B演習 (言語処理系演習) 資料 (0)
課題内容の説明

田浦

2006/10/2

1 課題

共通課題として,

- 簡単ながらも完全なプログラミング言語処理系 (インタプリタ) を作る.
- 自分で作った処理系で規定プログラムを走らせて動作をテスト，性能を測定する.

さらに後半は自分の処理系で自由なプログラムを書く，処理系を高速化する，Garbage Collection などの高度な機能を自分で実装する，分散処理などの機能拡張をする，などの発展的な課題に取り組む．詳細仕様は 4 節を参照.

2 学んで・経験してほしいこと

言語処理系の仕組み もちろん，**コンピュータになくてはならない道具であるプログラミング言語処理系の仕組みについて学ぶ**ことが目標であり，それは情報系の学科であればほとんどの場合必須の学習項目になっている．なぜそれほど重要かについては，課題内容の説明から若干それるので，3 節に語りを入れておいた．疑問・興味がある人は，読んでみてほしい.

少し大きなプログラムの作り方 しかし，処理系の仕組みを学ぶという目的と同じか，あるいはもっと重要なテーマは，**そこそこのサイズのプログラムを，長い時間をかけて作る**ということである．言語処理系であろうと何のプログラムであろうと，ある程度大きいプログラムは，闇雲にコードを書き連ねていっても決して動かない．ある程度の規模以上の，正しく動くプログラムを作るには，ある種の「原則・考え方」が必要である．言葉ではそれを「抽象化」「モジュール化」などの言葉でしばしば呼ぶが，その意味は実際にコードを書いて格闘をしないと分からない．これまで駒場や 3 年前期の授業や実験でいくつもプログラムを書いてきたと思うが，その多くは言語の構文や使い方を学ぶのが主目的であったり，ちょっとしたアルゴリズムを理解するためにコーディングをするといった小規模なものであったと思う．ここでは，言語の構文の習得や処理系の使い方の一段先にある，動くプログラムの「作り方」「考え方」を取得することを目標としよう．それらはどんな言語

でプログラムを書く場合でも必要な考え方である．ちなみにこの演習を設計するに当たって私自身が書いてみた結果は，共通課題部分だけで C のソースファイルにして 10 ファイル，合計 4,000 行程度であった．世の中で動いている多くのソフトウェアと比べれば全く大きくはないが，それでもなんの discipline もなく書いていって動く量でもない．このコードを書ききれば,「ソフトウェアプロジェクトを実現する」スキルにはかなりの自身がつくだろう．

C/C++言語の仕組み・低水準ソフトウェアの動き　さらに，この演習を通して学んでほしいことは，インタプリタを C 言語で書くことで，C/C++言語の実行の仕組み，およびそれを通してみたコンピュータソフトウェアの低水準な (機械語に近い) 部分の動きである．それらは駒場のプログラミング演習やコンピュータアーキテクチャの授業で学んだはずのことであるが，今回はそれをどっぷりと実地体験してもらいたい．C/C++言語で書いたプログラムがどう動作をするかは，プログラムが「正しければ」何も C/C++言語の仕組みを知らなくとも理解できる (もともと高級言語はコンピュータの仕組みの詳細を知らなくてもプログラムが書けるようにするのが目的であったのだからこれは当然である)．`x = 1` を実行すれば `x` が `1` になるというだけの話で，これを理解するのに，C/C++言語がどのような仕組みで実行しているかを知る必要はない．ところが，間違いのあるプログラム (世の中のほとんどのプログラムがそうなのだが) はそうはいかない．特に，C/C++言語で書かれたプログラムは親切なエラーメッセージを出す代わりにありとあらゆる「意味不明な挙動」をする.「C/C++言語の大体の仕組みを知っている」=「C/C++言語の式や文の実行がだいたいどんな機械語に対応しているかをイメージできる」人は，そのような意味不明な挙動に対して正確な推論・分析ができて，正確なアクションが取れるのである．プログラムをさっさと動かすだけなら最初から C/C++言語を使わないというのも選択肢の一つだが，ここでは C/C++言語を使いこなせるようになることよりも，コンピュータソフトウェアの低水準の仕組みを理解すること自体を目的のひとつとする．

