2021 年度 Q2 情報科学概論 演習
処理系の実装【草稿】*

横山 哲郎

目次

* 最終更新日: 2020/07/31

1 はじめに

本演習の目的は，これまで座学で学んできたプログラミング言語理論やソフトウェア開発技術の応用力を養成しようというものである．応用ということで，今まで学んできた知識や技術を組み合わせるという能力やソフトウェア開発プロジェクトをグループで遂行するという能力が必要とされる．実際に「もの」を作るという楽しみもあろう．是非，挑戦的なプロジェクト課題に各グループで力を合わせて取り組んで欲しい．

本演習では，プログラミング言語処理系の理論を実装の課題を通して実践する．課題はプロジェクト形式である．プロジェクトは，2 名のグループで行うことを基本とする．実装を行う言語やツールは各自で自由に選択してもらって構わない．推奨する言語は Haskell, OCaml, Java, C 言語である．C 言語はプログラミング基礎などで学んでいると思うが，木構造を操作するために言語レベルのサポートがないのでデバッグに苦労するかもしれない．他には，C++, C#, OCaml などでもよい．言語によって字句解析器や構文解析器などの適切なツールは異なってくる．本授業の課題では，BNFC（http://bnfc.digitalgrammars.com/）を用いて生成された字句解析器や構文解析器が用意されていることがある．Haskell, Java, C 言語を用いる場合は，字句解析器生成器と構文解析器生成器は，以下のものの知識があると実装が効果的に進められるであろう．

- Haskell
  - Alex: A lexical analyser generator for Haskell, http://www.haskell.org/alex/
  - Happy: The Parser Generator for Haskell, http://www.haskell.org/happy/
- OCaml: http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html, http://ocaml.jp/refman/ch12.html (日本語)
  - ocamllex
  - ocamlyacc
- Java (bnfc ver.2.8.2 以降)
  - ANTLRv4: a parser generator for Java, https://www.antlr.org/
- Java (bnfc ver.2.8 以降)
  - JFlex: a lexical analyzer generator (also known as scanner generator) for Java, written in Java, https://jflex.de/
- Java (bnfc ver.2.5.0 以降)
  - JLex: A Lexical Analyzer Generator for Java(TM), http://www.cs.princeton.edu/~appel/modern/java/JLex/
  - CUP: LALR Parser Generator in Java, http://www2.cs.tum.edu/projects/cup/
- C 言語
  - flex: The Fast Lexical Analyzer, http://flex.sourceforge.net/
  - bison: GNU parser generator, http://www.gnu.org/software/bison/

解釈系の実装は，それぞれが選択した言語で行う．Haskell の場合は状態モナド，Java の場合は Visitor パターン，C 言語の場合は共用体を用いたコードが現れるので，あらかじめこれらの知識があると良い．Visitor パターンの概要を短時間で知るには，入門書 [1, pp. 189–208] の「第 6 部構造を渡り歩く」の「第 13 章

Visitor—構造を渡り歩きながら仕事をする」の解説がおすすめである．

生成するコードは Java バイトコードを用いる．Java 仮想機械のためのアセンブラとして Jasmin (http://jasmin.sourceforge.net/) を用いる．

興味のある受講生のために参考文献を挙げておく．構造的操作的意味論に関する入門書として文献 [2] が読みやすい．プログラミング言語の実装については [3] が入門的である．（この書籍の元となったオンラインの講義資料 [4] も公開されている．）この文献でも BNFC を用いられているので参考になるであろう．本演習では命令型言語の処理系を実装するが，言語の種類が変わるとその実現も大きく異なる．他の種類の言語の実現に興味のある学生に対して紹介するならば，例えば，関数型言語の実装について詳しい [5] をあげておく．また，絶版ではあるが，日本語では [6] も入門書としておすすめである．

文献 [7] では，（加減乗除を含む）算術式言語の評価器を Haskell および C 言語で実装した例が掲載されている．

2 準備

本演習を行う前にグループに分かれてもらう．プロジェクト課題はグループごとに課されるので，同一のグループのメンバーは同じ環境を選択することをおすすめしたい．

2.1 必要となる前提知識

本節では，授業内で特に説明なく使われる前提知識について述べる．

■ホームディレクトリ Unix 系の OS では，個々のユーザのホームディレクトリが ~（チルダ）で表される．環境変数$HOME は，個々のユーザのホームディレクトリを保持するのに使われる．例えば，macOS では，yamad というユーザの環境変数$HOME は以下のようになる．

```
$ echo $HOME
/Users/yamada
```

ホームディレクトリをカレントディレクトリにするには以下のいずれかのコマンドを実行すれば良い．

```
$ cd ~
$ cd $HOME
$ cd
```

■コンパイル方法 コンパイルのやり方は各自で自習すること．例えば，Java では，パッケージを javac というコマンドでコンパイルを行う．さらにコンパイルなどの作業を自動化するために Makefile を用いる．

問題 1. ディレクトリ P の下にあるパッケージ P の A.java をコンパイルするコマンドと，その中の main メソッドを実行するコマンドは何か．

解答. 以下の通り：

```
$ javac P/A.java
$ java P/A
```

■知っておくべき単語

- 環境変数
- クラスパス（Java を使う場合）

2.2 グループ分けと環境の選択

1. 2 名ずつのグループを作る．特に希望が無ければ学生番号順に先頭から 2 人ずつのグループ分けをする．あまる人が出た場合は，1 グループだけ 3 名で構成するようにする．
2. 実装言語を選択する．
   Haskell, Java, C の順で推奨する．
3. 環境を整備する．
   前項の言語における開発環境，BNFC, 字句解析器，構文解析器，Jasmin[*1]をインストールする．
   Windows 環境の注意　GNU make なども用意する必要がある．Cygwin(http://www.cygwin.com/)を入れると良いようである．
   macOS 環境の注意　本稿のコードは，Java コンパイラのバージョン javac 1.8.0_101 でチェックされている．これと同じか新しいバージョンをインストールすると良い．

2.3 alex/happy のインストール【Haskell の場合】

Ubuntu の場合

```
sudo apt install alex happy
```

2.4 BNFC のインストール

本稿のコードは，BNFC 2.8.1 でチェックされている．

■cabal を使っている場合

```
cabal install bnfc
```

■mac の場合

```
brew install bnfc
```

■Ubuntu の場合

```
sudo apt install bnfc
```

[*1] Jasmin は，自作するコンパイラのコードを実行するのに用いる．

Mac でソースコードからコンパイルする場合

ソースコードを http://bnfc.digitalgrammars.com/ からダウンロードする（http://bnfc.digitalgrammars.com/download/BNFC-2.8.2.tar.gz で動作確認済み）.

```
tar xvzf BNFC-2.8.2.tar.gz
cd BNFC-2.8.2
make
cp dist/build/bnfc/bnfc ~/local/bin/bnfc-2.8.2-mac
cd ~/local/bin/
ln -s bnfc-2.8.2-mac bnfc
```

バイナリをダウンロードする場合

- バイナリを http://bnfc.digitalgrammars.com/ からダウンロード
  例えば，`$HOME/local/bin/`に置く.
- ファイル名を変更
  - Windows の場合：bnfc-2.8-win.exe から bnfc.exe
- 実行フラグを立てる
  - Windows の場合：不要のようである
  - Mac の場合：`chmod +x bnfc`
- パスを通す（Mac でパッケージ版 (.pkg) からインストールしたときは不要）
  - Mac で bash の場合：`export PATH="$HOME/local/bin":"$PATH"` を `.bash_profile` などに追加.
  - Windows で cygwin で bash の場合：`export PATH="$HOME/local/bin":"$PATH"` を `.bash_profile` などに追加.
- 【オプション】PATH の設定を，bash を使用している場合は`.bash_profile` に，zsh を使用している場合は`.zshrc` に書き込んでおく.

その他の方法

以下の方法でもうまくいった:

- Haskell Platform をインストールする．Homebrew がインストールされている場合は，Cask を用いて，コマンドラインから以下を実行することでインストールが完了する:[*2]
  `brew install haskell-platform`
- `cabal install cabal cabal-install` を実行する.
- `cabal install alex happy bnfc` を実行する.
- `export PATH="$HOME/Library/Haskell/bin":"$PATH"`を実行する.
- 【次回以降のシェル起動時に PATH の設定を自動的に行う】
  `export PATH="$HOME/Library/Haskell/bin":"$PATH"`を`.bashrc` などに追加する．なお，

[*2] かつては `brew cask install haskell-platform` というように cask をいれる必要があった.

export PATH="~/Library/Haskell/bin":"$PATH" ではチルダが展開されない．

2.5 JDK, CUP, JLex, ANTLRv4 のインストール【Java の場合】

bnfc ver.2.5.0 を使用する場合，JDK, CUP ver.0.10k, JLex ver.1.2.6 をインストールしたときの動作が確認されている．bnfc ver.2.8.2 を使用する場合，JDK と以下のいずれかをインストールしたときの動作が確認されている．

- CUP ver.0.11b と (JLex ver.1.2.6 または JFlex ver.1.7.0)
- ANTLRv4 ver.4.7.2

Java のインストール

最新の安定版の JDK をインストールする．

CUP と JLex のインストール

- CUP ver.0.11b と JLex ver.1.2.6 を利用する場合は，https://www-p.st.nanzan-u.ac.jp/faculty/tetsuo/sw_ensyu/2016/javatools.tgz からダウンロード
- 解凍して javatools 以下にあるディレクトリ Cup と JLex を$HOME/local/java/に置く．
- CLASSPATH に$HOME/local/java/Cup と$HOME/local/java を追加する．
  - Mac/Ubuntu の場合：
    export CLASSPATH=.:"$HOME/local/java/Cup":"$HOME/local/java"
    をコマンドラインから実行する．
- 【次回以降のシェル起動時に CLASSPATH の設定を自動的に行う】
  CLASSPATH の設定を，ホームディレクトリにある .bashrc(bash を使用している場合) や.zshrc(zsh を使用している場合) の最後などに書き込んでおく．
  export CLASSPATH=.:"$HOME/local/java/Cup":"$HOME/local/java"
  不慣れな人は，環境変数やクラスパスについて調べること．さらに，https://www.cs.princeton.edu/~appel/modern/java/JLex/current/sample.lex からサンプルファイルをダウンロードして以下のように動作したら恐らく JLex のインストールが成功

```
$ wget https://www.cs.princeton.edu/~appel/modern/java/JLex/current/sample.lex
$ java JLex.Main sample.lex
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 185 states.
Working on character classes.::::::::::::::::.....::..::.....:.........::............::..::.
NFA has 29 distinct character classes.
Creating DFA transition table.
Working on DFA states.............................................................................
Minimizing DFA transition table.
47 states after removal of redundant states.
```

```
Outputting lexical analyzer code.
```

- 【確認】以下のように表示されたら恐らく JLex のインストールが成功

```
$ java JLex.Main
Usage: JLex.Main <filename>
```

