# 可逆プログラミング言語 ROOPL++ のインタープリタの実装

2017SE004 青柳 裕樹　2017SE057 新美 伊織　2017SE086 竹市 翔哉　2017SE098 脇坂 怜輝
指導教員：横山 哲郎

## 1　はじめに

可逆計算は反転可能な 2 方向性計算モデルである．つまり，計算過程においてどの状態からもその直前と直後の状態が一意に決まる．計算は一方向に進み，その中で，前の状態が一意に決まり，その状態へ到達できるように情報損失を避けながら計算が進む．したがって，可逆計算は，計算過程において完璧に情報を保つことができるので，マイクロプロセッサにおける算モデルである．つまり，計算過程においてどの状態からもその直前と直後の状態が一意に決まる．計算は一方向に進み，その中で，前の状態が一意に決まり，その状態へ到達できるように情報損失を避けながら計算が進む．したがって，可逆計算は，計算過程において完璧に情報を保つことができるので，マイクロプロセッサにおける放熱の問題の有効な解決策として，考えられている．

これまで，マイクロプロセッサの使用電力の最小化に対して多くの取り組みがされてきたが，現代の計算モデルでは理論上の限界が存在する．ランダウアの原理では，次のように主張されている．ビットの消去や 2 つの計算経路の結合など論理的に非可逆な操作には，情報処理機器やその環境において，必ずエントロピーの増加が伴う [1]．つまり，システム上での情報の消去は常にエネルギー消費の増加につながるということである．また，$n$ ビットの情報を消去するために必要なエネルギーの量は，$n \cdot K_B \cdot T \cdot \log 2$ である．ここで，$T$ は回路の温度，$K_B$ はボルツマン定数である．これはランダウアの限界として知られており，情報の消去に関わる計算のエネルギー消費量はこれが限界値とされている．しかし，可逆計算においては，情報の損失がないため，ランダウアの限界の対象とならない．つまり，可逆な計算のみを行うならば，エネルギーの消費量を小さくするのに限界はないということである．

現代の可逆化へのアプローチは，非可逆プログラムの可逆化のアイデアに基づいている．非可逆プログラムを可逆にするには，プログラム実行中のすべての中間の値のトレースを保存しておけばよい．このテクニックはランダウアの埋め込みとして知られている [1]．このテクニックでは，どのような非可逆プログラムにも適用でき，完璧な情報の保存を実現するとされている．しかし，このテクニックには，プログラムの実行時間に比例したスペースが必要であり，実用的ではないと言える．一方で可逆プログラミング言語は, このような埋め込みをする必要がなく，完全に可逆性が保証されているため実用的である. 可逆プログラミング言語は個々に可逆な実行ステップからできている．それらをある計算状態から次の計算状態への関数とみなしたとき，それらはすべて単射であり，前方決定性と後方決定性が保証されている．非可逆のプログラミング言語では，多対一の関係になりえるため，その逆関数は一対多となるため，後方決定性が保証されない．したがって，前の状態を一意に決定することができない．また, 可逆言語において，プログラムが逆プログラムをもつ場合，それぞれ 1 つの逆プログラムしかもたなく, 単純な再帰で逆変換が実現できる. したがって, プログラムを逆変換することに追加のコストがかからない.

以上のように可逆プログラミング言語は可逆計算の分野において重要であり, 現在では, 可逆論理回路の設計 [2] や可逆アルゴリズムの開発などに用いられている.

本研究では, 可逆オブジェクト指向プログラミング言語 ROOPL のインタープリタの開発を行う. そして最終的にオンラインインタープリタを実装することで, 誰でもこの言語を試せるようにする.

## 2　関連研究

### 2.1　可逆プログラミング言語

可逆プログラミング言語とは, プログラムの実行過程が必ず可逆になるように言語が設計されているプログラミング言語である.

