第3章　MLインタプリタの設計と実装

そこでみなさん，プログラムを何か一つ書いてください．私がそれを構文木にしますから，そこでなにか一言．はい楽さん早かった.[a]

[a]「「楽さん」は古すぎる．せめて「圓楽さん」では」との五十嵐さんからのツッコミがあったが，気の利いたエピグラフを思いつかないのでそのままにしておく.「そもそも，これは構文解析の章に置くべきでは」とのツッコミもあったが，これもそのままにしておく．

この章では，言語を定義し，そのインタプリタを実装する方法を解説する．??章で述べたように，インタプリタは，文字列を受け取って，それを特定のプログラミング言語のプログラムとして解釈して，そのセマンティクスに従って実行結果を計算するソフトウェアである．

セマンティクスの定義（以下の説明は分からなければ飛ばして良い.）??章では言語にはシンタックスとセマンティクスが定められており，インタプリタはシンタックスに従って字句解析と構文解析を行い，セマンティクスに従って計算を行うと述べた．この関係を逆に見て，インタプリタがプログラミング言語のシンタックスとセマンティクスを定義していると見ることがある.[1]すなわち，インタプリタの構文解析器の挙動をもって言語のシンタックスとし，インタプリタの計算の過程をもって言語のセマンティクスとすることもできる．この視点の下では,「言語のシンタックスとセマンティクスを与える」とは，すなわちその言語のインタプリタを一つ与えることである．実際に，数学的に厳密な形でセマンティクスを与える方法の一つである**操作的意味論** (*operational semantics*) は，インタプリタの挙動を数学の言葉（写像や関係など）で表現する．これに対置されるのが，インタプリタの挙動を経由せずに，構文木からその木が表現する計算への写像を与えることでセマンティクスを与える手法である．これを**表示的意味論** (*denotational semantics*) と呼ぶ.

コンパイラもまたこの視点から見ることができる．すなわち，コンパイラは，入力プログラムを受け取ると，そのプログラムが表す計算を実行するターゲット言語のプログラムを出力するのであるが，出力されたプログラムの挙動を通じて，ソース言語のセマンティクスが定まると見るのである.

なにを訳の分からないことをフニャフニャと言っておるのかおまえはフニャコフニャ夫かと言いたくなるかもしれないが,「プログラミング言語のセマンティクスをどのように定義するか」は**プログラミング言語理論** (*programming language theory*) と呼ばれる分野の大きなトピックの一つになっている．実行が止まらないかもしれない言語，非決

[1]いわゆる「その挙動は仕様です」である.

定的実行を含む言語，確率的挙動を含む言語，並行性を含む言語，連続的挙動を含む言語，量子プログラミング言語など，さまざまな言語に対して数学的に厳密な形でセマンティクスを与えることは,「計算」の本質という理論的興味からも，プログラミング言語を明確かつ簡潔に定義してプログラマが安心してプログラムを作れるようにするという意味でも重要である.[2]興味があれば，五十嵐 []，Winskel [] を読むと良い.

さて，インタプリタ自体もプログラムであるから，なんらかのプログラミング言語で書かれている．このとき,「インタプリタ自体が書かれているプログラミング言語」を**定義する言語** (*defining language*) といい,「インタプリタが入力として受け取るプログラミング言語」を**定義される言語** (*defined language*) という． 本実験では,

$$\begin{aligned} \text{定義する言語} &= \text{OCaml} \\ \text{定義される言語} &= \mathsf{ML}\ (\text{OCaml のサブセット}) \end{aligned}$$

である．一般には，定義する言語と定義される言語は異なるが，今回のように両者が一致する場合，そのインタプリタを特に，**メタ・サーキュラ・インタプリタ** (*meta circular interpreter*) という．以下では，定義する言語でのプログラムの記述にはタイプライタ体 (abcde) を，定義される言語のプログラムにはサン・セリフ体 (abcde) を用いて，両者を区別する．

←— [具象構文と抽象構文について．]

典型的なインタプリタは，字句解析・構文解析・解釈部から構成される．字句解析・構文解析はコンパイラと同様に，文字列からプログラムの抽象構文木を生成する過程で，定義される言語のシンタックスを規定している．解釈部分は，セマンティクスを定義していて，抽象構文木を入力としてプログラムの実行結果を計算する部分で，インタプリタの核となる．

←— [流れ図？]

本書では，とてもシンプルな OCaml のサブセット ML^1 のインタプリタを実装し，これに以下のように徐々に言語機能を拡充していく．

ML^1 整数，真偽値，条件分岐，加算乗算とあらかじめ定義されている変数の参照のみが可能な OCaml のサブセット.

ML^2 ML^1 を変数定義機能で拡張した言語.

ML^3 ML^2 を（高階）関数で拡張した言語.

ML^4 ML^3 を再帰関数で拡張した言語.

ML^5 ML^4 をリストで拡張した言語.

3.1 プログラムファイルの構成・コンパイル方法

これから作成するインタプリタのソースコードは以下のファイルから構成される.[3]

[2]ただし，末永は主に言語に対する個人的なフェティシズムを満足させるためにこういう研究を行っているフシがある．プログラムによって面白いソフトウェアを作ることよりも，プログラムを記述するための言語自体に興味を持つという性格は，ある種の偏愛性を帯びているものかもしれない.