言語理論　さらに，言語処理系を作るにあたってはこれまでアルゴリズムの授業で学んだこともさまざまに応用される．言語処理系の基本はとにもかくにも与えられたプログラム (結局は文字列の形で処理系に入力される) を正確に解釈することである．正しいプログラムを正しく実行することはもちろんのこと，構文エラーを含むプログラムを拒否することは，言語処理系の最重要な機能である．入力されたプログラムに構文的な誤りがないかどうかを判定する処理は構文解析，パージング (parsing) などと呼ばれるが，つまりは「正確な文字列処理」ということである．それには前学期のコンピュータソフトウェアの授業にあった「形式言語理論」で学んだ正規表現や文脈自由文法の枠組みの活用が実際に重要な意味を持つ．一言で言って，構文的に正しいプログラムの集合を文脈自由文法の枠組みで定義・表現し，あとは文脈自由文法で規定された文法から構文解析プログラムを「形式的に」「ほぼ機械的に」導き出す．このやり方は，言語処理系に限らず少し複雑な構文の認識をする場合に共通の方法で，ありとあらゆるソフトウェアの中に多かれ少なかれ出現するものである．プログラミング言語のような複雑な文法を持つものに限らず，ちょっとした設定ファイルを持つソフトウェア，ネットワークからの入力を受け付けるソフトウェア，などなど，ありとあらゆる場面で出現する．

実はあまりにも多く出現するので，多くの言語には,「正規表現マッチングライブラリ」や文脈自由言語の構文解析器の生成器のような便利なツールが存在する．それらを使えば文法からパーザを「形式的に」「ほぼ機械的に」導き出す部分を，本当に機械 (コンピュータ) にやらせることもでき

る．今回の演習ではあえてそれらなしで構文解析をしてもらうのだが，それを行った後で，それらのツールの話も時間があればする予定である．

プログラムの速度　プログラムの速度はどんなソフトウェアにとっても重要であるが，言語処理系のようなさまざまなプログラムの基盤となるソフトウェアにとっては特に重要である．言語処理系が遅ければ，その言語で書かれたプログラムはことごとく遅いということになり，言語処理系が早ければその言語で書かれたプログラムを個別に高速化しなくてもプログラムは速く動く．高速化の手段は表面的なC/C++言語のコーディングテクニック，データ構造やアルゴリズムの工夫など，多岐に及ぶ．それらを試しては処理系の性能を測る，というプロセスを楽しんでほしい．授業で作る言語の仕様は，世の中で実際に広く使われている言語のサブセットとしてある．つまり，自分が作った処理系で走るプログラムは，そのまま (あるいはほんの少し変更すれば)，世の中で実際に使われている既存の処理系でも動く．そのような処理系と速度比べをしてみると面白い．実を言うと授業で説明する作り方で作られた処理系は，そのままでは決して速いとはいえない処理系である．各自の工夫を期待したい．さらに，言語処理系にインタプリタという方式と，コンパイラという方式があり，多くの場合両者には何倍もの速度差がある．本演習で作るのはインタプリタ (遅いほう) なのだが，それを作った後で，今学期のプログラミング言語処理系の授業を聞けば，きっとコンパイラが「究極の高速化の手法」として生き生きと理解できるであろう．

アルゴリズム　言語処理系の中身では，ハッシュ表や，リストなどの，コンピュータソフトウェアの授業に出てきたようなアルゴリズムやデータ構造がいくつも使われる．この演習では，これまでの授業で学んだ基本的なアルゴリズムやデータ構造が実際に使われる場面を見ることになるだろう．もちろんこれらについても,「よく使われる」が故に，多くの言語 (C++, Java, その他ありとあらゆる言語) ではライブラリまたは組み込みのデータ構造として提供されている．したがって自分で一から作ることはあまりないのだが，一度それらの経験をしておけば，大きなライブラリの中から，目的に合致した正しいデータ構造を選べるようになるだろう．

デバッグ　書いた直後からプログラムが正しく動くことはまずありえない．プログラムが正しく動かないときに，じーっとコードを眺めてどこかに間違いがないかを探す，という方法は，100 行までのプログラムにしか通用しない．デバッグをする際の正しい考え方というか，心構えについて，一般論を述べるのは難しいのだが，実際にデバッグ中のプログラムを題材にして授業中に説明を試みたい．なお，C/C++でプログラムをデバッグする際にはデバッガが必須である．デバッガを使いこなすことははデバッグのすべてではないが，使えるのとそうでないのとでは大きな差が出る．これについても説明を行う．

