システム
ソフトウェア
特論
2019

久野　靖
電気通信大学

2019.1.9 改訂

目 次

＃1　プログラミング言語とは

1.0　本科目の目的/内容/進め方

本科目「システムソフトウェア特論 (Advanced Topics on System Software)」の目的は、プログラミング言語処理系およびそれに関連のあるシステムソフトウェアについて、実例を中心に学ぶことです。その進め方としては、Java 言語を用いて実際にさまざまな実装を構築し、具体的な概念を身に付けて頂くようにします。

毎回、授業に際して例題を配布しますので、実際にそれを動かし、また改良してみて頂きます。実習環境としては西 10 の CED を使用しますが、Java 言語が動けばどのような環境でも実習できますので、自分の PC なども含めて各自でうまく進めてください。

本科目では何らかのプログラミング言語によるプログラミングができることを前提とします。実際に使う言語は Java 言語になりますが、Java 言語固有の部分については授業内で説明するようにします。Java 言語の参考書が必要な場合は「久野禎子, 久野　靖, Java によるプログラミング入門 第 2 版, 共立, 2011」をおすすめします。久野まで連絡いただければ著者割引価格で提供できます。

1.1　プログラミング言語とは

プログラミング言語 (programming languages) とは何かについて、さまざまな定義が可能だと思いますが、ここでは次のように定めます。

- プログラムを記述するために設計された人工言語

ここでさらにプログラムとは何か、人工言語とは何かが問題になるわけですが、人工言語の方はあとの回で厳密に取り上げることになりますのでそれまで保留しましょう。

プログラムは「コンピュータが実行するべき動作を記述したもの」くらいでいいでしょうか。コンピュータがどんなものかとかコンピュータが動作するとかはだいたい皆様知っているということで。「記述した」というのはつまり「外部化した」「文字などで表現された」という意味になります。外部化され表現されていないものはコンピュータに与えたり他人が読んで検討したりできませんから、これは必要最低限な定義といってよいと思います。

あと「設計された」ですが、設計だけで動かないでいいのかという突っ込みもあるかと思います。これは「設計されただけで実装されていない」プログラミング言語は沢山ありますから、無問題です。

1.2　様々なプログラミングパラダイム

1.2.1　プログラミングパラダイムとは

プログラミングパラダイム programming paradigm というのは、少し訳しにくいのですが (なのでカタカナのまま使われることが多い)、プログラミング言語の「枠組み」というか「方式」というふうに考えていただければいいと思います。

皆様の中で複数のプログラミング言語を学んだことがあり、ある言語で学んだことは他の言語でも使えるので 2 つ目からは学ぶのが容易である、というふうに思っている方もいると思います。これはつまり、言語の見た目は違っていても、パラダイムは共通なので、1 つ目の言語で身につけたことが 2 つう目の言語でもそのまま通用するから、という風に考えればよいでしょう。

しかし実は、パラダイムの異なる言語だとかなりその「共通」が減るので、学ぶのが大変だったりします。ではなんでそんなに大変なのに違うパラダイムの言語があるの、ということになりますが、それは記述したい題材や対象によって、パラダイムとの相性があるから、ということになるでしょうか (もちろん書く人との側の好みや相性も問題になります)。

では具体的に、どのようなプログラミングパラダイムが存在するのでしょう。網羅的はちょっと大変なので、思い付くままにいくつか挙げる程度にします。

- チューリングマシン
- ランダムアクセスマシン (RAM)、機械語
- 手続き型 (命令型) 言語
- 関数型言語
- 論理型言語

チューリングマシンは計算モデルですがプログラムを書けますので入れています。**RAM**(random access machine) というのはこれも計算モデルですが、普通の CPU の機械語がだいたいこれに相当する動作になります。その後の 3 つはこれから説明していきます。

1.2.2　論理型言語

論理型言語 (logic programming languages) というのは Prolog やその仲間に相当する言語で、述語論理に基づいています。ここでは Prolog を例にどのような感じかだけ見ていただきます。

Prolog ではさまざな事柄を**述語** (predicate) として表します。「`human(socrates)`」というのは「ソクラテスは人間である」を述語として表記しています。この場合、`human` が述語名、`socrates` は引数となります。引数には変数 (大文字で始まる名前) を指定することもできます (用例はすぐ後で)。

Prolog では述語の集まりを**ホーン節** (Horn clause) と呼ばれる次の 3 種類の形に限定してこれを並べることでプログラムを構成します。

q.
q :- p_1, p_2, ..., p_n.
:- p_1, p_2, ..., p_n.

1 番目の形は**事実** (fact) と呼ばれ、述語 q が「真である」ことを述べています。2 番目の形は**頭部** (head) と**本体** (body) から成る**規則** (rule) で、述語 $p_1 \cdots p_n$ がすべて真であるなら、節 p も真である、ということを述べています。3 番目の形は**目標** (goal) と呼ばれ、$p_1 \cdots p_n$ がすべて真であることを示すという動作を実行開始します。以下では、まず事実と規則を複数与え、そのあと目標を打ち込んでその真偽を確認します。

例を見てみます。この例では、事実は「ソクラテスは人間である」、規則は「X が人間なら、X は過ちを侵す」となっています。このように大文字で始まる X は**変数** (variable) であり、さまざまなものが入ります (ただし同じ名前の変数には同じものが入る必要がある)。

```
% gprolog
GNU Prolog 1.4.4 (64 bits)
| ?- [user]. ←直接プログラム入力
compiling user for byte code...
human(socrates).          ←事実
fallible(X) :- human(X). ←規則
user compiled...
yes
```

```
| ?- fallible(X).    ←目標を与える
X = socrates         ← X として socrated をあてはめると
yes                  ←成立することがわかった
```

最後に「X が fallible であるような X が存在するか」を目標として実行すると、X に socrates をあてはめた場合に成立するという答えが得られます。この例では「3 段論法」(「A である」「A ならば B である」から「B である」を導く) を実現しています。

もう少し意味のある例として、リストの連結を扱います。Prolog ではリストは [...] のように角かっこに囲まれた値の列ですが、[X|T] のように書いた場合は先頭の要素が X、残りが T という意味になります。では例を見てみましょう。

```
% gprolog
GNU Prolog 1.4.4 (64 bits)
| ?- [user]. ←直接プログラム入力
compiling...
app([], X, X).  ← 空リストと X を連結すると X
app([A|X], Y, [A|Z]) :- app(X, Y, Z).
 ↑ X と Y を連結して Z ならば、[A|X] と Y を連結すると [A|Z]
user compiled...
yes
| ?- app([1,2], [3, 4, 5], L).  ← 123 と 45 を連結すると何?
L = [1,2,3,4,5]
yes
| ?- app([1, 2], L, [1, 2, 3, 4]). ← 12 と何を連結すると 1234?
L = [3,4]
yes
| ?- app(L, [3, 4], [1, 2, 3, 4]). ←何と 34 を連結すると 1234?
L = [1,2] ? ;     ← 12 だけどまだあるかも (; でさらに探す)
no                ←やっぱりそれでおわりだった

| ?- app(X, Y, [1, 2, 3]). ← X と Y を連結すると 123 になる X と Y は?
X = []
Y = [1,2,3] ? ;   ← 1 つ目
X = [1]
Y = [2,3] ? ;     ← 2 つ目
X = [1,2]
Y = [3] ? ;       ← 3 つ目
X = [1,2,3]
Y = [] ? ;        ← 4 つ目
no                ←以上でおわり
| ?-
```

このように、Prolog では述語のどの位置を変数、どの位置を値にしてもそれを充足するような変数の値を探そうとするので、プログラムの書き方によってはさまざまな使い方ができます。

1.2.3 手続き型言語

コンピュータの命令は基本的にメモリからデータを取り出し、演算して、結果をメモリに格納します。そこで、この動作を自然な形で記述するため、メモリ上の領域を「変数」「配列」などに対応させ、「代入」操作によって値を書き換えて行くようなプログラミング言語が作られました。

たとえば次のような記述があったとします。

```
x = x + 1
```

これは等式ではなく、変数 x(実際にはどこかのメモリ番地に対応) の値を取り出し、1 と加算する演算を実行し、結果を x に格納 (代入) する、という内容なわけです。このような演算ど代入 (とその実行の流れを制御する機能) により、プログラムの実行が進んでいくわけです。

最初の**高水準言語** (特定の CPU に依存しない形で記述できるようなプログラミング言語) である Fortran はそのようなものでしたし、現在使われている言語の多くもこの流れを汲んでいます。このような言語を、演算や代入などの命令を順次実行していくという意味で**命令型言語** (imperative language)、または手続きを順番に実行していくという意味で**手続き型言語** (procedural language) と呼びます。

本科目で扱う Java も手続き型言語の仲間ですが、さらにオブジェクト指向の機能が追加されています。この点については Java の話題の中で説明していきます。

1.2.4　関数型言語

関数型言語 (functional language) とは、関数定義とその適用に基づいて実行が進むようなプログラミング言語です。そこでとくに重要なのが、**副作用** (side effect) の排除ということです。副作用とは、処理を実行したときにその処理結果に加えて、コンピュータ内に状態の変化をもたらすことを言います。

最も典型的な副作用は、手続きの中での (手続きの実行が終わった後も残っている) 変数への値の書き込みです。ですから、手続き型言語の実行は副作用の塊となりがちです。

副作用の何が問題なのでしょうか。副作用があると、手続きを 1 回実行したときともう 1 回実行したときで結果が異なる可能性があります。また、複数の手続きを並行して実行するとき、互いに干渉する可能性があります。これらのことがあると、プログラムを高速に実行する上で制約となることがあります。

これに対し、関数型言語、とくに**純粋関数型言語** (pure functional language) では変数の代入をはじめ一切の副作用がないため、並列実行がしやすく、干渉による誤りが置きにくいという利点が得られます。

イメージが湧かないと思いますので、ここで C 言語を使って純粋関数型のようなスタイルでプログラムを書いてみましょう。`main()` は次のようなものとします。

```
#include <stdio.h>
#include <stdlib.h>
int f(int);
void main(int argc, char *argv[]) {
  printf("%d\n", f(atoi(argv[1])));
}
```

ここで関数 `f()` が実際の計算をしますが、その中は「変数への代入を一切行わない」ように書くことにします (上記の `main()` もそうなっていますね)。変数の初期化は行ってよいことにします。

たとえば、階乗の計算をおこなう場合は次のように書けます。

```
int f(int n) {
  if(n < 1) { return 1; }
  else      { return n * f(n-1); }
}
```

演習 1-1 次の計算を C 言語の「関数型」スタイルで書いてみよ。

a. N を与えて $\sum_{i=1}^{N} i$
b. N を与えて $\sum_{i=1}^{N} i^2$
c. N を与えて $fib(N)$ ただし $fib(0) = fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)$ $(n \geq 2)$。
d. その他の「関数型」スタイルで書くのによいと思う任意の計算。

1.3 Java 言語の特徴と特性

1.3.1 Java とオブジェクト指向

手続き型言語はもともと、代入、式、制御構造とそれらを集めて呼び出せるようにした手続き (関数、サブルーチン、メソッドなどとも呼ぶ) が基本要素でしたが、プログラムが大きく複雑になるにつれて限界が明らかになりました。具体的には、複数の手続きが使うデータの定義をグローバル変数にすると、多数のグローバル変数が必要となり、それらのどれかを誰かが壊したためにプログラムが破綻するという問題が頻繁に起こるようになったというのが主なものです。

そこで、複数の手続きとそれらが共通に使うデータをひとまとめにして扱う**モジュール**と呼ばれる機能が作られるようになりました。モジュール内のデータはそこに所属する手続きだけからしかアクセスできないので、「誰かが壊す」ことは避けられます。

そのうち、1 つのモジュールが 1 つの「もの」に対応するデータと機能を持つとプログラムが作りやすいことが知られるようになり、モジュールは「そのモジュールのデータ」を複数つくり出せるようになりました。このような「もの」に対応するデータと手続きの組のことを**オブジェクト** (object)、そのようなデータや手続きを定義する構文のことを**クラス** (class)、これらの機能を提供する言語を**オブジェクト指向言語** (object-oriented language) と呼びます。Java もそのような言語の 1 つです。[1]

クラス方式のオブジェクト指向言語では、クラスから個々のオブジェクトを作り出しますが、これらのオブジェクトのことをそのクラスの**インスタンス** (instance) とも呼びます。これは、クラスそれ自身もオブジェクトとしての機能を持つことから、それと区別するためです。

1.3.2 Java 言語入門

本科目ではプログラムを書くのに Java 言語を使うのでここで簡単に紹介します。詳しい事項については必要になるつど追加します。

最初の例題を示しましょう。これは 2 つの数値を入力し、その和を出力するというものです。

```
import java.util.*;
public class Sam11 {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("s1> "); String s1 = sc.nextLine();
    System.out.print("s2> "); String s2 = sc.nextLine();
    double d1 = Double.parseDouble(s1);
    double d2 = Double.parseDouble(s2);
    System.out.println("sum: " + (d1+d2));
  }
}
```

説明は次の通り。

1. `import` はさまざまなライブラリの取り込みを指示する。ここでは `Scanner` クラスが `java.util` の中にあるためこの指定をしている。

[1] クラスを持たないオブジェクト指向言語もあります。

2. Java ではすべてのプログラムはクラスという単位で記述する。クラスの構文は「class クラス名 { … }」。クラス名は英字で始まる英数字の並びで、大文字から始める習慣。
3. 実行するプログラムの場合、クラス内にメソッド main() が含まれる必要があり、その先頭から実行が開始される。main() は public static メソッドであり、[2] 返値が void、引数が String[] である必要。
4. Java ではクラスからインスタンスを作るときには「new クラス名 (…)」という構文によります。この構文により、インスタンスが作られた後、そのクラスに定義された**コンストラクタ** (constructor) と呼ばれる特別なメソッドが呼び出されてインスタンスの初期化を行い、その後でインスタンスが返されます。かっこ内に指定するのはコンストラクタに渡すパラメタです。そしてクラス名は型でもあり、あるクラスのインスタンスを格納するにはそのクラス型の変数を使います。Scanner は行単位での入力などを可能にするオブジェクトあり、コンストラクタで入力に使うストリームを渡して作ります。System.in は標準入力から読み込むストリームであり、ここではそれを渡して作っています。
5. System.in には InputStream オブジェクト (クラス InputStream のインスタンス)、System.out と System.error には PrintStream オブジェクトが格納されており、それぞれ標準入力、標準出力、標準エラー出力につながっている。PrintStream オブジェクトはメソッド print()、println()、printf() を持ち、前 2 つは文字列を単に出力する (println() はそのあと改行も出力)。printf() は書式文字列と 0〜N 個の値を受け取り、機能としては C 言語の同名のものと同様である。
6. ここではプロンプトを出力し、次に Scanner オブジェクトのメソッド nextLine() により 1 行ぶん入力し、その文字列を変数に格納することを 2 回おこなう (2 行ぶん入力する)。
7. クラス Double のクラスメソッド parseDouble() は文字列を受け取ってその文字列が表す実数値 (double 型の値) を返す。これらをいずれも double 型の変数 d1、d2 に格納。
8. 本体最後の行で d1 + d2 を計算し、それを文字列と連結し (Java では「＋」は片方が文字列のとき他方も文字列に変換した上で文字列連結をおこなう)、出力する。

これを実行するには、まずこのソースを Sam11.java というファイルに格納します。Java ではソースプログラムのファイル名 (の拡張子を除いた部分) とクラス名とが一致する必要があるので、ファイル名はかならずこの名前にする必要があります。そして、次の 2 つのコマンドを使用します。

- 「javac ソースファイル名」 — ソースファイルをコンパイルし、.class ファイルを生成する。
- 「java クラス名」 — 「クラス名.class」というファイルを探してきてその中の main() から実行を開始する。

実行のようすを示します。

```
% javac Sam11.java
% java Sam11
s1> 1.5
s2> 2.7
sum: 4.2
%
```

演習 1-1　上の例題を打ち込んでそのまま動かせ。動いたら、次のことをやってみよ。

[2] public はクラス外から参照され得ることを示す。static はクラスメソッド (オブジェクトを作らなくても呼び出せるメソッド — オブジェクトについては後述) を示す。

a. 加算の代わりに減算、乗算、除算、剰余をやってみよ。演算子は C 言語と同じである。
b. 上の例は実数だったが、整数でやってみよ。整数型は `int`、文字列から整数への変換は `Integer.parseInt()` である。
c. 整数のかわりに長精度整数でやってみよ。長精度整数型は `long`、文字列から長精度整数への変換は `Long.parseLong()` である。整数では計算できないが長精度整数ではできる例を確認すること。

1.3.3 値とオブジェクト

Java では (C++もそうですが)「値」と「オブジェクト」を区別しています。そのあたりは次のようになっています。

- 値は四則演算、比較演算などが適用できる。値の型としては `byte`、`char`、`short`、`int`、`long`、`float`、`double`、`boolean` がある。それぞれ 8 ビット整数、16 ビットの文字コード値、16 ビット整数、32 ビット整数、64 ビット整数、32 ビット実数、64 ビット実数、論理値である。
- オブジェクトはクラス名 (大文字で始まる) で表される型を持ち、演算によってできる操作は代入のみである。それ以外の操作はすべてメソッド呼び出し「オブジェクト . メソッド名 (引数,...)」によって行う。
- オブジェクトの値はすべて**参照値** (reference value)、つまりオブジェクトが置かれたメモリ上の番地に対応する値である。オブジェクトの値を代入する場合、次に述べる値の変換の場合を除けば、参照値をそのままコピーし代入する。つまりコピー元とコピー先で同じオブジェクトを共有する。
- 8 種類ある値の型にはそれぞれ、その値を保持するようなオブジェクトのクラス `Byte`、`Character`、`Short`、`Integer`、`Long`、`Float`、`Double`、`Boolean` が対応して存在している。これらのクラスのことを (値を囲むということで) **包囲クラス** (wrapper class) と呼ぶ。そして、値と対応する包囲クラスオブジェクトの間では自動的な変換が行える。

```
Double d3 = 3.141;// double 値から Double オブジェクト
double d4 = d3;   // Double オブジェクトから double 値
```

なお、このような「値とオブジェクトの区別」は今日の言語では少くなっています。というのは、C++や Java が設計された当時は「オブジェクトはメモリに保存するので遅い」「だから速度が必要な値は別扱いしよう」という考えで言語が設計されていたわけですが、その後の研究の進展で「言語仕様上はオブジェクトでも実際に動くコートは値と同等」が実現できるようになり、人間が両方を使い分ける面倒は不要だと分かったためです。

1.3.4 クラスによるオブジェクト定義とその利用

値とオブジェクトについて分かったところで、次はオブジェクトについてもっと考えてみましょう。Java 言語の設計上の特徴として、「演算子はすべて値にのみ適用し、オブジェクトはすべてメソッド呼び出しによって操作する」という点があります。ただし例外は代入の「=」でして、これは左辺のオブジェクトの値を右辺の変数に代入するという形で統一されています。先に述べたように、実際に代入されるのは「参照値」なので、代入はオブジェクトの操作ではないと考えるのがよいかも知れません

ではオブジェクトを操作するメソッドとは何でしょうか。これは結局「手続き」「C 言語でいう関数」に相当します。その呼び出し方は次のようになります。

```
クラス名 . メソッド名 (パラメタ…)                    …(1)
オブジェクトを表す式 . メソッド名 (パラメタ…)    …(2)
```

Java ではすべての要素 (変数、メソッド) はどれかのクラスの中に入れる必要があるので、その外から参照するときは上記いずれかの書き方を取ります。このうち (1) はクラスの中で「オブジェクトを作らずに」使えるメソッド (static メソッド、クラスメソッド) です。main もそうなっていました。こちらは C 言語の関数に近いです。

もう 1 つの (2) は、クラスからオブジェクト (インスタンス) を作り出し、それに対して呼び出します。これをインスタンスメソッドと呼びます。たとえば次の例を見てください (この例は動かしません)。

```
class Dog {
  String name;
  double speed = 0.0;
  public Dog(String n) { name = n; }
  public String toString() { return "name="+name+" speed="+speed; }
  public String getName() { return name; }
  public double getSpeed() { return speed; }
  public void addSpeed(double d) { speed += d; }
}
```

このコードは Dog というクラスを定義します。このクラスは「犬」を複数扱う (?) アプリケーションで利用するもので、このクラスでは犬それぞれの名前と走っている速さを保持します。これらのデータは変数 name と speed に保持されます。

つまり、この 2 つの変数は「犬の 1 個体ごとに」つまり「犬のオブジェクト (インスタンス)」ごとに 1 セットずつあることになります。このような変数をインスタンス変数と呼びます。

さて、このクラスからインスタンスを作り出すときには、new 演算子を使って次のようにします。

```
Dog d1 = new Dog("pochi");
Dog d2 = new Dog("tama");
```

インスタンスを作ろうとすると、そのクラスに定義されたコンストラクタと呼ばれる特別なメソッドが動いてインスタンスを初期化します。コンストラクタはクラス名と同名なのでそれと分かります。Dog のコンストラクタでは名前の文字列を受け取るので、new では文字列を渡します。そしてコンストラクタでは name をその文字列で初期化します。speed の方は変数定義時に初期値を指定したのでそれがそのまま使われます。

残りのメソッドはすべてインスタンスメソッドで、次の例のように使います。

```
d1.toString() --- pochi の名前と速さの文字列を返す
d2.getName()  --- tama の名前を返す
d1.getSpeed() --- pochi の速さを返す
d2.addSpeed(2.0) --- tama の速さを 2.0 だけ増す
```

つまり、これらのメソッドの中で参照している name、speed はいずれもインスタンス変数なので、メソッド呼び出し時に指定したインスタンス (d1 や d2) に付随している変数がアクセスされる、ということです。これにより、多数の犬を扱うなかで犬の名前と速さの対応が分からなくなったりせずに済むわけです。

なお、この (2) のような呼び出し方のことを「オブジェクトに対して〜して欲しいと呼びかける」という類推から「メッセージ送信記法」と呼びます。

1.3.5 例題: Toknizer

プログラミング言語のソースファイルは「名前」とか「演算子」などの「かたまり」から成っています。このかたまりのことを「トークン (token)」と呼びます。言語処理系がソースコードを読むときも、このトークン単位で読むことが定石です。

今回は簡単のため、先に出て来た Scanner のメソッド next() によって読み出される 1 かたまりずつをトークンということにしましょう。具体的には、空白や改行で区切られた「白くない並び」がトークンになります。そして、次のような使い方ができるオブジェクトを作ります。

```
Toknizer tok = new Toknizer("t1.txt");  --- ファイル名指定して生成
tok.curTok()  ---  「現在位置」のトークンを文字列として返す
tok.isEOF()   ---  「終端」かどうかを論理値で返す
tok.fwd()     ---  「現在位置」を 1 つ進める
```

なお、終端に来たときは現在のトークンは"$"という文字列だということにします。

では、クラス Toknizer のソースを見てみましょう。

```
class Toknizer {
  Scanner sc;
  String tok;
  boolean eof = false;
  public Toknizer(String fname) throws Exception {
    sc = new Scanner(new FileInputStream(fname)); fwd();
  }
  public boolean isEof() { return eof; }
  public String curTok() { return tok; }
  public void fwd() {
    if(!eof && sc.hasNext()) { tok = sc.next(); }
    else { eof = true; tok = "$"; }
  }
}
```

読み取るための Scanner を保持する sc、現在のトークンを保持する tok、そして終わりかどうかを保持する eof がインスタンス変数です。コンストラクタではソースを読み取るファイル名を指定します。[3]

そして、そのファイルから読み込む FileInputStream を生成し、それを渡して Scanner を生成します。次に、最初のトークンを読むことで初期化を完了しますが、それは後で定義しているメソッド fwd() を呼びます。このように、1 つのインスタンスメソッド (コンストラクタも含まれます) の中から同じインスタンスのインスタンスメソッドを呼ぶときはメッセージ送信記法は不要でメソッド名だけで呼べます。[4]

isEof() は eof の値、curTok() は tok の値をそれぞれ返すだけです。ということは、「肝」は最後のメソッド fwd() ですね。これは、まだ eof でなくて、かつ Scanner から次のトークンが読み取れるなら、そのトークンを tok にいれます。それ以外の場合は終わりですから、eof は true を入れ、tok には"$"を入れます。

では、これを呼び出す側を見てみましょう。1 つのファイルにまとめて入れられるので、実際にはこの部分の末尾に上の Toknizer を入れてコンパイルします。

[3] thorws Exception という指定は、ファイル初期化エラーがあったときにそのエラーを呼び出し側に伝えるための指定ですが、詳しくは後日に説明します。

[4] ついでにですが、クラスで定義しているクラスメソッドもメソッド名だけで呼べます。

```
import java.util.*;
import java.io.*;

public class Sam12 {
  public static void main(String[] args) throws Exception {
    Toknizer tok = new Toknizer(args[0]);
    Scanner sc = new Scanner(System.in);
    while(true) {
      System.out.print("> ");
      String line = sc.nextLine();
      if(line.equals("p")) {
        System.out.println(tok.curTok());
      } else if(line.equals("e")) {
        System.out.println(tok.isEof());
      } else if(line.equals("f")) {
        tok.fwd(); System.out.println(tok.curTok());
      } else if(line.equals("q")) {
        System.exit(0);
      }
    }
  }
}
```

main は前と同様です。[5] そして、コマンド引数で指定したファイルを渡して Toknizer を作り、変数 tok に入れます。またキーボードから読み込む Scanner も前の例題と同様に作ります。

以後無限ループですが、プロンプトを読み込み、1 行文字列を読み込み、それが何であるかで動作を振り分けています。「文字列が等しいかどうか」は (文字列もオブジェクトなので)「==」は使えず、メソッド equals() を使う必要があります。どんなコマンドを入力したら何をするかは見れば分かりますね。[6]

演習 1-2 この演習問題を打ち込んでそのまま動かせ。適当なファイルを指定して動作を確認せよ。できたら、次のような変更を施してみよ。テスト用の main() も対応して適宜直すこと。

a. 間違って fwd() したときに 1 つ前に戻る back() を追加する。
b. 読むだけでなく新たなトークンを (次に読むものとして) 押し込む push(s) を追加する。引数はトークン文字列。
c. その時点からファイルを別のファイルに切替える open(s) を追加する。引数はファイル名。
d. その他、何か「あったらいいかも」という機能を追加。

1.4　課題 1A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

[5] ただし Toknizer を生成するときにエラーが帰ってくる可能性があるので、ここも throws Exception の指定が必要です。

[6] System.exit() は C ライブラリでいう exit() と同じです。

- タイトル – 「システムソフトウェア特論 課題＃ 1」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. プログラミングは好き/得意ですか、苦手ですか。それはなぜですか。どういうところが特にそう思う?

  Q2. さまざまなプログラミング言語があるということについてどのように考えていますか。

  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

1.5 付録: API ドキュメント

Java 言語は豊富な標準ライブラリでも知られています。ライブラリのドキュメントも整備されており、以下のサイトで見ることができます。この文書は **API ドキュメント**と呼ばれています。[7]

`http://docs.oracle.com/javase/8/docs/api/`

ライブラリはクラスの集まりであり、パッケージ (`java.util.*`とか `java.lang.*`とか) の単位でグループ化されています。[8]

図 1.1 は API ドキュメントで `java.lang.String` のところを表示したものです。特定クラスを表示するのは、左下の全クラスの ABC 順の一覧から選ぶか、まず左上のパッケージ一覧でパッケージを選び、次に左下でそのパッケージのクラス一覧から選択します。`String` の例で分かるように、API ドキュメントではクラスごとにどのようなクラスかという説明と、それに続いてそのクラスが持つメソッドやその機能の説明が書かれています。

課題 1-3 次のクラスの API ドキュメントを表示し、どのような機能が含まれているかチェックせよ。続いて、以下のクラスのところを確認し、実際に例題プログラムを手直しし、その機能の動作を確認してみよ。

a. `java.lang.String` — 文字列オブジェクト。文字列に対するさまざまな操作がおこなえる。
b. `java.util.Scanner` — 入力読み込みオブジェクト。さまざまな形での入力がおこなえる。
c. `java.lang.Integer`、`java.lang.Double` — 整数や実数の包囲オブジェクト。これらの値に対する加工がおこなえる。
d. `java.io.PrintStream` — `System.out` に格納されている出力オブジェクト。さまざまな出力機能を提供している。
e. その他自分で興味を持ったクラス。

[7] API は applicaiton programming interface の略。
[8] `java.lang.*`は言語の基本機能を実現するクラス群なので、自動的に `import` されることになっている。

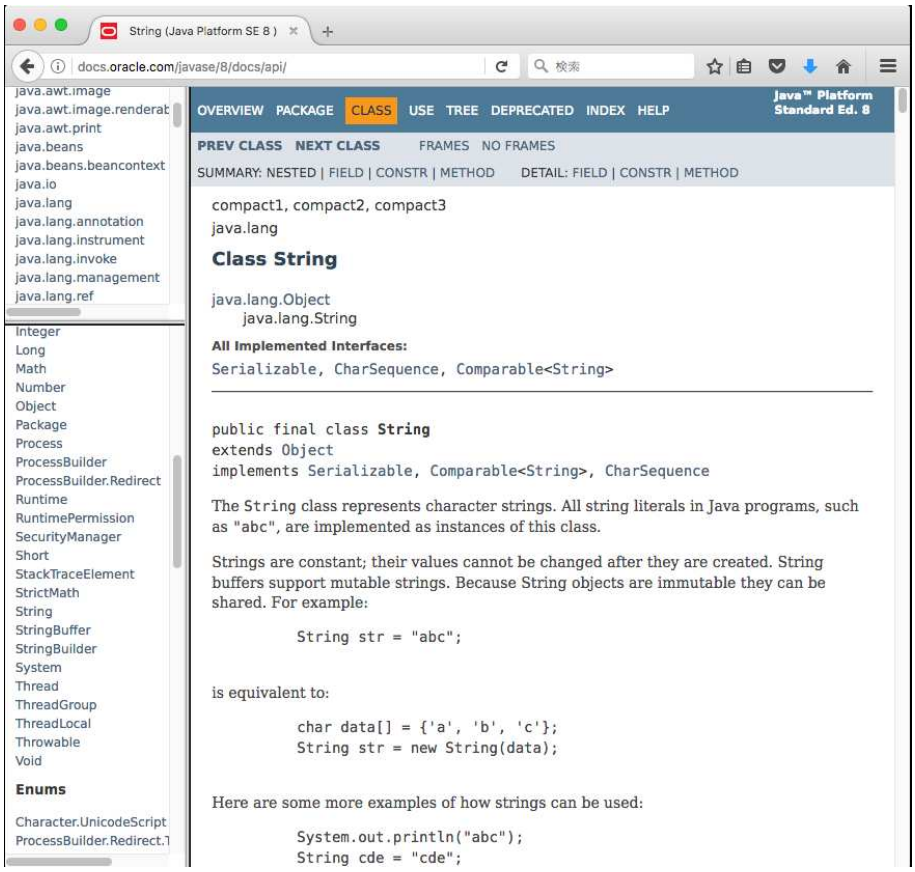

図 1.1: API ドキュメントの画面

＃2　構文の定義

2.1　プログラミング言語の定義

2.1.1　言語の定義と標準

プログラミング言語を設計したり検討したり実装するためには、まず言語を正確・厳密に定義する必要があります。なぜでしょうか? たとえば言語 X の定義が正確・厳密でなかったとします。そうすると、その言語で書いたプログラムをある処理系 A で動かしたときと、別の処理系 B で動かしたときで、微妙に動作が違うかも知れません。いや、動作が違うどころか、そもそも A 用に書いた X 語のプログラムを B に持っていって動かそうとしたら、いきなりエラーで受け付けてもらえない (動き出すことすらできない)、ということも十分起こり得ます。それでは困りますね?

実はこれはコンピュータ科学の初期においては「普通のこと」でした。たとえば IBM の PL/I コンパイラ用に作ったプログラムを FACOM 用の PL1 コンパイラ (なぜか名前が違えてありました) で動かそうとしたらこの機能は無いよとかで受け付けてもらえないなどは普通でした。

今日では、プログラミング言語の定義方法についての知見が進歩したので、まず X の言語仕様を厳密に定義した上で標準化します。その上で、その言語仕様に合致するようにさまざまな処理系を作るので、どこの処理系でも同じプログラムが動く…はずなのですが、微妙に問題が残ることは今でもあります。具体的には次のようなことです。

- 作るときの都合上で一部の機能を「作っていない」処理系があったりする。
- 逆に便利さのためや研究のため「独自拡張」を追加した処理系があったりする。

どちらの場合でも、ある処理系用に作ったプログラムが他の処理系で動かないことは同じで、頭痛の種です。まあ、独自拡張は使わないでプログラムを書き、標準化に従っていない処理系を使わなくすればだいたい大丈夫ですが…(そのほか、処理系の実装ミスもありますけれど)。

実はこれに加えて、次のようなケースもあります。

- 言語仕様に「不定」「処理系依存」つまり仕様としては決めないので適当に作ってね、という箇所がある。

このようなところは処理系の方でとりあえず決めるので、ある処理系と他の処理系で動作が違うことになります。その問題が認識されたので、今の言語ではできるだけ処理系依存をなくすように厳密に記述しますが、そのために言語仕様が大きく複雑になるわりには、完全に定義するというのはなかなか困難という問題を抱えています。

2.1.2　プログラミング言語をどのように定義するか

では次の問題として、ではプログラミング言語はどのようにしたら正確・厳密に定義できるでしょうか。たとえば、次のような課題にあなたはどのように答えますか。

演習 2-0 外国の人 (日本語を勉強中) に、「元号で年月日を書くやりかた」を教えるとしたら、どのようにするか考えなさい。西暦への換算はとりあえず後にして、まずは表記方法だけでよい (曜日も不要)。

(実際に少しやってみてから、この先を読むようにしてください。)

さて、上の問題をどのようにしましたか。多くの人は、まず元号として「明治」「大正」「昭和」「平成」があり、その後に年の数字 (1〜2 桁) があり (あと「元年」を忘れないように)、「年」があり、次に月の数字 (1〜12) があり、「月」があり、日の数字 (1〜31) があり、「日」がある、のように書き方の規則を説明したと思います。

そしてその後で、ではそれがどのような意味になるかを説明するわけです。なぜでしょうか。それは、「281 平成 26 月日 1」みたいにでたらめな書き方では解釈しようがないからです。書き方の規則に従っていてはじめて、ここはどういう意味、ということが説明できるのです。

プログラミング言語でもまったく同じで、「書き方の規則」を定めてから、それのあとで「それぞれの書き方に対応する意味」を定めます。すなわち、構文と意味をそれぞれ定めることでプログラミング言語を定義する、ということですね。

- **構文** (syntax) — 書き方、ないし「文字をどのように並べたものがその言語の正しいプログラムであるか」を定めた規則。
- **意味** (semantics) — 構文に合致した書き方のそれぞれの意味、ないし「そのように書いた場合にどのように動作するのか」を定めた規則。

意味から説明しましょう。意味についても数学のように厳密に記法から定めるやり方はあるのですが、この授業ではそれをやると大変すぎるので、「自然言語 (日本語) で」こういう書き方はこういう意味、というふうに説明することにします。ただし自然言語の弱点は曖昧さや不明瞭さですので、そのような問題が起きないように注意して記述します。

次に構文ですが、構文については何らかの決まった書き方を採用して、その書き方で記述することが普通です。それによって厳密性が保たれますし、言語の定義を読む人にとっても結局その方が簡潔で読みやすいのです。というわけで、この後はおもにその話題になります。

2.2　構文定義と構文木

2.2.1　BNF による構文定義

BNF(Backus Normal Form) とは、その名前通り Joun Backus というコンピュータ科学の先達が考案した「文法の書き方」です。**文法** (grammer) というと英文法のことが思い浮かぶかも知れませんが、「書き方の規則」という点では英語の書き方であってもプログラミング言語の書き方であっても別に変わりはないので、同じ用語を使います。英文法は English grammer ですが、プログラミング言語の場合は「grammer for programming language X」とかそういう感じで書きます。

さて話題を戻しますが、BNF では**メタ記号** (metasymbol) を用いて文法を記述します。メタ記号って? それは、たとえば C 言語ならそのプログラムには「+」とか「(」とかいろいろな記号が現れますね? BNF は C 言語などの言語の「書き方の規則を定める書き方」なので、その規則を記述するときに出て来る記号は「+などのプログラミング言語の記号」ではなく、「書き方の規則のための記号」つまり 1 レベル上の記号なのです。そういうものをメタ記号と言います。一般に「メタ (meta)」というのは「1 レベル上の」という意味ですから。

話が長くなると面白くないので細かいことは略し、とりあえず C 言語の関数定義 (の主に冒頭部分) を BNF で定義してみます (かなり略してあるので不正確ですが)。

```
関数定義 ::= 型指定 名前 ( パラメタ部 ) { 文列 }
型指定 ::= int | double | char | void | 型指定*
パラメタ部 ::= void | パラメタ
パラメタ列 ::= パラメタ | パラメタ列 , パラメタ
パラメタ ::= 型指定 名前 | 型指定 名前 []
文列 ::= ε | 文列 文
```

上の記述で「関数定義」というのはそういう文字が C 言語に直接現れるわけではなく、ここで C 言語の「関数定義というもの」を表しますよ、という指定なのでメタ記号です。この「関数定義」のように、プログラミング言語の中に直接現れないけれど、その言語の「〜というもの」を表すようなメタ記号のことを**非端記号** (nonterminal symbol) と呼びます。これに対し、丸かっことか `int` とか実際にプログラムの字面に現れるものは**端記号** (termnal symbol) と呼びます。そして、「::=」とか「|」とかはどちらでもない BNF の演算子 (メタ演算子) です (つまりプログラムには現れてこない)。なお、BNF にはいくつかもの流儀がありますが、ここでは一応筆者の好みの流儀でやっています (本質は同じです)。

「::=」は「左辺を右辺で定義する」なので、1 行目は「関数定義とは、型指定があり、名前があり、(があり、パラメタ部があり、) があり、{ があり、文列があり、} がある」と読みます。ここで C 言語の記号はそのままタイプフェースで書かれています。型指定、パラメタ列、文列はこの後で定義されます。それでは「名前」は? これは定義がないのですが、実は端記号で「C 言語の名前」(`main` とか `x` とか) を指します。ということは、「英字 (_を含む) で始まり、英数字の並んだ列」ということですね。このように、定義されていない名前はその言語の名前、文字列、数値などの端記号に対応しています。

2 行目は型指定ですが、ここで右辺に出て来る「|」は「または」を表します。つまり型指定は `int` か `double` か `char` か `void` か…その先は何でしょう? 「型指定の後に`*`」なので、たとえば `int` は型指定ですから、`int*`などが相当します。ところで `int*`も型指定と分かったので、ということは `int**` も型指定です。ということは`*`は何個あってもいいのですね。このように BNF では再帰 (定義の右辺の中に左辺が出て来る) を用いて繰り返しを表現します。

3 行目は簡単で、`void`(パラメタが無いことを表す) かパラメタ列かどちらかです。4 行目もまた再帰で、パラメタ列とは、α がパラメタ指定とすると、α だけでもいい (パラメタが 1 個) し、パラメタ列の後に「`,`」で区切って α をつけたものでもいいわけです。ということは「α」「α , α」「α , α , α」などカンマで区切った並びはこのようにして扱えるわけです。

5 行目はまた簡単で、パラメタ指定は型指定のあとに名前か、またはさらにその後に (配列を表す) `[]` をつけたものかどちらかということを意味します。

6 行目はまた目新しいもので、「ε (イプシロン)」が出てきます。これは伝統的に「空っぽの列」を意味することになっています。ということは、何もなくても文列ですし、「文列 文」は「ε 文」であってよいので、文 1 つでも文列ですし、あとは同様に再帰により何個の文の並びでも文列です。文が未定義ですが、そこから先はこれからの課題ということにしましょう。

演習 2-1 C 言語について次の部分の BNF を書いてみよ。

a. 文。式とかは後で作るものとしてよい。自分の知っている範囲の文だけでよい。
b. 式。とりあえず簡単にするため、整数定数、単独変数、四則演算だけでよい。
c. 上記に、配列、レコード、ポインタなども入れてみよ。
d. 上記に、関数呼び出しも入れてみよ。

2.2.2 文法と構文木

ある言語の文法が BNF で記述できたとして、実際に具体的なソースコードを読み込んだとき、それをどのように使うのでしょうか。それは「関数」とか「文」とか「式」などの非端記号と具体的なソースコードの文字列の範囲の対応をつける必要があります。たとえば、次の文法を見てください。

$$\text{列} ::= \epsilon \mid \texttt{a}\ \text{列}$$

これは、次のように考えると、「a が 0 個以上並んだ列」と分かります。

$$\text{列} \rightarrow \texttt{a}\ \text{列} \rightarrow \texttt{a a}\ \text{列} \rightarrow \texttt{a a a}\ \epsilon == \texttt{a a a}$$

ここで、入力「a a a」と構文の対応を、図 2.1(1) のように木構で書き表すことができます。

この木構造は、(a) 下端には入力列 (と必要なら ϵ) が並んでいて、(b) ノードは端記号であり、(c) 各ノードから下向きの枝は必ずいずれかの文法規則にあてはまる (たとえば一番上の「列」からは「a」と「列」に枝が出ているが、それは「列 ::= a 列」に対応している)、という約束を守っています。このような木構造を**構文木** (syntax tree) と呼びます。

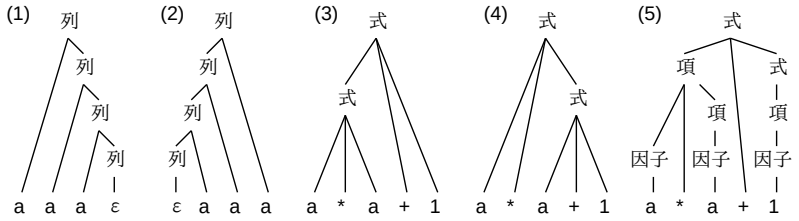

図 2.1: 構文木の例

ところで、先の類似品で次の文法を見てください。

列 ::= ϵ | 列 a

これも、「a が 0 個以上並んだ列」という点ではまったく同じですが、同じ入力に対応する構文木は図 2.1(2) のようになります。このように、構文木は入力と文法の対応を正確に表現でき、便利なのです。

ところで、次は式に対する次の文法を見てください。

式 ::= 式 + 式 | 式 * 式 | 1 | a

これを「a * a + 1」という入力にあてはめると、図 2.1(3) と (4) のように 2 通りの構文木を作ることができます。このように、入力に対して複数の構文木が作れる文法のことを**曖昧** (ambiguous) である、と言います。通常は、曖昧な文法は避けるようにします。

では、先の例を曖昧でなくするにはどうしたらいいでしょうか。2 つの構文木を見比べて、どちらが適切だと思いますか? 「乗除算は加減算より強く結びつく」のですから、(3) の形になるのが正しそうです。そこで、次の文法を見てください。

式 ::= 項 | 項 + 式
項 ::= 因子 | 因子 * 項
因子 ::= 1 | a

このようにすると、図 2.1(5) の構文木しか作れなくなります。つまり、曖昧でない文法に書き換えることができたわけです。そのかわり、文法はかなり複雑になっています。[1]

演習 2-2 構文木について次のことをやってみなさい。

a. 上の曖昧でない式の文法について、「a + a * a + 1」「a + 1 + a * a」に対する構文木を描き、確かに乗算が強く結び付くことを確認しなさい。

b. 上の曖昧でない式の文法では「a + a + a」の場合、左の足し算から順に実行する。これを右の足し算から実行するように文法を変更して構文木を描き、確認しなさい (足し算だとどちらが先でも良さそうだが、引き算だと違いがある)。

c. 上の曖昧でない文法の「因子」を次のように修正する。

因子 ::= 1 | a | (式)

[1]「因子」を無くして項の右辺に a と 1 をそれぞれ書くこともできますが、「因子」の種類が増えて来たら大変なので普通はそういうことはしません。

これでかっこのついた式を適宜考え、構文木を描いてかっこの中が強く結び付くことを確認しなさい。

d. 次の文法を考える (「式」はこれまでのものを使う)。

文列 ::= ϵ | 文 文列
文 ::= 式 ; | while (式) 文 | if (式) 文 | { 文列 }

これで while 文と if 文の組み合わさったコードを適宜考え、構文木を描きなさい。

e. 上の文法の「文」に次の選択肢を追加する。

文 ::= if (式) 文 else 文

このとき、この文法が曖昧であることを示すような例を考え、2 通りの構文木を描きなさい。C 言語か Java 言語でこのようなコードが実際にどのように動作するか調べなさい。

2.3 再帰下降解析による構文検査

2.3.1 構文検査とは

ずっとお話だとつまらないので、ここで簡単な**構文検査器** (syntax checker) を Java で作ってみます。[2] 次のような制約を設けます。

- 「言語」は 1 行ぶんの文字列で、トークンはすべて 1 文字。

そこで、たとえば次のような BNF を考えます。

prog ::= ϵ | *ab prog*
ab ::= a b

prog のように斜体で示しているのが非端記号です。もともとの BNF では<program>のように角かっこで囲むのが正式な非端記号の表し方でしたが、複数フォントが使えるときはそれを活用する方が見やすいです。先に出て来た例では、日本語の言葉はそのままで非端記号ということで扱っていました。

それで、この BNF はちょっと考えれば分かるように「ab の 2 文字が 0 個以上反復した列」になります。実際にプログラムを動かしているところを示します。

```
% java Sam21 'abab'
true
% java Sam21 'aba'
false
% java Sam21 ''
true
%
```

このように、「形があってるかどうか」だけ調べるプログラムは**構文検査器** (syntax checker) と呼ばれます。コンパイラ作るときには検査だけでは済まないですが、しばらくは簡単なので検査器だけで試していきます。実際にプログラムを見ていきましょう。

2.3.2 文字単位の Toknizer

前回は単語単位でよむ Toknizer をやりましたが、今回は上記のように 1 文字ずつで扱うので 1 文字ずつ取るだけの Toknizer をつくります。ソースを見てみましょう。

[2] 構文検査器ができればそれを修正して構文木を組み立てるようにもできるのですが、ここでは簡単のため検査だけです。

```
class CharToknizer {
  String str, tok;
  int pos = -1;
  boolean eof = false;
  public CharToknizer(String s) { str = s; fwd(); }
  public boolean isEof() { return eof; }
  public String curTok() { return tok; }
  public boolean chk(String s) { return tok.equals(s); }
  public void fwd() {
    if(eof) { return; }
    pos += 1;
    if(pos >= str.length()) { eof = true; tok = "$"; return; }
    tok = str.substring(pos, pos+1);
  }
  public boolean chkfwd(String s) {
    if(chk(s)) { fwd(); return true; } else { return false; }
  }
}
```

インタフェースは前回の Toknizer にほぼ合わせてあります。内部表現はファイルから読むのでなく、コンストラクタで文字列を渡すとそれをインスタンス変数 str に格納し、pos でそのどの位置を見ているかを管理します。最初は pos を-1 にしているのは、作ってすぐに fwd() を呼ぶことで最初の文字にセットするためです。

ほとんどの仕事は fwd() で行ないます。eof なら何もせず帰ります。そうでなければ、pos を 1 すすめ、もし文字列の終わりに来たら eof にします。そうでなければ、文字列のその位置を (長さ 1 の文字列として) 取り出します。文字列のメソッド length()、substring() の機能は API ドキュメントで確認してください。

さて、最後のメソッド chkfwd() は前回なかったものですが、chk() して OK だった場合はそのトークンを fwd() します。これは書きやすさのために用意しています (後で分かります)。

2.3.3　static の謎

この先実際のコードに入る前に、前回きちんと扱えなかった static について説明しましょう。Java では static とはひらたくいえば「インスタンスを作らなくても使える」という意味になります。

図 2.2 を見てください。クラス MyClass にはメソッド main と method1 がありますが、前者は static がついているので、MyClass のインスタンスを作らなくても呼び出せます。実際 main() はそうして呼び出していますね。これに対し、method1 はまず MyClass c = new MyClass(...) によりインスタンスを生成し、そのあと c.methods1(...) のようにして呼び出す形 (メッセージ送信記法) でしか呼び出せません。これが static の有無の違いです。

そして、変数 sx/sy と ix/iy の違いもこれに関係します。ix/iy はインスタンス変数で個々のインスタンスに付随して 1 つずつ存在するので、インスタンスのある状態でしかアクセスできません。ということは、インスタンスメソッド mehotd1 の中からは参照できますが、static メソッドである main の中からは参照できません。一方 sx/sy は static 変数なので、クラスに対応して 1 つだけ存在し (グローバル変数のようなもの)、method1 からでも main からでも参照できます。

さらに、これまではクラスは並べて書いてきましたが、クラスの中でクラスを定義することもできます。とくにクラス内で定義している static 変数をアクセスしたい場合はそのようにする必要があります。[3] これを**内部クラス** (inner class) と呼びます。

[3] 正確に言えば、MyClass.sx のように外側クラス名を前置すれば外部からもアクセスできるようにもできますが、複雑

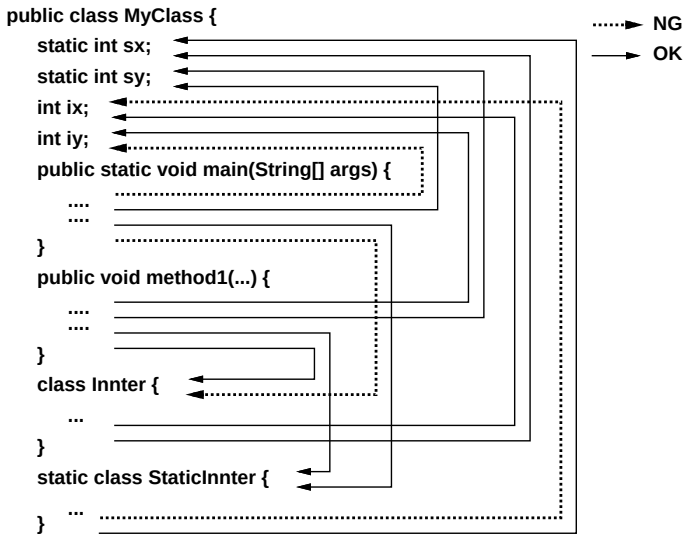

図 2.2: static の参照可否

内部クラスにも static ありと無しがあります。static なしの内部クラスは static なしのメソッド内からでしかインスタンスを生成できず、その中からはインスタンス変数 ix/iy がアクセスできます。つまり内部クラスのインスタンスは外側クラスのインスタンスに紐ついています。しかしこの機能は分かりにくいのであまり使いません。

一方、static つきの内部クラスは普通の (外側に書いた) クラスと同じで、static メソッドからでもインスタンスを生成でき、その中からは static のついた変数しか参照できません。通常はこちらを使うことが多いでしょう。

2.3.4 再帰下降解析器

ではいよいよ本体クラスを見ましょう。まず main では、コマンド引数の 1 番目の文字列を渡して CharToknizer を作り、static 変数 tok に入れます。これで、このクラス内のどのメソッドからも tok が参照できます。そして、最初のメソッドは「プログラム」に対応するものということにしたので、prog() を呼び、その正否 (渡した文字列が文法に合っていたか否か) を打ち出します。ただし、あてはまっても後ろにトークンが残ってはいけないので、さらに tok.isEof() であるという条件も必要です。

```
import java.util.*;
import java.io.*;

public class Sam21 {
  static CharToknizer tok;
  public static void main(String[] args) throws Exception {
    tok = new CharToknizer(args[0]);
    System.out.println(prog() && tok.isEof());
  }
  static boolean prog() {
    if(tok.chk("a")) {
      return ab() && prog();
    } else {
```

になるのでここでは扱っていません。

```
        return true;
      }
    }
    static boolean ab() {
      return tok.chkfwd("a") && tok.chkfwd("b");
    }
  }
  // ここに CharToknizer を入れる
```

さて、その後の 2 つのメソッドは、BNF に出て来る 2 つの端記号と同じ名前にしてあります。そして、それらが互いに再帰呼び出しをすることで、構文チェックができます。具体的には次のことが必要です。

1. 手続き P が呼び出され、成功した場合 (`true` を返す場合) は、ちょうどその P に対応するだけ入力を読み進む必要がある。`false` を返す場合はこのような制約はない。
2. 最初に呼び出される P については、`true` を返すときにはちょうど EOF まで読み終わっている必要がある。

このようなやりかたで再帰手続きの集まりによって構文検査 (または解析) を行なう方法を、**再帰下降解析** (recursive descent parsing) と呼びます。まあその理論は後の方で。

では、メソッド `prog()` から順に見ていきます。まず、「|」が含まれている規則の場合、そのどの枝 (選択肢) に行くかを決める必要があります。ここで説明している手法では「次の 1 トークンを見て決める」こととします (そのような文法だけを扱えます)。規則はこれですね。

$$prog \ ::= \ \epsilon \ | \ ab \ \ prog$$

そして、ϵ は何が来ても (空っぽでも) あてはまり得るのですぐ `true` を返しますが、しかし常にそうしてしまうと ab の枝が選ばれなくなります。なので、まず ab の枝かどうかをチェックし (それには次のトークンが「`a`」か調べます)、それが OK ならそちらを選びます。その枝であれば、`ab()` であり、かつさらに `prog()` であればいいわけです。OK でなければ ϵ の枝なので、すぐ `true` を返します。

メソッド `ab()` に対応する 2 番目の規則はこれです。

$$ab \ ::= \ \texttt{a b}$$

これはつまり、トークンがまず「`a`」、次に「`b`」であることを順に調べればいいです。この枝しかないので、調べたらすぐ `fwd()` で次のトークンに進みます。そのために `checkfwd()` が便利なわけです。どちらかが `false` を返したらこのメソッドもすぐ `false` を返します。

どうでしょうか。文法に対応して機械的に「どの枝か調べ」「その枝なら順番に処理する」という呼び出しをすれば完成すると思いませんか。このパターンを定式化したものを図 2.3 に示します。

まず、非端記号 S に対して `s()` というメソッドを作ります。そして、S を定義する規則の右辺の選択肢の数だけ if-else の選択肢を作り、どの選択肢に進むかを判定します。通常は、どこへ行くかは入力の現在位置のトークンを見ることで判断できます。ϵ は入力に対応しないので、ϵ の選択肢がある場合はそれは最後の else に対応させることになります (ϵ が無いなら順番に選択していって最後の 1 つを else に対応させてよい)。

それぞれの選択肢の中では、その選択肢にある記号の列に対応する呼び出しをおこない、それらがすべて `true` である場合にのみ `true` を返します。呼び出しは非端記号であればそれに対応するメソッドを呼び、端記号であれば `tok.chkfwd()` を呼べばよいです。

なぜこれでよいのでしょう。それぞれのメソッドは (複数の選択肢があれば適切な選択肢を選んだ上で)「そのメソッドに対応する端記号の列があったことを確認し、その分だけ入力を読み進める」働

```
S ::= ε | T11 x T13 | y T22 | T3

static boolean s() {
    if(T1 chosen) {
        return t11() && tok.chkfwd("x") && t3();
    } else if(T2 chosen) {
        return tok.chkfwd("y") && t22():
    } else if(T3 chosen) {
        return t3();
    } else {
        return true;
    }
}
```

図 2.3: 再帰下降解析用メソッドの構造

きを持ちます。それは具体的には、端記号については `tok.chkfwd()` を呼ぶことで行え、非端記号についてはそれに対応するメソッドを下請けに呼び出すことで行えます。ですから、最初の `prog()` に対して呼び出した結果が `true` であれば、`prog()` から始まる規則適用の列が正しくあてはまっていることになるわけです。

演習 2-3 上の例題を打ち込み、そのまま動かしなさい。動いたら、次の文法がどのような文字列を表現しているか考え、また再帰下降解析を実現して動かしてみなさい。

a. $prog$ ::= ϵ | ab $prog$
   ab ::= `a` | `bb`

b. $prog$ ::= `a` | `a` $prog$

c. $prog$ ::= ϵ | aa bb $prog$
   aa ::= `a` | `a` aa
   bb ::= `b` | `b` bb

d. $prog$ ::= `1` | `(` $prog$ `)`

e. $prog$ ::= $term$ | $term$ op $prog$
   $term$ ::= `a` | `1` | `(` $prog$ `)`
   op ::= `+` | `-` | `*` | `/`

f. その他自分が試してみたい文法

2.3.5 演習 2-3 の解答例

まだ慣れていないと思うので、演習 2-3 の各問題の解答例 (再帰下降解析用のメソッドのみ) を示して解説します。まず a. ですが、これは繰り返される要素が「a または bb」ということになります。

```
static boolean prog() {
  if(tok.chk("a") || tok.chk("b")) {
    return ab() && prog();
  } else {
    return true;
  }
```

```
  }
  static boolean ab() {
    if(tok.chk("a")) {
      return tok.chkfwd("a");
    } else {
      return tok.chkfwd("b") && tok.chkfwd("b");
    }
  }
```

prog() は同じでいいと一見思えますが、ab() に進むための条件は「次が a または b」になります。そして ab() の中では | が現れたのでどちらの枝かを選択してそれぞれで処理します。

次に b. ですが、これは「a が 1 個以上並んだもの」です。一見簡単そうですが、この規則は 1 文字目だけ見るとどちらも「a」なので選択できません (つまり先に挙げた規則の条件を満たさない)。しかし、a の先まで行ってから選択することはできます。

```
  static boolean prog() {
    if(tok.chk("a")) {
      tok.fwd();
      if(tok.chk("a")) {
        return prog();
      } else {
        return true;
      }
    } else {
      return false;
    }
  }
}
```

つまり、a の先まで行ってみて、さらにまた a なら prog() を再帰呼び出し、そうでなければそこまでで ture を返します。なお、そもそも a で始まらない場合は false です。このように、「途中まで共通になっている選択肢」があると少しややこしいことになります。

c. も実は aa() と bb() がその構造なのでそこがやっかいですが、prog() は簡単です。

```
  static boolean prog() {
    if(tok.chk("a")) {
      return aa() && bb() && prog();
    } else {
      return true;
    }
  }
  static boolean aa() {
    if(tok.chk("a")) {
      tok.fwd();
      if(tok.chk("a")) {
        return aa();
      } else {
        return true;
```

```
      }
    } else {
      return false;
    }
  }

  static boolean bb() {
    if(tok.chk("b")) {
      tok.fwd();
      if(tok.chk("b")) {
        return bb();
      } else {
        return true;
      }
    } else {
      return false;
    }
  }
}
```

次に d. はどうでしょうか。これは選択肢を選ぶのが簡単なので、作るのも簡単です。

```
  static boolean prog() {
    if(tok.chk("1")) {
      tok.fwd(); return true;
    } else if(tok.chk("(")) {
      return tok.chkfwd("(") && prog() && tok.chkfwd(")");
    } else {
      return false;
    }
  }
```

これは何ができるかというと、「1」の周囲をかっこで囲んだもので、かっこは何重でも大丈夫です。このような構造は BNF が得意とするところです。

最後の e. はかなり複雑です。実はこれは (不完全ですが)、算術式をイメージしています。*prog* が「式」に相当します。

```
  static boolean prog() {
    if(term()) {
      if(tok.chk("+")||tok.chk("-")||tok.chk("*")||tok.chk("/")) {
        return op() && prog();
      } else {
        return true;
      }
    } else {
      return false;
    }
  }
```

```
  static boolean term() {
    if(tok.chk("a")) {
      tok.fwd(); return true;
    } else if(tok.chk("1")) {
      tok.fwd(); return true;
    } else {
      return tok.chkfwd("(") && prog() && tok.chkfwd(")");
    }
  }
  static boolean op() {
    return tok.chkfwd("+") || tok.chkfwd("-") ||
           tok.chkfwd("*") || tok.chkfwd("/");
  }
}
```

prog() だけは先頭が共通の選択肢がありますが、あとは普通に区分できます。書き方はもうちょっと短くできるところが多いですが、多少長くなってもなるべく同じパターンで書くようにしています。

演習 2-4 先の演習の e. は式に対応しているが、演算が四則演算しかなかった。次のように強化してみよ。

a. C や Java では代入も演算の 1 つという位置付けである。代入も扱えるようにしてみよ。
b. if 文などで使うには比較演算子が不可欠である。比較演算子も使えるようにしてみよ。
c. 関数呼び出しも重要な式の要素である。関数呼び出しも使えるようにしてみよ。
d. その他、自分が追加したいと思う機能を追加してみよ。

