

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 10 月 20 日版)



目次

第1章 概要: 言語処理プログラミングによるこそ	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
第2章 課題1 - 字句解析	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 スキーマ用ヘッダファイル	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 スキーマの作り方のヒント	36
2.10 テストケースの意味	38
第3章 課題2 - 構文解析	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

第4章 課題3 - 意味解析	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
第5章 課題4 - コンパイラ	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
参考文献	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 sample11pp.mpl 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



プリティプリンタって、かわいくない?

課題内容関連講義予定日: 2024-10-21(月)

演習期間: 2024-10-21(月) ~ 2024-11-17(日)

レポート・プログラム提出期間: 2024-11-18(月) ~ 2024-11-25(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: 2024-11-18(月)

演習期間: 2024-11-18(月) ~ 2024-12-08(日)

レポート・プログラム提出期間: 2024-12-09(月) ~ 2024-12-16(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: 2024-12-09(月)

演習期間: 2024-12-09(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[4](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(□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タ (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は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) □ 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) □ 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) □ 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パート (PERT) 図といいます。

図 □ 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えます。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。

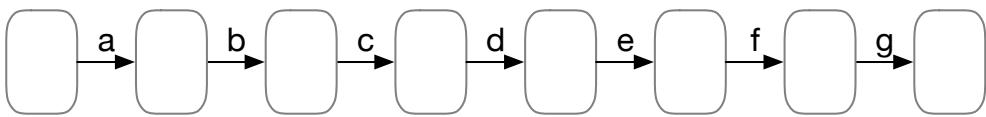


図 1.1 簡単な PERT 図

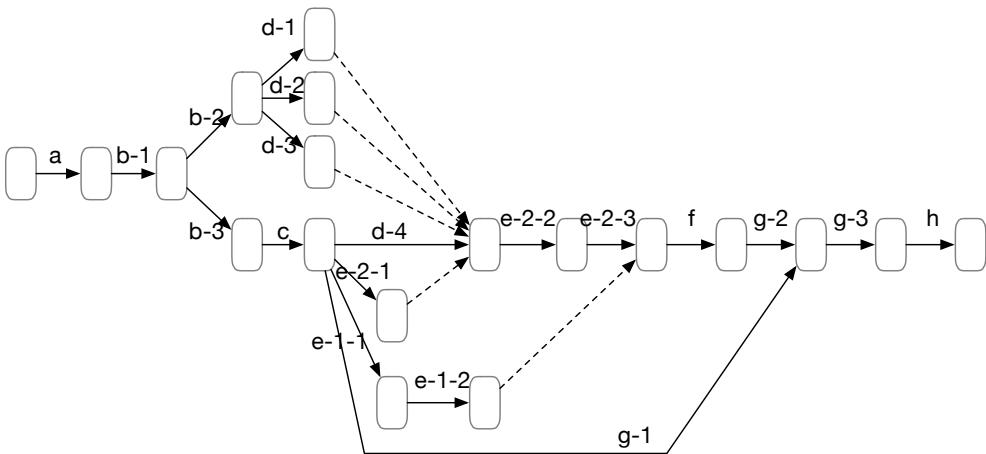


図 1.2 サブタスクレベルの PERT 図

図 □ 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 □ 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 □ の通り単純になってしまふ。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 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 2.2)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

スキャ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 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

#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", "+",
    "-", "*", "=", "<>", "<", "<=",
    ">", ">=", "(", ")",
    ";", ".",
    ":"};

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）

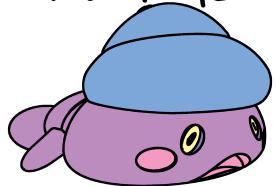
`sample012seofmpl` 文字列開始後に EOF（字句エラーにすべき）

`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) ~ 2024-11-17(日)

レポート・プログラム提出期間: 2024-11-19(月) ~ 2024-11-25(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 B-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
分岐文 (condition statement) ::= "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。
繰り返し文 (iteration statement) ::= "while" 式 "do" 文
脱出文 (exit statement) ::= "break"
    # この脱出文は少なくとも一つの繰り返し文に含まれていなくてはならない
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で"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"の次で改行するときは、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B-2と図B-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 [II] 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5-1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。[B.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、[B.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。





- [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.