[3]プログラムはこのリポジトリの `src` ディレクトリに格納されている.

`syntax.ml` 抽象構文木のデータ構造を定義している．抽象構文木は構文解析の出力であり，解釈部の入力なので，インタプリタの全ての部分が，この定義に (直接/間接的に) 依存する．

`parser.mly` OCaml のパーザジェネレータ Menhir に入力する文法定義である．Menhir は `.mly` という拡張子のファイルに記述された BNF 風の文法定義から，構文解析プログラムを生成する．定義の書き方は**??**章で説明する．

`lexer.mll` OCaml の字句解析器生成ツールである ocamllex の定義ファイルである．ocamllex は `.mll` という拡張子のファイルに定義されたパターン定義から，字句解析プログラムを生成する．定義の書き方は**??**章で説明する．

`environment.mli`, `environment.ml` インタプリタ・型推論で用いる，**環境** (*environment*) と呼ばれるデータ構造を定義する．

`eval.ml` 解釈部プログラムである．構文解析部が生成した構文木から計算を行なう．

`main.ml` 字句解析・構文解析・解釈部を組み合わせて，インタプリタ全体を機能させる．プログラム全体の開始部分でもある．

`Makefile` インタプリタをビルドするために使う Makefile である．

また，以下のファイルは**??**章において，**型推論** (*type inference*) アルゴリズムを実装するために用いる．

`mySet.ml`, `mySet.mli` 集合の抽象データ型の定義とインターフェースである．型推論の実装で用いる．

`typing.ml` 型推論アルゴリズムの実装である．最初は何も書いていない．

インタプリタのビルド方法はインタプリタのソースコードが格納されているディレクトリの README.md ファイルに書いてある．このファイルはテキストファイルなので，テキストエディタで開いて読んでほしい．シェルを立ち上げて，ソースコードが格納されているディレクトリで make と入力すればビルドできるようになっている．ML というバイナリができるので，以下のように実行するとインタプリタを立ち上げることができる．

```
> ./miniml
# x;;
val - = 10
# x + 3;;
val - = 13
```

インタプリタの起動時にはデフォルトでいくつか変数が定義されている．起動時に大域変数 x の値は 10 になっている．ただし，デフォルトのインタプリタの機能は限られており，変数や（再帰）関数を定義することはできない．例えば，インタプリタに以下の変数定義式を入力してみよう．

```
# let a = 2 in a + 3;;
Fatal error: exception Parser.Basics.Error
```

インタプリタがエラーを吐いて落ちてしまった．出力されたメッセージから，インタプリタが `Parser.Basics.Error` という例外を投げて落ちたことが分かる．これは構文解析器の中で定義されている例外で，文法エラーが起こったために処理を続行できなかったことを表している．これから，様々な式を処理できるようにインタプリタを拡張する．

ope

3.2 ML^1 インタプリタ — プリミティブ演算，条件分岐と環境を使った変数参照

まず，非常に単純な言語として，整数，真偽値，条件分岐，加算乗算と変数の参照のみ（新しい変数の定義すらできない!）を持つ言語 ML^1 から始める．

3.2.1 ML^1 のシンタックス

ML^1 のシンタックスは以下の BNF で与えられる．[文脈自由文法の説明と *BNF* の読み方の説明．] ←

$$
\begin{aligned}
\langle \text{プログラム} \rangle &::= \langle \text{式} \rangle ;; \\
\langle \text{式} \rangle &::= \langle \text{識別子} \rangle \\
&\quad | \quad \langle \text{整数リテラル} \rangle \\
&\quad | \quad \langle \text{真偽値リテラル} \rangle \\
&\quad | \quad \langle \text{式}_1 \rangle \langle \text{二項演算子} \rangle \langle \text{式}_2 \rangle \\
&\quad | \quad \text{if} \langle \text{式}_1 \rangle \text{ then } \langle \text{式}_2 \rangle \text{ else } \langle \text{式}_3 \rangle \\
&\quad | \quad (\langle \text{式} \rangle) \\
\langle \text{二項演算子} \rangle &::= + \mid * \mid <
\end{aligned}
$$

ML^1 のプログラムは，;; で終わるひとつの式である．式は，識別子による変数参照，整数リテラル，真偽値リテラル (true と false)，条件分岐のための if 式，または二項演算子式，括弧で囲まれた式のいずれかである．識別子は，英小文字で始まり，数字・英小文字・「'(アポストロフィ)」を並べた，予約語ではない文字列である．この段階では予約語は if, then, else, true, false の 5 つである．例えば，以下の文字列はいずれも ML^1 プログラムである．

```
3;;
true;;
x;;
3 + x';;
(3 + x1) * false;;
```

```
(* ML interpreter / type reconstruction *)
type id = string

type binOp = Plus | Mult | Lt

type exp =
    Var of id
  | ILit of int
  | BLit of bool
  | BinOp of binOp * exp * exp
  | IfExp of exp * exp * exp

type program =
    Exp of exp
```