演習 2-5 次のような文法の再帰下降解析器を構成してみよ。なお、*Expr* は上の演習の *prog* に対応しているものとする (名前だけ書き換えて使えばよい)。

a. $prog$ ::= { $statlist$ }
$statlist$ ::= ϵ | $stat$ $statlist$
$stat$ ::= $Expr$; | { $statlist$ }

b. 上記の *stat* の選択肢に次を追加せよ。

$stat$ ::= w ($Expr$) $stat$ | d $stat$ w ($Expr$) ;

c. 上記の *stat* の選択肢に次を追加せよ。

$stat$ ::= i ($Expr$) $stat$
| i ($Expr$) $stat$ e $stat$

d. 上記の文法に好きなものを追加せよ。

2.4　課題 2A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題＃ 2」、学籍番号、氏名、提出日付。

- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. BNF によって言語の文法を規定するのはどう思いましたか。

  Q2. 再帰下降解析で構文へのあてはめができるということに納得しましたか。どこが難しかったですか。

  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

＃3　言語処理系の構成

3.1　言語処理系の枠組み

3.1.1　言語処理系の分類

プログラミング言語処理系 (programming language processor、以下では語処理系と記す) とは、プログラミング言語の記述ないし**ソースコード** (source code) を入力とし、そこに記述された動作 (== プログラムの動作) を実現するようなソフトウェア全般を指します。

大まかな分類として、言語処理系は**インタプリタ** (interperter) ないし**解釈系**と**トランスレータ** (translater) ないし**翻訳系**に 2 分できます。前者はソースコードを読み込んだ後、その動作を直接そのソフトウェアが実行するのに対し、後者はソースコードを同等の動作を行う**目的コード** (object code) ないし**ターゲットコード** (target code) に変換して出力します。

ここで、目的コードの形式が**機械語** (machine language) ないしそれに近い水準のものである場合に、その翻訳系のことを**コンパイラ** (compiler) と呼ぶのが慣わしです。コンパイラ以外の翻訳系としては、簡便な形で他の高水準言語に変換する**プリプロセサ** (preprocessor) などがあります。目的コードが他の高水準言語であっても、その言語のプログラムとして読めないような変換を行う場合はコンパイラに分類することもあります。

コンパイラが特定 CPU の命令を生成する場合、さまざまなマシンで動かすにはそれぞれの CPU の命令を出力するようにコンパイラを複数用意する必要がありますが、これはなかなか大変です。

コンパイラの中には、目的コードが実在の CPU の機械語ではなく、仮想的な (汎用性のある) 機械語であるものもあります。この場合、その仮想的な機械語を実行するソフトウェアは**仮想マシン** (virtual machine) と呼ばれます。この方式であれば、コンパイラは 1 種類でよく、さまざまな CPU 用に仮想マシンだけ用意すれば済みます。Java 言語はまさにこのような形で処理系が構成されています (`javac` がコンパイラ、`java` が仮想マシンのプログラム)。

3.1.2　言語処理系の構造

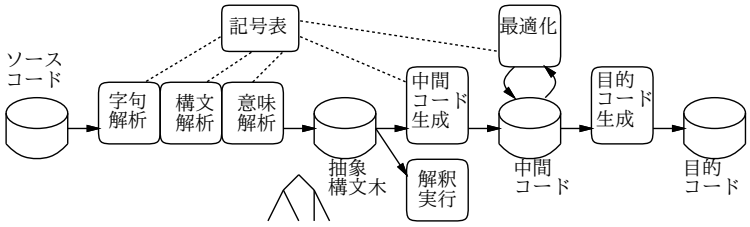

図 3.1: 言語処理系の一般的な構成

一般的な言語処理系の大まかな構造を、図 3.1 に示します。ディスク記号で記したのがデータ構造、四角で囲んだのが処理系内部のコンポーネントです。

コンパイラの場合、その要素は大きくわけてソースコードから情報を取り出す**解析部** (analysis phase) ないし**フロントエンド** (front end) と、その情報に基づいて目的コードを生成していく**生成部** (synthesis phase) ないし**バックエンド** (back end) から成ります。図 3.1 でいって、中央にある「抽象

構文木」の左側が解析部、右側が生成部になります。なお、ここで示しているのはあくまでも一般的な構成であり、処理系によってはこれと異なる形になることもあります。

各部分の詳細についてはこれから時間をかけて解説していきますが、ここでは各部分の大まかな機能や構造について説明します。

ソースコード (source code) — プログラミング言語の形で記述されたプログラム。言語処理系の入力となる。

字句解析 (lexcal analysis) — ソースコードを「名前」「定数」「記号」などのかたまり (トークン) に分割する。コメントの削除などの処理もここで行なう。

構文解析 (syntax analysis) — トークンの列に対して構文規則のあてはめを行ない、「メソッド」「文」「式」などの要素がどの範囲でどういう構造になっているかを決定する。

意味解析 (semantic analysis) — 関数、変数、型などの使用状況を解析し、名前の未定義や多重定義、型の不一致などの誤りを検出する。

記号表 (symbol table) — これは受動的なデータ構造である場合もある。コンパイラの各コンポーネントは名前、型などの情報を記録し、また必要に応じて参照する必要があるが、それらの情報を保持し必要なサポートを行なうのが記号表である。

抽象構文木 (abstract syntax tree) — ソースコードの情報から後の段に必要な構造や情報を抽出した結果は木構造のデータで表現することが多く、これを抽象構文木と呼ぶ。

解釈実行 (interpritive execution) — インタプリタの場合は、抽象構文木から直接実行することもでき、簡易な処理系で多くこの形が使われる。ただし実行速度が遅くなるので、より高速にしたい場合は仮想マシン型の構成が使われるので、コード生成の側に進む。

中間コード生成 (intermediate code generation) — 目的コード (機械語やアセンブリ言語) は内部処理には向いていないので、いったん中間的な形式のコードを生成し、最適化に進むことが普通である。簡易な処理系で最適化を最小限にしたものでは、抽象構文木から直接目的コード生成につながるものもある。

中間コード (intermediate code) — 最適化の処理に適した形でプログラムの動作を表現するコード形式。

最適化 (optimization) — プログラムの中の無駄を削減したり、意味を変えない範囲でより高速に実行できる形に書き換えるなどして、目的コードの実行が速くなるようにする。ここでは中間コードに対してさまざまな最適化を繰り返し行なうイメージで描いてあるが、そのほかに目的コード生成時に行なう最適化もある。

目的コード生成 (target code generation) — 中間コードから目的コードの形式に変換する。目的コードが機械語やアセンブリ言語など低水準 (CPU 依存) 形式の場合、他の CPU 向けにコンパイラを移植するときはこの部分を変更する。

目的コード (target code) — 最終的に出力されるコードで、機械語やアセンブリ言語のこともあるが、仮想マシン方式のように仮想マシン語を出力したり、他の高水準言語のソースコードを出力することもある。

3.2 抽象構文木とその利用

3.2.1 抽象構文木

抽象構文木 (abstract syntax tree, **AST**) は既に学んだ構文木に基づいていますが、構文木の「文法に正確に対応する」という制約は無くし、また言語処理系にとって必要な情報を適宜データ構造 (具体的には木のノード) に追加するという形で作られています。

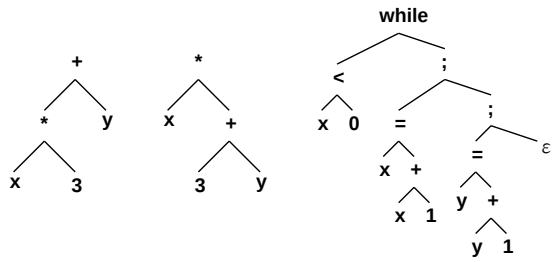

図 3.2: 抽象構文木の図解例

図 3.2 に抽象構文木を図解した例を示します。このように、抽象構文木ではノードのところにそのノードに対応する意味づけ (演算の種類など) を書くことが多いです。図 3.2 左と中はそれぞれ「`x * 3 + y`」と「`x * (3 + y)`」に対応する木構造を表しています。また、図 3.2 右は

```
while ( x < 0 ) { x = x + 1; y = y + 1; }
```

に対応する木構造を表しています。このように、文の並びは「並びを表すノード」を使って数珠つなぎにして表現することが普通です。いずれの場合も、「`( … )`」や「`{ … }`」などの構造を指定するための記法は木構造の上では無くなっていることに注意 (木構造自体でその構造が表せるためにそうなっています)。

3.2.2 Java による式木の実装

式木 (expression tree) とは、式を表現する抽象構文木を言います。ここでは整数のみの式を扱い、「変数」「定数」「加算」「乗算」のノードを作ることにしました。まず冒頭と `main()` の部分だけを示します。

```
import java.util.*;

public class Sam31 {
  static Map<String,Integer> vars = new TreeMap<String,Integer>();

  public static void main(String[] args) {
    vars.put("x", 5);
    Node expr = new Add(new Mul(new Var("x"), new Lit(3)), new Lit(1));
    System.out.println(expr);
    System.out.println(expr.eval());
  }
  // ここに他の static な内部クラスを入れる
}
```

`Map<`T_1`,`T_2`>`は型 T_1 をキーとし、型 T_2 を値として保持するような表を表す型です (Java の用語としてはインタフェース)。ここでは、文字列 (=変数名) をキーとし、その変数の値 (=整数) を保持す

る型 Map<String,Integer>の static 変数 vars を用意し、そこに TreeMap<String,Integer>のインスタンスを作って格納します。TreeMap は赤黒木 (という名前のデータ構造で、キーの昇順に項目が並べられる) を用いた表の実装です。

このように、Java では変数にはその型と互換性のあるさまざまな型のオブジェクトを格納することができます。互換性の規則についてはまた後で説明します。

そして、この変数 vars は static なので、後で出て来るノードのためのクラスもすべてこのクラスの static な内部クラスとすることでどこからでも参照できるようにしています。

そしてようやく、main() の動作ですが、まず冒頭で、vars の"x"の項目に 5 を入れます。これは変数 x に 5 が入っていることを表しているつもりです。その次に、様々なノードのコンストラクタを使って式木を組み立てます。すべてのノードは Node と互換性があるように作ってあります。ここで組み立てている式は「tt (x * 3) + 1」に相当します。

その後、まず作成したオブジェクトを文字列表示し、続いて eval() により値を計算して表示します。main() の後には多くのクラスが出て来ますが、その説明の前に実行例を示そう。x には 5 を入れてあるので、確かに正しい値が計算できています。

```
% java Sam31
((x*3)+1)
16
%
```

3.2.3　式木のノード群と継承

では、式木のノードに対応するクラス群を読んでいくことにします。これらはすべて、クラス Sam31 の内部クラスで、かつ static です。

ここで新たな概念として、**継承** (inheritance) について説明します。継承とは、複数の類似した (関連性のある) クラスを作るときに「あるクラス C を土台として別のクラス D を作る」操作です。このとき、C を**基底クラス** (base class) ないし**親クラス** (parent class)、D を**派生クラス** (derived class) ないし**子クラス** (child class) と呼びます。継承をおこなうには、子クラス D の冒頭に「extends C」という指定をおこないます。

継承をおこなうと、次の 3 つの効果が得られます。

1. クラス C で定義されている変数やメソッドの定義はそのまま D にも引き継がれる (コピーされてくると考えればよい)。
2. メソッド定義については、同じパラメタや返値をもつ同名のメソッドを再定義することで、継承してきたメソッドを**オーバライド** (override、上書きの意味) できる。
3. 型 D の値は型 C に**互換** (compatible) となる。つまり型 C の変数にクラス D のインスタンスを入れられる。

これらに加え、クラス D で新たなインスタンス変数やメソッドを追加することは自由です。

上記の事項のうち 3. については、D は C からメソッドやデータ構造を引き継いでいるので、D のインスタンスはほぼ C のインスタンスと互換性があるようになるので、こうなっています (実際にはオーバライドのしかたや機能に追加により互換な動作ができないこともあり得ますが、なるべくそうはしないというのがお約束です)。

また、D を親としてさらに継承した E を作ることもあり、その場合はこれらのクラスもすべて C と互換性があることになります。実は Java ではとくに指定のないクラスは extend Object を指定したとして扱われるので、すべてのクラスは Object に互換であり、このクラスから必要最小限の機能を継承しています。

継承の機能の話題に戻って、事項の 1. のおかけで、類似した機能を持つ多数のクラスを少ない行数で書くことができます。それは D を定義するときに、C から基本は引き継いできて、ことなる部分だけ記述すればよいからです。これを**差分プログラミング** (differential programming) と呼びます。事項 2. については、継承してきたそのままでは機能が十分でないことがあることによります。

また、クラス群の設計上、必ずオーバライドが必要なメソッドを定義することもあります。たとえば、今回の例ではクラス `Node` がすべてのノードの基底となり、そこで定義するメソッド `eval()` は「値を計算する」機能となりますが、実際の計算方法は「加算」「変数」など個別のクラスでなければ決められません。そこで、`Node` ではメソッド `eval()` を**抽象メソッド** (abstract method) と指定し (コード定義を持たない)、その子孫のクラスでオーバライドすることを指定します。抽象メソッドを持つようなクラスは**抽象クラス** (abstract class) と呼ばれ、インスタンスを作ることはできません。

では実際に、抽象クラス `Node` を見てみます。

```
abstract static class Node {
  List<Node> child = new ArrayList<Node>();
  public void add(Node n) { child.add(n); }
  public abstract int eval();
}
```

インスタンス変数 `child` はノードの並びを保持する `List<Node>`型の変数であり、その実装としては配列のように機能する `ArrayList<Node>`のインスタンスを入れます。メソッド `add` はそこに子ノードを追加するメソッドです。そしてメソッド `eval()` は前述にように抽象メソッドであり、コードを持ちません。

次に、このクラスを継承して作る定数と変数のクラスを見てみます。

```
static class Lit extends Node {
  int val;
  public Lit(int v) { val = v; }
  public int eval() { return val; }
  public String toString() { return ""+val; }
}
static class Var extends Node {
  String name;
  public Var(String n) { name = n; }
  public int eval() { return vars.get(name); }
  public String toString() { return name; }
}
```

定数の方は整数の値をインスタンス変数 `val` に保持します。その値は生成時にコンストラクタで受け取り代入します。`eval()` はその値を返せばよいです。なお、`toString()` は `Object` から継承しているメソッドであり、そのインスタンスを打ち出すなどのときに文字列に変換する際に呼び出されるます。ここでは `val` に値を空文字列と連結して得られる文字列オブジェクトを返します。[1]

変数では、名前の文字列を保持する `name` と値を格納する表 (`Map<String,Integer>`) を保持する `tbl` の 2 つがインスタンス変数であり、いずれもコンストラクタで受け取って初期設定します。`eval()` はその表から変数名を指定して値を取り出して返します。`toString()` は名前の文字列を返せば良いです。

次は 2 項演算である加算と乗算ですが、これらは共通する内容があるので、その共通部分を `BinOp` という抽象クラスにまとめました。

[1] Java では文字列と何かを連結するとその「何か」が文字列に自動変換されてから連結されることになっている。

```
abstract static class BinOp extends Node {
  String op;
  public BinOp(String o, Node n1, Node n2) { op = o; add(n1); add(n2); }
  public String toString() { return "("+child.get(0)+op+child.get(1)+")"; }
}
```

具体的には、演算を表す文字列 op をインスタンス変数として追加し、コンストラクタでその文字列と 2 つのノード (演算される 2 つの被演算式に対応) を受け取り、op は初期化し、2 つのノードは child(継承してきているインスタンス変数で、List<node>を保持) に順次追加しています。toString() では、2 つの被演算式を中央に演算子いを入れて連結し、全体を「(...)」で囲んだ文字列を返しています。

以上があれば Add と Mul は簡単で、eval() で 2 つの被演算式を計算してそれらを加算/乗算して返すようにするだけです。ただしあと 1 つ、自らのコンストラクタの中で初期化のために BinOp のコンストラクタを呼び出さなければなりません。Java では子クラスのコンストラクタから親クラスのコンストラクタを呼び出すときには「super(…)」という書き方で呼び出すことになっています。

```
static class Add extends BinOp {
  public Add(Node n1, Node n2) { super("+", n1, n2); }
  public int eval() { return child.get(0).eval() + child.get(1).eval(); }
}

static class Mul extends BinOp {
  public Mul(Node n1, Node n2) { super("*", n1, n2); }
  public int eval() { return child.get(0).eval() * child.get(1).eval(); }
}
```

演習 3-1 上の例題のコードを入手してそのまま動かせ。動いたら、次のことをやってみょ。ノードを追加したら正しく動作することを確認すること。

a. 別の計算式を構成し、その計算ができることを確認する。

b. 減算、除算、剰余のノード Sub、Div、Mod を追加する。

c. 比較演算子のノード Eq、Ne、Gt、Ge、Lt、Le を追加する。いずれも、条件が真なら「1」、偽なら「0」を値とする。

d. &&に相当する And、||に相当する Or、!に相当する Not を追加する。Not は被演算子が 1 つなので多少工夫が必要。

e. 代入のノード Assign を追加する。コンストラクタは変数のノードと一般の式のノードを受け取る。exec() では式を計算し、その値を変数表に格納するとともに、全体の値としては代入した値を返す。

f. 順次実行のノード Seq を追加する。動作としては 2 つのノードを保持し、それらを順に実行し、値としては 2 つ目のものの値を返す。2 つ目のノードは nil でもよく、その場合は 2 つ目は実行せずに 1 つ目の値を返す。

3.2.4　より複雑なノード

とりあえず、先の演習問題関係のノードを示します。

```
static class Sub extends BinOp {
  public Sub(Node n1, Node n2) { super("-", n1, n2); }
  public int eval() { return child.get(0).eval() - child.get(1).eval(); }
}
```

```
static class Div extends BinOp {
  public Div(Node n1, Node n2) { super("/", n1, n2); }
  public int eval() { return child.get(0).eval() / child.get(1).eval(); }
}
static class Mod extends BinOp {
  public Mod(Node n1, Node n2) { super("%", n1, n2); }
  public int eval() { return child.get(0).eval() % child.get(1).eval(); }
}
static class Eq extends BinOp {
  public Eq(Node n1, Node n2) { super("==", n1, n2); }
  public int eval() { return child.get(0).eval()==child.get(1).eval()?1:0; }
}
static class Ne extends BinOp {
  public Ne(Node n1, Node n2) { super("!=", n1, n2); }
  public int eval() { return child.get(0).eval()!=child.get(1).eval()?1:0; }
}

static class Gt extends BinOp {
  public Gt(Node n1, Node n2) { super(">", n1, n2); }
  public int eval() { return child.get(0).eval()>child.get(1).eval()?1:0; }
}
static class Ge extends BinOp {
  public Ge(Node n1, Node n2) { super(">=", n1, n2); }
  public int eval() { return child.get(0).eval()>=child.get(1).eval()?1:0; }
}
static class Lt extends BinOp {
  public Lt(Node n1, Node n2) { super("<", n1, n2); }
  public int eval() { return child.get(0).eval()<child.get(1).eval()?1:0; }
}
static class Le extends BinOp {
  public Le(Node n1, Node n2) { super("<=", n1, n2); }
  public int eval() { return child.get(0).eval()<=child.get(1).eval()?1:0; }
}
```

ここまでは普通でしたが、And と Or は多少の工夫が必要です。それは、And であれば、左辺が 0(false) だったら、もう右辺を評価する必要はない、ということです (Or も同様)。このことが分かるように if 文を使って実現しています。Not についてはそのような複雑さはありませんが、せっかく BinOp を作ったので被演算子が 1 個の場合用の UniOp も作りました。

```
static class And extends BinOp {
  public And(Node n1, Node n2) { super("&&", n1, n2); }
  public int eval() {
    int v = child.get(0).eval();
    if(v == 0) { return 0; } else { return child.get(1).eval(); }
  }
}
static class Or extends BinOp {
```

```
  public Or(Node n1, Node n2) { super("||", n1, n2); }
  public int eval() {
    int v = child.get(0).eval();
    if(v != 0) { return v; } else { return child.get(1).eval(); }
  }
}
abstract static class UniOp extends Node {
  String op;
  public UniOp(String o, Node n1) { op = o; add(n1); }
  public String toString() { return "("+op+child.get(0)+")"; }
}
static class Not extends UniOp {
  public Not(Node n1) { super("!", n1); }
  public int eval() { return child.get(0).eval()==0?1:0; }
}
```

代入ですが、右辺を評価して値を得るところは他の演算と同様で、その後値を vars に格納する必要があります。それには変数名が必要ですが、Var オブジェクトは toString() を呼べば名前の文字列が返されるのでそれを使っています。

```
static class Assign extends Node {
  Var v1; Node n1;
  public Assign(Var v, Node n) { v1 = v; n1 = n; }
  public int eval() {
    int v = n1.eval(); vars.put(v1.toString(), v); return v;
  }
  public String toString() { return v1+"="+n1; }
}
```

Seq ですが、問題に説明したものより使いやすくするため、コンストラクタでは Node を可変引数で受け取ることにしました。つまり Node 型の値をいくつでもパラメタとして書くことができ、受け取る側では 1 つの配列として受け取ります。内部では受け取った配列の各要素を順次 child に add() しています。この「for(変数:　式) …」というのは foreach ループと呼ばれ、「式」は複数の値を順次返すオブジェクト (イテレータ) を生成できるものである必要がある (配列はそのようにできています。また List<*T*>もそうである)。eval() では各ノードを順次実行して最後の値を返します。toString() では並びを中かっこで囲んで、各要素ごとに改行文字をつける (文を並べるのに使うことが多いので、その方が読みやすそうです)。

```
static class Seq extends Node {
  public Seq(Node... a) { for(Node n:a) { child.add(n); } }
  public int eval() {
    int v = 0;
    for(Node n:child) { v = n.eval(); }
    return v;
  }
  public String toString() {
    String s = "{\n";
    for(Node n:child) { s += n.toString() + ";\n"; }
```

```
    return s + "}";
  }
}
```

次は Read と Print です。に前者はコンストラクタで指定した変数の名前を含む文字列をプロンプトとして表示した後、整数を入力し、その値を変数に入れた上で結果としてもその値を返します。後者は式を 1 つ持ち、その式の評価結果を出力します。

```
static class Read extends Node {
  Var v1;
  public Read(Var v) { v1 = v; }
  public int eval() {
    System.out.print(v1+"? ");
    Scanner sc = new Scanner(System.in);
    String str = sc.nextLine();
    int i = Integer.parseInt(str);
    vars.put(v1.toString(), i); return i;
  }
  public String toString() { return "read "+v1; }
}
static class Print extends Node {
  public Print(Node n1) { child.add(n1); }
  public int eval() {
    int v = child.get(0).eval(); System.out.println(v); return v;
  }
  public String toString() { return "print "+child.get(0); }
}
```

最後は while 文と else のない if 文です。どちらも、中では結局 Java の同じ構文を使って実現しています。

```
static class While extends Node {
  public While(Node n1, Node n2) { child.add(n1); child.add(n2); }
  public int eval() {
    int v = 0;
    while(child.get(0).eval() != 0) { v = child.get(1).eval(); }
    return v;
  }
  public String toString() {
    return "while("+child.get(0)+")"+child.get(1);
  }
}
static class If1 extends Node {
  public If1(Node n1, Node n2) { child.add(n1); child.add(n2); }
  public int eval() {
    int v = 0;
    if(child.get(0).eval() != 0) { v = child.get(1).eval(); }
    return v;
```

```
    }
    public String toString() { return "if("+child.get(0)+")"+child.get(1); }
  }
}
```

では、これらを使ってそれらしいプログラムを組み立てて動かしてみましょう。次は N を入力すると 1 から N までの階乗を順次打ち出すプログラムを (抽象構文木として) 組み立て、実行しています。

```
import java.util.*;

public class Sam32 {
  static Map<String,Integer> vars = new TreeMap<String,Integer>();

  public static void main(String[] args) {
    Node prog = new Seq(
      new Read(new Var("x")),
      new Assign(new Var("i"), new Lit(0)),
      new Assign(new Var("p"), new Lit(1)),
      new While(new Lt(new Var("i"), new Var("x")),
        new Seq(
          new Assign(new Var("i"), new Add(new Var("i"), new Lit(1))),
          new Assign(new Var("p"), new Mul(new Var("p"), new Var("i"))),
          new Print(new Var("p")))));
    System.out.println(prog);
    System.out.println(prog.eval());
  }
  // ここに他の static な内部クラスを入れる
}
```

実行例は次のようになります。

```
% java Sam32
{          ← AST の表示
read x;
i=0;
p=1;
while((i<x)){
i=(i+1);
p=(p+i);
print p;
};
}
x? 5     ← Read による入力
1        ←順次階乗を出力
2
6
24
120
```

```
120          ←最終的な値も 120
%
```

演習 3-2 上記のノードを用いて次のようなプログラムを (抽象構文木として) 組み立てて動かしてみよ。

a. N を入力し、N 以下のフィボナッチ数を出力。
b. N を入力し、N の因数をすべて出力。
c. N を入力し、N 以下の素数を出力。
d. N を入力し、N の 2 進表現に「1」のビットが何個あるか出力。
e. その他、自分でやってみたいと思う計算をする。

演習 3-3 上記のノード群に、さらに次のような機能を持つノードを追加してみよ。プログラムを (抽象構文木で) 組み立てて動作を確認すること。

a. C や Java の do-while ループを実現するノード `DoWhile(`ループ本体のノード`,` 条件式のノード`)`。
b. else 部のある if 文を実現するノード `If2(`条件のノード`,` *then* 部のノード`,` *else* 部のノード`)`。
c. 回数を指定してループ本体をその回数だけ繰り返すノード `Times(`回数の式のノード`,` ループ本体のノード`)`。
d. 上記と同様だが、ただし周回ごとに指定した変数が $0, 1, 2, \cdots$ と変化していくノード `TimesFor(`回数の式のノード、変数のノード、ループ本体のノード`)`。
e. 式を指定してその式の値に応じてどれかの選択肢を実行するノード `Switch(`式のノード`,` `new Node[]{`$B_1, B_2, \ldots, B_n$`})`。式の値が 1 なら B_1, 2 なら B_2, 等が実行される (どれでもない場合はどれも実行されない)。
f. 自分で作ってみたいと思うノードを構想し作ってみよ。

3.3 課題 3A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (`y-kuno@uec.ac.jp`) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 3」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

Q1. コンパイラの構成についてどれくらい知っていましたか。また、どの部分にとくに興味がありますか。
Q2. 抽象構文木でプログラムを組み立ててみてどのように思いましたか。
Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

＃4　形式言語

4.1　形式言語入門

4.1.1　形式言語理論の位置付け

形式言語理論 (formal language theory) とは、言語の構文や意味を形式的に (formal に — 定義にもとづき厳密に) 取り扱うための理論全般を指します。

自然言語 (日本語・英語など人間がふだん使う言語) は多様性や曖昧さが大きいため、形式的に扱うのは難しいですが、プログラミング言語に代表されるような人工言語については、形式言語理論に基づいて扱うことで理論的基盤ができ、曖昧さのない扱いが可能になり、処理系も作りやすくなります。

なお、意味の形式的な扱い (形式的意味論) も多く研究されてはいますが、言語処理系を作るという点では、意味まで形式的に扱うことは必ずしも一般的ではありません。そこで、ここでは構文の扱いに限定して説明していきます。[1]

4.1.2　形式言語の諸定義

まず、どのような言語でもそれを組み立てるもととなる文字ないし記号 (の集まり) がなければ成立しません。形式言語ではこれを**アルファベット** (alphabet) と呼びます。

定義 有限個の記号から成る空でない集合 V をアルファベットと呼ぶ。V の要素を並べて得られる記号列を**語**と呼ぶ。

V の要素を 0 個以上並べて得られる語の全体を V^*、1 個以上並べて得られる語の全体を V^+ と記します。また長さ 0 の列を ε と記します。したがって、$V^* = V^+ \cup \{\varepsilon\}$ です。例えば $V = \{a, b\}$ のとき、$V^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, abb, baa, \ldots\}$ となります。そして、**言語**は次のように定義されます。

定義 V^* の任意の空でない部分集合 L を V の上の言語、L の要素を言語 L の**文** (sentence) と呼ぶ。

例えば上の例で $L = \{a, b, aab\}$ とすれば、これら 3 つの語だけが言語 L の文です。しかし「正しい文を列挙する」やり方では有限個の文を持つ L しか定義できません。一方、例えば「a がいくつか並び、続いて b がいくつか並ぶもの」という言語を考えると、これは V^* の部分集合であり、かつ無限の要素を持ちます。これを自然言語に頼らずに定義するにはどうしたらよいでしょう。それには「V^* の要素中でこのような規則にかなうもの」という形で言語を定義します。そのような「規則」を表現するのに、自然言語の代りに**生成文法**の力を借ります。

定義 T、N を互いに共通部分を持たない有限な記号の集まり、P を「$\alpha \to \beta$」(ただし $\alpha \in (T \cup N)^+$、$\beta \in (T \cup N)^*$) の形をした要素の有限集合、S を N の 1 要素としたとき、4 つ組 $G = (T, N, P, S)$ を生成文法と呼ぶ。

ここで T の要素はプログラミング言語などで言えば `if`、`x`、`:=`などプログラムの字面上に現れるトークンに相当し、**終端記号**ないし**端記号** (terminal symbol) と呼ばれます。

[1] ここで説明する形式言語は Noam Chomsky の提唱した文法理論によるもので、「Chomsky 文法」などと呼ばれることもあります。

一方、N の要素はプログラムの字面には現れないが、その構造を説明するのに都合がいいような概念ないし構文要素 (「文」、「代入文」、「変数」など) に対応するものであり、**非終端記号**ないし**非端記号** (non-terminal symbol) と呼ばれます。

そして S は「プログラム」に相当するもので、**出発記号** (start symbol) と呼ばれます。最後に P は**生成規則** (production rule) と呼ばれ、**導出** (derivation) に用いられます。

定義 $u = x\alpha y$、$v = x\beta y$ かつ $\alpha \to \beta \in P$ のとき、u から v が**単導出** (one-step derivation) できると言う ($u \Rightarrow v$ と記す)。$u = u_0 \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_n = v$ $(n \geq 0)$ のとき u から v が導出 (derivation) できると言う ($u \Rightarrow^* v$ と記す)。

そして、文法 G を用いて T 上の言語 $L(G)$ は次のように定義されます。

$$L(G) = \{x \in T^* | S \Rightarrow^* x\}$$

すなわち、出発記号 S から始めて次々に P の規則を適用していって生成される記号列の中で、T の要素のみから成るものの集合が $L(G)$ となります。これが P を生成規則、G を生成文法とよぶ理由です。

生成文法を用いて前出の「a がいくつか並び、続いて b がいくつか並ぶもの」という言語を定義するには次の生成規則を用いればよいのです。

$$P = \{S \to aAbB, A \to aA, A \to \varepsilon, B \to bB, B \to \varepsilon\}$$

例えば「$aabb$」という列は次のようにしてこの P により導出できます。

$$S \Rightarrow aAbB \Rightarrow aaAbB \Rightarrow aabB \Rightarrow aabbB \Rightarrow aabb$$

A や B は最後は ε に置き替わって消えることに注意。以下では当面 T の要素を英小文字、N の要素を英大文字、出発記号を S で表します。

生成文法に対して、各生成規則の形に応じて次のようなクラス分けを行います。

タイプ 0 文法: 上の定義と同じ。つまりもっとも一般的な場合。

タイプ 1 文法: 各生成規則において、左辺の記号列の長さは右辺の記号列の長さを超えないもの。これは実は各生成規則が次の形をしている、というのと等価です (証明は略します)。

$$mAn \to mtn, A \in N, t \in (N \cup T)^+, m, n \in (N \cup T)^*$$

タイプ 1 文法を**文脈依存文法** (context sensitive grammer、CSG) とも呼びますが、これは後者の書き方をした場合に非端記号 A が置き換えられるかどうかはその周囲 (つまり文脈) に何があるかによって決まることによります。

タイプ 2 文法: 各生成規則の左辺が 1 個の非端記号から成る。つまり各生成規則が

$$A \to t, A \in N, t \in (N \cup T)^*$$

の形をしているもの。タイプ 2 文法を**文脈自由文法** (context free grammer、CFG) とも言うが、これは文脈依存文法と対比して、A は周囲に何があろうと (文脈に依存せず) いつでも t に置き換えてよい、という形になっているからです。

タイプ 3 文法: 各生成規則が

$$A \to a \text{ または } A \to aB, A, B \in N, a \in T$$

の形をしているもの。タイプ 3 文法のことを**正規文法** (regular grammer) と呼びます。正規文法によって定義される言語は、それと等価な言語を表現する**正規表現** (regular expression) が存在する、という性質があることから、**正規言語** (regular language) と呼ばれます。

4.1.3 文法と言語の認識/解析

生成文法では G を与えて $L(G)$ の要素を何でも適当に作り出すのは容易です (適用できる P の規則を順に試していけばよい)。しかし言語処理系ではその逆、すなわち T^+ の要素 t(ソースプログラム) を与えて、それが確かに $L(G)$ に属するかどうかを判定すること、さらには S からどのように規則を適用して t が生成されるかを決めることが必要です。一般に前者 (判定) を行うプログラムを $L(G)$ の**認識器** (recognizer) ないし構文検査器、後者 (構造の決定) を行うプログラムを $L(G)$ の**解析器** (parser) と呼びます。

前掲の生成文法のクラスにおいて、後に述べたものほど認識器/解析器をつくるのが容易です。例えば「a がいくつか、続いて b がいくつか」という文法の例はタイプ 2(文脈自由文法) でしたが、これを次のように書き換えるとタイプ 3(正規文法) にできます (同一の言語を定義する生成文法は一般に複数存在し得ることに注意)。

$$P = \{S \to aA, A \to aA, A \to bB, B \to bB, B \to \varepsilon\}$$

図 4.1 に○と◎と矢印から成るグラフを示しましたが、これは先の正規文法の各非端記号が○や◎に、生成規則による置換りが矢印に対応したものとなっています。例えば「$S \to aA$」という規則は S の○から A の○へ a のラベルがついた矢印に対応しています。また、ε が単導出できる場合にはその非単記号は◎で表します [2]。

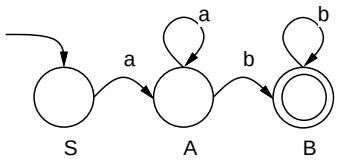

図 4.1: aaa...bbb... に対応する有限オートマトン

次に○と◎をプログラム中の goto で飛んでいけるラベルに対応させ、まず S のラベルから実行開始し、各ラベルの箇所で「次の記号」を読んで、それが a だったら a の矢印に従って A のラベルに飛ぶ、というプログラムを構成したとします。このプログラムに入力を与えて実行させ、もしどこかで入力に対応するラベルが見つからなければ入力は $L(G)$ の要素ではありません。◎のラベルに到着すれば、入力は $L(G)$ の要素であり、これまでにたどった矢印はそれぞれ P の規則に対応していたので、それらを覚えておけば構造もわかったことになります [3]。

このような○と◎とラベルつき矢印から成るグラフを**オートマトン** (automata、自動機械の意) と呼びます。そして、○と◎のことを**状態** (state)、最初にたどり始める状態を**初期状態**、◎の方を**最終状態**、ラベルつき矢印を**遷移** (transition) と呼び、状態の数が有限のものを**有限オートマトン**と呼びます。

ここで示したように、正規文法からは常に対応する (ということは文法が規定する言語を認識/解析する) 有限オートマトンが作り出せ、有限オートマトンは計算機プログラムに変換できるので、結果として正規言語を認識/解析するプログラムが得られることになります。

一般的なプログラミング言語について言えば、その定義には正規文法では制約が強過ぎるため、文脈自由文法を使用するのが普通です。文脈自由文法に対しても、その大部分について、効率良い解析器が作り出せることが知られています。

一方、タイプ 0 文法やタイプ 1 文法の場合にはそれを解析するアルゴリズムは存在しますが、認識する列の長さの 3 乗に比例した計算時間を要します (後でその 1 つを見てみます)。このため、これらの文法を用いてプログラミング言語の構文を記述することはあまりありません。

[2] $A \to a$ の形の規則があった場合には名なしの◎を用意し、A の○からそこへ a のラベルのついた矢印を描く。

[3] ある○から同じラベルをもつ矢印が 2 個以上出ていて行き先が一意でない場合もあり得ます。このような場合の扱いについてはまた別の回にやります。

4.1.4　文脈自由文法とその記法

ここまででは文法の記号を全て英字 1 文字で表してきました。しかし、プログラミング言語を記述する際には各文法記号に「意味のある名前」をつけたくなります。そこで、以下では文脈自由文法を書き記すやり方を次のように改めることにします。

- 各記号を、その意味をよく表すような名前を用いて書き表し、端記号もそのまま直接書く。[4]
- 矢印「→」の代りに「::=」を用いる。また「ε」の代りに何も書かないか、または「`nil`」と書く (キーボードにある文字だけで文法を書き表すため)。
- 同一の左辺をもつ規則はまとめて、右辺を「|」で区切って並べて書く。

この方式で先に出てきた「a がいくつか、b がいくつか」を書いたものを示してきます。

Start ::= SeqA SeqB
SeqA ::= ”a” SeqA | nil
SeqB ::= ”b” SeqB | nil

このような記法をその発明者の名前を取って **BNF**(Backus Normal Form) 呼ぶのでした。

次にこのようにして記した文法と実際の入力の対応関係を示す記法について考えましょう。1 つの方法は導出の系列を逐一記すことですが、非常に長々しく、記号列のどこをどの規則により置き換えたのかわかりづらいという欠点があります。

この問題を解決する方法の 1 つは、導出を**構文木**によって表すことです。構文木とは、(1) 出発記号を根に持ち、(2) 非端記号を中間の節に持ち、(3) 端記号を葉にもち、(4) 各節から葉の方向へ向かう枝はその節を左辺とする導出に対応するような木構造のグラフです。

先の文法での「`aabb`」の導出に対応する構文木を図 4.2 に示します。

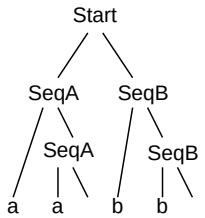

図 4.2: 構文木の例

ただし、構文木では導出列と比べて落ちている情報があります。例えばこの構文木は次のどちらに対応しているとも考えられます。

`Start` ⇒ `SeqA SeqB` ⇒ ”a” `SeqA SeqB` ⇒ ”a” ”a” `SeqA SeqB`
⇒ ”a” ”a” `SeqB` ⇒ ”a” ”a” ”b” `SeqB` ⇒ ”a” ”a” ”b” ”b” `SeqB`
⇒ ”a” ”a” ”b” ”b”

`Start` ⇒ `SeqA SeqB` ⇒ `SeqA` ”b” `SeqB` ⇒ `SeqA` ”b” ”b” `SeqB`
⇒ `SeqA` ”b” ”b” ⇒ ”a” `SeqA` ”b” ”b” ⇒ ”a” ”a” `SeqA` ”b” ”b”
⇒ ”a” ”a” ”b” ”b”

そこで次の定義を行います。

[4]両者の区別をつけるためには、非端記号は`<...>`で囲んで書き端記号はそのまま書く、逆に非端記号をそのまま書き端記号を`"..."`で囲んで書く、どちらも囲まずフォントで区別するなど、様々な流儀があります。

定義 導出の各ステップにおいて、記号列に含まれる非端記号のうち最も左側にあるものを常に置き換える導出を**最左導出** (leftmost derivation)、最も右側にあるものを常に置き換える導出を**最右導出** (rightmost derivation) と呼ぶ。

先の導出系列は前者が最左導出、後者が最右導出です。コンパイラで用いる解析アルゴリズムは通常、このどちらかの導出を扱います。このいずれかであると決まれば、構文木と導出系列は 1 対 1 で対応させられます。

ところで、文法によってはある 1 つの文に対し構文木が 2 つ以上存在することもあります。例えば次の文法を見てみましょう。

```
Expr ::= Expr "+" Expr | Ident
```

この文法で「`Ident + Ident + Ident`」を導出することができますが、その構文木は図 4.3 のように 2 つ存在します。

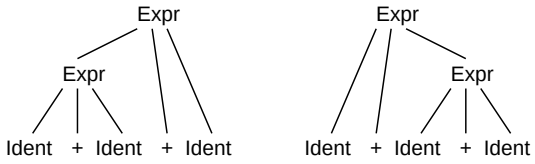

図 4.3: 曖昧な文法の構文木

このような文法を「**曖昧** (ambiguous) である」と言います。反対に、任意の文に対して常に構文木が 1 つだけしか存在しないような文法を「曖昧でない」と言います。通常、構文解析のアルゴリズムでは曖昧でない文法を扱います。

4.1.5 正規文法と正規表現

正規言語は文脈自由言語の特殊な場合なので、BNF で記述できます。しかし正規言語の場合は**正規表現** (regular expression) と呼ばれる、よりコンパクトな記法で表すことが一般的です。正規表現およびそれが表す言語とは次のようなものです。

1. 「ε」は正規表現である。これは空列を表す。
2. a が任意の終端記号のとき、「a」も正規表現である。これは a そのものを表す。
3. α、β が正規表現であれば、「$\alpha\beta$」も正規表現である。これは α が表す列の後に β が表す列を連結したものを表す。
4. α が正規表現であれば、「$\alpha*$」も正規表現である。これは α が表す列を 0 回以上反復したものを表す。

例えば、「a がいくつか、続いて b がいくつか」を正規表現で表すと「$aa*bb*$」となります。

このほか、表現の曖昧さを除くため適宜 () を使用します。さらに読みやすさのため、$\alpha\alpha*$(1 回以上の反復) を $\alpha+$、$(\alpha \mid \varepsilon)$(あってもなくてもよい) を $[\alpha]$ で表すことも多く行なわれます。

正規文法と正規表現の表現能力は等しく、一方で記述された言語は他方でも記述できることが知られています。また従って、正規表現を有限オートマトンに変換することもできます。

4.2 オートマトンによる正規言語の認識

理論の話ばかりでは面白くないので、先に出て来た「正規言語の認識器は容易に構築できる」を実際に確認してみましょう。先の説明では goto でそれぞれの状態にジャンプするということになっていましたが、Java では goto が使えないので、かわりに状態を番号であらわし、次のような形で扱います。

```
    int stat = 0; // 最初の状態は 0 番
    while(true) {
      switch(stat) {
case 0: 状態 0 の処理;
        break;
case 1: 状態 1 の処理;
        break;
        ...
      }
      System.out.println("error."); return;
    }
```

この場合、それぞれの状態の処理の中で stat に次の状態番号を入れて「continue;」で次の周回に進むことで、次の状態の処理に移ります。どの状態にも行けない場合は「break;」に来て switch 文から抜け、エラーを表示して戻ります。

では、図 4.1 に対応するプログラムを示します。入力を 1 文字ずつ調べるのには前に使った CharTok をそのまま利用しています。

```
import java.util.*;
import java.io.*;

public class Sam41 {
  static CharToknizer tok;
  public static void main(String[] args) throws Exception {
    tok = new CharToknizer(args[0]);
    int stat = 0;
    while(true) {
      switch(stat) {
case 0: if(tok.chkfwd("a")) { stat = 1; continue; }
        break;
case 1: if(tok.chkfwd("a")) { stat = 1; continue; }
        if(tok.chkfwd("b")) { stat = 2; continue; }
        break;
case 2: if(tok.chkfwd("b")) { stat = 2; continue; }
        if(tok.chkfwd("$")) { System.out.println("accept."); return; }
        break;
      }
      System.out.println("error."); return;
    }
  }
}

class CharToknizer {
  String str, tok;
  int pos = -1;
  boolean eof = false;
  public CharToknizer(String s) { str = s; fwd(); }
```

```
    public boolean isEof() { return eof; }
    public String curTok() { return tok; }
    public boolean chk(String s) { return tok.equals(s); }
    public void fwd() {
      if(eof) { return; }
      pos += 1;
      if(pos >= str.length()) { eof = true; tok = "$"; return; }
      tok = str.substring(pos, pos+1);
    }
    public boolean chkfwd(String s) {
      if(chk(s)) { fwd(); return true; } else { return false; }
    }
  }
```

3 つの状態がそれぞれ 0、1、2 となります。2 は◎ (最終状態) なので、どこにも遷移がない場合は `tok.chkfwd("$")` で入力の終わりか調べ、終わりなら「成功」と表示して終わります。動かしているところを見てみましょう。

```
% java Sam41 aaabb
accept.
% java Sam41 aaaba
error.
%
```

演習 4-1 例題をそのまま動かしてみよ。動いたら、次のような正規表現の認識器を作成してみよ (正規文法にまず変換し、それからオートマトンを描き、それをそのままプログラムにすること)。

a. $a + b + c+$。
b. $a + (b * |c)d+$。
c. $a + (b + |c) * d+$。
d. $(a + (b + |c) * d+)*$。
e. その他自分で書いた正規表現。

演習 4-2 プログラミング言語に出て来る次のような部分の正規表現を書き、前問と同様にして認識器を作れ。

a. 整数定数。正負の符号もつけられること。
b. 整数定数。16 進法も扱えること。
c. 実数定数。指数部は無くてよい。
d. 実数定数。指数部もつけられること。
e. 名前。名前の定義は自分で考えてよい。
f. その他自分の好きなもの。

4.3 CYK 構文解析アルゴリズム

プログラミング言語の定義には文脈自由文法が使われるわけなので、コンパイラではその文法に対する構文解析が必要となります。コンパイラで構文解析に使われる手法としては、既に学んだ再帰下降

解析がありましたし、そのほか複数のものをこの後取り上げます。しかし、これらはいずれも扱える文法にかなり制限があります。

ここではそのような制約のない、任意の文脈自由言語を扱える構文解析手法として、**CYK**(Cocke-Younger-Kasami、これらは考案した人の名前) と呼ばれるアルゴリズムを見てみます。このアルゴリズムは文脈自由文法のうち CNF(Chomskey Normal Form、チョムスキー標準形) と呼ばれる形のものに対する構文解析 (ないし認識) を行えます。CNF とは、すべての生成規則が次のいずれかの形であるものを言います。

$S \to \varepsilon$
$X \to AB$
$X \to a$

つまり、ε 規則は出発記号 S に対してしか現れず、それ以外の規則はすべて右辺の長さが 2 また 1 で、2 の場合は 2 つとも非端記号、1 の場合は端記号に限られる、ということになります。

制約だらけじゃないかと思うかも知れませんが、すべての文脈自由文法は同等の言語を表すの CNF に変形可能であることが知られています (詳細は略)。このため、CYK アルゴリズムは任意の文脈自由文法に対する認識 (解析) 器となります (そのようなものが存在するという証明となっている)。

では CYK について解説しましょう。CNF では ε 規則はあっても 1 つだけであり、別に扱えば済むので、残りの 2 種類の規則のみを考えます。このアルゴリズムは動的計画と呼ばれる手法に基づいていて、論理値型の 3 次元配列 $\mathtt{p}[j][i][c]$ を使用します。入力列の長さを n としたとき、これらの添字の範囲は次の通りです。

- j — $0 \sim n-1$。入力列の各文字 (端記号) の位置に対応。
- i — $1 \sim n$。記号列の長さに対応。
- c — $T \cup N$(端記号と非端記号の集合) に順に番号をつけたものとして、その番号の範囲に対応。

そして $\mathtt{p}[j][i][c]$ は「入力列の位置 j から始まる長さ i の列 α について、$X \Rightarrow^* \alpha$ の時 $\mathtt{p}[j][i][c]$ が `true`」になるように印をつけていきます。印をつけ終わった時に $\mathtt{p}[0][n][S]$ を見れば、出発記号から入力列が導出できるかどうか分かることになるわけです。

さて、動的計画法のアルゴリズムですが、まず最初は $i=1$ のものを処理します。これは $X \to a$ の形の規則を見て、入力列の j 番の位置が a_j であれば、$X \to a_j$ なるすべての X につて $\mathtt{p}[j][1][X]$ を真にすればよいのです。

以降は長さ $i=2,3,...,n$ について順に処理して行きます。入力列の長さ i の部分列について、それをさらに 2 つの部分列 (それぞれの長さは k と $i-k$、$1<k<i$) に分けます。

i について順に処理していくので、長さ k および $i-k$ については既に配列 $\mathtt{p}$ のデータは完成していることに注意。$X \to YZ$ なるすべての規則について、$\mathtt{p}[j][k][Y]$ も $\mathtt{p}[j+k][i-k][Z]$ も真であれば、入力列の位置 $j \sim j+k-1$ は Y から導出でき、位置 $j+k \sim j+i-1$ は Z から導出できるので、位置 $j \sim j+i-1$ 全体が X から導出できるとわかるため、$\mathtt{p}[j][i][X]$ を真にします。

これを k の範囲 $1 \sim i-1$、j の範囲 $0 \sim n-i$ に渡ってすべて行うことで、長さ i についてすべての場合をチェックできるのです。以上が CYK のあらましです。

たとえば、次の文法を考えてみましょう (CNF になっていることに注意)。

$S \to ST$
$S \to a$
$T \to US$
$U \to b$

この文法について、入力列「$ababa$」を認識させる場合の配列 $\mathtt{p}$ の内容を図 4.4 に示します。

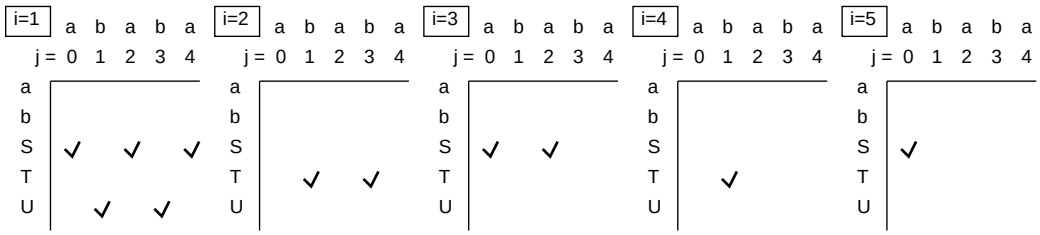

図 4.4: CYK による構文の認識

まず長さ $1(i=1)$ については、$S \Rightarrow a$、$U \Rightarrow b$、なので S と入力の a、U と入力の b に対応する箇所に印がつけられます。

次に長さ $2(i=2)$ のときは、$T \Rightarrow US$ で、U が $j=1,i=1$、S が $j=2,i=1$ で印がついているので、T の $j=1,i=2$ に印をつます。同様に U が $j=3,i=1$、S が $j=4,i=1$ で印がついているので、T の $j=3,i=2$ に印をつけます。それ以外に長さ 2 で印がつくところはありません。

長さ $3(i=3)$ のときは、$S \Rightarrow ST$ で、S が $j=0,i=1$、T が $j=1,i=2$ で印がついているので、S の $j=0,i=3$ に印をつけます。同様に S が $j=2,i=1$、T が $j=3,i=2$ で印がついているので、S の $j=2,i=3$ に印をつます。それ以外に長さ 3 で印がつくところはありません。

以下同様にして長さ 4、5 に記入し、そして最後に、S の $j=0,i=5$ に印があるので、入力列 $ababa$ は文法にあてはまると分かります。[5]

これを実際に Java プログラムにしたものを示しておきます。変数 psym に言語定義で使う端記号・非端記号 (いずれも 1 文字) の並びを与え、その順に番号を割り当てるものとします。ここでは記号は a,b,S,T,U にそれぞれ $0,1,2,3,4$ の番号を割り当てました。また 2 種類の規則はそれぞれ別の 2 次元配列 p1, p2 に格納します。

```
public class Sam42 {
  public static void main(String[] args) {
    String str = args[0], psym = "abSTU"; // a:0, b:1, S:2, T:3, U:4
    int n = str.length(), nsym = psym.length(), start = psym.indexOf("S");
    int[] a = new int[n];
    for(int i = 0; i < n; ++i) { a[i] = psym.indexOf(str.charAt(i)+""); }
    int[][] p1 = { {2,0}, {4,1} };         // S -> a, U -> b
    int[][] p2 = { {2,2,3}, {3,4,2} };     // S -> S T, T -> U S
    boolean[][][] p = new boolean[n][n+1][nsym];
    for(int i = 0; i < n; ++i) {
      for(int r = 0; r < p1.length; ++r) {
        int x = p1[r][0], y = p1[r][1];
        if(a[i] == y) { p[i][1][x] = true; }
      }
    }
    for(int i = 2; i <= n; ++i) {
      for(int k = 1; k < i; ++k) {
        for(int j = 0; j < n-i+1; ++j) {
          for(int r = 0; r < p2.length; ++r) {
            int x = p2[r][0], y = p2[r][1], z = p2[r][2];
```

[5]なお、こうして見ると行列の端記号に対応する列に印がつくことがないので、本当は非端記号の分だけで十分なわけですが、記号に通しで番号をつけるものと考え、全部入れてあります。

```
              if(p[j][k][y] && p[j+k][i-k][z]) { p[j][i][x] = true;}
//            System.err.println(i+":"+j+":"+k+":"+x+":"+y+":"+z+p[j][i][x]);
            }
          }
        }
      }

      System.out.println(p[0][n][start]); // 結果表示
      for(int i = 1; i <= n; ++i) {       // 配列 p 表示
        System.out.println("---- " + i + " -----");
        System.out.println("  " + str);
        for(int s = 0; s < nsym; ++s) {
          System.out.print(psym.charAt(s) + " ");
          for(int j = 0; j < n; ++j) { System.out.print(p[j][i][s]?"t":"."); }
          System.out.println();
        }
      }
    }
  }
```

結果の真偽値出力より後の部分は、配列 p の内容を表示させてみるためのものです。「真偽値 ？　X :　Y」というのは真偽値が真のときは X、偽のときは Y を値とするという演算子 (if-else の演算子バージョン) です。

では動かしてみましょう。入力は「ababa」です。

```
% java Sam42 ababa
true
---- 1 -----
  ababa
a .....
b .....
S t.t.t
T .....
U .t.t.
---- 2 -----
  ababa
a .....
b .....
S .....
T .t.t.
U .....
---- 3 -----
  ababa
a .....
b .....
S t.t..
T .....
U .....
```

```
---- 4 -----
  ababa
a .....
b .....
S .....
T .t...
U .....
---- 5 -----
  ababa
a .....
b .....
S t....
T .....
U .....
%
```

長さ 1 のところで、a は S、t は U から導出できることが示されます。次に長さ 2 のところで、T が 2 つの「ba」を導出できることが示されます。長さ 3 のところで、S から「aba」を導出できることが示されます (2 箇所あります)。長さ 4 のところで T から「baba」が導出でき、これに基づいて長さ 5 のところで S から「ababa」が導出できると分かります。

ところで、前にやった再帰下降解析器は開始記号 (「プログラム」) から初めて下に向かって構文木を組み立てていく方法 (下向き解析) でしたたが、この解析方法は端記号から始めて逆向きに規則を適用していく、**上向き解析**に相当します。

アルゴリズムの効率ですが、コードを見れば分かる通り、入力列の長さを n としたとき、CYK の時間計算量は $O(n^3)$ となります。これは先の再帰下降解析の $O(n)$ に比べるとかなり遅いですがが、そのかわりに。CYK は任意の文脈自由文法を (CNF に書き換えた上で) 解析できるます。

ただし、本物のコンパイラでは巨大な (何万行もの) ソースプログラムを解析するため、$O(n)$ より計算量の大きいアルゴリズムでは実用になりません。プログラミング言語の文法自体もそのことを前提として、($O(n)$ で解析できるように) 設計されているわけです。

演習 4-3 この例題をそのまま動かせ。動いたら文法を次のものに書き換えて正しく認識がなされることを確認せよ。[6]

a. $S \to TU, T \to TV, T \to a, U \to VU, U \to c, V \to b$。
b. $S \to LT, T \to SR, S \to 1, L \to (, R \to)$。
c. $S \to PT, T \to 1, T \to x, T \to LC, L \to (, C \to SR, R \to), P \to SQ, Q \to +$。
d. その他自分で試してみたい CNF 文法。

演習 4-4 これまでに出て来た構文の好きなもの (ただし端記号が 1 文字) を CNF に書き換え、CYK プログラムを手直しして解析してみなさい。

4.4 課題 4A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

[6] 文法を書き換えたらそこに出て来る記号のをすべて psym に入れる必要があることに注意。また記号の番号は psym に入れた文字列の何文字目かで決まることにも注意。

- タイトル – 「システムソフトウェア特論 課題＃ 4」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 形式言語についてどのくらい知っていましたか。また、どの部分にとくに興味がありますか。

  Q2. オートマトンによる認識器や CYK による認識器についてどう思いましたか。

  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

＃5　字句解析

5.1　字句解析とは

5.1.1　字句解析の位置付け

既に学んだように、**字句解析** (lexical analysis) は言語処理系がソースコードを読み込む最初の部分であり、その主な仕事はソースコードを「名前」「定数」「記号」などのかたまり、ないし**トークン** (token) に分割することです。そして、言語処理系の中で字句解析を行なう部分を**字句解析器** (lexcal analyzer)、lexer、tokenizer などと呼びます。

なぜ字句解析器が必要なのでしょう。それは、これまでに見て来たように、BNF などによる文法記述は、変数、定数、記号などに単位を端記号 (ソースコードに出て来る単位) とするのが通例だからです。

ソースコードは文字の並びなわけですから、変数や定数なども非端記号として扱い、「文字」のレベルまで細かく定義することもできます。そのようにすれば、構文解析のときに文字を 1 文字ずつ読めばよいだけなので、字句解析器は不要です。しかし、そうすると次のような問題が現れて来ます。

- 文法が細かくなり読みづらくなる
- 1 文字単位で構文解析の処理を行なうと処理が遅くなる

CPU 性能が高くなった現在では、2 番目の問題はだいぶ軽減されていて、そのため構文と字句をまとめて扱うような処理系も使われるようになってはいます。しかし文法が細かくなることには変わりがないので、ここではまず、字句解析を分けて扱う伝統的な構成を取り上げています。

5.1.2　字句解析器の仕事

しかしそもそも、字句解析器ってそんなに大げさな名前がつくほどの仕事があるのでしょうか?