可逆プログラミング言語の種類としては，可逆手続き型プログラミング言語の Janus[3]，可逆命令型プログラミング言語の R[4]，可逆関数型プログラミング言語の RFUN[5]，可逆アセンブリ言語の PISA[4], 可逆オブジェクト指向言語の Joule$^R$[6] や ROOPL[7] などがある.

#### 2.1.1　Janus

Janus は,1982 年に Christopher Lutz と Howard Derby によってカリフォルニア工科大学の授業で作られ,2007 年に [8] で形式化されたプログラミング言語である. また, 最初の可逆プログラミング言語であり, 他の可逆プログラミング言語の設計のためのモデルにもなっている.

これまで, 計算過程を可逆にするためには計算履歴を必要としていた. その観点で言えば Janus はコストをかけずに逆変換することができるため有用であると言える.

また, 可逆計算機において, エネルギー消費を最大限に減少させるには, 低レベルのハードウェアからソフトウェアレベルまで可逆にする必要がある. そのため, 非可逆アルゴリズムを Janus で記述したり, Janus を可逆機械語に変換し, 抽象機械やマイクロプロセッサで直接実行したりすることで効果を発すると考えられている.

Janus は, 局所的に可逆なプログラムの文と逆意味論への直接アクセスをもつ可逆手続き型言語であり, 現在では,

整数, 固定サイズの整数配列及び可変サイズの整数スタックの 3 つのデータ型を扱うことができ, 局所変数も実装されている. プログラムは main 関数とそれに続くいくつかの関数から成る. プログラムの開始点は main 関数であり,main 関数内で宣言された静的変数の値がプログラムの出力となる. 構文規則や意味規則で制限を与えることにより可逆性を実現しており, それにより, コストをかけず逆意味論へ直接アクセスでき, 逆実行することができる.

#### 2.1.2 R

R は,1999 年に Michael P.Frank の研究 [4] で開発された命令形可逆プログラミング言語である. また, その際に可逆アセンブラ言語 PISA へのコンパイラも開発された.

可逆計算機の作成の際, 直接, 機械語でプログラミングするより高水準言語でプログラミングすることが望まれる. それを実現するためのアプローチとして, 非可逆言語でプログラミングし, 可逆アーキテクチャで実行できるようにするためのインタープリタまたはトランスレータを使うことが 1 つの方法である. しかし, このアプローチの場合, 時間やスペースの増加が必要となり, オーバーヘッドが発生する. たとえ, 可逆アルゴリズムが元の非可逆アルゴリズムと等しい効率でも, 一般的に, それを最適な非可逆アルゴリズムに変換することができない. 一方で, 可逆言語のみを使用した場合, そのプログラムは, プログラマがアセンブリ言語を手動でコーディングした場合のものと比較して, オーバーヘッドが発生しないようにコンパイルできる. 従来の単純なプログラミング言語のための完全な PISA へのコンパイラの開発は時間がかかりすぎるため, 研究において,Flank は, この非常に単純な可逆命令型言語 R と R から PISA へのコンパイラを開発した.

R の構文は,LISP と C を混合したものであり, ネストされた文で表される. プログラムは main ルーチンと sub ルーチンから成り,main ルーチンがプログラムの開始点となる.

あるアルゴリズムを R で記述した場合のプログラムは, 同じアルゴリズムを可逆アセンブリ言語 PISA で記述した場合のプログラムと比べて, 非常に簡潔なプログラムとなる. 例えば, 以下の R で記述された単純な乗算ルーチンを PISA にコンパイルすると,66 個のアセンブリコード命令の列に変換される. 手動でコーディングした場合の PISA プログラムは 36 個のアセンブリ命令コード列で表すことができるが, それと比較しても R の方が簡潔になる.

```
(defsub mult (m1 m2 prod)
 ;; Use grade-school algorithm:
 (for pos = 0 to 31
 ; For each of the 32 bit-positions,
  (if (m1 & (1 << pos)) then
  ; if that bit of m1 is 1, then
   (prod += (m2 << pos)))))
   ; add m2, shifted over to that
   ; position, into prod.
```

したがって, 高水準言語の R でプログラミングし, コンパイルを行うことで PISA のコードを生成することは非常に有用である.

