様」[2] にて確認できる。以降の説明で【拡張】とした部分が拡張仕様にあたる。

#### 5.5.1

##### 1. システム COMET II の仕様

###### 1.1 ハードウェアの仕様

(1) 1 語は 16 ビットで、そのビット構成は、次のとおりである。



(2) 主記憶の容量は 65536 語で、そのアドレスは 0~65535 番地である。

(3) 数値は、16 ビットの 2 進数で表現する。負数は、2 の補数で表現する。

(4) 制御方式は逐次制御で、命令後は 1 語長又は 2 語長である。

(5) レジスタとして、GR (16 ビット), SP (16 ビット), PR (16 ビット), FR (3 ビット) の 4 種類がある。

- GR (汎用レジスタ, General Register) は、GR0~GR7 の 8 個があり、算術、論理、比較、シフトなどの演算に用いる。このうち、GR1~GR7 のレジスタは、指標レジスタ (Index Register) としてアドレスの修飾にも用いる。
- SP (スタックポインタ, Stack Pointer) は、スタックの最上段のアドレスを保持している。
- PR (プログラムレジスタ, Program Register) は、次に実行すべき命令語の先頭アドレスを保持している。
- FR (フラグレジスタ, Flag Register) は、OF (Overflow Flag), SF (Sign Flag), ZF (Zero Flag) と呼ぶ 3 個のビットからなり、演算命令などの実行によって次の値が設定される。これらの値は、条件付き分岐命令で参照される。

OF 算術演算命令の場合は、演算結果が-32768~32767 に収まらなくなったとき 1 になり、それ以外のとき 0 になる。論理演算命令の場合は、演算結果が 0~65535 に収まらなくなったとき 1 になり、それ以外のとき 0 になる。

SF 演算結果の符号が負 (ビット番号 15 が 1) のとき 1、それ以外のとき 0 になる。

ZF 演算結果が零 (全部のビットが 0) のとき 1、それ以外のとき 0 になる。

- (6) 論理加算又は論理減算は、被演算データを符号の無い数値とみなして、加算又は減算する。

## 1.2 命令

命令の形式およびその機能を示す。ここで、ひとつの命令コードに対し2種類のオペランドがある場合、上段はレジスタ間の命令、下段はレジスタと主記憶間の命令を表す。

- (1) ロード・ストア・ロードアドレス命令

命令	書き方		命令の説明	FR の設定
	命令コード	オペランド		
ロード LoaD	LD	r1, r2	r1 ← (r2)	○ *1
		r, adr[, x]	r ← (実効アドレス)	
ストア STore	ST	r, adr[, x]	実効アドレス ← (r)	—
ロードアドレス Load ADress	LAD	r, adr[, x]	r ← 実効アドレス	—

- (2) 算術、論理演算命令

命令	書き方		命令の説明	FR の設定
	命令コード	オペランド		
算術加算 ADD Arithmetic	ADDA	r1,r2	$r1 \leftarrow (r1) + (r2)$	○
		r,adr[,x]	$r \leftarrow (r) + (\text{実効アドレス})$	
論理加算 ADD Logical	ADDL	r1,r2	$r1 \leftarrow (r1) +_L (r2)$	○
		r,adr[,x]	$r \leftarrow (r) +_L (\text{実効アドレス})$	
算術減算 SUBtract Arithmetic	SUBA	r1,r2	$r1 \leftarrow (r1) - (r2)$	○
		r,adr[,x]	$r \leftarrow (r) - (\text{実効アドレス})$	
論理減算 SUBtract Logical	SUBL	r1,r2	$r1 \leftarrow (r1) -_L (r2)$	○
		r,adr[,x]	$r \leftarrow (r) -_L (\text{実効アドレス})$	
論理積 AND	AND	r1,r2	$r1 \leftarrow (r1) \text{ AND } (r2)$	○ *1
		r,adr[,x]	$r \leftarrow (r) \text{ AND } (\text{実効アドレス})$	
論理和 OR	OR	r1,r2	$r1 \leftarrow (r1) \text{ OR } (r2)$	○ *1
		r,adr[,x]	$r \leftarrow (r) \text{ OR } (\text{実効アドレス})$	
排他的論理和 eXclusive OR	XOR	r1,r2	$r1 \leftarrow (r1) \text{ XOR } (r2)$	○
		r,adr[,x]	$r \leftarrow (r) \text{ XOR } (\text{実効アドレス})$	
【拡張】算術乗算 MULTiply Arithmetic	MULA	r1,r2	$r1 \leftarrow (r1) * (r2)$	○
		r,adr[,x]	$r \leftarrow (r) * (\text{実効アドレス})$	
【拡張】論理乗算 MULTiply Logical	MULL	r1,r2	$r1 \leftarrow (r1) *_L (r2)$	○
		r,adr[,x]	$r \leftarrow (r) *_L (\text{実効アドレス})$	
【拡張】算術除算 DIVide Arithmetic	DIVA	r1,r2	$r1 \leftarrow (r1) / (r2)$	○ *3
		r,adr[,x]	$r \leftarrow (r) / (\text{実効アドレス})$	
【拡張】論理除算 DIVide Logical	DIVL	r1,r2	$r1 \leftarrow (r1) /_L (r2)$	○
		r,adr[,x]	$r \leftarrow (r) /_L (\text{実効アドレス})$	