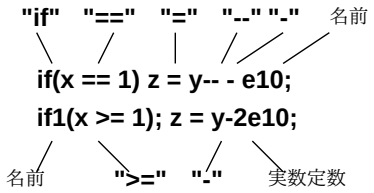

図 5.1: 字句解析の面倒さ

それが、あるのでず。図 5.1 のように、ごく普通のプログラムでも、よく見ないと何の字句か決められないことが沢山あります。「`if(x == 1) z = y-- - e10;`」では、最初の `if` は if 文をあらわす特別な名前です。`==`は「等しい」比較演算子ですが、`=`1 つだけだと代入ですから、`=`がいくつあるかで区分する必要があります。`--`(減少演算子) と`-`(引き算) もそうです。そして、「`if1(x >= 1); z = y-2e10;`」と対比すると、`if1` というのは普通の名前なので、こちらは `x` が 1 以上かどうかの結果をパラメタとして関数を呼び出しています。また、`2e10` は実数定数 (2×10^{10}) ですが、`e10` はただの名前 (変数) です。字句解析器はこれらの区分を正確につけながら、効率よく入力を読み取って行く必要があります。

5.1.3　代表的な字句

ここでは、字句解析器が扱う代表的な字句と、その扱いに際して問題になることを整理して示します。言語によってはそれらの一部が無かったり、扱いが大幅に違うこともありますが、そのような違いには深入りせず、あくまでも「代表的な」プログラミング言語では、ということで進めます。

- **識別子** (identifier) — 識別子とは要するに「名前」のことですが、プログラミング言語の世界では「名前 (name)」という用語を別の意味 (たとえば変数の実体などの意味) で用いることがあるので、それとの混乱を避ける意味で「識別子」を使います。

  識別子とは要するに、「どの変数や手続きであるか」を区別して指定するための文字列です。そして多くの言語では、識別子を「英字で始まり、英数字が並んだ長さ 1 以上の列」としています。ただし、何が英字なのかについては言語で異ります。たとえば C 言語では、英語のアルファベット 26 文字 (A-Za-z) に加えて、下線 (_) も英字としています。識別子に日本語文字 (ひらがな、漢字等) を許す言語もあり、その場合は規則はややこしくなります。[1]

- **予約語** (reserved words) — 予約語とは、識別子と同じ規則で認識できる語だけれど、特別な意味を持っていて通常の識別子としては使えないようなものを指します。まさに「予約」されているわけです。代表的なのは、C 言語における if、while、do、return など制御構造を表すのに使う語です。

  以前は、予約語を設けず、if や return などという名前の変数を使える言語も使われていましたが、言語処理系の作り方が面倒になるので、今では少なくなっています。予約語がある場合は、言語の仕様書にそのことが明記されます。

  字句解析器で予約語を認識するときには、直接認識する方法もありますし、まず識別子を認識して、それから表を検索して予約語だったら予約語として扱う、という方法もあります。

  このほか、一部の識別子を予約語ではないが予め意味の決まっている特別な識別子 — **疑似識別子** (pseudo identifier) として扱う言語もあります。

- **リテラル** (literal) — リテラルというのは「文字通り」という意味ですが、プログラミング言語では数値や文字などの定数を表すのに使います。たとえばソースコードに「`123`」とあれば、それはまさに 123 という整数の値を表すわけです。

  - 数値リテラル — 数値の定数は、整数と実数の 2 種類に分かれます。いずれもプラスやマイナスの符号がつくことがあります (プラス符号はない言語もある)。また、C 言語のように整数では通常の十進のほかに 8 進や 16 進のリテラルを許す言語もあります。
  - 文字 (列) リテラル — 文字や文字列の定数は、通常シングルクォート`'...'`やダブルクォート`"..."`で内容を囲んで表します。文字列しか扱わない言語では、どちらで囲んでもよいものが多いです。C や Java では、シングルクォートは文字リテラル、ダブルクォートは文字列リテラルとして区別しています。
  - 論理値リテラル — `true`、`false` は論理値のリテラルですが、予約語としている言語が多いです。

- **演算子** operator — 演算子はもともとは四則演算を表す`+`、`-`、`*`、`/`など 1 文字のものから始まりましたが、Algol で代入を表すのに`:=`を使うなど、2 文字のものも早くから含まれていました (Algol では`=`は「等しい」に使います)。C 以降は逆に`=`が代入、`==`が等しいですが、さらに`===`を持つ言語もあります。

[1] 以下では日本語文字などは考えないことにします。

- **区切り記号** (delimiter symgol) — 「(」、「)」などのかっこ類や「,」などは演算子ではありませんが (C 言語ではカンマは演算子にも使う)、言語を記述する上で必要な記号です。これは通常、区切り記号と呼びます。区切り記号にも 2 文字以上のものがある言語もあります。
- **空白** (blank space) — 多くの言語では識別子どうしがくっついていてはいけないので、識別子が連続する箇所には空白を入れます。ここで空白といっているのは、タブ文字や改行文字であってもよい場合が多いです。
- **注釈** (comment) — コメントは人間が読むだけでプログラムとしては扱わないので、空白と同等に扱う場合が多いです。コメントの規則も言語によりさまざまなので、扱いに工夫が必要な場合もあります。C 言語の「`/* ... */`」などもなかなか面倒です。

5.2 オートマトンに基づく字句解析

5.2.1 オートマトンに基づく字句解析のあらまし

既に学んだように、トークンの定義は正規表現で表現するのに適していて、さらに正規表現は有限オートマトンに変換でき、有限オートマトンはプログラムで効率よく実装できるのでした。本節ではこの話題をもうすこし詳しく取り上げて行きます。

前に触れたときは、正規表現から正規文法を経て有限オートマトンに変換していましたが、実はもっと直接的に正規表現をオートマトンに変換できます。まずその方法について説明します。

ただし、正規表現から素直に変換した有限オートマトンは**非決定性有限オーマトン** (non-deterministic finite automata, NFA) となります。非決定性というのは、ある 1 つの入力に対する状態遷移が 1 つではなく複数ある場合もある、ということです。

そしてプログラムで効率実装するには、1 つの入力に対して進むべき状態が 1 つでないと困ります。しかし幸いなことに、NFA を決定性有限オートマトン (DFA) に変換することができます。その方法についても説明しましょう。

そして、これらの変換を手で行なうのは非常に大変なので、実際には**字句解析器生成系** (lexcal analyzer generator) を使って正規表現のならびからそれらを認識する字句解析器を生成します。それがどのようなもので、どんな風に使うかも、経験しておきましょう。

5.2.2 正規表現から NFA への変換

ではまず、正規表現を NFA に変換する方法から見てみましょう。そのために例として、整数・実数の数値定数を定義する正規表現を図 5.2 に示します。これからこれを、NFA に変換してみます。

```
IConst = [ Sign ] Digit+
RConst = [ Sgin ] Digit+
           ( Expt [ Sign ] Digit+ | Dot Digit+ [ Expt [ Sign ] Digit+ ])
Sign   = "+" | "-"
Expt   = "E" | "e"
Digit  = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Dot    = "."
```

図 5.2: 数値定数を定義する正規表現

まず、IConst と RConst をそれぞれ「素直に」オートマトンに変換したものを図 5.3 に示します。

これらのオートマトンによってある文字列が実定数であるかどうかを判定できます。しかし実際に必要なのは、入力の文字の並びを順に見ていって「どの種のつづりがあったか」を知ることですね。

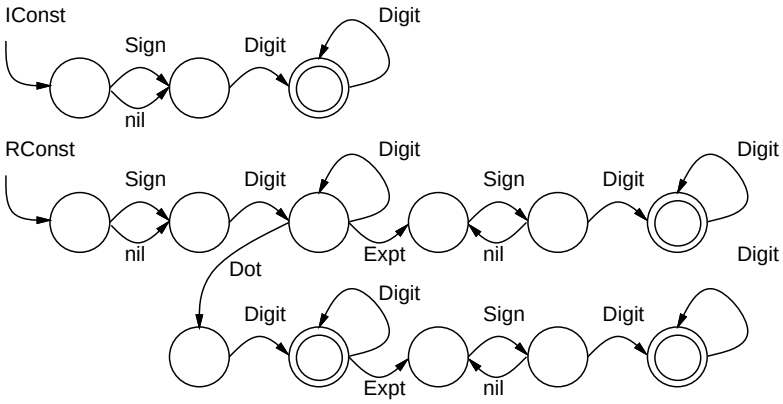

図 5.3: 図 3.1 に対応する有限オートマトン

それには、図 5.4 のように「本当の」初期状態を付加し、そこから各トークンを認識するオートマトンの初期状態に向かう空遷移 (空の入力列に対応する遷移) を加えます (併せて、各最終状態にそれはどのつづりに対応しているかの情報をつけ加えます)。

しかし、図 3.3 のようなオートマトンはそのままプログラムに変換して字句解析器とすることはできません。なぜなら、初期状態から出ている多数の空遷移のうちどれを選んでいいか分からないからです (例えば識別子などは次の 1 文字が英字かどうかでそちらへ行くかどうか決められますが、整定数と実定数の場合はずっと先まで見ないと区別できません)。

このように次の状態が一意に決まらない有限オートマトンが NFA です。図 5.3 のオートマトンも実は NFA です (空遷移を進むかどうかは常に非決定的なので)。

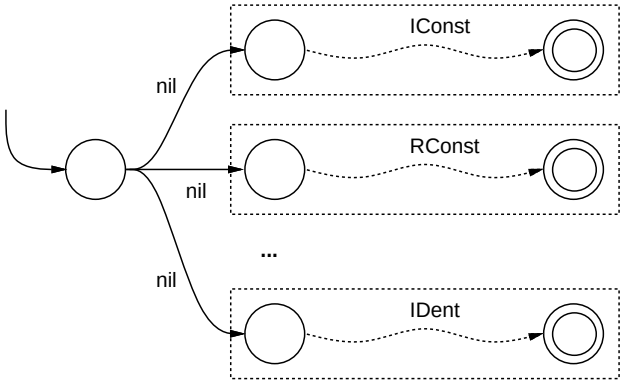

図 5.4: トークン認識のための有限オートマトン

これに対し、次の状態が一意に決まっているオートマトンを決定性有限オートマトンとよぶ。前章で述べた、容易にプログラムに変換できる有限オートマトンとは実は DFA のことである。しかし悲観するには及ばなくて、NFA を DFA に変換する方法が知られている。その前に、まず正規表現を NFA に機械的に変換する方法から見てみましょう。

図 5.3 の非決定性有限オートマトン (NFA) は図 5.2 の正規表現から手で構成しましたが、空遷移を多数作ってもかまわなければ、機械的にやるのは簡単です。そのやり方を図 5.5 に示します。

まず初期状態と最終状態を用意します。次に、文字に対応する正規表現の場合、初期状態から最終状態へのその文字による遷移をつくればすみます。

これ以外の場合は全て、既存の NFA を空遷移でつなげて行きます。例えば列 XYZ に対応する NFA では、正規表現 X、Y、Z に対応する NFA をつくり、それぞれ前のものの最終状態から次のものの初期状態への空遷移をつなげます。全体の初期状態からは X の初期状態への空遷移、Z の最終状態から

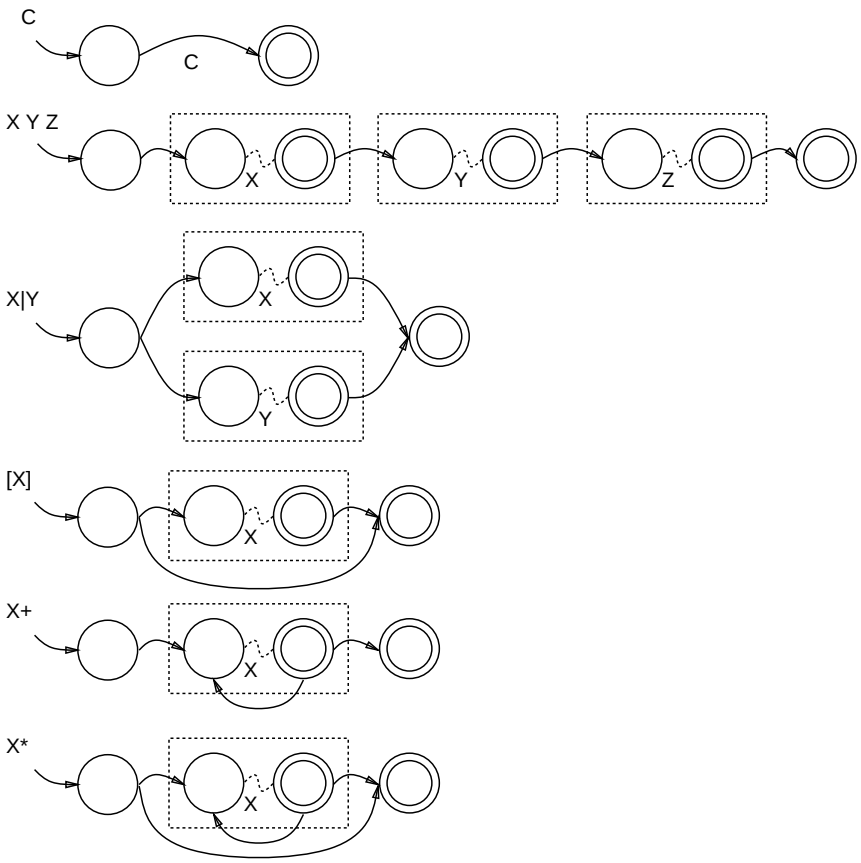

図 5.5: 正規表現を機械的に NFA に変換する方法

は全体の最終状態への空遷移を作ります。他のものも同様です。なお一般の NFA では最終状態は複数個有り得ますが、この方法で作っている限り、最終状態も初期状態同様 1 個だけです。

5.2.3 NFA から DFA への変換

次に NFA を DFA に変換するわけですが、その基本的なアイデアは次のようなものです。NFA では、ある状態である文字が来たときに進む「次の状態」が一般に複数個存在します。そこで、「NFA の状態の集合」をそれぞれ新たに 1 つの状態であると考えて有限オートマトンを構成します。すると、ある「もとの NFA の状態の集合」において、ある文字が来たときに進むことができる「もとの NFA の状態の集合」は (集合として)1 つですから、これは結果として DFA になります。

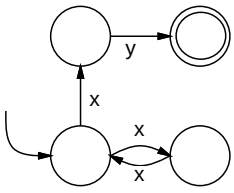

図 5.6: 簡単な NFA の例

例えば、図 5.6 のオートマトンは x が奇数個続いて最後に y がある語のみを受理しますが、最初の x が来たとき行く先が 2 つあるので NFA です。次に図 5.7 を見ると、この有限オートマトンの各状態は先の NFA の状態の集合 (斜線で塗られたもの) から成っています。初期状態は左の箱、つまり元の NFA の初期状態だけが塗られたものです。

ここで x が来ると、元の NFA で行ける両方の状態が塗られた状態、つまり中央の箱へ来きます。

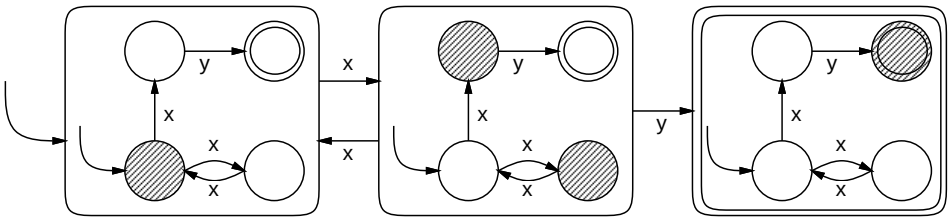

図 5.7: NFA の状態集合を状態とする DFA

ここで y が来ると、元の NFA で行けるのは右上の状態だけですから、それだけが塗られた右側の箱に来ます。一方 x が来た場合は、元の NFA で行けるのは初期状態ですから、左側の箱へ戻ります。

これで、無事全ての入力について行き先が一意に定まった有限オートマトン、つまり DFA が構成できました。最終状態は元の NFA での最終状態を含んだもの、つまり右側の箱だけとなります。

なお、この方法で生成した DFA は冗長な状態を持つことがあります。たとえば、図 5.8 は $(b+|c)aa$ を受理する DFA の例ですが、左も右も同じように動作するものの、左の方が状態が多くなっています。a がいちど現れた先は同じなので、そこは共通にすれば状態が減らせるわけで、右のものはそのようになっています。これを状態の最小化と言います。

状態の最小化を行なうアルゴリズムの概要だけ説明しておきます。DFA の中で互いに区別すべき状態をグループ化し (最初は異なる記号に対応する最終状態群とそれ以外のすべての状態群のグループができる)、次に繰り返し、異なるグループに進むような状態を分割していき、これ以上分割できなくなったところで各グループを 1 つの状態とすることで、最小の状態を持つ DFA を得ることができます。

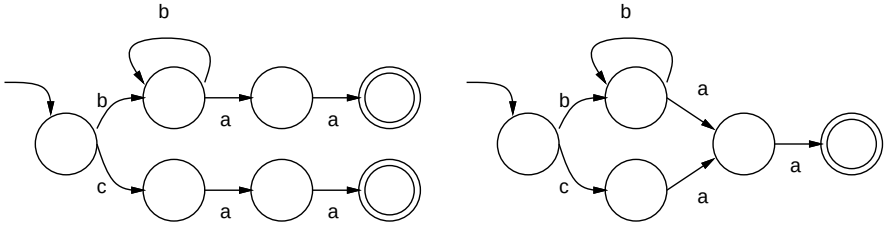

図 5.8: 冗長な状態を持つ DFA の例

5.3　字句解析器生成系

ここまでに述べた有限オートマトンの原理を利用して、実用的な字句解析器を自動生成するツールが多数作られています。Unix に含まれている字句解析器生成ツール **Lex**、それと同一仕様でつくられフリーソフトとして配布されている **Flex** などが代標的です。

これらはいずれも、正規表現から有限オートマトンを生成し、その構造を表の形で出力し、これと入力を走査しながらこの表をたどる (解釈実行する) ドライバルーチンを組み合せることで字句解析器を構成します。これらのツールで生成した字句解析器では、各正規表現ごとに、それを認識した際実行するプログラムの断片が指定できます。

Lex/Flex は C 言語のソースコードを生成しますが、この科目では Java を使っているので、同じ原理で Java による字句解析器を生成するフリーソフトである JFlex を取り上げ、どのようなものか体験してみます。

次に、ごく簡単な JFlex のソースファイルを示します。

```
%%
%class Lexer
```

```
%int
L = [A-Za-z_]
D = [0-9]
Ident = {L}({L}|{D})*
Iconst = [-+]?{D}+
String = \"(\\\"|[^\"])*\"
Blank = [ \t\n]+
%%
{Blank}   { /* ignore */ }
while     { return 4; }
{Ident}   { return 1; }
{Iconst}  { return 2; }
{String}  { return 3; }
```

JFlex では (Lex もそうですが)、「%%」でソースファイルをいくつかのセクションに区分し、また行の先頭にある「%名前」でさまざまな設定を行ないます。

最初のセクションは生成される Java ソースにそのまま取り込まれるので、package 文や import 文を書くのに使いますが、今回はとくに何も必要ないのでからっぽです。

2 番目のセクションの冒頭で、生成されるクラスの名前を Lexer とし、またトークンを取り出すメソッド yylex() の返値の型を int に指定しています。その後は、本体定義を書くのに利用するマクロ (定義) 群で、まず英字 L と数字 D を定義し、それを用いて「識別子 (英字のあとに 0 個以上の英数字)」「整数定数 (符号があってもなくてもよく、その後に数字の並び)」を定義しています。その次は文字列リテラルで、「"があり、\"または"以外の文字が 0 個以上あり、「"がある」と読みます。最後は空白で、「空白文字、タブ文字、改行文字の並び」です。

3 番目のセクションが本体部分で、上での定義を利用しつつ (しなくてもよい)、次の形で動作を指定します。

正規表現 { Java コード }

ここで「Java コード」はトークンを返す関数 yylex() の中で特定のトークン (正規表現にあてはまる入力) が認識されたときに実行されるので、「return 1;」などと書くと yylex() から 1 が返されます。逆に return を書かないと引続き次のトークンの認識に進むので、「無視する」動作になります。

そういうわけで、ここでは空白は無視し、識別子、整数定数、文字列はそれぞれ 1、2、3 を返すようにしています。また、予約語の例も入れたかったので、while が来たら 4 を返します。このように、字句解析器では「どの種類のトークンか」という区分を返せばよいわけです。

これを処理してクラスファイルを作るには、まず jflex を動かし、エラーがなければ生成されている Lexer.java をコンパイルします。

```
% jflex sam51.jflex
Reading "sam51.jflex"
Constructing NFA : 42 states in NFA
Converting NFA to DFA :
..............
17 states before minimization, 14 states in minimized DFA
Old file "Lexer.java" saved as "Lexer.java~"
Writing code to "Lexer.java"
% javac Lexer.java
%
```

では次に、これを呼び出す Java プログラムを見てみます。

```
import java.util.*;
import java.io.*;

public class Sam51 {
  public static void main(String[] args) throws Exception {
    Lexer lex = new Lexer(new InputStreamReader(System.in));
    while(true) {
      int tok = lex.yylex();
      if(tok == Lexer.YYEOF) { break; }
      System.out.printf("%d %s\n", tok, lex.yytext());
    }
  }
}
```

クラス Lexer のインスタンスを生成するときに、Reader オブジェクトを渡す必要があります。Reader は java.io パッケージに定義されているインタフェースで、文字単位での入力機能を持つオブジェクトを定めています。具体的な Reader の種類として、ここでは System.in(InputStream オブジェクト) を元にした Reader を作り出すため InputStreamReader を使いました。

その先はすぐ無限ループで、メソッド yylex() を呼んでトークンを 1 つずつ取り出して行きます。そのトークンが Lexer で定義している定数 YYEOF と等しいならファイルの終わりです。そうでないなら、トークン番号とそのトークンに対応する文字列 (yytext() を呼ぶと返される) を表示します。

では実際に動かしてみましょう。

```
% javac Sam51.java
% java Sam51
abc 123 while awhile 123while
1 abc    ←識別子
2 123    ←整数
4 while  ←予約語 while
1 awhile ←これは識別子
2 123    ←整数と while がくっついていても
4 while  ←トークンとしては分離される
"abc" "a\"bc" """"
3 "abc"   ←文字列
3 "a\"bc" ←「"」を含む文字列
3 ""      ←空文字列
3 ""
"aa       ←改行を含む文字列も OK
bc"
3 "aa
bc"
^D%       ← Ctrl-D を打つとファイルの終わりになる
```

演習 5-1 JFlex の例題を自分でも打ち込んで動かしてみよ。動いたら、次のような機能を追加してみよ。

a. 実数の数値定数を追加する。

b. さまざまな演算記号類やかっこ、カンマなどプログラミング言語の定義で必要なものを字句として追加する。
c. コメントを無視する機能。コメントの形式としては好きなものを使ってよい。
d. とくに何もしなければ、現在ソースコードの何行目にいるかは分からない。JFlex に組み込みの行番号機能もあるが、改行文字を独立したトークンとして認識することで何行目にいるかを併せて表示するようにしてみよ。
e. そのほか、JFlex のマニュアルなどを見て興味深いと思った機能があれば使ってみよ。

5.4 ハンドコーディングによる字句解析

実は字句解析というのはそれほど「難しい」作業ということはないので、字句解析部を全て手で書く、というのも十分実用的な方法です。実際、Lex などのツールによって生成された字句解析器は有限オートマトンを表現するデータ構造を入力に従ってたどりながら動作する、いわばインタプリタの形になります。一方、手で字句解析器を書いた場合には必然的に、現在プログラム上のどこを走っているかが状態に対応するので、実行速度の面ではこの方が有利です。

また、全てをプログラムとして書くわけなので、字句の認識と並行して様々な処理を行わせることができます。例えば数字を読み進めるのと並行して整定数の値を計算したり、名前の各文字を読み進めながらそれを文字列領域にコピーしたりできます。生成ツールに頼った場合にはこれらの文字はツールが定めた場所に蓄積され、つづりが認識された時点で改めて値を計算したりコピーを行うことになるので、結局頭から 2 回文字列を処理することになります。ソースコード中の名前や数値の数は非常に多いので、この差は結構無視できません。

この方法をとる場合には有限オートマトンがあまり複雑でないことが必要なので、Lex の場合のように予約語の認識を有限オートマトンによって行わせるのは困難です。そこで前述のように、予約語はいったん識別子として認識され、その後で簡単な表を引いて予約語かどうかを調べる方法が一般に使われます。

このように手で字句解析器を書く場合でも、どのようなものをトークンとして認識すべきかを正確に規定することはどのみち欠かすことができません。したがって、字句を正規文法や正規表現などで定義する、という道具立ては、このような場合にも十分役に立つのです。

では JFLex による字句解析器と同様に使える字句解析器を作ってみましょう。ここでは簡単のため、文字列は省略し、また符号つき整数の認識が単純化してあります。

```
import java.util.*;
import java.io.*;

public class Sam52 {
  public static void main(String[] args) throws Exception {
    Lexer lex = new Lexer(new InputStreamReader(System.in));
    while(true) {
      int tok = lex.yylex();
      if(tok == Lexer.YYEOF) { break; }
      System.out.printf("%d %s\n", tok, lex.yytext());
    }
  }
}
class Lexer {
  public static final int YYEOF = -1;
  Reader rd;
```

```
  int nc;
  String text = "";
  HashMap<String,Integer> map = new HashMap<String,Integer>();
  public Lexer(Reader r) throws Exception {
    rd = r; nc = rd.read(); map.put("while", 4);
  }

  public int yylex() {
    try {
      while(nc == ' ' || nc == '\t' || nc == '\n') { nc = rd.read(); }
      text = "";
      if(nc == YYEOF) {
        // do nothing
      } else if(alpha(nc)) {
        text += (char)nc; nc = rd.read();
        while(alpha(nc) || digit(nc)) { text += (char)nc; nc = rd.read(); }
        if(map.containsKey(text)) { return map.get(text); }
        return 1;
      } else if(sign(nc) || digit(nc)) {
        text += (char)nc; nc = rd.read();
        while(digit(nc)) { text += (char)nc; nc = rd.read(); }
        return 2;
      } else {
        System.err.printf("%x: invalid char\n", nc);
        nc = rd.read(); return yylex();
      }
    } catch(IOException ex) { }
    nc = YYEOF; return YYEOF;
  }

  public String yytext() { return text; }
  private boolean alpha(int c) {
    return c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z';
  }
  private boolean digit(int c) { return c >= '0' && c <= '9'; }
  private boolean sign(int c) { return c == '+' || c == '-'; }
}
```

メインの方は先と変わっていません。Lexer が作成したクラスです。コンストラクタでで Reader を受け取り、変数の初期設定をします。変数 map には、予約語のスペルと対応するトークン番号を登録しますが、ここでは while だけを登録しています。変数 nc には常に「次の 1 文字」が入っているようにしたので、コンストラクタ中でその 1 文字を読んでいます。

ほとんどの処理はメソッド yylex() の中で行なわれます。全体の構造としてまず、入力時の例外を補足するため try…catch で全体を囲み、例外が出た場合は最後に来て EOF を返します。最初に不要な空白類をスキップし、そのあと次の文字で分岐します。

- EOF なら下に抜けて末尾の処理に合流します。

- 英字なら名前を認識しますが、認識し終わったところで map を検索し、入っていれば格納されている番号を返し、そうでない場合は 1 を返します。
- 符号と数字なら数字を読み取って 2 を返します。

演習 5-2 上の例題をそのまま動かしなさい。動いたら、次のような改良を施してみなさい。

a. トークンとして「(」「)」「=」を追加する。番号は適当に定めてよい。
b. トークンとして「>=」「<=」「!=」「==」を追加する。もちろん「=」も使えること。
c. 例題では「+」「-」だけでも数値定数になってしまう。それはよくないので、「+」「-」は単独の演算子、後ろに数字がくっついていれば数値の符号というふうにしなさい。
d. 文字列が取れるようにしなさい。
e. そのほか、やってみたいと思う機能を追加しなさい。

5.5 課題 5A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 5」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

Q1. 字句や字句解析についてどのように感じましたか。
Q2. オートマトンに基づく字句解析生成系・ハンドコーディングによる字句解析機についてどう思いましたか。
Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

＃6　構文解析 (1)

6.1　文脈自由文法の解析手法

構文解析はコンパイラの中で単なる 2 番目のフェーズ、というよりはだいぶ重要な位置を占めます。というのは、構文解析部はコンパイラの認識部の中枢であり、構文解析部が各構文要素を認識するのに合せて種々の動作が駆動されるようにコンパイラ (または、少なくともそのフロントエンド部) を構成することが多いからです。既に繰り返し出て来たように、今日のコンパイラでは**文脈自由文法**によって言語の構文を定義し、それに沿ってソースプログラムの構造を認識します。

ここで、任意の文脈自由文法を扱うことができればよいのですが、CYK のところで見たように、任意の文脈自由言語の認識は文 (=プログラム) の長さ n に対して $O(n^3)$ の時間計算量となります。コンパイラが受け付けるプログラムは、もちろんそのプログラムの複雑さにもよりますが、非常に長くなることも珍しくないので、このような時間計算量は到底受け入れられません。

正規文法のところで、正規言語であれば効率のよい解析器 ($O(n)$ のもの) が作れる、という説明をしましたが、実際には、文脈自由言語であってもある程度の限定があれば、同様に $O(n)$ の解析器を作ることができます。以下ではそのような限定としてどのようなものがあるかを説明しつつ、代表的な解析アルゴリズムを説明していきます。

具体例があった方がわかりやすいので、ここでは例として、次のような簡単な言語を考えましょう。なお、肩字で番号がふってあるのは、あとで生成規則を番号で参照するためです。

$$
\begin{aligned}
&\mathit{Program} ::= \mathit{StatList}^{1}\\
&\mathit{StatList} ::= \mathit{Stat}\ \mathit{StatList}^{2} \mid \texttt{nil}^{3}\\
&\mathit{Stat} ::= \mathit{Ident}\ \texttt{=}\ \mathit{Expr}\ \texttt{;}^{4} \mid \texttt{read}\ \mathit{Ident}\ \texttt{;}^{5} \mid \texttt{print}\ \mathit{Expr}\ \texttt{;}^{6}\\
&\qquad \mid \texttt{if (}\ \mathit{Cond}\ \texttt{)}\ \mathit{Stat}^{7} \mid \texttt{\{}\ \texttt{StatList}\ \texttt{\}}^{8}\\
&\mathit{Cond} ::= \mathit{Expr} < \mathit{Expr}^{9} \mid \mathit{Expr} > \mathit{Expr}^{10}\\
&\mathit{Expr} ::= \mathit{Ident}^{11} \mid \mathit{Iconst}^{12}
\end{aligned}
$$

解析部の出力としては、当面構文木を生成するものとします。そこでさらに具体例として、次のプログラムが入力されたとき、これに対応する構文木を手で組み立てみてください (ここで時間を取ってやってみてください)。

```
read x; read y; if(x > y) { z = x; x = y; y = z; } print x; print y;
```

演習　上記のプログラムを前述の文法にあてはめて構文木を描きなさい。

どうだったでしょうか。結果は図 6.1 のようになるはずです。ここで構文木を組み立てる過程を振り返ってみると、おおむねね上 (根) の方から描いていくか、または下 (葉) の方から描いていくかのどちらかだと思われます。

文脈自由文法の解析アルゴリズムも同様に分類でき、前者を**下向き解析** (top-down parsing)、後者を**上向き解析** (bottom-up parsing) と呼びます。今回は以下、下向き解析について説明します。

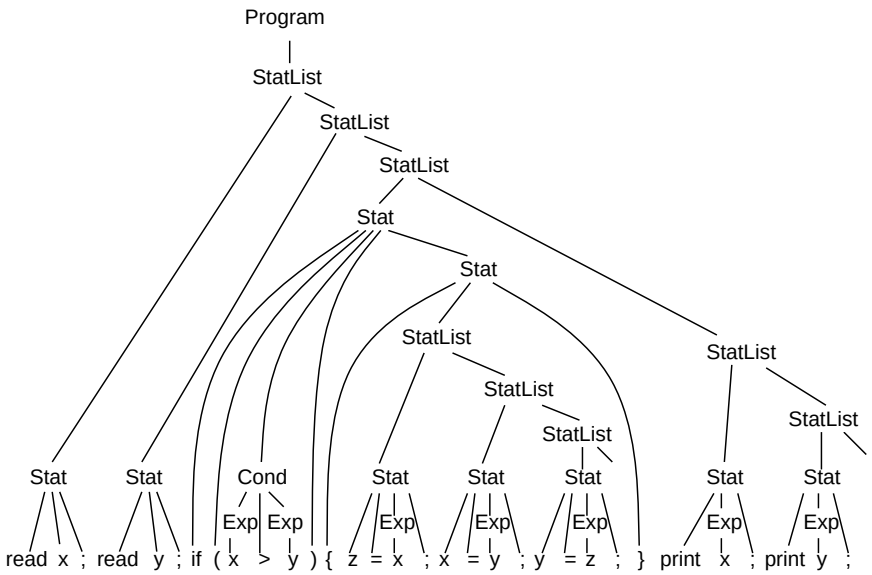

図 6.1: プログラムに対応する構文木

6.2 下 向 き 解 析

6.2.1 下向き解析と First/Follow

それではここで、図 6.1 の構文木を下向きに描く過程を少し細かく見てみます。まず構文木の根は出発記号 $Program$ に決まっています。次に $Program$ を左辺にもつ規則は 1 つしかないから、$StatList$ が根の直下の節となります。次ですが、ここで $StatList$ は 2 通りに置換できるので、そのどちらかを選択する必要があります。まず $Stat\ StatList$ に置換していいか考えましょう。その場合、入力の最初に $Stat$ がないといけませんが、具体的には最初の入力記号は `read` ですね。一方 $Stat$ は `read Ident ;` に置換できるので、こちらが正しそうだとわかります。そこで $Stat\ StatList$ を選び、この $Stat$ を `read Ident ;` に置換すると、これらが入力の最初の 3 つのトークンに対応します。次は先の $Stat\ StatList$ の $StatList$ 以下の対応に戻り、4 トークン目の `read` 以下とこれを対応させ…のように進めばよいでしょう。

これを見て「構文規則は有限だから可能なものを順にあてはめて入力との対応を検査していけばよさそうだ」と思うかも知れませんが、それでは困ります。なぜならそれだと「やってみてだめなら戻って別の枝を試す」(バックトラックする) ことになるので、入力トークンの数を n として、解析にかかる計算量のオーダが $O(n)$ より大きくなってしまいます。

そうではなく、「$StatList$ をどちらに置換するか」などの選択肢が現れたとき、先の方まで試してみることなく正しい選択を行う必要があるのです。上の過程を振り返ると、$Stat\ StatList$ を展開して行った先が `read`…になるから、こちらの枝を選んだのでした。そして、構文記号は有限個しかないので、全ての構文記号 A について「展開していくとどんな端記号から始まり得るか」の集合 (これを $First(A)$ と記す) をあらかじめ計算しておけます。併せて、各記号 A ごとに「その後に来ることができる端記号の集合」(これを $Follow(A)$ と記す) も計算しておくことにします (その用途は後述)。

6.2.2 First/Follow の計算

本節では $First$ と $Follow$ の計算方法を示します (プログラムにすると大変なので、アルゴリズムの形で説明します)。まず準備として、$Emp = \{X | X \in N \wedge \Rightarrow^* \epsilon\}$、つまり空列が導出可能な非端記号の集合を求めておきます。これは次の手順によりできます。

- Emp に $X \to \epsilon$ であるような生成規則を持つ非端記号 X をすべて入れる

- Emp がこれ以上変化しなくなるまで繰り返し
- 　$X \to Y_1 Y_2 \cdots Y_N$ において $^{\forall} Y_i \in Emp$ であるような生成規則を持つ X を Emp に追加
- 以上を繰り返し

ではこれを用いて、$First(X)$ は次のようにして求められます (なお、$\epsilon \in First(X)$ とは $X \in Emp$ であることを意味します)。

- 端記号なら、$Firxt(X) = \{X\}$ とする
- 全ての非端記号 X について $First(X) = \{\}$ とする
- $X \in Emp$ であれば、$First(X)$ に ϵ を追加
- すべての $First(X)$ が変化しなくなるまで繰り返し
- 　$X \to Y_1 Y_2 \cdots Y_N$ において i を $1 \cdots N$ の順に繰り返し
- 　　$First(X)$ に $First(Y_i) - \{\epsilon\}$ の各要素を追加
- 　　$Y_i \notin Emp$ なら繰り返しを抜け出す
- 　以上を繰り返し
- 以上を繰り返し

要するに、ある記号の先頭に来る端記号はその記号を左辺とする生成規則の右辺の先頭に来る端記号ですが、その先頭の記号が空列になり得るならその次、それも空列になり得るならさらにその次、…の先頭も加えるということです。

上では 1 つの記号 X について求めましたが、記号列 $X_1 X_2 \cdots X_N$ についての $First$ も次のように定めておきます。

- $First(X_1 X_2 \cdots X_N)$ を $\{\}$ とおく
- $N = 0$ または $^{\forall} X_i \in Emp$ であれば、$First(X_1 X_2 \cdots X_N)$ に ϵ を追加
- i を $1 \cdots N$ の順に繰り返し
- 　$First(X_1 X_2 \cdots X_N)$ に $First(X_i) - \{\epsilon\}$ の各要素を追加
- 　$X_i \notin Emp$ なら繰り返しを抜け出す
- 以上を繰り返し

$Follow(X)$ については X が非端記号のときだけ定義されます。その計算のアルゴリズムは次の通りです。

- すべての非端記号 X について $Follow(X) = \{\}$ とする
- X が開始記号であれば、$Follow(X)$ に\$ (入力終わりの印) を追加
- すべての $Follow(X)$ が変化しなくなるまで繰り返し
- 　$X \to Y_1 Y_2 \cdots Y_N$ においてi を $1 \cdots N$ の順に繰り返し
- 　　Y_i が非端記号でなければ、次の周回に進む
- 　　$Follow(Y_i)$ に $First(Y_{i+1}, \cdots Y_N) - \{\epsilon\}$ の各要素を追加
- 　　$\epsilon \in First(Y_{i+1}, \cdots Y_N)$ なら、$Follow(Y_i)$ に $Follow(X)$ の各要素を追加
- 　以上を繰り返し
- 以上を繰り返し

では実際に、先に出て来た文法で $First/Follow$ がどうなるかを見てみましょう。

- $First(Prog) \to \{$ `if print read` $Ident$ `nil`
- $First(StatList) \to \{$ `if print read` $Ident$ `nil`

- $First(Stat) \rightarrow$ { `if print read` $Ident$
- $First(Cond) \rightarrow Ident$
- $First(Expr) \rightarrow Ident$
- $Follow(Prog) \rightarrow$ $
- $Follow(StatList) \rightarrow$ $ }
- $Follow(Stat) \rightarrow$ `$ }` { `if print read` $Ident$
- $Follow(Cond) \rightarrow$ `)`
- $Follow(Expr) \rightarrow$ `< > ) ;`

6.2.3 LL(1) 構文解析器

何のために $First/Follow$ を求めていたかというと、下向き解析においてどの構文規則を選ぶべきかを判断するためでした。具体的には、非端記号 X を置き換えようとして $X \rightarrow Y_1 Y_2 \cdots Y_N$ という規則が複数あったときに、どれを選ぶかは $First(Y_1 Y_2 \cdots Y_N)$ を見ることで判断できます (より厳密にいえば、$\epsilon \in First(Y_1 Y_2 \cdots Y_N)$ である場合もあるので、その場合には $Follow(X)$ を使います)。

この情報は、予め文法に基づいて計算し、表の形で保持しておきます。このような、構文解析に必要な情報を集約した表のことを**構文解析表** (parsing table) と呼びます。先に出て来た文法と $First/Follow$ をもとに作成した構文解析表を図 6.2 に示しました。

	1 $	2 Ident	3 Iconst	4 (	5)	6 <	7 >	8 {	9 }	10 =	11 ;	12 read	13 print	14 if
Program	1	1						1				1	1	1
StatList	3	2						2	3			2	2	2
Stat		4						8				5	6	7
Cond		9,10	9,10											
Expr		11	12											

図 6.2: LL(1) 構文解析表

ここで説明している方法は、ソースコードを左から (先頭から) 順に (Left-to-right) 見ていき、生成される導出が最左導出 (Leftmost derivation) であり、そして常に「次の 1 記号」を見て動作を決めることから **LL(1)** 解析器と呼ばれています。その具体的な動作方法を説明しましょう。プログラム例を再掲します。

```
read x; read y; if(x > y) { z = x; x = y; y = z; } print x; print y;
```

解析器の構造とその動作を図 6.3 に示します。縦線が 2 本ありますが、その左側はスタックになっていて、左から要素をプッシュ/ポップします。そして右側はトークンが 1 つだけ見えていて、これが入力に現れるトークンです。

スタックに開始記号 (Program) が積まれていて、最初のトークン read が見えている状態から始まります。解析表を見ると、Program/read のところは「1」とあります。なので、生成規則 1(Program ::= StatList) が選択され、スタックから先頭要素 Program をポップして、代わりに右辺 StatList をプッシュします。これが 2 行目です。今度は解析表の StatList/read を見ると「2」ですから、生成規則「StatList ::= Stat StatList」が選ばれ、StatList がポップされてから右辺「Stat StatList」を (右から順に) プッシュします。これで 3 行目へ行きます。

こんどはスタック先頭が Stat なので、Stat/read を見ると「5」ですから、生成規則は「Stat ::= read Ident ;」であり、read がポップされて「read Ident ;」がプッシュされます。今度は先頭が端記

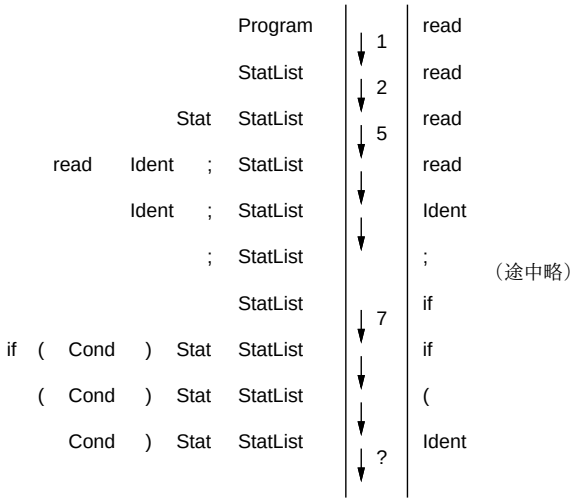

図 6.3: LL(1) 解析器の動作

号であり、それが先読みトークン read と一致しているので、両方とも取り除き (入力は読み進み)、次のトークンは Ident です。これもスタック先頭と一致していますから、再び両方とも取り除きます。次は「;」でこれも両方とも取り除きます。

長いので途中省略して、同様にもう 1 つの read 文も終わったものとして、次は「if」が見えます。スタック先頭は StatList なので、解析表の StatList/if を見ると「2」とあり、生成規則 2 が再度使われて「Stat StatList」となり、Stat/if で「7」が選ばれ、Stat が降ろされて「if (Cond) Stat」が積まれます。そして if、(が読み進められ、次は Cond/Ident です。

さてここですが、9 と 10 が両方書かれていますね。ということは、どちらに行けばよいか分からないわけです。それはそうで、該当する 2 つの生成規則は次のものです。

$$Cond ::= Expr \texttt{ < } Expr \mid Expr \texttt{ > } Expr$$

この 2 つの規則の右辺は先頭が同じものなので、1 トークンだけ見ていたのでは区別できません。つまり、この文法は LL(1) 解析器では解析できない (LL(1) 文法ではない)、ということになります (実際には解析表を作っている段階で分かります)。

実はこの場合は式というのは定数か変数 1 個だけなので、見えるトークンを 2 にすれば次の比較演算子が見えてどちらかは判断できます (つまりこの文法は LL(2) です)。

しかしより一般的には、式は任意の複雑さを持てるので、この方法は使えません。しかしこの問題解決方法はあります。それは、文法を次のように書き直せばよいのです。

$$Cond ::= Expr \ Cond1$$
$$Cond1 ::= \texttt{<} \ Expr \mid \texttt{>} \ Expr$$

このようにすれば、1 番目の規則をとりあえず選んで読み進み、式が済んだところで $Cond1$ を置き換えるところで、次の 1 トークンを見て規則を判別できます。

6.2.4 LL(1) 文法が持つ制約

文法が LL(1) でなくなる場合の 1 つは前述ように適用規則が一意に決まらない場合ですが、他に $A \Rightarrow^+ A\beta$ なる A が存在する場合 (これを文法に**左再帰性**がある、と言います) も、A を展開して行くとまた A になってしまい、無限に同じ規則の適用が続くだけで解析が進まなくなるため、LL(1) 解析器で解析できません。

文法が LL(1) でなくても、言語は同じままで文法を書き換えて LL(1) にできる場合があります。まず適用規則が一意に決まらない場合には、先の例のように共通部分を「くくり出す」ように規則を書き換えればよいです。

また、次のような左再帰性がある場合を考えてみます。

$A \to A\beta$
$A \to \gamma$

これから生成される言語は $\gamma\beta\beta\ldots\beta$ の形になります。そこで生成規則を次のように書き換えれば、言語は同じままで左再帰性のない文法になります。

$A \to \gamma A'$
$A' \to \beta A'$
$A' \to \varepsilon$

このような直接の左再帰でない場合でも、中間に現れる規則を展開して埋め込むことで直接左再帰に書き換えてから同様に処理すればよいのです。このような書き換えで LL(1) にできない文法も存在しますが、プログラミング言語の構文として現れることはまれです。

ただ、文法の書き換えで問題なのは、認識される言語は同じでも文法記述が理解しにくいものとなり、また構文木に沿った後段の処理も記述しにくくなってしまう点です。これについて対応する方法は、再帰下降解析と合わせて説明します。

6.3 LL(1) 解析器

ではここで、解析表を使った LL(1) 解析器を構成してみましょう。木構造を作るのは面倒なので、実際には認識器です。また、先の曖昧さの問題を避けるため、規則 10 は削除しておきます (比較演算子「>」は使わなこととします)。

まず、字句解析は JFlex を使いますが、字句解析と受け渡すトークン番号をこれまでは端記号だけにしていたのに対し、今回は非端記号まで含めて番号を振ります。このため `Symbol` という次のクラスを使います。

```
public class Symbol {
  public static final int NL = 0, EOF = 1, IDENT = 2, ICONST = 3,
    LPAR = 4, RPAR = 5, LT = 6, GT = 7, LBRA = 8, RBRA = 9,
    ASSIGN = 10, SEMI = 11, READ = 12, PRINT = 13, IF = 14,
    Program = 15, StatList = 16, Stat = 17, Cond = 18, Expr = 19;
  public static final int N = 15;
}
```

`EOF` も表にいれる都合上、正の値にしています。また、端記号を前の方にあつめて、その後ろの値を非端記号とし、境界を定数 `N` として用意しました (後で端記号かどうか判定するのに使う)。

次にこれを参照した JFLex のソースファイルを示します。必要な記号類が追加されているだけで、とくに変わったことはありません。

```
%%
%class Lexer
%int
L = [A-Za-z_]
D = [0-9]
Ident = {L}({L}|{D})*
Iconst = [-+]?{D}+
Blank = [ \t\n]+
%%
```

```
{Blank}   { /* ignore */ }
\(        { return Symbol.LPAR; }
\)        { return Symbol.RPAR; }
;         { return Symbol.SEMI; }
\{        { return Symbol.LBRA; }
\}        { return Symbol.RBRA; }
=         { return Symbol.ASSIGN; }
\>        { return Symbol.GT; }
\<        { return Symbol.LT; }
read      { return Symbol.READ; }
print     { return Symbol.PRINT; }
if        { return Symbol.IF; }
{Ident}   { return Symbol.IDENT; }
{Iconst}  { return Symbol.ICONST; }
```

今回はこの Lexer をそのまま使うのでなく、前の回でやった Toknizer と同じインタフェースになるように、下請けとして Lexer を呼ぶクラス Tokenizer を作りました。

```
class Tokenizer {
  Lexer lex;
  int tok, line = 1;
  boolean eof = false;
  public Tokenizer(String s) throws Exception {
    lex = new Lexer(new FileReader(s));
    tok = lex.yylex();
  }
  public boolean isEof() { return tok == Lexer.YYEOF; }
  public int curTok() { return tok; }
  public String curStr() { return lex.yytext(); }
  public int curLine() { return line; }
  public boolean chk(int t) { return tok == t; }
  public void fwd() {
    if(isEof()) { return; }
    try {
      tok = lex.yylex();
      while(tok == Symbol.NL) { ++line; tok = lex.yylex(); }
    } catch(IOException ex) { tok = Lexer.YYEOF; }
  }
  public boolean chkfwd(int t) {
    if(chk(t)) { fwd(); return true; } else { return false; }
  }
}
```

fwd() で複雑なことをやっていますが、おもに改行がきたときに行カウントを増やすためです。このほか、EOF のときに Symbol.EOF を使う必要がありますが、それは main() 側でやるようにしました。

では本体部分です。プログラムの他に、解析表とそれぞれの規則の右辺が必要です。分かりやすさのため、規則は左辺と右辺を並べた「配列の配列」として記述しました。番号 (添字) は先の文法と一

致させてあります。

```
import java.util.*;
import java.io.*;

public class Sam61 {
  static int[][] rules = {
    { },
    { Symbol.Program, Symbol.StatList }, //1
    { Symbol.StatList, Symbol.Stat, Symbol.StatList }, //2
    { Symbol.StatList }, //3
    { Symbol.Stat, Symbol.IDENT, Symbol.ASSIGN, Symbol.Expr, Symbol.SEMI },//4
    { Symbol.Stat, Symbol.READ, Symbol.IDENT, Symbol.SEMI }, //5
    { Symbol.Stat, Symbol.PRINT, Symbol.Expr, Symbol.SEMI }, //6
    { Symbol.Stat, Symbol.IF, Symbol.LPAR, Symbol.Cond,
                   Symbol.RPAR, Symbol.Stat }, //7
    { Symbol.Stat, Symbol.LBRA, Symbol.StatList, Symbol.RBRA }, //8
    { Symbol.Cond, Symbol.Expr, Symbol.LT, Symbol.Expr }, //9
    { Symbol.Cond, Symbol.Expr, Symbol.GT, Symbol.Expr }, //10
    { Symbol.Expr, Symbol.IDENT }, //11
    { Symbol.Expr, Symbol.ICONST }, //12
  };
```

解析表は非端記号のところだけ必要ですが、添字の位置を合わせるため始めの方に空の配列をいれています。また、トークンは 1 から始まるので最初の要素として 0 が詰めてあります。内容は前に示した LL(1) の解析表と同じですが、10 番の規則 (>に対応) は削除してあります。

```
  static int [][] ptab = {
    {},{},{},{}, {},{},{},{}, {},{},{},{}, {},{},{},
    { 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1 },
    { 0, 3, 2, 0, 0, 0, 0, 0, 2, 3, 0, 0, 2, 2, 2 },
    { 0, 0, 4, 0, 0, 0, 0, 0, 8, 0, 0, 0, 5, 6, 7 },
    { 0, 0, 9, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0,11,12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  };
```

最後に main() を見ていただきましょう。最初にスタックに Program をプッシュした状態からはじめ、ループの中でスタックの先頭が端記号か非端記号かで分かれます。非端記号であれば、次のトークンとの一致を確認してスタックを取り降ろし、入力も進めます (不一致ならエラー終了します)。

```
  public static void main(String[] args) throws Exception {
    Tokenizer tok = new Tokenizer(args[0]);
    Stack<Integer> stk = new Stack<Integer>();
    stk.push(Symbol.Program);
    while(stk.size() > 0) {
//    System.out.printf("%s : %s\n", stk.toString(), tok.curStr());
      if(stk.peek() < Symbol.N) {
        if(!tok.chkfwd(stk.pop())) {
```

```
            System.err.printf("token mismatch: %s at %d\n",
              tok.curStr(), tok.curLine()); return;
          }
        } else {
          int t = tok.curTok(); if(tok.isEof()) { t = Symbol.EOF; }
          int r = ptab[stk.peek()][t];
          if(r == 0) {
            System.err.printf("cannot determine rule for %d: %s at %d\n",
              stk.peek(), tok.curStr(), tok.curLine()); return;
          }
          System.out.printf("rule: %d\n", r);
          int[] a = rules[r];
          stk.pop();
          for(int i = a.length-1; i > 0; --i) { stk.push(a[i]); }
        }
      }
    }
  }
  // 後ろに Tokenizer をいれる
```

非端記号の場合は解析表を見て動作を決めます。具体的には、解析表を引くと生成規則番号が分かるので、スタックの先頭は取り降ろして成績規則の右辺の内容を右から順に積みます。もし番号が 0 ならエラーです。

先のプログラムの「>を「<」に変更したもので実行してみました。次のように、確かに生成規則が順に出力されています。

```
% java Sam61 test.min
rule: 1
rule: 2
rule: 5
...
rule: 3
%
```

演習 6-1 例題をそのまま動かせ。簡単なプログラムを何通りか作って動かしてみて、構文木も描いた上で照合し、構文規則番号が正しいことを確認すること。

演習 6-2 条件演算子「>」も扱えるように変更せよ。本文で説明したように文法を変更したものとして、それに対応して解析表や規則を変更すればできる。

演習 6-3 次のような算術式の文法を認識するように LL(1) 解析器を変更してみよ。文法を LL(1) になるよう書き換えてから $First/Follow$ を作り、解析表を作ること。

$$
\begin{aligned}
&Prog \ ::= \ Expr \\
&Expr \ ::= \ Term \ \texttt{+} \ Expr \mid Term \ \texttt{-} \ Expr \\
&Term \ ::= \ Fact \ \texttt{*} \ Term \mid Fact \ \texttt{/} \ Term \\
&Fact \ ::= \ Ident \mid Iconst \mid \texttt{(} \ Expr \ \texttt{)}
\end{aligned}
$$

演習 6-4 前問の文法だと、演算子が右結合になってしまう (なぜか?)。これを避けるためには、次の文法を使えばよい。

$Prog$ ::= $Expr$
$Expr$ ::= $Expr$ `+` $Term$ | $Expr$ `-` $Term$
$Term$ ::= $Term$ `*` $Fact$ | $Term$ `/` $Fact$
$Fact$ ::= $Ident$ | $Iconst$ | `(` $Expr$ `)`

しかし今度は左再帰を含む文法なのでそのままでは LL(1) 解析器を作れない。左再帰を解消するためには次のように書き換えることが 1 つの方法である。

$Prog$ ::= $Expr$
$Expr$ ::= $Term$ $Expr0$
$Expr0$ ::= ϵ | `+` $Term$ $Expr0$ | `-` $Term$ $Expr0$
$Term$ ::= $Fact$ $Term0$
$Term0$ `::=` ϵ | `*` $Fact0$ | `/` $Fact0$
$Fact$ ::= $Ident$ | $Iconst$ | `(` $Expr$ `)`

この文法に対して LL(1) 解析器を構成せよ。

演習 6-5 好きな文法を決めて、その文法を認識する LL(1) 解析器を作れ。

6.4 課題 6A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (`y-kuno@uec.ac.jp`) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 6」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. First、Follow の計算方法と LL(1) 解析器の原理について納得しましたか。
  Q2. 左再帰やその除去など、文法を LL(1) 文法にするための変形について納得しましたか。
  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

＃7　構文解析 (2)

7.1　拡張 BNF と構文図

ここまで BNF は文脈自由文法の定義に忠実に従った形で扱ってきました (「|」については、左辺が同じ複数の定義と同じことなので同等です)。しかし、プログラム言語の構文として考えると、正規表現に類似した次のようなものが書けるとより便利です。

(α)* — α の 0 回以上の繰り返し
(α)$^+$ — α 1 回以上の繰り返し
(α | β) — α または β (右辺の途中での選択肢)
[α] — α があってもなくてもよい

繰り返しの書き方については、$\cdots$ で表すなど別の流儀もあります。一般に、このような追加の記法を取り入れた BNF のことを**拡張 BNF** と呼びます。今回例題として実装する小さな言語の構文を拡張 BNF で記述したものを示します。

$prog$ ::= ($stat$)*
$stat$ ::= $ident$ `=` $expr$ `;` | `read` $ident$ `;` | `print` $expr$ `;`
　　| `if (` $expr$ `)` $stat$ | `while (` $expr$ `)` $stat$ | `{` $prog$ `}`
$expr$ ::= $term$ (`+` $expr$ | `-` $expr$)*
$term$ ::= $fact$ (`*` $term$ | `/` $term$)*
$fact$ ::= $ident$ | $iconst$ | `(` $expr$ `)`

BNF はどうしても数式っぽく見えますが、これと同等のものを図的に表す**構文図** (syntax diagram) または railroad diagram と呼ばれる記法があります。これは Niklaus Wirth が Pascal 言語のマニュアルで始めたものです。

構文図では、1 つの端記号の規則ごとに 1 つの始点と終点を持つ有向グラフを描きます。規則中に現れる端記号は円や長円、非端記号は長方形で表し、それらに間をつながりを表す矢線で結びます。そして、始点から終点まで矢線に沿って通れるとき、記号をその順で並べたものが生成できることを意味します。図 7.1 に上の文法を構文図にしたものを示します。

拡張 BNF でも構文図でも、「または」による部分的な分岐と合流、および規則の途中位置に戻る「ループ」が表せることが BNF と比べた場合の特徴となります。

7.2　再帰下降解析

再帰下降解析 (recursive descent parser) については既に取り上げましたが、その名前通り下向き解析の 1 つの手法であり、手でパーサを構成できるのでツールが使えない場合に有力な選択肢となります。改めて整理すると、その要点は次のようになります。

- 非端記号 N に対応して 1 つの手続き P を作る。
- P の中では生成規則の右辺に対応して入力を読み進める。このとき、端記号は自分でトークンを読み進めるが、非端記号については対応する箇所でその記号に対応する手続きを呼び出す。

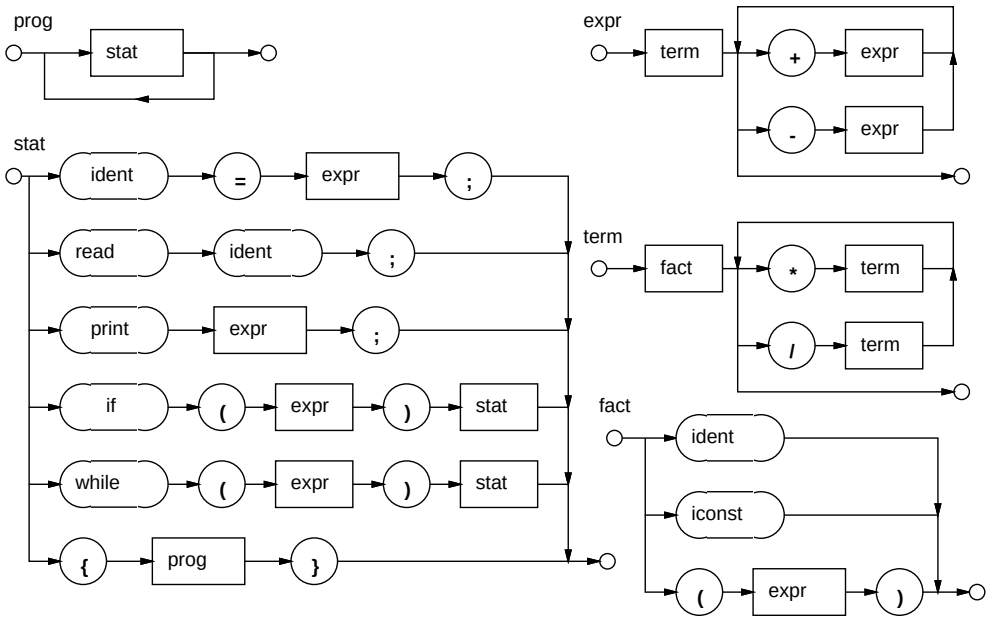

図 7.1: 構文図による小さな言語の文法

- 適用する生成規則に複数の選択肢がある場合は、入力に基づいて適切な選択肢を選んで上記をおこなう。

再帰下降という名称は、呼び出し関係の構造がちょうど構文木を上から下に向かってたどるのと同じになること、そして構文規則の再帰性に対応して手続が直接/間接に自分を呼び出す (再帰呼び出しを行なう) ことによります。

再帰下降解析は、その本体が手続きのコードとして実行されることから、前に取り上げた LL(1) 解析器と比較して次のような利点を持ちます。

- 選択肢を選ぶときに先読み記号を多くしたり (LL(k) 文法に対応することになります)、先読み記号以外の情報に基づいたりでき、扱える文法クラスが広くなる。
- 解析の作業と合わせて構文木の生成や記号表への登録などの作業を柔軟におこなえる。
- 規則の途中での分岐/合流やその一部分の反復など、拡張 BNF や構文図で表現されるような機能を直接的に実装できる。

7.3 小さい言語のツリーインタプリタ

本節以下では再帰下降解析の特徴を活かして、先に文法を示した言語の解析をおこないつつ、抽象構文木を構成するような再帰下降解析器を作ってみます。

抽象構文木としては、前に扱った `eval()` メソッドを持つノード群をそのまま利用するので、それを直ちに実行してみることができます。つまり小さな言語のツリーインタプリタができるわけです。以前のものをそのまま使うので、取り扱う値は整数のみで、論理値はなく、C 言語と同様に「0 でない整数値は真」として条件を扱います。

まず前回同様、各トークンに固有の番号を割り当てるために 1 つクラスを作ります (今回は非端記号は不要なので `Token` というクラス名にしました)。このクラスはコードは一切なく、定数を定義するだけです。

```
public class Token {
  public static final int EOF = -1, NL = 0, IDENT = 1, ICONST = 2,
    STR = 3, LPAR = 4, RPAR = 5, LBRA = 6, RBRA = 7, SEMI = 7,
```

```
    ASSIGN = 8, PLUS = 9, MINUS = 10, ASTER = 11, SLASH = 12,
    WHILE = 13, IF = 14, READ = 15, PRINT = 16;
}
```

次に、JFlex ソースを示します。各種の記号や予約語をそれぞれ認識するようにしただけで、根本は変わっていません。予約語よりも識別子の規則が後なのは、JFlex では複数のあてはまりがある場合は「上に」書いたものが優先されるためです。

```
%%
%class Lexer
%int
L = [A-Za-z_]
D = [0-9]
Ident = {L}({L}|{D})*
Iconst = [-+]?{D}+
String = \"(\\\"|.)*\"
Newline = \n
Blank = [ \t]+
%%

\=        { return Token.ASSIGN; }
\+        { return Token.PLUS; }
\-        { return Token.MINUS; }
\*        { return Token.ASTER; }
\/        { return Token.SLASH; }
\;        { return Token.SEMI; }
\(        { return Token.LPAR; }
\)        { return Token.RPAR; }
\{        { return Token.LBRA; }
\}        { return Token.RBRA; }
while     { return Token.WHILE; }
if        { return Token.IF; }
read      { return Token.READ; }
print     { return Token.PRINT; }
{Ident}   { return Token.IDENT; }
{Iconst}  { return Token.ICONST; }
{String}  { return Token.STR; }
{Newline} { return Token.NL; }
{Blank}   { /* ignore */ }
```