#### 2.1.3 RFUN

RFUN は, 可逆言語による研究の基礎とするために開発された言語で, 関数型言語 FUN に逆写像を呼び出す構文等を追加し可逆性を持たせた可逆関数型プログラミング言語である.

Holger Bock Axelsen, 横山 哲郎, Robert Glück によって開発された.

プログラムは, いくつかの関数定義から成り, 各関数は 1 つの引数をもつ.

#### 2.1.4 PISA

PISA は, 可逆コンピュータアーキテクチャ Pendulum のための可逆アセンブリ言語である.Pendulum では,3 つの専用レジスタを扱う.

1. 現在の命令のアドレスを格納するためのプログラムカウンタ (PC2)
2. ジャンプオフセットを格納するための分岐レジスター (BR3)
3. 実行方向を追跡するための方向ビット (DIR)

#### 2.1.5 Joule$^R$

Joule$^R$ は,2016 年に提案された可逆オブジェクト指向プログラミング言語である.Joule$^R$ は,Janus にクラス, 継承, コンストラクタ及びカプセル化などのオブジェクト指向概念を拡張した言語である. また, リージョンベースのメモリ管理で実装しているので, メモリが完全にカベージフリーとなる. この言語は,Janus に変換することで実行される.

【要チェック：Theseus[9]. James と Sabry の言語 [10]. 並行計算の言語 [11]. ハードウェア記述言語（HDL）SyReC[12]，可逆オブジェクト指向言語 JOOL[13]/Joole$^R$[6]，サーベイ論文の可逆プログラミング言語の部分も参照 [14]】

### 2.2 可逆フローチャート

フローチャートとは，プロセスの各ステップを基本となる処理を長方形，条件分岐する処理は菱形など，特別な処理を意味する記号を組み合わせて表現し，流れをそれらの図形の間の矢印で表すことで，アルゴリズムやプロセスを表現する図である．フローチャートの書き方は，JIS（日本産業規格（旧名：日本工業規格））で規格化されている. 現在では, プログラムの構造や，プロセスを表すことにも使われている.

一方で, 可逆フローチャートは，可逆プログラミング言語の構造やプロセスを表すのに適したフローチャートである．通常のフローチャートと異なり，演算が可逆なものだけに制限され，また，フローの結合点では，必ず可逆化のための条件式が必要となる．可逆フローチャートは通常の

フローチャートに可逆化を適応することにより，非可逆フローチャートもシミュレートすることができる．

#### 2.2.1 SRL, RL

SRL と RL は，可逆フローチャートを表す可逆プログラミング言語である [15]．SRL は if 文や繰り返し文などからなる高水準の構造化プログラミング言語である．RL はジャンプ命令や goto 文などのブロックのかたまりからなる低水準の非構造化プログラミング言語である．また，これらは相互に変換可能である．

### 2.3 可逆オブジェクト指向プログラミング言語 ROOPL

可逆オブジェクト指向プログラミング言語 ROOPL は，オブジェクト指向とユーザ定義型がサポートされている可逆プログラミング言語である [7]．ROOPL は静的に型付けされ，継承，カプセル化，部分型多相もサポートされている．Roopl（Reversible Object-Oriented Programming Language）は、2016 年後半に導入された.

### 2.4 可逆オブジェクト指向プログラミング言語 ROOPL++

可逆オブジェクト指向プログラミング言語 ROOPL++ は,ROOPL を拡張した言語である.[16]ROOPL では, オブジェクトが construct/destruct ブロック内でしか存在できなく, 使いにくいものになっていた. さらに, 配列型の不足により, 言語の表現力にも問題があった. このような欠点を解決するため,ROOPL++ が設計された.ROOPL++ は, 新しいオブジェクト定義文の追加により, オブジェクトが construct/destruct ブロック外でも存在でき, 配列型も追加されているため, 複雑な可逆プログラムの作成が可能である.