- 【確認】以下のようにして止まったら Cup のインストールが多分成功

```
$ java java_cup.Main
```

注意: 2015 年 9 月 15 日現在，公式サイト (http://www2.cs.tum.edu/projects/cup/) から入手できるのは，より新しいバージョンの CUP 0.11b である．

ANTLRv4 のインストール

BNFC は ver.2.8 以降で ANTLRv4 をサポートしている（https://bnfc.readthedocs.io/en/latest/user_guide.html#java）．Mac の場合は以下のようにインストールを行う．

```
brew install antlr
```

以下のように CLASSPATH の設定を行う．バージョンは適切に変更する必要がある．

```
export CLASSPATH=.:/usr/local/Cellar/antlr/4.7.2/antlr-4.7.2-complete.jar
```

2.6 Jasmin のインストール

Jasmin は Java 仮想機械のためのアセンブラである．

Ubuntu の場合

以下でインストールできた．

```
sudo apt install jasmin-sable
```

Mac で Homebrew を使っている場合

コマンドラインで以下のように実行することでインストールが完了する[*3]：

```
brew install jasmin
```

この場合は，PATH の設定を自分でやる必要はない．

それ以外の環境の場合

それ以外の環境の場合は，Jasmin の公式ページ（http://jasmin.sourceforge.net/）からダウンロードしてインストールする．

Mac では，`$HOME/local/java` に `jasmin.jar` を配置して，

```
java -jar $HOME/local/java/jasmin.jar
```

[*3] Mac で Homebrew を使っていない場合は，Homebrew のインストールに時間が掛かるので別の方法をおすすめする。

で実行することができる．スクリプト jasmin に

jasmin

```
#!/bin/bash
exec java  -jar $HOME/local/java/jasmin.jar "$@"
```

などと記述して，パスの通った場所に配置すればよい．

Windows では，例えば，$HOME/local/java に jasmin.jar を配置して，以下のようなバッチファイルをパスの通ったところに配置する．

jasmin.bat

```
java -jar C:\Users\m20se999\Local\java\jasmin.jar %1
```

2.7 GNU make のインストール

macOS の場合

GNU make はインストールされているが，初めて使う場合は Xcode のライセンスに Agree する必要があるようである．

Ubuntu の場合

```
sudo apt install make
```

3 算術式の解釈と翻訳 — Calc

本章では，プログラミング言語を設計して，その解釈器（インタープリタ）と翻訳器（コンパイラ）を実現する．解釈器は，ソースコードを一部ずつ逐次解釈をすることを繰り返しながら実行するプログラムである．翻訳器は，ソースコードをオブジェクトコードに変換するプログラムである．

本章で用いる表記法について説明する．ソースコードはタイプライタ体（例，1+3）で表記する．数学的な対象は数式フォント（例，123）を用いる．集合には太字体（例，**Digit**）を用いる．ソースコードを引数に取る意味関数は二重括弧 $[\![\cdot]\!]$ を用いる．たとえば，$\mathcal{N}[\![\texttt{123}]\!] = 123$ は，ソースコード 123 を引数にとる意味関数 $\mathcal{N}$ が数 123 を返すことを意味する．

3.1 言語の設計

自然数と加算・減算・乗算といった二項演算子から構成される算術式を考える．たとえば，1+2-3*4 は算術式である．本節では，算術式のみからなる簡単なプログラミング言語 Calc を設計する．まずは，プログラミング言語の構文と意味論を定める．

構文にはいくつかの定め方がある．本稿では，まず集合の上で帰納的に定義する方法を述べる．次に，より簡潔な BNF(Backus-Naur form) を用いる方法を述べる．

3.1.1 帰納的定義で定める構文

10 進表記の一桁の自然数を表す記号からなる集合

$$\mathbf{Digit} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\} \tag{1}$$

を考える．Digit は英語で一桁のアラビア数字のことを指す．**Digit** の要素は単なる記号であり，数の集合ではない．たとえば，タイプライタ体で書かれた 3 は数 3 ではない．**Digit** の要素がアラビア数字である必然性はない．後に関連づける数の類推がしやすいために用いているに過ぎない．類推しにくくてもいいのであれば，たとえば，**Digit** $=$ { 壱, 弐, . . .} でも **Digit** $=$ { ひとつ, ふたつ, . . .} でも構わない．

次に二桁以上の自然数を考慮に入れる．10 進表記の自然数を表す記号列からなる集合 **Num** を考える．非形式的には，**Num** $= \{\mathtt{0}, \mathtt{1}, \ldots\}$ や

$$\{d_n \ldots d_2 d_1 \mid d_i \in \mathbf{Digit}, 1 \leq i \leq n, n \geq 1\} \tag{2}$$

と表現できる．しかし，この 3 点リーダ「. . .」による省略部分の意味が曖昧である．そこで，3 点リーダを用いずに帰納的定義を用いる．すなわち，**Num** は次を満たす最小の集合である[*4]:

$$\mathbf{Digit} \subseteq \mathbf{Num} \tag{3}$$

$$n \in \mathbf{Num} \wedge d \in \mathbf{Digit} \Longrightarrow nd \in \mathbf{Num} \tag{4}$$

なぜ「最小」という条件がついているのであろうか．**Num** が，式 3 と式 4 を満たすという条件のみである場合，「壱」という記号が集合 **Num** に含まれていても構わない．こうした余分な記号が **Num** に含まれないように「最小」という条件がついているのである．

同様にして帰納的定義を用いて算術式の集合 **Aexp** を定める．**Aexp** は，算術式を表す英単語Arithmetic expressions を省略した名前である．自然数を表す記号 n は算術式であるとする:

$$n \in \mathbf{Num} \implies n \in \mathbf{Aexp} \tag{5}$$

たとえば，31 は自然数を表す記号であるので算術式である．すなわち，31 $\in$ **Num** であるので 31 $\in$ **Aexp** である．

ふたつの算術式の加算式は

> a_1 および a_2 が算術式であるならば，a_1+a_2 は算術式である

という規則から構成する．この規則を言い換えると，

$$a_1, a_2 \in \mathbf{Aexp} \implies a_1\mathtt{+}a_2 \in \mathbf{Aexp} \tag{6}$$

である．たとえば，1 や 2 は算術式であるので，1+2 は算術式である．ここでメタ変数 a_0 と a_1 は，任意の算術式を表すのに用いられている．「メタ」とは，「超えて」「高次の」といった意味を表す接頭語である．ここでは，（そのようなものは定義していないが）算術式に出現する変数ではなく，算術式を記述するときに使われている変数という意味で，メタ変数という用語を使っている．減算式や乗算式も同様の規則によって定まっているものとする．

[*4] 定義を簡単にするために，ここでは 013 のような二桁以上で 0 から始まる記号列も **Num** に含んでいる．

以上の規則をまとめる．算術式は規則

$$n \in \mathbf{Num} \implies n \in \mathbf{Aexp} \tag{7}$$
$$a_1, a_2 \in \mathbf{Aexp} \implies a_1\texttt{+}a_2 \in \mathbf{Aexp} \tag{8}$$
$$a_1, a_2 \in \mathbf{Aexp} \implies a_1\texttt{-}a_2 \in \mathbf{Aexp} \tag{9}$$
$$a_1, a_2 \in \mathbf{Aexp} \implies a_1\texttt{*}a_2 \in \mathbf{Aexp} \tag{10}$$

からのみ定められるものとする．なぜ「からのみ」と限定したのであろうか．これは **Num** を定義するときに「最小」という条件をつけたのと同様に余分な要素が **Aexp** に含まれないようにするためである．たとえば，1+2*3 は算術式であることは次のように確かめられる．式 1 より，$1, 2, 3 \in \mathbf{Digit}$ である．よって，式 3 より $1, 2, 3 \in \mathbf{Num}$ である．さらに式 7 より，$1, 2, 3 \in \mathbf{Aexp}$ である．したがって，式 8 と 10 より，$1\texttt{+}2 \in \mathbf{Aexp}$ であり，$1\texttt{+}2\texttt{*}3 \in \mathbf{Aexp}$ である．

このように帰納的定義によって構文を形式的に定めることができるが，現在，論文やテキストではこの方法を目にすることはあまりない．これは帰納的定義をより簡潔に記述することができる BNF が広く使われているためである．

3.1.2 BNF で定める構文

構文を定めるときに BNF(Backus-Naur form) やそれを一部改変した記法が良く用いられる．ここではこれらを総称して BNF と呼ぶ．BNF を用いると上記のような帰納的定義を簡潔に記述することができる．たとえば，前小節の 10 進表記の自然数を表す記号列は

$$d ::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$
$$n ::= d \mid nd$$

と記述することができる．“::=” は “can be”，“|” は “or” と読むことができる．すなわち，これらの構文規則は

- d can be 0 or 1 or ... or 9.
- n can be d or n followed by d.
- d と n はこれ以外の値をとることはない．

ということを表している．

問題 2. n が 23 であることはあるか．あるとしたらなぜか．また，n が「壱」であることはあるか．ないとしたらなぜか．

BNF を用いると，前小節の算術式の帰納的定義は

$$a ::= n \mid a_1\texttt{+}a_2 \mid a_1\texttt{-}a_2 \mid a_1\texttt{*}a_2$$

と表される．

この構文規則は，記号列 2+3*4 を (2+3)*4 と 2+(3*4) のどちらの構文に解釈すべきかについて何も述べていないことに注意して欲しい．この構文規則は，記号列の解釈を与えているのではなく，算術式の木構造，すなわち**抽象構文木**を定めているのである．

3.1.3 Calc の構文

プログラミング言語 Calc には自然数を表すための記号列が有限個あるものとする：

$$n ::= \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{10} \mid \texttt{11} \mid \cdots \mid \texttt{2147483647}$$

なお，$2147483647 = 2^{31} - 1$ である[*5]．Calc の算術式は，構文規則

$$a ::= n \mid a_1\texttt{+}a_2 \mid a_1\texttt{-}a_2 \mid a_1\texttt{*}a_2 \mid \texttt{(}a_1\texttt{)}$$

から構成されるもののみとする．ここで構文の中に括弧 () が含まれていることに注意してほしい．プログラミング言語 Calc の実装を簡単にするため，ここでは構文解析をして得られる**具象構文木**を定めているのである．

問題 3. 次の文字列の中からプログラミング言語 Calc の算術式として適当なものを選びなさい．

- `1`
- `1+2`
- `-1`
- `1` （←数式フォント）
- `a1+a2`
- `1+(2-3)`
- `((1))`
- `1+2-3`

問題 4. `1+2-3` および `1+2*3` をそれぞれ 2 種類の構文木で表しなさい．

3.1.4 Calc の意味論

記号列を数という抽象的な対象に対応づけるのに意味関数を用いる．意味関数 $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$ は

$$\begin{aligned} \mathcal{N}[\![\texttt{0}]\!] &= 0 \\ \mathcal{N}[\![\texttt{1}]\!] &= 1 \\ &\vdots \\ \mathcal{N}[\![\texttt{2147483647}]\!] &= 2147483647 \end{aligned}$$