JFlex が生成した Lexer クラスを中で保持して外部とインタフェースするクラス Tokenizer については、前回と同じなので略します。

抽象構文木のクラス群も前にやったものと同一ですが、Tree というクラスの中に入れて、外部からは Tree.Node、Tree.Add などの名前で参照できるようにします (そのため、各クラスの冒頭に public 修飾子を追加します)。また、変数の値のための表もここで保持します。

```
import java.util.*;
public class Tree {
```

```
public static Map<String,Integer> vars = new TreeMap<String,Integer>();
public abstract static class Node {
  List<Node> child = new ArrayList<Node>();
  public void add(Node n) { child.add(n); }
  public abstract int eval();
}
public static class Lit extends Node {
  int val;
  public Lit(int v) { val = v; }
  public int eval() { return val; }
  public String toString() { return ""+val; }
}
public static class Var extends Node {
  String name;
  public Var(String n) { name = n; }
  public int eval() { return vars.get(name); }
  public String toString() { return name; }
}
public abstract static class BinOp extends Node {
  String op;
  public BinOp(String o, Node n1, Node n2) { op = o; add(n1); add(n2); }
  public String toString() { return "("+child.get(0)+op+child.get(1)+")"; }
}
public static class Add extends BinOp {
  public Add(Node n1, Node n2) { super("+", n1, n2); }
  public int eval() { return child.get(0).eval() + child.get(1).eval(); }
}
public static class Sub extends BinOp {
  public Sub(Node n1, Node n2) { super("-", n1, n2); }
  public int eval() { return child.get(0).eval() - child.get(1).eval(); }
}
public static class Mul extends BinOp {
  public Mul(Node n1, Node n2) { super("+", n1, n2); }
  public int eval() { return child.get(0).eval() * child.get(1).eval(); }
}
public static class Div extends BinOp {
  public Div(Node n1, Node n2) { super("/", n1, n2); }
  public int eval() { return child.get(0).eval() / child.get(1).eval(); }
}
public static class Mod extends BinOp {
  public Mod(Node n1, Node n2) { super("%", n1, n2); }
  public int eval() { return child.get(0).eval() % child.get(1).eval(); }
}
public static class Eq extends BinOp {
  public Eq(Node n1, Node n2) { super("==", n1, n2); }
  public int eval() { return child.get(0).eval()==child.get(1).eval()?1:0; }
```

```
  }
  public static class Ne extends BinOp {
    public Ne(Node n1, Node n2) { super("!=", n1, n2); }
    public int eval() { return child.get(0).eval()!=child.get(1).eval()?1:0; }
  }
  public static class Gt extends BinOp {
    public Gt(Node n1, Node n2) { super(">", n1, n2); }
    public int eval() { return child.get(0).eval()>child.get(1).eval()?1:0; }
  }
  public static class Ge extends BinOp {
    public Ge(Node n1, Node n2) { super(">=", n1, n2); }
    public int eval() { return child.get(0).eval()>=child.get(1).eval()?1:0; }
  }
  public static class Lt extends BinOp {
    public Lt(Node n1, Node n2) { super("<", n1, n2); }
    public int eval() { return child.get(0).eval()<child.get(1).eval()?1:0; }
  }
  public static class Le extends BinOp {
    public Le(Node n1, Node n2) { super("<=", n1, n2); }
    public int eval() { return child.get(0).eval()<=child.get(1).eval()?1:0; }
  }
  public static class And extends BinOp {
    public And(Node n1, Node n2) { super("&&", n1, n2); }
    public int eval() {
      int v = child.get(0).eval();
      if(v == 0) { return 0; } else { return child.get(1).eval(); }
    }
  }
  public static class Or extends BinOp {
    public Or(Node n1, Node n2) { super("||", n1, n2); }
    public int eval() {
      int v = child.get(0).eval();
      if(v != 0) { return v; } else { return child.get(1).eval(); }
    }
  }
  public abstract static class UniOp extends Node {
    String op;
    public UniOp(String o, Node n1) { op = o; add(n1); }
    public String toString() { return "("+op+child.get(0)+")"; }
  }
  public static class Not extends UniOp {
    public Not(Node n1) { super("!", n1); }
    public int eval() { return child.get(0).eval()==0?1:0; }
  }
  public static class Neg extends UniOp {
    public Neg(Node n1) { super("-", n1); }
```

```
    public int eval() { return -child.get(0).eval(); }
  }
  public static class Assign extends Node {
    Var v1; Node n1;
    public Assign(Var v, Node n) { v1 = v; n1 = n; }
    public int eval() {
      int v = n1.eval(); vars.put(v1.toString(), v); return v;
    }
    public String toString() { return v1+"="+n1; }
  }
  public static class Seq extends Node {
    public Seq(Node... a) { for(Node n:a) { child.add(n); } }
    public int eval() {
      int v = 0;
      for(Node n:child) { v = n.eval(); }
      return v;
    }
    public String toString() {
      String s = "{\n";
      for(Node n:child) { s += n.toString() + ";\n"; }
      return s + "}";
    }
  }
  public static class Read extends Node {
    Var v1;
    public Read(Var v) { v1 = v; }
    public int eval() {
      System.out.print(v1+"? ");
      Scanner sc = new Scanner(System.in);
      String str = sc.nextLine();
      int i = Integer.parseInt(str);
      vars.put(v1.toString(), i); return i;
    }
    public String toString() { return "read "+v1; }
  }
  public static class Print extends Node {
    public Print(Node n1) { child.add(n1); }
    public int eval() {
      int v = child.get(0).eval(); System.out.println(v); return v;
    }
    public String toString() { return "print "+child.get(0); }
  }
  public static class While extends Node {
    public While(Node n1, Node n2) { child.add(n1); child.add(n2); }
    public int eval() {
      int v = 0;
```

```
        while(child.get(0).eval() != 0) { v = child.get(1).eval(); }
        return v;
      }
      public String toString() {
        return "while("+child.get(0)+")"+child.get(1);
      }
    }
    public static class If1 extends Node {
      public If1(Node n1, Node n2) { child.add(n1); child.add(n2); }
      public int eval() {
        int v = 0;
        if(child.get(0).eval() != 0) { v = child.get(1).eval(); }
        return v;
      }
      public String toString() { return "if("+child.get(0)+")"+child.get(1); }
    }
  }
```

では本体です。Tokenizer のインスタンスは変数 tok に格納してクラス内どこからでもアクセスできるようにします。main では Tokenizer を生成して最初の手続き prog() を呼び出し、正しく結果が得られたらそれを表示して実行します。

```
 import java.util.*;
  import java.io.*;

  public class Sam71 {
    static Tokenizer tok;
    public static void main(String[] args) throws Exception {
      tok = new Tokenizer(args[0]);
      Tree.Node n = prog();
      if(n != null) { System.out.println(n.toString()); n.eval(); }
    }
```

ここから再帰下降解析の各手続きになります。前にやったときは認識器だったので OK かどうかを boolean で返していましたが、こんどは構文木を返したいので、OK なら対応する構文木オブジェクトを返し、OK でなければ nil を返す、という形にします。そのため、各手続きの返値は Tree.Node になります。

以下の各手続きを読むときは、図 7.1 の構文図または文法を見ながらにしてください。また、表 7.1 にこの文法の各非端記号の $First/Follow$ を挙げておきます。

まず prog() ですが、最初は空の Seq を作ります。そして、中では繰り返し stat() を呼び、返された値を Seq に追加していきます。それで、いつまで繰り返し呼べばいいでしょうか? 厳密には次のようにするべきです。

- 次の記号が $First(stat)$ である間 stat() を呼び、
- 上記でなくなって、かつ次の記号が $Follow(prog)$ のとき終了。

ただ、$First(stat)$ は沢山あって面倒ですし、$First(stat)$ のチェックは stat() の中でどのみちあるので、ここでは後者の条件だけをチェックしています。stat() を呼んで失敗したときはエラーなのでその旨出力して nil を返します。

表 7.1: 小さな言語の非端記号の $First/Follow$

端記号	$First$	Follow
$prog$	*Ident* `read print if while {`	`} $`
$stat$	*Ident* `read print if while {`	*Ident* `read print if while { } $`
$expr$	*Ident Iconst* `(`	*Ident* `read print if while { } $ ) ;`
$term$	*Ident Iconst* `(`	*Ident* `read print if while { } $ ) ;`
$fact$	*Ident Iconst* `(`	*Ident* `read print if while { } $ ) ;`

```
static Tree.Node prog() {
  Tree.Seq n1 = new Tree.Seq();
  while(!tok.chk(Token.RBRA) && !tok.chk(Token.EOF)) {
    Tree.Node n2 = stat();
    if(n2 != null) { n1.add(n2); continue; }
    System.err.printf("%d: error stat at: %s\n", tok.curLine(), tok.curStr());
    return null;
  }
  return n1;
}
```

`stat()` は文に対応し、文は代入文、read 文、print 文、while 文、if 文、ブロックのいずれかです。それぞれについて、その中身が正しく認識できたらそのノードを返し、どこかで失敗したら最後に来て `null` を返します。

```
static Tree.Node stat() {
  if(tok.chk(Token.IDENT)) {
    Tree.Var n1 = new Tree.Var(tok.curStr()); tok.fwd();
    boolean b1 = tok.chkfwd(Token.ASSIGN);
    Tree.Node n2 = expr();
    boolean b2 = tok.chkfwd(Token.SEMI);
    if(b1 && n2 != null) { return new Tree.Assign(n1, n2); }
  } else if(tok.chkfwd(Token.READ)) {
    Tree.Var n1 = null;
    if(tok.chk(Token.IDENT)) { n1 = new Tree.Var(tok.curStr()); tok.fwd(); }
    if(n1 != null && tok.chkfwd(Token.SEMI)) { return new Tree.Read(n1); }
  } else if(tok.chkfwd(Token.PRINT)) {
    Tree.Node n1 = expr();
    if(n1 != null && tok.chkfwd(Token.SEMI)) { return new Tree.Print(n1); }
  } else if(tok.chkfwd(Token.IF)) {
    boolean b1 = tok.chkfwd(Token.LPAR);
    Tree.Node n1 = expr();
    boolean b2 = tok.chkfwd(Token.RPAR);
    Tree.Node n2 = stat();
    if(b1 && n1 != null && b2 && n2 != null) { return new Tree.If1(n1,n2); }
  } else if(tok.chkfwd(Token.WHILE)) {
    boolean b1 = tok.chkfwd(Token.LPAR);
    Tree.Node n1 = expr();
```

```
      boolean b2 = tok.chkfwd(Token.RPAR);
      Tree.Node n2 = stat();
      if(b1 && n1!=null && b2 && n2!=null) { return new Tree.While(n1, n2); }
    } else if(tok.chkfwd(Token.LBRA)) {
      Tree.Node n1 = prog();
      if(tok.chkfwd(Token.RBRA)) { return n1; }
    }
    System.err.printf("%d: error stat at: %s\n", tok.curLine(), tok.curStr());
    return null;
  }
```

expr() はまず term() を呼び、次に $Follow(expr)$ がくるまで繰り返し、加算/減算のノードを作ります。木構造でははじめに出て来た部分式ほど深い位置になることに注意。

```
  static Tree.Node expr() {
    Tree.Node n = term();
    while(n != null && !tok.chk(Token.RPAR) && !tok.chk(Token.SEMI)) {
      if(tok.chkfwd(Token.PLUS)) {
        Tree.Node n1 = term(); n = (n1 == null) ? null : new Tree.Add(n, n1);
      } else if(tok.chkfwd(Token.MINUS)) {
        Tree.Node n1 = term(); n = (n1 == null) ? null : new Tree.Sub(n, n1);
      } else {
        n = null;
      }
    }
    if(n != null) { return n; }
    System.err.printf("%d: error expr at: %s\n",tok.curLine(),tok.curStr());
    return null;
  }
```

term() は expr() と同様ですが、$Follow(term)$ の方が要素が多いのでそこが違っています。

```
  static Tree.Node term() {
    Tree.Node n = fact();
    while(n != null && !tok.chk(Token.RPAR) && !tok.chk(Token.SEMI) &&
                       !tok.chk(Token.PLUS) && !tok.chk(Token.MINUS)) {
      if(tok.chkfwd(Token.ASTER)) {
        Tree.Node n1 = term(); n = (n1 == null) ? null : new Tree.Mul(n, n1);
      } else if(tok.chkfwd(Token.SLASH)) {
        Tree.Node n1 = term(); n = (n1 == null) ? null : new Tree.Div(n, n1);
      } else {
        n = null;
      }
    }
    if(n != null) { return n; }
    System.err.printf("%d: error term at: %s\n",tok.curLine(),tok.curStr());
    return null;
  }
```

最後に fact() は因子で、変数、整数定数、かっこで囲まれた式のいずれかになります。

```
  static Tree.Node fact() {
    if(tok.chk(Token.IDENT)) {
      Tree.Node n = new Tree.Var(tok.curStr()); tok.fwd(); return n;
    } else if(tok.chk(Token.ICONST)) {
      Tree.Node n = new Tree.Lit(Integer.parseInt(tok.curStr()));
      tok.fwd(); return n;
    } else if(tok.chkfwd(Token.LPAR)) {
      Tree.Node n = expr();
      if(tok.chkfwd(Token.RPAR)) { return n; }
      System.err.printf("%d: no closing ')'\n", tok.curLine());
      return null;
    }
    System.err.printf("%d: error term at: %s\n", tok.curLine(), tok.curStr());
    return null;
  }
}
// Tokenizer をここに
```

では、動かすプログラムを見てみましょう。整数を入力すると、その値から 1 まで値を減らしながら順に打ち出しますが、ただし 5 だけは打ち出しません。

```
% cat test.min
read x;
while(x) {
  if(x - 5) { print x; }
  x = x - 1;
}
%
```

実行例は次の通り。

```
% java Sam71 test.min
{
read x;
while(x){
if((x-5)){
print x;
};
x=(x-1);
};
}
x? 7
7
6
4
3
```

```
2
1
%
```

演習 7-1 例題をそのまま動かし、いくつか簡単なプログラムを実行してみよ。できたら、言語に次のような変更を行なってみよ。

a. 今の版では read は「read 文」だが、計算の途中で「read」という項が現れたらそこで入力が行なわれるという形に変更する。
b. do-while 文のような「末尾で条件を調べる」ループ文を追加する。
c. if 文に else 部がつけられるようにする。もちろん else 部があってもなくてもよいようにすること。
d. if 文を Ruby 等のように「if...elsif...elsif...end」の形のものに変更する。
e. 今の版では代入は「代入文」であるが、C 言語のように「代入演算子」にする。
f. その他、自分の好きな構文の変更をおこなう。

演習 7-2 例題では計算式が普通に「加減算より乗除算が強く、左から計算する」方式になっているが、次のような変更を行なってどんな感じか試してみよ。

a. 乗除算より加減算の方が強く結び付くように変更する。
b. べき乗演算子「**」を追加する。2 ** 3 ** 2 は「2 ** (3 ** 2)」と同じ、つまり右側を先に計算する。
c. 加減算、乗除算とも「右側から計算する」ように変更する。
d. 計算式を前置記法で記述する。「x + 1」でなく「+ x 1」のように書くことになる。
e. その他、自分の好きな計算式記法の変更をおこなう。

演習 7-3 自分の好きなミニ言語の構文を設計し、拡張 BNF や構文図で記述したのち、実装してみよ。

7.4 課題 7A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題＃ 7」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

Q1. 構文図の読み方や、それを元にして再帰下降解析器を組み立てるやり方が分かりましたか。
Q2. 再帰下降解析器とツリーインタプリタによる言語の実装を動かしてみてどのように思いましたか。
Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

＃８　構文解析 (3)

8.1　上向き解析

8.1.1　シフト還元解析器

前回までで、構文木を上の方 (出発記号の側) から組み立てて行く下向き解析を扱いました。今回は構文木を下の方から組み立てていく**上向き解析** (bottom-up parsing) を扱います。上向き解析を行う構文解析器は通常、**シフト還元解析器** (shift-reduce parser) の形を取ります。そこで具体的な解析方式の説明に先立ち、シフト還元解析器の枠組について説明しましょう。

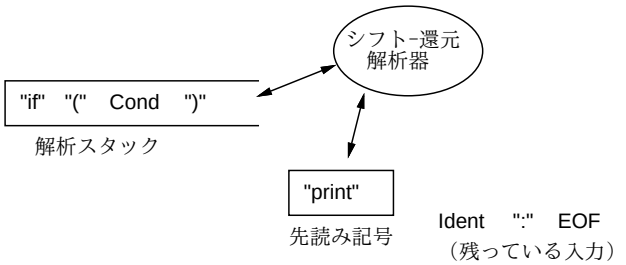

図 8.1: シフト還元解析器の枠組

シフト還元解析器の基本的な道具立てを図 8.1 に示します。解析には構文記号を積むスタック (解析スタック) 1 つと、先読み記号 1 個を使用します (より一般的には入力記号を n 個先まで見ることも考えられますが、動作の原理としては同じです)。そして、解析器の行う動作は必ず次の 2 つのうちいずれかです。

- シフト — 先読み記号をスタックに積み、入力を進める。
- 還元 — 生成規則 $A \to \alpha$ に対応し、α の長さ分スタックを取り降ろして代りに A を積む。

ただしこれでは終りがないので、文法で出発記号 S を左辺に持つ生成規則が 1 つだけになるようにし、その規則番号を 1 番とします。そして還元時に規則番号が 1 番だったら解析を終了します。以下では例題として次の文法を使用しますが、これも上記の条件を満たしています。

$$
\begin{aligned}
&\mathit{prog} ::= \mathit{statlist} \\
&\mathit{statlist} ::= \mathit{statlist}\ \mathit{stat} \mid \epsilon \\
&\mathit{stat} ::= \mathit{ident}\ \texttt{=}\ \mathit{expr}\ \texttt{;} \mid \texttt{read}\ \mathit{ident}\ \texttt{;} \mid \texttt{print}\ \mathit{expr}\ \texttt{;} \\
&\quad \mid \texttt{if (}\ \mathit{cond}\ \texttt{)}\ \mathit{stat} \mid \texttt{while (}\ \mathit{cond}\ \texttt{)}\ \mathit{stat} \mid \texttt{\{}\ \mathit{statlist}\ \texttt{\}} \\
&\mathit{cond} ::= \mathit{expr}\ \texttt{<}\ \mathit{expr} \mid \mathit{expr}\ \texttt{>}\ \mathit{expr} \\
&\mathit{expr} ::= \mathit{ident} \mid \mathit{iconst}
\end{aligned}
$$

先の言語の最小限のプログラム「`read x; print x;`」をシフト還元解析器で解析する様子を図 8.2 に示します。これからわかるように、シフト還元解析器では入力は次々にスタックに移され、スタック上に生成規則の右辺と一致するものができると左辺の非端記号に置き換える、という形で解析が進んで行きます。

ここまでで「なるほど、スタック上に規則の右辺が現れたら還元、それ以外ではシフトをすればいいのか」と思われたかも知れませんが、それほど単純ではありません。上の例でも print 文の *ident* はスタックに積まれた直後に *expr* に還元されていますが、一方 read 文の *ident* はそのまま残されています。これはもちろん、read 文の規則の右辺が `read` *ident* `;` であるからですが、ともかく右辺が現れたらすぐ還元してよい、というものでないわけです。

このシフトと還元の選択を正しく行うところがシフト還元解析器の「きも」です。その手法としてここでは最も広く使われている **LR 解析器** (LR parser) を説明します。

LR の最初の L は LL と同様「left-to-right に 1 回入力を捜査するだけで解析を行う」という意味、2 文字目の R は「rightmost derivation の逆順の系列を生成する」という意味です。逆順なのは上向き、つまり出発記号から遠いところから構文木をつくっていくため出発記号が最後になるからで、逆順にする前で考えれば左側の導出が先に出てきます (入力を左から読むのでそれが普通です)。

実際、図 8.2 に現れる規則を下から順に並べて導出列を作ると、*prog* ⇒ *statlist* ⇒ *stat statlist* ⇒ *stat stat statlist* ⇒ *stat stat* ⇒ *stat* `print` *expr* `;` ⇒ *stat* `print` *ident* `;` ⇒ `read` *ident* `;` `print` *ident* `;` となり、確かに最右導出になっています。

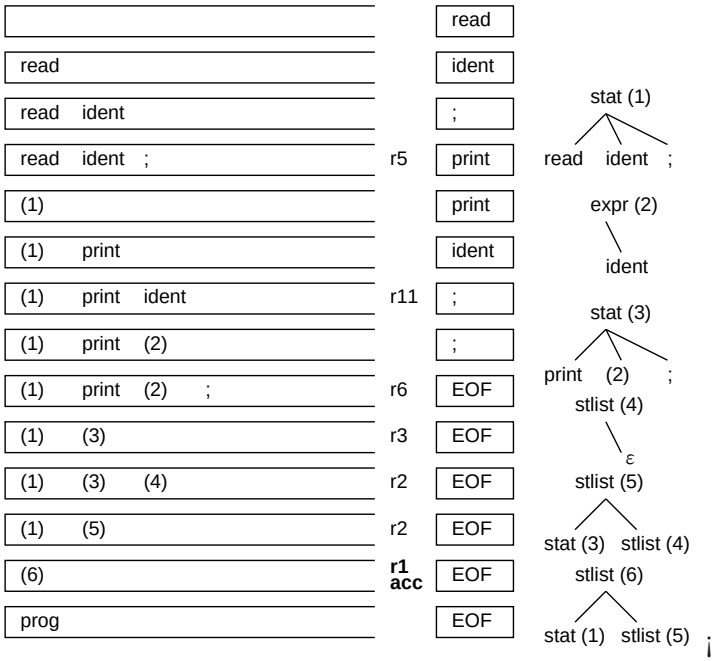

図 8.2: シフト還元解析器による構文解析

8.1.2 LR オートマトンと項

先の *ident* を還元するかどうかは、先に `read` をシフトしたか `print` をシフトしたかによって選択が違っていました。「置かれた状態によって同じ入力に対する動作が変化する」ことが問題ですが、この種の問題はオートマトンで表現するのが適しています。

ここでは **LR オートマトン**とよばれるものを使いますが、それは字句解析のときの有限オートマトンとはだいぶ違います。まず、LR オートマトンでは各状態が「構文規則の右辺のどの位置にいるか」に対応します。これを表すために、構文規則の右辺の任意の位置に • を記入して「現在いる位置」を表します。これを**項** (item) と呼びます (より厳密には LR(0) 項。0 は項の中に先読み記号の情報が含まれていないことを意味する)。

例えば、規則 *stat* → `if (` *cond* `)` *stat* からは 6 つの項ができます (項は `[]` で囲んで表す)。

`[` *stat* → • `if (` *cond* `)` *stat* `]`
`[` *stat* → `if` • `(` *cond* `)` *stat* `]`

[*stat* → `if` `(` • *cond* `)` *stat*]
[*stat* → `if` `(` *cond* • `)` *stat*]
[*stat* → `if` `(` *cond* `)` • *stat*]
[*stat* → `if` `(` *cond* `)` *stat* •]

8.1.3 LR(0) オートマトンの作成

それでは、上向き構文解析のためのオートマトンの構成方法について説明しましょう。字句解析の決定性オートマトンでは、1 つの状態は複数の正規表現のそれぞれ特定の場所に「並行して」対応していました。ここでも同様なことを考える必要があります。例えば、次の状態にいるものとします。

[*stat* → `if` `(` • *cond* `)` *stat*]

つまり *cond* の直前にいるわけですが、そこで次の生成規則を参照します。

cond → *expr* `<` *expr*
cond → *expr* `>` *expr*

ということは、上の状態は同時に次の 2 つの状態にも「並行して」存在していることになります。

[*cond* → • *expr* `<` *expr*]
[*cond* → • *expr* `>` *expr*]

さらに次の規則も参照します。

expr → *ident*
expr → *ionst*

ということは、これらの状態にいるということは次の状態にもいることになります。

[*expr* → • *ident*]
[*expr* → • *iconst*]

このように「並行していることになる」項の集合を (LR(0) 項集合の) **閉包** (closure) と呼びます。したがって、LR オートマトンの各状態は項の集合で、なおかつ閉包になっている必要があります。そして、状態間の遷移については、• の直後にある記号 (端記号・非端記号の双方) をラベルに持つ遷移により、• がその記号の後ろに移動した項 (の閉包) の状態に移ることになります。

先の文法に対応する LR(0) オートマトンを作成すると、図 8.3 のようになります。中身の項集合は表 8.1 に別に示しています。

この LR オートマトンを用いた解析過程を手でシミュレートしてみましょう (図 8.4)。プログラム例は再び「`read x; print x;`」です。

今度は道具立てとして、解析スタックと先読み記号に加えて「現在の状態」が必要なので、解析スタックには構文記号と LR オートマトンの状態を交互に載せます。まず状態 S1 から開始し、入力記号を `read`、*ident*、`;`、とシフトしながら S3、S24、S25 と遷移すると同時に、これらの記号と状態をそれぞれスタックに載せていきます。

S25 まで来ると「行き止り」になってしまいますが、この状態は生成規則 R6 の一番最後に • をもった項から成っているので、R6 に従って還元を行います。具体的には、R6 の右辺の記号数の倍 (状態と記号を対にして載せたから) だけスタックを取り降ろし (S1 だけが残る)、左辺 (つまり *stat*) を積みます。これは「S1 で *stat* が読めた」という状況に対応しています。そこで S1 から *stat* のラベルがついた遷移を行い、S8 へ来ます。

次は S8 で `print` なので S4 に行き、次が *ident* で S13 です。すると今度は先程と違い、ここで「行き止り」になって *ident* が *expr* に還元さて `S4` へ戻り、ここから改めて S22 へ進み、ここでようやく

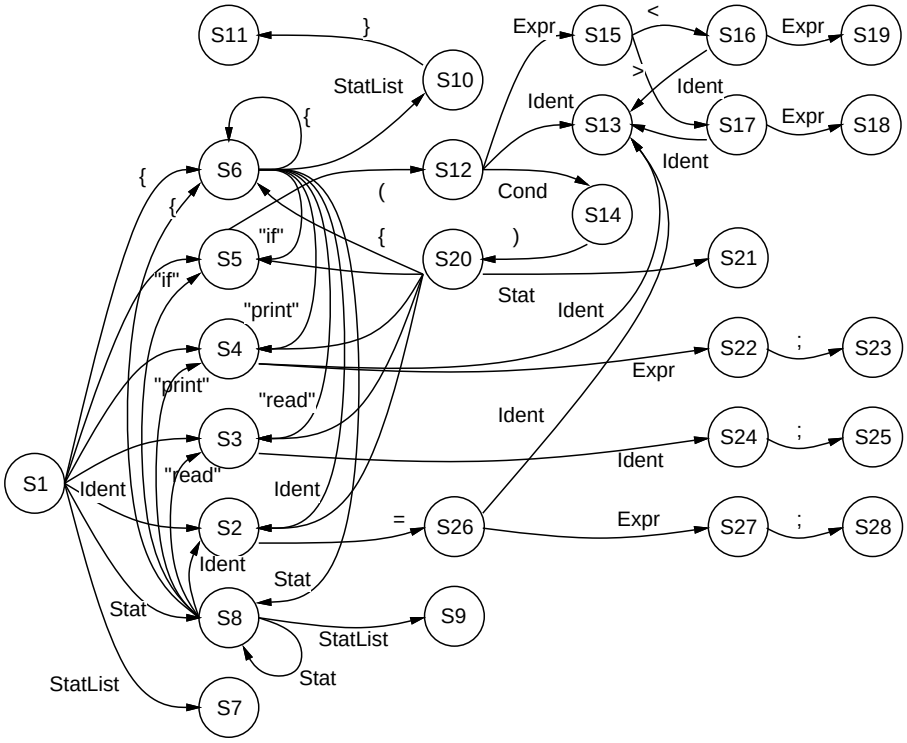

図 8.3: LR(0) オートマトン

S1	s3	read
S1 read S3	s24	ident
S1 read S3 ident S24	s25	;
S1 read S3 ident S24 ; S25	r5	print
S1 stat	s8	print
S1 stat S8	s4	print
S1 stat S8 print S4	s13	ident
S1 stat S8 print S4 ident S13	r11	;
S1 stat S8 print S4 expr	s22	;
S1 stat S8 print S4 expr S22	s23	;
S1 stat S8 print S4 expr S22 ; S23	r6	EOF
S1 stat S8 stat	s8	EOF
S1 stat S8 stat S8	r3	EOF
S1 stat S8 stat S8 stlist	s9	EOF
S1 stat S8 stat S8 stlist S9	r2	EOF
S1 stat S8 stlist	s9	EOF
S1 stat S8 stlist S9	r2	EOF
S1 stlist	s7	EOF
S1 stlist S7	**r1 acc**	EOF

図 8.4: LR 解析器による解析のようす

表 8.1: LR(0) オートマトンの各状態

S1: [*prog* → •*statlist*]
[*statlist* → •*statstatlist*]
[*statlist* → •]
[*stat* → •*ident* = *expr* ;]
[*stat* → • read *ident* ;]
[*stat* → • print *expr* ;]
[*stat* → • if (*cond*) *stat*]
[*stat* → • { *statlist* }]

S2: [*stat* → *ident*• = *expr* ;]

S3: [*stat* → read •*ident* ;]

S4: [*stat* → print •*expr* ;]
[*expr* → •*ident*]
[*expr* → •*iconst*]

S5: [*stat* → if • (*cond*) *stat*]

S6: [*statlist* → •*statstatlist*]
[*statlist* → •]
[*stat* → •*ident* = *expr* ;]
[*stat* → • read *ident* ;]
[*stat* → • print *expr* ;]
[*stat* → • if (*cond*) *stat*]
[*stat* → • { *statlist* }]
[*stat* → { •*statlist* }]

S7: [*prog* → *statlist*•]

S8: [*statlist* → •*statstatlist*]
[*statlist* → *stat* • *statlist*]
[*statlist* → •]
[*stat* → •*ident* = *expr* ;]
[*stat* → • read *ident* ;]
[*stat* → • print *expr* ;]
[*stat* → • if (*cond*) *stat*]
[*stat* → • { *statlist* }]

S9: [*statlist* → *statstatlist*•]

S10: [*stat* → { *statlist*• }]

S11: [*stat* → { *statlist* } •]

S12: [*stat* → if (•*cond*) *stat*]
[*cond* → •*expr* < *expr*]
[*cond* → •*expr* > *expr*]
[*expr* → •*ident*]
[*expr* → •*iconst*]

S13: [*expr* → *ident*•]

S14: [*stat* → if (*cond*•) *stat*]

S15: [*cond* → *expr*• < *expr*]
[*cond* → *expr*• > *expr*]

S16: [*cond* → *expr* < •*expr*]
[*expr* → •*ident*]
[*expr* → •*iconst*]

S17: [*cond* → *expr* > •*expr*]
[*expr* → •*ident*]
[*expr* → •*iconst*]

S18: [*cond* → *expr* > *expr*•]

S19: [*cond* → *expr* < *expr*•]

S20: [*stat* → •*ident* = *expr* ;]
[*stat* → • read *ident* ;]
[*stat* → • print *expr* ;]
[*stat* → • if (*cond*) *stat*]
[*stat* → if (*cond*) •*stat*]
[*stat* → • { *statlist* }]

S21: [*stat* → if (*cond*) *stat*•]

S22: [*stat* → print *expr*• ;]

S23: [*stat* → print *expr* ; •]

S24: [*stat* → read *ident*• ;]

S25: [*stat* → read *ident* ; •]

S26: [*stat* → *ident* = •*expr* ;]
[*expr* → •*ident*]
[*expr* → •*iconst*]

S27: [*stat* → *ident* = *expr*• ;]

S28: [*stat* → *ident* = *expr* ; •]

; がシフトされて S23 へ進み、print 文全体が還元されて S8 へ戻ります。このように、「シフトすべきか、還元すべきか」という選択はオートマトンの状態を通じて的確に指示されるわけです。

このあとはもう次が EOF なので、ϵ を *statlist* に還元し、それと前に認識した *stat* を併せて *statlist* に還元することを繰り返していき、S1 まで戻ったところで *statlist* により S7 へ行きます。次の動作は最初の規則 R1 による還元なので、これで終了となります。

8.1.4 SLR(1) 解析器

先の説明では「行き止り」になったとき、• が最後にある項 (「行き止り」だから必ずそういう規則がある) を用いて還元していました。

しかし、行き止りではないが • が最後に持つ項も含んでいる状態もあり得ますし、還元に使える項が複数ある可能性もあります。これらの場合の動作はオートマトンだけでは決まらず、別の情報を用いなければなりません。

1 つの考え方として、選択を $Follow$ 集合に基づいて行うという方針が挙げられます。つまり、ある状態に [$A \to \alpha\bullet$] なる項が含まれ、かつ次の先読み記号 t が $Follow(A)$ に含まれている時のみ、この項に対応する規則による還元を行うわけです。

入力が構文的に正しいならば A の後に来る端記号は $Follow(A)$ に含まれるはずなので、もしそうでないなら還元してはならないのは明らかです。この方針に基づく解析器を SLR(1) 解析器と呼びます (最初の S は Simlpe の意)。

	ident	=	;	(	)	{	}	<	>	eof	if	read	print	stlist	stat	cond	expr
S1	s2					s6	r3			r3	s5	s3	s4	7	8		
S2		s26															
S3	s24																
S4	s13																22
S5				s12													
S6	s2					s6	r3			r3	s5	s3	s4	9	8		
S7										accept							
S8	s2					s6	r3			r3	s5	s3	s4	9	8		
S9							r2			r2							
S10							s11										
S11	r8					r8	r8			r8	r8	r8	r8				
S12	s13															14	15
S13			r11		r11			r11	r11								
S14					s20												
S15								s16	s17								
S16	s13																18
S17	s13																18
S18					r10												
S19					r9												
S20	s2					s6					s5	s3	s4		21		
S21	r7					r7	r7			r7	r7	r7	r7				
S22			s23														
S23	r6					r6	r6			r6	r6	r6	r6				
S24			s25														
S25	r5					r5	r5			r5	r5	r5	r5				
S26	s13																27
S27			s28														
S28	r4					r4	r4			r4	r4	r4	r4				

動作表　　行先表

図 8.5: SLR(1) 構文解析表

LR 解析器の動作は一般に、状態と先読み記号の対から「シフトしてどの状態へ行く」か「どの規則で還元する」のいずれかを指示する**動作表** (action table) と、状態と非端記号の対から「還元した後どの状態に進むか」を指示する**行き先表** (goto table) の組からなる **LR 構文解析表** LR parsing table で表せます。先の例に対応する解析表を図 8.5 に示します。

先の図 8.4 の動作は、この解析表に従って解析した場合を反映したものです。どこに $Follow$ 集合

の情報が使われていたか分かりますか? それは、$statlist \to \epsilon$ による還元を行なうのは先読み記号がEOF か } のいずれかの時だけ、という部分です。この制約がないと、空列はいつでも $statlist$ に還元できてしまうので、それをどこで適用すべきか分からないことになります。

図 8.4 についてもう 1 つ補足です。先にはわかりやすさのためスタック上に構文記号と状態を交互に積むとしていましたが、実際には構文記号は必要としません。その理由は、オートマトンの状態には「現在どのような入力列を読んだところか」という情報が必要なだけ含まれているためです。このため、実際には解析スタック上には状態だけ積めば十分です。実際、動作表ではシフト時には「どの状態に進む」かは直接示されていますし、還元時には還元された記号が分かるのでそれを状態と照らし併せて次の状態が決められます。

8.1.5 正準 LR(1) 解析器と LALR(1) 解析器

SLR(1) 構文解析では還元の選択を $Follow$ 集合によっていますが、$Follow$ はもともと構文全体を通してどこかで後続記号になっているかを見るものなので、細かい文脈の区別には不向きです。

例えば次のような文法を考えてみます。

$$
\begin{aligned}
&prog \ ::= \ stat \\
&stat \ ::= \ left \ \texttt{=} \ right \mid right \\
&left \ ::= \ ident \mid \texttt{*} \ right \\
&right \ ::= \ left
\end{aligned}
$$

これは C のように「式は単独でも文になり得、また代入の左辺はポインタ参照でもよい」ような言語の文法を簡略化したものになっています。これから先のプログラムにより SLR(1) 構文解析器をつくろうとすると、$left$ を認識したあとに「=」が先読み記号になった場合、シフトすべきか $right \to left$ で還元すべきかが判別できません。これは、$right$ =の前に来ることが可能なため $Follow(right)$ に= が含まれているからです。しかし実際には代入記号の直前にいるのだから、$left$ を $right$ に還元するのは誤りです。

このような $Follow$ による「おおざっぱな判定」の弱点を克服するため、最初から状態の中に先読み記号も含めて計算を行うことを考えます。そこで、今度は項の中に次のようにして先読み記号を含めることにします。

$$
\begin{aligned}
&[\ prog \to \bullet \ stat \ \texttt{;} \ \texttt{EOF} \ ] \\
&[\ right \to left \ \bullet \ \texttt{;} \ \texttt{=} \ ]
\end{aligned}
$$

このような項を **LR(1) 項**と呼びます (1 というのは先読み記号の長さを示す)。そしてあとは、この項どうしの遷移により、これまで通りに解析表を作ります。これを正準 (cannonical)LR(1) 構文解析表、それに従って動作する解析器を正準 LR(1) 解析器と呼びます。

正準 LR(1) 文法は SLR(1) よりも広い文法クラスを含んでいます。実は、正準 LR(1) 文法は左から右へ後戻りなしで解析可能な最大の文法クラスとなっていることが知られています。

一方、正準 LR(1) 解析器の問題点はその状態数が非常に多くなるということです。それは文法における端記号の数が n として、項の数が SLR(1) の n 倍になっていることに起因します。このため、正準 LR(1) 解析器をそのままコンパイラに採用するのは実用的でないとされます。

実用のコンパイラでは、SLR(1) より受け入れられる文法の範囲が広く、しかも LR(1) ほどには状態数が多くない実用的な構文解析器として、LALR(1) 解析器が多く使われています。

これは、LR(1) でオートマトンを構成したあと、先読み記号部分 (; の後ろ) を取り除いて同じになる状態を互いに併合することで構成されます (併合をおこなうときに行き先となる状態が食い違って併合できないなどのことは起きないことが知られています)。

併合をおこなった後の状態数は LR(0) オートマトンと同じになるため、状態数の問題が解消され、なおかつ先読み記号の情報が使えるため、SLR(1) よりは広いクラスの文法が扱えます (ただし併合により正確さが減じるため、正準 LR(1) よりは狭くなる)。

なお、解析表の生成時だけとは言え、正準 LR(1) オートマトンをいったん作るのでは計算の手間が大きくなりますが LR(1) オートマトンを経ずに直接必要な状態のみを生成しながら合せて先読み記号の情報を計算する手順が知られています。

演習 8-1 例題のオートマトンや構文解析表で別のプログラムを与えたとき確かに解析ができることを確認してみなさい。解析のようすを記録すること。

演習 8-2 簡単な言語を BNF で記述し、その LR(0) オートマトンを構成してみなさい。SLR(1) 構文解析表までできるとなおよい。これらを使って簡単なプログラムを解析するようすを示すこと。

8.2 曖昧な文法の活用

ここまでは曖昧でない文法のみを想定して来ましたが、場合によっては曖昧な文法を許す方が人間、コンピュータ双方にとって有利なこともあります。その代表的な例として**ぶらさがり else**(dangling else) があります。多くの言語では `if` 文に対して次の構文定義を用いています。

stat → `if` *cond* `then` *stat* `else` *stat*
stat → `if` *cond* `then` *stat*

この場合、`if C1 then if C2then A else B` という文は次の 2 通りに解釈でき、曖昧です。

```
if C1 then ( if C2 then A else B )  -- (1)
if C1 then ( if C2 then A ) else B  -- (2)
```

そこで「ただし、おのおのの `else` はまだ対応する `else` をもたない最も近い `if` に対応させる」という規則を (自然言語で) 付すのが恒例です (これにより上の解釈 (2) は削除される)。

なお、この例の場合は代りに文法を次のように直しても曖昧さは解消できますが、明らかに繁雑になります。

balstat → `if` *cond* `then` *balstat* `else` *stat* | 各種の文
stat → `if` *cond* `then` *stat* | *balstat*

このように文法が曖昧なまま構文解析器を作ろうとすると、LR 構文解析の場合に**シフト還元衝突** (shift-reduce conflict) や**還元還元衝突** (reduce-reduce conflict) が発生します。つまり、構文解析表を構成しているときに、特定の位置の動作表に「シフト」「還元」の両方の動作が記入できることになったり、複数の規則による「還元」が記入できるようになることを言います。

たとえば、ぶらさがり else の場合、文法の曖昧さに対応して「`then` 部だけから成る `if` 文を全て読み終った時点とも、`else` 部まである `if` 文の `else` の直前とも取れる」状態が現れ、そこでシフト還元衝突となります。

そこで、このような状態では常にシフトする—つまり、後者の解釈を取る—ことにして解析表を作っていまうことができます。これによって構成される解析器は上述の「`else` を最も近い `if` に対応させる」解析動作を行います。LL(1) 解析器の場合でもこれと同様なことが行えます。

曖昧な文法が役に立つもう 1 つの代表例は式と演算子の構文です。「*expr* → *expr* `+` *expr*」という文法は「`1 + 2 + 3`」を「`(1 + 2) + 3`」とも「`1 + (2 + 3)`」とも解釈できます。

この曖昧さを除くには、これまでに見てきたように、式-項-因子のように演算子の強さごとに別の非端記号を使用すればよいわけですが、これも読みやすくはありません。ここでは+は左結合的なので、前者を選択し、還元を行うように解析表を作ればよいわけです。同様に、演算子間の順位や右結合なども、そのことを予め考慮して解析表を作ることで対応できるわけです。

曖昧な文法を使用することの利点は 1 つは、構文記号の数が少なく人間にとってわかりやすくなることですが、同時に解析器の側でも状態数が少なくてすみ、また余分な還元がなくなるので効率よく解析が行えるという利点があります。

8.3 構文解析器生成系

8.3.1 Cup とその構文定義

解析表とドライバが分かれた形の構文解析器では、コンパイラ作成者が自ら解析表の計算を行うことはほとんどなく、構文記述を入力すると解析表を作成してくれるツールを使用するのが普通です。これを**構文解析器生成系** (parser generator) と呼びます。

たとえば、Unix に古くから備わっている **Yacc** やそのフリーソフト版である Bison は LALR(1) 解析器を生成する生成系で、出力は C 言語のコードになっています。ここでは言語として Java を使っていて、JFlex との相性もよいことから、Cup と呼ばれる生成系を使ってみます (細かい書き方は Yacc と違っていますが、大筋は同じです)。

まず先頭の `import` は CUP のライブラリにアクセスするため必須のもので、そのままコードの先頭の入ります。そのあと、端記号および非端記号として使う名前を宣言します。ここでは端記号を大文字、非端記号を小文字にしています。

その次ですが、上にのべたように曖昧な文法を使うため、優先順位の低いものから演算子の端記号を列挙します。また、`nonassoc`、`left`、`right` により結合のしかた (非結合、左結合、右結合) を指定します。左結合は「$x \mathbin{☆} y \mathbin{☆} z \to (x \mathbin{☆} y) \mathbin{☆} z$」の意味で、非結合は「$x \mathbin{☆} y \mathbin{☆} z$」のようにつなげては書けないことを意味します (比較演算子は通常そうですね)。なお、最後の `UMINUS` という端記号は最も優先順位が高くなりますが、実際には現れません。これについてはすぐ後で述べます。

```
import java_cup.runtime.*;

terminal READ, PRINT, IF, WHILE, ASSIGN, SEMI, LPAR, RPAR, LBRA, RBRA;
terminal GT, LT, PLUS, MINUS, ASTER, SLASH, UMINUS;
terminal ICONST, IDENT;

non terminal prog, statlist, stat, expr;

precedence nonassoc LT, GT;
precedence left PLUS, MINUS;
precedence left ASTER, SLASH;
precedence left UMINUS;

prog ::= statlist
         ;
statlist ::= stat statlist
         |
         ;
stat ::= IDENT ASSIGN expr SEMI
       | READ IDENT SEMI
       | PRINT expr SEMI
       | IF LPAR expr RPAR stat
       | WHILE LPAR expr RPAR stat
       | LBRA statlist RBRA
       ;
expr ::= expr PLUS expr
       | expr MINUS expr
```

```
        | expr ASTER expr
        | expr SLASH expr
        | expr GT expr
        | expr LT expr
        | MINUS expr  %prec UMINUS
        | IDENT
        | ICONST
        | LPAR expr RPAR
        ;
```

さて文法ですが、見て分かるとおり普通の BNF で、ただし上述のように式のところは曖昧な文法を使っています (なので簡潔です)。「MINUS exp %prec UMINUS」というところが謎ですが、これは単項のマイナス演算子で、端記号としては MINUS が使われますが、ただし順位は他の演算よりも高い UMINUS の強さにします、という指定です。

演習 8-3 上の例題を Cup で処理してみなさい。うまくいったら、自分独自の簡単な言語を BNF で定義し、次に Cup で構文記述を書いて変換してみなさい。エラーがあれば直すこと。

8.3.2 JFlex と Cup の連携

上記の文法指定を cup コマンドで処理すると、parser.java と sym.java という 2 つのファイルが生成されます。前者はパーサのファイルですが、後者は Token.java のように端記号の番号を定義するためのもので、使い方も同様です。これを用いてこの言語用の JFlex ファイルを作成します。

まず冒頭で Cup と同じライブラリの import を行ないます。これは、字句解析器が返すものが Symbol オブジェクトである必要があるためです。「%cup」という指定は字句解析用のメソッド名を Cup 用の名前にするために必要です。次の 3 行は、Symbol オブジェクトを生成するためのオブジェクトを用意するものです。%eofval の 3 行は EOF になったときに EOF シンボルを返す指定です。

```
import java_cup.runtime.*;
%%
%class Lexer
%cup
%{
  SymbolFactory sf = new DefaultSymbolFactory();
%}
%eofval{
  return sf.newSymbol("EOF", sym.EOF);
%eofval}
L = [A-Za-z]
D = [0-9]
Ident = {L}({L}|{D})*
Iconst = [-+]?{D}+
Blank = [ \t\n]+
%%
\+        { return sf.newSymbol("PLUS", sym.PLUS); }
\-        { return sf.newSymbol("MINUS", sym.MINUS); }
\*        { return sf.newSymbol("ASTER", sym.ASTER); }
\/        { return sf.newSymbol("SLASH", sym.SLASH); }
```

```
\<        { return sf.newSymbol("LPAR", sym.LT); }
\>        { return sf.newSymbol("RPAR", sym.GT); }
\(        { return sf.newSymbol("LPAR", sym.LPAR); }
\)        { return sf.newSymbol("RPAR", sym.RPAR); }
\{        { return sf.newSymbol("LPAR", sym.LBRA); }
\}        { return sf.newSymbol("RPAR", sym.RBRA); }
\;        { return sf.newSymbol("SEMI", sym.SEMI); }
\=        { return sf.newSymbol("ASSIGN", sym.ASSIGN); }
if        { return sf.newSymbol("IF", sym.IF); }
while     { return sf.newSymbol("WHILE", sym.WHILE); }
read      { return sf.newSymbol("READ", sym.READ); }
print     { return sf.newSymbol("PRINT", sym.PRINT); }
{Ident}   { return sf.newSymbol("IDENT", sym.IDENT, yytext()); }
{Iconst}  { return sf.newSymbol("ICONST", sym.ICONST, yytext()); }
{Blank}   { /* ignore */ }
```

その後は基本的にこれまでの JFlex と同じですが、ただし返す値は上述のように Symbol オブジェクトを返すようにします。そのときのパラメタとして、1 番目は表示用の文字列、2 番目がトークンの値 (クラス sym で定義されているものを使います)、3 番目は *Ident* と *Const* についてのみ、値としてトークンの文字列を渡しています。これは後で使います。

main は次のようになります。ここでは何も動作を指定していないので、構文エラーがあればエラーが出るというだけです (つまり認識器です)。

```
import java.util.*;
import java.io.*;
import java_cup.runtime.*;

public class Sam81 {
  public static void main(String[] args) throws Exception {
    parser p1 = new parser(new Lexer(new FileReader(args[0])));
    p1.parse();
  }
}
```

なお ComplexSymbolFactory は CUP ランタイムが提供する記号用クラスで、これを標準で渡すことになっています。

実行のようすを一応のせましょう。

```
% cup sam81.cup
(Cup のメッセージ)
% jflex sam81.jflex
(JFlex のメッセージ)
% javac Sam81.java
% java Sam81 test.min  ←後述
%              ←出力なし: エラーがないことは分かる
```

8.3.3 属性とアクション

前節まででは、構文解析はしても何も動作がついていないので、出力が何もありませんでした。Cup で (Yacc や Bison もそうですが) 実際にコンパイラを作るにはどうするのでしょう。それには、次の

2 つの道具だてがあります。

(a) それぞれの構文記号 (端記号、非端記号) に**属性** (attribute) をつけられる。

(b) 構文規則に**動作** (action) を付随させられる。

(a) については、記号ごとにその記号が値を伴うことができる、ということです。これは実装としては、構文解析に使うスタックと並行してもう 1 つ**意味スタック** (semantic stack) というスタックを用意し、構文記号 (実際には対応する状態) を入れるのと同じ場所の意味スタック側に属性の値を格納します。Cup では属性値は Java のオブジェクト型である必要があります。

たとえば今回は、IDENT と ICONST は字句解析からの文字列 (既に `yytext()` の値を渡すように作ってありました) を属性値とし、端記号はすべて `Tree.Node` オブジェクトを属性値とすれば、木構造を組み立てることができます。

(b) については、それぞれの構文規則の右端に「{: … :}」という形で囲んだ Java コードを記述しておくと、そのコードはその構文規則が還元されるときに実行されます。上向き解析なので、下から順に木のノードを組み立てて行くのが自然な使い方です。

しかし、子の構文規則で組み立てたノードを親側から取り出すのにはどうすればいいでしょうか? そこで属性値を使います。Cup では構文規則の右辺に現れる端記号、非端記号には「:名前」という指定がつけられ、アクション中ではその名前の変数が、対応する記号の属性値を保持しています。そして、`RESULT` という特別な名前に代入したものが、その構文規則の左辺の記号の属性値となります。

では具体例を見てみましょう。文法は先の例と同じですが、属性とアクションが追加されています。まず属性の型として、IDENT と ICONST は `String`、非端記号は `Tree.Node` を指定しています。そして、規則の右辺で属性を受け渡す記号は `:x` とか `:y` など、受け渡す変数を指定しています。

prog の属性値は *statlist* の属性値そのままです。*statlist* は、空に対応するときは空の `Tree.Seq` を作り、そのあと 1 つ文が現れるごとにその文のノードを追加し、値としては `Tree.Seq` を受け渡して行きます。

```
import java_cup.runtime.*;

terminal READ, PRINT, IF, WHILE, ASSIGN, SEMI, LPAR, RPAR, LBRA, RBRA;
terminal GT, LT, PLUS, MINUS, ASTER, SLASH, UMINUS;
terminal String ICONST, IDENT;

non terminal Tree.Node prog, statlist, stat, expr;

precedence nonassoc LT, GT;
precedence left PLUS, MINUS;
precedence left ASTER, SLASH;
precedence left UMINUS;

prog ::= statlist:x            {: RESULT = x; :}
         ;
statlist ::= statlist:x stat:y {: RESULT = x; x.add(y); :}
             |                 {: RESULT = new Tree.Seq(); :}
             ;
stat ::= IDENT:x ASSIGN expr:y SEMI
         {: RESULT = new Tree.Assign(new Tree.Var(x), y); :}
       | READ IDENT:x SEMI  {: RESULT = new Tree.Read(new Tree.Var(x)); :}
```

```
        | PRINT expr:x SEMI  {: RESULT = new Tree.Print(x); :}
        | IF LPAR expr:x RPAR stat:y    {: RESULT = new Tree.If1(x, y); :}
        | WHILE LPAR expr:x RPAR stat:y {: RESULT = new Tree.While(x, y); :}
        | LBRA statlist:x RBRA {: RESULT = x; :}
        ;
expr ::= expr:x PLUS expr:y  {: RESULT = new Tree.Add(x, y); :}
        | expr:x MINUS expr:y {: RESULT = new Tree.Sub(x, y); :}
        | expr:x ASTER expr:y {: RESULT = new Tree.Mul(x, y); :}
        | expr:x SLASH expr:y {: RESULT = new Tree.Div(x, y); :}
        | expr:x GT expr:y    {: RESULT = new Tree.Gt(x, y); :}
        | expr:x LT expr:y    {: RESULT = new Tree.Lt(x, y); :}
        | MINUS expr:x        {: RESULT = new Tree.Neg(x); :} %prec UMINUS
        | IDENT:x  {: RESULT = new Tree.Var(x); :}
        | ICONST:x {: RESULT = new Tree.Lit(Integer.parseInt(x)); :}
        | LPAR expr:x RPAR {: RESULT = x; :}
        ;
```

それぞれの文や式はこれまでやってきたように、各種のノードを作って自分の値として返すだけです。そのとき子ノードの値は、構文規則の属性として受け渡されてくる値を使えばよいわけです。このようにして、パーサジェネレータを使うことでコンパクトに動作のついたパーサを構成し、抽象構文木を組み立てることができるわけです。

最後に `main()` ですが、組み立てた木構造は `parse()` が返す `Symbol` オブジェクトの変数 `value` に入っているので、それを `Tree.Node` 型にキャストして `eval()` することで実行できます。

```
import java.util.*;
import java.io.*;
import java_cup.runtime.*;

public class Sam82 {
  public static void main(String[] args) throws Exception {
    parser p1 = new parser(new Lexer(new FileReader(args[0])));
    Symbol s1 = p1.parse();
    ((Tree.Node)s1.value).eval();
  }
}
```

では、最大公約数のプログラムを実行してみましょう。

```
% cat test.min
read x; read y;
while(x - y) {
  if(x > y) { x = x - y; }
  if(x < y) { y = y - x; }
}
print x;
% java Sam82 test.min
x? 60
y? 18
```

```
6
%
```

演習 8-4 例題の文法の if 文に else 部をつけてみなさい。曖昧な文法と曖昧でない文法の両方でできるとなおよい (JFlex で ELSE を追加する必要。曖昧な文法は Cup のマニュアルをよく読む必要があるかも。低い順位で「right ELSE」を設定するとよいかも)。

演習 8-5 例題の文法の if 文や while 文を end のある形に直してみなさい。if 文については、if-elsif-elsif-else-end の構文もきちんとサポートすること。

演習 8-6 例題に自分の好きな言語機能 (制御構造でもそれ以外でも) を追加してみなさい。

演習 8-7 実行のされかたは例題と同じだが、書き方の見た目がまったく違う構文になっている言語を実装してみなさい。

8.4 課題 8A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 7」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 上向き解析の原理は分かりましたか。LR オートマトンの構成は無理としても、オートマトンができたらそれを使って動作するやり方が分かればよいです。

  Q2. cup を使ってパーサを構築するのはどうでしたか。

  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

＃9　意味解析と記号表

9.1　コンパイラコンパイラとは

コンパイラコンパイラ (compiler compiler) とは「コンパイラを生成する翻訳系」という意味です。しかし実際には、何かちょっと記述したらコンパイラが完全にできあがる、というのは現状では困難です。

これまで見て来たように、フロントエンド (字句解析、構文解析、そして意味解析の入口程度) であれば、コンパクトな記述から処理系を生成できるので、コンパイラコンパイラという用語はその範囲をおこなうものとして長く使われて来ました (図 9.1 左)。類似した用語として**パーサジェネレータ** (parser generator) がありますが、生成するのがパーサだけではないツールも多いので少し意味が違うかも知れません。

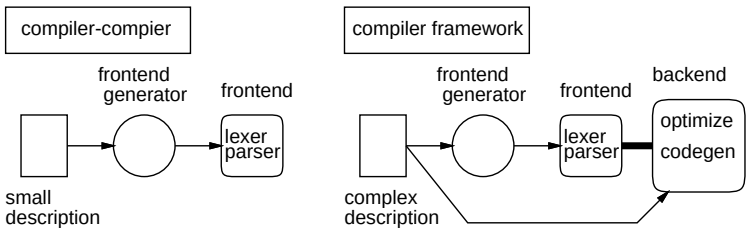

図 9.1: コンパイラコンパイラの概念

近年では、最適化の技術が進歩し、ある程度の記述 (実際にはかなり専門性が必要) を行なえば最適化を含んだバックエンドまで実現できるようなツールも現れています (LLVM などが有名)。この場合、コンパイラ作成者は共通の中間コードと (まだ無ければ) ターゲットマシン記述を生成し、ツールに含まれるバックエンドがそれを処理するので、コンパイラが生成されるというよりは、コンパイラを構築するツール群、という方が合っています。このようなものは、**コンパイラフレームワーク** (compiler framework) と呼ばれることもあります (図 9.1 右)。また、フロントエンドはどうやって作ってもいいということから、バックエンドしか関与しないコンパイラフレームワークもあります (LLVM はこれです)。

今回はフロントエンドを生成するという意味でのコンパイラコンパイラの例として、SableCC を取り上げます。ただしその前に、いくつかの前置き (予習) が必要なので、それから始めましょう。

9.2　いくつかの準備

9.2.1　Java の動的な型の扱い

Java では親クラスの型の変数には子クラスのインスタンスが入れられるということを以前に説明しました。ということは何が起きているかというと、メソッドを呼び出す時に実際に呼び出されるメソッドは実行時に変数にどのクラスのインスタンスが入っているかによって異なる、ということになります。たとえば、次の例を見てください。

```
class A {
```

```
  ...
  public void method1(...) { .... (A) .... }
}
class B extends A {
  ...
  public void method1(...) { .... (B) .... }
  public void method2(...) { .... }
}
```

ここで、A 型の変数 x があり、x.method1(...); という呼び出しが書かれていたとします。このとき、実際に動くコードは、x にクラス A のインスタンスが入っていれば (A) であり、クラス B のインスタンスが入っていれば (B) ということになります。また A の子クラスがほかにもあれば、そこで定義されているメソッドのコードかも知れません。このように、実行時にならないと対象が決まらないようなものを**動的束縛** (dynamic binding) と呼びます。

次に、クラス B に含まれているメソッド method2() について考えて見ましょう。これは、B 型の変数に入っている B のインスタンスに対してはもちろん普通に呼べます。しかし、クラス A にはこのメソッドは定義されていないので、A 型の変数に入れてしまうと呼ぶことができません (コンパイルエラーになる)。

```
A x = new B(...);    // B のインスタンスは A 型に代入 OK
x.method2(...);      // エラー: A 型には method2 がない
```

ではどうしたらいいかというと、x に入っているオブジェクトを B 型の変数に入れ直せばいいのです。ただし、このときは「親クラスの型から子クラスの型に」変換するので、そのままではだめで、キャストを書く必要があります。

```
B y = (B)x;          // x の中の値を B 型にキャスト
y.method2(...);      // OK
```

このキャストを**ダウンキャスト** (downcasting) と呼び、このときにインスタンスが実際にクラス B(またはその子クラス) のインスタンスであることをチェックします (それ以外であれば例外 ClassCastException が発生)。このように、Java ではオブジェクトに付随するクラスの情報を用いて実行時にチェックが行なわれます。

ところで、Java ではすべてのオブジェクトはクラス Object を継承している、という話は前にしました。ということは、すべてのクラスは Object のサブクラスであり、従って Object 型の変数に入れられます。これはコンテナなど「何でも入れられる」データ構造を作るのには便利なのですが、取り出して来た後「元の型に戻す」のにはやはりダウンキャストが必要です。

9.2.2　インタフェースとその扱い

Java の特徴的な機能として、インタフェースがあります。インタフェースとは「実装のない、メソッドの名前と型だけを規定したもの」だと言えます。たとえば次のコードを見てください。

```
interface Doable {
  public void doit(...);
}
class X implements Doable {
  ...
  public void doit(...) { .... (X) ... }
}
```

この場合、Doable 型の変数 d にクラス X のインスタンスを入れることができます (キャストなしに)。そして、d.doit(...):が呼ばれたときには、(X) のコードが動くわけです。

インタフェースの何がいいのでしょう? 継承と同じようなものでは? いいえ、継承は変数やメソッドを引き継いで来るので、必然的に似たような構造のクラス (あるクラスを拡張したクラス) を作ることにしか使えません。まったく無関係な 2 つのクラスでは駄目なのです。

しかしインタフェースは「このようなメソッドを持つ」ということだけが条件なので、まったく別のクラスに同じインタフェースを実装 (implements) させることができますし、1 つのクラスが多数のインタフェースを実装しても問題ないです (継承は 1 つの親クラスに限定)。

それでは、X クラスのインスタンスを Object 型の変数 z に入れてあるとき、これを Doable 型の変数 d に入れるには? それもやはりダウンキャストを使います (クラス階層の上下とはもはや無関係ですが)。これもやはり、実行時のクラス情報でチェックされます。

```
Doable d = (Doable)z;  // ダウンキャスト:実行時にチェック
```

