

# 言語処理プログラミング

対象: 情報工学課程 3 回生

担当教員: 水野 修

2022 年 9 月 27 日版

# 目次

<b>第1章 概要</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	7
1.5 質問、連絡、資料等の配布について	8
1.6 実際の演習について	9
1.7 テストについて	9
<b>第2章 課題1 - 字句解析</b>	12
2.1 課題名	12
2.2 演習期間	12
2.3 レポート・プログラム提出期間	12
2.4 課題	12
2.5 プログラム作成条件	15
2.6 例	16
2.7 拡張仕様	17
2.8 スキャナ用ヘッダファイル	17
2.9 課題1メインプログラム例(サンプル)	20
2.10 おまけ	21
2.11 スキャナの作り方のヒント	22
2.12 言語処理プログラミングを行うための事前計画(スケジュール)と実際の 進捗状況について	24
<b>第3章 課題2 - 構文解析</b>	29
3.1 課題名	29
3.2 演習期間	29
3.3 レポート・プログラム提出期間	29
3.4 課題	29
3.5 LL(1) 構文解析系の作り方(課題2相当)	34
3.6 課題3以降への対応について	37
<b>第4章 課題3 - 意味解析</b>	38
4.1 課題名	38
4.2 演習期間	38

## 言語処理プログラミング

---

4.3	レポート・プログラム提出期間	38
4.4	課題	38
4.5	クロスリファレンサの作り方	42
第5章	課題4 - コンパイラ	45
5.1	課題名	45
5.2	演習期間	45
5.3	レポート・プログラム提出期間	45
5.4	課題	45
5.5	コード生成の方法	49
5.6	CASL II アセンブラ・COMET II エミュレータ	62
参考文献		63

# 目次

2.1	MPPL のマイクロ構文	13
2.2	入力ファイル例 (sample11pp.mpl)	16
2.3	プログラム sample11pp.mpl に対する出力例	17
2.4	簡単な PERT 図	26
2.5	サブタスクレベルの PERT 図	26
3.1	EBNF による MPPL の構文	30
3.2	プリティプリントの例 (1)	32
3.3	プリティプリントの例 (2)	33
4.1	MPPL の構文に対する制約規則 (1/2)	39
4.2	MPPL の構文に対する制約規則 (2/2)	40
4.3	課題 3 の実行例 (入力)	42
4.4	課題 3 の実行例 (出力)	43
5.1	MPPL のセマンティクス (1/3)	46
5.2	MPPL のセマンティクス (2/3)	47
5.3	MPPL のセマンティクス (3/3)	48

# 表目次

2.1	事前作業計画の例 (日付は過去の例である)	27
2.2	簡単な事前作業計画の例 (日付は過去の例である)	28

# 第 1 章

## 概要

### 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する。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る。

### 1.2 構成とスケジュール

---

本科目は、次の 4 ステップからなる。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る。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すること。

#### 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: 2022-10-03


演習期間: 2022-10-03 ~ 2022-10-23

レポート・プログラム提出期間: 2022-10-24 ~ 2022-10-31(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

 C 言語の教科書はいつでも必携ですよ。

 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: 2022-10-24

演習期間: 2022-10-24 ~ 2022-11-20

レポート・プログラム提出期間: 2022-11-21 ~ 2022-11-28(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

 かわいくない?

### 1.2.3 課題 3: クロスリファレンスの作成

---

課題内容関連講義予定日: 2022-11-21

演習期間: 2022-11-21 ~ 2022-12-11

レポート・プログラム提出期間: 2022-12-12 ~ 2022-12-19(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: 2022-12-12

演習期間: 2022-12-12 ~ 2023-01-22

レポート・プログラム提出期間: 2023-01-23 ~ 2023-01-30(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(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.

### 1.3.1 レポートの構成

---

レポートには, 次の事項を含むこと.