図 3.1: ML[1] インタプリタ: syntax.ml

また，+, * は左結合，結合の強さは，強い方から，*, +, <, if 式 とする．

3.2.2 各モジュールの機能

実装するインタプリタは 6 つのモジュールから構成される．[4]それぞれのモジュールについて簡単に説明する．

3.2.3 Syntax モジュール：抽象構文のためのデータ型の定義

Syntax モジュールはファイル syntax.ml に定義されており，抽象構文木を表すデータ型を定義している．具体的には，このモジュールでは上の BNF に対応する抽象構文木を表す以下の型定義が含まれている．id は変数の識別情報を示すための型で，ここでは変数の名前を表す文字列としている．(より現実的なインタプリタ・コンパイラでは，変数の型や変数が現れたファイル名と行数などの情報も加わることが多い．) binOp，exp，program 型に関しては上の文法構造を (括弧式を除いて) そのまま写した形の宣言になっていることがわかるだろう．例えば，3+x' は BinOP(Plus, ILit 3, Var "x'") で表現される．

[4]OCaml を含め多くのプログラミング言語には，**モジュールシステム** (*module system*) と呼ばれる，プログラムを部分的な機能（**モジュール** (*module*)）ごとに分割するための機構が備わっている．この機構は，プログラムが大規模化している現代的なプログラミングにおいて不可欠な機構であるが，その解説は本書の範囲を超える．「OCaml 入門」の該当する章を参照されたい．さしあたって理解しておくべきことは，(1) OCaml プログラムを幾つかのファイルに分割して開発すると，ファイル名に対応したモジュールが生成されること（例えば，foo.ml というファイルからは Foo というモジュールが生成される）(2) モジュール内で定義されている変数や関数をそのモジュールの外から参照するにはモジュール名を前に付けなければならないこと（例えばモジュール Foo の中で定義された x という変数を Foo 以外のモジュールから参照するには Foo.x と書く）の二点である．

3.2.4 Parser モジュール，Lexer モジュール：字句解析と構文解析

Parser と Lexer はそれぞれ構文解析と字句解析を行うモジュールである．Parser モジュールは Menhir というツールを用いて parser.mly というファイルから，Lexer モジュールは ocamllex というツールを用いて lexer.mll というファイルからそれぞれ自動生成される．

Menhir は **LR(1) 構文解析** (*LR(1) parsing*) という手法を用いて，BNFっぽく書かれた文法定義（ここでは parser.mly）から，構文解析を行う OCaml のプログラム（ここでは parser.ml と parser.mli）を自動生成する．また，ocamllex は**正則表現** (*regular expression*) を使って書かれたトークンの定義（ここでは lexer.mll）から，字句解析を行う OCaml のプログラム（ここでは lexer.ml）を自動生成する．生成されたプログラムがどのように字句解析や構文解析を行うかはこの講義の後半で触れる．そのような仕組みの部分を抜きにして，ここでは .mly ファイルや .mll ファイルの書き方を説明する．

文法定義ファイルの書き方

拡張子 .mly 文法定義ファイルは一般に，以下のように 4 つの部分から構成される．

```
%{
  ⟨ヘッダ⟩
%}
  ⟨宣言⟩
%%
  ⟨文法規則⟩
%%
  ⟨トレイラ⟩
```

⟨ヘッダ⟩, ⟨トレイラ⟩ は OCaml のプログラムを書く部分で，Menhir が生成する parser.ml の，それぞれ先頭・末尾にそのまま埋め込まれる．⟨宣言⟩ はトークン (終端記号) や，開始記号，優先度などの宣言を行う．parser.mly では演習を通して，開始記号とトークンの宣言のみを使用する．⟨文法規則⟩ には文法記述と還元時のアクションを記述する．コメントは OCaml と同様 (* ... *) である．[5]

それでは parser.mly を見てみよう (図 3.2)．[6] この文法定義ファイルではトレイラは空になっていて，その前の %% は省略されている．

- ヘッダにある open Syntax 宣言はモジュール Syntax 内で定義されているコンストラクタや型の名前を，Syntax. というプレフィクス無しで使うという OCaml の構文である．(これがないと，例えばコンストラクタ Var を参照するときに Syntax.Var と書かなくてはならない．[7])

[5] ヘッダ部分とトレイラ部分以外では /* ... */ と //... が使えるらしい．

[6] 以降の話は結構ややこしいかもしれないので，全部理解しようとせずに，parser.mly と lexer.mll を適当にいじって遊ぶ，くらいの気楽なスタンスのほうがよいかもしれない．