#### 2.4.1 構文

ROOPL++ の構文と構文領域を図 1 に示す.ROOPL++ のプログラムは, フィールドとメソッドを持つ 1 つ以上のクラス定義から成る. プログラムは引数のない main メソッドから実行され,main メソッドが含まれているクラスのフィールドの値がプログラムの実行結果となる.

#### 2.4.2 意味領域と意味関数

ROOPL++ の意味を表示的意味論で記述する．意味領域を図 2 に, 意味関数を図 3 示す.

*Locs* は環境やストアで用いられるロケーションを表し, その値は非負整数である.*Envs* は環境を表し, 識別子からロケーションへの部分関数である.*ClassEnvs* はクラス環境を表し, クラス名からフィールドとメソッドの組への部分関数である.*Stores* はストアを表し, ロケーションから値への部分関数である. また, 環境やストアは演算子 $[\cdot \mapsto \cdot]$ で更新される. 演算子 $[\cdot \mapsto \cdot]$ の定義を以下に示す.

$$\gamma[x' \mapsto l](x) = \begin{cases} l & \text{if } x' = x \\ \gamma(x) & \text{if } x' \neq x \end{cases}$$

$$\mu[l' \mapsto v](l) = \begin{cases} v & \text{if } l' = l \\ \mu(l) & \text{if } l' \neq l \end{cases}$$

環境の省略記法 $[x_1 \mapsto l_1, \ldots, x_n \mapsto l_n]$ は, 識別子 $x_1$ から $x_n$ をロケーション $l_1$ から $l_n$ にそれぞれ対応付けた環境に更新することを表す. ストアの省略記法 $[l_1 \mapsto v_1, \ldots, l_n \mapsto v_n]$ も同様にロケーション $l_1$ から $l_n$ を値 $v_1$ から $v_n$ にそれぞれ対応付けたストアに更新することを表す．*Objects* はオブジェクトを表し, そのオブジェクトのクラス ID とそのオブジェクトのフィールドから値への部分関数の組である.*Values* は値を表し, それは, 整数, オブジェクト, ロケーション, ロケーションのベクトルのいずれかである.

意味関数 $\mathcal{E}$ は式を受け取り値を返す関数,$\mathcal{S}$ は文, 環境, クラス環境, ストアを受け取り文の実行により更新されたストアを返す関数,$\mathcal{P}$ はプログラムを受け取り環境とストアの組を返す関数である.

#### 2.4.3 意味関数 $\mathcal{E}$ の定義

意味関数 $\mathcal{E}$ の各式に対する定義を図 4 に示す. 整数 $n$ の場合, 値としての整数 $\overline{n}$ が返される. 変数 $x$ の場合, まず環境 $\gamma$ から $x$ に対応するロケーションを求める. そしてストア $\mu$ からそのロケーションに対応する値が返される.2 つの式の演算の場合, まず $e_1, e_2$ を評価し, その 2 つの値を関数 $[\![\otimes]\!]$ で評価した値を返す. ここで, 関数 $[\![\otimes]\!]$ の定義を図 5 に示す. 配列の場合, まず式 $e$ が評価され, インデックス $v$ を求める. そして, 環境 $\gamma$ とストア $\mu$ から識別子 $x$ に対応する, ロケーションが格納されている配列 $l'$ を求め, その配列からインデックスに対応するロケーションを求める. 最後に, ストア $\mu$ からそのロケーションに対応する値が返される.

#### 2.4.4 意味関数 $\mathcal{S}$ の定義

意味関数 $\mathcal{S}$ の各文に対する定義を図 6,7 に示す.

skip 文は, 実行ステップをスキップする文である. ストア $\mu$ に変化が起きずそのままのストアが返される.