Emacs の少し高度な使い方　プログラムを作る際は，ソースコードを編集するエディタから，コンパイラ，プログラムを実行するためのシェル，デバッガなど，さまざまなツールを使う．それらのツールのよりよい使いこなし方，特に Emacs (Windows 版は Meadow と呼ばれている) というエディタに備わっている，プログラム開発を効率化するさまざまな機能についても，この機会に習得してみよう．

　たとえば，プログラムを作っている最中には，何度かソースコード全体に及ぶ変更を余儀なくされることがある．たとえば何十個もの関数すべてに引数をひとつ追加しなくてはならない，という

ような場合が出てくる．これを目で追いかけて一つ一つ変更していくのは大変だし何よりもやる気がしない．エディタにはこのような繰り返しを効率的に行ったり，自動化する機能が含まれている．とくに Emacs/Meadow には，エディタ内でシェル，コンパイラ，デバッガなどを走らせたり，文字列の置換，定型的な処理の繰り返しを自動化するなどの機能 (キーボードマクロ) が充実している．Microsoft Visual Studio などの「統合開発環境」も似たような機能を持っているが，Emacs でそれらの機能を覚えると，どんなプログラミング言語でも同じエディタを使うことができるし，プログラム開発のための機能も開発環境に合わせて覚えなおさなくてすむ．

3　なぜ言語処理系を作るのがそんなに重要なワケ?

これまでの説明で,「よし，これまでのコンピュータ，ソフトウェア，プログラミング，アーキテクチャなどに関して学んだことを総動員してひとつの，小さくないソフトウェアとして結実させることを楽しんでみるか」という気になってくれることを期待するが，今もって「なぜ，自分が言語処理系を作るのか?」疑問の向きもあるかもしれない．特に,「え? 言語処理系なんて C や Java を使えばいいんでしょ? それってわざわざ自分で作るものなの?」なんていうことを思うかもしれない．また，ネットワークの研究がしたい，画像処理の研究がしたい，etc. という人は自分がまかり間違っても言語処理系を作ることはない，と思うかもしれない．ありとあらゆる情報処理の基本だから一度は学んでおきましょう，という以上に強く，言語処理系を作る理由はあるのだろうか，と思ったとしても，それはもっともではある．

そのひとつの答えは，言語処理系，またはそれに近いものがさまざまなソフトウェアの中に組み込まれている，ということである．たとえばおなじみの Microsof Word, Excel, PowerPoint といったソフトウェアがある．それらには，Visual Basic という言語処理系が組み込まれていて，ユーザの要求に合った新しい処理を Basic という言語で記述できるようになっている．つまり，それらの office アプリケーションを作った人は最初に Visual Basic という言語処理系を作ったのである．他にも，Emacs/Meadow には Emacs Lisp という言語処理系が組み込まれていて，やはりその言語でプログラムを書くと，Emacs にいろいろと高度な処理をやらせることができる．これも office 同様，Emacs を作った人は最初に，Emacs Lisp という言語処理系を作ったのである．Unix のコマンドを組み合わせてちょっと高度なことをするために，シェルスクリプトを書いたことがある人もいると思うが，あれも，もちろん言語処理系の一種である．シェルを作った人は，単に打ち込まれたコマンドを実行するという，最低限の機能ではなく,「シェル言語」という言語の処理系を作り，いわばそのもっとも単純な場合として「ユーザがうちこんだプログラムを実行する」という機能を位置づけたのである．

これらの例を見ても分かるように，いろいろなアプリケーションで，一皮向いたところにプログラミング言語とその処理系が活躍している．なぜ，このようなソフトウェアの作り方をするのであろうか? Word であれば文章が書ければよいのに，わざわざ「Word を操るための言語」を作るのはなぜか? ひとつの理由は，ユーザに拡張性，つまり新しい処理を付け加えるための自由度を与える，ということである．これは「マクロ」などとしばしば呼ばれているものであるが，立派なプログラミング言語である．もうひとつの重要な理由は，たとえ拡張性を考えなかったとしても，ある大きなソフトウェアを作るのに，