[7] OCaml 以外にもこの手の機構が用意されていることが多い．例えば Java ではパッケージの import，Python では import 文がこれに相当する．なお，open はモジュール内の名前に容易にアクセスすることを可能にするが，外のモジュールで定義されている名前との衝突も起きやすくするという諸刃の剣である．この辺の話は時間があれば講義で少し触れる．

```
%
open Syntax
%

%token LPAREN RPAREN SEMISEMI
%token PLUS MULT LT
%token IF THEN ELSE TRUE FALSE

%token <int> INTV
%token <Syntax.id> ID

%start toplevel
%type <Syntax.program> toplevel
%%

toplevel :
    e=Expr SEMISEMI { Exp e }

Expr :
    e=IfExpr { e }
  | e=LTExpr { e }

LTExpr :
    l=PExpr LT r=PExpr { BinOp (Lt, l, r) }
  | e=PExpr { e }

PExpr :
    l=PExpr PLUS r=MExpr { BinOp (Plus, l, r) }
  | e=MExpr { e }

MExpr :
    l=MExpr MULT r=AExpr { BinOp (Mult, l, r) }
  | e=AExpr { e }

AExpr :
    i=INTV { ILit i }
  | TRUE   { BLit true }
  | FALSE  { BLit false }
  | i=ID   { Var i }
  | LPAREN e=Expr RPAREN { e }

IfExpr :
    IF c=Expr THEN t=Expr ELSE e=Expr { IfExp (c, t, e) }
```

図 3.2: ML[1] インタプリタ: parser.mly

- %token ⟨トークン名⟩ ... は，**属性** (*attribute*) を持たないトークンの宣言である．属性とは，トークンに関連付けられた（以下で説明する）還元時アクションの中で参照することができる値のことである．属性を持つトークンを見ればなるほどと納得が行くかもしれない．parser.mly 中では括弧 “(”，“)” と，入力の終了を示す “;;” に対応するトークン LPAREN, RPAREN，SEMISEMI と，プリミティブ (+, ∗, <) に対応するトークン PLUS, MULT, LT, 予約語 if, then, else, true, false に対応するトークンが宣言されている．(図 3.1 に現れる構文木のコンストラクタ Plus などとの区別に注意すること．トークン名は全て英大文字としている．) この宣言で宣言されたトークン名は Menhir の出力する parser.ml 中で，token 型の (引数なし) コンストラクタになる．字句解析プログラムは文字列を読み込んで，この型の値 (の列) を出力することになる．

- %token <⟨型⟩> ⟨トークン名⟩ ... は，属性つきのトークン宣言である．数値のためのトークン INTV（属性はその数値情報なので int 型）と変数のための ID（属性は変数名を表す Syntax.id 型[8]）を宣言している．[*Menhir* でも Syntax. は必要？] この宣言で宣言されたトークン名は parser.ml 中で，⟨型⟩ を引数とする token 型のコンストラクタになる．

- %start ⟨開始記号名⟩ ... で (一つ以上の) 開始記号の名前を指定する．Menhir が生成する parser.ml ファイルでは，同名の関数が構文解析関数として宣言される．ここでは toplevel という名前を宣言しているので，後述する main.ml では Parser.toplevel という関数を使用して構文解析をしている．開始記号の名前は，次の %type 宣言でも宣言されていなくてはならない．

- %type <⟨型⟩> ⟨名前⟩ ... 名前の属性を指定する宣言である，toplevel はひとつのプログラムの抽象構文木を表すので属性は Syntax.program 型となっている．

- 文法規則は，

  ⟨非終端記号名⟩ :
  　(⟨変数名 $_{11}$⟩=)⟨記号名 $_{11}$⟩ ... (⟨変数名 $_{1n_1}$⟩=)⟨記号名 $_{1n_1}$⟩
  　{ ⟨還元時アクション $_1$⟩ }
  | (⟨変数名 $_{21}$⟩=)⟨記号名 $_{21}$⟩ ... (⟨変数名 $_{2n_2}$⟩=)⟨記号名 $_{2n_2}$⟩
  　{ ⟨還元時アクション $_1$⟩ }
  ...

  のように記述する．⟨記号名⟩ の場所にはそれぞれ非終端記号か終端記号を書くことができる．「⟨変数名⟩=」の部分は省略してもよい．⟨還元時アクション⟩ の場所には OCaml の式を記述する．

  構文解析器は，開始記号から始めて，与えられたトークン列を生成するために適用すべき規則を適切に発見し，それぞれの規則の還元時アクションを評価して，評価結果を規則の左辺の非終端記号の属性とすることで，開始記号の属性を計算する．と言わ

[8] ヘッダ部の open 宣言はトークン宣言部分では有効ではないので，Syntax. をつけることが必要である．

れてもよく分からないと思うので，図 3.2 の文法定義を例にとって説明する．この文法定義から生成される構文解析器に TRUE SEMISEMI というトークン列が与えられたとしよう.[9]このトークン列は開始記号 toplevel から始めて以下のように規則を適用すると得られることが分かる.[10]

```
toplevel
--（規則 toplevel: Expr SEMISEMI を用いて）-->
Expr SEMISEMI
--（規則 Expr: LTExpr を用いて）-->
LTExpr SEMISEMI
--（規則 LTExpr: PExpr を用いて）-->
PExpr SEMISEMI
--（規則 PExpr: MExpr を用いて）-->
MExpr SEMISEMI
--（規則 MExpr: AExpr を用いて）-->
AExpr SEMISEMI
--（規則 AExpr: TRUE を用いて）-->
TRUE SEMISEMI
```