代入文は, 変数に値を代入するための文である. この文では, 識別子 $x$ の値と式 $e$ の値を演算子 $\odot$ で評価し, 値 $(v)$ を得る. その後, 識別子 $x$ を $v$ に対応付けたストアが返される.

loop 文は, 繰り返し処理を行うための文である. この文では,$e_1$ の値が真 (0 でない) 場合, 文 $s_1$ が実行され, その後ストア $\mu'(s_1$ 実行後のストア) のもとで意味関数 $\mathcal{L}$ が実行される.$e_1$ が偽の場合は $_2$ も偽でなければならない. この条件式 $e_2$ は文を逆にする際に必要となり, これにより可逆化を実現することができる．また,loop 文の意味論に現れる意味関数 $\mathcal{L}$ は $\mathcal{S}$ と同様に環境, ストア, マップを受け取り, 更新されたストアを返す関数であり,loop 文の意味を簡潔

$$
\begin{array}{rcll}
prog & ::= & cl^{+} & \text{(program)} \\
cl & ::= & \textbf{class}\ c\ (\textbf{inherits}\ c)^{?}\ (t\ x)^{*}\ m^{+} & \text{(class definition)} \\
d & ::= & c \mid c\texttt{[}e\texttt{]} \mid \textbf{int}\texttt{[}e\texttt{]} & \text{(class and arrays)} \\
t & ::= & \textbf{int} \mid c \mid \textbf{int}\texttt{[]} \mid c\texttt{[]} & \text{(data type)} \\
y & ::= & x \mid x\texttt{[}e\texttt{]} & \text{(variable identifiers)} \\
m & ::= & \textbf{method}\ q\texttt{(}t\ x\texttt{, }\ldots\texttt{, }t\ x\texttt{)}\ s & \text{(method)} \\
s & ::= & y\ \odot\texttt{=}\ e \mid y\ \texttt{<=>}\ y & \text{(assignment)} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s\ \textbf{fi}\ e & \text{(conditional)} \\
 & \mid & \textbf{from}\ e\ \textbf{do}\ s\ \textbf{loop}\ s\ \textbf{until}\ e & \text{(loop)} \\
 & \mid & \textbf{construct}\ c\ x\quad s\quad \textbf{destruct}\ x & \text{(object block)} \\
 & \mid & \textbf{local}\ t\ x = e\quad s\quad \textbf{delocal}\ t\ x = e & \text{(local variable block)} \\
 & \mid & \textbf{new}\ d\ y \mid \textbf{delete}\ d\ y & \text{(object con- and destruction)} \\
 & \mid & \textbf{copy}\ d\ y\ y \mid \textbf{uncopy}\ d\ y\ y & \text{(reference con- and destruction)} \\
 & \mid & \textbf{call}\ q\texttt{(}x\texttt{, }\ldots\texttt{, }x\texttt{)} \mid \textbf{uncall}\ q\texttt{(}x\texttt{, }\ldots\texttt{, }x\texttt{)} & \text{(local method invocation)} \\
 & \mid & \textbf{call}\ x\texttt{::}q\texttt{(}x\texttt{, }\ldots\texttt{, }x\texttt{)} \mid \textbf{uncall}\ x\texttt{::}q\texttt{(}x\texttt{, }\ldots\texttt{, }x\texttt{)} & \text{(method invocation)} \\
 & \mid & \textbf{skip} \mid s\ s & \text{(statement sequence)} \\
e & ::= & \overline{n} \mid x \mid x\texttt{[}e\texttt{]} \mid \textbf{nil} \mid e\ \otimes\ e & \text{(expression)} \\
\odot & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{\^{}} & \text{(operator)} \\
\otimes & ::= & \odot \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{<} \mid \texttt{>} \mid \texttt{=} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} & \text{(operator)}
\end{array}
$$

$$
\begin{array}{lll}
prog \in \mathit{Programs} & s \in \text{Statements} & n \in \text{Constants} \\
cl \in \text{Classes} & e \in \text{Expressions} & x \in \text{VarIDs} \\
t \in \text{Types} & \odot \in \text{ModOps} & q \in \text{MethodIDs} \\
m \in \text{Methods} & \otimes \in \text{Operators} & c \in \text{ClassIDs}
\end{array}
$$

