第2章　概論的な話

「新しい字を発明しました。」「……どう読むのかね？」

吉田戦車「伝染るんです」

2.1　プログラミング言語を定義する

プログラム (*program*) とは，計算機に行わせるべき処理を記述した文字列である．世の中には様々な**プログラミング言語** (*programming language*) が存在する．（なお，以下では「言語」と言ったら，特に断らない限りプログラミング言語を意味する．）例えば，

```
int main(void) {
  printf("%d", 3);
  return 0;
}
```

は標準出力に”3” という文字列を表示する C 言語のプログラムであり，

```
let x = 3 in
  x + 2
```

は x が束縛された整数値 3 から整数値 5 を計算する OCaml のプログラムである．ソフトウェアを作る際には，様々な制約[1]を考慮してどの言語を用いるかを決定することになる．

この講義で扱うのは，これらの言語を使う方法ではなく，**作る**方法である．言語を作るにはどうすればよいかを説明する前に，プログラミング言語を作るとはどういうことかを説明しよう．プログラミング言語を作るとは，言語の**シンタックス（統語論）** (*syntax*) と**セマンティクス（意味論）** (*semantics*) を定義することである．[2]シンタックスは大まかに言えば言語

[1]例えば「解決すべき問題に適した言語機構があるか」「使えるライブラリは豊富にあるか」「その言語を使える開発者が十分にいるか」「その言語はバズっているか」「おれはこの言語が好きだ」「おれはこの言語が嫌いだ」などがある．

[2]シンタックスとセマンティクスは言語のためだけの概念ではなく，言語学や論理学など，言語を扱うより広い学問分野で生まれた概念である．プログラミング言語のように人工的に作られた言語に対し，日本語，英語，ポルトガル語などの人間が自然に使い始めた言語を**自然言語** (*natural language*) と呼ぶ．自然言語においてもシンタックスとセマンティクスが考えられるが，ほとんどの自然言語においてそれらはプログラミング言語のようにクリアに定まるものではない．また，自然言語においてはプログラミング言語では考慮されることの少ない「音韻」や「発話者等の言葉が発せられた文脈」等も言語を構成する重要な要素である．これらに興味がある場合は，自然言語処理や言語学の教員を誰か捕まえて読むべき本を教えてもらうと良いであろう．

の語彙と文法のことであり，どのような文字列が文法的に正しいプログラムと言えるかを決定する言語の要素である.[3]例えば，

```
int main(void) { int x = 3; return 0; }
```

は文法的に正しいC言語のプログラムであるが，

```
public class Main { static public void main(String[] argv) { int x = 3; return; } }
```

は（大体同じように振る舞う文法的に正しいJavaのプログラムであるが）文法的に正しいC言語のプログラムではない．また，

```
I hereby define a function named main.  This
function takes no arguments and returns an integer value.  The
function main first allocates a space on the stack enough to
accomodate an integer value and initializes it with an integer value
3.  Then, main returns 0 to the caller.
```

は（定義しようとしている関数の振る舞いが人間に正しく伝わるものの）やはり文法的に正しいC言語のプログラムではない．言語のシンタックスは**文脈自由文法**(*context-free grammar*)やその変種である**BNF**(*Backus-Naur form*)等で定義される．詳細は「プログラミング言語」の講義資料を復習されたい．

セマンティクスは大まかにはプログラムの意味のことである．例えば先程のC言語のプログラム

```
int main(void) { int x = 3; return 0; }
```

は「変数xに整数3を代入し0を返す」という計算を行うということが，C言語の仕様が定めているセマンティクスから分かる．

シンタックスは文脈自由文法やBNF等で定められると上に書いたが，それではセマンティクスはどのように定められるのだろうか．もっとも一般的な方法は自然言語による定義である．例えばC言語の言語仕様ドラフト [?] の6.5節にはC言語の式のセマンティクスが載っており，これを読み解くと文法に沿ったC言語のプログラムがどのような計算をすべきかが分かる．これに対し，数学的に厳密にセマンティクスを定義する方法もある．興味のある読者は五十嵐 [?] やWinskel [?] の教科書を読んでみるとよいだろう．