### (3) 比較演算命令

命令	書き方		命令の説明	FR の設定																							
	命令コード	オペランド																									
算術比較 ComPare Arithmetic	CPA	r1,r2	(r1) と (r2), 又は (r) と (実効アドレス) の算術比較又は論理比較を行い、比較結果によって、FR に次の値を設定する。	○ *1																							
		r,adr[,x]																									
論理比較 ComPare Logical	CPL	r1,r2	<table border="1"> <thead> <tr> <th rowspan="2">比較結果</th> <th colspan="2">FR の値</th> </tr> <tr> <th>SF</th> <th>ZF</th> </tr> </thead> <tbody> <tr> <td>(r1) &gt; (r2)</td> <td>0</td> <td>0</td> </tr> <tr> <td>(r) &gt; (実効アドレス)</td> <td>0</td> <td>0</td> </tr> <tr> <td>(r1) = (r2)</td> <td>0</td> <td>1</td> </tr> <tr> <td>(r) = (実効アドレス)</td> <td>0</td> <td>1</td> </tr> <tr> <td>(r1) &lt; (r2)</td> <td>1</td> <td>0</td> </tr> <tr> <td>(r) &lt; (実効アドレス)</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	比較結果	FR の値		SF	ZF	(r1) > (r2)	0	0	(r) > (実効アドレス)	0	0	(r1) = (r2)	0	1	(r) = (実効アドレス)	0	1	(r1) < (r2)	1	0	(r) < (実効アドレス)	1	0	○ *1
比較結果	FR の値																										
	SF	ZF																									
(r1) > (r2)	0	0																									
(r) > (実効アドレス)	0	0																									
(r1) = (r2)	0	1																									
(r) = (実効アドレス)	0	1																									
(r1) < (r2)	1	0																									
(r) < (実効アドレス)	1	0																									
r,adr[,x]																											

### (4) シフト演算命令

命令	書き方		命令の説明	FR の設定
	命令コード	オペランド		
算術左シフト Shift Left Arithmetic	SLA	r, adr[, x]	符号を除き (r) を実効アドレスで指定したビット数だけ左又は右にシフトする。 シフトの結果、空いたビット位置には、左シフトの時は 0、右シフトの時には符号と同じものが入る。	○ *2
算術右シフト Shift Right Arithmetic	SRA	r, adr[, x]		
論理左シフト Shift Left Logical	SLL	r, adr[, x]	符号を含み (r) を実効アドレスで指定したビット数だけ左又は右にシフトする。 シフトの結果、空いたビット位置には、0 が入る。	
論理右シフト Shift Right Logical	SRL	r, adr[, x]		