9.2.3 Visitor パターンとその拡張

ここまでに使って来た抽象構文木の木構造を見て「すばらしい」「分かりやすい」と思いましたか? もしそうなら、何か見落としています。何が問題かというと、「ツリーを実行する」ためのメソッドが各クラスにバラバラに散らばっているという点です。たとえば、式の計算について見ると、加算、減算などの処理はすべて Add、Sub など各クラスのメソッド eval() として書かれています (図 9.2)。

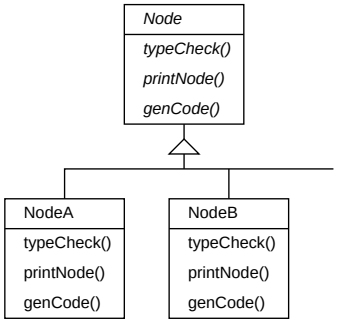

図 9.2: 素朴な再帰による木構造の処理

それで別にいいじゃないか、と思いますか? これには、次のような問題点があります。

- 「計算をする」というひとまとまりの処理が各クラスにバラバラに散らばってしまう。
- 「計算する」「型チェックする」「コード生成する」などの処理ごとにそのバラバラのクラスに全部手を入れてメソッドを追加しなければならない。
- それぞれのクラスでやる処理全体を通して使うデータの置き場所がない (子供メソッドの呼び出しごとにデータ構造を持ち回るのも繁雑)。

この問題を解消するために考案されたのが Visitor パターンです (図 9.3)。こんどは、各ノードは NodeVisitor オブジェクト v を引数として受け取る accept() というメソッドだけを持っていて、その中で「v.visit ノード種別 ()」というメソッドを呼び出すことで「自分の種別の処理」を呼び出します。

NodeVisitor はインタフェースであり、型検査、コード生成などの仕事ごとにそれを実装するさまざまな Visitor オブジェクトをこのインタフェースに従うものとして用意します。いずれかの Visitor オブジェクトを引数として渡して accept() を呼び出すと、ノードの種類ごとに visit ノード種別 () が呼ばれて来ますから、その中で子ノードをさらにたどることが必要なら accept(this) を呼び出せばつぎつぎにだどりが行えます。

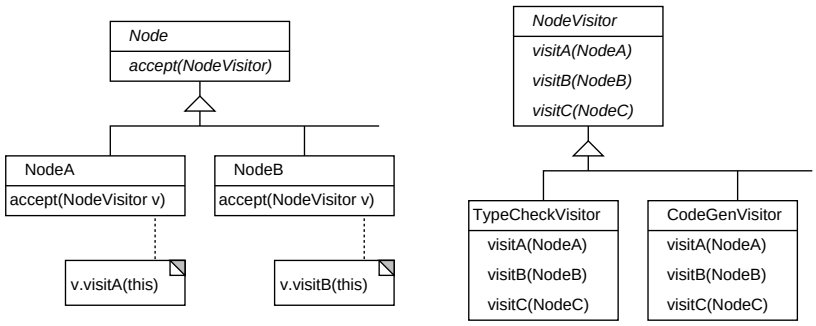

図 9.3: Visitor パターンへの変換

なお、これらの呼び出しはすべて 1 つの `Visitor` オブジェクトのメソッド呼び出しなので、そのオブジェクトのインスタンス変数が「ずっと保持しておくデータの置き場所」に使えます。

さて、ここまでがガンマ本 (デザパタ本) に載っている Visitor の話で、Visitor には元の形式の弱点を解消した代償として次の弱点が生じているという指摘があります。

> オブジェクト構造のクラスをしばしば変更する場合には、すべての Visitor のインタフェースを再定義する必要があり、潜在的にコストは高くつく。

確かにそれはその通りですが、だからといって元の形に戻るのも面白くありません。そこですぐ後で出て来る SableCC は、次のように Visitor パターンをさらに拡張しています。

- 切り替えを行うインタフェースを非常に抽象的なものとして (何も操作なしで) 定義する。

```
interface Visitor { }
```

- 各ノード側はこれを受け取る `accept()` を定義するため、次のインタフェースを実装する。

```
interface Visitable { void apply(Visitor v); }
```

- 実際の Visitor は上のインタフェースを拡張して個々の種別を定義。

```
interface BaseVisitor extends Visitor {
  void visitA(A obj);
  void visitB(B obj);
  ...
}
```

- 別の種別も増やすことができる。

```
interface ExcendedVisitor extends Visitor {
  void visitX(X obj);
  void visitY(Y obj);
  ...
}
```

- 個々のノードの `accept()` は自分が属しているインタフェースにキャストして自分用のメソッドを呼び出すことができる。

```
class NodeA extends implements Visitable {
  ...
  void accept(Visitor v) { ((BaseVisitor)v).visitA(this); }
}
class NodeX extends implements Visitable {
  ...
  void accept(Visitor v) { ((ExtendedVisitor)v).visitX(this); }
}
```

- 実際に使うときには、これらをすべてまぜたインタフェースを作る。

  ```
  interface AllVisitor extends BaseVisitor, ExtendedVisitor { }
  ```

  そして、Visitor クラスはこのインタフェースを実装した上で、すべての種別に対応する `visit種別()` を定義すればよい。

なんだかごちゃごちゃで頭がこんがらがりますが、このようにすれば個々のノードは `Visitor` インタフェースの変更の影響を受けませんし (そもそも `Visitor` インタフェースはからっぽで変更されません)、Visitor オブジェクトも自分が取り扱う種類のインタフェース群だけ対処すればいいわけです (といっても `AllVisitor` を実装することが結局多そうな気がしますが…)。

9.3 SableCC コンパイラコンパイラ

9.3.1 SableCC とは…

SableCC は、米国マギル大学の院生 (当時)Étienne Gagnon が 1997 年に修論で作ったコンパイラコンパイラです。元が修論作品ではありますが、よくできていて使いやすいので一定のファンがいて、今でもメンテナンスされ続けています。どういうところが「よくできている」かというと…

ここまでに字句解析器 JFlex とパーサジェネレータ cup を見て来ました。これらはうまく連携できますが、やはり別々のソフトなので 2 つのファイルを取り扱う必要があります。あと、パーサ側から「何行目の何文字目にエラー」みたいなメッセージを出そうとすると、文字を読むのは字句解析器の担当なので、連携が複雑になりがちです。

また、cup では還元が起きるときの動作を構文記述と一緒に書いていましたが、ということはその動作が起きるタイミングは固定で 1 回だけということになります。ですから通常は、木構造を組み立ててそこで処理をするということになります。また、文法記述と動作が混ざって書かれているので読みづらいという弱点があります。

これに対し、SableCC は次のようにできています。

- 構文解析と字句解析が一体で 1 つの記述ファイルから生成し、連携も内部で自動的になされる。
- 構文記述にアクションを書く必要はなく、構文木は自動で生成される。
- 生成された構文木は前述の Visitor パターンに合った形になっていて、コンパイラ作成者はさまざまな Visitor クラスを作ることで繰り返しソース情報を処理できる (マルチパス)。

以下ではまず記述ファイルの書き方、次に構文木のたどりの 2 段階に分けて SableCC の使い方と機能を見て行きます。

9.3.2　記述ファイルの書き方

SableCC では生成ソースを 1 つのパッケージに入れるので、たとえばパッケージが sam91 であれば、すべてのソースはディレクトリ sam91/以下に置かれます。自分が書くソースもそうすることになります。作成するファイルは冒頭に「package sam91;」を入れ、置き場所は「sam91/なんとか.java」で、そのファイル名でコンパイルします。そして実行開始するときは「java sam91. なんとか」になります (クラス名を指定するので)。

SableCC の記述ファイルは上述のように字句解析と構文解析を一緒に書くので、いくつかのセクションに分かれています。簡単な具体例で見てみましょう。

```
Package sam91;

Helpers
  digit = ['0'..'9'] ;
  lcase = ['a'..'z'] ;
  ucase = ['A'..'Z'] ;
  letter = lcase | ucase ;

Tokens
  iconst = ('+'|'-'|) digit+ ;
  blank = (' '|13|10)+ ;
  if = 'if' ;
  read = 'read' ;
  print = 'print' ;
  semi = ';' ;
  assign = '=' ;
  lt = '<' ;
  gt = '>' ;
  lbra = '{' ;
  rbra = '}' ;
  lpar = '(' ;
  rpar = ')' ;
  ident = letter (letter|digit)* ;

Ignored Tokens
  blank;

Productions
  prog = {stlist} stlist
       ;
  stlist = {stat} stlist stat
         | {empty}
         ;
  stat = {assign} ident assign expr semi
       | {read}   read ident semi
       | {print}  print expr semi
       | {if}     if lpar cond rpar stat
```

```
        | {block}  lbra stlist rbra
        ;
  cond = {gt} [left]:expr gt [right]:expr
        | {lt} [left]:expr lt [right]:expr
        ;
  expr = {ident}  ident
        | {iconst} iconst
        ;
```

規則のあちこちに [名前]: というものが書かれていますが、これは SableCC では BNF の右辺に同じ記号名が 2 回現れるときはそれを区別する名前つける必要があるためです。

main() は次のように、PushbackReader クラスを渡すようになっています。ここでは解析するだけなので、構文チェッカができます。

```
package sam91;
import sam91.parser.*;
import sam91.lexer.*;
import sam91.node.*;
import java.io.*;
import java.util.*;

public class Sam91 {
  public static void main(String[] args) throws Exception {
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0])), 1024)));
    Start tree = p.parse();
  }
}
```

では実行を前回の例で試してみます。

```
% sablecc sam91.grammer
...
% javac sam91/Sam91.java
...
% cat test.min
read x; read y;
if(x > y) { z = x; x = y; y = z; }
print x; print y;
% java sam91.Sam91 test.min
%
```

「何も言わない」ということは構文解析が成功したという意味になります。

演習 9-1 上の例をそのまま動かしてみよ。動いたら、いく通りかのプログラムを打ち込み、構文エラーは構文エラーとして検出されるこも確認しなさい。その後、次のような変更を行ってみなさい。

a. while 文を追加してみなさい。

b. 算術式の計算ができるようにしてみなさい。
c. if 文に else 部がつけられるようにしてみなさい。
d. その他好きな拡張をしてみなさい。

なお、SableCC には cup のような「あいまい文法」の機能がないため、文法を曖昧さの無い BNF で書く必要があり、記述がやや面倒です。

9.3.3　構文木のたどり

いよいよ、構文木をたどって動作を行う部分を見てみましょう。既に述べてきたように、SableCC では構文木はパーサによって自動的に作られ、それをたどるのには拡張された Visitor パターンが使われています。

Visitor の土台となるクラスとして `DepthFirstAdapter` というクラスが生成されていて、ここには文法に現れるすべてのノードの visit メソッドが予め「何もしない」形で用意されているので、このクラスを継承して必要なところだけをオーバライドしていくことで必要な処理を記述します。

オーバライドするためには、メソッド名が分かっている必要がありますね。SableCC では、構文規則に対応してメソッド名が次のように決められます。まず構文規則が次のものだとします。

```
xxx : {yyy} aa bb cc
    | {zzz} [left]:aa bb [right]:aa
    ;
```

まずノードクラスについて説明しましょう。1 つのノードは 1 つの規則に対応しているので、上の場合は 2 つのノードオブジェクトが定義されています。それらのクラス名はそれぞれ、「`AYyyXxx`」と「`AZzzXxx`」になります (先頭が A、次が {} 内に書かれた規則の名前を Capitalize したもの、次が左辺の記号名を Capitalize したもの)。

そして、これらのノードクラスはそれぞれ、右辺の各要素を取り出すメソッドとして `getAa()`、`getBb()`、`getCc()` の 3 つ、および `getLeft()`、`getBb()`、`getRight()` の 3 つを持ちます (このため、同じ名前の記号に対しては区別のための別の名前を指定する必要があったわけです)。これらが返すのはそれぞれのノードオブジェクトですが、そのノードが端記号の場合はその端記号に対応していた文字列が `getText()` によって取得できます。

いよいよ Visitor のためのメソッドですが、これは `DepthFirstAdapter` において各ノードごとに 3 つのメソッドが用意されています。たとえば上の例で 1 番目のノードでは次のようになります。

```
public void inAYyyXxx(AYyyXxx node) { ... }
public void outAYyyXxx(AYyyXxx node) { ... }
public void caseAYyyXxx(AYyyXxx node) { ... }
```

構文木は名前通り深さ優先順でたどられますが、最初にそのノードに到達するときに `in` メソッドが呼ばれ、最後にそのノードから出ていくときに `out` メソッドが呼ばれ、その間で子ノードに対する `apply()` が呼ばれます。多くのノードはこの「最初」「最後」だけで用が足りるのですが、「途中」でも処理が必要な場合は `case` メソッドをオーバライドして使用します。ただし `case` メソッドをオーバライドした場合、その中で自分で子ノードの `apply()` を呼ばなければ、子ノードはたどられません (したがって、たどりたくない場合にも `case` をオーバライドします)。つまり、次のようにするのが標準です。

```
@Override  ←名前を間違えやすいので必ずこのアノテーションをつける
public void caseAYyyXxx(AYyyXxx node) {
  // 最初に到達したときの処理…
  node.getAa().apply(this);
```

```
    // aa と bb の間の処理…
    node.getBb().apply(this);
    // bb と cc の間の処理…
    node.getCc().apply(this);
    // 終って出て行くときの処理
  }
```

なお、上の case... をオーバライドしなかった場合は、子ノードの処理を呼ぶ前と後にそれぞれ次のメソッドを呼ぶという定義が有効です。

```
  public void inAYyyXxx(AYyyXxx node) { ... }
  public void outAYyyXxx(AYyyXxx node) { ... }
```

これらはそれぞれ、木をたどって来てノードに入って来た時と、ノードから出て行く時に呼ばれるメソッドということになります。入口だけ、出口だけで処理が必要なら、こちらをオーバライドするのが簡単で。これもオーバライドしなかったら? そのときはこれらの「何も動作がない」版が動きます。

さて、これでオーバライドのしかたは分かりましたが、あと 1 つ説明すべきことが残っています。構文木をたどりながら処理をするとき、ノード間でデータを受け渡していくのが普通ですが、メソッドの形は上のように決まっているので、受け渡すデータのためのパラメタを追加することができません。この問題に対処するため、SableCC では `DepthFirstAdapter` において、データの受け渡し用に、次のメソッドを用意しています。

```
  void setIn(Node node, Object x);
  Object getIn(Node node);
  void setOut(Node node, Object x);
  Object getIn(Node node);
```

ここで In 側は木の上側から葉に向かってデータを流すのに使い、Out 側は葉から上側に向かってデータを戻すのに使うという想定です。格納されるのは `Object` 値なので、適宜キャストが必要です (古い Java のコンテナのスタイル)。

では具体的に見てみましょう。文法記述は次の通り。

```
  Package sam92;

  Helpers
    digit = ['0'..'9'] ;
    lcase = ['a'..'z'] ;
    ucase = ['A'..'Z'] ;
    letter = lcase | ucase ;

  Tokens
    iconst = ('+'|'-'|) digit+ ;
    blank = (' '|13|10)+ ;
    if = 'if' ;
    while = 'while' ;
    read = 'read' ;
    print = 'print' ;
    semi = ';' ;
```

```
  assign = '=' ;
  add = '+' ;
  sub = '-' ;
  lt = '<' ;
  gt = '>' ;
  lbra = '{' ;
  rbra = '}' ;
  lpar = '(' ;
  rpar = ')' ;
  ident = letter (letter|digit)* ;

Ignored Tokens
  blank;

Productions
  prog = {stlist} stlist
       ;
  stlist = {stat} stlist stat
         | {empty}
         ;
  stat = {assign} ident assign expr semi
       | {read}   read ident semi
       | {print}  print expr semi
       | {if}     if lpar expr rpar stat
       | {while}  while lpar expr rpar stat
       | {block}  lbra stlist rbra
       ;
  expr = {gt}  [left]:nexp gt [right]:nexp
       | {lt}  [left]:nexp lt [right]:nexp
       | {one} nexp
       ;
  nexp = {add}  nexp add term
       | {sub}  nexp sub term
       | {term} term
       ;
  term = {ident}  ident
       | {iconst} iconst
       ;
```

main() では解析木を取得して Executor で実行します。

```
package sam92;
import sam92.parser.*;
import sam92.lexer.*;
import sam92.node.*;
import java.io.*;
import java.util.*;
```

```
public class Sam92 {
  public static void main(String[] args) throws Exception {
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0])), 1024)));
    Start tree = p.parse();
    tree.apply(new Executor());
  }
}
```

Executor は次のようになります。

```
package sam92;
import sam92.analysis.*;
import sam92.node.*;
import java.io.*;
import java.util.*;

class Executor extends DepthFirstAdapter {
  Scanner sc = new Scanner(System.in);
  PrintStream pr = System.out;
  HashMap<String,Integer> vars = new HashMap<String,Integer>();
  @Override
  public void outAIdentTerm(AIdentTerm node) {
    String s = node.getIdent().getText().intern();
    if(!vars.containsKey(s)) vars.put(s, new Integer(0));
    setOut(node, vars.get(s));
  }
  @Override
  public void outAIconstTerm(AIconstTerm node) {
    setOut(node, new Integer(node.getIconst().getText()));
  }
  @Override
  public void outATermNexp(ATermNexp node) {
    setOut(node, getOut(node.getTerm()));
  }
  @Override
  public void outAAddNexp(AAddNexp node) {
    int v = (Integer)getOut(node.getNexp()) + (Integer)getOut(node.getTerm());
    setOut(node, new Integer(v));
  }
  @Override
  public void outASubNexp(ASubNexp node) {
    int v = (Integer)getOut(node.getNexp()) - (Integer)getOut(node.getTerm());
    setOut(node, new Integer(v));
  }
  @Override
```

```
public void outAOneExpr(AOneExpr node) {
  setOut(node, getOut(node.getNexp()));
}
@Override
public void outAGtExpr(AGtExpr node) {
  if((Integer)getOut(node.getLeft()) > (Integer)getOut(node.getRight())) {
    setOut(node, new Integer(1));
  } else {
    setOut(node, new Integer(0));
  }
}
@Override
public void outALtExpr(ALtExpr node) {
  if((Integer)getOut(node.getLeft()) < (Integer)getOut(node.getRight())) {
    setOut(node, new Integer(1));
  } else {
    setOut(node, new Integer(0));
  }
}
@Override
public void outAAssignStat(AAssignStat node) {
  String s = node.getIdent().getText().intern();
  vars.put(s, (Integer)getOut(node.getExpr()));
}
@Override
public void outAReadStat(AReadStat node) {
  String s = node.getIdent().getText().intern();
  pr.print(s + "> ");
  vars.put(s, sc.nextInt()); sc.nextLine();
}
@Override
public void outAPrintStat(APrintStat node) {
  pr.println(getOut(node.getExpr()).toString());
}
@Override
public void caseAIfStat(AIfStat node) {
  node.getExpr().apply(this);
  if((Integer)getOut(node.getExpr()) != 0) { node.getStat().apply(this); }
}
@Override
public void caseAWhileStat(AWhileStat node) {
  while(true) {
    node.getExpr().apply(this);
    if((Integer)getOut(node.getExpr()) == 0) { return; }
    node.getStat().apply(this);
  }
```

```
    }
  }
```

基本的に、式の中では、それぞれの式の値を out メソッドで計算して setOut() でノードの出力値として保持します。中間のノードは子ノードの値を取って来て必要に応じて計算し、自ノードの値とします。read 文や print 文はその場でそれぞれの動作をします。if 文や while 文は、条件部をまず実行し、その結果に応じて本体部の実行を制御するわけです。

実行例は次の通り (フィボナッチ数を指定最大値まで表示します)。

```
% cat test2.min
read max;
x0 = 0;
x1 = 1;
while(x1 < max) {
  x2 = x0 + x1;
  x0 = x1;
  x1 = x2;
  print x0;
}
% java sem1.Compiler sem1.txt
max> 10
1
1
2
3
5
8
%
```

演習 9-2 上の例をそのまま動かしなさい。動いたら、いく通りかのプログラムを打ち込み、思った通りに動作することを確認しなさい。その後、言語に次のような変更を行ってみなさい。

a. 乗除算も追加してみなさい。
b. do-while 文のような「下端で条件を調べるループ」を追加してみなさい。
c. if 文に else 部がつけられるようにしてみなさい。

演習 9-3 この方式で自分の好きな言語を設計して実装してみなさい。

9.3.4 言語の見た目とその効果

上で見たのは比較的「普通の」構文を持つおもちゃ言語でしたが、本質は同じでも、もっと見た目を変えることができます。そこで、日本語を使ったヘンな言語を定義してみます。

```
Package sam93;
Helpers
  digit = ['0'..'9'];
  lcase = ['a'..'z'];
  ucase = ['A'..'Z'];
  letter = lcase | ucase;
```

```
Tokens
  woireru = ’を入れる’;
  niyomikomu = ’に読み込む’;      ←「に」が先だとまずいので注意
  ni = ’に’;                          (先に書かれているものが優先なので)
  wouchidasu = ’を打ち出す’;
  naraba = ’ならば’;
  wojikkou = ’を実行’;
  noaida = ’の間’;
  gt = ’>’;
  lt = ’<’;
  add = ’+’;
  sub = ’-’;
  number = digit+;
  ident = letter (letter|digit)*;
  blank = (’ ’ | 10 | 13 )+;

Ignored Tokens
  blank;

Productions
  prog = {stlist} stlist
       ;
  stlist = {empty}
       | {stat} stlist stat
       ;
  stat = {assign} ident ni expr woireru
       | {read}   ident niyomikomu
       | {print}  expr wouchidasu
       | {if}     expr naraba stlist wojikkou
       | {while}  expr noaida stlist wojikkou
       ;
  expr = {term} term
       | {gt} [left]:term gt [right]:term
       | {lt} [left]:term lt [right]:term
       ;
  term = {fact} fact
       | {add}  term add fact
       | {sub}  term sub fact
       ;
  fact = {ident} ident
       | {number} number
       ;
```

この言語のプログラム例を示します。トークンの間は空けてもいい (というか空けた方が読みやすい) のですが、日本語ぽくわざとくっつけて書いてみました。

```
n に読み込む
```

x に 1 を入れる
n>0 の間 x に x+x を入れる n に n-1 を入れるを実行
x を打ち出す

演習 9-4 次のような計算をするプログラムをこの言語で書け。

a. 2 つの数を読み込んで合計を打ち出す。
b. 2 つの数を読み込んで大きい順に打ち出す (2 数は等しくないものとしてよい)
c. 入力した値 n を超えないフィボナッチ数を順番に打ち出す。

コンパイラドライバは先と変わらないので、ツリーインタプリタのみを示します。

```
package sam93;
import sam93.analysis.*;
import sam93.node.*;
import java.io.*;
import java.util.*;

class Executor extends DepthFirstAdapter {
  Scanner sc = new Scanner(System.in);
  PrintStream pr = System.out;
  HashMap<String,Integer> vars = new HashMap<String,Integer>();
  @Override
  public void outAAssignStat(AAssignStat node) {
    vars.put(node.getIdent().getText(), (Integer)getOut(node.getExpr()));
  }
  @Override
  public void outAReadStat(AReadStat node) {
    String s = node.getIdent().getText();
    pr.print(s + "> "); vars.put(s, sc.nextInt()); sc.nextLine();
  }
  @Override
  public void outAPrintStat(APrintStat node) {
    pr.println(getOut(node.getExpr()).toString());
  }
  @Override
  public void caseAIfStat(AIfStat node) {
    node.getExpr().apply(this);
    if((Integer)getOut(node.getExpr()) != 0) { node.getStlist().apply(this); }
  }
  @Override
  public void caseAWhileStat(AWhileStat node) {
    while(true) {
      node.getExpr().apply(this);
      if((Integer)getOut(node.getExpr()) == 0) { break; }
      node.getStlist().apply(this);
    }
```

```
    }
    @Override
    public void outATermExpr(ATermExpr node) {
      setOut(node, getOut(node.getTerm()));
    }
    @Override
    public void outAGtExpr(AGtExpr node) {
      if((Integer)getOut(node.getLeft()) > (Integer)getOut(node.getRight())) {
        setOut(node, new Integer(1));
      } else {
        setOut(node, new Integer(0));
      }
    }
    @Override
    public void outALtExpr(ALtExpr node) {
      if((Integer)getOut(node.getLeft()) < (Integer)getOut(node.getRight())) {
        setOut(node, new Integer(1));
      } else {
        setOut(node, new Integer(0));
      }
    }
    @Override
    public void outAFactTerm(AFactTerm node) {
      setOut(node, getOut(node.getFact()));
    }
    @Override
    public void outAAddTerm(AAddTerm node) {
      int v = (Integer)getOut(node.getTerm()) + (Integer)getOut(node.getFact());
      setOut(node, new Integer(v));
    }
    @Override
    public void outASubTerm(ASubTerm node) {
      int v = (Integer)getOut(node.getTerm()) - (Integer)getOut(node.getFact());
      setOut(node, new Integer(v));
    }
    @Override
    public void outAIdentFact(AIdentFact node) {
      setOut(node, vars.get(node.getIdent().getText()));
    }
    @Override
    public void outANumberFact(ANumberFact node) {
      setOut(node, Integer.parseInt(node.getNumber().getText()));
    }
  }
```

基本的に、式の中では、それぞれの式の値を out メソッドで計算して setOut() でノードの出力値として保持します。中間のノードは子ノードの値を取って来て必要に応じて計算し、自ノードの値とし

ます。read 文や print 文はその場でそれぞれの動作をします。if 文や while 文は、条件部をまず実行し、その結果に応じて本体部の実行を制御するわけです。

では、さっきのプログラムを実行してみます。

```
% cat test.min
n に読み込む
x に 1 を入れる
n>0 の間 x に x+x を入れる n に n-1 を入れるを実行
x を打ち出す
% java sam93.Sam93 test.min
n> 8
256
%
```

やっていることは前の言語と同じですが、見た目が違うとそれだけで、それを読んだり書いたりする人にとっての「違い」もはっきり現れます。皆様はこれを見てどう思いましたか?

演習 9-5 上の例をそのまま動かしなさい。動いたら、演習 9-4 で書いたプログラムを動かしてみなさい。その後、言語に次のような変更を行ってみなさい。

a. if 文に else 部が書けるように直して、その上で「2 数の最大」を動かしてみなさい。
b. do-while 文のような「下端で条件を調べるループ」を追加してみなさい。
c. その他、好きな変更を行ってみなさい。

9.4 課題 9A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 9」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

Q1. SableCC のようなコンパイラコンパイラについて、使ってみてどのように思いましたか。
Q2. Visitor パターンについて知っていましたか。使ってみてどのように思いましたか。
Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

＃10　コンパイラフレームワーク

10.1　意味解析の位置付け

プログラミング言語の定義は構文と意味の組み合わせによりなされるという説明をしてきました。それらのうち構文については、ここまでに文脈自由文法による形式的 (formal) な定義や、それを認識する解析器を扱って来ました。これはつまり、理論的な背景があり、それに基づいた厳密なやり方で扱う方法が確立している (やり方が分かっている)、ということです。

しかし逆にいえば、プログラミング言語という複雑な対象のうちで、理論的な扱いやそれに対応するツールがまだ確立していない「その他」の部分が「意味」として残されているとも取れます。そしてそれを引き受けるのが**意味解析** (semantic analysis) です (図 10.1)。

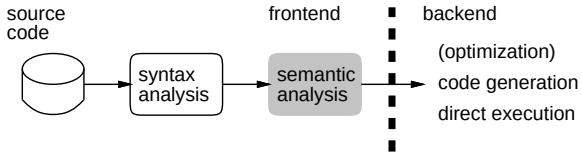

図 10.1: 意味解析の位置づけ

ただし、意味解析でも扱わない部分があります。それは「それぞれのコード記述がどのように動作するか」という部分です。言語の動作の実現は、インタプリタ (解釈系) であれば実行部分、コンパイラ (翻訳系) であれば出力されるコードによって定められますから、意味解析の中では扱わなくても済みます (検査や最適化などを目的に扱うこともあります)。

意味解析は解析部 (フロントエンド) の一部ですから、プログラムコードに記述された情報を抽出して解析することが仕事であり、それら情報のうちで、構文では扱えない部分を取り扱います。具体的には、次のような事項です。

- ソースコード中で使われている名前の情報を抽出する
- それぞれの名前に付随すべき情報を整理し整合性を検査
- 目的コード生成で使用される情報を準備

具体的にはどういうことでしょうか。たとえば C 言語をはじめ多くの強い型の言語 (型検査を行なう言語) では、変数は宣言してからでなければ使用できず、また使用するときに宣言と整合していない使い型をするとエラーになります。この「使用できない」「エラーになる」というのはすべてコンパイラが実現すべきことで、これが意味解析の仕事に含まれます。

前回までに扱ってきた「小さな言語」では意味解析がなく、構文解析が終わったら直ちに実行に入っていました。それは、「変数は宣言不要で、使ったら勝手に (その時点で) 用意される」「変数はすべて整数型」という設計になっていたからです。このような設計で、なおかつインタプリタであれば、実行を開始して最初に変数を読み書きした時にその場所を (実際にはハッシュ表の 1 エントリとしてですが) 用意すれば十分でした。

しかし多くの言語ではもっと「きちんと」変数を扱う必要がありますし、変数以外に手続きの名前や型の名前もあります。手続きを呼ぶときには、その引数の個数や型が手続き定義と整合している必

要もあります。そのような言語では、意味解析の仕事はそれなりに複雑です。[1]

以下ではまずソースコードに現れる名前の情報を収集する手段である記号表について、続いて強い型の言語でとくに重要になる型検査について取り上げていきます。

10.2 記号表

10.2.1 記号表に登録される情報

上で挙げたように、意味解析の仕事の多くは、変数名、手続き名などの「名前」に関することです。このため意味解析では、これらの名前の情報を記録した「表」を保持し、そこに情報を登録したり、その情報を参照してチェックするなどの形で多くの仕事をします。この表のことを伝統的に**記号表** (symbol table) と呼んでいます。

記号表に登録される名前の情報としては、次のようなものがあります (もちろん言語によって違ってきます)。

- 名前の文字列 — 名前の文字列は生成コードに現れることもありますが、生成コードでは番地などの数値に変換されて消えるため、メッセージの表示にしか名前を使わない処理系もあります。
- 名前の種別 — 変数名、定数名、手続き名、モジュール名、型名、レコードのフィールド名、構造体のタグなど、言語により多様な種別の名前があります。
- 名前のスコープ — 変数でいえば、同じ変数でもローカル変数、グローバル変数、手続きのパラメタ、モジュール内の変数など、様々なスコープのものがあります。他の名前でも (変数とまったく同じではないにせよ) 同様です。
- 型情報 — 強い型の言語では、変数、定数、手続き (返値)、フィールドなど多くの名前に**型** (type) が付随しています。この情報は型検査のために重要ですし、コード生成でも使われます。
- その他の属性 — 定数の値、変数の配置された位置、フィールドの位置など、名前ごとに固有のさまざまな属性があります。

これらの各種情報はどこから来るのでしょうか。名前の文字列は字句解析から来ます。また、名前の種別は「手続き定義の冒頭に現れた名前なら手続き名」「変数定義の箇所に現れた名前なら変数名」のように、多くは構文と対応しています。

そして、その後のスコープ、型情報、その他の属性については、記号表を解した処理、つまり意味解析の処理を通じて決まっていくことになります。具体例についてはこの後で取り上げます。

10.2.2 ブロックスコープとその実現

記号表も表なので、線形探索による表、2 分探索木、ハッシュ表などさまざまな技法で実現できますが、それとは別にプログラミング言語を扱う上での考慮点が多くあります。ここではその中でも特徴的な、**ブロック型スコープ**の実現について取り上げます。

ブロック型スコープとは、多くの言語に採用されている設計であり、1 つの名前をブロックの内側と外側で別の用途に使うことを許す設計です。次の C 言語コードの例を見てください。

```
char x[100];
int test(int x) {
  ... /* (1) */
  while(...) {
    ... /* (2) */
    double x;
```

[1] そして、上記の 3 番目に「準備」とありましたが、変数であれば「どの場所にする」ということを決めて置かないと目的コードが生成できないので、それも担当します。

```
    ... /* (3) */
    if(...) { float x; ... /* (4) */ }
    else    { int x[20]; ... /* (5) */ }
    ... /* (6) */
  }
  ... /* (7) */
}
```

ここで、外側のグローバル変数 x は文字の配列ですが、関数 test では同名のパラメタ x が定義されています。このため、この関数内では外側の x は使えません。(1) の箇所で x という名前を使うとパラメタの x になります。

さて、while 文の本体はブロックになっていますが、その途中に double 型の変数の宣言があります (今日の C 言語では宣言はブロックの途中でも構いません)。ただし、C 言語ではその変数のスコープは宣言から後ろなので、(2) の箇所では x はパラメタの方を意味し、(3) の箇所では double の方を意味します。なお、言語によっては、宣言が後に出て来ても、ブロックの中で宣言があればそれを参照することになっているものもあり、その場合は (2) も double の方になります。

そして、if 文の中ではさらに別の x が宣言されていて、(4) ではこちらを指しますが、else の枝では配列の x が宣言されていてこちらを指します。もしこの宣言が無ければ、(5) では double の x を参照します。内側のスコープが閉じた (6)、(7) ではそれぞれ double のもの、パラメタのものというふうに参照先が戻ります。

このような意味に対応する記号表はどのように実現すればいいでしょうか。それは実は難しくありません。線形探索で表を実現するとして、図 10.2 のように考えます。

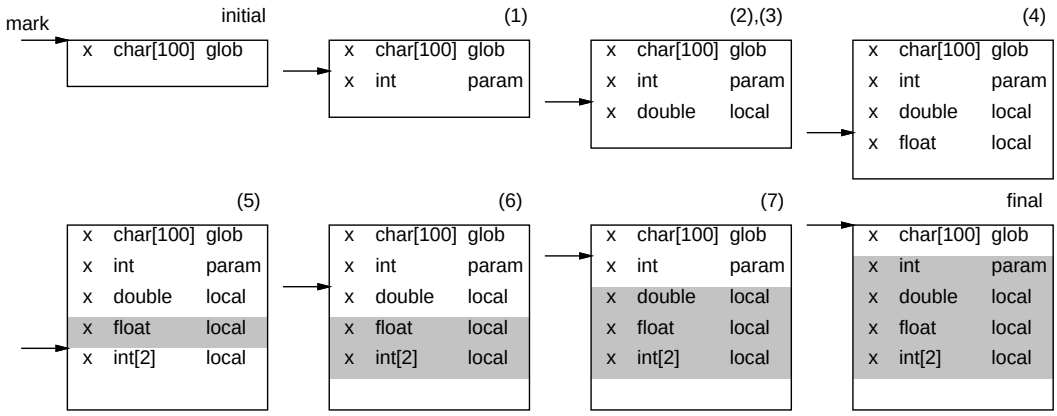

図 10.2: 線形探索で実現した記号表

表にはマーク (矢印) がつけてあり、そこが現在の最内側スコープのはじまりを意味します。最初はグローバルなスコープで、マークは先頭にあります。次に関数 test に入ると、その関数のトップレベルのスコープに入り、そこでパラメタを登録します。このほか、トップレベルで (つまり (1) や (7) で) 変数定義があったら、表の末尾に追加します。

検索については、表の末尾から上に向かって逆向きに検索します。こうすることによって、一番新しい (近くの) 宣言が見つかるので、(1) や (7) で x を参照するとパラメタが見つかります。マークは何のためにあるかというと、新しい変数定義を追加するときに、マークよりも下に同名の変数が定義されていたら、2 重定義としてエラーにするためです。マークより上は別の (外の) スコープなので、同じ名前でも内側スコープの別の変数になり、追加できます。

while のブロックに入った (2) や (3) のところでは、マークがパラメタより下に来ていて、同名の変数が定義できます。もちろん (2) のところではまだ double x に遭遇していないので、(3) まできたと

きにこの項目は入ります。同様にして if のブロックでもマークが進みます。

さて、次はスコープが閉じるところです。if の then 側のスコープが終わり、else で新しいスコープが始まると、マークは進みますが、同時にスコープが閉じたところでそのスコープ内の変数 (マークがあったところから現在位置まで) は探せない状態にします (図では網掛けしています)。閉じたスコープ内のローカル変数は以後参照できないわけですから。

同様に、else が閉じたところ、while が閉じたところ、関数が閉じたところでもそれぞれのスコープの変数を探せなくします。同時に、マークは「そのスコープに入る前の」位置に戻します。スコープは入れ子構造になっていますから、マークを保存しておいて戻すのにはスタック (Last-In, First-Out の記憶領域) を使うことができます。

10.2.3 例題: ブロックスコープの記号表

では、前節で述べた方式の記号表を作ってみます。まず最初に、エラーメッセージを出力するクラスを簡単に用意しておきます。メッセージを出してエラーをカウントし、後でカウントした数を取得できるようにしています。これは、エラーがあったら実行に進まないなどの動作を実現するためです。

```
public class Log {
  public static int err = 0;
  public static void pError(String s) { System.out.println(s); ++err; }
  public static int getError() { return err; }
}
```

では次に記号表のクラス Symtab を見てみます。後の例題で同じクラスをそのまま使う都合上、今回の例題で不要な機能も少しくっついています。まず、型を扱うため、ITYPE、ATYPE という定数を定義しました。未定義のものを表すのは UNDEF です。

次に表の項目は Symtab.Ent という内側クラスのインスタンスで表します。その部分を先に見て頂きたいですが、1 つの項目の情報としては name(名前文字列)、alive(網掛けの場合は false)、type(上記の型情報)、pos(表の何番目の項目かを示す) があります。

記号表のインスタンス変数ですが、Ent の並びが表本体、あと前節で述べたマーク用の変数 mark と、マークを対比回復するためのスタック stk があります。

以下メソッドですが、duplCheck() は指定した変数を定義すると 2 重定義かどうかのチェックで、マークより下に網掛けでない同名の名前が定義済みなら 2 重定義です。

addDef() は変数を定義しますが、まず 2 重定義ならエラーを表示した後、適当なエラー項目を返します。そうでない場合は与えられた名前で指定された型の項目を作って表に追加し、その項目を返します。

lookup は名前を指定して項目を探しますが、前に述べた理由で末尾から上に向かって探します。見つかればその項目を返し、見つからない場合は適当なエラー項目を返します。

enterScope() はスコープに入るところですが、マークをスタックに保存して表の末尾位置を新たなマーク位置とします。exitScope() はその逆ですが、ただしマークを戻す前にこれまでのマーク位置から末尾までを網掛けにします。

あとは getGsize() が表のサイズを返し、show() が表の内容を一覧表示するものです。

```
import java.util.*;

public class Symtab {
  public static final int ITYPE = 1, ATYPE = 2, UNDEF = -1;
  List<Ent> tbl = new ArrayList<Ent>();
  Stack<Integer> stk = new Stack<Integer>();
  int mark;
```

```
  private boolean duplCheck(String n) {
    for(int i = mark; i < tbl.size(); ++i) {
      Ent e = tbl.get(i);
      if(e.alive && e.name.equals(n)) { return true; }
    }
    return false;
  }
  public Ent addDef(String n, int t) {
    if(duplCheck(n)) {
      Log.pError("dublicate: "+n); return new Ent(n, UNDEF, 0);
    }
    Ent e = new Ent(n, t, tbl.size()); tbl.add(e); return e;
  }
  public void enterScope() { stk.push(mark); mark = tbl.size(); }
  public void exitScope() {
    for(int i = mark; i < tbl.size(); ++i) { tbl.get(i).alive = false; }
    mark = stk.pop();
  }
  public Ent lookup(String n) {
    for(int i = tbl.size()-1; i >= 0; --i) {
      Ent e = tbl.get(i);
      if(e.alive && e.name.equals(n)) { return e; }
    }
    return new Ent(n, UNDEF, 0);
  }
  public int getGsize() { return tbl.size(); }
  public void show() { for(Ent e: tbl) { System.out.println(" " + e); } }
  public static class Ent {
    public String name;
    public boolean alive = true;
    public int type, pos = 0;
    public Ent(String n, int t, int p) { name = n; type = t; pos = p; }
    public String toString() {
      return String.format("%s%s[%d %d]", name, alive?"_":"!", type, pos);
    }
  }
}
```

ではこの記号表に値を出し入れするドライバを使って試してみます。記号表を作った後、次のコマンドを入力できるようになっています。

- `quit` — 終わる
- `def` 名前 — 指定した名前の変数を定義
- `ref` 名前 — 指定した名前の変数を参照
- `enter` — 新しいスコープに入る

- exit — 最内側のスコープを閉じる
- show — 表全体を表示 [2]

```
import java.util.*;

public class SamA1 {
  public static void main(String[] args) throws Exception {
    Symtab st = new Symtab();
    Scanner sc = new Scanner(System.in);
    while(true) {
      System.out.print("> ");
      String[] cmd = sc.nextLine().split(" +");
      if(cmd[0].equals("quit")) {
        System.exit(0);
      } else if(cmd[0].equals("def")) {
        System.out.println(" +> " + st.addDef(cmd[1], Symtab.ITYPE));
      } else if(cmd[0].equals("ref")) {
        System.out.println(" -> " + st.lookup(cmd[1]));
      } else if(cmd[0].equals("enter")) {
        st.enterScope();
      } else if(cmd[0].equals("exit")) {
        st.exitScope();
      } else {
        st.show();
      }
    }
  }
}
```

では、動かしているところを見てみましょう。

```
% java SamA1
> def i       ←i を定義。
 +> i_[1 0]   ←i のオフセットは 0
> def i       ←もういちど定義。
dublicate: i  ←重複定義エラー
 +> i_[-1 0]
> def j       ←j を定義
 +> j_[1 1]   ←j のオフセットは 1
> enter       ←新しいスコープに入る
> def i       ←i を定義
 +> i_[1 2]   ←今度は新しい i が作れてオフセットは 2
> ref i       ←ここで参照すると
 -> i_[1 2]   ←オフセット 2 の i が取れる
> exit        ←スコープを出ると
> ref i       ←再度 i を参照すると
```

[2]簡単のため上記のコマンドどれかでなければ show として扱います。

```
  -> i_[1 0]    ←外側のオフセット 0 の i が取れる
> show          ←記号表表示
  i_[1 0]
  j_[1 1]
  i![1 2]       ←内側の i は網掛けになっている
> quit
```

演習 10-1 上の例をそのまま動かしてみなさい。さらに、より複雑な C 言語のプログラム例でやるとどうなるかも、確認してみなさい。

10.2.4 より高度な記号表の構成

前節で示した記号表は極めてシンプルなものでした。とくに、線形探索で「網掛け」のところは単に飛ばすという実装だと、大きなプログラムになったときに速度が問題になる可能性があります。

1 つのすぐ考え付く改良としては、記号表本体とは別に、現在見える名前のリストを (前節の例題のように) 並べた表を作り、その上で探索を行なう方法があります (図 10.3)。

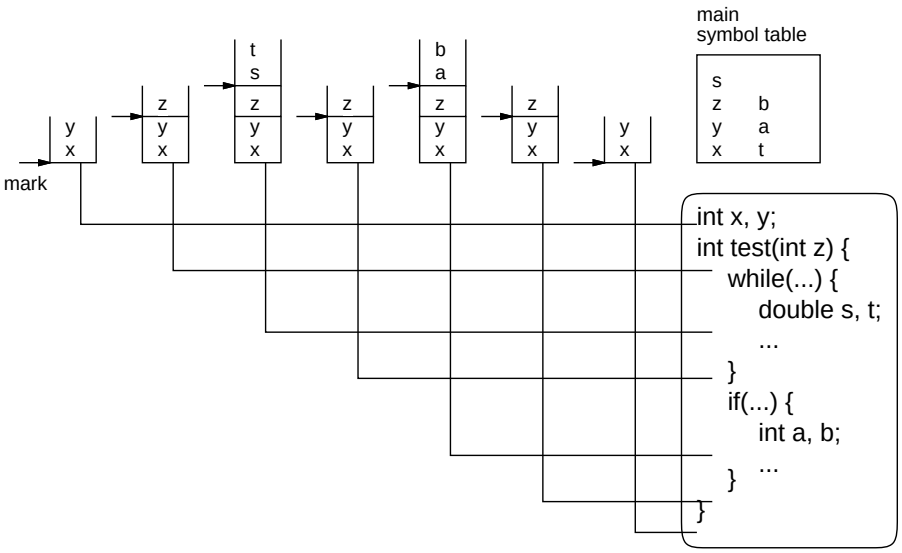

図 10.3: スコープをスタックで管理する記号表

図で左上の名前が記されている部分は、実際にはテーブル本体の項目へのポインタなどで表現できます。この方法では、スコープに入るごとにマークをセットしてその上にそのスコープで定義された名前を積んで行き、スコープから出るときにそのスコープのぶんは消去してマークも 1 つ外側のスコープに対応する位置まで戻せばよいので、分かりやすくなります。また、線形探索する長さも現在の位置から見える名前のぶんだけになるので、先の方法よりも効率が良くなります。

ここまででは、記号表に名前だけを入れていましたが「どの関数の中でどの変数が定義されているか」などの情報を保持しようと思うと、むしろ関数ごとに 1 つの記号表を作るようにする方が自然です。

また、モジュールやクラスなどの機能のある言語では (Java もそうです)、変数や手続きがモジュールやクラスに所属することになるので、それぞれのモジュールごとに表があり、その中に手続きの表がある、など複雑な構造が必要になります。基本的に記号表は、ソースプログラムにある名前の情報を何らかの形で表現したデータ構造になっている必要があるわけです。

演習 10-2 先の例題の記号表を図 10.3 のような内部構造で作り直してみなさい。機能は同じままにすること。

演習 10-3 Java 言語のさまざまな名前の情報をすべて保持できるような記号表のデータ構造を設計してみなさい。

10.2.5 前方参照と分割翻訳

前方参照 (forward reference) とは一般に、ソースコード上で先 (下) のにある情報を参照することを言います。これに対して後方参照 (backward reference) は、前 (上) にある情報を参照することです (図 10.4)。

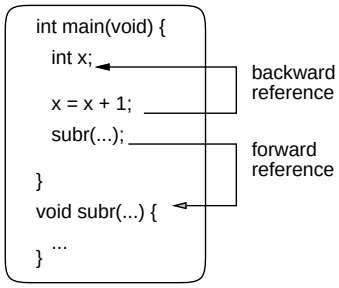

図 10.4: 前方参照・後方参照

後方参照は処理系にとっては「既に出て来たものの情報を取得」するだけなので、ソースコードを順に処理しながら記号表に登録し、参照するときは記号表を検索すればよいわけです。では前方参照はどうやって扱えばいいでしょうか。

基本的には、まずソースコードを処理しながら名前の情報を記号表に登録する処理だけをおこない、その後で改めて名前の参照が必要な処理をおこなう、というふうにコンパイラの処理を複数回に分けます。このような「ひとまとまりの処理」をコンパイラの**パス** (pass) といい、複数回のパスを持つコンパイラを**マルチパス** (multi-pass) コンパイラと呼びます。[3]

これに対し、1 回のパスだけですべての処理を済ませるものは**ワンパス** (single-pass) コンパイラです。過去においてはコンピュータの CPU 能力やメモリ容量が限られていたのでパス数を減らすことがしばしば目標となりましたが、今は抽象構文木を保持して複数パスするやりかたが主流です。ただ、プログラミング言語の仕様については、上記の理由からワンパスで処理できるように作られている言語がまだ使われています。実は C や C++がそうです。

C や C++では関数呼出はそれに先だって (ヘッダファイルを使うなどして) プロトタイプ宣言を読ませておく必要がありますね。それは、呼び出しのところで引数や返値の型検査を行なうためにプロトタイプの情報を使うからですが、マルチパスで前方にある関数定義の情報が取れればわざわざプロトタイプを書かせる必要はないはずです。

Java ではプロトタイプ宣言は不要で呼び出し箇所より下にあるメソッドのチェックも問題なくできますが、これはマルチパスを前提とした言語仕様だからです。

ところで、C 言語でも前方にある関数が呼べるではないか、と思ったかも知れませんが、トップレベルにある関数や変数の最終的な番地や位置はコンパイラの最終段階でライブラリ内の関数や別ファイルにある関数と合わせて正しい番地が埋め込まれるようになっています。この作業を行なうプログラムのことを**リンケージエディタ** (linkage editor) ないし**リンカ** (linker) と呼びます。

そういうわけで、話題が前後しますが、リンカのおかげでプログラムを複数のファイルに分割して作成し、最後にひとまとめにして実行することができるのです。これを**分割コンパイル** (separate compilation) と呼び、大規模なプログラム作成には不可欠な機能です。

ただし、多くのリンカは番地を埋めるだけで型検査はやらないので、分割コンパイル時にファイル間でのチェックを行なうには別の方法が必要です。C や C++では、分割コンパイルする各ファイル

[3]マルチパスでも構文解析を 2 回やる必要は無く、2 パス目は 1 パス目で作った抽象構文木をたどりながら処理するなどでもよいわけです。

で共通のヘッダファイルを使うことで、型検査の情報を流通させています。

Java では、クラスファイルに型情報も記載されていて、必要に応じてクラスファイルを読み込む (無ければそのクラスも一緒にコンパイルしてクラスファイルを作る) ことにより、分割コンパイル時の型検査を可能としています。

演習 10-4 C 言語で `main.c` と `sub.c` という 2 つのソースファイルを作り、`sub.c` ではパラメタを 1 個以上受け取る関数 `sub()` を定義するが、`main.c` 側では「本物とは違う」プロトタイプ宣言を与え、両者をコンパイルして結合し、実行してみよ。さまざまな違え方でどのような値が渡るかを検討すること。

10.3 型と型検査

10.3.1 型の存在意義・強い型と弱い型

型 (type) とは「データの種別」を表すプログラミング言語の用語である。型がプログラミング言語に取り入れられた最初の理由は、コード生成時に適切な演算命令を出力するためです。たとえば、「1 + 2」も「1.0 + 2.0」も同じ足し算だから演算記号「+」を使用したいけれど、コンピュータのハードウェア上では整数の加算命令と実数の加算命令は違う命令であるので、どちらを出力するかを決める必要があります。型がある言語であれば、被演算子の型が整数なら整数用、実数なら実数用の加算命令を出力すればよいわけです。

型を扱うもう 1 つの理由は、適切でないソースコード記述を検出してエラーにすることです。たとえば `a` が配列の名前であるとき、`a * 5` という式は意味を持ちません。今日ではどちらかというと、こちらの方 (適切でない用途を検出すること) が型の主要な役割になってきています。

上記の説明はいずれも、ソースコードにおいて変数等の型宣言を行ない、翻訳時にチェックや適切な命令の生成を行なう、という文脈で説明していました。このような、翻訳時に型検査を行なう言語を、**強い型** (strongly typed) の言語と呼びます。C や Java は強い型の言語の例です。

これと対照的に、ソースコード上では型宣言を行なわず、翻訳時の型検査をおこなわない (ないし行なうこともあるが強制しない) 言語を、**弱い型** (weakly typed) の言語と呼びます。JavaScript、Python、Ruby など多くのスクリプト言語はこちらに属します。

弱い型の言語でも、型は存在する。これらの言語でも、整数、実数、配列など多様なデータを扱い、それらの種類が型に対応するからです。ただし、ソースコード上では型を宣言しないので、翻訳時にはそれぞれの箇所で扱っているデータの型は分かりません。

ではどうするのかというと、これらの言語では実行時に値に型の情報が付随していて (RTTI — runtime type information)、それを参照して適切な動作を行なうようになっています。たとえば、次の Ruby の実行例を見てみましょう。

```
irb> def addorconcat(x, y) return x + y end
=> :addorconcat
irb> addorconcat 1, 2
=> 3
irb> addorconcat "a", "b"
=> "ab"
```

メソッド `addorconcat` は、2 つ値を受け取り、それに対して「+」演算を施して返しますが、データが数値なのか (足し算になる)、文字列なのか (文字列連結になる) かは、実行時まで分かりません。「+」が実際に実行されるときに、RTTI を参照して数値なら加算、文字列が含まれていれば連結をおこないます (片方が文字列でないなら文字列への変換も合わせておこないます)。

この方法は型宣言が不要なのでコードは簡潔になります、そのかわり翻訳時に型をチェックしないため、翻訳時には型の間違いが検出できなません。不適切な操作はそこを実行たときにエラーとして

検出されますが、あらゆる経路の実行を行なうテストというのは実質的には不可能です。また、実行時に RTTI を用いて判定し、適切な操作を行なうように切替えるため、処理時間が長くなるという弱点もあります。[4]

強い型の言語の得失はこの裏返しで、宣言が繁雑になりがちですが、翻訳時に型の間違いまでチェックでき、実行が高速にしやすい (たとえば整数どうしの加算と分かっていれば加算命令を 1 つ実行するだけですむ)、という特徴があります。また、プログラムを書いたり読んだりする際に、「このデータはこのような種類のものである」ということが明示され意識される、ということも利点の 1 つだと言えます。

10.3.2 型検査のアルゴリズム

では実際に強い型の言語における型検査はどのようにすればいいのでしょう。それは基本的には簡単です。式を表す抽象構文木の葉のところには変数や定数が来ますが、変数の型は型宣言により定まりますし、定数はその形から型が分かります。そして、各ノードはその種類ごとに子ノードの型から自分の型を決めることができます。図 10.5 を見てください。

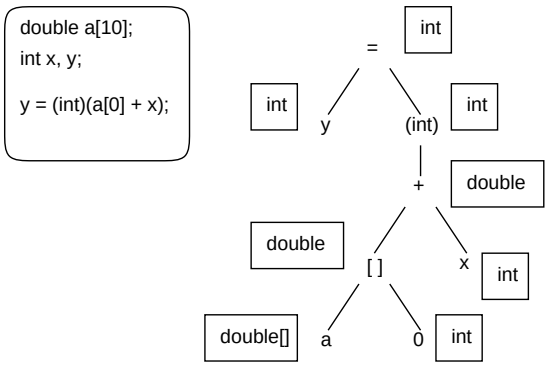

図 10.5: 抽象構文木の上での型割り当て

配列 a は double の配列であり、定数 0 は int ですから、添字演算は OK で (添字の型は整数である必要があります)、その結果は double です。それと int の変数 x を足すことは OK で [5] その結果は double です。キャストで double の値を int にすることは OK で、代入の左辺 y も int 型ですから、代入も OK でその結果は int になります。

このように、式のすべての箇所に型が規則通り割り当てられるなら型検査は OK になりますし、型が未定だったり矛盾が生じるようならエラーです。どのような演算や代入でどのような条件を満たすべき、ということはそれぞれの言語仕様により定められています (ここの例よりもかなり処理が複雑になる言語もあります)。

10.4 例題: 配列型を持つ強い型の小さな言語

では前節の内容を実践する「強い型の小さな言語」を実装してみます。文法は次の通りです。基本的にはこれまでにやってきた「小さな言語」と同じですが、変数宣言があり、配列も宣言できます。

statlist ::= *statlist stat* | ϵ
stat ::= `int` *ident* `;` | `int` *ident* `[` *iconst* `]` `;` | *ident* `=` *expr* `;`
| *ident* `[` *expr* `]` `=` *expr* `;` | `read` *ident* `;` | `print` *expr* `;`
| `if` `(` *expr* `)` *stat* | `while` `(` *expr* `)` *stat* | `{` *statlist* `}`
expr ::= *expr* `>` *expr* | *expr* `<` *expr* | *nexp*

[4] ただし今日では CPU が高速になったことと、実行時の最適化技術が進歩したことから、速度のハンディキャップはかなり小さくなっています。

[5] このとき int を double に変換してから足す必要があるので、構文木上で変換のノードを挿入するのが普通でしょう。

$nexp$ ::= $nexp$ + $nexp$ | $nexp$ - $nexp$ | $term$
$term$ ::= $term$ * $fact$ | $term$ / $fact$ | $fact$
$fact$ ::= $iconst$ | $ident$ | $ident$ [$expr$] | ($expr$)

では、これに対応する SableCC のソースを見てみましょう。

```
Package sama2;

Helpers
  digit = ['0'..'9'] ;
  lcase = ['a'..'z'] ;
  ucase = ['A'..'Z'] ;
  letter = lcase | ucase ;

Tokens
  iconst = digit+ ;
  blank = (' '|13|10)+ ;
  int = 'int' ;
  if = 'if' ;
  while = 'while' ;
  read = 'read' ;
  print = 'print' ;
  semi = ';' ;
  assign = '=' ;
  add = '+' ;
  sub = '-' ;
  aster = '*' ;
  slash = '/' ;
  lt = '<' ;
  gt = '>' ;
  lbra = '{' ;
  rbra = '}' ;
  lpar = '(' ;
  rpar = ')' ;
  lsbr = '[' ;
  rsbr = ']' ;
  ident = letter (letter|digit)* ;

Ignored Tokens
  blank;

Productions
  prog = {stlist} stlist
       ;
  stlist = {stat} stlist stat
         | {empty}
         ;
```

```
  stat = {idcl}    int ident semi
       | {adcl}    int ident lsbr iconst rsbr semi
       | {assign}  ident assign expr semi
       | {aassign} ident lsbr [idx]:expr rsbr assign expr semi
       | {read}    read ident semi
       | {print}   print expr semi
       | {if}      if lpar expr rpar stat
       | {while}   while lpar expr rpar stat
       | {block}   lbra stlist rbra
       ;
  expr = {gt}  [left]:nexp gt [right]:nexp
       | {lt}  [left]:nexp lt [right]:nexp
       | {one} nexp
       ;
  nexp = {add}  nexp add term
       | {sub}  nexp sub term
       | {one}  term
       ;
  term = {mul}  term aster fact
       | {div}  term slash fact
       | {one}  fact
       ;
  fact = {iconst} iconst
       | {ident}  ident
       | {aref}   ident lsbr expr rsbr
       | {one}    lpar expr rpar
       ;
```

```
package sama2;
import sama2.analysis.*;
import sama2.node.*;
import java.io.*;
import java.util.*;

class TypeChecker extends DepthFirstAdapter {
  Symtab st;
  HashMap<Node,Integer> vtbl;
  public TypeChecker(Symtab tb, HashMap<Node,Integer> vt) { st = tb; vtbl = vt; }
  private void ckv(Node n, int t, int l, String s) {
    if(getOut(n) instanceof Integer && (Integer)getOut(n) == t) { return; }
    Log.pError(l+": "+s);
  }
  private int cki(String i, int t, int l) {
    Symtab.Ent e = st.lookup(i);
    if(e.type != t) { Log.pError(l+": identyfier is not of expexted type: "+i); }
    return e.pos;
```

```
    }
    @Override
    public void outAIdclStat(AIdclStat node) {
      st.addDef(node.getIdent().getText(), Symtab.ITYPE);
    }
    @Override
    public void outAAdclStat(AAdclStat node) {
      Symtab.Ent e = st.addDef(node.getIdent().getText(), Symtab.ATYPE);
      vtbl.put(node, e.pos);
    }
    @Override
    public void outAAssignStat(AAssignStat node) {
      int p = cki(node.getIdent().getText(), Symtab.ITYPE, node.getIdent().getLine());
      ckv(node.getExpr(), Symtab.ITYPE, node.getAssign().getLine(), "non-int val");
      vtbl.put(node, p);
    }
    @Override
    public void outAAassignStat(AAassignStat node) {
      int p = cki(node.getIdent().getText(), Symtab.ATYPE, node.getIdent().getLine());
      ckv(node.getIdx(), Symtab.ITYPE, node.getLsbr().getLine(), "non-int index");
      ckv(node.getExpr(), Symtab.ITYPE, node.getAssign().getLine(), "non-int val");
      vtbl.put(node, p);
    }
    @Override
    public void outAReadStat(AReadStat node) {
      int p = cki(node.getIdent().getText(), Symtab.ITYPE, node.getIdent().getLine());
      vtbl.put(node, p);
    }
    @Override
    public void outAPrintStat(APrintStat node) {
      ckv(node.getExpr(), Symtab.ITYPE, node.getPrint().getLine(), "non-int val");
    }
    @Override
    public void outAIfStat(AIfStat node) {
      ckv(node.getExpr(), Symtab.ITYPE, node.getLpar().getLine(), "non-int cond");
    }
    @Override
    public void outAWhileStat(AWhileStat node) {
      ckv(node.getExpr(), Symtab.ITYPE, node.getLpar().getLine(), "non-int cond");
    }
    @Override
    public void inABlockStat(ABlockStat node) { st.enterScope(); }
    @Override
    public void outABlockStat(ABlockStat node) { st.exitScope(); }
    @Override
    public void outAGtExpr(AGtExpr node) {
```

```
    ckv(node.getLeft(), Symtab.ITYPE, node.getGt().getLine(), "non-int val");
    ckv(node.getRight(), Symtab.ITYPE, node.getGt().getLine(), "non-int val");
    setOut(node, new Integer(Symtab.ITYPE));
  }
  @Override
  public void outALtExpr(ALtExpr node) {
    ckv(node.getLeft(), Symtab.ITYPE, node.getLt().getLine(), "non-int val");
    ckv(node.getRight(), Symtab.ITYPE, node.getLt().getLine(), "non-int val");
    setOut(node, new Integer(Symtab.ITYPE));
  }
  @Override
  public void outAOneExpr(AOneExpr node) { setOut(node, getOut(node.getNexp())); }
  @Override
  public void outAAddNexp(AAddNexp node) {
    ckv(node.getNexp(), Symtab.ITYPE, node.getAdd().getLine(), "non-int val");
    ckv(node.getTerm(), Symtab.ITYPE, node.getAdd().getLine(), "non-int val");
    setOut(node, new Integer(Symtab.ITYPE));
  }
  @Override
  public void outASubNexp(ASubNexp node) {
    ckv(node.getNexp(), Symtab.ITYPE, node.getSub().getLine(), "non-int val");
    ckv(node.getTerm(), Symtab.ITYPE, node.getSub().getLine(), "non-int val");
    setOut(node, new Integer(Symtab.ITYPE));
  }
  @Override
  public void outAOneNexp(AOneNexp node) { setOut(node, getOut(node.getTerm())); }
  @Override
  public void outAMulTerm(AMulTerm node) {
    ckv(node.getTerm(), Symtab.ITYPE, node.getAster().getLine(), "non-int val");
    ckv(node.getFact(), Symtab.ITYPE, node.getAster().getLine(), "non-int val");
    setOut(node, new Integer(Symtab.ITYPE));
  }
  @Override
  public void outADivTerm(ADivTerm node) {
    ckv(node.getTerm(), Symtab.ITYPE, node.getSlash().getLine(), "non-int val");
    ckv(node.getFact(), Symtab.ITYPE, node.getSlash().getLine(), "non-int val");
    setOut(node, new Integer(Symtab.ITYPE));
  }
  @Override
  public void outAOneTerm(AOneTerm node) { setOut(node, getOut(node.getFact())); }
  @Override
  public void outAIconstFact(AIconstFact node) {
    setOut(node, new Integer(Symtab.ITYPE));
  }
  @Override
  public void outAIdentFact(AIdentFact node) {
```

```
      Symtab.Ent e = st.lookup(node.getIdent().getText());
      setOut(node, new Integer(e.type)); vtbl.put(node, e.pos);
    }
    @Override
    public void outAArefFact(AArefFact node) {
      int p = cki(node.getIdent().getText(), Symtab.ATYPE, node.getIdent().getLine());
      ckv(node.getExpr(), Symtab.ITYPE, node.getLsbr().getLine(), "non-int index");
      setOut(node, new Integer(Symtab.ITYPE)); vtbl.put(node, p);
    }
    @Override
    public void outAOneFact(AOneFact node) { setOut(node, getOut(node.getExpr())); }
  }
```