というように，記号を数に対応づける[*6]．

問題 5. 意味関数 $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$ は，全域関数であることを確認しなさい．

次のメタ変数は，それぞれの集合の範囲の値をもつものとする:

$$\begin{aligned} n &\in \mathbf{Num} \\ a &\in \mathbf{Aexp} \\ v &\in \mathbb{Z} \end{aligned}$$

[*5] この制限は，言語処理系の実装を容易にするために入れてある．

[*6] $\mathbb{Z}$ は，整数の集合．

ここでメタ変数 v は値を表す英単語 value の頭文字から命名している．

算術式とその整数値の関係は

$$a \to v$$

と表す．この関係は，算術式 a が整数値 v に評価される，と読む．この関係が成立するかは以下の意味規則により定める．意味規則を定めることで，例えば，`1+3` $\to 4$ は成立するとか，`1-3` $\to 4$ は成立しないとかを議論できるようになる．

意味規則 NUM: 任意の n ($\in$ **Num**) と任意の整数 v ($\in \mathbb{Z}$) に対して，$\mathcal{N}[\![n]\!] = v$ ならば $n \to v$ である．

意味規則 PLUS: 任意の算術式 a_1, a_2 ($\in$ **Aexp**) と任意の v_1, v_2 ($\in \mathbb{Z}$) に対して，$a_1 \to v_1$, $a_2 \to v_2$, かつ $v = v_1 + v_2$ ならば a_1`+`$a_2 \to v$ である．

意味規則 MINUS: 任意の算術式 a_1, a_2 ($\in$ **Aexp**) と任意の v_1, v_2 ($\in \mathbb{Z}$) に対して，$a_1 \to v_1$, $a_2 \to v_2$, かつ $v = v_1 - v_2$ ならば a_1`-`$a_2 \to v$ である．

意味規則 MULT: 任意の算術式 a_1, a_2 ($\in$ **Aexp**) と任意の v_1, v_2 ($\in \mathbb{Z}$) に対して，$a_1 \to v_1$, $a_2 \to v_2$, かつ $v = v_1 \cdot v_2$ ならば a_1`*`$a_2 \to v$ である．

意味規則 PAREN: 任意の算術式 a_1 ($\in$ **Aexp**) と任意の v_1 ($\in \mathbb{Z}$) に対して，$a_1 \to v_1$ ならば `(`a_1`)` $\to v$ である．

意味規則は，意味規則名，前提，結論から構成される．「○○ならば××」の○○が前提で××が結論である．意味規則は前提が満たされる場合に限り結論を導くことができる．規則を繰り返し適用して結論を得る過程を**導出**と呼ぶ．

問題 6. 以下の関係が成り立つことを確認しなさい:

- `2017` $\to 2017$
- `2+3` $\to 5$
- `(2+3)*(4-6)` $\to -10$

解答例．最後の関係のみ示す．以下のステップにより `(2+3)*(4-6)` $\to -10$ は成り立つ:

ステップ 1 意味規則 NUM より，`2` $\to 2$ である．
ステップ 2 意味規則 NUM より，`3` $\to 3$ である．
ステップ 3 意味規則 PLUS，ステップ 1 と 2 の結論より，`2+3` $\to 5$ である．
ステップ 4 意味規則 NUM より，`4` $\to 4$ である．
ステップ 5 意味規則 NUM より，`6` $\to 6$ である．
ステップ 6 意味規則 PLUS，ステップ 4 と 5 の結論より，`4-6` $\to -2$ である．
ステップ 7 意味規則 PAREN とステップ 3 の結論より，`(2+3)` $\to 5$ である．
ステップ 8 意味規則 PAREN とステップ 6 の結論より，`(4-6)` $\to -2$ である．
ステップ 9 意味規則 MULT，ステップ 7 と 8 の結論より，`(2+3)*(4-6)` $\to -10$ である．

□

このように意味規則と導出を日本語で記述するのは煩雑であり一目で理解しにくいので，次のような省略記

法を用いるのが一般的である:

$$\mathrm{NUM}\frac{}{n \to \mathcal{N}[\![n]\!]} \tag{11}$$

$$\mathrm{PLUS}\frac{a_1 \to v_1 \qquad a_2 \to v_2}{a_1\texttt{+}a_2 \to v_1 + v_2} \tag{12}$$

$$\mathrm{MINUS}\frac{a_1 \to v_1 \qquad a_2 \to v_2}{a_1\texttt{-}a_2 \to v_1 - v_2} \tag{13}$$

$$\mathrm{MULT}\frac{a_1 \to v_1 \qquad a_2 \to v_2}{a_1\texttt{*}a_2 \to v_1 \cdot v_2} \tag{14}$$

$$\mathrm{PAREN}\frac{a_1 \to v_1}{\texttt{(}a_1\texttt{)} \to v_1} \tag{15}$$

一般に,「意味規則 X: $\mathcal{J}_1$ かつ $\cdots$ かつ $\mathcal{J}_n$ ならば $\mathcal{J}_0$」は

$$\mathrm{X}\frac{\mathcal{J}_1 \ \cdots \ \mathcal{J}_n}{\mathcal{J}_0} \tag{16}$$

という形で示すことにする．導出を次のように再帰的に定義する．関係 $\mathcal{J}_i$ を結論とする導出 $\mathcal{D}_i$ $(i = 1, \ldots, n)$ が得られていて，意味規則 X 中のメタ変数を具体化して,

$$\mathrm{X}\frac{\mathcal{J}_1 \ \cdots \ \mathcal{J}_n}{\mathcal{J}_0} \tag{17}$$

が得られたとすると,

$$\mathrm{X}\frac{\mathcal{D}_1 \ \cdots \ \mathcal{D}_n}{\mathcal{J}_0} \tag{18}$$

は，$\mathcal{J}_0$ を結論とする導出である．導出は，関係を節点とする木と見なすことができるので，**導出木**と呼ぶ．例えば,

$$\mathrm{NUM}\frac{}{\texttt{2} \to \texttt{2}}$$

や

$$\mathrm{PLUS}\frac{\mathrm{NUM}\dfrac{}{\texttt{2} \to \texttt{2}} \qquad \mathrm{NUM}\dfrac{}{\texttt{3} \to \texttt{3}}}{\texttt{2+3} \to \texttt{5}}$$

は導出木である．

問題 7. 上の 2 つの導出木が，確かに導出木であることを定義を当てはめて考えてみなさい．

算術式の評価は**導出木**によって行うことができる．例えば,

$$\mathrm{MULT}\frac{\mathrm{PAREN}\dfrac{\mathrm{PLUS}\dfrac{\mathrm{NUM}\dfrac{}{\texttt{2} \to \texttt{2}} \quad \mathrm{NUM}\dfrac{}{\texttt{3} \to \texttt{3}}}{\texttt{2+3} \to \texttt{5}}}{\texttt{(2+3)} \to \texttt{5}} \qquad \mathrm{PAREN}\dfrac{\mathrm{MINUS}\dfrac{\mathrm{NUM}\dfrac{}{\texttt{4} \to \texttt{4}} \quad \mathrm{NUM}\dfrac{}{\texttt{6} \to \texttt{6}}}{\texttt{4-6} \to \texttt{-2}}}{\texttt{(4-6)} \to \texttt{-2}}}{\texttt{(2+3)*(4-6)} \to \texttt{-10}}$$

という導出木が得られるので，式 `(2+3)*(4-6)` は整数値 -10 に評価される．

以上のように，高水準言語である Calc の意味を具体的な計算機を仮定せずに形式的に定めることができた．

■ノート 本章で扱った意味論は，操作的意味論と呼ばれ，表示的意味論と公理的意味論などと共によく使われる．特に，情報科学概論の張先生の担当において公理的意味論の有名な例であるホーア論理が取り上げられる．

- 初学者の人が本章の内容についてさらに学びたい場合は，文献 [8] をお薦めしたい．
- 大学院レベルの発展的な内容は，文献 [9] を参考にして欲しい．

- 「2016 年度 ソフトウェア工学演習 前半」の第 3 章を元にした.
  https://www-p.st.nanzan-u.ac.jp/faculty/tetsuo/sw_ensyu/2016/lec_note/lecnote_softsem_2016.pdf

3.2 字句解析器と構文解析器の実現

本節では BNFC によってプログラミング言語 Calc の字句解析器（レクサ，スキャナ）と構文解析器（パーサ）の実現を行う方法を説明する.

ほぼ BNF の記法にしたがったファイル `Calc.cf` を用意する[*7]：

Calc.cf

```
EAdd. Exp  ::= Exp  "+" Exp1 ;
ESub. Exp  ::= Exp  "-" Exp1 ;
EMul. Exp1 ::= Exp1 "*" Exp2 ;
EInt. Exp2 ::= Integer ;

coercions Exp 2 ;
```

BNFC を用いて Haskell, Java, C, C, C#, OCaml で字句解析器と構文解析器の実現をするには，それぞれ

```
bnfc -m -haskell Calc.cf
bnfc -m -java    Calc.cf
bnfc -m -c       Calc.cf
bnfc -m -cpp     Calc.cf
bnfc -m -csharp  Calc.cf
bnfc -m -ocaml   Calc.cf
```

と打鍵する．ここで，オプション`-m` の指定により `Makefile` が生成される.

3.2.1 GNU make について

カレントフォルダに生成された `Makefile` には，GNU make で用いるルールが記述されている．コマンド

```
make
```

を打鍵すると，`Makefile` に記述された一番上のルールが実行されて，処理系のコンパイルが行われる．ここで

```
make clean
```

と打鍵すると，ターゲット clean に記述されたコマンドが実行されて，コンパイルで生成されたファイルが削除される.

GNU make の公式ページは http://www.gnu.org/software/make/ である．少し古いバージョンの GNU make は，有志によって日本語訳がされている（http://www.ecoop.net/coop/translated/

[*7] 以降の作業は作業用フォルダで行うと良い．たとえば，`Calc.cf` を`$HOME/sw_jisyu` というような作業用フォルダを作成してそのフォルダの中に `Calc.cf` を作成すると良い.

GNUMake3.77/make_toc.jp.html).