#### (5) 分岐命令

命令	書き方		命令の説明	FR の設定																												
	命令コード	オペランド																														
正分岐 Jump on PLus	JPL	adr[, x]	FR の値によって、実効アドレスに分岐する。分岐しないときは次の命令に進む。  <table border="1"><thead><tr><th>命令</th><th colspan="3">分岐時の FR の値</th></tr><tr><th></th><th>OF</th><th>SF</th><th>ZF</th></tr></thead><tbody><tr><td>JPL</td><td>0</td><td>0</td><td>0</td></tr><tr><td>JMI</td><td>1</td><td>0</td><td>0</td></tr><tr><td>JNZ</td><td></td><td></td><td>0</td></tr><tr><td>JZE</td><td></td><td></td><td>1</td></tr><tr><td>JOV</td><td>1</td><td></td><td></td></tr></tbody></table>	命令	分岐時の FR の値				OF	SF	ZF	JPL	0	0	0	JMI	1	0	0	JNZ			0	JZE			1	JOV	1			—
命令	分岐時の FR の値																															
	OF	SF	ZF																													
JPL	0	0	0																													
JMI	1	0	0																													
JNZ			0																													
JZE			1																													
JOV	1																															
負分岐 Jump on MInus	JMI	adr[, x]																														
非零分岐 Jump on Non Zero	JNZ	adr[, x]																														
零分岐 Jump on ZEro	JZE	adr[, x]																														
オーバーフロー分岐 Jump on OVerflow	JOV	adr[, x]																														
無条件分岐 unconditional JUMP	JUMP	adr[, x]	無条件に実効アドレスに分岐する。																													

#### (6) スタック操作命令

命令	書き方		命令の説明	FR の設定
	命令コード	オペランド		
プッシュ PUSH	PUSH	adr[, x]	SP $\leftarrow$ (SP) $-_L 1$ , (SP) $\leftarrow$ 実効アドレス	—
ポップ POP	POP	r	r $\leftarrow$ ((SP)), SP $\leftarrow$ (SP) $+_L 1$	—

#### (7) コール、リターン命令

命令	書き方	命令の説明	FR の設定
	命令コード オペランド		
コール CALL subroutine	CALL adr[,x]	SP $\leftarrow$ (SP) $-_L$ 1, (SP) $\leftarrow$ (PR) PR $\leftarrow$ 実効アドレス	—
リターン RETurn from subroutine	RET	PR $\leftarrow$ ((SP)), SP $\leftarrow$ (SP) $+_L$ 1	—

(8) その他

命令	書き方	命令の説明	FR の設定
	命令コード オペランド		
スーパバイザコール SuperVisor Call	SVC adr[,x]	実効アドレスを引数として割り出しを行う。実行後の GR と FR は不定となる。	—
ノーオペレーション No OPeration	NOP	何もしない	—

【拡張】SVC 命令は有効アドレスを 1 つとる命令である。本実装では、次のような意味を与えている。

- SVC 0 : ユーザによるプログラム終了
- SVC 1 : ユーザによるプログラム終了
- SVC 2 : ユーザによるプログラム終了
- SVC 3 : ユーザによるプログラム終了
- SVC #fff0 : IN 命令の実体
- SVC #fff2 : OUT 命令の実体

(注)

<b>r, r1, r2</b>	いずれも GR を示す。指定できる GR は GR0～GR7。
<b>adr</b>	アドレスを示す。指定できる値の範囲は 0～65535。
<b>x</b>	指標レジスタとして用いる GR を示す。指定できる GR は GR1～GR7。
<b>[ ]</b>	[ ] 内の指定は省略できることを示す。
<b>( )</b>	( ) 内のレジスタ又はアドレスに格納されている内容を示す。
<b>実効アドレス</b>	adr と x の内容との論理加算値又はその値が示す番地。
<b>←</b>	演算結果を、左辺のレジスタ又はアドレスに格納することを示す。
<b>+L, -L</b>	論理加算、論理減算を示す。
<b>【拡張】*L, /L</b>	論理乗算、論理除算を示す。
<b>FR の設定</b>	<p>○：設定されることを示す。</p> <p>○<sub>*1</sub>：設定されることを示す。ただし、OF には 0 が設定される。</p> <p>○<sub>*2</sub>：設定されることを示す。ただし、OF にはレジスタから最後に送り出されたビットの値が設定される。</p> <p>【拡張】○<sub>*3</sub>：設定されることを示す。ただし、オーバフローが起こったときは OF のみが 1 になり、零除算が起こったときは OF と ZF のみが 1 になり、それ以外では OF は 0 で、SF と ZF は演算結果によって決定される。</p> <p>－：実行前の値が保持されることを示す。</p>