ではこれを動かす main() を見てみます。記号表を作り、パーサを作り、解析し、記号表を表示します。この後の実行部分はコメントアウトしてあります。

```
  package sama2;
  import sama2.parser.*;
  import sama2.lexer.*;
  import sama2.node.*;
  import java.io.*;
  import java.util.*;

  public class SamA2 {
    public static void main(String[] args) throws Exception {
      Parser p = new Parser(new Lexer(new PushbackReader(
        new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
          1024)));
      Start tree = p.parse();
      Symtab st = new Symtab();
      HashMap<Node,Integer> vtbl = new HashMap<Node,Integer>();
      TypeChecker tck = new TypeChecker(st, vtbl); tree.apply(tck); st.show();
//    if(Log.getError() > 0) { return; }
//    Executor exec = new Executor(vtbl, st.getGsize()); tree.apply(exec);
    }
  }
```

演習 10-5 「型のある小さな言語」で簡単なプログラムを書いてみなさい。変数の未定義や配列と変数の間違いなどを入れてみて、確かに型検査されていることを確認しなさい。

演習 10-6 「型のある小さな言語」を次のように拡張してみなさい。もちろん、型検査はきちんと行なわれること。

a. 変数に初期値が書けるようにする。
b. 配列の変数どうしの代入もできるようにする。
c. 配列リテラル ([1, 2, 3] のようなもの) を入れる。
d. その他、やってみたいと思う拡張。

10.5 課題 10A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 9」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 強い型の言語と弱い型の言語のどちらが好みですか。またそれはなぜ。
  Q2. 記号表と型検査の実装について学んでみて、どのように思いましたか。
  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

10.6 実行するための Exectutor のコード

```
package sama2;
import sama2.analysis.*;
import sama2.node.*;
import java.io.*;
import java.util.*;

class Executor extends DepthFirstAdapter {
  Scanner sc = new Scanner(System.in);
  PrintStream pr = System.out;
  HashMap<Node,Integer> pos;
  Object[] vars;
  public Executor(HashMap<Node,Integer> p, int s) { pos = p; vars = new Object[s]; }
  @Override
  public void outAAdclStat(AAdclStat node) {
    vars[pos.get(node)] = new Object[Integer.parseInt(node.getIconst().getText())];
  }
  @Override
  public void outAAssignStat(AAssignStat node) {
    vars[pos.get(node)] = getOut(node.getExpr());
  }
  @Override
  public void outAAassignStat(AAassignStat node) {
    Object[] a = (Object[])vars[pos.get(node)];
    a[(Integer)getOut(node.getIdx())] = getOut(node.getExpr());
  }
```

```
@Override
public void outAReadStat(AReadStat node) {
  String s = node.getIdent().getText().intern();
  pr.print(s + "> "); vars[pos.get(node)] = sc.nextInt(); sc.nextLine();
}
@Override
public void outAPrintStat(APrintStat node) {
  pr.println(getOut(node.getExpr()).toString());
}
@Override
public void caseAIfStat(AIfStat node) {
  node.getExpr().apply(this);
  if((Integer)getOut(node.getExpr()) != 0) { node.getStat().apply(this); }
}
@Override
public void caseAWhileStat(AWhileStat node) {
  while(true) {
    node.getExpr().apply(this);
    if((Integer)getOut(node.getExpr()) == 0) { return; }
    node.getStat().apply(this);
  }
}
@Override
public void outAGtExpr(AGtExpr node) {
  if((Integer)getOut(node.getLeft()) > (Integer)getOut(node.getRight())) {
    setOut(node, new Integer(1));
  } else {
    setOut(node, new Integer(0));
  }
}
@Override
public void outALtExpr(ALtExpr node) {
  if((Integer)getOut(node.getLeft()) < (Integer)getOut(node.getRight())) {
    setOut(node, new Integer(1));
  } else {
    setOut(node, new Integer(0));
  }
}
@Override
public void outAOneExpr(AOneExpr node) { setOut(node, getOut(node.getNexp())); }
@Override
public void outAAddNexp(AAddNexp node) {
  int v = (Integer)getOut(node.getNexp()) + (Integer)getOut(node.getTerm());
  setOut(node, new Integer(v));
}
@Override
```

```
  public void outASubNexp(ASubNexp node) {
    int v = (Integer)getOut(node.getNexp()) - (Integer)getOut(node.getTerm());
    setOut(node, new Integer(v));
  }
  @Override
  public void outAOneNexp(AOneNexp node) { setOut(node, getOut(node.getTerm())); }
  @Override
  public void outAMulTerm(AMulTerm node) {
    int v = (Integer)getOut(node.getTerm()) * (Integer)getOut(node.getFact());
    setOut(node, new Integer(v));
  }
  @Override
  public void outADivTerm(ADivTerm node) {
    int v = (Integer)getOut(node.getTerm()) / (Integer)getOut(node.getFact());
    setOut(node, new Integer(v));
  }
  @Override
  public void outAOneTerm(AOneTerm node) { setOut(node, getOut(node.getFact())); }
  @Override
  public void outAIconstFact(AIconstFact node) {
    setOut(node, new Integer(node.getIconst().getText()));
  }
  @Override
  public void outAIdentFact(AIdentFact node) {
    setOut(node, vars[pos.get(node)]);
  }
  @Override
  public void outAArefFact(AArefFact node) {
    Object[] a = (Object[])vars[pos.get(node)];
    setOut(node, a[(Integer)getOut(node.getExpr())]);
  }
  @Override
  public void outAOneFact(AOneFact node) { setOut(node, getOut(node.getExpr())); }
}
```

では、次のような簡単なプログラムを実行してみましょう。n を入力し、配列に n 個の値を入力し、逆順に出力しています。なお「-1」がうまく扱えないので、「0-1」と演算しています。

```
int a[100]; int n; int i; int d;
read n;
i = 0; while(i < n) { read d; a[i] = d; i = i + 1; }
i = n - 1; while(i > 0-1) { print a[i]; i = i - 1; }
```

実行のようすは次の通り。

```
% java Sam92 test.min
 a_[2 0]
 n_[1 1]
```

```
 i_[1 2]
 d_[1 3]
n? 3
d? 8
d? 9
d? 10
10
9
8
%
```

＃11　実行時環境とコード生成

11.1　実行時環境と記憶域の配置

11.1.1　実行時環境とは

実行時環境 (runtime environment) とは，簡単にいえばコンパイラが生成した目的コードが実行されるときの約束ごとです。具体的には，次のようなものが実行時環境の規約に含まれます。

- 記憶領域の配置や割り当て方
- 変数の種類ごとのアクセス方法
- 関数の呼び出し方/呼び出され方や引数の渡し方/渡され方
- 生成コードと実行時ライブラリの分担

これらの約束ごとを設ける主な目的は、ソースコードに現れてくる名前 (変数名、手続き名、ラベル等) とそれらが表す実体 (記憶領域や命令列の特定の場所) の対応ないし**束縛** (binding) の管理を効率よく実現することだと言えます。

例えばソースプログラム中の変数 x は、実行時にはいずれかのメモリ番地に無ければなりません (または CPU レジスタのどれかに置かれることもあるかも知れませんが)。x がグローバル変数であれば、特定の番地に**割り当てる** (allocate) ことができます。

ローカル変数であれば、手続きが実行開始される時にローカル変数群を配置する領域割り当て、実行終了時にそれを**解放** (deallocate) しますが、その領域内での「何番目の位置 (オフセット)」ということはコンパイル時に決めておき、その領域の先頭を指すレジスタからどれだけ先、というアクセス方法を使うことで、効率よく扱います。パラメタについても同様のことをします。

このような、コンパイル時に変数の位置 (領域内のオフセットも含む) を決めてしまうことを、**静的束縛** (static binding) ないし**早期束縛** (early binding) と呼びます。

一方、名前とその実体の結び付きがコンパイル時には決まっていなくて、実行時に行う場合は**動的束縛** (dynamic binding) ないし**遅延束縛** (late binding) といいます。動的束縛が必要な例として、オブジェクト指向言語 (Java がそうです) のメソッド呼び出しが挙げられます。ある名前でメソッドを呼び出しても、実際にどのメソッドが呼ばれるかは実行時にしか決まらないからです。

動的束縛については回を改めて扱うこととし、以下では静的束縛を前提として説明していきます。まず記憶域の種別と配置について述べ、スタックとそれに関連する引数や返値の受け渡し、環境の切換えについて説明します。

11.1.2　記憶領域の種別と割当て

プログラムが実行時に参照する記憶領域は、基本的には次のように分類できます。

- コード領域 — プログラムの命令を保持する。
- 定数領域 — 定数 (初期設定され、書き換えられないデータ) を保持する。
- 初期設定データ領域 — 初期設定され、書き換えられるデータの領域。
- 非初期設定データ領域 — 初期設定されないデータの領域。

- ヒープ領域 — 実行時に動的に割り当てられるデータの領域。
- スタック領域 — 局所変数など手続き呼出しに付随して割り当てられるデータ、戻り番地、その他管理情報の領域。

これらの区分はOSやハードウェアによってサポートされる場合もありますし、処理系の中だけの規約として実現される場合もあります。多くのシステムではハードウェアの機能を活用してコード領域や定数領域を書換え不可能なように保護し、誤りによってプログラムが書き換わってしまうことを防ぎます。

図 11.1 に、典型的な記憶域配置の例を示します (CPU の機能や OS の作り方によっては、もっとアドレス空間がバラバラになっている場合もあります)。ここでは**スタック** (stack) はアドレスが若い方向に伸びるように描きましたが、その配置や伸びる方向についても、通常ハードウェアやオペレーティングシステムによってはこれと違っています。

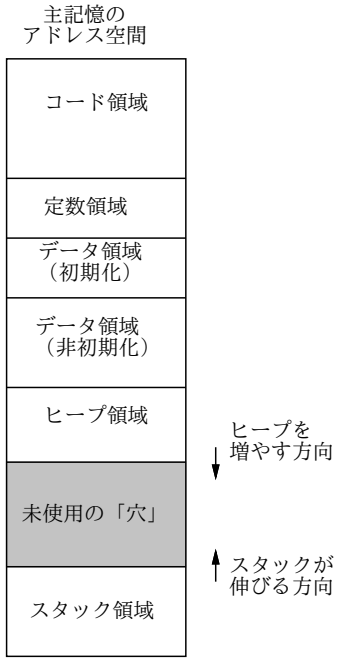

図 11.1: 典型的な実行時記憶域配置の例

実行時に動的に大きさが変化するのはスタックと**ヒープ** (heap) なので、この図のようにそれらを「向かい合せに」配置することがよくあります。現在では**メモリマッピング** (memory mapping) の機能が CPU に備わっていることが多いので、これらをアドレス空間上では離してとっておき、必要のつど実際のメモリページを割り当てることが通例です。

スタックとヒープ以外の領域については、大きさが実行時に変化することはないので、単に設計した配置に従って位置を割り当てるだけで済みます。さらに、リンカを用いる場合 (これが普通) は、目的コード中に「コード」「初期化データ」「非初期化データ」などの種別を含めるようになっていて、リンカが同種の領域をまとめながら詰め合せます。

また、記憶配置にはデータの種類に応じた**境界揃え** (boundary alignment) も考慮する必要があります。例えば多くのハードウェアでは 4 バイト長の整数や実数は 4 の倍数番地、8 バイト長の実数やアドレス値は 8 の倍数番地に配置する必要があります (そうしないとエラーになる CPU と、エラーにはならないけれど速度が低下する CPU があります)。

このため、変数に番地を割り当てる際には、そこに入るデータの種類に応じてあきを挿入して番地を境界に揃えます (または境界揃えを指定するアセンブラ用の指示を挿入します)。レコードのように複数のデータ型が混在する領域の場合には、その内部にも境界揃えのためのすきまが必要かもしれま

せん。スタックやヒープ上の領域でも同様の配慮が必要です。

11.2 スタックとスタックフレーム

11.2.1 スタックの用途

スタック (stack) とは一般には、一番最後に割り当てられた領域が一番先に解放される (LIFO — last in, first out) ような割当てを実現するデータ構造です。

プログラミング言語における手続き呼出しでは、手続きからの戻りでは、一番最後に呼び出された手続きの呼び出し地点に戻るので、call 命令 (手続き呼び出し命令) は戻り番地 (call 命令の次の命令の番地) をスタックに積んでから手続きにジャンプし、ret 命令 (戻り命令) はスタックから戻り番地を取り降ろしてそこにジャンプするようにするようになっています。[1]

そして、一番最近に呼び出された手続きが一番最初に戻るので、手続き実行に付随する記憶領域 (局所変数や一時変数) もスタックに割り当てるのが自然です。これを含めて、手続き呼出しに付随する情報としては次のものがあげられます。

(a) 手続きに局所的な作業領域

(b) 戻り番地の情報

(c) 引数の情報/戻り値の情報

(d) 呼出し元の領域を示す情報

(e) 外側のスコープを示す情報

(f) 呼びに伴って壊れると困るレジスタ内容の写し

多くの手続き型言語の実現では戻り番地用とその他の領域用のスタックを兼ねて 1 本ですませますが、Prolog などでは呼出しと領域割当て/解放が同期しないため、呼出しスタックとデータ領域スタックを分ける必要がある。また Lisp のようにごみ集めを必要とする場合は、一般のデータを割り当てるスタック (データスタック) と手続きの呼び/戻り情報を積むスタック (制御スタック) を分けることもあります。以下では 1 本のスタックを用いた実現について説明していきます。

11.2.2 スタックフレームとフレームポインタ

1 本のスタックを用いる実現では、前節 (a)〜(f) の情報を各手続き呼出しごとに 1 組にまとめてスタック上に割り当てます。これを**スタックフレーム** (stack frame) ないし**活性レコード** (activation record) と呼びます。典型的なスタックフレームの形を図 11.2 に示しました。フレームの大きさは、局所変数に大きさが実行時に変化するもの (可変長配列など) を含まない限り、手続きごとに翻訳時に決まります。

引数の受け渡しと環境の切換えについては次節以降で説明するので、以下ではそれら以外の部分について説明します。

まず手続きが呼び出された時点でのスタックの状態について考えてみます。多くの命令セットアーキテクチャでは、スタックの先頭を指すレジスタである**スタックポインタ** (stack pointer、SP) が決められています。[2]

上で述べたように、手続き呼出し命令 (ないし呼出し規約) によって、戻り番地がスタックの上に積まれます。この様子を図 11.3(a) に示しました。

次にこの上に局所変数の領域を確保しますが、それにはスタックポインタを必要なだけずらせばよいです。もっとも、スタックは無限にあるわけではないので、領域を確保する際にスタック領域に十

[1] 極めて古い CPU ではそういう技法が開発されていなくて色々変わった動作のものがありましたが、今はほとんどの CPU がそのような設計になっています。

[2] ハードウェアとしては決まっていなくて、オペレーティングシステムの規約としてスタックポインタに使うレジスタを規定する場合もあります。

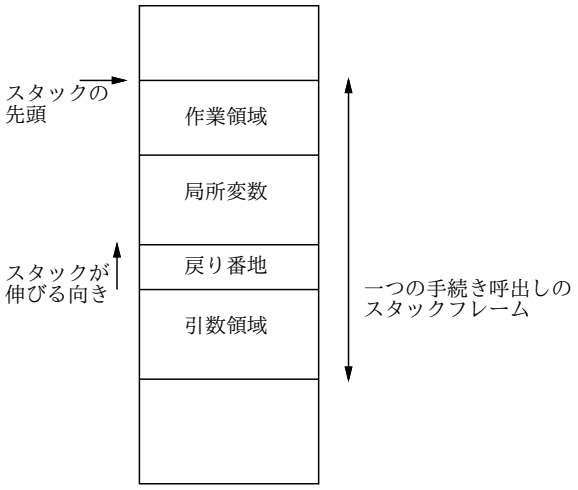

図 11.2: 典型的なスタックフレームの例

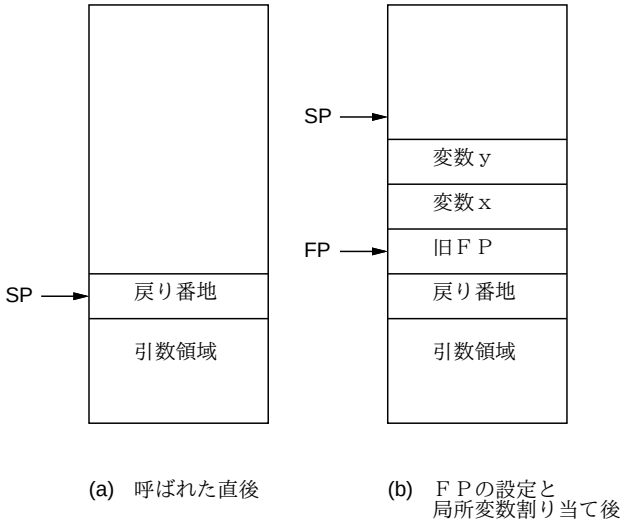

図 11.3: フレームポインタ

分なあきがあるかどうか検査する必要があります。ただし、メモリマッピングを用いてスタック領域を自動管理してくれる OS の場合には、あらかじめ決められた大きさまでスタックは自動拡張され、それを越えるとシステムがその旨通知してくれるので検査は不要です。

スタック上に確保した領域のアクセスは、この領域の番地が手続き呼出しごとに変わるため、固定番地による指定では行えません。ほとんどの CPU ではこのために「あるレジスタの指す場所から n バイト先」という番地指定ができます。

そこで、局所変数領域の先頭を指すレジスタを用いて、そのレジスタを起点にスタック上の領域にアクセスします。このレジスタを、スタックフレームの起点を指すことから**フレームポインタ** (frame pointer、FP) ないし**ベースポインタ** (base pointer) と呼びます。この様子を図 11.3 (b) に示します。例えばある手続きで大きさ 4 バイトの局所変数 x を起点の 4 バイト上、y を 8 バイト上に割り付けたとすれば、y の値を x に代入するには次のようなコードを出せばよいのです (命令やレジスタ名は x86-64 のもの。x86-64 については後述します)。

```
movel  -4(%rbp),%eax
movel  %eax,-8(%rbp)
```

しかし、フレームポインタは別の手続きを呼ぶとそこでも同様に利用するので、戻って来たときには別の値に書き変わっています。そこで、どの手続きでも呼ばれたらまずフレームポインタをスタックに格納し、その場所をフレームの起点だと考えてフレームポインタにはその番地を入れるようにします。手続きから戻るときには起点に入っている旧フレームポインタ値をフレームポインタに入れ直すことで、もとの値が復旧できます。

このようにすると、フレームポインタは常に 1 つ前のスタックフレームのフレームポインタを保存した番地を指しているので、フレームポインタから始まる連鎖をたどることでスタック上のフレームを新しいものから順にたどることができます。これを利用して、誤りや実行中断点 (ブレークポイント) への到達によって停止したプログラムの状況を調べ、その時点で各手続きの局所変数などを調べるデバッガをつくることができます。

コード上の各場所でスタックポインタとフレームポインタがどれくらい離れているかは、局所変数として大きさが実行時に変化する領域を割り当てない限りは、翻訳時に分かります。したがって、スタックポインタとフレームポインタという 2 つのレジスタをこのために割り当て管理するのは無駄であり、どちらか 1 つだけですませることも可能ではあります (可変長配列などを割り当てる手続き内では特別に両方使うようにすればよい)。実際、そのようなコンパイラも存在しますが、ただし、その場合にはデバッガなどのために「どの番地を走っているときはスタックポインタとフレームポインタはどれだけ離れているか」の情報を別に用意して参照させる必要があります。ここでは当面簡単さを優先させて、2 つのレジスタを使用することを前提としておきます。

11.2.3 引数と返値の受け渡し

引数 (arguments) と**返値** (return value) は呼ぶ手続ないし**呼び側** (caller) と呼ばれる手続きないし**呼ばれ側** (callee) の間で受け渡されるため、両方の手続きからアクセスされます。言い換えれば、スタックフレームはある手続きに固有の環境を表しますが、例外として引数だけは隣接フレームと共有されます。返値も手続き間で受け渡されますが、戻りによって呼ばれ側のスタックフレームは消滅するので、共有は起きません。

以下では引数の各種受け渡し機構および返値の渡し方について説明します。なお、今日のシステムでは高速化のため、引数を多数あるレジスタに載せて渡しますが、ここではまずスタック (メモリ) 経由の方法を基本として説明しています。また、引数について呼び側から見る場合と呼ばれ側から見る場合で区別するため、前者を実引数、後者を仮引数と記します。

a. 値呼び

値呼び (call by value) は最も単純かつ基本的な受け渡し機構であり、呼び側では任意の式の値を実引

数として渡します。呼ばれ側では仮引数は渡された値を初期値として持つような局所変数として扱います (ただし Pascal などでは値渡し引数への代入を許しません)。

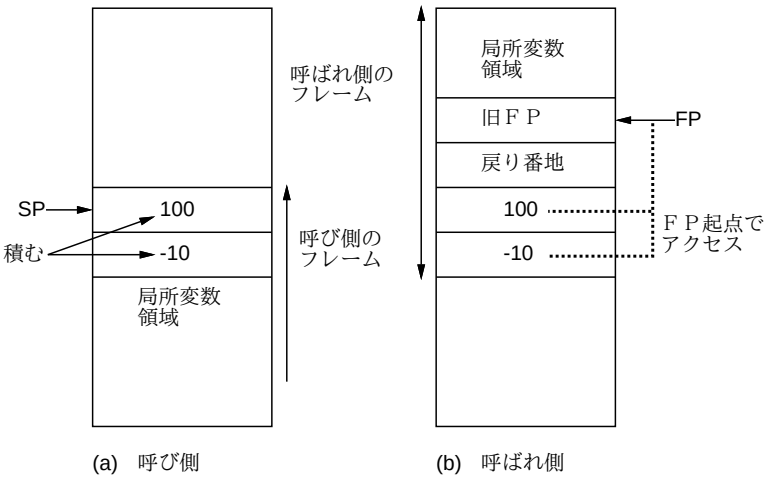

図 11.4: 値呼びの実現

値呼びを実現するには、各実引数の値を計算し、それを順にスタック上に積むだけですみます。例えば `sub(100, -10)` という呼出しの場合、図 11.4(a) のように、`100` と `-10` をスタックに積み、そのまま `sub` を呼び出します。`sub` の方では仮引数を `a`、`b` という名前でアクセスするとすれば、それらは (旧フレームポインタと戻り番地が 84 バイトの領域であるとすれば) フレームポインタ起点で 16 バイトおよび 20 バイト手前 (大きい番地) にあることになります。そこで、例えば `a + b` という式であれば次のように命令を出力します。

```
movl   16(%rbp),%eax
addl   20(%rbp),%eax
```

つまり、他の局所引数と同等の命令でアクセスすることができるわけです。なお、この例ではスタック上に実引数を右から順に積んでいますが、こうしておくと引数の数が可変で、実際に渡した数は第 1 引数を調べるとわかるような関数 (C の `printf` など) が素直に実現できます (第 1 引数の位置は常に `16(%rbp)` で、残る引数はそれに隣接するから)。可変引数を使わなかったり、実引数の数を別の方法で受け渡すなら左から積むのでもかまわないことになります。

b. 参照呼び

値呼びは単純で分かりやすいですが、呼ばれ側から呼び側に情報を渡すうえでは不便です。これに対し、仮引数に対する更新がただちに実引数にも及ぶような呼出し機構が**参照呼び** (call by reference) です。

参照呼びでは図 11.5(a) のように、スタックには実引数の入った場所の番地を積んで渡し、呼ばれ側ではこの番地情報を読み出して実引数にアクセスします。

例えば上の例で呼ばれ側において `a = a + b` を実行するコードは次のようになります (アドレスが 8 バイトとすると、2 つのパラメタのオフセットは今度は 16 と 24 になります。また q がついた命令は 64 ビットを扱います)。

```
movq  24(%rbp),%rax
movl  (%rax),%edx
movq  16(%rbp),%rax
addl  (%rax),%edx
movl  %edx,(%rax)
```

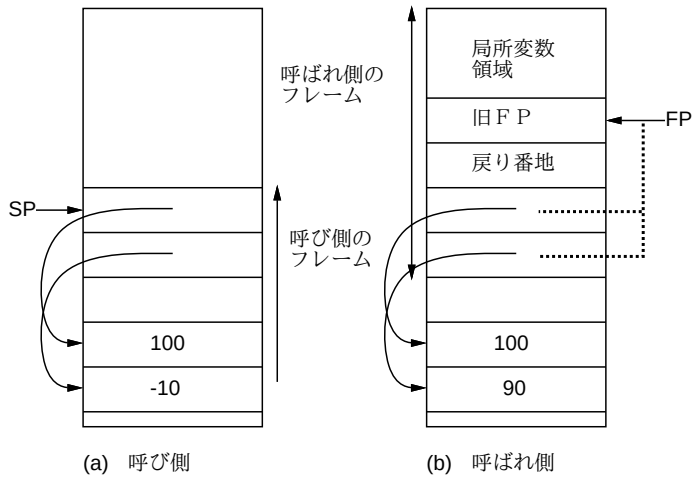

図 11.5: 参照呼びの実現

参照呼びにおいて実引数が定数の場合、その定数 (例えば-10) が呼ばれ側で更新されてしまうのでは困ります。この問題に対しては、Pascal のように参照呼びの実引数には変数しか書けないという言語仕様で対処することもありますが、そうでなければ定数や式を渡すときには値を適当な作業領域に格納してその番地を渡す必要があります。

c. 名前呼び

参照呼びのような引数渡し機構を採用する動機の 1 つは前述のように「呼ばれた側から読んだ側への情報伝達」ですが、もう 1 つの考え方として「手続きを、その呼出し箇所に手続き本体のコードを (適切な名前置換えの後) 埋め込むものとして理解する」という立場があり得ます。仮にそのように考えることを許すとして、例えばつぎのような手続きがあったとします。

```
procedure sums(int k, a, b; real x, result)
begin
    result := 0.0;
    for k := a to b do result := result + x;
end;
```

これを次のように呼び出すとします。

```
sums(i, 1, 10, a[i,i], r);
```

これは、以下のように書くのと等しいわけです。

```
r := 0.0;
for i := 1 to 10 do r := r + a[i,i];
```

したがって大きさ 10 の行列 a の対角要素の和が求まるはずです。しかし、実際には参照呼びの場合、仮引数 x に相当する番地を呼出し時に計算して以後それを用いるため、上記のようなことはできません。これを可能にするには**名前呼び** (call by name) とよばれる引数渡し機構が必要です。名前呼びは Algol-60 言語で最初に採用されました。

名前呼びでは、呼ばれた側で仮引数が参照されるたびに、その仮引数のありかや値を計算し直す必要があります。そのため、参照呼びのように実引数の番地を渡す代りに、実引数の番地や値を計算する手続きを渡します。この手続きを伝統的に**サンク** (thunk) と呼びます。実現方法にもよりますが、サンクは各引数ごとにその値が参照されたとき (右辺値) 用とその場所に代入するとき (左辺値) 用の対で用意することが自然です。

例えば上の例だと、k、res に対応するサンクは呼ばれると常に i、r の値 (右辺値用) と場所 (左辺値用) を返します。また a、b に対応するサンクは右辺値用のみで、常に 1 や 10 を返します。一方、x に対応するサンクは呼ばれるごとにそのときの i の値に応じて適切な a[i,i] の値や場所を計算して返します。したがって、このサンクは変数 i を参照できなければなりません。そしてもし呼ばれた手続きが別の i という変数をもっていたとしても、間違ってそれを参照してはいけません。これを実現するには、後で述べる環境の切換えを正しく行う必要があります。

名前呼びの実現は複雑であり、効率上も不利であるので、最近の言語での採用例はほとんどありません。ただし、マクロ機構 (構文上は手続き呼出しに見えるが、その部分を字面上で定義本体により置き換えたうえで翻訳する機構) を用いる場合には、起きることは名前呼びと等価になります。ただしマクロ機構ではその場に本体を展開してしまうので、呼出し機構もサンクも不要です。逆に、参照呼びや値呼びの言語で実行効率向上のためにその場展開を行う場合には、これらの呼出し機構と名前呼びとで結果が異なる場合に留意する必要があるわけです。

d. 複写復元呼び

呼ばれ側から引数を通じて呼び側に情報を返したいが、番地の間接参照が遅いなどの理由で参照呼びにしたくないときに代りに使われる方法として**複写復元呼び** (copy-restore linkage) があります。この方式では図 11.6 に示すように、値呼びと同様に値を渡してしまい、呼ばれ側でその場所を参照/更新し、戻り時にはその場所から対応する実引数の場所に値をコピーし戻すことで呼び側に情報を返します。

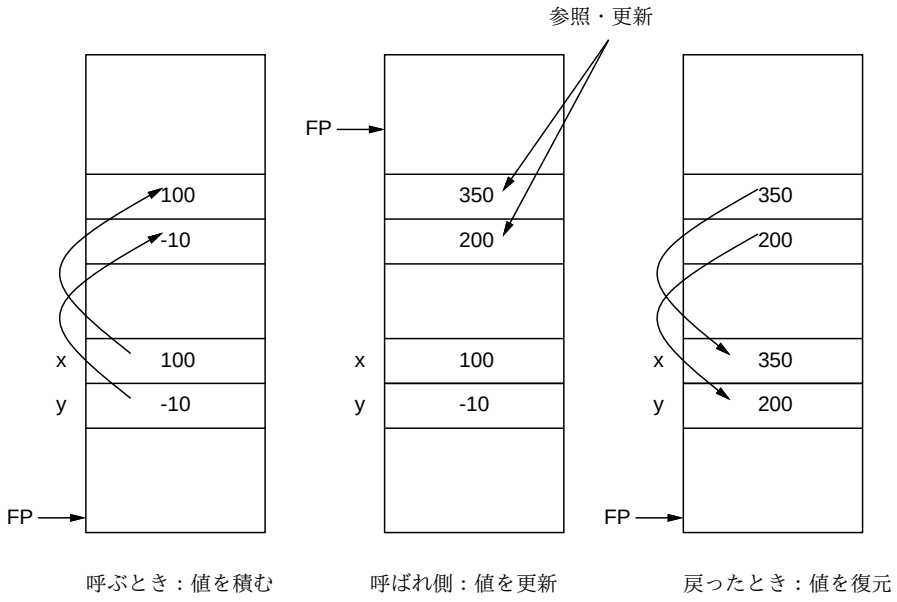

図 11.6: 複写復元呼び

複写復元呼びと参照呼びは常に同じ結果をもたらすとは限りません。例えば、次のような手続きを考えます。

```
procedure sub(integer i, j) begin i := i + 1; j := j + 1 end;
```

そして、呼ばれ方が以下のようだったとします。

```
x := 10; sub(x, x);
```

参照呼びでは sub の中で x が 2 回増やされるので、戻ってきたときの x の値は 12 になります。しかし複写復元呼びでは x の値 10 が複写されて渡され、それらが別個に増やされて 11 になり、戻りのときに x の場所に重ね書きで復元されるので x は 11 になります。言語によってはこの種の (引数渡し

機構によって動作が変化する) コードを禁止し、引数渡しを参照呼びと複写復元呼びのどちらで実現してもよいものもある。

e. レジスタによる受け渡し

ここまででは引数は全てスタック上に積まれて受け渡されるものとして扱ってきました。しかし、毎回全引数をスタックに積み (すなわち主記憶上のどこかに書き込み)、呼ばれ側でまたレジスタに読み出すのは無駄です。そこで、引数の一部または全部をレジスタに入れたまま渡す工夫も多く行われまする。その際は、どんな場合どの引数をどのレジスタで渡すかについ規約が必要になります。

ここまでは引数についての話でしたが、返値についてはとくに、特定のレジスタに入れて戻る方法が簡単で公立もよいのでよく使われます。ただし、レジスタに入らない大きさのデータの場合が問題になりますが、対策としては、そのような例外的な場合のみ呼び側で領域を確保したり、どこか適当な場所 (たぶんスタックの上の方) に返値を置き、その番地をレジスタに入れて返すなどがあります (その場合には呼び側がただちに返値を適切な場所にコピーする必要があります)。

11.2.4 レジスタの退避回復

レジスタはデータのアクセスや演算のために多様に使われますが、ある手続きから別の手続きを呼んだときには、その呼ばれ側でも同じようにレジスタを使用するでしょうから、呼ぶ前と戻ってきた後では各レジスタの値は違っているかも知れません。

レジスタを式の途中結果や引数/返値の受け渡しのみに使用するのであればあまり問題はありません (式の評価の途中で関数を呼んだりするときは注意が必要)。しかし、効率よいコードのためにはよく参照される使われる値をできるだけ長くレジスタに置いて主記憶アクセスを避ける必要があります。その場合、それらのレジスタの値が手続き呼出しによって変わってしまうのは不都合です。これに対処するやり方としては、次の 2 つの方法があります。

- **呼び側での保存** (caller-save) — 手続きを呼ぶ側で、内容を壊されては困るレジスタの内容を保存してから呼び、返って来たらその内容を復元する。
- **呼ばれ側での保存** (callee-save) — 呼ばれた手続きの側で、呼ばれた直後に自分が内容を壊すレジスタを保存し、戻る直前に復元する。

どちらにも固有の利点と欠点があります。前者は、実際に壊されると困るレジスタのみを保存できますが、手続き呼出しが多数あると保存/復元用コードが多量に生成されます。後者は、実際には使っていないレジスタを保存してしまうかもしれませんが、保存/復元のコードは手続きの入口と出口に 1 箇所ずつで済みます。いずれにせよ、1 つの言語処理系ではどの方法を採用し、具体的にどのレジスタを退避回復するかを統一する必要があります。上記の特徴を考慮して、数個のレジスタは呼び側での保存、他の数個は呼ばれ側での保存とすることもあります。

11.3 コード生成

11.3.1 x86-64 CPU のデータサイズとレジスタ

以下では実際の CPU のコード生成を (アセンブラ出力により) 行ってみます。その場合、命令語の細かい仕様はアセンブラに任せればよいのですが、CPU がどのようなデータサイズを扱い、どのようなレジスタを持ち、どのような命令を持っているかは知っておく必要があります。

ここでは広く普及している x86-64 命令セットアーキテクチャについて簡単に説明します。厳密にはこのアーキテクチャでも Intel と AMD で違うところがあるのですが、主に OS 関係の部分なので、ここで説明する範囲では違いはありません。

図 11.7 に x86-64 CPU が扱うデータのサイズを示します。このアーキテクチャは 8 ビットの 8086 から順次進化 (無理矢理拡張?) してきたので、名前にもその名残りがあります。バイトは普通ですが、その倍の 16 ビットが「ワード」で、普段整数に使っている 32 ビットは「ロングワード」になります。

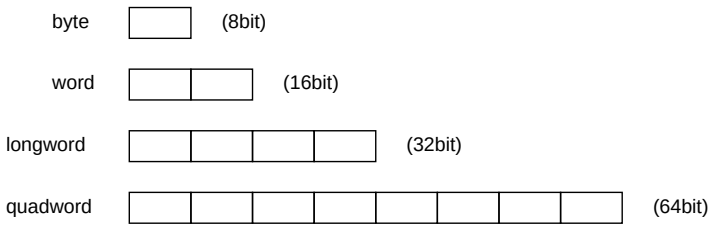

図 11.7: x86-64 CPU が扱うデータサイズ

そして 64 ビットアーキテクチャなので、アドレスのビット数が 64 ビットですが、これは「クワド (4 倍) ワード」と呼ばれます。

そして、これらのデータを扱うためのレジスタが図 11.8 のように構成されています (浮動小数点用のレジスタは略)。これも歴史的経緯のためたいへんやっかいです。もともと 8086 は特定用途向けの名前のついた少数のレジスタを持っていたのですが、レジスタが多い方が性能的に有利なのでどんどんレジスタを増やし、なおかつ上記のデータ幅も増やしたことによっています。

ともあれ、一番多く使うアキュムレータは昔は a レジスタ (8 ビット) と ax レジスタ (16) ビットでしたが、ax の下半分の 8 ビットが a レジスタと一緒でした。そして 32 ビット、64 ビットと増やしたときに上にビットを追加して eax と rax ができました。今回主に使うのは 32 ビットと 64 ビットですから、この 2 つを使います。

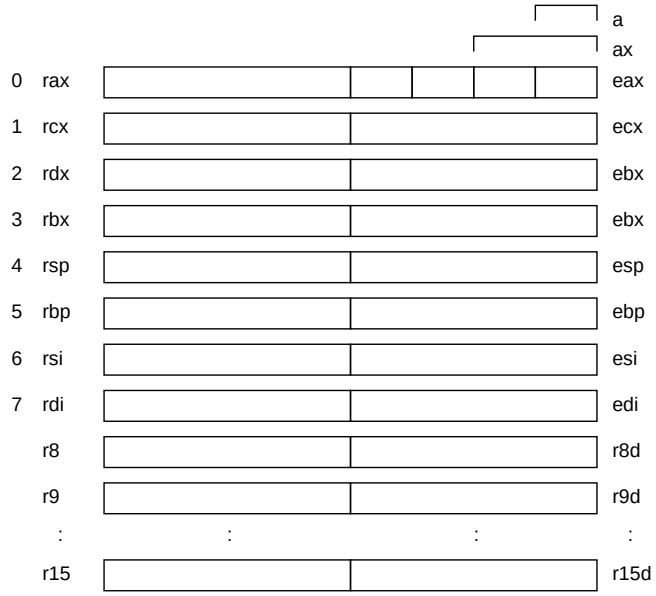

図 11.8: x86-64 CPU が持つレジスタ群

図にあるように、残りのレジスタも個々の役割と名前を持っていましたが、今では特定目的に使うのは rsp と rbp くらいで、あとは「同等の汎用目的レジスタが沢山ある」ものと思ってよいです。x86-64 になったときにレジスタを 8 個増やしたので、これらはもう固有の名前をつけずに r8-r15 となっています。すべて、32 ビットの名前 (右側) と 64 ビットの名前 (左側) を持つのに注意。機械語の上ではレジスタは番号で表すので、その番号も付記してあります。

特殊目的のレジスタが 2 つと書きましたが、そのうち rsp についてはハードウェア命令と関係があります。命令「pushq レジスタ」は、rsp を 8 減らしてから、指定したレジスタの内容 (8 バイト) を rsp の指している番地以下 8 バイトに書き込みます。「popq レジスタ」はその逆です。これにより、スタックにレジスタ内容を退避/回復できます。なお、スタックに積むときに rsp を「減らす」ということは、x86-68 ではスタックは上向き (低い番地の向き) に延びるということになります。

また、call(手続き呼び出し) 命令も同様に、現在の命令ポインタを戻り番地としてプッシュしてから指定宛先にジャンプし、ret(手続きからの戻り) 命令は戻り番地をポップして命令ポインタに戻すことで呼んだ箇所に戻るようになっています。

rbp については、先に説明したフレームポインタとして使用します。これについては次節で説明しましょう。

11.3.2 呼び出し規約

プログラムを実装するときに手続き呼び出しは不可欠ですが、ハードウェア命令として提供されている call/ret では引数や返値には感知していません。そこで、呼ぶ側と呼ばれる側でどのようにして引数や値を受け渡すかを決めてそれに従いコードを生成することになります。この約束ごとのことを**呼び出し規約** (calling convention) と呼びます。

ライブラリの手続きとも整合が取れている必要があるため、呼び出し規約は 1 つの OS の中で統一されていることが普通です (OS のライブラリを呼ばない閉じた言語処理系ではこれと別でもよい)。

ここでは x86-64 が動く Unix 系 OS の規約を説明します。上で説明したように、もともとは手続きの引数はスタックで渡すのが通常でしたが、レジスタが多数ある CPU ではメモリ (スタックは結局メモリの一部です) に格納するよりレジスタで渡す方が高速なため、先頭のいくつかの引数はレジスタで渡すようになりました。

具体的には先頭から 6 個の引数は順に`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` に載せられて渡されます。それらより多い引数はスタックに置かれます (実際には引数がそんなに多いことはあまりないですが)。返値については`%rax`(128 ビット値のときは加えて`%rdx` も使う) に置かれて返されます。[3]

レジスタの退避回復については、`%rsp`, `%rbp`, `%rbx`, `%r12`-`%r15` は呼ばれ側保存なので (最初の 2 つはスタックとベースポンタですが) 使う場合には退避回復が必要です。あとのレジスタは値を変更して構いません。

では練習として、「2 つの整数を受け取りその和を返す」手続きを作り、C 言語から呼び出してみましょう。まず C 言語側を示します。

```
#include <stdio.h>
int sub(int a, int b);
int main(void) {
  printf("%d\n", sub(3, 9)); return 0;
}
```

簡単ですね。これと一緒に使うアセンブリ言語で書いた `sub` を示します。

```
        .text
        .globl  sub
        .type   sub, @function
sub:    pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, %eax
        addl    %esi, %eax
        leave
        ret
```

最初の 3 行はアセンブリ言語への指示で「実行コード (テキスト) セグメントであること」「`sub` は外部参照される記号であり」「種別は関数であること」を示しています。これらの情報は翻訳後のファイルに含まれてリンカに渡されます。

[3] いずれも 64 ビットレジスタの名前を挙げましたが、32 ビットの値では 32 ビット側だけを使います。以下同様。

その後の sub:からがコードです。まず%rbp をスタックに積み、その場所は%rsp が指しているのでそれを%rbp にコピーします。これで動的チェインができました。次に%rsp を動かしてローカル変数の場所を確保するのが通常ですが、ここでは全部レジスタでやっていてスタック上の場所は使わないので何もしません。

次の 2 命令が sub の本体で、1 番目のパラメタは%edi(整数なので 32 ビット) に入っているので、それを%eax にコピーし、次の命令で%esi に入っている 2 番目のパラメタを加算します。これでぶじ、返値を入れるべきレジスタ%eax に和が入りました。

あとは戻りですが、%rsp は変更しなかったので、すぐに leave 命令 (スタック位置から%rbp をポップ) と ret 命令 (スタック位置から戻り番地をポップしてジャンプ) を実行します。

では動かしてみましょう。

```
% gcc sam_b1.c sam_b1sub.s
% ./a.out
12
```

もう少し込み入った、配列アクセスとジャンプ・条件ジャンプのある例を示しましょう。今度は sub は配列内の値の最大値を求めます。

```
#include <stdio.h>
int sub(int n, int a[]);
int b[] = { 5, 1, 6, 8, 2, 7, 4, 3 };
int main(void) {
  printf("%d\n", sub(8, b)); return 0;
}
```

ジャンプ命令は jmp、条件ジャンプ命令は演算命令か比較命令 (cmpl 等) の直後でのみ利用でき、jl(より小)、jle(以下)、jg(より大)、jge(以上)、je(等しい)、jne(等しくない) があります。

「0(%rsi)」は前にやったようにレジスタの指している場所から 0 ずれた位置 (ということはレジスタの指している場所) をアクセスするので、ここでは 2 番目のパラメタで渡された配列の先頭要素を取り出します。そして、「0(%rsi,%rdi,4)」というのは、上記位置からさらに%rdi に添字が入っているものとしてその添字に対応する位置をアクセスします (4 は 1 つの要素が 4 バイトであることを表す)。問題はこの場合 64 ビットの%rdi を使う必要があることで、そのため%edi を演算したあと (dec は 1 減らす演算)、cltq(convert long to quad) 命令で 32 ビットを 64 ビットに拡張しています。

```
        .text
        .globl  sub
        .type   sub, @function
sub:    pushq   %rbp
        movq    %rsp, %rbp
        movl    0(%rsi), %eax
.L1:    dec     %edi
        cltq
        jl      .L2
        movl    0(%rsi,%rdi,4), %edx
        cmpl    %edx, %eax
        jge      .L1
        movl    %edx, %eax
        jmp     .L1
```

```
.L2:    leave
        ret
```

プログラムの構造としては、まず%eax に配列の先頭要素を入れ、それから配列の各要素を順に取り出して%eax とくらべ、大きいようならその値を%eax にコピーします。順に取り出すには、%edi に要素数が入っているので、それを 1 ずつ減らし、負なら終わるというループを使っています。

実行のようすを示します。あまり面白くないですが、要素数が 100 万でもちゃんと動きます。

```
% gcc sam_b2.c sam_b2sub.s
% ./a.out
8
```

演習 1 上の 2 つの例題を打ち込んで動かせ。動いたら次のことをやってみなさい。

a. 1 番目の例題で、演算を足し算以外のものにして動かしてみよ。
b. 2 番目の例題で、演算を「配列要素の合計」にして動かしてみよ。
c. 2 番目の例題を改造して「配列を全部クリアする」「配列の要素を全部 1 つずつ前に動かす (先頭にあったものは末尾に移す)」などを作ってみよ。
d. C 言語で同じ処理を書いたものと実行時間を比較してみよ。C 言語側で最適化 (-O4) をオプションを指定した場合としない場合で違うので両方比べること。
e. gcc では「-S」オプションでコンパイルして出力されたアセンブリ言語コードを残すようになっている。これを用いて、C コンパイラの出力コートと手で書いたコードの比較をおこなってみよ。C 言語側で最適化 (-O4) をオプションを指定した場合としない場合で違うので両方比べること。

11.4 C から呼べる関数コードを生成する処理系

それではいよいよ、実際に x86-64 のコードを生成する処理系を作ってみます。といっても、型検査までは前回と同様におこなうので新しいところは生成部分だけです。ただ、言語の構文を少し変えていますので、その差分だけ見ていきます。まず SableCC の記述ファイルから。

```
Package samb3;   ←パッケージ名が違う
(途中略)
Productions
  prog = {main} int ident lpar dcls rpar stat  ←関数ぽい形
       ;
  dcls = {lst} dcls semi dcl      ←パラメタ部
       | {one} dcl
       ;
  dcl  = {idcl} int ident
       | {adcl} int ident lsbr rsbr
       ;
  stlist = {stat} stlist stat
         | {empty}
         ;
  stat = {idcl}    int ident semi    ←以下は同じだが read/print は無し
(以下略)
```

コンパイラドライバは Executor の変わりに Generator にしました。

```
package samb3;
import samb3.parser.*;
import samb3.lexer.*;
import samb3.node.*;
import java.io.*;
import java.util.*;

public class SamB3 {
  public static void main(String[] args) throws Exception {
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
        1024)));
    Start tree = p.parse();
    Symtab st = new Symtab();
    HashMap<Node,Integer> vtbl = new HashMap<Node,Integer>();
    TypeChecker tck = new TypeChecker(st, vtbl); tree.apply(tck); st.show();
    if(Log.getError() > 0) { return; }
    Generator gen = new Generator(st, vtbl); tree.apply(gen);
  }
}
```

TypeChecker は同じですが、構文が変わったところとコード生成の準備に関して対応して少しだけ変更しています。まずメインの入口で記号表にダミー変数と関数名の変数を登録しています。後者は、この言語では関数が返す値は関数名と同名の変数に代入しておくという仕様にしたので、そのための場所に使います。前者は変数のオフセットが 0 のところは動的チェインの場所なので、そこをあけておくためです。あとは、関数のパラメタ部の宣言も後で使うのでそれぞれオフセットテーブルに登録するだけです。

```
package samb3;              ←パッケージが変更
import samb3.analysis.*;
import samb3.node.*;
import java.io.*;
import java.util.*;

class TypeChecker extends DepthFirstAdapter {
  (途中略)
  @Override
  public void inAMainProg(AMainProg node) {
    st.addDef("$dummy$", Symtab.ITYPE);
    st.addDef(node.getIdent().getText(), Symtab.ITYPE);
  }
  @Override
  public void outAIdclDcl(AIdclDcl node) {
    Symtab.Ent e = st.addDef(node.getIdent().getText(), Symtab.ITYPE);
    vtbl.put(node, e.pos);
  }
  @Override
```

```
  public void outAAdclDcl(AAdclDcl node) {
    Symtab.Ent e = st.addDef(node.getIdent().getText(), Symtab.ATYPE);
    vtbl.put(node, e.pos);
  }
  (以下略)
```

では Generator を呼んでいきましょう。内部でファイル「asm.s」に書き込む PrintStream を保持しておきます。そしてメインの入口では先に見て来たような定型の部分を出力します。メインの出口では終わりの部分を出力します。なお、スタックの所要量については最後まで生成しないとわからないので、冒頭では.STSIZE という記号で値を参照しておき、最後でその実際の値を定義しています。

```
package samb3;
import samb3.analysis.*;
import samb3.node.*;
import java.io.*;
import java.util.*;

class Generator extends DepthFirstAdapter {
  HashMap<Node,Integer> pos;
  Symtab st;
  PrintStream pr;
  static String preg[] = {"%rdi", "%rsi", "%rdx", "%rcx", "%r8", "%r9"};
  int pcnt = 0, lcnt = 0;
  public Generator(Symtab s, HashMap<Node,Integer> p) throws Exception {
    st = s; pos = p; pr = new PrintStream(new File("asm.s"));
  }
  @Override
  public void inAMainProg(AMainProg node) {
    String name = node.getIdent().getText();
    pr.printf(" .text\n");
    pr.printf(" .globl %s\n", name);
    pr.printf(" .type %s, @function\n", name);
    pr.printf("%s: pushq %%rbp\n", name);
    pr.printf(" movq %%rsp, %%rbp\n");
    pr.printf(" subq $.STSIZE, %%rsp\n");
  }
  @Override
  public void outAMainProg(AMainProg node) {
    pr.printf(" movl -8(%%rbp), %%eax\n");
    pr.printf(" addq $.STSIZE, %%rsp\n");
    pr.printf(" leave\n");
    pr.printf(" ret\n");
    pr.printf(".STSIZE = %d\n", st.getGsize()*8);
  }
```

次は関数の入口部分の変数宣言ですが、配列も整数もスタック上のオフセットの位置 (すべて%rbp より上なのでマイナスの値です) にパラメタとして渡されて来た値をを格納します。どのレジスタが何番目かは配列 preg に入れてあります (スタック渡しには対応していません)。

```
@Override
public void outAIdclDcl(AIdclDcl node) {
  pr.printf(" movq %s, -%d(%%rbp)\n", preg[pcnt++], pos.get(node)*8);
}
@Override
public void outAAdclDcl(AAdclDcl node) {
  pr.printf(" movq %s, -%d(%%rbp)\n", preg[pcnt++], pos.get(node)*8);
}
```

さて、ここから様々な構文のコード生成です。この処理系では、すべての式の値はスタック上の変数に格納されるようにしています。このため、変数でない式があるごとに、その式の値を入れるテンポラリ (temporary、作業変数) を生成し、setOut() でそのオフセットを上に返すようにしています。なので、代入の場合は式の値を取り出して左辺の変数 (そのオフセットは表 pos に入っていましたね) に格納すればすみます。配列代入の場合は、添字式も同様に取り出しますが、取り出すレジスタは%edx にして 64 ビットに変換してアクセスに使用します。

```
@Override
public void outAAssignStat(AAssignStat node) {
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getExpr()));
  pr.printf(" movl %%eax, -%d(%%rbp)\n", pos.get(node)*8);
}
@Override
public void outAAassignStat(AAassignStat node) {
  pr.printf(" movq -%d(%%rbp), %%rcx\n", pos.get(node)*8);
  pr.printf(" movl -%s(%%rbp), %%edx\n", getOut(node.getIdx()));
  pr.printf(" cltq\n");
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getExpr()));
  pr.printf(" movl %%eax, 0(%%rcx,%%rdx,4)\n");
}
@Override
```

いよいよ、制御構造の if と while です。いずれも、分岐先のラベルがいるのでそのラベル番号を変数 lbl に入れてカウンタを進めます。先頭でこれをやるのは、処理の途中で別の (条件や本体の) コードを生成するとそこでもラベルを生成するため番号がごちゃまぜにならないようにしたものです。if はまず条件式を持って来てその真偽 (0 でなければ真) に基づき本体を迂回するラベルにジャンプします。While もよく似ていますが、先頭へのジャンプと迂回のジャンプで 2 つラベルが必要なところが主な違いです。

いずれもジャンプ命令の後、本体を生成し、最後のラベルを生成します。このように子ノードの処理を呼び出すタイミングを制御する必要があるため、case... のメソッドをオーバライドしています。

```
public void caseAIfStat(AIfStat node) {
  int lbl = lcnt; lcnt += 1;
  node.getExpr().apply(this);
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getExpr()));
  pr.printf(" testl %%eax,%%eax\n");
  pr.printf(" je .L%d\n", lbl);
  node.getStat().apply(this);
  pr.printf(".L%d:\n", lbl);
```

```
}
@Override
public void caseAWhileStat(AWhileStat node) {
  int lbl = lcnt; lcnt += 2;
  pr.printf(".L%d:\n", lbl);
  node.getExpr().apply(this);
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getExpr()));
  pr.printf(" testl %%eax,%%eax\n");
  pr.printf(" je .L%d\n", lbl+1);
  node.getStat().apply(this);
  pr.printf(" jmp .L%d\n", lbl);
  pr.printf(".L%d:\n", lbl+1);
}
@Override
```

条件式では、2 つの値を持って来て cmpl 命令で比較して正否により 0 または 1 を%eax に入れ、その値を最後にテンポラリに格納します。このため、格納する前にテンポラリを記号表に追加登録し、そのオフセットを使用しています。また、最後にそのオフセットを setOut() で登録して親ノードに渡します。

```
public void outAGtExpr(AGtExpr node) {
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getRight()));
  pr.printf(" cmpl -%s(%%rbp), %%eax\n", getOut(node.getLeft()));
  pr.printf(" jg .L%d\n", lcnt);
  pr.printf(" mov $1, %%eax\n");
  pr.printf(" jmp .L%d\n", lcnt+1);
  pr.printf(".L%d: movl $0, %%eax\n", lcnt++);
  Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
  pr.printf(".L%d: movl %%eax, -%d(%%rbp)\n", lcnt++, f.pos*8);
  setOut(node, new Integer(f.pos*8));
}
@Override
public void outALtExpr(ALtExpr node) {
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getRight()));
  pr.printf(" cmpl -%s(%%rbp), %%eax\n", getOut(node.getLeft()));
  pr.printf(" jl .L%d\n", lcnt);
  pr.printf(" mov $1, %%eax\n");
  pr.printf(" jmp .L%d\n", lcnt+1);
  pr.printf(".L%d: movl $0, %%eax\n", lcnt++);
  Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
  pr.printf(".L%d: movl %%eax, -%d(%%rbp)\n", lcnt++, f.pos*8);
  setOut(node, new Integer(f.pos*8));
}
```

ここから先は演算命令なので、条件分岐がないぶん、先の条件演算子よりは簡単です。いずれも値を格納するためのテンポラリを割り当てます。構文上の必要から生じている単一規則では、テンポラリのオフセットを上にコピーする作業だけおこないます。

```
    @Override
    public void outAOneExpr(AOneExpr node) { setOut(node, getOut(node.getNexp())); }
    @Override
    public void outAAddNexp(AAddNexp node) {
      pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getNexp()));
      pr.printf(" addl -%s(%%rbp), %%eax\n", getOut(node.getTerm()));
      Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
      pr.printf(" movl %%eax, -%d(%%rbp)\n", f.pos*8);
      setOut(node, new Integer(f.pos*8));
    }
    @Override
    public void outASubNexp(ASubNexp node) {
      pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getNexp()));
      pr.printf(" subl -%s(%%rbp), %%eax\n", getOut(node.getTerm()));
      Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
      pr.printf(" movl %%eax, -%d(%%rbp)\n", f.pos*8);
      setOut(node, new Integer(f.pos*8));
    }
    @Override
    public void outAOneNexp(AOneNexp node) { setOut(node, getOut(node.getTerm())); }
    @Override
    public void outAMulTerm(AMulTerm node) {
      pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getTerm()));
      pr.printf(" imull -%s(%%rbp), %%eax\n", getOut(node.getFact()));
      Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
      pr.printf(" movl %%eax, -%d(%%rbp)\n", f.pos*8);
      setOut(node, new Integer(f.pos*8));
    }
    @Override
    public void outADivTerm(ADivTerm node) {
      pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getFact()));
      pr.printf(" cltd\n");
      pr.printf(" idivl -%s(%%rbp)\n", getOut(node.getTerm()));
      Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
      pr.printf(" movl %%eax, -%d(%%rbp)\n", f.pos*8);
      setOut(node, new Integer(f.pos*8));
    }
    @Override
    public void outAOneTerm(AOneTerm node) { setOut(node, getOut(node.getFact())); }
    @Override
```

一番最後の因子のところですが、整数はその整数を割り当てたテンポラリに格納します。変数は新たな変数を使わなくても、もとの変数のオフセットをそのまま使えば済みます。配列アクセスは配列代入と類似ですが、こちらは取り出した値をテンポラリに格納する必要があります。

```
    public void outAIconstFact(AIconstFact node) {
      Symtab.Ent e = st.addDef(".t" + pcnt++, Symtab.ITYPE);
```

```
    pr.printf(" movl $%s, %%eax\n", node.getIconst().getText());
    pr.printf(" movl %%eax, -%d(%%rbp)\n", e.pos*8);
    setOut(node, new Integer(e.pos*8));
  }
  @Override
  public void outAIdentFact(AIdentFact node) {
    setOut(node, new Integer(pos.get(node)*8));
  }
  @Override
  public void outAArefFact(AArefFact node) {
    pr.printf(" movq -%d(%%rbp), %%rdx\n", pos.get(node)*8);
    pr.printf(" movl -%s(%%rbp), %%ecx\n", getOut(node.getExpr()));
    pr.printf(" cltq\n");
    pr.printf(" movl 0(%%rdx,%%rcx,4), %%eax\n");
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    pr.printf(" movl %%eax, -%d(%%rbp)\n", f.pos*8);
    setOut(node, new Integer(f.pos*8));
  }
  @Override
  public void outAOneFact(AOneFact node) { setOut(node, getOut(node.getExpr())); }
}
```

では、小さな言語で書いた「バブルソート」のプログラムを示します。C とまったく一緒に見えますがちょっとだけ文法が違うのに注意 (どこでしょう?)。

```
int sub(int n; int a[]) {
  int remain; remain = 1;
  while(remain) {
    remain = 0; int i; i = 1;
    while(i < n) {
      if(a[i-1]>a[i]) { int z; z=a[i-1]; a[i-1]=a[i]; a[i]=z; remain=1; }
      i = i + 1; } } }
```