3.2.2 Java の場合

JLex で構文解析器を生成して，CUP で字句解析器を生成して，それらをソースコードに追加して，Java コンパイラである javac で解釈器のテンプレートをコンパイルし，コンパイル結果を実行する．これらを行うコマンドは Make ファイルに記述されているので，実際に打鍵するコマンドは

```
$ make
```

のみである．

以下のような実行結果が出力されたら成功である：

```
$ echo "1 + 3 * (4 - 2)" | java Calc/Test

Parse Succesful!

[Abstract Syntax]

(EAdd (EInt 1) (EMul (EInt 3) (ESub (EInt 4) (EInt 2))))

[Linearized Tree]

1 + 3 * (4 - 2)
```

3.2.3 Haskell の場合

Happy で構文解析器を生成して，Alex で字句解析器を生成して，GHC で解釈器のテンプレートをコンパイルして，コンパイル結果を実行する：

```
$ happy -gca ParCalc.y
$ alex -g LexCalc.x
$ ghc --make TestCalc.hs -o TestCalc
$ echo "1 + 3 * (4 - 2)" | ./TestCalc
echo "1 + 3 * (4 - 2)" | ./TestCalc

Parse Successful!

[Abstract Syntax]

EAdd (EInt 1) (EMul (EInt 3) (ESub (EInt 4) (EInt 2)))

[Linearized tree]

1 + 3 * (4 - 2)
```

3.2.4 OCaml の場合

ocamllex で字句解析器を生成して，ocamlyacc で構文解析器を生成して，ocamlc でコンパイルして，コンパイル結果を実行する．これらを行うコマンドは Make ファイルに記述されているので，実際に打鍵するコマンドは

```
make
```

のみである．場合によっては生成されたコマンドに実行権限を付与する必要がある：

```
chmod +x ./TestCalc
```

以下のような実行結果が出力されたら成功である：

```
$ echo "1 + 3 * (4 - 2)" | ./TestCalc
[Abstract syntax]

EAdd (EInt (1), EMul (EInt (3), ESub (EInt (4), EInt (2))))

[Linearized tree]

1 + 3 * (4 - 2)
```

3.3 解釈器の実現

3.3.1 Java の場合

図 1 のような Visitor パターンを用いる．Exp クラスは，「訪問者」を受け入れる accept メソッドと，「訪問者」を表す Visitor インタフェースをメンバとしてもつ．抽象クラスである Exp クラスは，加算式，減算式，乗算式，整数値をそれぞれ表すクラスによって継承される．「訪問」を受けたときの振る舞いはそれぞれの継承クラスの accept による．「訪問者」は，Calculator クラスの多相メソッド visit によって訪問する継承クラスによって振る舞いを変える．

参考までに，クラス図の読み方を簡単に述べる．クラスやインタフェースは長方形で表される．長方形は水平線で 3 つに分割される．上から順にクラス名やインタフェース名などの情報，フィールドの情報，メソッドの情報が書かれる:

クラス名やインタフェース名などの情報
フィールドの情報
メソッドの情報

抽象クラスの名前は斜体で書かれる．たとえば，抽象クラスである Exp の名前が斜体で書かれている．抽象メソッドの名前は斜体で書かれる．たとえば，Exp クラスの accept メソッドの名前が斜体で書かれている．静的メソッドには下線が引かれる．たとえば，Interpret クラスの main メソッドや Interpreter クラスの interpret メソッドには下線が引かれている．フィールド名やメソッド名の前に書かれる「+」は，そのメソッドが public であることを表す．インタフェース名の上には，明示的に << interface >> というステレオタイ

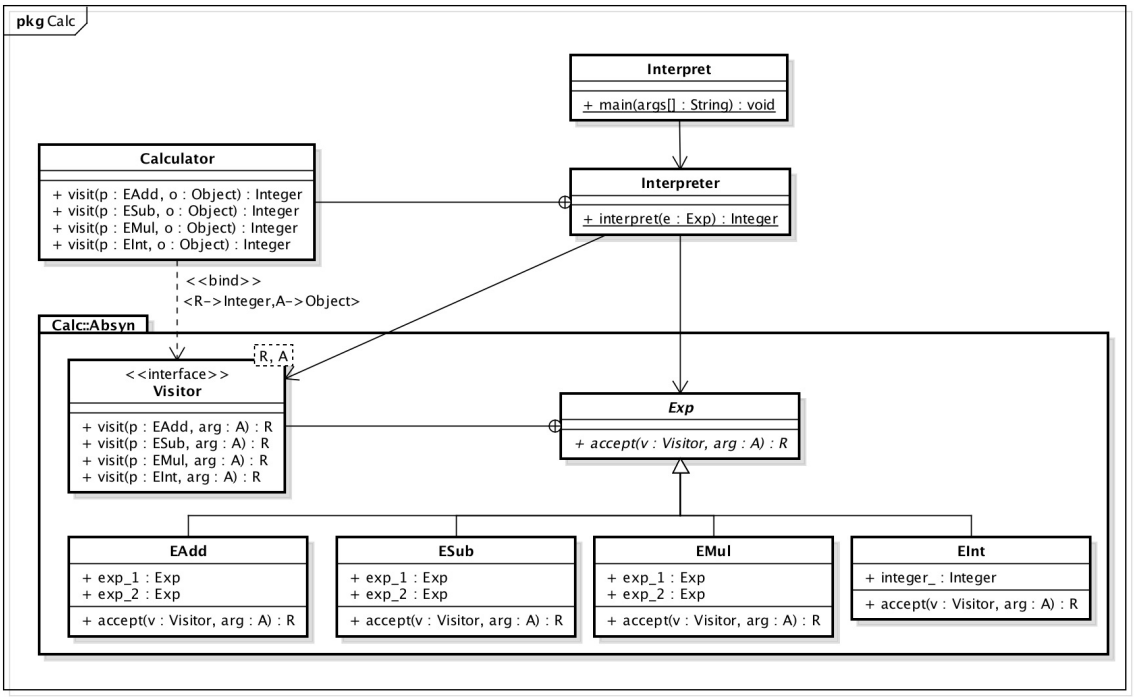

図 1　Calc のクラス図

プが書かれる．引数や返却値のクラスが総称型になることでテンプレート化（パラメトライズド）されたインタフェースは右上の破線で囲まれた箱の中に総称型が列挙される．総称型の束縛は，実現関係を表す矢印の付近に << bind >> と書いてその下に束縛関係を列挙する．たとえば，Visitor クラスは総称型 R, A でパラメータ化されて Calculator クラスで実現するときに R が Integer に A が Object に束縛されている．クラスのメンバとしてクラスやインタフェースがあることは矢印の先が $\oplus$ である関連によって表される．たとえば，Interpreter クラスは Calculator クラスをメンバにもつ．この場合，Calculator クラスは Interpreter クラスで宣言されたもので，ネストしたクラスである．

以下，コードの詳細をみていく．Exp クラスでは算術式の抽象構文木の構成と木のノードへの「訪問」を扱う:

Calc/Absyn/Exp.java

```
package Calc.Absyn; // Java Package generated by the BNF Converter.

public abstract class Exp implements java.io.Serializable {
  public abstract <R,A> R accept(Exp.Visitor<R,A> v, A arg);
  public interface Visitor <R,A> {
    public R visit(Calc.Absyn.EAdd p, A arg);
    public R visit(Calc.Absyn.ESub p, A arg);
    public R visit(Calc.Absyn.EMul p, A arg);
    public R visit(Calc.Absyn.EInt p, A arg);

  }

}
```

「訪問」を受け入れるためのメソッド accept と「訪問先」のクラスの種類によって振る舞いを変える多相メソッド visit は，Exp を継承したクラス（後述）で実装される．

Calc の処理は，ラッパークラスである Interpret の main メソッドから始まる:

Interpret.java

```
package Calc;
import java_cup.runtime.*;
import Calc.*;
import Calc.Absyn.*;
import java.io.*;

public class Interpret {
    public static void main(String args[]) throws Exception {
        Yylex l  = new Yylex(System.in);
        parser p = new parser(l) ;
        Calc.Absyn.Exp parse_tree = p.pExp() ;
        System.out.println(Interpreter.interpret(parse_tree)) ;
    }
}
```

標準入力 System.in から得られた文字列を字句解析し l を得る．この結果に構文解析を行って構文木 parse_tree を得る．この構文木を Interpreter クラスの interpret メソッドに渡す．

Interpreter クラスの interpret メソッドでは,「訪問」を受け取った Exp クラスに accept メソッドによって受け入れてもらう．Calc の解釈器は整数値を返すのでパラメータ R には Integer を用いる．また，引数には情報を渡す必要がないので Null オブジェクトを渡す．「訪問者」がどのような振る舞いをするかは Calculator クラスの多相メソッド visit で書く．たとえば，引数が EAdd クラスであった場合，その二つの子どもの式に「訪問」を accept メソッドで受け入れてもらって得られた Integer の結果の和を返却値とする．また，たとえば，引数が EInt クラスであった場合，（新たに「訪問」行うことなく）もっている整数値を返却値とする．accept メソッドが呼び出されているところでは，必ず第一引数に Calculator のオブジェクト自身が渡されていることに注意してほしい．以下でみるが，受け入れてもらった先で Calculator クラスの visit が再度呼ばれるようになっている．

Interpreter.java

```
package Calc;
import Calc.Absyn.*;

public class Interpreter {
    public static Integer interpret(Exp e) {
        return e.accept(new Calculator(), null) ;
    }

    private static class Calculator implements Exp.Visitor<Integer,Object> {
        public Integer visit(Calc.Absyn.EAdd p,Object o) {
            Integer a = p.exp_1.accept(this, null);
            Integer b = p.exp_2.accept(this, null);
            return a + b;
        }
        public Integer visit(Calc.Absyn.ESub p,Object o) {
            Integer a = p.exp_1.accept(this, null);
            Integer b = p.exp_2.accept(this, null);
            return a - b;
        }
        public Integer visit(Calc.Absyn.EMul p,Object o) {
            Integer a = p.exp_1.accept(this, null);
            Integer b = p.exp_2.accept(this, null);
            return a * b;
        }
        public Integer visit(Calc.Absyn.EInt p, Object o) {
            return p.integer_;
        }
    }
}
```

Exp クラスを継承したクラスでは,「訪問者」を受け入れる抽象メソッド accept が実装される．accept の中では，visit メソッドにそのクラスのオブジェクト自身（すなわち Exp クラスのサブクラス）と引数が渡される．