### 1.3 文字の組

- (1) JIS X 0201 ラテン文字・片仮名用 8 ビット符号で規定する文字の組を使用する。
- (2) 以下に符号表の一部を示す。1 文字は 8 ビットからなり、上位 4 ビットを列で、下位 4 ビットを行で示す。たとえば、間隔、4, H, \ のビット構成は、16 進表示で、それぞれ 20, 34, 48, 5C である。16 進表示で、ビット構成が 21～7E（及び表では省略している A1～DF）に対応する文字を図形文字という。図形文字は、表示（印刷）装置で、文字として表示（印字）できる。
- (3) この表にない文字とそのビット構成が必要な場合は、問題中で与える。

		列					
		2	3	4	5	6	7
行	0	間隔	0	@	P	'	p
	1	!	1	A	Q	a	q
	2	"	2	B	R	b	r
	3	#	3	C	S	c	s
	4	\$	4	D	T	d	t
	5	%	5	E	U	e	u
	6	&	6	F	V	f	v
	7	,	7	G	W	g	w
	8	(	8	H	X	h	x
	9	)	9	I	Y	i	y
	10	*	:	J	Z	j	z
	11	+	;	K	[	k	{
	12	,	<	L	\	l	
	13	-	=	M	]	m	}
	14	.	>	N	^	n	~
	15	/	?	O	_	o	

【拡張】(参考) 機械語のバイナリ表現は次の通りである。

- 機械語の1語目は命令部(上位8ビット)とオペランド部(下位8ビット)からなる。
- オペランド部にadrを含む命令は2語命令となり、2語目にadrが割り当てられる。
- 各命令の命令部は次の通り。

```
'NOP'    #00      'LD'      #10,    'ST'      #11,    'LAD'     #12,
'ADDA'   #20,    'SUBA'    #21,    'ADDL'    #22,    'SUBL'    #23
'MULA'   #28,    'DIVA'    #29,    'MULL'    #2A,    'DIVL'    #2B
'AND'    #30,    'OR'      #31,    'XOR'     #32,
'CPA'    #40,    'CPL'     #41,
'SLA'    #50,    'SRA'     #51,    'SLL'     #52,    'SRL'     #53
'JMI'    #61,    'JNZ'     #62,    'JZE'     #63,    'JUMP'    #64,    'JPL'     #65,    'JOV'     #66
'PUSH'   #70,    'POP'     #71,
'CALL'   #80,    'RET'     #81
'SVC'    #f0
```

## 5.5.2

### 2. アセンブラー言語 CASL II の仕様

#### 2.1 言語の仕様

- (1) CASL II は、COMET II のためのアセンブラー言語である。
- (2) 【拡張】プログラムは、命令行、ラベル行、および注釈行からなる。
- (3) 1 命令は 1 命令行で記述し、次の行へ継続できない。
- (4) 【拡張】ラベル行のラベルは次の命令行のアドレスを保持する。
- (5) 【拡張】命令行、ラベル行および注釈行は、次に示す記述の形式で、行の 1 文字目から記述する。

行の種類	記述の形式
命令行	[ラベル]{空白}{命令}{空白}{オペランド}[{空白}[コメント]]
	[ラベル]{空白}{命令}[{空白}[;][コメント]]
注釈行	[空白]{;}{コメント}
【拡張】ラベル行	[ラベル][空白]{;}{コメント}

- (6) 【拡張】命令・レジスタ名は英大文字・小文字のどちらで書いても構わない。
- (7) 【拡張】プログラムは START 命令の行から始まり、END 命令の行で終わるものとする。

(注)