これを呼び出す C 言語側はたとえばこんな感じです。

```
#include <stdio.h>
int sub(int n, int a[]);
int main(void) {
  int a[] = { 7, 1, 6, 3, 2, 4, 8 };
  sub(7, a);
  for(int i = 0; i < 7; ++i) { printf("%d\n", a[i]); }
}
```

演習 2 「小さな言語」処理系を動かし、上の例題を翻訳して `asm.s` を生成し、`gcc main.c asm.s` で C 言語コードと結合して動かしてみなさい。動いたら次のことをやってみなさい。

a. もっと長い配列を渡すようにして (配列には逆順に並んだ値を入れておくとよい)、所要時間を計測してみなさい。C 言語で同じプログラムを (文法は少し修正必要) コンパイルして動かし、CPU 消費量を比較しなさい。`-O` で最適化してみるとさらによい。CPU 消費は「`time ./a.out`」のように `time` コマンドを使うことで計測できる。

b. 上記において生成された `asm.s` を見て「効率が悪い原因」を検討してみなさい。手で `asm.s` を修正して速度の向上を試み、実際にどれくらい向上したか計測してみなさい。

c. 「小さな言語」でもっと別のプログラムを作って動かしてみなさい。性能計測もできるとなおよい。

11.5 課題 11A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (`y-kuno@uec.ac.jp`) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題 # 11」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 強い型の言語と弱い型の言語のどちらが好みですか。またそれはなぜ。

  Q2. 記号表と型検査の実装について学んでみて、どのように思いましたか。

  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

＃12　コード解析と最適化

12.1　最適化の原理と分類

最適化 (optimization) とは、コードにさまざまな変形や工夫を施すことで、性能を向上させる作業です。すなわち、その名前とは裏腹に、コードの実行性能を「改良する」わけです。名前通りであれば「最適」にすることになりますが、実際に「最適」にすることは極めて困難ですし、また CPU やデータの状況により何が「最適」となるかは変化してしまいます。

ここで最適化に関して重要な原則を挙げておきます。

> **原則:** 最適化のための変形により、コードが正しく動作しなくなってはいけない。

たとえば次のようなコードがあったとします (本来は中間コードでやりますが、見やすさのため以下ではソースコードレベルでの説明も多く出て来ます)。

```
while(条件) {
  x = ... /* 複雑な計算 */
  ...
}
```

ここで、x の値はループの周回があっても一定であるものとします。そうすると、ループは何万回も実行されるかもしれないので、x の計算をループの外に出すことで実行時間を削減したくなります。

```
x = ... /* 複雑な計算 */
while(条件) {
  ...
}
```

このような変形をしても大丈夫でしょうか? 一見良さそうですが、実はまずい可能性があります。というのは、x の計算は元のコードでは while の「条件」が満たされた時だけ実行されます。もしかしたらその計算は、「条件」が満たされていないとエラーになるかも知れません。そうなると、元は正しく動いていたプログラムがエラーを出すようになってしまいます。

ですから、最適化に際して行なうべき正しい変形は次のようになります。

```
if(条件) {
  x = ... /* 複雑な計算 */
  while(条件) {
    ...
  }
}
```

面倒だと思うかも知れませんが、「コードが正しく動作しなくなってはいけない」を守るというのはこういうことなのです。

では次に、最適化によって具体的にどのようにして実行速度を高めることが可能なのでしょうか。基本的な原理としては次のものが挙げられます。

(a) 実行しなくてもよい命令列を発見し、取り除く。

(b) 複数回実行されるがその間で結果が変わらない命令列を発見し、1 回の実行ですませるように変更する。

(c) 複数回実行される命令列を、より少ない実行回数ですませる。

(d) ある命令列を、それと同じ結果をもたらすより高速な命令列に取り替える。

(a) と (b) は、言い換えれば静的/動的に重複した計算を発見して削除することを意味します。先の while 文の例はこのうちの (b) に対応します。また、ループの実行回数が固定回でかつ小さいなら、ループ本体のコードをその回数だけコピー (展開) してしまえば、ループを制御する命令は不要になります。または、回数が多い場合でも、本体を N 回展開することで、周回数を $\frac{1}{N}$ 倍にできますから、ループ制御命令の実行回数を減らせます。これは (c) に相当します。(d) は、たとえば乗算命令よりは加算命令の方が高速なのが普通ですから、2 倍するときは乗算の代わりに 2 回足す命令を使う、などの場合が相当します。

ここまででも分かるように、最適化には極めて多様な手法があり、それだけで本が何冊も書けるほどです。ここでは、ごく基本的な概念に絞って、できるだけ具体例を挙げる形で説明します。それでも例題として実行できる部分はごく一部になりますが、ご容赦ください。また、ループ最適化は付録としますので、興味があれば読んでください。

12.2 中間コード生成と制御フロー解析

12.2.1 中間コードと 4 つ組

標準的なコンパイラの構成においては、抽象構文木から**中間コード** (IR — intermediate representation) を生成し、最適化を行なった後、目的コードへの変換を行なうことは既に学びました。そのようにする目的は、抽象度の高い抽象構文木や、ターゲット CPU に強く依存する目的コードでは、最適化が行ないにくいからです。

中間コードにもさまざなな形式のものがありますが、最適化に適した形式としては**レジスタ転送言語** (RTL — register transfer language) や **4 つ組** (quadleple) と呼ばれるものがよく使われます。

RTL は GCC(Gnu C Compiler) で採用されていることからよく知られています。RTL はその名前通り、多数あるレジスタの間で値を転送しながら計算していくというモデルであり、本質的には 4 つ組と似ていますが、たとえば演算に伴い条件コードビットが辺かすることなど、CPU の様々な機能を併せて扱うことができます。

一方、4 つ組は典型的には「命令, 代入先, 被演算子 1, 被演算子 2」の 4 つの情報が 1 つの命令を構成することからこの名前があるもので、たとえば次のように見慣れた代入文の書き方で記述することができます。

```
  t2 = t1 + 1
  t3 = t4[t2]
  t5 = 0
  if t3 > t5 then L1
  ...
L1:
  ...
```

ただし、ここに現れる名前はいずれも**テンポラリ** (temporary) ないし作業レジスタであり、実際にはコード生成時にメモリ上の位置またはいずれかの CPU レジスタに割り付けられることで動作します。また、個々の命令はアセンブリ言語レベルのものであり、複雑な計算式は存在しません。条件分

岐命令やラベルがあることからも、アセンブリ言語に近い水準であると言えます。以下の例題では、主に 4 つ組を説明に使用します (このほか、ソースコードで説明する部分もあります)。

具体例がないと分かりにくいと思うので、ここではいつもの言語で書いた (文法は前回と同じなので省略) 図 12.1 のプログラムの中間コードを図 12.2 に掲載しておきます。

```
int sub(int n; int a[]) {
  int remain; remain = 1;
  while(remain) {
    remain = 0; int i; i = 1;
    while(i < n) {
      if(a[i]>a[i-1]) { int z; z=a[i-1]; a[i-1]=a[i]; a[i]=z; remain=1; }
      i = i + 1; } } }
```

図 12.1: 小さい言語で書いたバブルソート

```
L1000: // B0
  t2 = %rdi
  t3 = %rsi
  t7 = 1
  t4 = t7
L0: // B1
  ifz t4 then L1
L1001: // B2
  t8 = 0
  t4 = t8
  t9 = 1
  t5 = t9
L2: // B3
  if t5 < t2 then L4
L1002: // B4
  t10 = 0
  jmp L5
L4: // B5
  t10 = 1
L5: // B6
  ifz t10 then L3
L1004: // B7
  t11 = t3[t5]
  t12 = 1
  t13 = t5 - t12
  t14 = t3[t13]
  if t11 > t14 then L7
L1005: // B8
  t15 = 0
  jmp L8
L7: // B9
  t15 = 1
L8: // B10
  ifnz t15 then L6
L1007: // B11
  t16 = 1
  t17 = t5 - t16
  t18 = t3[t17]
  t6 = t18
  t19 = 1
  t20 = t5 - t19
  t21 = t3[t5]
  t3[t20] = t21
  t3[t5] = t6
  t22 = 1
  t4 = t22
L6: // B12
  t23 = 1
  t24 = t5 + t23
  t5 = t24
  jmp L2
L3: // B13
  jmp L0
L1: // B14
```

図 12.2: 小さな言語で書いたバブルソートの中間コード

12.2.2 基本ブロックとフローグラフ

最適化の観点からは、IR 命令の列は**基本ブロック** (basic block) の集まりとして取り扱います。基本ブロックとは、途中からの飛び出しや途中への飛び込みが無いような IR 命令の列、言い替えれば「先頭から 1 直線に実行される列」を言います。

IR 命令の中で、飛び込みはラベルの箇所にしか起こりません。逆に言えば、ラベルにはよそから飛んで来るので、ラベルがあるとそこから先が 1 つのブロックになります。そして、分岐命令か条件分岐命令があると、そこからは飛び出すことになるので、これらの命令があるとブロックが終わります (このほかに、次のラベルが現れた場合にもブロックが終わります)。

整理すると、ブロックの先頭は手続きの先頭命令、ラベル、分岐命令の直後の命令のいずれかであり、ブロックの最後の命令はラベルの直前の命令と分岐命令のいずれかです。以下では扱いの統一のため、ラベルで始まらないブロックの先頭にもラベルを付加し、ブロックを同定するのに先頭のラベルを用います。図 12.2 のコードでは、ラベルの後にブロック番号を記載しました。

基本ブロックの重要な性質は、そこに含まれる命令列は必ず最初から最後まで順に実行される、ということです。したがって、例えばブロック中で同じ式を複数回計算していて、なおかつその間にその式の値を変更する可能性を持つ命令がないなら、最初に計算した値を保存しておいて繰り返し利用してもよいことになります。

このように、各ブロック内に範囲を限って最適化を行うことを**局所最適化** (local optimization) と呼びます。これに対し、1 つの手続き内でブロックをまたがって最適化を行うことを**広域最適化** (global

optimization) と呼びます。さらに、他の手続きを呼び出した先にまたがって行なう最適化である**手続き間最適化** (interprocedural optimization) までありますが、それをやるとその手続きは単独で呼べなくなるので、必ずしも普及していません (以下でも取り上げません)。以下ではまず、局所最適化の例を挙げ、そのあと広域最適化に必要なことがらを取り上げていきます。

あるブロックを実行した後実行が進む先は、そのブロックの末尾が無条件分岐命令ならばその飛び先、条件分岐ならば飛び先と後続ブロックの双方、それ以外では後続ブロックのみとなります。1 つの手続きについて、ブロックを節、行き先関係を矢線で表した有向グラフをその手続きの**フローグラフ** (flow graph) と呼びます。図 12.3 に、先の中間コードのフローグラフを示します。広域最適化を行う場合にはフローグラフを構成し、その上で各種解析と最適化を行います。

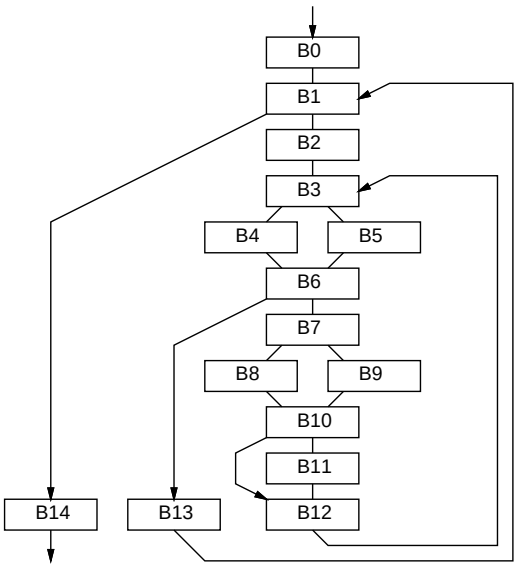

図 12.3: バブルソートのコードのフローグラフ

12.2.3 制御フロー解析

フローグラフを作成した後まず**制御フロー解析** (control flow analysis) を行うことが通例ですが、その主な目的はプログラム中のループを同定することです。これは、後で述べるように、ループ最適化によって多くの性能向上が見込めるためです。

今日の言語では、ループはソースコードの繰返し構文に対応しているので、フローグラフからループを再発見する代りに繰返し構文の情報を保持しておく方法もありますが、ここでは一般のフローグラフからのループを同定する方法を説明しておきます。まず、いくつかの用語を定義します。

定義 フローグラフ G に含まれる節 d が節 n を**支配する** (dominate) とは、グラフの出発点から n に至る全ての経路が d を通ることを言う。

定義 フローグラフ G に含まれる辺 $b \to h$ が**帰辺** (back edge) であるとは、h が b を支配する節である場合を言う。

定義 フローグラフ G に含まれる帰辺 $b \to h$ に関する **自然ループ** (natural loop) とは、n から b へ h を通らずに行ける経路があるような節 n(b 自身を含む) と h を合せたものである。ここで h をループの**ヘッダ** (header) と呼ぶ。

図 12.3 を例に考えると、B1 は B2〜B14 をすべて支配し、B3 は B4〜B13 を支配します。支配とは「そこを通らなければ行けない」ということですね。この 2 つのブロックがループヘッダです。そ

して、B13 → B1 と B12 → B3 がそれぞれ帰辺になります (B1、B3 へ行くこれ以外の辺は支配された節から出ていません)。そして、B1〜B14、B3〜B13 がそれぞれ自然ループとなります。

ところで「自然な」ループがあるからには「自然でない」ループも存在します。具体的には、自然でないループとは入口が 2 つ以上あるようなループですが、今日の制御構文を使って作られたプログラムではそのようなものはできないので、自然ループのみを最適化の対象とするのが普通です。

自然ループは入口が 1 つ、出口が 1 つで、2 つの自然ループで共有される節がある場合には、片方が他方に完全に含まれるという性質があります。時として、複数の帰辺が 1 つの行き先を持つことがありますが、そうすると 2 つの自然ループでヘッダを共有してしまい、最適化のとき不便なので、適宜空のブロックを追加して、自然ループとヘッダが 1 対 1 で対応するようにします。さらに、ヘッダの直前に空のブロックを追加し、ループ外から入って来るときにはこを通過するようにします。これを**プリヘッダ**と呼び、ループ最適化のときループ内からコードを移して来るのに使います。

12.3 中間コード生成とブロックの構築

12.3.1 中間コードのデータ構造

それでは実際に中間コードを生成する部分を見ていきましょう。まず中間コードのためのクラス `IntCode` を見ていきますが、その中に入っているクラス `Inst` とそのサブクラス (個々の中間コード命を表す) は後回しにします。

```
package samc1;
import java.util.*;

public class IntCode {
  List<Block> code = new ArrayList<Block>();
  Block cb;
  int lblno = 1000;
  public IntCode() {
    cb = new Block("L1000", code.size()); code.add(cb);
  }
  public List<Block> getBlocks() { return code; }
  public void gLabel(int l) {
    if(cb.getTmp() && cb.getSize()==0) { cb.setLabel("L"+l); }
    else { cb = new Block("L"+l, code.size()); code.add(cb); }
  }
```

まず、中間コードはブロックの集まりです。変数 `cb` が現在コードを追加中の (最後の) ブロックです。クラス `Block` はすぐ後で読みます。ブロックの区切りのラベルは L1000 以降を使うことにして、その連番を生成するカウンタもインスタンス変数とします。コンストラクタでは最初の空ブロックを生成します。コード内のブロックは ID 番号 (0 からの連番) を持たせ、この番号は変数 `code` に入っているリストの内の位置と一致しています。`gLabel()` はラベルを生成しますが、現在のブロックがまだ空で、そのブロックにもともとのラベルがないなら、そのブロックのラベルを取り換えて終わります。そうでないならそのラベルを持つ新しいブロックを追加します。

以下のメソッドはすべて、中間コード命令を末尾に追加するものです。個々のクラスは後で読みます。

```
  public void gMovereg(int t1, String r) { cb.add(new Movereg(t1, r)); }
  public void gMoveint(int t1, int v) { cb.add(new Moveint(t1, v)); }
  public void gMove(int t1, int t2) { cb.add(new Move(t1, t2)); }
```

```
  public void gAdef(int t1, int t2, int t3) { cb.add(new Adef(t1, t2, t3)); }
  public void gAref(int t1, int t2, int t3) { cb.add(new Aref(t1, t2, t3)); }
  public void gAdd(int t1, int t2, int t3) { cb.add(new Add(t1, t2, t3)); }
  public void gSub(int t1, int t2, int t3) { cb.add(new Sub(t1, t2, t3)); }
  public void gMul(int t1, int t2, int t3) { cb.add(new Mul(t1, t2, t3)); }
  public void gDiv(int t1, int t2, int t3) { cb.add(new Div(t1, t2, t3)); }
  public void gJump(int l) { cb.add(new Jump(l)); eb(); }
  public void gIfz(int t1, int l) { cb.add(new Ifz(t1, l)); eb(); }
  public void gIfnz(int t1, int l) { cb.add(new Ifnz(t1, l)); eb(); }
  public void gIfne(int t1, int t2, int l) { cb.add(new Ifne(t1,t2,l)); eb(); }
  public void gIfeq(int t1, int t2, int l) { cb.add(new Ifeq(t1,t2,l)); eb(); }
  public void gIfgt(int t1, int t2, int l) { cb.add(new Ifgt(t1,t2,l)); eb(); }
  public void gIflt(int t1, int t2, int l) { cb.add(new Iflt(t1,t2,l)); eb(); }
  public void gIfge(int t1, int t2, int l) { cb.add(new Ifge(t1,t2,l)); eb(); }
  public void gIfle(int t1, int t2, int l) { cb.add(new Ifle(t1,t2,l)); eb(); }

  private void eb() { gLabel(++lblno); cb.setTmp(true); }
  public void show() { for(Block b: code) { b.show(); } }
  public void calcconnect() {
    Map<String,Block> map = new HashMap<String,Block>();
    for(Block b: code) { map.put(b.getLabel(), b); }
    for(int i = 0; i < code.size(); ++i) {
      Block b1 = code.get(i);
      for(int j: b1.getJdsts()) {
        if(j < 0 && i < code.size()-1) { b1.connect(code.get(i+1)); }
        if(j >= 0) { b1.connect(map.get("L"+j)); }
      }
    }
  }
  public void showconnect() { for(Block b: code) { b.showconnect(); } }
```

eb() は分岐命令の後に呼び出され、そこでブロックを切ります。その後はコードの内容を表示するメソッド、ブロック間の接続を計算するメソッド、そして接続を表示するメソッドです。

クラス Block は基本的にラベルを 1 つ持つ命令の並びですが、制御フロー解析のため相互の接続関係を保持できるようになっていて、またデータフロー解析のため、このブロックで定義されるテンポラリ、参照されるテンポラリの情報を返せます (後で説明します)。

```
  public static class Block {
    String label;
    List<Inst> body = new ArrayList<Inst>();
    Set<Integer> prev = new TreeSet<Integer>();
    Set<Integer> succ = new TreeSet<Integer>();
    Set<Integer> ref = null;
    Set<Integer> def = null;
    int bid = 0;
    boolean tmp = false;
    public Block(String l, int i) { label = l; bid = i; }
```

```
    public int getId() { return bid; }
    public String getLabel() { return label; }
    public void setLabel(String lbl) { label = lbl; }
    public boolean getTmp() { return tmp; }
    public void setTmp(boolean t) { tmp = t; }
    public int getSize() { return body.size(); }
    public void add(Inst i) { body.add(i); }
    public List<Inst> getBody() { return body; }
    public Set<Integer> getPrev() { return prev; }
    public Set<Integer> getSucc() { return succ; }
    public Set<Integer> getRef() { if(ref==null) { calcDR(); } return ref; }
    public Set<Integer> getDef() { if(def==null) { calcDR(); } return def; }
    private void calcDR() {
      def = new TreeSet<Integer>(); ref = new TreeSet<Integer>();
      for(Inst op: body) {
        for(Integer i: op.getRefs()) { if(!def.contains(i)) { ref.add(i); }}
        for(Integer i: op.getDefs()) { def.add(i); }
      }
    }
    public void connect(Block b1) { succ.add(b1.getId()); b1.prev.add(bid); }
    public int[] getJdsts() {
      if(body.size() == 0) { return new int[]{ -1 }; }
      Inst last = body.get(body.size()-1);
      int dst = last.getJdst();
      if(last instanceof Jump || dst < 0) { return new int[]{ dst }; }
      return new int[]{ dst, -1 };
    }
    public void show() {
      System.out.println(label + ": // B" + bid);
      for(Inst i: body) { System.out.println("  "+i); }
    }
    public void showconnect() {
      System.out.print("B" + bid);
      System.out.print(" prev(");
      for(int i: prev) { System.out.printf(" B%d", i); }
      System.out.print(" ) succ(");
      for(int i: succ) { System.out.printf(" B%d", i); }
      System.out.println(" )");
    }
  }
  // ここにクラス Inst とそのサブクラス
}
```

12.3.2 中間コード生成

それでは、実際に中間コードを生成する部分です。フロントエンド (意味解析まで) はこれまでと同じなので、SableCC の記述ファイル、Log.java、Symtab.java、TypeChecker.Java は省略します

(パッケージ名のみ修正)。以下に中間コード生成用クラスを示します。

```
package samc1;
import samc1.analysis.*;
import samc1.node.*;
import java.io.*;
import java.util.*;

class GenIntcode extends DepthFirstAdapter {
  HashMap<Node,Integer> pos;
  Symtab st;
  IntCode ic;
  int pcnt = 0, lcnt = 0;
  static String preg[] = {"%rdi", "%rsi", "%rdx", "%rcx", "%r8", "%r9"};
  private int ti(Object o) { return (Integer)o; }
  public GenIntcode(Symtab s, HashMap<Node,Integer> p, IntCode i) {
    st = s; pos = p; ic = i;
  }
```

インスタンス変数としてはこれまでのものに加え、IntCode のインスタンスを保持します。ti() というのは Object 型から整数にキャストするための下請けメソッドです。

まず、冒頭のパラメタ部では Movereg というのを生成して固定レジスタからテンポラリに値を移します。以降はすべてテンポラリのみで値をすべて扱います。式などはすべて意味スタックにテンポラリ番号を入れて戻るようにしています。そして各命令の生成は IntCode のメソッド「g なんとか」を呼べばよいように作ってあったのでした。

```
  @Override
  public void outAIdclDcl(AIdclDcl node) {
    ic.gMovereg(pos.get(node), preg[pcnt++]);
  }
  @Override
  public void outAAdclDcl(AAdclDcl node) {
    ic.gMovereg(pos.get(node), preg[pcnt++]);
  }
  @Override
  public void outAAssignStat(AAssignStat node) {
    ic.gMove(pos.get(node), ti(getOut(node.getExpr())));
  }
  @Override
  public void outAAassignStat(AAassignStat node) {
    ic.gAdef(pos.get(node), ti(getOut(node.getIdx())), ti(getOut(node.getExpr())));
  }
```

if 文や while 文は条件ジャンプとラベルが必要なので、ラベル番号を用意し、条件ジャンプ命令の行き先や生成ラベルで使用します。あと、比較演算も大小に応じて 0 か 1 を返すので、内部ではラベルと条件ジャンプを使って組み立てています。

```
  @Override
```

```
public void caseAIfStat(AIfStat node) {
  int lbl = lcnt; lcnt += 1;
  node.getExpr().apply(this);
  ic.gIfnz(ti(getOut(node.getExpr())), lbl);
  node.getStat().apply(this);
  ic.gLabel(lbl);
}
@Override
public void caseAWhileStat(AWhileStat node) {
  int lbl = lcnt; lcnt += 2;
  ic.gLabel(lbl);
  node.getExpr().apply(this);
  ic.gIfz(ti(getOut(node.getExpr())), lbl+1);
  node.getStat().apply(this);
  ic.gJump(lbl);
  ic.gLabel(lbl+1);
}
@Override
public void outAGtExpr(AGtExpr node) {
  Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
  ic.gIfgt(ti(getOut(node.getLeft())), ti(getOut(node.getRight())), lcnt);
  ic.gMoveint(f.pos, 0);
  ic.gJump(lcnt+1);
  ic.gLabel(lcnt++);
  ic.gMoveint(f.pos, 1);
  ic.gLabel(lcnt++);
  setOut(node, new Integer(f.pos));
}
@Override
public void outALtExpr(ALtExpr node) {
  Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
  ic.gIflt(ti(getOut(node.getLeft())), ti(getOut(node.getRight())), lcnt);
  ic.gMoveint(f.pos, 0);
  ic.gJump(lcnt+1);
  ic.gLabel(lcnt++);
  ic.gMoveint(f.pos, 1);
  ic.gLabel(lcnt++);
  setOut(node, new Integer(f.pos));
}
```

以下は普通の式の演算類なので、これまでと基本的に変わりません。ただ中間コードを生成するというだけです。

```
@Override
public void outAOneExpr(AOneExpr node) { setOut(node, getOut(node.getNexp())); }
@Override
public void outAAddNexp(AAddNexp node) {
```

```
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gAdd(f.pos, ti(getOut(node.getNexp())), ti(getOut(node.getTerm())));
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outASubNexp(ASubNexp node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gSub(f.pos, ti(getOut(node.getNexp())), ti(getOut(node.getTerm())));
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outAOneNexp(AOneNexp node) { setOut(node, getOut(node.getTerm())); }
  @Override
  public void outAMulTerm(AMulTerm node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gMul(f.pos, ti(getOut(node.getTerm())), ti(getOut(node.getFact())));
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outADivTerm(ADivTerm node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gDiv(f.pos, ti(getOut(node.getTerm())), ti(getOut(node.getFact())));
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outAOneTerm(AOneTerm node) { setOut(node, getOut(node.getFact())); }
  @Override
  public void outAIconstFact(AIconstFact node) {
    Symtab.Ent e = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gMoveint(e.pos, new Integer(node.getIconst().getText()));
    setOut(node, new Integer(e.pos));
  }
  @Override
  public void outAIdentFact(AIdentFact node) {
    setOut(node, new Integer(pos.get(node)));
  }
  @Override
  public void outAArefFact(AArefFact node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gAref(f.pos, pos.get(node), ti(getOut(node.getExpr())));
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outAOneFact(AOneFact node) { setOut(node, getOut(node.getExpr())); }
}
```

これが生成する中間コードについては、既に図 12.2 で示してあります。

12.3.3 コンパイラドライバ

今回はかなり色々な処理を試したり結果を表示したりしなかったりしたいので、コマンド引数で「java samc1/SamC1 ソース コマンド…」のようにしてコマンドを順番に実行するようなコンパイラドライバになりました。実は、図 12.2 の中間コードは「pi」コマンドで出力させたものです。なお、とりあえず、まだ出て来ていない部分はコメントアウトしたので、使う時はコメントを外してください。

```
  package samc1;
  import samc1.parser.*;
  import samc1.lexer.*;
  import samc1.node.*;
  import java.io.*;
  import java.util.*;

  public class SamC1 {
    public static void main(String[] args) throws Exception {
      if(args.length == 0) { showoptions(); return; }
      Parser p = new Parser(new Lexer(new PushbackReader(
        new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
          1024)));
      Start tree = p.parse();
      Symtab st = new Symtab();
      HashMap<Node,Integer> vtbl = new HashMap<Node,Integer>();
      TypeChecker tck = new TypeChecker(st, vtbl); tree.apply(tck);
      if(Log.getError() > 0) { return; }
      IntCode ic = new IntCode();
      GenIntcode gen = new GenIntcode(st, vtbl, ic); tree.apply(gen);
      ic.calcconnect();
//    Dataflow df = new Dataflow(ic.getBlocks());
      for(int i = 1; i < args.length; ++i) {
        if(args[i].equals("pt")) { st.show(); }
        else if(args[i].equals("pi")) { ic.show(); }
        else if(args[i].equals("pc")) { ic.showconnect(); }
//      else if(args[i].equals("vn")) { ValNumber.run(ic, false); }
//      else if(args[i].equals("pvn")) { ValNumber.run(ic, true); }
//      else if(args[i].equals("clio")) { df.calcLiveInOut(); }
//      else if(args[i].equals("plio")) { df.showLiveInOut(); }
        else { System.err.println("unknown command: "+args[i]); }
      }
    }
    private static void showoptions() {
      String[] msg = {
        "pt: print table",
        "pi: print intcode",
        "pc: print block-connection",
//      "vn: value numbering",
```

```
//      "pvn: value numvering with printout",
//      "clio: calculate LiveIn/LIveOut",
//      "plio: print LiveIn/LiveOut",
      };
      System.out.println("usage: java samc1/SamC1 <source> <option>...");
      for(String s: msg) { System.out.println("  " + s); }
    }
  }
```

では使用例として、ブロックの接続関係を表示させてみましょう (図 12.4)。

```
% java samc1/SamC1 test.min pc
B0 prev( ) succ( B1 )
B1 prev( B0 B13 ) succ( B2 B14 )
B2 prev( B1 ) succ( B3 )
B3 prev( B2 B12 ) succ( B4 B5 )
B4 prev( B3 ) succ( B6 )
B5 prev( B3 ) succ( B6 )
B6 prev( B4 B5 ) succ( B7 B13 )
B7 prev( B6 ) succ( B8 B9 )
B8 prev( B7 ) succ( B10 )
B9 prev( B7 ) succ( B10 )
B10 prev( B8 B9 ) succ( B11 B12 )
B11 prev( B10 ) succ( B12 )
B12 prev( B10 B11 ) succ( B3 )
B13 prev( B6 ) succ( B1 )
B14 prev( B1 ) succ( )
%
```

図 12.4: ブロックの接続関係表示

演習 1 自分でも同じものを動かしてみよ。動いたら、次のことをやってみよ。

a. 別のプログラムを作成して中間コードを表示してみよ。そのあと、手でフローグラフ (図 12.3 のようなもの) を描き、bc コマンドで表示させたものと一致していることを確認せよ。
b. 今回のコードでは自然ループの検出機能は実装していない。実装してみよ。
c. 上記に加え、ループごとに 1 つループヘッダのブロックがあるように修正してみよ。

12.4 局所最適化

12.4.1 局所最適化の考え方と手法

前述のように、局所最適化とは基本ブロックの中だけで実行できるような最適化を言います。具体的に考えるため、ソースコードで「`int z; z=a[i-1]; a[i-1]=a[i]; a[i]=z; remain=1;`」に対応する部分を図 12.2 のコードから抜き出したものを図 12.5 に示します。

よく見ると色々な無駄が見つかると思います。定数「`1`」を何回も別のテンポラリに入れていますし、添字「`i-1`」の計算も 2 回行なっていますし、代入するのにわざわざ別のテンポラリに計算して、それを改めてコピーしています。このような無駄は、普通にコード生成を行なうとどうしても生じて

```
L1007: // B11
  t16 = 1
  t17 = t5 - t16
  t18 = t3[t17]
  t6 = t18
  t19 = 1
  t20 = t5 - t19
  t21 = t3[t5]
  t3[t20] = t21
  t3[t5] = t6
  t22 = 1
  t4 = t22
```

図 12.5: 比較交換部分の中間コード (原型)

しまいます (頑張れば減らせますが、速度が問題になるようなコンパイラであれば、どのみち最適化で対処するのでほっていあるというべきかも)。

これらに対処するための局所最適化の手法として、次のようなものがあります。

- **定数伝播** (constant propagation) — 定数どうしの演算式があったら、それをコンパイル時に計算してしまい、定数で置き換える。そんな無駄なコードは書かないと思うかも知れませんが、たとえば `SIZE` を 100 と定義し、添字上限として `SIZE-1` と書く、みたいなのは結局定数計算になります。
- **コピー伝播** (copy propagation) — 値を次々に代入 (コピー) している場合、最初の値を最後の行き先に直接代入すれば途中のコピーが不要になります。
- **共通部分式の削減** (common sub-expression elimination, CSE) — 演算式やその一部で同じ計算が重複している場合、最初の計算結果を保存して利用することで 2 回目以降の計算を省く。
- **不要コードの削減** (usless code elimination) — コピー伝播や CSE の結果、後から使われないテンポラリへの代入があれば、その代入コード自体を削除できます。また、その代入のための式の計算も芋づる式に削除できることがあります。

そのほか、上とは毛色が違いますが、計算の効率化やコード削減を行なう次のものもあります。

- 演算の知識を用いた簡略化や高速化 — 2 倍する代わりに足し算というのを先に挙げましたが、「1 倍する」「0 を足す」などはそもそも演算が不要になります。これらは定数伝播の結果変数だったものの値が分かり、それにより適用できることがあります。
- **不到達コードの削減** (dead code elimination) — if の条件が常に成立/不成立などと分かれば (これも定数伝播の結果分かることがあります)、決して通らないコードが分かり、削除できます。これはブロック内の最適化というより、定数伝播のあと制御フロー解析を行なうことで実行できる最適化になります。

12.4.2 命令クラス群の機能と構造

本題に入る前に、そろそろ避けて通れないので、保留にしてきた命令を表すクラス群 (`IntCode.Inst` とそのサブクラス群) を見ておきます。基本的な機能は次の通りです。

- 最初に命令を作ると、文字列表現を用意する。また、その命令が**定義する** (define、代入する) テンポラリと**参照する** (refer) テンポラリの番号を記録しておきます。

- 値番号による最適化のため、オペランド (0/1/2 個) を与えて右辺の計算式を表す文字列を返すメソッドと、新たなテンポラリ番号 (0/1/2 個) を与えて自身と同じ種別の命令を返すメソッドを用意しています。

```
public static class Inst {
  int[] defs = {};
  int[] refs = {};
  String s;
  int jdst = -1, val;
  protected void sd(int... a) { defs = a; }
  protected void sr(int... a) { refs = a; }
  public int[] getDefs() { return defs; }
  public int[] getRefs() { return refs; }
  public int getJdst() { return jdst; }
  public String toString() { return s; }
  public int assDest() { return -1; }
  public int refNum() { return 0; }
  public String refExp() { return null; }
  public String refExp(String v1) { return null; }
  public String refExp(String v1, String v2) { return null; }
  public Inst renew() { return this; }
  public Inst renew(int t1) { return this; }
  public Inst renew(int t1, int t2) { return this; }
}
static class Movereg extends Inst {
  public Movereg(int t1, String reg) {
    s = String.format("t%d = %s", t1, reg); sd(t1);
  }
}
static class Move extends Inst {
  public Move(int t1, int t2) {
    s = String.format("t%d = t%d", t1, t2); sd(t1); sr(t2);
  }
  public int assDest() { return defs[0]; }
  public int refNum() { return 1; }
  public String refExp(String v1) { return v1; }
}
static class Moveint extends Inst {
  public Moveint(int t1, int v) {
    val = v; s = String.format("t%d = %d", t1, val); sd(t1);
  }
  public int assDest() { return defs[0]; }
  public int refNum() { return 0; }
  public String refExp() { return val+""; }
}
static class Adef extends Inst {
```

```
    public Adef(int t1, int t2, int t3) {
      s = String.format("t%d[t%d] = t%d", t1, t2, t3); sd(t1); sr(t2, t3);
    }
    public int refNum() { return 2; }
    public Inst renew(int t1, int t2) { return new Adef(defs[0], t1, t2); }
  }
  static class Aref extends Inst {
    public Aref(int t1, int t2, int t3) {
      s = String.format("t%d = t%d[t%d]", t1, t2, t3); sd(t1); sr(t2, t3);
    }
    public int assDest() { return defs[0]; }
    public int refNum() { return 2; }
    public String refExp(String v1, String v2) { return v1+"["+v2+"]"; }
    public Inst renew(int t1, int t2) { return new Aref(defs[0], t1, t2); }
  }
  static class Add extends Inst {
    public Add(int t1, int t2, int t3) {
      s = String.format("t%d = t%d + t%d", t1, t2, t3); sd(t1); sr(t2, t3);
    }
    public int assDest() { return defs[0]; }
    public int refNum() { return 2; }
    public String refExp(String v1, String v2) { return v1+"+"+v2; }
    public Inst renew(int t1, int t2) { return new Add(defs[0], t1, t2); }
  }
  static class Sub extends Inst {
    public Sub(int t1, int t2, int t3) {
      s = String.format("t%d = t%d - t%d", t1, t2, t3); sd(t1); sr(t2, t3);
    }
    public int assDest() { return defs[0]; }
    public int refNum() { return 2; }
    public String refExp(String v1, String v2) { return v1+"-"+v2; }
    public Inst renew(int t1, int t2) { return new Sub(defs[0], t1, t2); }
  }
  static class Mul extends Inst {
    public Mul(int t1, int t2, int t3) {
      s = String.format("t%d = t%d * t%d", t1, t2, t3); sd(t1); sr(t2, t3);
    }
    public int assDest() { return defs[0]; }
    public int refNum() { return 2; }
    public String refExp(String v1, String v2) { return v1+"*"+v2; }
    public Inst renew(int t1, int t2) { return new Mul(defs[0], t1, t2); }
  }
  static class Div extends Inst {
    public Div(int t1, int t2, int t3) {
      s = String.format("t%d = t%d / t%d", t1, t2, t3); sd(t1); sr(t2, t3);
    }
```

```
  public int assDest() { return defs[0]; }
  public int refNum() { return 2; }
  public String refExp(String v1, String v2) { return v1+"/"+v2; }
  public Inst renew(int t1, int t2) { return new Div(defs[0], t1, t2); }
}
static class Jump extends Inst {
  public Jump(int lbl) {
    s = String.format("jmp L%d", lbl); jdst = lbl;
  }
}
static class Ifz extends Inst {
  public Ifz(int t1, int lbl) {
    s = String.format("ifz t%d then L%d", t1, lbl); sr(t1); jdst = lbl;
  }
  public int refNum() { return 1; }
  public Inst renew(int t1) { return new Ifz(t1, jdst); }
}
static class Ifnz extends Inst {
  public Ifnz(int t1, int lbl) {
    s = String.format("ifnz t%d then L%d", t1, lbl); sr(t1); jdst = lbl;
  }
  public int refNum() { return 1; }
  public Inst renew(int t1) { return new Ifnz(t1, jdst); }
}
static class Ifeq extends Inst {
  public Ifeq(int t1, int t2, int lbl) {
    s = String.format("if t%d == t%d then L%d", t1, t2, lbl); sr(t1, t2);
   jdst = lbl;
  }
  public int refNum() { return 2; }
  public Inst renew(int t1, int t2) { return new Ifeq(t1, t2, jdst); }
}
static class Ifne extends Inst {
  public Ifne(int t1, int t2, int lbl) {
    s = String.format("if t%d != t%d then L%d", t1, t2, lbl); sr(t1, t2);
   jdst = lbl;
  }
  public int refNum() { return 2; }
  public Inst renew(int t1, int t2) { return new Ifne(t1, t2, jdst); }
}
static class Ifgt extends Inst {
  public Ifgt(int t1, int t2, int lbl) {
    s = String.format("if t%d > t%d then L%d", t1, t2, lbl); sr(t1, t2);
   jdst = lbl;
  }
  public int refNum() { return 2; }
```

```
    public Inst renew(int t1, int t2) { return new Ifgt(t1, t2, jdst); }
  }
  static class Iflt extends Inst {
    public Iflt(int t1, int t2, int lbl) {
      s = String.format("if t%d < t%d then L%d", t1, t2, lbl); sr(t1, t2);
     jdst = lbl;
    }
    public int refNum() { return 2; }
    public Inst renew(int t1, int t2) { return new Iflt(t1, t2, jdst); }
  }
  static class Ifge extends Inst {
    public Ifge(int t1, int t2, int lbl) {
      s = String.format("if t%d >= t%d then L%d", t1, t2, lbl); sr(t1, t2);
     jdst = lbl;
    }
    public int refNum() { return 2; }
    public Inst renew(int t1, int t2) { return new Ifge(t1, t2, jdst); }
  }
  static class Ifle extends Inst {
    public Ifle(int t1, int t2, int lbl) {
      s = String.format("if t%d <= t%d then L%d", t1, t2, lbl); sr(t1, t2);
     jdst = lbl;
    }
    public int refNum() { return 2; }
    public Inst renew(int t1, int t2) { return new Ifle(t1, t2, jdst); }
  }
```

12.4.3 値番号法による最適化

値番号法 (value numbering method) とは、比較的古くからある最適化手法で、共通部分式の削減とコピー伝播がまとめて行なえます。ブロックをまたがった広域最適化に拡張した **GVN**(global value numbering) もありますが、ここでは分かりやすい局所最適化の範囲で説明します。

その考え方は次のようなものです。プログラムの中で各テンポラリが保持する値は実行時まで分かりませんが、これを「1 番目の値」「2 番目の値」のように番号づけすることで区分します。表記が長いので `v1`、`v2` のように記すこととして、演算命令で例えば `v1+v2` というのができたら、その式に対応する値番号を割り当てます。後で再度同じものが出て来たら、それらは同じ値なので片方だけ計算すれば済みます。

それぞれのテンポラリについて、値がいきなり参照されたら (それはもっと前のブロックで計算された値が入っているわけなので) 新しい値番号を割り当て、また代入されたら右辺の値番号に対応づけます。

こうすることで、見た目は (コード上では) 同じ式でも、中に現れる変数に別の値が入っている場合は、値番号も代わるので同じ式であると勘違いすることは避けられるわけです。

では、先の中間コード辺に値番号をつけてみましょう (図 12.6)。かなり演算が削減できることが分かります。

では、これを行なうコードを見てみます。局所最適化なので、各ブロックごとに独立に `ValNumber.run` を (ブロックの本体を渡して) 呼び出します。その中では `ValNumber` オブジェクトを (ブロック本体を渡して) 生成したあと、`exec()` メソッドを呼びます。オブジェクトの中では「テンポラリ番号→値番

```
L1007: // B11
  t16 = 1           ; t16: 1
  t17 = t5 - t16    ; t17: v2 = v1 - 1
  t18 = t3[t17]     ; t18: v4 = v3[v2]
  t6 = t18          ; t6:  v4
  t19 = 1           ; t19: 1
  t20 = t5 - t19    ; t20: v2
  t21 = t3[t5]      ; t21: v5 = v3[v1]
  t3[t20] = t21     ;      v3[v2] = v5
  t3[t5] = t6       ;      v3[v1] = v4
  t22 = 1           ; t22: 1
  t4 = t22          ; t4:  1
```

図 12.6: 中間コードに値番号をつけたもの

号」「値番号→テンポラリ番号」「式文字列→値番号」の対応を保持する 3 つの表が保持されています。

exec() の中では 1 命令ずつ順に取り出し、まず定義なしで参照されるテンポラリに値番号を割り振ります (insttmp())。次に、命令の種別ごとに分かれます。

- 代入しない命令の場合、作り直すだけでよい。このとき参照しているテンポラリは必要なら値番号に応じてコピー元のテンポラリに置き換えられる。
- 代入する場合は、その代入する値番号を保持しているテンポラリがあるか調べる。もしなければ、計算は必要なので同じ種別の計算命令を生成し直し、そのテンポラリに値番号を登録する。
- 代入する場合で、同じ値番号を持つテンポラリがある場合、そのテンポラリの値番号が既に変更されてしまっていないかチェックする。OK なら、この命令をコピー命令に置き換え、代入先のテンポラリの値番号を登録する。変更されてしまっているなら、前項と同じ動作。

```
package samc1;
import java.util.*;

public class ValNumber {
  Map<Integer, String> tmpval = new HashMap<Integer,String>();
  Map<String, Integer> valtmp = new HashMap<String,Integer>();
  Map<String, String> expval = new HashMap<String,String>();
  int vcnt = 1;
  public void exec(List<IntCode.Inst> lst, boolean pr) {
    for(int k = 0; k < lst.size(); ++k) {
      IntCode.Inst op = lst.get(k), op1 = op;  insttmp(op);
      int d = op.assDest();
      if(d < 0) {
        op1 = renew(op);
      } else {
        String exp = newexp(op), v = exp2val(exp);
        if(!valtmp.containsKey(v)) {
          op1 = renew(op);
          tmpval.put(d, v); valtmp.put(v, d);
```

```
          if(pr) { System.out.printf(" t%d:%s = %s\n", d, v, exp); }
        } else {
          int t = valtmp.get(v);
          if(tmpval.get(t).equals(v)) {
            op1 = new IntCode.Move(d, t);
            tmpval.put(d, v);
            if(pr) { System.out.printf(" t%d:%s = t%d\n", d, v, t); }
          } else {
            op1 = renew(op);
            tmpval.put(d, v); valtmp.put(v, t);
            if(pr) { System.out.printf(" t%d:%s = %s\n", d, v, exp); }
          }
        }
      }
      if(pr) { System.out.println("        " + op + " ==> " + op1); }
      lst.set(k, op1);
    }
    if(!pr) { return; }
    System.out.println("--------");
    for(Integer i: tmpval.keySet()) { pl("  t"+i+" = "+tmpval.get(i)); }
    System.out.println("--------");
    for(String e: valtmp.keySet()) { pl("  "+e+" = t"+valtmp.get(e)); }
    System.out.println("--------");
    for(String e: expval.keySet()) { pl("  "+e+" = "+expval.get(e)); }
    System.out.println("--------");
  }
  private void pl(String s) { System.out.println(s); }
  private boolean alldigit(String s) { return s.matches("^[0-9]+$"); }
  private String exp2val(String e) {
    if(expval.containsKey(e)) { return expval.get(e); }
    if(alldigit(e)) { return e; }
    if(e.charAt(0) == 'v' && alldigit(e.substring(1))) { return e; }
    String v = "v"+vcnt++; expval.put(e, v); return v;
  }
  private void insttmp(IntCode.Inst op) {
    for(int i = 0; i < op.refNum(); ++i) {
      int t = op.getRefs()[i];
      if(!tmpval.containsKey(t)) {
        String v = "v"+vcnt++; tmpval.put(t, v); valtmp.put(v, t);
      }
    }
  }
  private String newexp(IntCode.Inst op) {
    String e = "?";
    int[] ref = op.getRefs();
    if(op.refNum() == 0) {
```

```
        e = op.refExp();
      } else if(op.refNum() == 1) {
        e = op.refExp(tmpval.get(ref[0]));
      } else {
        e = op.refExp(tmpval.get(ref[0]), tmpval.get(ref[1]));
      }
      return e;
    }
    private IntCode.Inst renew(IntCode.Inst op) {
      int[] r = op.getRefs();
      if(op.refNum() == 0) {
        return op.renew();
      } else if(op.refNum() == 1) {
        Integer i1 = valtmp.get(tmpval.get(r[0]));
        return i1 == null ? op : op.renew(i1);
      } else {
        Integer i1 = valtmp.get(tmpval.get(r[0]));
        Integer i2 = valtmp.get(tmpval.get(r[1]));
        return i1 == null || i2 == null ? op : op.renew(i1, i2);
      }
    }
    public static void run(IntCode ic, boolean pr) {
      for(IntCode.Block b: ic.getBlocks()) {
        if(pr) { System.out.println(b.getLabel()); }
        new ValNumber().exec(b.getBody(), pr);
      }
    }
  }
```

では「java samc1/SamC1 test.min vn pi」で出力した局所最適化すみ中間コードの当該ブロック部分を見てみましょう (図 12.7)。

```
  L1007: // B11
    t16 = 1
    t17 = t5 - t16
    t18 = t3[t17]
    t6 = t18
    t19 = t16
    t20 = t17
    t21 = t3[t5]
    t3[t17] = t21
    t3[t5] = t18
    t22 = t16
    t4 = t16
```

図 12.7: 最適化を行なった中間コード

確かに、余分な演算が削減されています。ただ、使われなくなったコピー文も (ここまででは不要命令の削除をしないため) 残っている。参照されていないテンポラリへの代入は削除していいでしょうか? それは NO で、このブロックから外に出たところで参照さているかも知れないので、やみくもに削除はできません。つまり、それをやるためにはより詳しい解析情報の抽出、具体的にはデータフロー解析が必要になるのです。

演習 2 例題をそのまま動かせ。動いたら、自分で作成したさまざまなコードに対して、値番号による局所最適化の有無で中間コードがどのように違うかを検討せよ。

12.5 データフロー解析

12.5.1 データフロー解析とデータフロー方程式

前節で述べたように、ある程度以上の最適化のためには「この変数の値はこの後参照されるだろうか。」(されないなら、その変数に値を格納しておく命令は不要)、「この場所でこの変数に入っている値を設定する命令はどれとどれだろうか。」(それが 1 つだけで、その値がたまたま別の変数にも入っているなら実は変数そのものが要らないかもしれない)、などの情報が必要になります。

この種の情報をフローグラフから抽出するのが**データフロー解析** (dataflow analysis) です。以下では代表的なデータフロー解析の問題について説明していきます。

生きている変数の問題

データフロー解析の最初の例として、先にあげた「この変数の値はこの後参照されるだろうか。」という問題を考えます。これは変数が**生きている** (live)、つまりその変数の値が後で参照される可能性があるか、あるいは**死んでいる** (dead)、つまりその可能性がないか、を決めるものです。

個々のブロック b について見ると、その入口で生きている変数と出口で生きている変数の間には次の関係があります。

$$\begin{aligned} LiveOut[b] &= \bigcup_{b' \in Succ[b]} LiveIn[b'] \\ LiveIn[b] &= Ref[b] \cup (LiveOut[b] - Ass[b]) \end{aligned}$$

これは、b の出口で生きている変数の集合とは b に引き続くようなブロック b' の入口で生きている変数の和集合であり、また b の入口で生きている変数の集合は出口で生きている変数の集合から b で値を設定してしまう変数は除き、代りに b で (もし値を設定するならそれより前に) 参照する変数を加えたものであることを表しています。

次にフローグラフ G に関してこの**データフロー方程式** (dataflow equation) を解きます。それには次の手順を用います。

1. まず、各ブロックについて $Ref[b]$、$Ass[b]$ を求める。
2. $LiveIn[b]$、$LiveOut[b]$ についてはとりあえず空集合とする。
3. 各ブロックについて上の方程式に従って $LiveIn[b]$、$LiveOut[b]$ を計算することを反復することをもはや各集合が変化しなくなるまで行う。

各反復において、$LiveIn[b]$ と $LiveOut[b]$ は (変化するとすれば) 要素がつけ加わる方向にのみ変化し、そして変数の個数は有限ですから、この手順は必ず停止し解が求まります。

アルゴリズムは上記の通りですが、上の方程式はブロック出口の値に基づいて入口の値を求めます。そのため、実際にブロックを調べる順番は後のブロックから前に向かって行なうと効率がよくなります。このような問題を**後向きフロー解析** (backward flow analysis) の問題と呼びます。

UD 連鎖の問題

次の例として、「この場所でこの変数に入っている値を定義する命令はどれとどれか」の問題を取り上げます。その前にいくつか用語を説明しましょう。

定義 **変数 x への定義** (definition) とは、変数に値を設定する可能性をもつ命令を言う。典型的には x への代入がそうだが、x を参照渡し引数とする手続き呼出しは x を更新する可能性があるため x の定義となる。x に必ず値が設定される場合にはその定義は**曖昧でない** (unambiguous) という。

定義 変数 x に対する定義 d と命令 s について、d から s へ至る経路上に x に対する別の曖昧でない定義 d' があるとき、d' は d を**殺す** (kill) という。言い換えれば d' によってその経路を通るときは s への d の影響が及ばなくなる。

定義 定義 d と命令 s について、d を殺す別の定義が存在しないような d から s への経路が 1 つ以上存在するとき、d は s に**到達** (reach) する、という。

以上の用語から、この問題は**到達する定義** (reaching definition) の問題、または変数を使う (use) 場所ごとに、それに影響し得る定義 (definition) の連鎖を求めるため **UD 連鎖** (use-definition chain) の問題と呼ばれます。この問題については、個々のブロック b について次のデータフロー方程式が成り立ちます。

$$\begin{aligned} DefIn[b] &= \bigcup_{b' \in Pred[b]} DefOut[b'] \\ DefOut[b] &= Def[b] \cup (DefIn[b] - Kill[b]) \end{aligned}$$

すなわち、ブロック b に入ってくる定義の集合はそのブロックの上流である全ブロックから出てくる定義を全部合せたものであり、また b から出ていく定義は b における定義と、入ってきた定義のうち b で殺されないものとを合せたものとなる、ということです。

このデータフロー方程式は先の「生きている変数」の式と In/Out が逆になった形をしています。このため、解を求める手順は前と同様でよいのですが、実用的には手続きの入口点から先に向かって反復計算すると効率が良くなります。このような問題を**前向きフロー解析** (forward flow analysis) の問題と呼びます。

DU 連鎖の問題

UD-連鎖が「各参照ごとに、そこに到達し得る定義を求める」のに対し、その逆、つまり「各定義ごとに、それに到達され得る参照を求める」問題を **DU 連鎖** (definition-use chain) の問題と呼びます。これを求めるデータフロー方程式は次のようになります。

$$\begin{aligned} UseOut[b] &= \bigcup_{b' \in Succ[b]} UseIn[b'] \\ UseIn[b] &= ExposedRef[b] \cup (UseOut[b] - UseKill[b]) \end{aligned}$$

ただし $ExposedRef[b]$ は b の中で定義されないか、または定義されてもそれより前の位置で値を参照する文の集合、$UseKill[b]$ は b で定義される値を参照する b 以外の場所にある文の集合を意味します。各集合の意味は違っても、方程式の形は最初に述べた生きている変数の問題と同じですね。

利用可能式の問題

ここまでに示したデータフロー方程式はいずれも、上流/下流のどこかで問題としている事象 (値の定義とか変数の参照) が起き得るかどうかを知るためのものであり、その主旨から**いずれかの経路解析** (any-path analysis) と呼ばれます。

データフロー解析にはそれと対をなす**全ての経路解析** (all-path analysis) も存在します。その例として、ブロックにまたがる共通部分式の最適化を考えましょう。コード上のある場所に出てきた式の値を改めて計算しないですむためには、その上流の「全ての経路で」その式の値が計算ずみである必要がありますね。この問題を**利用可能式** (available expression) の問題と呼び、次のデータフロー方程式で表されます。

$$\begin{aligned} AvailIn[b] &= \bigcap_{b' \in Pred[b]} AvailOut[b'] \\ AvailOut[b] &= DefExp[b] \cup (AvailIn[b] - KillExp[b]) \end{aligned}$$

ここで $DefExp[b]$ は b で計算される式の集合、$KillExp[b]$ は b において式に含まれる変数のどれかが殺されるため計算ずみの値が有効でなくなる式の集合を意味します。この方程式もこれまでと同様の解法で扱うことができますが、ただし計算が $\bigcap$ を用いて集合を小さくする方向に進むので、$AvailIn[b]$、$AvailOut[b]$ の初期値は初期ブロックのみ空集合、あとは全ての式の集合とする必要があります。

コピー文の問題

利用可能式と類似した問題に**コピー文** (copy statement) の問題があります。これは例えばある場所に `y = x` という代入 (コピー文) があり、後続するブロックに `y` の参照があったとき、これを `x` に置き換えてしまってよいかどうかを決めるものです。つまり、値番号法でやったものを広域最適化でやるわけです。置換えが可能なためには、次の条件が必要です。

- `y` を参照している箇所に到達する全ての `y` に対する定義が `y := x` である。
- `y = x` が行われた後 `y` の参照までに、`x` の値を変更する可能性がない。

このうち前者は UD 連鎖の情報をもとに知ることができ、後者は次のデータフロー方程式により計算できます。

$$\begin{aligned} CopyIn[b] &= \bigcap_{b' \in Pred[b]} CopyOut[b'] \\ CopyOut[b] &= DefCopy[b] \cup (CopyIn[b] - KillCopy[b]) \end{aligned}$$

ここで $CopyIn[b]/CopyOut[b]$ はそれぞれ b の入口/出口で置き換え可能なコピー文の集合、$DefCopy[b]$ は b に含まれるコピー文でその後 b 内で左辺、右辺どちらも変更されないものの集合、$KillCopy[b]$ は b **以外の**場所にある全てのコピー文から b 内で左辺、右辺どちらかが変更されるものを除いた集合です。

12.5.2 生きている変数の問題を解く

それでは 1 つだけ具体例として、生きている変数の問題を解くコードを見ていただきましょう。今回のコードはこれで最後です (おつかれ様です)。クラスの前半にサポート用のメソッドがあり、そこで「各ブロックごとに空集合を用意」とか和集合、共通集合の演算を作成しています。

実際にデータフロー方程式を解くところは、先に述べた定義通りに集合演算をおこない、変化がなくなるまで繰り返すだけです。

```
package samc1;
import java.util.*;
public class Dataflow {
  List<IntCode.Block> blocks;
  public Dataflow(List<IntCode.Block> b) { blocks = b; }
```

```
  private List<Set<Integer>> newListSet() {
    List<Set<Integer>> lst = new ArrayList<Set<Integer>>();
    for(IntCode.Block b: blocks) { lst.add(new TreeSet<Integer>()); }
    return lst;
  }
  private static Set<Integer> add(Set<Integer> s1, Set<Integer> s2) {
    Set<Integer> s3 = new TreeSet<Integer>();
    for(Integer i: s1) { s3.add(i); }
    for(Integer i: s2) { s3.add(i); }
    return s3;
  }
  private static Set<Integer> sub(Set<Integer> s1, Set<Integer> s2) {
    Set<Integer> s3 = new TreeSet<Integer>();
    for(Integer i: s1) { if(!s2.contains(i)) { s3.add(i); } }
    return s3;
  }

  List<Set<Integer>> liveIn, liveOut;
  List<Set<Integer>> blkRef, blkDef;
  public List<Set<Integer>> getLiveIn() { return liveIn; }
  public List<Set<Integer>> getLiveOut() { return liveOut; }
  public void calcLiveInOut() {
    liveIn = newListSet(); liveOut = newListSet();
    boolean chg = true;
    while(chg) {
      chg = false;
      for(IntCode.Block b: blocks) {
        Set<Integer> li1, lo1 = new TreeSet<Integer>();
        for(Integer i: b.getSucc()) { lo1 = add(lo1, liveIn.get(i)); }
        li1 = add(b.getRef(), sub(lo1, b.getDef()));
        int i = b.getId();
        if(!(liveIn.get(i).equals(li1))) { liveIn.set(i, li1); chg = true; }
        if(!(liveOut.get(i).equals(lo1))) { liveOut.set(i, lo1); chg = true; }
      }
    }
  }
  public void showLiveInOut() {
    for(IntCode.Block b: blocks) {
      int id = b.getId();
      System.out.printf("LiveIn/Out[B%d] = {", id);
      for(Integer i: liveIn.get(id)) { System.out.print(" "+i); }
      System.out.print(" } {");
      for(Integer i: liveOut.get(id)) { System.out.print(" "+i); }
      System.out.println(" }");
    }
  }
```

```
}

% java samc1/SamC1 test.min vn clio plio
LiveIn/Out[B0] = { } { 2 3 4 }
LiveIn/Out[B1] = { 2 3 4 } { 2 3 }
LiveIn/Out[B2] = { 2 3 } { 2 3 4 5 }
LiveIn/Out[B3] = { 2 3 4 5 } { 2 3 4 5 }
LiveIn/Out[B4] = { 2 3 4 5 } { 2 3 4 5 10 }
LiveIn/Out[B5] = { 2 3 4 5 } { 2 3 4 5 10 }
LiveIn/Out[B6] = { 2 3 4 5 10 } { 2 3 4 5 }
LiveIn/Out[B7] = { 2 3 4 5 } { 2 3 4 5 }
LiveIn/Out[B8] = { 2 3 4 5 } { 2 3 4 5 15 }
LiveIn/Out[B9] = { 2 3 4 5 } { 2 3 4 5 15 }
LiveIn/Out[B10] = { 2 3 4 5 15 } { 2 3 4 5 }
LiveIn/Out[B11] = { 2 3 5 } { 2 3 4 5 }
LiveIn/Out[B12] = { 2 3 4 5 } { 2 3 4 5 }
LiveIn/Out[B13] = { 2 3 4 } { 2 3 4 }
LiveIn/Out[B14] = { } { }
%
```

図 12.8: 生きている変数の結果

生きている変数の問題は変数 (テンポラリ) の集合を対象にしていたので比較的作りやすかったのですが、UD 連鎖などの問題は「計算を行なう命令」が対象なので、個々の命令に番号をつけて扱う必要があり、だいぶ面倒になると思います。そういうわけで、ここでは作成していません。

とりあえず、局所最適化したあとの中間コードについて生きている変数の結果を表示させてみました (図 12.8)。

これで、先に出て来たブロック (B11) の出口で生きている変数は t2, t3, t4, t5 だけと分かります。そうなれば、コードを下から順に処理し、生きていない変数を定義する命令には削除の印をつけ、またそうでない命令が参照している変数は生きている集合に追加します (図 12.9)。