2.2 プログラミング言語の実装方式

言語のシンタックスをセマンティクスを定めると，一応プログラミング言語を作ったことになる．しかしながら，これだけでは計算機を使ってプログラムに実際に計算を行わせることはできない．計算機でプログラムを動かすには，セマンティクスに従ってプログラムを実行するためのソフトウェアが必要である．このようなソフトウェアを**言語処理系**(*implementation of a programming language*)と呼ぶ．

[3]このテキストでは「シンタックス」と「セマンティクス」という言葉が大量に出てくる．これらの言葉がテキストを理解する上での妨げになるようであれば，「シンタックス」を「文法」と読み替え，「セマンティクス」を「意味」と読み替えてもそんなに間違いではない．

言語処理系の例を見てみよう．コンピュータに GCC がインストールされているとしよう．その上で，先程のプログラム

```
int main(void) { int x = 3; return 0; }
```

を main.c という名前で保存し，シェルに

```
> gcc -o main main.c
```

と入力して実行すると，main というファイルができる．このファイルは実行可能なバイナリであり，シェルに

```
> ./main
```

と入力して実行することができる.（このプログラムは入出力を行わないので，プログラムを実行しても傍目には何も起こらない.）ここで gcc は，C 言語のシンタックスに沿ったプログラムを受け取り，そのプログラムと（C 言語のセマンティクスの上で）同等の動きをする別のプログラム（実行可能バイナリ）を生成した．このように，**入力プログラムと同じ動きをする別のプログラムを生成する**ことで，プログラミング言語を実装することができる．

別の例を見てみよう．OCaml 処理系がインストールされている計算機でシェルを立ち上げて

```
> ocaml
```

と入力してみよう．すると，インストールされている OCaml のバージョンが表示されたあとに，

```
#
```

という文字が表示されて入力待ち状態になる.（#のような入力待ち時に表示される文字を一般に**プロンプト** (*prompt*) と言う.）ここで

```
# let x = 3;;
```

と入力してエンターキーを押すと

```
val x : int = 3
#
```

と画面に表示され，再びプログラムの入力待ち状態になる．画面に表示された文字列は「整数型の値 3 が計算され，変数 x が計算結果に束縛された」ことを表現している．ocaml コマンドは (1) ユーザから文字列を受け取り，(2) その入力をプログラムとして解釈して，(3) 必要であれば自分自身の状態を変更して（ここでは変数 x を 3 という名前にして）(4) 計算結果を表示するというループを繰り返している．このように，**受け取ったプログラムを解釈してセマンティクスに従って実行し計算結果を出力する**プログラムによってプログラミング言語を実装することができる.（GCC が出力していたのは計算結果ではなく，計算を行う別のプログラム（実行可能バイナリ）であったことに注意されたい.）なお，上記の (1)–(4) のループには **read-eval-print ループ** (*read-eval-print loop*) という名前がついている．

なお，上記の 2 つの方式は，両方組み合わせて使われることがある．ocaml コマンドを

```
> ocaml -dinstr
```

というオプション付きで起動してみよう．すると，なにやらたくさん文字が表示された後にプロンプトが表示される．先ほどと同様に `let x = 3;;` と入力してエンターキーを押すと

```
const 3
push
acc 0
push
const "x"
push
getglobal Toploop!
getfield 1
appterm 2, 4
```