#### 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, 関数〇〇が課題 1 で説明済みであり何も変更されていないければ,

関数〇〇: 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

#### (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、 $\mu$ 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、氏名のみを記載すること

**提出先** Moodle上の指示されたところ

**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Moodleを経由して提出すること。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る。

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)である。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せよ。

本演習は、個人演習である。各自がプログラムとレポートを提出すること。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ある。他人のプログラムを見てはいけないし、見せてもいけない。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)となる。

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)を生ずることがあった(過去の例)。コンパイルエラーなどでこちらのテストを通らない場合、最低評価となる。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勧める。

どのような環境でもコンパイル、実行できることを期待するが、みなさんがそれを確認することはかなり困難である。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を指定する(Linux上のEclipse環境ではないことに注意)。ここで、コンパイル、実行ができることを確かめること。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 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ること。

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われるが、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ある。

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一つのファイルにまとめて提出してもよい。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されない。

## 1.5 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を用いて行う。

質問は、Redmineにて受け付ける。URLはMoodleにて指示する。また、o-mizuno@kit.ac.jpへメールを送って質問しても良い。メールの本文の最初に学生番号と氏名を明記すること。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)の初めに8-205演習室で受け付ける(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)まで質問しに来ててもよいが、不在や会議等で質問を受け付けられない場合がある。

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。

- メールの場合は名乗る。
- プログラムが動かない場合は、動かない状況を詳述する。
- 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、自身が開発している構成をzipやtarでアーカイブして、どのようにすれば実行ファイルが生成できるのかを明示して送ること。

## 1.6 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るが、密になるのを避けるために、同時時間帯に CIS 演習室も利用可能である。実際には、C コンパイラ等があれば演習可能なので、上記の確認作業以外は、個人持ちの PC を含めてどこで演習を行ってもよい。実際週 1 コマの演習だけでは完成しないと思われるので、十分な時間外演習を期待する。

## 1.7 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るならば、テストは単体テストと統合テストに分けられる。

### 1.7.1 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る。

### 1.7.2 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らない。

#### 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る。

#### 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る。埋め草 (stub) とも呼ばれる。

#### 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らない。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なくてはならない。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。

### 1.7.3 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

トデータを用意する。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ある。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、必ずテストしなくてはならない。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である。

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うことが望ましい。なぜなら、システムの内部設計を知っている人が行くと、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であるためである。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いことになる。

#### 1.7.4 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。実装に基づいてテストがどの程度十分かを評価できる。標準的な基準に次の二つがある。

**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する。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する。

**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する。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る。

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あるが、モジュールサイズが大きくなった場合、コストや時間の制限から C0 カバレッジがやっと、という場合も多い。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る。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よい。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。

#### gcov

ホワイトボックステストを実行するツールとして、gcovが知られている。

gcovはgccのカバレッジ検出ツールであり、自分のプログラムを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る。

```
$ gcc --coverage -o tc myprogram.c
```

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・C1カバレッジの計算に必要な情報が蓄積される。

```
$ ./tc sample1.mpl
$ ./tc sample2.mpl
$ ./tc sample3.mpl
```

この後、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。これがカバレッジ情報である。

```
$ ls
tc*      myprogram.c      tc.gcda  tc.gcno
```

ここで生成された gcda ファイルに対して、gcov を実行する。

```
$ gcov -b tc.gcda
File 'myprogram.c'
Lines executed:83.33% of 12
Branches executed:100.00% of 4
Taken at least once:100.00% of 4
No calls
myprogram.c:creating 'myprogram.c.gcov'
```

この結果からは以下のことが読み取れる。

- 命令網羅 (lines executed) は 83.33%
- 分岐命令の数 (4) は、条件判定の分岐の数を表す。今回は if 文, for 文につき、2 つずつ。
- Branches executed: 通過した分岐命令の数 (真偽は問わない)
- Taken at least once: 分岐命令の真偽を 1 度でも通った数 (即ち、分岐網羅)
- 分岐網羅 (Taken at least once) は 100%

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る。テキストエディタで開いて読むことができる。

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る。\*.gcov, \*.gcda, \*.gcno のファイルをすべて削除すればよい。

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る。

## 第 2 章

# 課題 1 - 字句解析

### 2.1 課題名

---

字句の出現頻度表の作成

### 2.2 演習期間

---

2022-10-03 ~ 2022-10-23

### 2.3 レポート・プログラム提出期間

---

2022-10-24 ~ 2022-10-31(月)

### 2.4 課題

---

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 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。

```

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" "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 (シングルクォート)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

- 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、
- 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意の表示文字を表す、
- 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、あったとしても、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。

**[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。表の 1 行の行頭に字句 (余分なスペースを含んでも良い) をダブルクォーテーションで囲んで表示する。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を表示する。以上の仕様に基づいて、たとえば、

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

のように出力せよ。ただし、名前 (NAME)、文字列 (STRING)、符号なし整数 (NUMBER) はその実体が異なってもそれぞれを同じ字句として扱い、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)。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。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) を標準出力へ出力し、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。

## 2.5 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 (スキャナ) 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。

### 2.5.1 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。

### 2.5.2 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。

### 2.5.3 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、32767 よりも大きい値の場合は、エラーである。

### 2.5.4 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。

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、エラーである。

### 2.5.5 行番号関数