これで「X」とついている命令は削除でき、11 命令が 7 命令に減らせました。

演習 3 この例題をそのまま動かしてみよ。また、自分でも別のプログラムを処理してみて、どれくらい削除できるか検討せよ。

演習 4 上記の手順による不要コード削除を実装してみよ。

12.6 課題 12A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 12」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。

```
L1007: // B11
  t16 = 1            ↑ ; t2 t3 t4 t5 t16 t17 t18 t21
  t17 = t5 - t16     ↑ ; t2 t3 t4 t5 t16 t17 t18 t21
  t18 = t3[t17]      ↑ ; t2 t3 t4 t5 t16 t17 t18 t21
  t6 = t18           ↑ ; X
  t19 = t16          ↑ ; X
  t20 = t17          ↑ ; X
  t21 = t3[t5]       ↑ ; t2 t3 t4 t5 t16 t18 t21
  t3[t17] = t21      ↑ ; t2 t3 t4 t5 t16 t18 t21
  t3[t5] = t18       ↑ ; t2 t3 t4 t5 t16 t18
  t22 = t16          ↑ ; X
  t4 = t16           ↑ ; t2 t3 t4 t5 t16
```

図 12.9: 不要コードの削除のようす

- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 強い型の言語と弱い型の言語のどちらが好みですか。またそれはなぜ。
  Q2. 記号表と型検査の実装について学んでみて、どのように思いましたか。
  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

12.7 付録: ループ最適化

ループの内部はコードの他の部分に比べて多数回実行されるため、ループ外コードの実行時間を犠牲にしてでもループ内コードの実行時間を短縮することはプログラム全体の実行時間短縮に寄与します。これを**ループ最適化** (loop optoimization) と呼びます。ループ最適化も中間コードに対して適用しますが、読みやすさのため以下ではソースコード表現を用いて説明します。

ループ不変文の移動

ループ不変文の移動 (loop invariant code motion) とは、ループの周回を通じて結果が変化しない計算を行う文をプリヘッダに移動して 1 回の実行で済ませることを言います。まず、ループ L について不変な文を次のようにして求めます。

1. オペランドが定数、または到達する定義が全て L 外にある変数のみから成る文に「ループ不変」の印をつける。
2. オペランドが定数、または到達する定義が全て L 外にある変数、または到達する定義が 1 つだけであり、それが L に含まれていて、その定義に「ループ不変」の印がついているような変数のみから成る文にも「ループ不変」の印をつける。

上記により見つかったループ不変文をそのままプリヘッダに移してよいわけではありません。今回の冒頭の説明のように、ループの条件が成り立った時しか実行したらまずいものもあるからです。そこで、次の条件を満たす時だけ移動するという方法があります。

(a) s は必ず 1 回以上実行される。

(b) x と同じ変数を定義する文は L 内では s だけである。

(c) L 内の全ての x の使用について、そこに到達する定義は s だけである。

(a) を調べるにはループの出口となるブロック (L に含まれない後続ブロックをもつような L のブロック) 全てが s を含むブロックに支配されるかどうかを調べます。もしそうなら、s を通らずにループを出る方法はないのだから、s は必ず 1 回以上実行されます。これらの制約を満たすことは s を移動できる必要十分条件ではありませんが、手持ちのコード解析情報から調べるのが容易であり、移動をそのような s に限ったとしてもある程度の最適化が行えることが経験的に知られています。

`while` ループや `for` ループではループ本体が 1 回も実行されない可能性があるので、条件 (a) がある限りループ本体に含まれる文はどれも移動できません。それに対しては、冒頭でやったように if で囲むようにすれば、移動できるようになります。

ループ内分岐の移動

ループ不変文の検出に伴い、if 文などの条件式がループ不変であることが分かる場合があります。そのときは、その if を次のように外に出すことが考えられます。つまり、左を右のように変更するわけです。これによりコード量は増えますが、実行時間は短くできます。

```
t1 = ...            t1 = ...
while(...) {        if(t1)
    C;                  while(...) {
    if(t1)                C; A; D;
         A;             }
    else            else
         B;             while(...) {
    D;                    C; B; D;
}                       }
```

帰納変数の最適化

ループ最適化で重要なものの 1 つに、ループとともに変化する変数 (特にループ周回ごとに定数値だけ増減するもの) に関するものがあります。まず次の用語を定義します。

定義 **永続ループ変数** (persistent loop variable) とは、ループ入口において生きていて、なおかつループ内で定義される (言い換えればループ内でまず参照され、続いて定義される) 変数を言う。

定義 **ループ帰納変数** (loop induction variable) とは、ループの周回とともに値が一定値ずつ増加 (または減少) するような変数を言う

多くの場合ループを制御する変数は永続ループ変数になります。次の例を考えてみてください。

```
i := 0;                         p := top;
while i < x do begin            while p <> nil do begin
    ...                             ...
    i := i + 1                      p := p^.next
end                             end
```

左の場合 `i` は永続ループ変数でありかつ帰納変数でもあります。右の場合 `p` は永続ループ変数だが帰納変数ではありません。

帰納変数は永続ループ変数である場合 (基本帰納変数とよばれる) 以外に、別の帰納変数の値をもとに値を計算している場合もありますが、そのような帰納変数ば基本機能変数に変換できます (ルー

プ 1 周回ごとに定数を加減するコードに置き換える)。このような変換は多くの場合乗算を加減算で置き換えることになるので、演算強さの軽減が行えることになります。

さらに、別の帰納変数と常に同じ値をもつことがわかる帰納変数や、条件判定にしか現れず、その判定条件を別の帰納変数を使うように書き換えられる帰納変数は、取り除いてしまうことができます。また、帰納変数はループ入口での値と終了条件を調べることで取り得る値の範囲を比較的容易に知ることができ、この情報を用いることで条件式が定数式である場合を多く検出できます。

まず基本帰納変数の検出は次のようにして行えます。

- 変数 i が永続ループ変数であり、かつループ内での i に対する定義 s が `i := i` $\pm$ C(ただし C はループ不変) の形であり、s がループ周回ごとに必ず 1 回実行されるならば i は帰納変数である。

s がループ周回ごとに必ず 1 回実行されるかどうかは、ループ帰辺の直前のブロック b が s を含むブロックに支配されるかどうかで調べられます。次に、基本帰納変数以外の帰納変数を考慮します。

- 変数 j が永続ループ変数でなく、かつループ内の j に対する定義 s が `j := i * c` $\pm$ `d`(ただし `i` は帰納変数、`c`、`d` はループ不変) の形であり、s がループ周回ごとに必ず 1 回実行されるなら j は帰納変数である。

ここで i が基本帰納変数であるとき、j は i の**家族** (family) であると言います。i が基本帰納変数でないとき、i は別の基本帰納変数 i_0 の家族なので、次の 2 条件が成り立てば j も i_0 をもとに計算するように (つまり i_0 の家族に) できます。

1. i への定義と s の間に i_0 への定義がはさまっていない。
2. ループ外の i への定義が s に到達することがない。

例として、次のコードを考えてみます。

```
i0 = 0;                          i0 = 0;
while(...) {                     while(...) {
    i = i0 * 2 + 1;                  j = i * 2 + 3;   /* s */
    ...                              i = i0 * 2 + 1;
    i0 = i0 + 1;                     ...
    j = i * 2 + 3;  /* s */          i0 = i0 + 1;
    ...                          }
 }
```

ここで左側のコードは条件 1、右側のコードは条件 2 を満たしません。右の場合は j の値がループの最初の周回と 2 周回目の間で定数増減にならないので、帰納変数ではありません。左の場合は j の値が s の時点におけるより 1 周回前の i_0 の値によって計算しなければならないことを注意しておけば、帰納変数として最適化することができます。

次に永続ループ変数でない帰納変数 j を永続ループ変数に変換しますが、その手順は次の通り。

1. 新しい変数 u を導入し、その初期値設定をループのプリヘッダに追加する。
2. 家族の基となっている変数の更新の直後に、u の増減を行う定義を追加する。
3. j の定義を j `:=` u に取り替える。

もちろん、新しく追加される複数の変数が同じ初期値、増減値、挿入位置であるなら、これらを 1 つで兼ねます。上の例に対して適用した結果は次のようなコードになるでしょう。

```
i0 = 0; u = 1; v = 5;
while(...) {
    i = u;
    ...
    i0 = i0 + 1; u = u + 2; v = v + 4; j = v;
    ...
}
```

この状態では演算強さは軽減された代りに値のコピーが増えていますが、コピー伝播を行なえば不要なコピーは削除できます。加えて、帰納変数を基本機能変数に変換した結果、これまで他の帰納変数を計算するためだけに使われていた帰納変数の参照がなくなる場合があります。その場合はその変数の計算も含めて不要コードとして削除できます。

また、同じ参照でも条件式においてループ不変式と比較されるもののみが残った場合は、これを別の帰納変数に関する判定に置き換えて取り除くことができます。

ループ展開

ループ展開 (loop unrolling) とは、ループ本体を複数回重複させることによりループのための分岐や判定の実行回数を減らす最適化を言います。最も極端には、ループの周回回数が翻訳時にわかり、その値があまり大きくなく、ループ本体も小さい場合には、その回数だけ本体を展開することでループを全くなくしてしまう場合もあります。

そのように極端でなくても、ループ周回数が偶数であるとわかればループ本体を 2 回展開することにより、展開後のループ周回数は半分になり、ループ制御に要するオーバヘッドは半分にできます。また、展開された 2 個のループ本体について、共通式のくくり出しができる場合も多く、そのときはさらに実行速度を高められます。ループ周回数がわからない場合には、ループの終了判定を展開した本体 1 個につき 1 回ずつ行います (例えば下記の左のコードを右のように変形する)。

```
while(C) {              while(C) {
    B;                      B;
}                           if(!C) break;
                            B;
                        }
```

この場合でもループ先頭への分岐は減りますし、この変形の結果、共通式のくくり出しが可能になる場合もあります。

ループ融合

ループ融合 (loop fusion) とは、同一周回数をもつループが隣接している場合にそれらをまとめて 1 つのループにしてしまうことを言います。例えば次のような場合です。

```
for i := 1 to n do a[i] := i;       for i := 1 to n do begin
for i := 1 to n do b[i] := n - i;       a[i] := i; b[i] := n - 1
                                    end;
```

融合によってループ制御のオーバヘッドは半分になります。制御変数の範囲が同一でない場合には、単純に半分ではありませんが、終了判定とループ先頭への分岐は節約できます。もちろん、融合しても元のコードと意味が変らないためには、双方のループ本体に干渉がないことが必要です。

配列の線形化

ループ融合と類似していますが、2 次元配列 (一般には多次元配列) を順番にアクセスするコードはループの入れ子になります。

```
a: array[1..10][1..10] of ...;
...
for i := 1 to 10 do
    for j := 1 to 10 do a[i][j] := 0.0;
```

この場合、配列 a は主記憶上に連続して配置されているので、それを仮想的に 1 次元配列だとみなして 1 重ループに変換できる場合があります。

```
for i := 1 to 100 do a[i] := 0.0;
```

これを**配列の線形化** (linearizaton) と呼びます。線形化はループのオーバヘッドを減らす効果があり、またベクトル命令 (1 つの命令で多数の番地に同じ演算を行なえる命令) を持つプロセサの場合には、ベクトル長を長くするという点でも実行効率を高める効果があります。

線形化が行えるためには、最内側のループにおいて配列の一番最後の次元の添字式が対応する次元の上限から下限まで変化することと、その 1 つ外側のループで 1 つ前の次元の添字式が 1 つずつ変化することが条件となります。ここで最後 (最左側) の次元となっているのは配列が行単位 (column-wise) で配置される言語の場合であり、Fortran のように列単位 (row-wise) で配置される場合には最初 (最右側) から順に調べます。

＃13　オブジェクト指向の実装

13.1　オブジェクト指向言語の実装の留意点

前回までで一般的な手続き型言語の実装については一通り扱い終わりましたが、今日では手続き型言語の中でもオブジェクト指向言語が広く使われるようになっています。そして、オブジェクト指向言語の実装にはこれまでに取り上げて来た手続き型言語に無い次のような留意点があります。

(a) オブジェクト (インスタンス) の生成 — 複数の変数 (インスタンス変数) 群と手続き群 (メソッド) が組になったインスタンスが作り出せる。

(b) メソッド呼び出し — インスタンスを指定してメソッドを呼び出すと、そのインスタンスに付随するメソッドが呼び出される (動的分配)。

(c) インスタンス変数アクセス — インスタンスメソッド内から変数にアクセスすると、そのインスタンスのインスタンス変数がアクセスできる。

これらのうち (a) は抽象データ型 (内部で複雑な構造を持つような値を複数作り出す機能) を実現できるという意味を持ちます。(b) は様々なオブジェクトに対し同一のコード (呼び出し) が適用でき、オブジェクトの種類が多くても記述の複雑さを抑制できるという意味を持ちます。(c) は情報隠蔽ないしカプセル化 (オブジェクト内部のデータを外部から見られたり変更されたりすることを防ぐ) が可能になるという意味を持ちます。

そして、ここから先は言語のデザインによって有無が違って来ますが、次のような機能を持つ言語も複数見られます。

(d) クラス (オブジェクトの形を定義する「型枠」のような構文単位) が定義でき、クラスに基づいて複数のオブジェクト (インスタンス) を生成する。

(e) クラス間またはオブジェクト間に継承関係が定義でき、それに基づいて「あるオブジェクトに類似した (しかし同一ではなく違いもある) オブジェクト」を定義できる。

(f) クラスやメソッドについて静的な型が定まっていて、それに基づき型検査がおこなえる。

(g) コンテナ型 (配列などのように他のオブジェクトを格納することを目的としたオブジェクトの型) など、さまざまな種別のオブジェクトを扱うようなオブジェクトに対して、扱うオブジェクトの種類 (型) をパラメタとして与えて全体の型を定める型パラメタ機能を持つ。

これらは後の方にいくほど機能的にも理論的にも複雑になります。ここでは具体的な実装例を見るため、まず (a)〜(d) について取り上げ、そのあと (e)〜(f) について簡単に説明します。(g) の型パラメタについては内容が多くなりすぎるため、ここでは扱わない。

13.2　インスタンス変数と動的分配の実装

13.2.1　C 言語上での実装記述

ここまででの言語処理系の実装例では翻訳系の目的コードとして機械語 (アセンブリ言語) を採用して来ましたが、ここから先はコードが複雑になるので C 言語を目的コードとして使用します。

C 言語を目的コードとして使用することは、C コンパイラによる翻訳が必要になるという手間や、C 言語の型検査をパスするようにコードを生成する必要があるという手間が掛かりますが、一方で次のような利点もあります。

- C コンパイラは多くのシステムで利用可能なので、生成コードを多くのシステムで動かせる。
- 実用レベルの C コンパイラは最適化に力を入れているため、自前で最適化を作成するより高品質な (性能のよい) コードが結果的に得られることが多い。
- C の方がアセンブリ言語より読みやすいため、生成コードの検討が容易である。
- 生成コードに間違いがあった時に、C コンパイラの型検査によって検出できることが多くある。

13.2.2 データ構造の設計

以下では、前節の (a)〜(c) を実現するようなデータ構造を検討し、続いてそれを C 言語のデータ構造として表現します。まず最初に一番複雑そうな (b) から検討しましょう。(b) の本質は、あるオブジェクトについて、そのオブジェクトが持つメソッドの名前を指定して呼び出す、というところにあります。これを実現するため、図 13.1 のような構造を使用します。これをメソッドを検索する表であることから「メソッドテーブル」と呼びます。

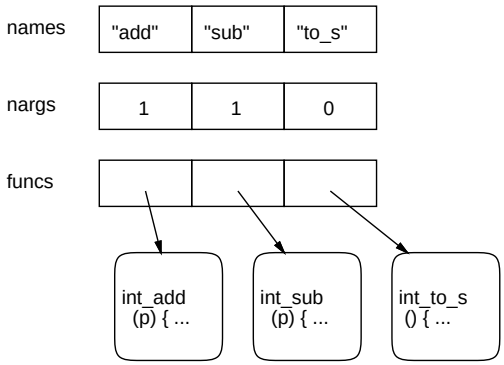

図 13.1: メソッドテーブルの実装例

使い方ですが、メソッド名を文字列として与えられたら、配列 `names` の上で探索して該当位置をみつけ、その対応位置にある配列 `funcs` の関数のコードを呼び出せばよいわけです。C 言語には「関数へのポインタ」を扱う機能があるため、これが素直に書けます。図 13.1 は整数オブジェクト用の表を想定しているので、各関数きは整数オブジェクト用ですが、実数オブジェクト用の表であれば実数オブジェクト用の関数へのポインタを入れておきます。

なお、配列 `nargs` はそれぞれのメソッドが (オブジェクト自身に加えて) いくつパラメタを受け取るかを表しています。整数オブジェクトを文字列に変換する `to_s` は「`i.to_s`」のようにパラメタを受け取りませんが、足し算は「`i.add(3)`」のようにパラメタが 1 つ必要です。

実際には、オブジェクトごとにそのオブジェクト用の表を使う必要があるので、オブジェクト 1 つごとにメソッドテーブルへのポインタを持つようにします。メソッドテーブルは 3 つの配列なので、それをレコードにすればポインタは 1 つでいいのですが、ここではあとで読むコードが簡単になるため、3 つのポインタでそれぞれの配列を指しています。

そして、整数オブジェクトであれば (`ty_int` という型名をつけていますが)、先頭にその種別 (整数は 1 としました) を表すタグ (番号)、その次にメソッドの個数 (検索時に上限が必要なので)、その後に 3 つの配列へのポインタがあり、そのさらに下にそのオブジェクトのインスタンス変数を必要なだけ取ります。整数オブジェクトであればその整数の値があれば十分なので `val` というフィールドを 1 つ持たせました。

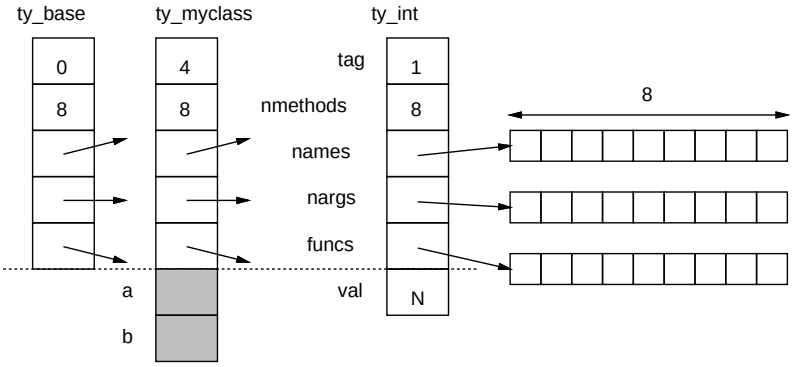

図 13.2: オブジェクトの構造

整数オブジェクトはすべてこの同じ形ですが、オブジェクトの種類が違えば形が違います (そして指すメソッドテーブルも違います)。たとえば myclass にインスタンス変数 a、b があれば、それはオブジェクトの末尾に置かれるのでその部分が整数と違います。インスタンス変数がまったく無ければ ty_base のような形になります。

ここで重要なのは、点線から上の部分はどのオブジェクトでも共通であり、同じコードで扱えるという点です。C 言語でいえば、ty_int へのポインタでも ty_myclass へのポインタでも ty_base へのポインタにキャストできますから、そうしてから扱うことでどの種類のオブジェクトでも同じに処理が書けます。

13.2.3 実行時ライブラリのソース

前節で紹介した方針によるオブジェクトとして nil クラス (値がないことを表す)、int クラス (整数)、str クラス (文字列) の 3 種類のクラスをまず手で作成します。後で作成する言語ではこれらは**組み込みクラス** (bulit-in class) として「既に存在する」状態で始まります。そのようなものが無いと言語として成り立たないことは明らかですから。

まず、ヘッダファィルを示します。これはこのライブラリを使う C 言語ソースで読み込むことを意図しています。まず 3 種類のクラスに対応するタグ値と、整数および実数オブジェクトを生成する関数を宣言します (nil は 1 つだけインスタンスを用意すればいいので生成は不要)。オブジェクト値は C 言語上では「任意のポインタ」を表す void*型として扱います。

```
/* objlib.h -- header file for simple O-O runtime. */
#define TAG_NIL 0
#define TAG_INT 1
#define TAG_STR 2
void *int_new(int v);
void *str_new(char *s);
```

続いて、3 種類のレコード型を定義します。構造は先に説明した通りです。nil 値はその 1 つだけのインスタンスをグローバル変数として用意し、そのアドレスを使うようになります。

```
typedef struct {
  int tag, nmethods; void* (**funcs)(); char* *names; int *nargs;
} ty_base;
extern ty_base nil_val;
typedef struct {
  int tag, nmethods; void* (**funcs)(); char* *names; int *nargs;
  int val;
```

```
} ty_int;
extern ty_int int_1_val, int_0_val;
typedef struct {
  int tag, nmethods; void* (**funcs)(); char* *names; int *nargs;
  char *str;
} ty_str;
```

最後に、メソッドを呼び出すための関数 send0、send1 と条件判断の際に呼び出す関数 tst の宣言があります。これらの内容は後で読みます。

```
void *send0(void *p, char *name);
void *send1(void *p, char *name, void *q);
int tst(void *p);
```

ではここから、ライブラリ本体のソースです。必要な include から始まります。

```
/* objlib.c --- runtime for simple O-O language. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "objlib.h"
```

まず nil オブジェクトの実装です。メソッドとしては to_i、to_s、print、println があります。to_i は整数の 0、to_s は文字列の"nil"を返します。残りり 2 つは nil と出力するもので、改行の有無が違うだけです。

```
void *nil_to_i(void *p) {
  return &int_0_val;
}
void *nil_to_s(void *p) {
  return str_new("nil");
}
void *nil_print(void *p) {
  printf("nil"); return p;
}
void *nil_println(void *p) {
  printf("nil\n"); return p;
}
char *nil_names[] = {"to_i", "to_s", "print", "println"};
int nil_nargs[] = {0,0,0,0};
void* (*nil_funcs[])() = {nil_to_i, nil_to_s, nil_print, nil_println};

ty_base nil_val = {
  TAG_NIL, sizeof(nil_nargs)/sizeof(int), nil_funcs, nil_names, nil_nargs
};
```

そして末尾に names、nargs、funcs の配列を用意し、1 つだけある nil オブジェクトはタグ、メソッド数、3 つの配列へのポインタで初期化します。

次は順番を入れ換えて文字列を読みましょう (この方が少しシンプル)。まず文字列への変換 to_s は、すでに自身が文字列オブジェクトなので自分を返すだけです。to_i は atoi を呼んで文字列を整

数にし、整数オブジェクトを返します。print、println は値の文字列を打ち出します。どのメソッドも最初の引数として自オブジェクトへのポインタを受け取ります。add は 2 つの文字列の連結 add で、これはバッファに 2 つの文字列オブジェクトの文字列値をくっつけた文字列を構成した後、その文字列を内容とする新たなオブジェクト値を返します prompt は文字列をプロンプトとして出力し、入力した文字列を返します。

```
void *str_to_s(void *p) {
  return p;
}
void *str_to_i(void *p) {
  return int_new(atoi(((ty_str*)p)->str));
}
void *str_print(void *p) {
  printf("%s", ((ty_str*)p)->str); return p;
}
void *str_println(void *p) {
  printf("%s\n", ((ty_str*)p)->str); return p;
}
void *str_add(void *p, void *q) {
  char buf[100];
  sprintf(buf, "%s%s", ((ty_str*)p)->str, ((ty_str*)send0(q, "to_s"))->str);
  return str_new(buf);
}
void *str_prompt(void *p) {
  char buf[100];
  printf(((ty_str*)p)->str); fgets(buf, 100, stdin);
  buf[strlen(buf)-1] = '\0'; return str_new(buf);
}
char *str_names[] = {"to_i", "to_s", "print", "println", "add", "prompt"};
int str_nargs[] = {0,0,0,0,1,0};
void* (*str_funcs[])() = {str_to_i, str_to_s, str_print, str_println,
  str_add, str_prompt};
void *str_new(char *s) {
  ty_str *p = (ty_str*)malloc(sizeof(ty_str));
  p->tag = TAG_STR; p->nmethods = sizeof(str_nargs)/sizeof(int);
  p->funcs = str_funcs; p->names = str_names; p->nargs = str_nargs;
  p->str = (char*)malloc(strlen(s)+1); strcpy(p->str, s);
  return (void*)p;
}
```

3 つの配列 (メソッドテーブル) は先と同様。そして str_new ですが、まずレコード領域を malloc で割り当て、それぞれのフィールドを初期化していきます。文字列値については、値の文字列が入る領域を割り当て、そこに値をコピーします (元のポインタをそのまま使っていると値が変化してしまう恐れがあるので)。初期化が終わったらレコードのポインタを返します。

最後に整数です。to_i は自身をそのまま返し、to_s は sprintf でバッファに文字列を出力し、文字列オブジェクトにして返します。print、println は数値を出力します。その先ですが、+-*/%の演算はみなメソッド名と演算を除けば同じ形になるので、その同じ形をマクロで定義し、マクロを 5

回呼び出してそれぞれを定義します。いずれも 2 つのオブジェクト (2 つ目は to_i を呼んで整数にする) から値を取り出し、マクロ引数で渡した式 exp により演算して結果を整数オブジェクトとして返します。6 つの比較演算については、値の真偽により予め用意してある 0 または 1 の整数オブジェクトへのポインタ値を返すところが違います。

```
void *int_to_i(void *p) {
  return p;
}
void *int_to_s(void *p) {
  char buf[100];
  sprintf(buf, "%d", ((ty_int*)p)->val); return str_new(buf);
}
void *int_print(void *p) {
  printf("%d", ((ty_int*)p)->val); return p;
}
void *int_println(void *p) {
  printf("%d\n", ((ty_int*)p)->val); return p;
}
#define icalc(name, exp) \
void *name(void *p, void *q) {\
  int x = ((ty_int*)p)->val, y = ((ty_int*)send0(q, "to_i"))->val; \
  return int_new(exp); }
icalc(int_add, x+y)
icalc(int_sub, x-y)
icalc(int_mul, x*y)
icalc(int_div, x/y)
icalc(int_mod, x%y)
#define icmp(name, cmp) \
void *name(void *p, void *q) {\
  int x = ((ty_int*)p)->val, y = ((ty_int*)send0(q, "to_i"))->val; \
  return (cmp) ?  &int_1_val : &int_0_val; }
icmp(int_gt, x>y)
icmp(int_lt, x<y)
icmp(int_ge, x>=y)
icmp(int_le, x<=y)
icmp(int_eq, x==y)
icmp(int_ne, x!=y)
char *int_names[] = {"to_i", "to_s", "print", "println", "add", "sub",
  "mul", "div", "mod", "gt", "lt", "ge", "le", "eq", "ne"};
int int_nargs[] = {0,0,0,0,1,1,1,1,1,1,1,1,1,1,1};
void* (*int_funcs[])() = {int_to_i, int_to_s, int_print, int_println,
  int_add, int_sub, int_mul, int_div, int_mod, int_gt, int_lt,
  int_ge, int_le, int_eq, int_ne};
void *int_new(int v) {
  ty_int *p = (ty_int*)malloc(sizeof(ty_int));
  p->tag = TAG_INT; p->nmethods = sizeof(int_nargs)/sizeof(int);
  p->funcs = int_funcs; p->names = int_names; p->nargs = int_nargs;
```

```
  p->val = v;
  return (void*)p;
}
ty_int int_1_val = {
  TAG_INT, sizeof(int_nargs)/sizeof(int), int_funcs, int_names, int_nargs, 1
};
ty_int int_0_val = {
  TAG_INT, sizeof(int_nargs)/sizeof(int), int_funcs, int_names, int_nargs, 0
};
```

メソッドテーブルはこれまでと同じです (だいぶメソッド数が多くなりましたが)。そして int_new も文字列のときと同じです。そのあと、前述の 1 と 0 のオブジェクトがあります。

では、これらの値を取り扱ってメソッドを起動する C プログラムの側を見てみましょう。まず配列 names の中を指定されたメソッド名と一致するものを探します (見つからなければエラー終了)。見つかったら配列 funcs の対応位置にアドレスが入っている関数を呼び出します。send0、send1 とも最初のパラメタはオブジェクト自身で、send1 はさらにもう 1 つパラメタを渡します。今回は簡単のため、渡せるパラメタの個数は 0 か 1 にしたので、これで十分です。

```
void *send0(void *p1, char *name) {
  ty_base *p = (ty_base*)p1;
  for(int i = 0; i < p->nmethods; ++i) {
    if(strcmp(p->names[i], name) == 0) {
      if(p->nargs[i] == 0) { return (p->funcs[i])(p1); }
      fprintf(stderr, "tag %d, name %s, wrong args: 0\n", p->tag, name);
      exit(1);
    }
  }
  fprintf(stderr, "tag %d, name %s, undefined method\n", p->tag, name);
  exit(1);
}
void *send1(void *p1, char *name, void *q1) {
  ty_base *p = (ty_base*)p1;
  for(int i = 0; i < p->nmethods; ++i) {
    if(strcmp(p->names[i], name) == 0) {
      if(p->nargs[i] == 1) { return (p->funcs[i])(p1, q1); }
      fprintf(stderr, "tag %d, name %s, wrong args: 1\n", p->tag, name);
      exit(1);
    }
  }
  fprintf(stderr, "tag %d, name %s, undefined method\n", p->tag, name);
  exit(1);
}
int tst(void *p1) {
  ty_base *p = (ty_base*)p1;
  if(p->tag == TAG_NIL) { return 0; }
  if(p->tag == TAG_INT) { return ((ty_int*)p)->val; }
  return 1;
```

```
}
```

最後の tst は、渡されたオブジェクトが nil なら偽、整数なら 0 が偽、それ以外は真、他のオブジェクトはすべて真として扱い、0 か 1 を返します。

では、これらのライブラリを呼び出してプログラムを実行する C 言語のプログラムを作ってみます。「整数を入力」をプロンプトとして用意し、文字列を入力して整数に変換し n に入れます。続いて i は n - 1 とし、while ループで n > 1 である間繰り返し、もし n % i == 0 なら「素数でない」と出力して終わります。そうでなければ i を 1 減らします。ループを抜けて来たら「素数です」と表示して終わります。

```
#include "objlib.h"

int main(void) {
  void* s = str_new("input an integer> ");
  void* n = send0(send0(s, "prompt"), "to_i");
  void* i = send1(n, "sub", &int_1_val);
  while(tst(send1(i, "gt", &int_1_val))) {
    if(tst(send1(send1(n, "mod", i), "eq", &int_0_val))) {
      send0(str_new("not a prime number."), "println");
      return 0;
    }
    i = send1(i, "sub", &int_1_val);
  }
  send0(str_new("a prime number."), "println");
  return 0;
}
```

このように、値の操作はすべてオブジェクトと send$N$ で構成していますが、制御構造は C の制御構造を使います。実行のようすは次のようになります。

```
% gcc test3.c objlib.c
% ./a.out
input an integer> 97
a prime number.
% ./a.out
input an integer> 99
not a prime number.
%
```

演習 1 例題をそのまま動かせ。動いたら (ライブラリのオブジェクト群を使用することは前提として) 次のことをしてみよ。

a. 上の例題で使っていないメソッドを使った例題を作って動かせ。

b. 例題とは異なる枝分かれやループを含むプログラムを作ってみよ。

c. 自分でもクラスを何か 1 つ定義してみよ (大変)。

d. 配列がないと不便なので、配列に相当するクラスを作ってみよ。パラメタが 1 個なので「put は最後に get した位置に格納する」とかして 1 個で済ますか、send2 を実装してパラメタの最大を 2 個に増やすなどする (とても大変)。

13.3 小さなオブジェクト指向言語

13.3.1 文法記述

それでは、上のライブラリを使って小さなオブジェクトを実現してみます。コンパイラの出力は C 言語のコードで、obj.c というファイルに出力されます。これと先のライブラリを一緒にコンパイルすると動くプログラムになります。

では文法記述から。演算等はすべてメソッド呼び出しなので演算子は言語に含まれていません (ただし後の演習問題を参照)。前述のように簡単にするため、メソッドのパラメタは 0 個か 1 個だけにしています。

```
Package samd1;

Helpers
  digit = ['0'..'9'] ;
  lcase = ['a'..'z'] ;
  ucase = ['A'..'Z'] ;
  letter = lcase | ucase | '_' ;
  graph = [' '..'~'] ;
  nodq = [graph - '"'] ;

Tokens
  sconst = '"' (nodq | '\' graph)* '"' ;
  iconst = digit+ ;
  blank = (' '|13|10)+ ;
  klass = 'class' ;
  method = 'method' ;
  end = 'end' ;
  var = 'var' ;
  nil = 'nil' ;
  new = 'new' ;
  if = 'if' ;
  while = 'while' ;
  semi = ';' ;
  comma = ',' ;
  dot = '.' ;
  assign = '=' ;
  lbra = '{' ;
  rbra = '}' ;
  lpar = '(' ;
  rpar = ')' ;
  ident = letter (letter|digit)* ;

Ignored Tokens
  blank;

Productions
```

```
prog = {class} cls prog
     | {empty}
     ;
cls  = {one} klass ident vars meths end
     ;
vars = {one} ident vars
     | {empty}
     ;
meths = {one} meth meths
      | {empty}
      ;
meth = {para} method ident lpar [para]:ident rpar lbra stlist rbra
     | {none} method ident lbra stlist rbra
     ;
stlist = {stat} stlist stat
       | {empty}
       ;
stat = {assign}  ident assign expr semi
     | {expr}    expr semi
     | {if}      if lpar expr rpar stat
     | {while}   while lpar expr rpar stat
     | {block}   lbra stlist rbra
     ;
expr = {fact} fact
     | {send0} expr dot ident
     | {send1} expr dot ident lpar [arg]:expr rpar
     ;
fact = {iconst} iconst
     | {sconst} sconst
     | {nil}    nil
     | {new}   new ident
     | {ident}  ident
     | {one}    lpar expr rpar
     ;
```

13.3.2 コンパイラドライバ

次はコンパイラのメインを見ます。動かし方ですが、コンパイラにはファイル名に加えて「最初に動かし始めるべきクラスとメソッド」を指定します。

記号表はこれまでとはまったく違う機能なので ObjSymtab というクラスを作りました。構文解析が終わったら、記号表を作り、まずクラス群をすべて登録します。これは ObjChecker というビジタークラスで行ないます。登録後に記号表の概要を表示しています。

続いて、各メソッドの中の名前参照をチェックします。パスが分けてあるのは、ソースプログラム上で先の方にあるクラスでも参照できるようにするためなのでした。この処理は VarChecker というビジタークラスで行ないます。ここまででエラーがあればコード生成せずに終わります。

エラーが無ければ、C 言語ヘッダファイル obj.h と C 言語ソースファイル obj.c を生成する準備をします。ヘッダファイル側は記号表のメソッド emithdr を呼ぶことで必要なヘッダ用定義を出力し

ます。

```
package samd1;
import samd1.parser.*;
import samd1.lexer.*;
import samd1.node.*;
import java.io.*;
import java.util.*;

public class SamD1 {
  public static void main(String[] args) throws Exception {
    if(args.length != 3) {
      Log.pError("usage: SamD1 srcfile startclass startmethod.");
      System.exit(1);
    }
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
        1024)));
    Start tree = p.parse();
    ObjSymtab st = new ObjSymtab();
    ObjChecker ck = new ObjChecker(st); tree.apply(ck); st.show();
    VarChecker vc = new VarChecker(st); tree.apply(vc);
    if(Log.getError() > 0) { return; }
    PrintStream hp = new PrintStream("obj.h"); st.emithdr(hp); hp.close();
    PrintStream pr = new PrintStream("obj.c");
    pr.println("#include <stdlib.h>");
    pr.println("#include \"objlib.h\"");
    pr.println("#include \"obj.h\"");
    Generator gen = new Generator(st, pr); tree.apply(gen);
    st.emitbody(pr, args[1], args[2]); pr.close();
  }
}
```

クラス Log は変更していませんが一応掲載します。

```
package samd1;
public class Log {
  public static int err = 0;
  public static void pError(String s) { System.out.println(s); ++err; }
  public static int getError() { return err; }
}
```

13.3.3 記号表

次は記号表ですが、今度の言語は「クラスとその中のメソッド」という 2 段階になっているのでこれまでとは全く構造が違い、作り直しています (ただし、非常に簡素化してあります)。

定数と変数から説明しましょう。まず、クラスごとに固有の ID を持たせますが、その最初は 4 からとします。次に、変数の種類としては「インスタンス変数」「メソッドの結果用変数」「ローカル変数」の 3 種類を扱います。ただし簡単のため、ローカル変数はパラメタだけで、通常の変数はすべて

インスタンス変数です。そして、メソッド名と同じ名前の変数が 1 つ用意されていて、返値を設定するにはこの変数に代入します。これがメソッドの結果変数です。

クラスに関する情報は内部のクラス ClassData が扱うので、主要なデータ構造は名前文字列から ClassData オブジェクトを検索できる表 cmap です。そして、現在処理中のクラスを変数 cur に保持します。また、現在のメソッドのパラメタ名実行を開始するクラスとそのメソッドのパラメタ名を pnm、メソッド名を mnm に保持します。

```
package samd1;
import java.util.*;
import java.io.*;

public class ObjSymtab {
  public static int newid = 4;
  public static int IVAR = 1, MVAR = 2, LVAR = 3, UNDEF = 0;
  public Map<String,ClassData> cmap = new TreeMap<String,ClassData>();
  public ClassData cur = null;
  String pnm = null, mnm = null;
  public void enterClass(String n) {
    if(!cmap.containsKey(n)) { cmap.put(n, new ClassData(n)); }
    cur = cmap.get(n);
  }
  public void addIvar(String n) {
    if(!cur.vset.contains(n)) { cur.vset.add(n); return; }
    Log.pError(cur.name + ": dupl var " + n);
  }
  public void addMethod(String m, String p) {
    if(cur.mmap.containsKey(m)) { Log.pError(cur.name+": dupl method "+m); }
    else { cur.mmap.put(m, p); }
  }
  public void addMethod(String m) {
    if(cur.mmap.containsKey(m)) { Log.pError(cur.name+": dupl method "+m); }
    else { cur.mmap.put(m, null); }
  }
  public void enterMethod(String m, String p) { mnm = m; pnm = p; }
  public void enterMethod(String m) { mnm = m; pnm = null; }
  public int vkind(String n) {
    if(pnm != null && pnm.equals(n)) { return LVAR; }
    if(mnm != null && mnm.equals(n)) { return MVAR; }
    if(cur.vset.contains(n)) { return IVAR; }
    return UNDEF;
  }
  public boolean isClass(String n) { return cmap.containsKey(n); }
  public void show() {
    for(String n: cmap.keySet()) { cmap.get(n).show(); }
  }
  public void emithdr(PrintStream pr) {
    for(String n: cmap.keySet()) { cmap.get(n).emithdr(pr); }
```

```
  }
  public void emitbody(PrintStream pr, String scls, String smeth) {
    ClassData c = cmap.get(scls);
    if(c == null || !c.mmap.containsKey(smeth) || c.mmap.get(smeth) != null) {
      Log.pError("invalid start class "+scls+" or method "+smeth); return;
    }
    for(String n: cmap.keySet()) { cmap.get(n).emitbody(pr); }
    pr.println("int main(void) {");
    pr.printf(" send0(%s_new(), \"%s\"); return 0; }\n", scls, smeth);
  }
```

enterClass() はクラスに入るときに呼ばれ、そのクラスの ClassData が無ければインスタンスを生成して cmap に登録し、cur にそのクラスの ClassData オブジェクトを保持さます。

addIvar() はインスタンス変数を追加します。実際には classData オブジェクトが保持するデータ構造に登録するだけです。2 重定義ならエラーとします。

addMethod() はメソッドを追加しますが、これも ClassData オブジェクトが保持するデータ構造に追加します。enterMethod() は 2 パス目以降でメソッドに入る時に呼び、パラメタ文字列 (または null)、メソッド名を pnm、mnm に保持します。

isClass() は渡された文字列がクラス名として登録されているか否かを調べて返します。show()、emithdr() はいずれも各クラスについて show()、emithdr() を呼ぶだけです。emitbody() は実行開始クラス/メソッド名が正しいかチェックしたあと、各クラスについて emitbody() を呼び、最後に「開始クラスのインスタンスを生成して開始メソッドを呼ぶ」だけの C 言語の main() を生成します。

ClassData は基本的にそのクラスのメソッド群を保持する mmap とインスタンス変数群を保持する vset を持つことが主な役割です。これらのインスタンス変数は直接外から (といっても ObjSymtab の中だけですが) 参照できます。

show() はこのクラスの名前、インスタンス変数のリスト、そして各メソッドの名前と (あれば) パラメタを順次表示します。emithdr() はこのクラスに対応する構造体定義を出力します。ほとんどはどのクラスでも同じで、クラス名の部分とインスタンス変数のところが違うだけです。

```
  static class ClassData {
    public String name;
    public int id = newid++;
    public Map<String,String> mmap = new TreeMap<String,String>();
    public Set<String> vset = new TreeSet<String>();
    public ClassData(String n) { name = n; }
    public void show() {
      System.out.println(name + ":");
      for(String v: vset) { System.out.print(" " + v); }
      System.out.println();
      for(String k: mmap.keySet()) {
        String v = mmap.get(k);
        if(v == null) { System.out.printf(" %s()\n", k); }
        else          { System.out.printf(" %s(%s)\n", k, v); }
      }
    }
    public void emithdr(PrintStream pr) {
      pr.println("typedef struct {");
```

```
        pr.println(" int tag, nmethods; void* (**funcs)();");
        pr.println(" char* *names; int *nargs;");
        for(String v: vset) { pr.println(" void *_"+v+";"); }
        pr.println("} ty_"+name+";");
        pr.println("void *"+name+"_new();");
      }
      public void emitbody(PrintStream pr) {
        String n = name;
        pr.printf("char *%s_names[] = {", n);
        for(String m: mmap.keySet()) { pr.printf("\"%s\",", m); }
        pr.println("};");
        pr.printf("int %s_nargs[] = {", n);
        for(String m: mmap.keySet()) { pr.print(mmap.get(m) == null ? "0,":"1,"); }
        pr.println("};");
        pr.printf("void* (*%s_funcs[])() = {", n);
        for(String m: mmap.keySet()) { pr.printf("%s_%s,", n, m); }
        pr.println("};");
        pr.printf("void *%s_new() {\n", n);
        pr.printf(" ty_%s *p = (ty_%s*)malloc(sizeof(ty_%s));\n", n, n, n);
        pr.printf(" p->tag=%d; p->nmethods=sizeof(%s_names)/sizeof(int);\n", id, n);
        pr.printf(" p->funcs=%s_funcs; p->names=%s_names; p->nargs=%s_nargs;\n",
                  n, n, n);
        for(String v: vset) { pr.printf(" p->_%s=&nil_val;\n", v); }
        pr.println(" return (void*)p; }");
      }
    }
  } // ObjSymtab の終わり
```

emitbody() はまず前半でメソッドテーブルの 3 つの配列を出力します。そして後半でこのクラスのインスタンスを生成するメソッドの定義を出力します。構造はどのクラスでもほとんど同じですが、インスタンス変数群をすべて nil に初期化するところはクラスごとに違っています。

13.3.4 意味解析

意味解析はそれ自体が 2 パスになっています (クラスについて前方参照したいため)。1 パス目を実行するのが ObjChecker で、クラスの入口でクラスオブジェクトを用意し、インスタンス変数宣言ごとに記号表にインスタンス変数を追加し、またメソッドの入口でメソッド名と (あれば) パラメタ名を登録します。

```
package samd1;
import samd1.analysis.*;
import samd1.node.*;

public class ObjChecker extends DepthFirstAdapter {
  ObjSymtab st;
  public ObjChecker(ObjSymtab st1) { st = st1; }
  @Override
  public void inAOneCls(AOneCls node) {
```

```
    st.enterClass(node.getIdent().getText());
  }
  @Override
  public void outAOneVars(AOneVars node) {
    st.addIvar(node.getIdent().getText());
  }
  @Override
  public void inAParaMeth(AParaMeth node) {
    st.addMethod(node.getIdent().getText(), node.getPara().getText());
  }
  @Override
  public void inANoneMeth(ANoneMeth node) {
    st.addMethod(node.getIdent().getText());
  }
}
```

2パス目はVarCheckerが実行しますが、クラスの入口やメソッドの入口では記号表をそのクラス/メソッドの状態に設定するためにenterClass()、enterMethod()を呼びます。実際のチェックは代入文のところで左辺の変数があることを確認すること、変数名が現れたときにその変数があることを確認すること、そして「new クラス名」が現れたときにそのクラスが存在することを確認することです。

```
package samd1;
import samd1.analysis.*;
import samd1.node.*;

public class VarChecker extends DepthFirstAdapter {
  ObjSymtab st;
  public VarChecker(ObjSymtab st1) { st = st1; }
  @Override
  public void inAOneCls(AOneCls node) {
    st.enterClass(node.getIdent().getText());
  }
  @Override
  public void inAParaMeth(AParaMeth node) {
    st.enterMethod(node.getIdent().getText(), node.getPara().getText());
  }
  @Override
  public void inANoneMeth(ANoneMeth node) {
    st.enterMethod(node.getIdent().getText());
  }
  @Override
  public void outAAssignStat(AAssignStat node) {
    if(st.vkind(node.getIdent().getText()) != ObjSymtab.UNDEF) { return; }
    Log.pError("undefined var: " + node.getIdent().getText());
  }
  @Override
```

```
    public void outAIdentFact(AIdentFact node) {
      if(st.vkind(node.getIdent().getText()) != ObjSymtab.UNDEF) { return; }
      Log.pError("undefined var: " + node.getIdent().getText());
    }
    @Override
    public void outANewFact(ANewFact node) {
      if(st.isClass(node.getIdent().getText())) { return; }
      Log.pError("undefined class in new: " + node.getIdent().getText());
    }
  }
```

13.3.5 コード生成

では最後にコード生成です。方針として、式はその式に対応する C 言語の式のコードを文字列として構成し、文から上では 1 行ずつコードを出力しています。まずメソッドの入口ではメソッドの定義とオブジェクトの型へのキャストを生成し、結果変数を初期化します。出口では結果変数を値として return します。

代入文は変数の種類によって変数名そのまま、または「self->_」を先頭につけた形にし、右辺は式の文字列をそのまま埋め込んで C の代入を生成します。if や while は入口で条件部をまず文字列として取得し、それを用いて C の if/while の先頭を生成してから、本体部分を生成し、最後に「}」を生成します。

```
  package samd1;
  import samd1.analysis.*;
  import samd1.node.*;
  import java.io.*;

  public class Generator extends DepthFirstAdapter {
    ObjSymtab st;
    PrintStream pr;
    String cname, mname, pname;
    public Generator(ObjSymtab s, PrintStream p) { st = s; pr = p; }
    @Override
    public void inAOneCls(AOneCls node) {
      cname = node.getIdent().getText(); st.enterClass(cname);
    }
    @Override
    public void inAParaMeth(AParaMeth node) {
      pname = node.getPara().getText(); mname = node.getIdent().getText();
      pr.printf("void *%s_%s(void *_self, void *%s) {\n", cname, mname, pname);
      pr.printf(" ty_%s *self = (ty_%s*)_self;\n", cname, cname);
      pr.printf(" void *%s = &nil_val;\n", mname);
    }
    @Override
    public void outAParaMeth(AParaMeth node) {
      pr.printf(" return %s; }\n", mname);
    }
    @Override
```

```
public void inANoneMeth(ANoneMeth node) {
  pname = ""; mname = node.getIdent().getText();
  pr.printf("void *%s_%s(void *_self) {\n", cname, mname);
  pr.printf(" ty_%s *self = (ty_%s*)_self;\n", cname, cname);
  pr.printf(" void *%s = &nil_val;\n", mname);
}
@Override
public void outANoneMeth(ANoneMeth node) {
  pr.printf(" return %s; }\n", mname);
}
@Override
public void outAAssignStat(AAssignStat node) {
  String v = node.getIdent().getText(), e = (String)getOut(node.getExpr());
  if(st.vkind(v) == ObjSymtab.IVAR) { v = "self->_" + v; }
  pr.printf(" %s = %s;\n", v, e);
}
@Override
public void outAExprStat(AExprStat node) {
  pr.printf(" %s;\n", ((String)getOut(node.getExpr())));
}
@Override
public void caseAIfStat(AIfStat node) {
  node.getExpr().apply(this); String e = (String)getOut(node.getExpr());
  pr.printf(" if(tst(%e)) {\n", e);
  node.getStat().apply(this);
  pr.println(" }");
}
@Override
public void caseAWhileStat(AWhileStat node) {
  node.getExpr().apply(this); String e = (String)getOut(node.getExpr());
  pr.printf(" while(tst(%s)) {\n", e);
  node.getStat().apply(this);
  pr.println(" }");
}
@Override
public void outAFactExpr(AFactExpr node) {
  setOut(node, getOut(node.getFact()));
}
@Override
public void outASend0Expr(ASend0Expr node) {
  setOut(node, String.format("send0(%s, \"%s\")",
          (String)getOut(node.getExpr()), node.getIdent().getText()));
}
@Override
public void outASend1Expr(ASend1Expr node) {
  setOut(node, String.format("send1(%s, \"%s\", %s)",
```

```
            (String)getOut(node.getExpr()), node.getIdent().getText(),
            (String)getOut(node.getArg())));
  }
  @Override
  public void outAIconstFact(AIconstFact node) {
    setOut(node, String.format("int_new(%s)", node.getIconst().getText()));
  }
  @Override
  public void outASconstFact(ASconstFact node) {
    setOut(node, String.format("str_new(%s)", node.getSconst().getText()));
  }
  @Override
  public void outANilFact(ANilFact node) {
    setOut(node, "&nil_val");
  }
  @Override
  public void outANewFact(ANewFact node) {
    setOut(node, String.format("%s_new()", node.getIdent().getText()));
  }
  @Override
  public void outAIdentFact(AIdentFact node) {
    String v = node.getIdent().getText();
    if(st.vkind(v) == ObjSymtab.IVAR) { v = "self->_" + v; }
    setOut(node, v);
  }
  @Override
  public void outAOneFact(AOneFact node) {
    setOut(node, getOut(node.getExpr()));
  }
}
```

式から先は send$N$(...) やオブジェクトの生成を C コードの文字列として構築していくだけです。

13.3.6 実行例と生成コード

では小さなオブジェクト指向言語による例題コードを示します。まず、myclass というクラスは put された値を次々に加算し、get で現在の累計が取得できるというものです。そして main クラスの start で実行を開始しますが、これはプロンプトを出して 0 が入力されるまで次々に上記の put を呼んで値を累計し、最後に合計を出力します。

```
class main
  acc v
  method start {
    acc = new myclass;
    v = "enter number or '0'> ".prompt.to_i;
    while(v.ne(0)) {
      acc.put(v);
      v = "enter number or '0'> ".prompt.to_i;
```

```
    }
    "total = ".add(acc.get).println;
  }
end
class myclass
  mem
  method put(v) { mem = v.add(mem); }
  method get { get = mem; }
end
```

動かす様子を示します。

```
% java samd1/SamD1 test.obj main start
main:
 acc v
 start()
myclass:
 mem
 get()
 put(v)
% gcc obj.c objlib.c
% ./a.out
enter number or '0'> 3
enter number or '0'> 5
enter number or '0'> 7
enter number or '0'> 0
total = 15
%
```

参考までに生成されたコードを見てみましょう。まずヘッダですが、構造体の定義と型名の定義、そして生成メソッドのプロトタイプ宣言をクラスごとに出力します。構造体の中身はタグとメソッド表までは同じですが、インスタンス変数についてはクラス毎に違います。

```
typedef struct {
 int tag, nmethods; void* (**funcs)();
 char* *names; int *nargs;
 void *_acc;
 void *_v;
} ty_main;
void *main_new();
typedef struct {
 int tag, nmethods; void* (**funcs)();
 char* *names; int *nargs;
 void *_mem;
} ty_myclass;
void *myclass_new();
```

次にコードですが、まず各クラスのインスタンスメソッドがすべて出力され、その後でクラス毎にメソッドテーブルの配列定義とオブジェクトを生成する関数が生成されます。

```
#include <stdlib.h>
#include "objlib.h"
#include "obj.h"
void *main_start(void *_self) {
 ty_main *self = (ty_main*)_self;
 void *start = &nil_val;
 self->_acc = myclass_new();
 self->_v = send0(send0(str_new("enter number or '0'> "), "prompt"), "to_i");
 while(tst(send1(self->_v, "ne", int_new(0)))) {
 send1(self->_acc, "put", self->_v);
 self->_v = send0(send0(str_new("enter number or '0'> "), "prompt"), "to_i");
 }
 send0(send1(str_new("total = "), "add", send0(self->_acc, "get")), "println");
 return start; }
void *myclass_put(void *_self, void *v) {
 ty_myclass *self = (ty_myclass*)_self;
 void *put = &nil_val;
 self->_mem = send1(v, "add", self->_mem);
 return put; }
void *myclass_get(void *_self) {
 ty_myclass *self = (ty_myclass*)_self;
 void *get = &nil_val;
 get = self->_mem;
 return get; }
char *main_names[] = {"start",};
int main_nargs[] = {0,};
void* (*main_funcs[])() = {main_start,};
void *main_new() {
 ty_main *p = (ty_main*)malloc(sizeof(ty_main));
 p->tag=4; p->nmethods=sizeof(main_names)/sizeof(int);
 p->funcs=main_funcs; p->names=main_names; p->nargs=main_nargs;
 p->_acc=&nil_val;
 p->_v=&nil_val;
 return (void*)p; }
char *myclass_names[] = {"get","put",};
int myclass_nargs[] = {0,1,};
void* (*myclass_funcs[])() = {myclass_get,myclass_put,};
void *myclass_new() {
 ty_myclass *p = (ty_myclass*)malloc(sizeof(ty_myclass));
 p->tag=5; p->nmethods=sizeof(myclass_names)/sizeof(int);
 p->funcs=myclass_funcs; p->names=myclass_names; p->nargs=myclass_nargs;
 p->_mem=&nil_val;
 return (void*)p; }
int main(void) {
 send0(main_new(), "start"); return 0; }
```

演習 2 「小さなオブジェクト指向言語」を次のように拡張してみよ。

a. 現在の言語では「`1 + 2`」のような中置記法は使えず、「`1.add(2)`」のようにメッセージ送信記法で書く必要がある。これでは見にくいので、演算子が扱えるように構文を変更してみよ (ヒント: 構文上は中置記法であっても、生成される C 言語コードはこれまでと同様に send$N$ を呼べばよい)。

b. 現在の言語ではそれぞれのクラスは独立している。これを拡張して、継承機構を追加してみよ。クラスに親クラスが指定されている場合は、インスタンス変数とメソッドをその親クラスから取り込んでくる、というのが想定される継承の機能である。

c. その他自分の好きなように言語を拡張してみよ。

13.4 型検査と静的なメソッドテーブル

13.4.1 型のあるオブジェクト指向言語

ここまで見て来た「小さなオブジェクト指向言語」の実装では、メソッドテーブルは「メソッドが定義された順」で並んでいて、実行時にメソッド名 (文字列) をキーとして線形探索で実際に呼ぶメソッドを決定していました。しかしこれでは高速に実行できないのは明らかです。

また、クラス名の未定義は検出されるが、メソッド名の未定義は実行時に呼び出すまで分かりません。これは型検査を行なわないため、どうにもなりません。

Smalltalk-80 など初期のオブジェクト指向言語ではこれらを言語仕様として受け入れていましたが、オブジェクト指向が普及し始めた時期 (1980 年代) に、これらの問題を解消するためにオブジェクト指向言語に型検査を導入しようと、多くの研究がなされました。今日の C++言語や Java 言語はそれらの研究の成果を取り入れて設計されています。ここでは Java を例として説明しましょう。

Java 言語ではすべてのオブジェクトは対応するクラスが型となっています。たとえば `MyClass` のインスタンスであれば型も `MyClass` です。

そして次に、`Object` を除くすべてのクラスは 1 つだけ親クラスを持ちます。クラス定義の冒頭において `extends` キーワードで親クラスを指定しますが、その指定がない場合は `Object` が指定されたものとして扱います。従って、すべてのクラスは直接または間接に `Object` の子孫となります。

ここで型検査の規則は次のようになります。

> クラス T, T' において、T が T' の子孫であることを $T < T'$ と表す。変数の型が T' のとき、その変数に代入できる値は $T \leq T'$ なる T 型の値に限られる。

この規則の「根拠」ですが、継承関係にある場合、子クラスは親クラスからメソッドを引き継ぐため、親クラス T' が持つメソッドの情報に基づいて型検査を行なった場合、子クラス T のオブジェクトは同じメソッド群を持つので正しく実行できる、というところにあります。

また、メソッドを親クラスからコピーしてきてそのまま使えるかという点については、子クラスのインスタンスは親クラスのインスタンス変数をすべて持ち、さらに独自のインスタンス変数を追加する、という設計であるため、配置を工夫すれば問題なく扱えます。この点は次節で具体的に説明します。

13.4.2 単一継承向けのインスタンスとメソッドテーブル設計

具体例があった方が分かりやすいので、「図形クラスと、そのサブクラスとしての円と長方形」という例を示します。図形クラスは抽象クラスで、両方の図形に共通のインスタンス変数・メソッド群を持ちます。そして円と長方形は図形の子クラスで、それぞれ独自のインスタンス変数とメソッドを追加しています。また、メソッド `draw()` は抽象メソッドで、図形クラスでは宣言のみされていて、円と長方形で具体的な定義がなされています。

```
abstract class Figure {
  int xpos, ypos;
  Color col;
  public Figure(int x, int y, Color c) { ... 初期化 ... }
  public void moveTo(int x, int y) { xpos = x; ypos = y; }
  public void setColor(Color c) { col = c; }
  public abstract void draw(Graphics g);
}
class Circle extends Figure {
  int rad;
  public Circle(int x, int y, int r) { ... 初期化 ... }
  public void draw(Graphics g) { ... 円の描画 ... }
  public void scale(float r) { rad = (int)(rad * r); }
}
class Rect extends Figure {
  int with, height;
  public Rect(int x, int y, int w, int h) { ... 初期化 ... }
  public void draw(Graphics g) { ... 長方形の描画 ... }
  public void rot90() { int z = w; w = h; h = z; }
}
```

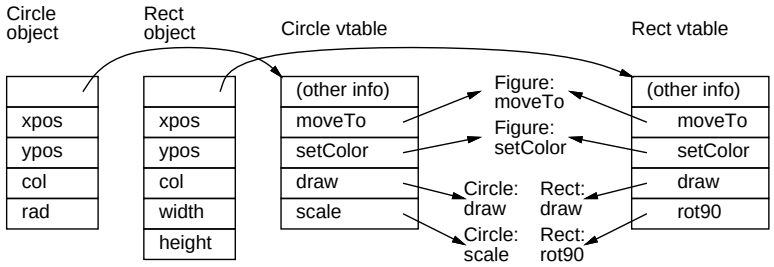

図 13.3: 単一継承むけの静的な配置

このような単一継承の体系では、図 13.3 のように、どの子クラスのオブジェクトも、親クラスの部分の「後ろに」自分独自のインスタンス変数の領域を確保する配置とすることで、親クラスの部分はすべて共通に保つことができます。

そして、各オブジェクトは自クラスのメソッドテーブルへのポインタを持つようにしますが、それぞれのクラスのメソッドテーブルもインスタンスと同様、親クラスのメソッド群をまず配置し、その後ろに独自のメソッドのスロットを配置します。

このようにしておけば、たとえば Figure:moveTo は Circle に対しても Rect に対しても呼び出されますが、インスタンス変数としてアクセスする部分は Figure で定義されているものだけなので、後ろの方が違っていても感知しません。また、draw はどちらのクラスのメソッドテーブルでも 3 スロット目と共通の位置にありますが、実際に呼ばれるものは Circle と Rect で別のものです。そしてその下には、それぞれのクラス独自のメソッドが配置されます。

これにより、インスタンス変数のアクセスもメソッドの呼び出しもオブジェクトやメソッドテーブルの先頭から固定オフセットにコンパイルでき、効率のよい実行が行なえます。

この方法は、親クラスが 1 つだけという「単一継承」の設計だから可能になるものであり、親が複数ある多重継承ではうまく行きません。多重継承の場合は、実行時に探索を行なうか、もっと複雑な構造を取り入れる必要がありますが、ここでは触れません。

演習 3 静的な型検査とここで説明したメソッドテーブルを持つオブジェクト指向言語を実装してみなさい。

13.5 課題 13A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 13」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 強い型の言語と弱い型の言語のどちらが好みですか。またそれはなぜ。
  Q2. 記号表と型検査の実装について学んでみて、どのように思いましたか。
  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

システムソフトウェア特論　2019