図 1 ROOPL++ の構文と構文領域 ([16] より転載)

$$
\begin{array}{rl}
l : \mathit{Locs} & = \mathbb{N}_0 \\
\gamma : \mathit{Envs} & = (\mathit{VarIDs} \to \mathit{Locs}) \\
\mu : \mathit{Stores} & = (\mathit{Locs} \to \mathit{Values}) \\
\Gamma : \mathit{ClassEnvs} & = (\mathit{ClassIDs} \to (\mathit{fs} \times \mathit{ms}))\ ) \\
\mathit{fs} : \mathit{Fields} & = \{t_1\ f_1\ \cdots\ t_n\ f_n\} \\
\mathit{ms} : \mathit{Methods} & = \{\textbf{method}\ q_1\ (\cdots)\ s_1\ \cdots\ \textbf{method}\ q_n\ (\cdots)\ s_n\} \\
\mu : \mathit{Stores} & = (\mathit{Locs} \to \mathit{Values}) \\
\mathit{Objects} & = \{\ \langle c_f,\ \gamma_f \rangle \mid c_f \in \mathit{ClassIDs}\ \wedge\ \gamma_f \in \mathit{Envs}\ \} \\
v : \mathit{Values} & = \mathbb{Z} \cup\ \mathit{Objects} \cup\ \mathit{Locs} \cup\ [\mathit{Locs}]
\end{array}
$$

図 2 意味領域 ([16] から一部改変)

に記述するために定義されている.$\mathcal{L}$ は,$e_2$ の値が偽 (0) の場合,$s_2$,$s_1$ の順に文が実行されその結果であるストアのもとで再び $\mathcal{L}$ が実行される.$e_2$ の値が真である場合は, ループが終了しそのままのストアが返される.

if 文は, 条件分岐処理を行うための文である. この文で

$$
\begin{array}{l}
\mathcal{E} : \mathit{Expressions} \to (\mathit{Envs} \times \mathit{Stores}) \to \mathit{Values} \\
\mathcal{S} : \mathit{Statement} \to (\mathit{Envs} \times \mathit{ClassEnvs}) \to \mathit{Stores} \to \mathit{Stores} \\
\mathcal{P} : \mathit{Programs} \to (\mathit{Envs} \times \mathit{Stores})
\end{array}
$$

図 3 意味関数 ([16] を参考に作成)

は,$e_1$ の値が真である場合, 文 $s_1$ が実行されその結果のストアが返される.($e_2$ の値も真とならなければならない.)$e_1$ の値が偽である場合, 文 $s_2$ が実行されその結果のストアが返される.($e_2$ の値も偽とならなければならない.)

ローカルメソッドの呼び出し文の call は, 現在のクラスに定義されているメソッドを呼び出すための文である. ここではまず, 識別子 this に対応する現在のクラスのオブジェクト (クラス名 $c$ とクラスフィールド $\gamma'$ の組) を取り出す. 次に, クラス環境から現在のクラスのフィールドとメソッドを取り出し, 呼び出されているメソッド $q$ を探し出す. そして, メソッド $q$ の仮引数に実引数のロケーションを対応付けたものを $\gamma'$ に追加し, 拡張する ($\gamma''$). この操作は

$$
\begin{aligned}
\mathcal{E}[\![n]\!](\gamma,\mu) &= \overline{n} \\
\mathcal{E}[\![x]\!](\gamma,\mu) &= \mu(\gamma(x)) \\
\mathcal{E}[\![nil]\!](\gamma,\mu) &= 0 \\
\mathcal{E}[\![e_1 \otimes e_2]\!](\gamma,\mu) &= [\![\otimes]\!](\mathcal{E}[\![e_1]\!](\gamma,\mu), \mathcal{E}[\![e_2]\!](\gamma,\mu)) \\
\mathcal{E}[\![x\,[\,e\,]\,]\!](\gamma,\mu) &= \mu(l'\,[\,v\,]\,) \\
&\quad \text{where } v = \mathcal{E}[\![e]\!](\gamma,\mu),\ l' = \mu(\gamma(x))
\end{aligned}
$$