```
int get_linenum()
```



```

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 2.2 入力ファイル例 (sample11pp.mpl)

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。

### 2.5.6 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フェース関数を用意してもかまわない。

## 2.6 例

図 2.2 と図 2.3 に本課題の入力、出力の例をそれぞれ示す。

```

"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

```

図 2.3 プログラム sample11pp.mpl に対する出力例

## 2.7 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に拡張せよ。つまり、n, sum 等の名前毎にも出現個数を数えて出力せよ。

## 2.8 スキャナ用ヘッダファイル

```

/* token-list.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 TNOTEQ   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

/* token-list.c */

#define KEYWORDSIZE 28

extern struct KEY {
```

```
    char * keyword;
    int keytoken;
} key[KEYWORDSIZE];

extern void error(char *mes);

/* scan.c */
extern int init_scan(char *filename);
extern int scan(void);
extern int num_attr;
extern char string_attr[MAXSTRSIZE];
extern int get_linenum(void);
extern void end_scan(void);
```

**2.9** 課題 1 メインプログラム例 (サンプル)

```

#include "token-list.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) {
    printf("File name id not given.\n");
    return 0;
}
if(init_scan(np[1]) < 0) {
    printf("File %s can not open.\n", np[1]);
    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;
}

void error(char *mes) {
    printf("\n ERROR: %s\n", mes);
    end_scan();
}

```

## 2.10 おまけ

id-list.c

```

#include "token-list.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) == 0) 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) {
        printf("can not malloc in id_countup\n");
        return;
    }
    if ((cp = (char *)malloc(strlen(np)+1)) == NULL) {
        printf("can not malloc-2 in id_countup\n");
        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 the data structure */
    struct ID *p, *q;

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

```

## 2.11 スキャナの作り方のヒント

スキャナの作成は、かなり難しそうに思えるが、次の点を踏まえれば、見通しがよくなる。

- 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

ワードになる。また、数字であれば、符号なし整数である。"+"であれば、記号"+"しかあり得ない。他も同様である。分離子があるかもしれないので、それを考慮すると全体の構成は次のようになる。

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 だけでも名前であるが、ab も abc も名前である。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 つの字句となる。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る (10a などはず句ではない)。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る (記号"+"のように確定するものもある)。従って、入力は常に 1 文字先読みしておくとし便利である。即ち、1 文字分の文字バッファを持っていて、それに次の文字が入っているようにする。以降、この文字バッファを

```
int cbuf;
```

として、説明する。

### 2.11.1 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。オープンに失敗したらエラーで終わる。

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

### 2.11.2 トークンを一つスキャンする関数

```
int scan()
```

switch 文で処理が分かれる

```
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(...)...` を使ってもよいし、これらを組み合わせてもよい。

EOF には、気を付けること。特に、文字列やコメント内の処理中に EOF が現れても無限ループにならないようにすること。

### 2.11.3 行番号関数