各ステップで規則が適用された非終端記号に下線を付した．各ステップで用いられた規則を確認してほしい．

構文解析器は，この導出列を遡りながら，還元時アクションを評価し，各規則の左辺にある非終端記号の属性を計算する．例えば，

```
AExpr SEMISEMI
--（規則 AExpr: TRUE を用いて）-->
TRUE SEMISEMI
```

の規則が適用されている場所では，左辺の非終端記号 AExpr の属性が還元時アクション BLit true の評価結果（すなわち，BLit true という値）となる．ここで計算された属性は，その一つ手前の導出

```
MExpr SEMISEMI
--（規則 MExpr: AExpr を用いて）-->
AExpr SEMISEMI
```

で MExpr の属性を計算するのに使われる．ここで図 3.2 の対応する規則の右辺は e=AExpr となっているが，これは先程計算した AExpr の属性を e という名前で還元時アクションの中で参照できることを表している．ここでは還元時アクションは e なので，MExpr の属

[9]このトークン列は true;; という文字列を lexer.mll から生成される字句解析器に与えることで生成される．

[10]ちなみに，なぜこれが「分かる」のかが構文解析アルゴリズムの大きなテーマである．構文解析アルゴリズムについては講義中に扱うので，それまでは何らかの方法でこれが分かるのだと流してほしい．

性は e，すなわち AExpr の属性である BLit true となる．これを繰り返すと，開始記号 toplevel の属性が Exp (BLit true) と計算され，これがトークン列 TRUE SEMISEMI に対する構文解析器の出力となる．

図 3.2 の文法規則が，3.2.1 節で述べた結合の強さ，左結合などを実現していることを確かめてもらいたい．

トークン定義ファイルの書き方

さて，この構文解析器への入力となるトークン列を生成するのが字句解析器である．より正確には，字句解析器は文字の列を受け取って，その文字列をトークン列に変換する関数である．この関数をアルゴリズムの実装には，文字をアクションとする有限状態オートマトンを用いることが多い.[11]ただし，必要な有限状態オートマトンとその実行を一から実装するのは大変なので，どの文字列をどのトークンに対応付けるべきかを記述したファイルから，有限状態オートマトンを用いて字句解析を行うプログラムを自動生成する lex や flex と呼ばれるツールを使うことが多い．本講義では実装言語として OCaml を用いる関係上，OCaml から使うのに便利な ocamllex と呼ばれるツールを用いることにする．

ocamllex は正則表現を使ってどのような文字列からどのようなトークンを生成すべきかを指定する．(正則表現は lex や flex においても同様に用いられる.) この指定は拡張子 .mll を持つファイルに以下のように記述する．

```
{ 〈ヘッダ〉 }

let 〈名前〉 = 〈正則表現〉
...

rule 〈エントリポイント名〉 =
  parse 〈正則表現〉 { 〈アクション〉 }
    |   〈正則表現〉 { 〈アクション〉 }
    |   ...
and 〈エントリポイント名〉 =
  parse ...
and ...
{ 〈トレイラ〉 }
```

ヘッダ・トレイラ部には，OCaml のプログラムを書くことができ，ocamllex が生成する lexer.ml ファイルの先頭・末尾に埋め 込まれる．次の let を使った定義部は，よく使う正則表現に名前をつけるための部分で，lexer.mll では何も定義されていない．続く部分がエントリポイント，つまり字句解析の規則の定義で，同名の関数が ocamllex によって生成される．規則としては正則表現とそれにマッチした際のアクションを (OCaml 式で) 記述する．アクションは，基本的には (parser.mly で宣言された) トークン (Parser.token 型) を返す

[11]有限状態オートマトンについては，京大の情報学科では「言語・オートマトン」という講義で習うはずである．

ような式を記述する．また，字句解析に使用する文字列バッファが lexbuf という名前で使えるが，通常は以下の使用法でしか使われない．

- Lexing.lexeme lexbuf で，正則表現にマッチした文字列を取り出す．
- Lexing.lexeme_char lexbuf n で，マッチした文字列の n 番目の文字を取り出す．
- Lexing.lexeme_start lexbuf で，マッチした文字列の先頭が入力文字列全体でどこに位置するかを返す．末尾の位置は Lexing.lexeme_end lexbuf で知ることができる．
- ⟨ エントリポイント ⟩ lexbuf で，⟨ エントリポイント ⟩ 規則を呼び出す．

それでは，具体例 lexer.mll を使って説明を行う．ヘッダ部では，予約語の文字列と，それに対応するトークンの連想リストである，reservedWords を定義している．後でみるように，List.assoc 関数を使って，文字列からトークンを取り出すことができる．

エントリポイント定義部分では，main という (唯一の) エントリポイントが定義されている．最初の正則表現は空白やタブなど文字の列にマッチする．これらは ML では区切り文字として無視するため，トークンは生成せず，後続の文字列から次のトークンを求めるために main lexbuf を呼び出している．次は，数字の並びにマッチし，int_of_string を使ってマッチした文字列を int 型に直して，トークン INTV (属性は int 型) を返す．続いているのは，記号に関する定義である．次は識別子のための正則表現で，英小文字で始まる名前か，演算記号にマッチする．アクション部では，マッチした文字列が予約語に含まれていれば，予約語のトークンを，そうでなければ (例外 Not_found が発生した場合は) ID トークンを返す．最後の eof はファイルの末尾にマッチする特殊なパターンである．ファイルの最後に到達したら exit するようにしている．