のような文字が表示される．（インストールされている OCaml のバージョンによって表示される情報は変わりうる．）この文字列は，OCaml バイトコードという，オリジナルの OCaml プログラムよりもマシン語に近い別のプログラミング言語で書かれたプログラムの文字列表現である．すなわち，`ocaml` コマンドは入力された式を，それと同じセマンティクスを持つバイトコードに内部で一旦変換し，それを解釈することで式の評価を行っている．このように，前者の実装方式（プログラムを変換する方式）と後者の実装方式（プログラムを解釈実行して結果を返す方式）とは相反するものではなく，組み合わせて使われることもある．

代表的な言語処理系の実装方法には**インタプリタ** (*interpreter*) と**コンパイラ** (*compiler*) がある．インタプリタとは，プログラムを解釈し，言語のセマンティクス従って計算を行い，その実行結果を返すソフトウェア，コンパイラとはプログラムを別のプログラムやバイナリに変換するソフトウェアである．（この変換のことを**コンパイル** (*compile*) という．）上述した GCC はコンパイラの例で，`ocaml` コマンドはコンパイラとインタプリタを組み合わせた例である．（このように，受け取ったプログラムをコンパイルしてから実行するプログラムを**インタラクティブコンパイラ** (*interactive compiler*) と呼ぶ．）

コンパイラ言語とインタプリタ言語 Web 上の情報やプログラミングに関する書籍の中には，主要な言語処理系がコンパイラとして実装されている言語をコンパイラ言語，主要な言語処理系がインタプリタとして実装されている言語をインタプリタ言語としたものが見られる．しかしながら，言語の定義と，その言語の処理系をどのように実装するかは別の問題であり，また上で見たようにコンパイラとインタプリタが一つの処理系に共存できないわけでもないので，言語がコンパイラ言語とインタプリタ言語に分類できるというわけではない．末永は個人的には「コンパイラ言語」「インタプリタ言語」という言葉には強い違和感を覚える．

2.3 中身の話

コンパイラやインタプリタの中身について簡単に説明してみよう．

2.3.1 字句解析・構文解析

インタプリタに対してもコンパイラに対しても，プログラムは文字列として与えられる．処理に先立って，この文字列の文法構造を解析する必要がある．例として先程挙げたOCamlのプログラム

```
let x = 3 in
  x + 2
```

を考える．このフェーズの目的は，上の（文字列として与えられた）プログラムから以下の**抽象構文木** (*abstract syntax tree*) を得ることである．インタプリタやコンパイラは，得られた抽象構文木を用いてそれぞれの処理を行う．

抽象構文木の構築は通常**字句解析** (*lexing*) と**構文解析** (*parsing*) という二段階の処理で行われる．字句解析は，与えられた文字列を（プログラミング言語における）単語列に切り分ける処理である．例えば，上記のプログラムを字句解析器に入力すると，

$$\mathbf{LET}, \mathbf{ID}(''x''), \mathbf{EQ}, \mathbf{INT}(3), \mathbf{IN}, \ldots$$

のようなトークン列が出力される．ここで，**LET**，**ID**($''x''$)，**EQ**, **INT**(3) 等はそれぞれプログラム中の `let`，x，$=$, 3 等の単語に対応するデータ型の値である．このようなプログラム言語の「単語」を**トークン** (*token*) と呼ぶ．トークンには別のデータが付帯していてもよい．上記の例では **ID** はプログラム中で用いる（変数名や型名などの）名前（**識別子** (*identifier*)）に対応するトークンであるが，このトークンにはプログラム中で出現した実際の名前（”fact” や”n” など）がデータとして付帯している．また，**INT** はプログラム中の整数定数に対応するトークンであるが，実際の整数値を付帯データとして持っている．

構文解析は，字句解析によって切り出された[4]トークン列を文法構造を表現した木構造（parse tree と呼ばれる）に組み立てる処理である．本講義では構文解析に使われているアルゴリズムの一部について解説する予定である．