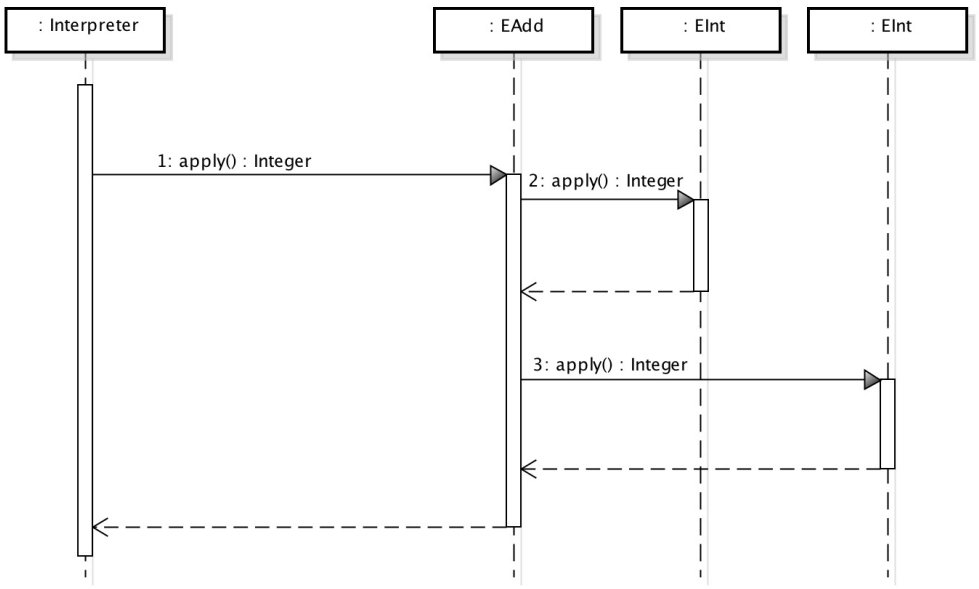

図 2　データ構造と処理を同一のクラス Interpreter で定義したと仮定したときに 2+3 を解釈したときのシーケンス図

Calc/Absyn/EAdd.java

```
package Calc.Absyn; // Java Package generated by the BNF Converter.

public class EAdd extends Exp {
  public final Exp exp_1, exp_2;

  /* Java のコンストラクタのメソッド名には，クラス名を用いる． */
  public EAdd(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }

  public <R,A> R accept(Calc.Absyn.Exp.Visitor<R,A> v, A arg) {
    return v.visit(this, arg);
  }

  public boolean equals(Object o) {
    if (this == o) return true;
    if (o instanceof Calc.Absyn.EAdd) {
      Calc.Absyn.EAdd x = (Calc.Absyn.EAdd)o;
      return this.exp_1.equals(x.exp_1) && this.exp_2.equals(x.exp_2);
    }
    return false;
  }

  public int hashCode() {
    return 37*(this.exp_1.hashCode())+this.exp_2.hashCode();
  }
}
```

シーケンス図を用いて解釈器の処理の流れをみてみる．図 3 は，式 2+3 を理するときの流れを示している．

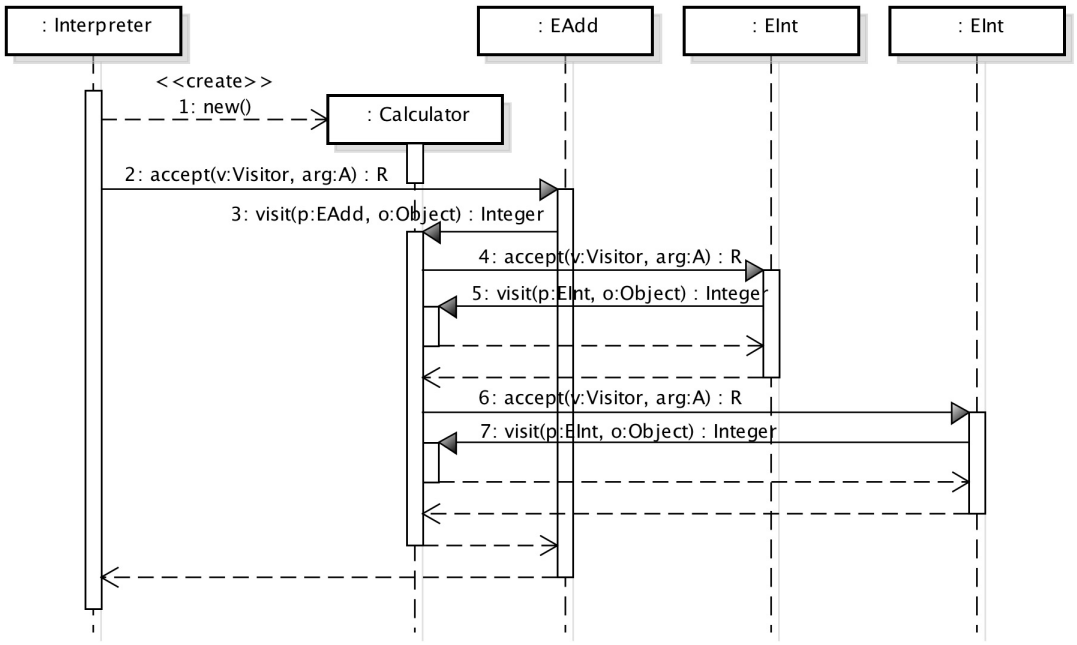

図 3　2+3 を解釈したときのシーケンス図

ただし Interpret のインスタンスから Interpreter が呼ばれ値が返却される部分は省略してある．なお，たとえば，:Interpreter は Interpreter のインスタンスを表す．

1. :Interpreter は:Calculator を作る．（interprete メソッドにおいて `new Calculator()` でこれを行う．）
2. :Interpreter は，:EAdd に対して，accept メソッドを呼び出す．このとき第一引数には 1 で作成した:Calculator を渡す．
3. :EAdd は，渡された:Calculator の visit メソッドを呼び出す．このとき第一引数には:EAdd 自身を渡す．
4. :Calculator は，:EInt に対して，accept メソッドを呼び出す．このとき第一引数には:Calculator 自身を渡す．この:EInt は部分式 2 に対応する左の子どものフィールドである．
5. :EInt は，渡された:Calculator の visit メソッドを呼び出す．このとき第一引数には:EInt 自身を渡す．続いて，整数値 2 が:Calculator → :EInt → :Calculator という順に返却される．
6. 4 と同様に，:Calculator は，:EInt に対して，accept メソッドを呼び出す．
7. 5 と同様に，:EInt は，渡された:Calculator の visit メソッドを呼び出す．:EInt から:Calculator に整数値 3 が返却される．5 で返却された整数値 2 とここで得られた整数値 3 の和 5 が最終的に:Interpreter の返却値になる．

■Java プログラムのコンパイルと実行　Java コンパイラのコマンド名は javac である．パッケージ Calc の中の Interpret.java をコンパイルするには，Calc の含まれたディレクトリにおいて

```
$ javac Calc/Interpret.java
```

とタイプする．

サンプルセッションは以下の通りである：

```
$ echo "1 + 3 * (4 - 2)" | java Calc/Interpret
7
```

3.3.2 Haskell の場合

意味規則を実装する．Haskell の場合は，AbsCalc.hs における代数データ型で抽象構文が表現されている．この抽象構文：

```
data Exp =
   EAdd Exp Exp
 | ESub Exp Exp
 | EMul Exp Exp
 | EInt Integer
```

を再帰的に下降することで意味規則を実装する：

Interpreter.hs

```
module Interpreter where

import AbsCalc

interpret :: Exp -> Integer
interpret x = case x of
  EAdd exp0 exp -> interpret exp0 + interpret exp
  ESub exp0 exp -> interpret exp0 - interpret exp
  EMul exp0 exp -> interpret exp0 * interpret exp
  EInt n  -> n
```

ラッパーとして Main モジュールを用意する：

Interpret.hs

```
module Main where

import LexCalc
import ParCalc
import AbsCalc
import Interpreter

import ErrM

main = do
  interact calc
  putStrLn ""

calc s =
  let Ok e = pExp (myLexer s)
  in show (interpret e)
```

コンパイルをする：

```
$ ghc --make Interpret
```

サンプルセッションは以下の通りである：

```
$ echo "1 + 3 * (4 - 2)" | ./Interpret
7
```

3.3.3 OCaml の場合

意味規則を実装する．OCaml の場合は，AbsCalc.ml における代数データ型で抽象構文が表現されている．この抽象構文：

```
type exp =
   EAdd of exp * exp
 | ESub of exp * exp
 | EMul of exp * exp
 | EInt of int
```

を再帰的に下降することで意味規則を実装する：

Interpreter.ml

```
open AbsCalc

let rec interpret (x : exp) : int = match x with
  | EAdd (e1, e2) -> interpret e1 + interpret e2
  | ESub (e1, e2) -> interpret e1 - interpret e2
  | EMul (e1, e2) -> interpret e1 * interpret e2
  | EInt (n) -> n
```

ラッパーとして Main モジュールを用意する：

Interpret.ml

```
open Lexing

let parse (c : in_channel) : AbsCalc.exp =
  ParCalc.pExp LexCalc.token (Lexing.from_channel c)
;;

let showTree (t : AbsCalc.exp) : string =
  "[Abstract syntax]\n\n"^
    (fun x -> ShowCalc.show (ShowCalc.showExp x)) t^ "\n\n"^
      "[Linearized tree]\n\n"^ PrintCalc.printTree PrintCalc.prtExp t^
        "\n"
;;

let main () =
  let channel =
    if Array.length Sys.argv > 1 then open_in Sys.argv.(1)
    else stdin
  in
  try print_int (Interpreter.interpret (parse channel));
      flush stdout;
      exit 0
  with BNFC_Util.Parse_error (start_pos, end_pos) ->
    Printf.printf "Parse error at %d.%d-%d.%d\n"
      start_pos.pos_lnum (start_pos.pos_cnum - start_pos.pos_bol)
      end_pos.pos_lnum (end_pos.pos_cnum - end_pos.pos_bol);
    exit 1
;;

main ();;
```

コンパイルをする：

```
$ ocamlc  -o Interpret BNFC_Util.ml AbsCalc.ml SkelCalc.ml ShowCalc.ml PrintCalc.ml\
  ParCalc.mli ParCalc.ml LexCalc.ml Interpreter.ml Interpret.ml
```

サンプルセッションは以下の通りである：

```
$ echo "1 + 3 * (4 - 2)" | ./Interpret
7
```

3.3.4 C の場合

構造体と共用体を用いて式を表現する.