なお，この部分は，今後もあまり変更が必要がないので，正則表現を記述するための表現についてはあまり触れていない．興味のあるものは lex を解説した本や OCaml マニュアルを参照すること．

3.2.5 Environment モジュールと Eval モジュール：環境と解釈部

式の表す値 さて，本節冒頭でも述べたように，解釈部は，定義される言語のセマンティクスを定めている．プログラミング言語のセマンティクスを定めるに当たって重要なことは，どんな類いの**値** (*value*) を（定義される言語の）プログラムが操作できるかを定義することである．例えば，C 言語であれば整数値，浮動小数値，ポインタなどが値として扱えるし，OCaml であれば整数値，浮動小数値，レコード，ヴァリアント，クロージャ，オブジェクトなどが値として扱える．

この時**式の値** (*expressed value*) の集合と**変数が指示する値** (*denoted value*) を区別することがある．前者は式を評価した結果得られる値であり，後者は変数が指しうる値である．この 2 つの区別は，普段あまり意識することはないかもしれないし，実際に今回の実験を行う範囲で実装する機能の範囲では，このふたつは一致する（式の値の集合 = 変数が指示する値

```
{
let reservedWords = [
  (* Keywords in the alphabetical order *)
  ("else", Parser.ELSE);
  ("false", Parser.FALSE);
  ("if", Parser.IF);
  ("then", Parser.THEN);
  ("true", Parser.TRUE);
]
}

rule main = parse
  (* ignore spacing and newline characters *)
  [' ' '\009' '\012' '\n']+     { main lexbuf }

| "-"? ['0'-'9']+
    { Parser.INTV (int_of_string (Lexing.lexeme lexbuf)) }

| "(" { Parser.LPAREN }
| ")" { Parser.RPAREN }
| ";;" { Parser.SEMISEMI }
| "+" { Parser.PLUS }
| "*" { Parser.MULT }
| "<" { Parser.LT }

| ['a'-'z'] ['a'-'z' '0'-'9' '_' '']*
    { let id = Lexing.lexeme lexbuf in
      try
        List.assoc id reservedWords
      with
      _ -> Parser.ID id
     }
| eof { exit 0 }
```

図 3.3: ML[1] インタプリタ: `lexer.mll`

の集合）．しかし，この 2 つが異なる言語も珍しくない.[12]例えば，C 言語では，変数は，値そのものにつけられた名前ではなく，値が格納された箱につけられた名前と考えられる．そのため，denoted value は expressed value への参照と考えるのが自然になる．ML[1] の場合，式の表しうる集合 Expressed Value は

$$\begin{aligned} \text{Expressed Value} &= \text{整数}\,(\ldots,-2,-1,0,1,2,3,\ldots) \oplus \text{真偽値} \\ \text{Denoted Value} &= \text{Expressed Value} \end{aligned}$$

と与えられる．$\oplus$ は直和を示している．

このことを表現した OCaml の型宣言を以下に示す．

```
(* Expressed values *)
type exval =
    IntV of int
  | BoolV of bool
and dnval = exval
```

環境　もっとも簡単な解釈部の構成法のひとつは，抽象構文木と，変数と denoted value の束縛[13]関係の組から，実行結果を計算する方式である．この，変数の束縛を表現するデータ構造を**環境** (*environment*) といい，この方式で実装されたインタプリタ (解釈部) を**環境渡しインタプリタ** (*environment passing interpreter*) ということがある．

環境はここでは変数と denoted value の束縛を表現できれば充分なのだが，あとで用いる型推論においても，変数に割当てられた型を表現するために同様の構造を用いるので，汎用性を考えて，環境の型を多相型 `'a t` とする．ここで `'a` は変数に関連付けられる情報（ここでは denoted value）の型である．こうすることで，同じデータ構造を変数の denoted value への束縛としても，変数の別の情報への束縛としても使用することができるようになる．

環境を操作する値や関数の型，例外を示す.（`environment.mli` の内容である.）

```
type 'a t
exception Not_bound
val empty : 'a t
val extend : Syntax.id -> 'a -> 'a t -> 'a t
val lookup : Syntax.id -> 'a t -> 'a
val map : ('a -> 'b) -> 'a t -> 'b t
val fold_right : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

最初の値 `empty` は，何の変数も束縛されていない，空の環境である．次の `extend` は，環境に新しい束縛をひとつ付け加えるための関数で，`extend id dnval env` で，環境 `env` に対して，変数 `id` を denoted value `dnval` に束縛したような新しい環境を表す．関数 `lookup` は，環境から変数が束縛された値を取り出すもので，`lookup id env` で，環境 `env` の中を，

[12] この二種類の値の区別はコンパイラの教科書で見られる**左辺値** (*L-value*)，**右辺値** (*R-value*) と関連する．

[13] 一般に変数 x が何らかの情報 v に結び付けられていることを x **が** v **に束縛されている** (x *is bound to* v) と言う．値 v が変数 x に束縛されている**とはいわない**ので注意すること．