[ ]	[ ] 内の指定が省略できることを示す。
{ }	{ } 内の指定が必須であることを示す。
ラベル	その命令の(先頭の語の)アドレスを他の命令やプログラムから参照するための名前である。 【拡張】ラベルの長さは任意で、先頭の文字は英大文字、英小文字、_(下線)、%(パーセント)、\$(ドル)、.(ピリオド)でなければならない。以降の文字は、英大文字、英小文字、数字、_(下線)、%(パーセント)、\$(ドル)、.(ピリオド)のいずれでもよい。なお、予約語である GR0～GR7、および、gr0～gr7 は、使用できない。
空白	1 文字以上の間隔文字の列である。
命令コード	命令ごとに記述の形式が定義されている。
オペランド	命令ごとに記述の形式が定義されている。
コメント	覚え書きなどの任意の情報であり、処理系で許す任意の文字を書くことができる。

#### 2.2 命令の種類

命令は、4 種類のアセンブラー命令 (START, END, DS, DC)、4 種類のマクロ命令 (IN, OUT, RPUSH, RPOP) および機械語命令 (COMET II の命令) からなる。その仕様を次に示す。

命令の種類	ラベル	命令コード	オペランド	機能
アセンブラー命令	ラベル	START	[実行開始番地]	プログラムの先頭を定義 プログラムの実行開始番地を定義 他のプログラムで参照する入り口名を定義
		END		プログラムの終わりを明示
	[ラベル]	DS	語数	領域を確保
	[ラベル]	DC	定数 [, 定数] ...	定数を定義
マクロ命令	[ラベル]	IN	入力領域, 入力文字長領域	入力装置から文字データを入力
	[ラベル]	OUT	出力領域, 出力文字長領域	出力装置へ文字データを出力
	[ラベル]	RPUSH		全GRの内容をスタックに格納
	[ラベル]	RPOP		スタックの内容を全GRに格納
機械語命令	[ラベル]			「1.2 命令」を参照

### 2.3 アセンブラー命令

アセンブラー命令は、アセンブラーの制御などを行う。

(1) 

START	[実行開始番地]
-------	----------

START 命令は、プログラムの先頭を定義する。実行開始番地は、そのプログラム内で定義されたラベルで指定する。指定がある場合はその番地から、省略した場合は START 命令の次の命令から、実行を開始する。また、この命令につけられたラベルは、他のプログラムから入口名として参照できる。

(2) 

END	
-----	--

END 命令は、プログラムの終わりを定義する。

(3) 

DS	語数
----	----

DS 命令は、指定した語数の領域を確保する。語数は、10進定数 ( $\geq 0$ ) で指定する。語数を 0 とした場合、領域は確保しないが、ラベルは有効である。

(4) 

DC	定数 [, 定数] ...
----	---------------

DC 命令は、定数で指定したデータを（連続する）語に格納する。定数には、10進定数、16進定数、文字定数、アドレス定数の4種類がある。

定数の種類	書き方	命令の説明
10進定数	n	nで指定した10進数値を、1語の2進数データとして格納する。ただし、nが-32768～32767の範囲にないときは、その下位16ビットを格納する。
16進定数	#h	hは4桁の16進数(16進数字は0～9, A～F)とする。 hで指定した16進数値を1語の2進数データとして格納する( $0000 \leq h \leq FFFF$ )。
文字定数	' 文字列 '	【拡張】文字列の文字数( $> 0$ ) +1語分の連続する領域を確保し、最初の文字は第1語の下位8ビットに、2番目の文字は第2語の下位8ビットに、…と順次文字データとして格納する。文字数 +1語目、すなわち、最後の文字は自動的に0(ヌル文字を表す)が格納される。各語の上位8ビットには0のビットが入る。文字列には、間隔および任意の図形文字を書くことができる。ただし、アポストロフィ(')は2個続けて書く。
アドレス定数	ラベル	ラベルに対応するアドレスを1語の2進数データとして格納する。

## 2.4 マクロ命令

マクロ命令は、あらかじめ定義された命令群とオペランドの情報によって、目的の機能を果たす命令群を生成する(語数は不定)。

### (1) IN 入力領域、入力文字長領域

IN命令は、あらかじめ割り当てた入力装置から、1レコードの文字データを読み込む。入力領域は、256語長の作業域のラベルであり、この領域の先頭から、1文字を1語に対応させて順次入力される。レコードの区切り符号(キーボード入力の復帰符号など)は、格納しない。格納の形式は、DC命令の文字定数と同じである。入力データが256文字に満たない場合、入力領域の残りの部分は実行前のデータを保持する。入力データが256文字を超える場合、以降の文字は無視される。入力文字長領域は、1語長の領域のラベルであり、入力された文字の長さ( $\geq 0$ )が2進数で格納される。ファイルの終わり(end of file)を検出した場合は、-1が格納される。IN命令を実行すると、GRの内容は保存されるが、FRの内容は不定となる。