本講義の後半では字句解析や構文解析のアルゴリズムを扱うが[5]自分の言語処理系で字句解析や構文解析を使う際にこれらのアルゴリズムを一から実装することはほとんどない．**パーザジェネレータ** (*parser generator*) というツールは BNF で書かれた文法の定義を入力として，構文解析器を出力する.[6]このツールを使えば字句解析器や構文解析器を比較的容易に作ることができる．よく使われているパーザジェネレータには Yacc や Bison がある．また，OCaml では Menhir と呼ばれるツールがよく用いられる．本講義ではこれらのツールの使い方も扱う.[7]

---

[4]字句解析によって文字列としてのプログラムからトークン列を得ることをよく「切り出す」と言う．切り出し感がどこから出てくるのかはよく分からない．

[5]ちなみに，これらのアルゴリズムを理解するには,「言語・オートマトン」で学んだ文字列に対する有限状態オートマトン，正則言語，文脈自由文法，文脈自由言語の理論を理解している必要がある．理解が行き届いていない場合は，各自復習されたい．

[6]パーザジェネレータは BNF という言語で書かれた文字列を入力として構文解析を行うプログラムを出力するので，コンパイラの一種である．

[7]ツールの使い方さえ分かれば字句解析や構文解析のアルゴリズムなど学ぶ必要はないではないかという（末永も学生時代に持った）問については，以下のいくつかの答えが考えられる．(1) 単位はほしいかね？(2) これ

2.3.2　インタプリタの中身

インタプリタにおいては，構文解析器によって生成された抽象構文木をなぞりながら，言語のセマンティクスにしたがって計算を行う．例えば，先に挙げた OCaml のプログラムを考えてみよう.（「プログラミング言語」で学んだ）OCaml のセマンティクスによれば，このプログラムは

1. 式 3 を評価して値 3 を得て，
2. 変数 x を得られた値 3 に束縛して，
3. in の後の式 x + 2 を評価する．

インタプリタにこの計算をさせるには，生成された構文木を**再帰的に**たどればよい．すなわち，式を計算する関数を eval_exp とすると，

1. Root ノードにぶら下がっている式 3 を eval_exp によって**再帰的に**計算し，計算結果 3 を得て，
2. Root ノードにぶら下がっている識別子を見て，変数 x を 3 に束縛し，
3. Root ノードにぶら下がっている別の式 x + 2 を eval_exp によって x **が** 3 **に束縛されていることを考慮しつつ再帰的に**計算する，

ようにすればよい．

インタプリタを実装する際の基本は構文木を上記のように再帰的にたどることである．ただし

- どのような順番で構文木をたどるべきか，
- 変数が束縛されている値をどのように保持するか，
- 高階関数等の複雑なデータをどのようなデータ構造で保持するか，

などを考える必要がある．本書ではこれらの詳細について実際に小さなインタプリタを作成しながら学ぶことになる．

らのツールは万能ではなく，人間が背景理論を理解した上でチューニングをしなければならない場合が多々ある．例えば，入力された文法が曖昧であった場合（すなわち，同一の文字列に対して 2 つ以上の抽象構文木がありえる場合）には，どちらが意図された構文木であるかをツールが理解することはできない．この場合には，パーザジェネレータが出力した警告を読んで，文法の曖昧性を解消する必要がある．そのためには構文解析の理論を理解する必要がある．(3) 字句解析・構文解析のアルゴリズムは，オートマトンの理論をとてもうまく用いて作られている．このアルゴリズムを学ぶことは，他のアルゴリズムを設計する場合にも有用である．

2.3.3 コンパイラの中身

コンパイラも構文木をたどりながら処理を行う点はインタプリタと共通である．ただし，インタプリタがセマンティクスに従って計算を行っていたのに対し，コンパイラは自身が計算をするのではなく,「セマンティクスに従って計算を行う変換先言語のプログラム」を生成する．