```
type 'a t = (Syntax.id * 'a) list

exception Not_bound

let empty = []
let extend x v env = (x,v)::env

let rec lookup x env =
  try List.assoc x env with Not_found -> raise Not_bound

let rec map f = function
    [] -> []
  | (id, v)::rest -> (id, f v) :: map f rest

let rec fold_right f env a =
  match env with
      [] -> a
    | (_, v)::rest -> f v (fold_right f rest a)
```

図 3.4: ML[1] インタプリタ: 環境の実装 (environment.ml)

新しく加わった束縛から順に変数 id を探し，束縛されている値を返す．変数が環境中に無い場合は，例外 Not_bound が発生する．

また，関数 map は，map f env で，各変数が束縛された値に f を適用したような新しい環境を返す．fold_right は環境中の値を新しいものから順に左から並べたようなリストに対して fold_right を行なう．これらは，後に型推論の実装などで使われる．

この関数群を実装したものが図 3.4 である．環境のデータ表現は，単なる連想リストである．ただし，environment.mli では 'a t の定義を示していないので，環境を使う側は，その事実を活用することはできない．

> **「実装の隠蔽」について** environment.mli と environment.ml の関係を理解しておくのはとても重要なので，ここで少し説明しておこう．どちらも Environment モジュールを定義するために用いられるファイルなのだが，environment.ml は Environment モジュールの**実装** (*implementation*) を定義し，[14]environment.mli はこのモジュールの**インターフェイス** (*interface*) を宣言する．[15]
>
> これを頭に入れて，environment.ml と environment.mli を見返してみよう．environment.ml は型 'a t を連想リスト (Syntax.id * 'a) list として定義し，'a t 型の値を操作する関数を定義している．これに対して，environment.mli は (1) なんらかの多相型 'a t が**存在する**ことのみを宣言しており，この型の実体が何であるかには言及しておらず，(2) 各関数の型を 'a t を用いて宣言している．(.mli ファイル中の各関数の型宣

[14]すなわち，environment.ml は，Environment モジュールがどう動作するかを決定している．

[15]すなわち，environment.mli は，このモジュールがどのように使われてよいかを決定している．一般にインターフェイスとは，2 つ以上のシステムが相互に作用する場所のことを言う．Environment モジュールの内部動作と外部仕様との相互作用を environment.mli が決めているわけである．

言は 'a t の実体が (Syntax.id * 'a) list であることには言及していないことに注意.)

Environment モジュール中の定義を使用するモジュール（例えばあとで説明する Eval モジュールなど）は，environment.mli ファイルに書かれている定義のみを，書かれている型としてのみ使うことができる．例えば Environment モジュールの empty という変数を Environment モジュールの外から使う際には Environment.empty という名前で参照することになる．Environment.empty は 'a t 型なのでリストとして使うことはできない．すなわち，environment.ml 内で 'a t がリストとして実装されていて empty が [] と実装されているにも関わらず，1 :: Environment.empty という式は型エラーになる．

なぜこのように実装とインターフェイスを分離する言語機構が提供されているのだろうか．一般によく言われる説明はプログラムを変更に強くするためである．例えば，開発のある時点で Environment モジュールの効率を上げるために，'a t 型をリストではなく二分探索木で実装し直したくなったとしよう．今の実装であれば，'a t 型が実際はどの型なのかがモジュールの外からは隠蔽されているので，environment.ml を修正するだけでこの変更を実装することができる．このような隠蔽のメカニズムがなかったとしたら，Environment モジュールを使用する関数において，'a t 型がリストであることに依存した記述を行うことが可能となる．そのようなプログラムを書いてしまうと，二分木の実装への変更を行うためには全プログラム中の Environment モジュールを利用しているすべての箇所の修正が必要になる．この例から分かるように，実装とインターフェイスを分離して，モジュール外には必要最低限の情報のみを公開することで，変更に強いプログラムを作ることができる．

以下は後述する main.ml に記述されている，プログラム実行開始時の環境 (大域環境) の定義である．

```
let initial_env =
  Environment.extend "i" (IntV 1)
    (Environment.extend "v" (IntV 5)
       (Environment.extend "x" (IntV 10) Environment.empty))
```

i，v，x が，それぞれ 1，5，10 に束縛されていることを表している．この大域環境は主に変数参照のテスト用で，(空でなければ) 何でもよい．

解釈部の主要部分　以上の準備をすると，残りは，二項演算子によるプリミティブ演算を実行する部分と式を評価する部分である．前者を apply_prim, 後者を eval_exp という関数として図 3.5 のように定義する．eval_exp では，整数・真偽値リテラル (ILit, BLit) はそのまま値に，変数は Environment.lookup を使って値を取りだし，プリミティブ適用式は，引数となる式 (オペランド) をそれぞれ評価し apply_prim を呼んでいる．apply_prim は与えられた二項演算子の種類にしたがって，対応する OCaml の演算をしている．if 式の場合には，まず条件式のみを評価して，その値によって then 節/else 節の式を評価している．関数 err は，エラー時に例外を発生させるための関数である (eval.ml 参照のこと).

eval_decl は ML[1] の範囲では単に式の値を返すだけのものでよいのだが，後に，let 宣言などを処理する時のことを考えて，新たに宣言された変数名 (ここではダミーの "-") と宣言によって拡張された環境を返す設計になっている．