Absyn.h

```
#ifndef ABSYN_HEADER
#define ABSYN_HEADER

/* C++ Abstract Syntax Interface generated by the BNF Converter.*/

/********************   TypeDef Section    ********************/
typedef int Integer;
typedef char Char;
typedef double Double;
typedef char* String;
typedef char* Ident;


/********************   Forward Declarations    ********************/

struct Exp_;
typedef struct Exp_ *Exp;


/********************   Abstract Syntax Classes    ********************/

struct Exp_
{
  enum { is_EAdd, is_ESub, is_EMul, is_EInt } kind;
  union
  {
    struct { Exp exp_1, exp_2; } eadd_;
    struct { Exp exp_1, exp_2; } esub_;
    struct { Exp exp_1, exp_2; } emul_;
    struct { Integer integer_; } eint_;
  } u;
};

Exp make_EAdd(Exp p0, Exp p1);
Exp make_ESub(Exp p0, Exp p1);
Exp make_EMul(Exp p0, Exp p1);
Exp make_EInt(Integer p0);

#endif
```

構造体と共用体でつくられた構文を再帰的に下降して，式の評価を行う：

Interpret.h

```
#ifndef INTERPRET_HEADER_FILE
#define INTERPRET_HEADER_FILE

#include "Absyn.h"

int interpret(Exp p);

#endif
```

Interpreter.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Absyn.h"

int interpret(Exp p)
{
    switch (p->kind) {
    case is_EAdd:
        return interpret(p->u.eadd_.exp_1) + interpret(p->u.eadd_.exp_2) ;
    case is_ESub:
        return interpret(p->u.eadd_.exp_1) - interpret(p->u.eadd_.exp_2) ;
    case is_EMul:
        return interpret(p->u.eadd_.exp_1) * interpret(p->u.eadd_.exp_2) ;
    case is_EInt:
        return p->u.eint_.integer_ ;
    }
}
```

Interpret.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "Parser.h"
#include "Printer.h"
#include "Absyn.h"
#include "Interpret.h"

void usage(void) {
  printf("usage: Call with one of the following argument combinations:\n");
  printf("\t--help\t\tDisplay this help message.\n");
  printf("\t(no arguments)        Evaluate the input from stdin.\n");
  printf("\t(files)\t\tEvaluate the content of files.\n");
}

int main(int argc, char ** argv)
{
  FILE *input;
  Exp parse_tree;
  int quiet = 0;
  char *filename = NULL;

  filename = argv[1];
  if (filename) {
    input = fopen(filename, "r");
    if (!input) {
      usage();
      exit(1);
    }
  } else {
      input = stdin;
  }
  parse_tree = pExp(input);
  if (parse_tree) {
      printf("%d\n", interpret(parse_tree));
      return 0;
  } else {
      return 1;
  }
}
```

コンパイルをする：

```
$ gcc -g -W -Wall -c Interpreter.c
$ gcc -g -W -Wall -c Interpret.c
$ gcc -g -W -Wall Absyn.o Lexer.o Parser.o Printer.o Interpreter.o Interpret.o -o Interpret
```

表 1　Java バイトコード

命令	説明
`bipush` n	byte 値 n をプッシュする
`iadd`	2 つの値をポップして，合計をプッシュする
`isub`	2 つの値をポップして，差をプッシュする
`imul`	2 つの値をポップして，積をプッシュする

サンプルセッションは以下の通りである：

```
$ echo "1 + 3 * (4 - 2)" | ./Interpret
7
```

3.4　コード生成

Java バイトコードを生成する解釈器を実現する．表 1 に Calc を翻訳するのに必要な Java バイトコードを示す．

3.4.1　Java の場合

コード生成を行う主要なクラスは以下の通り：

```
public class CodeGenerator {
    public static Object compileExp(Exp e) {
        return e.accept(new ExpCompiler(), null);
    }

    private static class ExpCompiler implements Exp.Visitor<Object,Object> {
        public Object visit(Calc.Absyn.EAdd p, Object arg) {
            p.exp_1.accept(this, arg) ;
            p.exp_2.accept(this, arg) ;
            System.err.println("iadd");
            return null;
        }

        /* ... ここに isub や imul のコードを書き込む ... */

        public Object visit(Calc.Absyn.EInt p, Object arg) {
            System.err.println("bipush " + p.integer_);
            return null;
        }
    }
}
```

クラス `ExpCompiler` の中の `visit` は，後順走査でコード生成を行う．

ラッパーとして CodeGenerate クラスを作成する．(前節で作成した Interpreter のラッパーである Interpret を用意したのと同様)

以下のように出力されたら成功である：

```
$ echo "1 + 3 * (4 - 2)" | java Calc/CodeGenerate
bipush 1
bipush 3
bipush 4
bipush 2
isub
imul
iadd
```

ヘッダとフッタを埋め込んで jasmin でアセンブルすると，以下のように実行できる：

```
$ jasmin Main.j
Generated: Main.class
$ java Main
7
```

ヘッダは

```
.class public Main
.super java/lang/Object
;
; standard initializer
.method public <init>()V
  aload_0
  invokenonvirtual java/lang/Object/<init>()V
  return
.end method

.method public static main([Ljava/lang/String;)V
.limit locals 20
.limit stack 1000 ;
```

であり，フッタは，

```
invokestatic runtime/iprint(I)V

return ;
.end method ;
```

である．この間に CodeGenerator で生成されたコード片を挟み込む．ここでオブジェクト runtime は，

runtime.j

```
.class public runtime
.super java/lang/Object
;
; standard initializer
.method public <init>()V
  aload_0
  invokenonvirtual java/lang/Object/<init>()V
  return
.end method

.method public static iprint(I)V
.limit locals 1
.limit stack 2
  getstatic java/lang/System/out Ljava/io/PrintStream;
  iload_0
  invokevirtual java/io/PrintStream/println(I)V
  return
.end method
```

と定義されている．Main.class の実行前に，runtime.j もコンパイルしておく必要がある．

```
jasmin runtime.j
```

3.4.2 Haskell の場合

コード生成をするための主要な関数は以下の通りである：

```
compileExp :: Exp -> State Env ()
compileExp e = case e of
                 EAdd a b -> do
                        compileExp a
                        compileExp b
                        emit iadd_Instr
                 ...
                 EInt i -> emit (bipush_Instr i)
```

この関数を CodeGenerator.hs に用意する．

CodeGenerator.hs

```
module Main where

import Control.Monad.State
import AbsCalc
import LexCalc
import ParCalc
import AbsCalc

import ErrM

data Env = Env {
      code :: [Instruction]
    }

type Instruction = String

compileExp :: Exp -> State Env ()
compileExp e = case e of
                 EAdd a b -> do
                        compileExp a
                        compileExp b
                        emit iadd_Instr
                 {- ... 以下続く ... -}
                 EInt i -> emit (bipush_Instr i)

iadd_Instr, imul_Instr, isub_Instr :: Instruction
iadd_Instr = "iadd"
isub_Instr = "isub"
imul_Instr = "imul"

bipush_Instr :: Integer -> Instruction
bipush_Instr i = "bipush " ++ show i

emit :: Instruction -> State Env ()
emit i = modify (\s -> s{code = i : code s})

main = do
  interact gen
  putStrLn ""

gen s =
  let Ok e = pExp (myLexer s)
      insts = reverse (code (snd (runState (compileExp e) (Env []))))
  in concat (map (++"\n") insts)
```

コンパイルする：

```
ghc --make CodeGenerator
```

以下のようなバイトコードが出力される：

```
$ echo "1 + 3 * (4 - 2)" | ./CodeGenerator
bipush 1
bipush 3
bipush 4
bipush 2
isub
imul
iadd
```

4　言語の拡張 — Bims

4.1　言語の設計

簡単なプログラミング言語 **Bims**[*8]を設計する．構文領域

$$
\begin{array}{ll}
n \in \mathbf{Num} & \text{数字} \\
x \in \mathbf{Var} & \text{変数} \\
a \in \mathbf{Aexp} & \text{算術式} \\
b \in \mathbf{BExp} & \text{ブール式} \\
S \in \mathbf{Stm} & \text{文}
\end{array}
$$

および構文規則

$$
\begin{array}{lcl}
S & ::= & x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if (}b\texttt{) then \{ }S_1\texttt{ \} else \{ }S_2\texttt{ \}} \mid \texttt{while (}b\texttt{) do \{ }S\texttt{ \}} \mid \texttt{print } x \\
b & ::= & a_1 = a_2 \mid a_1 < a_2 \mid \neg b_1 \mid b_1 \wedge b_2 \mid \texttt{(}b_1\texttt{)} \\
a & ::= & n \mid a_1\texttt{+}a_2 \mid a_1\texttt{*}a_2 \mid a_1\texttt{-}a_2 \mid \texttt{(}a_1\texttt{)} \mid x
\end{array}
$$

から構成される算術式を考える．ただし，$\neg$ は $\wedge$ よりも優先順位が高いとする．すなわち，$\neg b_1 \wedge b_2$ は，$\neg(b_1 \wedge b_2)$ ではなく $(\neg b_1) \wedge b_2$ を表すとする．意味領域には，整数 $\mathbb{Z}$，論理値 $\mathbb{B}(= \{tt, ff\})$，および状態

$$\mathbf{State} = \mathbf{Var} \rightharpoonup \mathbb{Z}$$

を用いる．文 S, 論理式 b, 算術式 a に対しての意味規則がそれぞれある：

$$\langle S, s\rangle \to s' \tag{19}$$

$$s \vdash b \to_b v_b \tag{20}$$

$$s \vdash a \to_a v_a \tag{21}$$

実装では，s' は状態であり，v_b, v_a は整数値であることに注意して欲しい．**Bims** の意味規則を図 4 に示す．ここで，意味関数の型は $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$ である．`print` x は x の値を表示する．

定義 1. s を状態とする．状態の更新 $s[x \mapsto v]$ は

$$
\begin{array}{rcll}
s'(y) & = & s(y) & \text{if } y \neq x \\
 & = & v & \text{if } y = x
\end{array}
$$

となる状態 s' である．

[*8] Basic imperative statements

$$\text{PLUS}\ \frac{s \vdash a_1 \to_a v_1 \quad s \vdash a_2 \to_a v_2}{s \vdash a_1\texttt{+}a_2 \to_a v} \quad \text{where } v = v_1 + v_2$$

$$\text{MINUS}\ \frac{s \vdash a_1 \to_a v_1 \quad s \vdash a_2 \to_a v_2}{s \vdash a_1\texttt{-}a_2 \to_a v} \quad \text{where } v = v_1 - v_2$$

$$\text{MULT}\ \frac{s \vdash a_1 \to_a v_1 \quad s \vdash a_2 \to_a v_2}{s \vdash a_1\texttt{*}a_2 \to_a v} \quad \text{where } v = v_1 \cdot v_2$$

$$\text{PAREN}\ \frac{s \vdash a_1 \to_a v_1}{s \vdash \texttt{(}a_1\texttt{)} \to_a v_1} \qquad \text{NUM}\quad s \vdash n \to_a v \quad \text{if } \mathcal{N}[\![n]\!] = v \qquad \text{VAR}\quad s \vdash x \to_a v \quad \text{if } s(x) = v$$

$$\text{EQUALS-1}\ \frac{s \vdash a_1 \to_a v_1 \quad a_2 \to_a v_2}{s \vdash a_1 = a_2 \to_b \mathit{tt}} \quad \text{if } v_1 = v_2 \qquad \text{EQUALS-2}\ \frac{s \vdash a_1 \to_a v_1 \quad a_2 \to_a v_2}{s \vdash a_1 = a_2 \to_b \mathit{ff}} \quad \text{if } v_1 \neq v_2$$

$$\text{GREATERTHAN-1}\ \frac{s \vdash a_1 \to_a v_1 \quad s \vdash a_2 \to_a v_2}{s \vdash a_1 < a_2 \to_b \mathit{tt}} \quad \text{if } v_1 < v_2$$

$$\text{GREATERTHAN-2}\ \frac{s \vdash a_1 \to_a v_1 \quad a_2 \to_a v_2}{s \vdash a_1 < a_2 \to_b \mathit{ff}} \quad \text{if } v_1 \not< v_2$$

$$\text{NOT-1}\ \frac{s \vdash b \to_b \mathit{tt}}{s \vdash \neg b \to_b \mathit{ff}} \qquad \text{NOT-2}\ \frac{s \vdash b \to_b \mathit{ff}}{s \vdash \neg b \to_b \mathit{tt}}$$