```
int get_linenum()
```

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。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せよ。初期化関数でこの変数を初期化しておく。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るが、スキャ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る。それを避けるために、トークンの 1 字めを読んだときに、そのトークン用の行番号を確定するようにするとよい。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する。

## 2.12 言語処理プログラミングを行うための事前計画 (スケジュール) と実際の進捗状況について

### 2.12.1 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ある。特に作業結果の質を落とせなくて、`バッチ`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る。

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ある。スケジュールがあると次のようなメリットがある。

- 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。`バッチ`前日に遅れに気づいても時間がないので対処できない (`バッチ`を守れない)。

従って、スケジュールを立てるためには、次の情報が必要である。



「進捗は？」あくあたんの出番だ！

- (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、作業の重要度やバッチの切迫度で幾分変動する。

(a) については、作業はできるだけ細かな部分作業に分割するのが望ましい。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ある。また、(b) を考えることで、部分作業の抜け (考え落とし) を防止する効果もある。

とはいえ、初めて行う作業 (今回の演習が該当するかもしれない) の場合は、何をすればよいかがよくわかっていないので、(a) で細かく分割できないかもしれない。その結果 (c) での見積もりも甘いものにならざるを得ない。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わかってくるものである。従って、再スケジュールが必要となる。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随時行うべきものである。もちろん、再スケジュールを行っても、最後のバッチを守るように再スケジュールしなくてはならない。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る。

### 2.12.2 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る。次はその例 (見積もり時間なし) である。段付けは部分作業 (サブタスク) を表す。

- (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 (スキャナ) 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) 2.9 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) 2.9 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 (スキャナを利用))
  - (d-3) 2.9 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) スキャナの作成
- (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

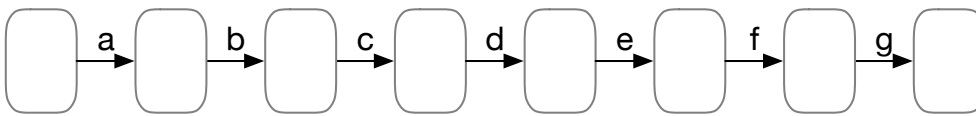


図 2.4 簡単な PERT 図

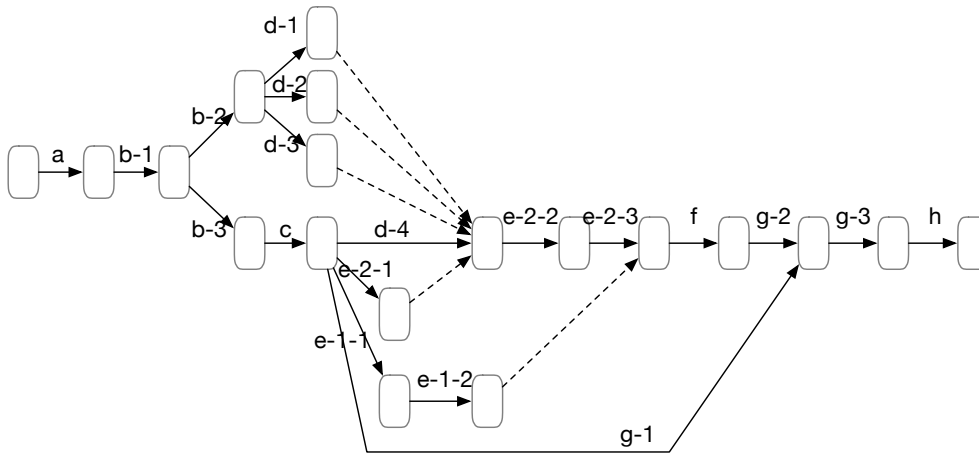


図 2.5 サブタスクレベルの PERT 図

- (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する。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パート (PERT) 図という。

図 2.4 は書くまでもない簡単な図であるが、サブタスクレベルまで分解して書くという意味のある図になる。例えば、(e-1) は (c) の後直ちに実行可能である。また、早い段階から開始することができる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る。このようなタスクはさらにサブタスクに分割できることが多い。

図 2.5 は、サブタスクレベルで書いたパート図の例である。ただし、点線の矢印はダミーのタスクである (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るので、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、実際の実行順を決定する。

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) を前から順に決定する (これをレポートに書くこと)。

## 言語処理プログラミング

表 2.1 事前作業計画の例 (日付は過去の例である)

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 (スキャナ) 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) スキャナの作成
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

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ない。また、最後のタスクの実行終了予定日はメ切り日より前でなくてはならないし、もっと言うなら、十分な予備日が必要である。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し、低ければそれなりの日数を用意するのが必要である。見積もり時間を多く取ると、(メ切り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る。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、ということである。

万一、実際の進捗がスケジュールよりも遅れた場合、対処法は次の通りである。

- 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。
- 本来別のことに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る。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ある。質問等は、どんな内容でも積極的に行うこと。

表 2.1 に事前作業計画の例を挙げる。注意点を以下に挙げる。

- 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、それをコピーするだけで済むはず。

進捗は？  
睡眠時間は削っちゃダメだよ。あくあたんの約束だよ。

表 2.2 簡単な事前作業計画の例 (日付は過去の例である)

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 (スキャナ) の概略設計
10/10	10/20	10	(d) スキャナの作成 (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

- プログラムとレポートの提出予定日がㄆ切日より早いのは、予備時間を取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 2.2 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。

## 参考文献

- [1] 辻野嘉宏. コンパイラ. オーム社, 第 2 版, 2018.