### (2) OUT 出力領域、出力文字長領域

OUT命令は、あらかじめ割り当てた出力装置に、文字データを、1レコードとして書き出す。出力領域は、出力しようとするデータが1文字1語で格納されている領域のラベルである。格納の形式は、DC命令の文字定数と同じであるが、上位8ビットは、OSが無視するので0でなくてもよい。出力文字長領域は、1語長の領域のラベルであり、出力しようとする文字の長さ( $\geq 0$ )を2進数で格納しておく。OUT命令を実行すると、GRの内容は保存されるが、FRの内容は不定となる。

### (3) RPUSH

RPUSH命令は、GRの内容を、GR1, GR2, …, GR7の順でスタックに格納する。

### (4) RPOP

RPOP 命令は、スタックの内容を順次取り出し、GR7, GR6, …, GR1 の順で GR に格納する。

【拡張】 IN / OUT マクロ命令の実装は次のようにになる。

### IN の実装

次のコードがあるとき、

```
IN      INBUF, INLEN
...
INLEN   DS      1
INBUF   DS      20
```

この IN 命令は以下のように展開される。

```
PUSH 0, GR1
PUSH 0, GR2
LAD  GR1, INBUF
LAD  GR2, INLEN
SVC  #ffff0
POP   GR2
POP   GR1
```

### OUT の実装

次のコードがあるとき、

```
OUT    OUTBUF, OUTLEN
...
OUTLEN  DC      13
OUTBUF  DC      'Hello, CASL2.'
```

この OUT 命令は以下のように展開される。

```
PUSH 0, GR1
PUSH 0, GR2
LAD  GR1, OUTBUF
LAD  GR2, OUTLEN
SVC  #ffff2
POP   GR2
POP   GR1
```

同様に、RPUSH, RPOP もマクロなので、7つの PUSH, POP に展開される。

## 2.5 機械語命令

機械語命令のオペランドは、次の形式で記述する。

r, r1, r2	GR は, 記号 GR0～GR7 で指定する.
x	指標レジスタとして用いる GR は, 記号 GR1～GR7 で指定する.
adr	アドレスは, 10 進定数, 16 進定数, アドレス定数又はリテラルで指定する. リテラルは, ひとつの 10 進定数, 16 進定数又は文字定数の前に等号 (=) を付けて記述する. CASL II は, 等号の後の定数をオペランドとする DC 命令を生成し, そのアドレスを adr の値とする.

## 2.6 その他

- (1) アセンブラーによって生成される命令語や領域の相対位置は, アセンブラ言語での記述順序とする. ただし, リテラルから生成される DC 命令は, END 命令の直前にまとめて配置される.
- (2) 生成された命令語, 領域は, 主記憶上で連続した領域を占める.

## 5.5.3

### 3. プログラム実行の手引

#### 3.1 OS

プログラムの実行に関して, 次の取決めがある.

- (1) 【拡張】 エミュレータは 1 つのプログラムしか認めない. 従って, プログラム中で未定義のラベルはアセンブルエラーである.
- (2) プログラムは, OS によって起動される. プログラムがロードされる主記憶の領域は不定とするが, プログラム中のラベルに対応するアドレス値は, OS によって実アドレスに補正されるものとする.
- (3) プログラムの起動時に, OS はプログラム用に十分な容量のスタック領域を確保し, その最後のアドレスに 1 を加算した値を SP に設定する.
- (4) 【拡張】 エミュレータがプログラムをアセンブル, ロードしたとき, 実行開始番地を PR にセットして実行開始直前の状態になっている. プログラムを終了するときは, メインルーチンで RET を使用するか, SVC 命令を利用する.
- (5) IN 命令に対応する入力装置, OUT 命令に対応する出力装置の割り当ては, プログラムの実行に先立って利用者が行う.
- (6) OS は, 入出力装置や媒体による入出力手続の違いを吸収し, システムでの標準の形式及び手続（異常処理を含む）で入出力をを行う. したがって, IN, OUT 命令では, 入出力装置の違いを意識する必要はない.