$$\text{PAREN-B}\ \frac{s \vdash b_1 \to_b v_1}{s \vdash \texttt{(}b_1\texttt{)} \to_b v_1}$$

$$\text{AND-1}\ \frac{s \vdash b_1 \to_b \mathit{tt} \quad s \vdash b_2 \to_b \mathit{tt}}{s \vdash b_1 \wedge b_2 \to_b \mathit{tt}} \qquad \text{AND-2}\ \frac{s \vdash b_i \to_b \mathit{ff}}{s \vdash b_1 \wedge b_2 \to_b \mathit{ff}} \quad i \in \{1, 2\}$$

$$\text{ASN}\quad \langle x := a, s\rangle \to s[x \mapsto v] \quad \text{where } s \vdash a \to_a v \qquad \text{SKIP}\quad \langle \texttt{skip}, s\rangle \to s$$

$$\text{COMP}\ \frac{\langle S_1, s\rangle \to s'' \quad \langle S_2, s''\rangle \to s'}{\langle S_1; S_2, s\rangle \to s'}$$

$$\text{IF-TRUE}\ \frac{\langle S_1, s\rangle \to s'}{\langle \texttt{if (}b\texttt{) then \{ }S_1\texttt{ \} else \{ }S_2\texttt{ \}}, s\rangle \to s'} \quad \text{if } s \vdash b \to_b \mathit{tt}$$

$$\text{IF-FALSE}\ \frac{\langle S_2, s\rangle \to s'}{\langle \texttt{if (}b\texttt{) then \{ }S_1\texttt{ \} else \{ }S_2\texttt{ \}}, s\rangle \to s'} \quad \text{if } s \vdash b \to_b \mathit{ff}$$

$$\text{WHILE-TRUE}\ \frac{\langle S, s\rangle \to s'' \quad \langle \texttt{while (}b\texttt{) do \{ }S\texttt{ \}}, s''\rangle \to s'}{\langle \texttt{while (}b\texttt{) do \{ }S\texttt{ \}}, s\rangle \to s'} \quad \text{if } s \vdash b \to_b \mathit{tt}$$

$$\text{WHILE-FALSE}\quad \langle \texttt{while (}b\texttt{) do \{ }S\texttt{ \}}, s\rangle \to s \quad \text{if } s \vdash b \to_b \mathit{ff}$$

図 4　**Bims** の意味規則

4.2　実装

- 字句解析器と構文解析器

  `Bims.cf` を作成して BNFC で生成すると良い．BNFC の Tutorial の Complete grammar for `C--`なども参考にしなさい．Windows 環境では，エディタで開かれているファイルをコマンドラインで実行されたコマンドによって上書きできないことに注意しなさい．言い換えると，BNFC の実行前に，生成して上書きしたいファイルは閉じておく必要がある．
- 解釈器

  BNF で生成されたテンプレートを元にして，Calc の `Interpreter` を参考に，作成すると良い．

Bexp 中で数学記号で表されたものはテキストに置き換える．$\wedge$ は `&&`，$\neg$ は `not` にする．
Bims の構文を LBNF で記述したものを以下に示す．

Bims.cf

```
-----------------------------------------
-- プログラミング言語 Bims の LBNF Grammar
-----------------------------------------

-- プログラム
PDefs.      Program ::= [Stm] ;

-- 文 (statement)
SAsn.       Stm ::= Ident ":=" AExp ;
SSkip.      Stm ::= "skip" ;
SIf.        Stm ::= "if" "(" BExp ")" "then" "{" [Stm] "}" "else" "{" [Stm] "}";
SWhile.     Stm ::= "while" "(" BExp ")" "do" "{" [Stm] "}";
SPrint.     Stm ::= "print" Ident ;

separator Stm ";" ;

-- 論理式 (Boolean expression)
BEq.        BExp  ::= AExp "=" AExp ;
BGt.        BExp  ::= AExp "<" AExp ;
BNeg.       BExp2 ::= "not" BExp3 ;
BAnd.       BExp2 ::= BExp2 "&&" BExp3 ;

coercions BExp 3 ;

-- 算術式 (Arithmetic expression)
APlus.      AExp1 ::= AExp1 "+" AExp2 ;
AMinus.     AExp1 ::= AExp1 "-" AExp2 ;
ATimes.     AExp2 ::= AExp2 "*" AExp3 ;
AVar.       AExp3 ::= Ident ;
AInt.       AExp3 ::= Integer ;

coercions AExp 3 ;

comment "//" ;                  -- 1 行コメント
comment "/*" "*/" ;             -- 複数行コメント
```

4.2.1 Java の場合

状態には，例えば，連想配列（HashMap）を使うと良い．実装例を以下に示す．

State.java

```
package Bims;
import java.util.HashMap;

public class State {
    public static HashMap<String,Integer> state ;

    State() {
        state = new HashMap<String,Integer>();
    }

    public static int lookupVar(String id) {
        return state.get(id);
    }
    public static void updateVar(String id, Integer v) {
        state.put(id,v);
    }
}
```

解釈器は，標準入力から文を入力として取り，標準出力へ状態を出力するものを考える．

Java では HashMap 型の値は `System.out.println` によって整形されて表示される．例えば，以下の `test01.bims` を実行して得られた状態は

```
{i=0, x=12}
```

と表示される．

test01.bims

```
x := 0;
i := 6;
while (not(i = 0)) do {x := x + i; i := i - 2}
```

単項演算子の `not` は二項演算子より優先順位が高いとせよ．したがって，`not(2=3) && (2=3)` は，`(not(2=3)) && (2=3)` であり，`not((2=3) && (2=3))` ではない．Bims 解釈器上では

```
if (not(2=3) && (2=3)) then { x := 1 } else { x := 0 }
```

の実行結果が `{x=0}` となることを確認すると良い．

test09p.bims

```
a := 2;
skip;
if (not (a = 2)) then { b := 99; print b } else { a := a * 6 - 7; print a };
while (not (a < 1)) do { a := a - 1; b := a; print b }
```

から生成されたコードの出力が

```
5
4
3
2
```

```
1
0
```

となることを確認せよ．

以下は Java 言語で実装で行うときのためのヒントである：

- 環境も連想配列を用いて実装すると良い．
- 2 つの Integer 型のオブジェクト a1 と a2 が格納する整数値が一致するかは
  ```
  a1.compareTo(a2) == 0
  ```
  と記述し，a1==a2 と記述するのではないことに注意せよ．a1==a2 はオブジェクト a1 がオブジェクト a2 と一致するときに真となる．
- C 言語などとは異なり，Java には boolean 型がある．論理値を表すには，boolean 型のラッパクラス Boolean を用いると良い．
- コレクションクラスのオブジェクトをひとつずつ取り出して処理を行う場合，拡張 for 文を用いると良い．例えば，p.liststm_の Stm 型の要素をひとつづつ取り出して処理を行う場合は，
  ```
  for (Stm s : p.liststm_) {
      ...
  }
  ```
  となる．

4.2.2 Haskell の場合

状態を扱う抽象データ型を自分で初めから作りたい場合は，[6, 第 2 章] を参考にすると良い．

状態の抽象データ型に対する演算

```
empty :: State
insert :: Ident -> Integer -> State -> State
mylookup :: Ident -> State -> Maybe Integer
```

を考える．empty は空の状態を表す．insert x v s は，状態 s において変数 x が値 v に更新された状態を表す．mylookup x s は，状態 s において変数 x の値が関連づけられているときはその値 v を Just v として，そうでない場合は，Nothing を返す．これらの演算の間の関係を定めることで仕様を定める．すべての x に対して，

```
mylookup x empty = Nothing
```

が成り立つ．また，すべての x と x とは異なるすべての x' に対して，

```
mylookup x (insert x v state) = Just v
mylookup x' (insert x v state) = mylookup x' state
```

が成り立つ．

ここでは，既にあるモジュール Data.Map を用いる方法を推奨したい．状態を扱う抽象データ型の実装例を以下に示す．

StateBims.hs

```
module StateBims(State,empty,insert,mylookup) where

import AbsBims
import qualified Data.Map as Map
type State = Map.Map Ident Integer

empty = Map.empty

insert :: Ident -> Integer -> State -> State
insert = Map.insert

mylookup :: Ident -> State -> Maybe Integer
mylookup = Map.lookup
```

式 21 を実現するには，変数を評価するために状態 State を用いる必要がある．

Interpreter.hs の一部

```
interpretAExp :: AExp -> State -> Integer
interpretAExp x store = case x of
  APlus  exp0 exp -> interpretAExp exp0 store + interpretAExp exp store
  AMinus exp0 exp -> interpretAExp exp0 store - interpretAExp exp store
  ATimes exp0 exp -> interpretAExp exp0 store * interpretAExp exp store
  AVar ident -> (case mylookup ident store of Just x -> x)
  AInt n ->n
```

式 20, 式 19 も同様に以下の型の関数で実現する：

```
interpretBExp :: BExp -> State -> Bool
interpretStm :: Stm -> State -> State
```

さらに，以下の補助関数を用いると良い：

```
interpretStms :: [Stm] -> State -> State
interpretProgram :: Program -> State -> State
```

5 Java 仮想機械

Jasmin の Web ページに命令（Instruction）についての説明がある（http://jasmin.sourceforge.net/instructions.html）．表 2 に主なバイトコードを示す．

記憶域 $M(\in \mathbf{Addr} \to \mathbb{Z})$, スタック S, プログラムカウンタ $P(\in \mathbf{Addr})$, ラベルからアドレスへの関数 $\tau \in \mathbf{Label} \to \mathbf{Addr}$ の 4 つ組みからなる環境を考える．記憶域 M は，与えられたアドレスにある命令やデータを返す．記憶域 M のアドレス i の値を v に書き換えることを $M[i \mapsto v]$ と表すことにする．意味規

表 2　Java バイトコード

命令	説明
`bipush` n	byte 値 n をプッシュする
`iadd`	2 つの値をポップして，合計をプッシュする
`isub`	2 つの値をポップして，差をプッシュする
`imul`	2 つの値をポップして，積をプッシュする
`iload` i	アドレス i に格納されている値をプッシュする
`istore` i	値をポップし，その値をアドレス i に格納する
`goto` L	ラベル L の位置に制御を移す
`ifeq` L	値をポップし，その値が 0 であるならばラベル L の位置に制御を移す
`if_icmpne` L	2 つの値をポップし，それらの値が等しくないならばラベル L の位置に制御を移す
`if_icmpge` L	2 つの値をポップし，2 番目の値がトップの値より
	大きいか等しいならばラベル L の位置に制御を移す