- 最初に言語処理系を作り

- その後，そのできた言語でアプリケーションの多数の機能を作っていく

という 2 段階を経ることで，ソフトウェアを早く作ることができ，しかも安定性が向上するということである．もちろん開発者があとで機能を追加，修正したりするのもずっと容易になる．たとえばワープロを C 言語で書くプロジェクトを考える．実装しなくてはいけない機能はカーソルを一文字移動することから始まって，文字の表示，文章の保存や読み込み，など無数に及ぶ．それらの機能をすべて C 言語で書く代わりに，

- 非常に原始的な機能だけを C 言語で実装する
- それらの原始的な機能を簡単に呼び出すことのできる高水準言語 (Visual Basic に相当) を作る
- 残りのほとんどの機能を，作った処理系で記述する

という段階を経る．一見して回り道のようだが，まさに「急がば回れ」であって，それによって全体の生産性が圧倒的に向上するし，後々機能拡張をするのも容易になる．言い換えると，諸君がこの先新しいソフトウェアの構想を実現にうつす際に，そのソフトウェアが本格的なものであればあるほど，そのアプリケーション用言語を設計，実装するというステップを経ることが多いのである．そのような言語はほとんどの場合，そのアプリケーションが必要とする基本機能を備えており，C 言語で記述すると何十-何千行に及ぶような処理を一行で行えるような「高水準」なものであるのが特徴である．

今回構築する言語は特定のエンドユーザアプリケーションを指向したものではないが，C 言語よりはるかに高水準で，まともに作れば，ちょっとしたアルゴリズムを記述するのに C で書くよりもはるかに生産性が高い，というレベルのものにはなるはずである．複雑なプログラムが「動き出し」，動き出した後は自分がプログラムを書く際の効率 (生産性) をあげてくれる，つまり自分の仕事を助けてくれる，そういう喜びに浸れるところまで，是非行っていただきたい．

今回作る言語は巷でも良く使われている言語 Python の仕様の一部を取り出してきたものである．したがって，今回諸君は副産物として，言語 Python を習得することにもなる．Python は大まかに「スクリプト言語」と呼ばれる範疇に分類される．他にもスクリプト言語と呼ばれる言語の代表者には，シェル，Perl, Ruby などがある．そうでない言語で有名なものは，C/C++, Java であろう．両者の間に明確な境界線はないが，スクリプト言語は小さなプログラムが少ない記述量でさっと書け，余分な手間をかけずに即実行できることが命である．したがって，日常的に現れる仕事で，手作業ではいやになるような繰り返しを自動化したりするのによく用いられる．たとえばたくさんのファイルに入っている実験データを読み込んでひとつにまとめてグラフにする，などの仕事をその場でプログラムを書いて，すぐに実行して，終わったらすぐに捨てる，というようなことが良く行われる．Python は，このような日常よく行われる仕事を非常に少ないコード量でてきぱきと記述・実行できる言語で，多くの人がこの演習後も重宝することになるのではないかと思われる．

4 課題の仕様・進め方

4.1 概要

- **一人で行うか，または二人一組でチームを作る．**できるだけ二人一組でチームを作り，以下

に書かれた要領 (ペアプログラミング) を用いて，共同作業をすることを薦める．チームの相手は自由に探してよい．

- 前半と後半にそれぞれ一回，都合 2 回の課題を提出する．
- 授業の最終回，または最後の 2 回程度を使って，完動した人の発表会 (デモ) を行う．レポートの期限というわけではないが，このときまでに完動することを目指す．
- 課題の作り方がさっぱりわからないということがないように，それなりに詳細な説明をするが，細部まで真似をせよという意味ではない．詳細を聞かなくてもわかる人はとにかく作っていけばよい．
- 出席を取る．もちろん自分のため，相手チームメンバーのために出席をしなくてはならないのだが，それでも出席をしない，という人はこちらで把握する．
- 力の余りそうな人に限り，最初からオリジナルな設定で課題をやってもよい．たとえば Python 処理系を作る代わりに Perl の処理系を作ってもよい．ただしその場合の課題は，規定の課題と同等あるいはそれ以上のものであること．言語処理系を作成するという大枠から外れないこと．そして，課題を始める前に必ず田浦に連絡・相談をすること．締め切りぎりぎりになって，規定課題ができそうもないという理由で，思いつきのオリジナル課題を提案することは認めない．これはあくまで，規定課題はむしろ生ぬるすぎて退屈，と感じる人のためのオプションである．