#### 3.2 未定義事項

プログラムの実行などに関し, この仕様で定義しない事項は, 処理系によるものとする.

## 5.6

### CASL II アセンブラー・COMET II エミュレータ



#### 5.6.1

##### GUI での利用

生成した CASL II のアセンブリコードは、CASL II アセンブラーによって機械語に変換できる。また、その機械語を COMET II エミュレータに渡すことによって実行できる。本実習では、ブラウザ上で実行できる CASL II アセンブラー・COMET II エミュレータを準備した。以下の URL からアクセス可能である。

<https://se.is.kit.ac.jp/casl2/>



はかせとそふらぼの愉快な仲間たちが必死こいて作ったから、バグがあっても文句言わないでね。

The screenshot shows the CASL2/COMET2 interface. On the left, there's a text area for assembly code:

```
1 PRG0000 START
L1 DATA DC 'ABC' DATA
DATA2 ST PC GR1 RESULT
DATA3 RET
RESULT DE 1
END
```

Below it is a green "Assemble" button. To the right, the COMET2 window displays machine code:

```
PR F0000 [ NOP ]  
SP #FFF0C(-256) FK 000C( 0)[ --- ]  
BK1 0000C( 0) GR1 1000C( 0) GR2 2000C( 0)  
GR3 3000C( 0) GR4 4000C( 0) GR5 5000C( 0)  
GR6 6000C( 0) GR7 7000C( 0)  
comet2>
```

At the bottom of the COMET2 window are buttons for "Stop", "Restart", and "Quiet".

生成した.csl ファイルを、左上のテキストエリアにコピーペーストし、Assemble のボタンを押すことでアセンブラーが実行される。アセンブリコードにエラーが無かった場合は、右側の COMET II の対話型エミュレータにオブジェクトコードが転送され、実行が可能になる。COMET II のプロンプトから、run(または r) で実行、step(または s) でステップ実行が行える。help にて利用可能なコマンドが表示される。このエミュレータにはアセンブリコードの実行・デバッグに必要な機能は一通り実装してある。

#### 5.6.2

##### CUI での利用

コマンドラインからの実行に慣れている場合は c2c2.js によって、CUI でのアセンブル・実行を一度に実施できる。コマンドラインツール c2c2.js は、前述の docker イメージに含まれている。実行パスは通っているので、生成した csl ファイルに対して次のように c2c2 コマンド (.js は不要) を実行すれば良い。

```
c2c2 sample11.csl
```

アセンブルに成功すると GUI と同様の対話式エミュレータが起動する。

### 5.6.3

#### 開発体制

開発は以下のリポジトリにおいて、オープンソースとして実施している。アセンブラー・エミュレータに関するバグ報告は、このリポジトリの issue として報告してほしい。

<https://github.com/omzn/casljs>





## あとがき

MPPL コンパイラの作成はいかがでしたか？人によっては、これまでで一番大きいプログラムになったかもしれません。

この演習の中では、コンパイラのプログラムを作る過程で、自動テストや Docker といった技術を活用しました。皆さんはこれからもどんどん大きなプログラムを作成していくと思いますが、この演習で実施したようなスケジュール管理やソースコード管理をこれからも活用して頂ければと思います。また、テストプログラムも自分で実施するためには、本体以上のコーディングが必要になることもあります。今回の演習が1つの経験になれば幸いです。

## 謝辞

本演習は、辻野嘉宏教授が講義「コンパイラ」の実践のために構築したものです。長年にわたる先生のご努力により、大変完成度の高いものになっています。先生のご尽力に感謝いたします。





- [1] 辻野嘉宏. コンパイラ. オーム社, 第2版, 2018.
- [2] 独立行政法人情報処理推進機構. 「情報処理技術者試験」「情報処理安全確保支援士試験」試験で使用する情報技術に関する用語・プログラム言語など, 第4.3版, 2021.