則を

$$
\begin{array}{lll}
\langle M, S, P, \tau\rangle & \to \langle M, S.v, P+1, \tau\rangle & \text{if } M(P) = \texttt{bipush}\ v \\
\langle M, S.v.w, P, \tau\rangle & \to \langle M, S.(v+w), P+1, \tau\rangle & \text{if } M(P) = \texttt{iadd} \\
\langle M, S.v.w, P, \tau\rangle & \to \langle M, S.(v\times w), P+1, \tau\rangle & \text{if } M(P) = \texttt{imul} \\
\langle M, S, P, \tau\rangle & \to \langle M, S.M(i), P+1, \tau\rangle & \text{if } M(P) = \texttt{iload}\ i \\
\langle M, S.v, P, \tau\rangle & \to \langle M[i \mapsto v], S, P+1, \tau\rangle & \text{if } M(P) = \texttt{istore}\ i \\
\langle M, S, P, \tau\rangle & \to \langle M, S, i, \tau\rangle & \text{if } M(P) = \texttt{goto}\ L \wedge \tau(L) = i \\
\langle M, S.0, P, \tau\rangle & \to \langle M, S, i, \tau\rangle & \text{if } M(P) = \texttt{ifeq}\ L \wedge \tau(L) = i \\
\langle M, S.v, P, \tau\rangle & \to \langle M, S, P+1, \tau\rangle & \text{if } M(P) = \texttt{ifeq}\ L \wedge v \neq 0
\end{array}
$$

で与える．

5.1　字句解析器と構文解析器の実現

BNFC の入力ファイルは，次の `JavaVM.cf` のようなものを用意すればよい．

JavaVM.cf

```
PDefs.     Program ::= [Inst] ;

ILbl.      Inst ::= Ident ":";  -- ラベル
IBipush.   Inst ::= "bipush" Integer ;
/* ... 以下続く ... */

separator Inst "" ;

comment ";" ;                   -- 1 行コメント
```

5.2　Bims から Java バイトコードの生成

本節では，Bims からの Java バイトコードの生成を行う．BNF で生成されたテンプレートを元にして，Calc の `CodeGenarate` を参考に，作成すると良い．

コード生成には，環境 **Env** と記憶場所 **Loc**

$$\begin{aligned}\mathbf{Env} &= \mathbf{Var} \rightharpoonup \mathbf{Loc}\\ \mathbf{Loc} &= \mathbb{N}\end{aligned}$$

が必要である．環境は，変数の記憶場所を表すのに用いる．

5.2.1 Java の場合

CodeGenerator.java の一部

```
public State visit(Bims.Absyn.SIf s, State env) {
    int label = genlab();
    compile(s.bexp_, env);
    System.out.println("ifeq false" + label);
    env = compile(s.liststm_1, env);
    System.out.println("goto true" + label);
    System.out.println("false" + label + ":");
    env = compile(s.liststm_2, env);
    System.out.println("true" + label + ":");
    return env;
}
```

コード生成では，異なるラベルには異なる識別子を用いることに注意せよ．このことを，上のコード辺では `genlab()` というメソッドを用いて実現している．`genlab()` は初めて呼ばれたとき 0 を返し，その後は呼ばれるたびに 1 ずつ大きい整数を返している．例えば，生成された整数 3 が `label` に格納された場合は，ラベルは `false3` および `true3` になる．

6 言語の拡張 — BimsT

本章では，型検査という意味解析を学ぶ．

Bims を拡張する．文に宣言文を，算術式に浮動小数点数をそれぞれ追加する．

$$\begin{array}{lcl} S & ::= & x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if (}b\texttt{) then \{ }S_1\texttt{ \} else \{ }S_2\texttt{ \}} \mid \texttt{while (}b\texttt{) do \{ }S\texttt{ \}} \mid \texttt{print }x \mid \\ & & \texttt{int }x \mid \texttt{double }x \\ b & ::= & a_1 = a_2 \mid a_1 < a_2 \mid \neg b_1 \mid b_1 \wedge b_2 \mid \texttt{(}b_1\texttt{)} \\ a & ::= & n \mid d \mid a_1\texttt{+}a_2 \mid a_1\texttt{*}a_2 \mid a_1\texttt{-}a_2 \mid \texttt{(}a_1\texttt{)} \mid x \end{array}$$

型の文脈は変数と型の組 $x : T$ の列である．

型の文脈 Γ，構文要素 e (e は b もしくは a)，型 t が与えられたときの型の判定は

$$\Gamma \vdash e : t$$

と表される．文脈 Γ の下で，構文要素 e が型 t をもつことを表す．論理式と算術式の型の推論規則は以下の通りである：

$$\frac{\Gamma \vdash a_1 : t \quad \Gamma \vdash a_2 : t}{\Gamma \vdash a_1 = a_2 : \mathit{bool}} \text{ if } t \text{ is } \mathit{int} \text{ or } \mathit{double} \qquad \frac{\Gamma \vdash a_1 : t \quad \Gamma \vdash a_2 : t}{\Gamma \vdash a_1 < a_2 : \mathit{bool}} \text{ if } t \text{ is } \mathit{int} \text{ or } \mathit{double}$$

$$\frac{\Gamma \vdash b : \mathit{bool}}{\Gamma \vdash \neg b : \mathit{bool}} \qquad \frac{\Gamma \vdash b_1 : \mathit{bool} \quad \Gamma \vdash b_2 : \mathit{bool}}{\Gamma \vdash b_1 \wedge b_2 : \mathit{bool}} \qquad \frac{\Gamma \vdash b : \mathit{bool}}{\Gamma \vdash (b) : \mathit{bool}}$$

$$\Gamma \vdash n : \mathit{int} \qquad \Gamma \vdash d : \mathit{double}$$

$$\frac{\Gamma \vdash a_1 : t \quad \Gamma \vdash a_2 : t}{\Gamma \vdash a_1 \texttt{+} a_2 : t} \text{ if } t \text{ is } \mathit{int} \text{ or } \mathit{double} \qquad \frac{\Gamma \vdash a_1 : t \quad \Gamma \vdash a_2 : t}{\Gamma \vdash a_1 \texttt{*} a_2 : t} \text{ if } t \text{ is } \mathit{int} \text{ or } \mathit{double}$$

$$\frac{\Gamma \vdash a_1 : t \quad \Gamma \vdash a_2 : t}{\Gamma \vdash a_1 \texttt{-} a_2 : t} \text{ if } t \text{ is } \mathit{int} \text{ or } \mathit{double} \qquad \frac{\Gamma \vdash a : t}{\Gamma \vdash \texttt{(}a\texttt{)} : t} \text{ if } t \text{ is } \mathit{int} \text{ or } \mathit{double}$$

$$\Gamma \vdash x : t \quad \text{if } x : t \text{ is in } \Gamma$$

文の型の判定は

$$\Gamma \vdash S \Rightarrow \Gamma'$$

であり，推論規則は以下の通りである：

$$\frac{\Gamma \vdash x : t \quad \Gamma \vdash a : t}{\Gamma \vdash x := a \Rightarrow \Gamma} \text{ if } t \text{ is } \mathit{int} \text{ or } \mathit{double} \qquad \Gamma \vdash \texttt{skip} \Rightarrow \Gamma$$

$$\frac{\Gamma \vdash S_1 \Rightarrow \Gamma' \quad \Gamma' \vdash S_2 \Rightarrow \Gamma''}{\Gamma \vdash S_1 ; S_2 \Rightarrow \Gamma''} \qquad \frac{\Gamma \vdash b : \mathit{bool} \quad \Gamma \vdash S_1 \Rightarrow \Gamma' \quad \Gamma \vdash S_2 \Rightarrow \Gamma''}{\Gamma \vdash \texttt{if (}b\texttt{) then \{ }S_1\texttt{ \} else \{ }S_2\texttt{ \}} \Rightarrow \Gamma}$$

$$\frac{\Gamma \vdash b : \mathit{bool} \quad \Gamma \vdash S \Rightarrow \Gamma'}{\Gamma \vdash \texttt{while (}b\texttt{) do \{ }S\texttt{ \}} \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash x : t}{\Gamma \vdash \texttt{print } x \Rightarrow \Gamma} \text{ if } t \text{ is } \mathit{int} \text{ or } \mathit{double}$$

$$\Gamma \vdash \texttt{int } x \Rightarrow \Gamma, x : \mathit{int} \qquad \Gamma \vdash \texttt{double } x \Rightarrow \Gamma, x : \mathit{double}$$

例えば，型検査では以下のように不正なプログラム片を実行前に検知することができる．

- 宣言をしていない変数を使用している場合，任意の文脈の下で判定が成立しない．
- `1 + 3.14` は，整数と浮動小数点数を足し合わせる式であり，任意の文脈の下で判定が成立しない．

6.1 型の表現

Java においては，型の表現には列挙型[*9]を用いると良い：

```
enum Type {
    BOOL,
    INT,
    DOUBLE;
}
```

7 Web ページの作成

プログラミング言語の研究者は，新しいアイデアを試すために新しいプログラミング言語を提案し続けている．そうした新しい言語の実装をユーザが試すときに，ソースコードをダウンロードしてコンパイルしてインストールするのは簡易ではない．オンラインインタープリタを試すにはこうしたプロセスは不要である．そこで，研究者はオンラインインタープリタを公開することがある．

開発したプログラミング言語を実行することができる Web ページを作成する．

[*9] http://docs.oracle.com/javase/jp/6/api/java/lang/Enum.html

7.1 参考ページ

- プログラミング言語 Janus の解釈系の Web ページ（http://tetsuo.jp/janus-playground/）が公開されている．Web ページのソースコードは https://github.com/mbudde/jana/tree/master/web に公開されている．
- プログラミング言語 R-WHILE の解釈系の Web ページ（http://tetsuo.jp/rwhile-playground/）が公開されている．Web ページのソースコードは https://github.com/tyoko-dev/rwhile-C-ocaml/tree/master/web に公開されている．

参考文献

[1] 結城　浩：増補改訂版 Java 言語で学ぶデザインパターン入門，SB クリエイティブ (2004).

[2] Hüttel, H.: *Transitions and Trees: An Introduction to Structural Operational Semantics*, Cambridge University Press (2010).

[3] Ranta, A.: *Implementing Programming Languages*, College Publications (2012).

[4] Ranta, A.: Implementing Programming Languages. Available from http://www.cse.chalmers.se/edu/year/2012/course/DAT150/lectures/plt-book.pdf.

[5] Peyton Jones, S. and Lester, D.: *Implementing functional languages: A tutorial*, Prentice Hall (1992). Available from http://research.microsoft.com/en-us/um/people/simonpj/Papers/pj-lester-book/.

[6] 武市正人：プログラミング言語，岩波書店 (1994).

[7] 池田竜平，三輪峻大，須藤拓也：単純な算術式言語の処理系の試作 (2019). 南山大学 2018 年度卒業論文要旨, http://www.st.nanzan-u.ac.jp/info/gr-thesis/2018/yokoyama/pdf/15se019.pdf.

[8] 五十嵐淳：プログラミング言語の基礎概念，サイエンス社 (2011).

[9] Pierce, B. C.: 型システム入門：プログラミング言語と型の理論，オーム社 (2013). 住井 英二郎（監訳)), 遠藤 侑介，酒井 政裕，今井 敬吾，黒木裕介，今井 宣洋，才川 隆文，今井 健男 (翻訳).