4.2 規定課題

- オブジェクト指向スクリプト言語 Python をさらに小さくした言語 (mini-Python) の処理系 (インタプリタ) を構築する．標準的な言語仕様はこちらで決める．
- 構築した処理系に対して，いくつかこちらで用意した規定のテストを通過させる．正しいプログラムは正しく実行し，そうでないプログラムは正しくエラーを表示して終了する．
- mini-Python で書かれた，規定プログラムに対して性能測定を行う．

ここまでを最低ラインとして，このほかに，授業中に紹介する話題に関連した，または独自に考えた，自由課題を行う．たとえば以下のようなものが考えられる．

- 徹底して高速化をする
- 自動メモリ管理の実装．本来は言語処理系に必須の機能だが，作るのは大変なので標準課題では必須としていない．
- 処理系の機能拡張．ネットワーク計算など．
- ビッグなエンドユーザアプリケーションを，mini-Python を使って構築する．

つまり，最低ラインを早めに完成させて，残りを何か，オリジナルな課題のために当てる，というスケジュールで望んでほしい．

4.3 規則

4.3.1 チーム作業の仕方: ペアプログラミング

二人一組のチームを作ることを推奨しているが，これは労力を半分にしたり，ましてや片方の人がすべての課題を行うことを推奨しているのではない．あくまで，二人が共同で議論をし，疑問をぶつけ合い，役に立つ実践的知識を教えあう，などのコミュニケーションを図るのが，チームを作る目的である．それをする気がないのであれば一人でチームを作ればよい．このようなプロジェクトを行う過程では，授業でわざわざ教わるほどのことではないのだが，知らないと仕事の効率が変わる小技の類 (エディタやコマンドの使い方，情報検索のコツ) も多数ある．それは，チームを作って，些細なものを含め，問題を共有することで，覚えていくものである．

したがって最初から仕事を「分担」してはいけない．第○週までは○○君の担当，それ以降は××さんの担当，などとやってはいけない．ましてや相手を無視して，自分が眠いときは寝て，気の向いたときに勝手にコードを書き出す，というようなことはしてはならない．それならば最初からチームを作らないこと．

仕事の進め方の基本フォームはあくまで,「ひとつのディスプレイを二人で覗き込みながら」共同で，議論をしながらひとつのコードを書いていくことである．もちろん実際には一人がキーボードをたたき，もう一人は口しか出せないので，時々役割を入れ替える．これは少し変わったルールだが，意図があるので,「ばらばらにやったほうが早い」と思っても，可能な限り遵守すること．十分な議論を経た後で，似たようなコードをたくさん書かなくてはならないような場合などに，少し仕事を分割するのはかまわないが，相手の書いたコードを必ずチェックすること．相手のコードを部品として，ブラックボックスとして信じて使い続ける，といった分担の仕方はしないこと．

この方式は，大人数プロジェクトで仕事の分割をする際の慣習とは少し違う．大人数プロジェクトでは必然的に，各人の「担当部分」を決め，それらの間の呼び出し規約 (インタフェース) をしっかり決めたら，相手の担当部分の中身は気にしなくて良い，という風に仕事を分担するのが普通である．今回の演習のチームでも，二人ともがある程度プログラミングの中級者であれば，そうした方が効率が良いこともあり得る．[1] しかし，この演習の主目的は学ぶことであって，最終作品をいかに早く仕上げるかではない．ひとつのコードを眺めながら，良いコードの書き方，なぜこのコードは正しい・間違っているのか，デバッグの作戦，うまいツールの使い方，くだらない繰り返し仕事をぱっと片付ける小技，など，プログラミング中に発生するありとあらゆる問題をチームの相手と議論してほしい．実際には諸君の中の多くの人にとって初めて到達する規模のプロジェクトであるのだから，一人だと知らずにつまづいていたところを相手の知識に助けられたりして，この方式の方が，分担するよりも**結果的に仕事が早く進む**場合が多いと予想される．

4.3.2 開発環境

- 実装に用いる言語はC またはC++. それ以上の高級言語は使わない．授業でコードを見せて解説するときはC で行い，C++に関する知識は仮定しない．C++を知っていて，C での解説を適宜C++に置き換えて理解できる人は，C よりもだいぶ見通しの良いコードが書ける可能