図 4 意味関数 $\mathcal{E}$ の定義 ([16] から一部改変)

実引数を参照渡ししていることを表す. 最後に, 環境 $\gamma''$ とクラス環境 $\Gamma$ のもとでメソッドの本体を実行し, その結果のストアを返す.

ローカルメソッドの逆呼び出し文の uncall は, メソッドを逆実行し, その結果のストアを返す文である.

ローカルでないメソッド呼び出し文の call は, 指定したオブジェクトに対してメソッドを呼び出す文である. この文での処理は, ローカルメソッド呼び出しとほとんど等しく, オブジェクトを取り出すまでの処理が異なる.

ローカルでないメソッド逆呼び出し文の uncall は, ローカルメソッド逆呼び出し文と同じく, メソッドを逆実行し, その結果のストアを返す文である.

construct/destruct 文は, 指定されたクラスをインスタンス化するための文である. この文では, を未使用のロケーション $l'$ に指定されたクラス $c$ のオブジェクトを対応付け, クラスフィールドの値をゼロに初期化する. そして, 文 s を実行し, 最後にオブジェクトのクラスフィールドをゼロに初期化する.

new 文と delete 文は, オブジェクトをインスタンス化するための文である.construct/destruct 文と異なり,new 文と delete 文が独立しているため, 扱いやすくなっている.new 文では, ストアを指定されたクラス $c$ のオブジェクトを未使用のロケーションに対応付けたものに更新する. 一方で delete 文では,new 文で更新したストアを更新する前に戻す処理をする.

copy 文は参照をコピーするための文である. つまり,copy $x$ $x'$ の場合,$x'$ の参照先が $x$ の参照先と等しくなる. これは,$x$ のロケーションと対応している値を $x'$ のロケーションに対応付けることで実現している.

local/delocal 文は, 局所変数を定義するための文である. この文では, 識別子 $x$ を未使用のロケーション $r$ に対応付けた環境とそのロケーション $r$ に式 $e_1$ の値を対応付けたストアの下で文 $s$ を実行する.$s$ の実行後, 変数 $x$ の値が式 $e_2$ の値と等しいことを確認し, 識別子 $x$ のロケーションをストアから取り除く. ここで, 関数 $remove$ の定義は以下である.