```
let rec apply_prim op arg1 arg2 = match op, arg1, arg2 with
    Plus, IntV i1, IntV i2 -> IntV (i1 + i2)
  | Plus, _, _ -> err ("Both arguments must be integer: +")
  | Mult, IntV i1, IntV i2 -> IntV (i1 * i2)
  | Mult, _, _ -> err ("Both arguments must be integer: *")
  | Lt, IntV i1, IntV i2 -> BoolV (i1 < i2)
  | Lt, _, _ -> err ("Both arguments must be integer: <")

let rec eval_exp env = function
    Var x ->
      (try Environment.lookup x env with
        Environment.Not_bound -> err ("Variable not bound: " ^ x))
  | ILit i -> IntV i
  | BLit b -> BoolV b
  | BinOp (op, exp1, exp2) ->
      let arg1 = eval_exp env exp1 in
      let arg2 = eval_exp env exp2 in
      apply_prim op arg1 arg2
  | IfExp (exp1, exp2, exp3) ->
      let test = eval_exp env exp1 in
        (match test with
            BoolV true -> eval_exp env exp2
          | BoolV false -> eval_exp env exp3
          | _ -> err ("Test expression must be boolean: if"))

let eval_decl env = function
    Exp e -> let v = eval_exp env e in ("-", env, v)
```

図 3.5: ML[1] インタプリタ: 評価部の実装 (`eval.ml`) の抜粋

3.2.6 main.ml

メインプログラム main.ml を図 3.6 に示す．関数 read_eval_print で，

1. 入力文字列の読み込み・構文解析
2. 解釈
3. 結果の出力

処理を繰返している．まず，let decl = の右辺で字句解析部・構文解析部の結合を行っている．lexer.mll で宣言された規則の名前 main が関数 Lexer.main に，parser.mly (の %start) で宣言された非終端記号の名前 toplevel が関数 Parser.toplevel に対応している．これらの関数はそれぞれ ocamllex と Menhir によって自動生成された関数である．Parser.toplevel は第一引数として構文解析器から呼び出す字句解析器を，第二引数として読み込みバッファを表す Lexing.lexbuf 型の値を取る．標準ライブラリの Lexing モジュールの説明を読むと分かるが，Lexing.lexbuf の作り方にはいくつか方法がある．ここでは標準入力から読み込むため Lexing.from_channel を使って作られている．pp_val は eval.ml で定義されている，値をディスプレイに出力するための関数である．

> **標準ライブラリ** 本書を書いている時点では，OCaml の標準ライブラリは http://ocaml.org/ で “Documentation” → “OCaml Manual” → “The standard library” の順にリンクをたどると出て来る．このページには標準ライブラリで提供されている関数がモジュールごとに説明されている．
>
> なお，OCaml の標準ライブラリは必要最低限の関数のみが提供されているため，OCaml でソフトウェアを作る際にはその他のライブラリの力を借りることが多い．様々なライブラリをパッケージマネージャの opam を用いてインストールすることができる．

Exercise 3.2.1 ML[1] インタプリタのプログラムをコンパイル・実行し，インタプリタの動作を確かめよ．大域環境として i, v, x の値のみが定義されているが，ii が 2，iii が 3，iv が 4 となるようにプログラムを変更して，動作を確かめよ．例えば，

```
iv + iii * ii
```

などを試してみよ．

Exercise 3.2.2 [⋆⋆] このインタプリタは文法にあわない入力を与えたり，束縛されていない変数を参照しようとすると，プログラムの実行が終了してしまう．このような入力を与えた場合，適宜メッセージを出力して，インタプリタプロンプトに戻るように改造せよ．

Exercise 3.2.3 [⋆] 論理値演算のための二項演算子 &&, || を追加せよ．

Exercise 3.2.4 [⋆⋆] lexer.mll を改造し，(* と *) で囲まれたコメントを読み飛ばすようにせよ．なお，OCaml のコメントは入れ子にできることに注意せよ．ocamllex のドキュメントを読む必要があるかもしれない．(ヒント：comment という再帰的なルールを lexer.mll に新しく定義するとよい．)

```
open Syntax
open Eval

let rec read_eval_print env =
  print_string "# ";
  flush stdout;
  let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
  let (id, newenv, v) = eval_decl env decl in
    Printf.printf "val %s = " id;
    pp_val v;
    print_newline();
    read_eval_print newenv

let initial_env =
  Environment.extend "i" (IntV 1)
    (Environment.extend "v" (IntV 5)
       (Environment.extend "x" (IntV 10) Environment.empty))

let _ = read_eval_print initial_env
```

図 3.6: ML[1] インタプリタ: `main.ml`