再び先程のプログラム `main.c` を gcc を用いてコンパイルしてみよう．ただし，今度は以下のように gcc を起動してみよう．

```
> gcc -S -o main.s main.c
```

gcc は上のように`-S` オプション付きで起動されると，入力プログラム（ここでは `main.c`）を実行可能バイナリではなく，**アセンブリ** (*assembly*)，すなわち実行可能バイナリを人間が読みやすい文字列表現にしたファイルにコンパイルする．生成されたアセンブリは`-o` の後に指定されたファイル（ここでは `main.s`）に記録される．

Exercise 2.3.1 自分の手元の環境で上記のコマンドを実行し，生成された `main.s` の中身を見てみよ.（この課題については自分でやればよい．提出は不要である.）

`main.s` 内容を理解する必要は現時点では無いのだが，重要なのはアセンブリも**アセンブリ言語** (*assembly language*) という言語で書かれたプログラムの一種であるということである．アセンブリ言語は計算機の命令を一つ一つ指定することで行うべき計算を記述するための言語で，様々な計算機アーキテクチャに固有のアセンブリ言語が定義されている．

アセンブリは**アセンブラ** (*assembler*) というコンパイラを用いて実行可能バイナリにコンパイルすることができる．gcc は実はアセンブラとしての機能も持っており，拡張子.s を持つファイルをアセンブリと仮定する．したがって，生成された `main.s` を

```
> gcc -o main main.s
```

のように gcc に与えれば，実行可能バイナリ main が得られ

```
> ./main
```

と実行することができる．

少し長くなったが，ここでの要点はインタプリタはプログラムを受け取って，言語のセマンティクスに従って計算を行い，計算結果を返すのに対し，コンパイラはプログラムを受け取って，言語のセマンティクスに従って計算を行う別のプログラムを返すという点である．変換元の言語（上の例では C 言語）を**ソース言語** (*source language*) と呼び，変換先の言語（上の例ではアセンブリ言語）を**ターゲット言語** (*target language*) と呼ぶ.[8]

したがって，コンパイラを実装するためには，構文解析器が作った構文木をたどりながら，入力されたプログラムに対応するターゲット言語のプログラムを生成すれば良い．しかし，ほとんどのコンパイラはいきなりターゲット言語のプログラムを生成するのではなく，間に

[8]本書では，実行可能バイナリもフォーマットも広い意味ではプログラミング言語であると考えている．したがって，C プログラムを入力として受け取り実行可能バイナリを生成する gcc は，ソース言語が C 言語，ターゲット言語が「実行可能バイナリを記述するためのプログラミング言語」のコンパイラであると捉える．

いくつかの**中間言語** (*intermediate language*) をはさみながら変換を行う．また，各中間言語のレベルにおいて，プログラム解析を行い，生成されるプログラムの実行効率を上げるための変換（**最適化** (*optimization*) と呼ばれる）を行ったり，プログラム中に存在するバグの発見を試みたりすることが多い．これらについては講義中で扱う．

2.4 これ以降の構成

以上，本書で扱うトピックを概論的に説明した．これ以降では，それぞれのトピックを掘り下げて説明する．本書のこれ以降の構成は以下の通りである．

- まず OCaml のサブセットを解釈実行するインタプリタの実装を説明する．この章を一通りやれば，言語を設計し実装するための基礎が身につくはずである．
- 次に静的プログラム解析という，プログラムを実行することなくプログラムの実行に関する情報を得るための手法を解説する．こういうと結構おどろおどろしいが，実際には OCaml の型推論のメカニズムを説明する．型推論は，プログラムの実行前に実行時型エラーが無いかどうかを解析するという意味で，静的なプログラム解析である．
- その後方向性が少し変わって，コンパイラの実装について簡単に解説する．ここでは，**??**章で定義した言語（のさらにサブセット）をコンパイラとして実装してみる．同じ言語をインタプリタとしてもコンパイラとしても実装してみることで，両者の違いと利害得失が分かるようになるであろう．
- 最後にここまでブラックボックスとして扱ってきた，字句解析器と構文解析器と，それらを生成するツールの中身について解説する．