[1] ただし，今回のプロジェクトはそれほど大きなプロジェクトではないので，中級者以上であれば一人で全部書いてしまうのが早いのかもしれない．

性があるので，積極的に使ってよい．

- 開発環境は自由．Linux でもよいし Windows でもよい．Windows で cygwin を用いても，Microsoft のコンパイラを用いても良い．ただし，授業での解説は，時々，以下のような環境の人に，ちょっといい話をする．
  - エディタ: Emacs (Windows では Meadow)
  - コンパイラ・デバッガ: gcc (g++), gdb

4.3.3 ソースコードの共有と提出方法

近山・田浦研のサーバに演習用のアカウントを作り，CVS というソースコード管理ツールを使ってソースコードを管理する．これは，チームのメンバ間で円滑にソースコードの修正をやりとりするため，毎週の進捗を把握するため，および課題提出の際のソースコード提出を円滑に行うために導入する．

CVS は複数人での，ソースコード (実際には任意のファイル) の共有を円滑におこうなうためのツールである．基本的な機能は，各自が作業用のソースコードのコピーを持ち，誰かが施した変更をほかの人へ簡単に伝播させることである．

近山・田浦研のサーバに各自のソースコードの置き場所 (CVS レポジトリという) が作成される．普段は自分の PC で，ソースコードの作業用コピーに対して編集作業を行う．ある程度修正がされたところで，その変更をレポジトリに書き込む (コミット)．それを行うと，もう一人のチームメンバも，変更内容を自分のソースコードのコピーに適用することができる．このようにして，お互いが変更したソースコードの内容を円滑に交換できる．

また，こちらも，各自の進捗状況をレポジトリを通して把握することができ，課題提出のためにソースコードの塊をメールに添付する必要もなくなる．

CVS を使うためには，レポジトリをおくマシンが必要で，それは近山・田浦研のサーバを用いる．そこへアクセスするために，ネットワークへの接続，ssh コマンド，cvs コマンドが利用可能である必要がある．

4.4 初回までの宿題

チーム作り: 各自，チームを組む相手を探しておくこと．演習初回の時間までに結成できているチームは，その場で承認する．あえて一人でやりたいという人の希望もそこで承認する．それ以外の人のチームはこちらで決めて，2 回目までに発表する．

wireless.i: 演習室 (241) で，ネットワーク (wireless.i) に接続して利用可能にしておく．ウェブの閲覧などができるかどうかをチェックしておく．

ssh: ssh コマンドが使えるようにしておく．Linux であればほぼ自動的に満たされいているはず．Windows の場合，貸し出されてる PC には cygwin というソフトウェアがインストールされているはずで，そこで ssh, cvs, gcc などが使えるはずである．

cvs: 同上.

C 言語処理系: C (C++を使うのであれば C++) 言語処理系一式が使えるようにしておく．Linux であれば自動的に満たされている (gcc コマンド) が，Windows で課題を遂行したい場合，前項の cygwin 一式の中でインストールされている gcc を使うか，または Microsoft のツール (Visual Studio) などをインストールしておく．Visual Studio の CD は，電子・情報系図書室で CD が借りられる．

CVS が初めてという人は，この機会に覚えることになる (後々，大きなプロジェクトを始める際にはきっと重宝する)．必要最低限のコマンドの説明は演習中にも行うが，本屋やウェブなどでおおまかな使い方を把握しておくとよい．

4.5 進度の予定

現在のところ，進度は以下のような予定．毎回説明を 1.5 時間程度行って，残りを実習の時間とする予定 (説明にかかる時間によって前後する)．

第 1 回	10/16	課題の説明．演習環境設定，ウォームアップ
第 2 回	10/23	字句の定義と字句解析器
第 3 回	10/30	構文の定義と構文解析器 (1)
第 4 回	11/6	構文の定義と構文解析器 (2)
第 5 回	11/13	評価・実行器 (1)
第 6 回	11/20	評価・実行器 (2)
第 7 回	11/27	全時間演習・質問
第 8 回	12/4	自動メモリ管理 (Garbage Collection)
第 9 回	12/11	高速化．言語の拡張
第 10 回	12/18	全時間演習・質問
第 11 回	1/15	自由課題発表会 (1)
第 12 回	1/22	自由課題発表会 (2)

表 1: 進度 (あくまで予定)

課題締め切り日は以下のとおりとしておく．ただし進捗を見ながら変更することがあるかもしれない．

前半レポート	11/26 (日)
発表会	1/15,22 (月) 演習時間中
最終レポート	2/5 (日)

表 2: イベントとレポート締め切り