$$remove(l,\mu)(l) = \mu(l)$$

$$
\begin{aligned}
[\![\texttt{+}]\!](v_1,\ v_2) &= v_1 \ + \ v_2 \\
[\![\texttt{\%}]\!](v_1,\ v_2) &= v_1 \ mod \ v_2 \\
[\![\texttt{-}]\!](v_1,\ v_2) &= v_1 \ - \ v_2 \\
[\![\texttt{\&}]\!](v_1,\ v_2) &= v_1 \ and \ v_2 \\
[\![\texttt{*}]\!](v_1,\ v_2) &= v_1 \ \times \ v_2 \\
[\![\texttt{|}]\!](v_1,\ v_2) &= v_1 \ or \ v_2 \\
[\![\texttt{/}]\!](v_1,\ v_2) &= \frac{v_1}{v_2} \\
[\![\texttt{\^{}}]\!](v_1,\ v_2) &= v_1 \ xor \ v_2 \\
[\![\texttt{\&\&}]\!](v_1,\ v_2) &= \begin{cases} 0 & \text{if } v_1 = 0 \vee v_2 = 0 \\ 1 & \text{otherwise} \end{cases} \\
[\![\texttt{<=}]\!](v_1,\ v_2) &= \begin{cases} 1 & \text{if } v_1 \leq v_2 \\ 0 & \text{otherwise} \end{cases} \\
[\![\texttt{||}]\!](v_1,\ v_2) &= \begin{cases} 0 & \text{if } v_1 = v_2 = 0 \\ 1 & \text{otherwise} \end{cases} \\
[\![\texttt{>=}]\!](v_1,\ v_2) &= \begin{cases} 1 & \text{if } v_1 \geq v_2 \\ 0 & \text{otherwise} \end{cases} \\
[\![\texttt{<}]\!](v_1,\ v_2) &= \begin{cases} 1 & \text{if } v_1 < v_2 \\ 0 & \text{otherwise} \end{cases} \\
[\![\texttt{=}]\!](v_1,\ v_2) &= \begin{cases} 1 & \text{if } v_1 = v_2 \\ 0 & \text{otherwise} \end{cases} \\
[\![\texttt{>}]\!](v_1,\ v_2) &= \begin{cases} 1 & \text{if } v_1 > v_2 \\ 0 & \text{otherwise} \end{cases} \\
[\![\texttt{!=}]\!](v_1,\ v_2) &= \begin{cases} 1 & \text{if } v_1 \neq v_2 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

図 5 関数 $[\![\otimes]\!]$ の定義 ([16] より転載)

#### 2.4.5 意味関数 $\mathcal{P}$ の定義

意味関数 $\mathcal{P}$ の定義を図 8 に関数 gen の定義を図 9 にその他の関数を図 10 に示す.

関数 gen は, 複数のクラス定義を受け取り, クラス環境 (クラス名をそのクラスのフィールドとメソッドの組に対応付けたもの) を生成する関数である. また, 関数 gen で使われる関数 $\alpha$ は, クラス定義からクラス名を返す関数である. 関数 $\beta$ は, 関数 fields と関数 methods の組を返す関数である. 関数 fields はクラス定義からそのクラスの全てのフィールドを返す. 継承したクラスの場合, 継承元のフィールドも含まれて返される. 関数 methods はクラス定義からそのクラスで定義された全てのメソッドを返す. 継承したクラスの場合, 継承元のメソッドも含まれ, また, 継承元と同じ関数がある場合は, 継承先のメソッドの方が返される (オーバーライド).

意味関数 $\mathcal{P}$ では, まず, 関数 gen によりクラス定義から

$\mathcal{S}[\![\textbf{skip}]\!]_{\gamma,\Gamma}(\mu) = \mu$

$\mathcal{S}[\![x \ \odot\texttt{=}e]\!]_{\gamma,\Gamma}(\mu) = \mu[\gamma(x) \mapsto v]$
where $v = [\![\odot]\!](\ \mu(\gamma(x)), \mathcal{E}[\![e]\!](\gamma,\mu)\ )$

$\mathcal{S}[\![x_1\texttt{<=>}x_2]\!]_{\gamma,\Gamma}(\mu) = \mu[\gamma(x_1) \mapsto v_2,\ \gamma(x_2) \mapsto v_1]$
where $v_1 = \mu(\gamma(x_1)),\ v_2 = \mu(\gamma(x_2))$

$\mathcal{S}[\![\textbf{from}\ e_1\ \textbf{do}\ s_1\ \textbf{loop}\ s_2\ \textbf{until}\ e_2]\!]_{\gamma,\Gamma}(\mu)$
$= \mathcal{L}[\![(e_1, s_1, s_2, e_2)]\!]_{\gamma,\Gamma}(\mu')$ if $\mathcal{E}[\![e_1]\!](\gamma,\mu) \neq 0$
where $\mu' = \mathcal{S}[\![s_1]\!]_{\gamma,\Gamma}(\mu)$

$\mathcal{L}[\![(e_1, s_1, s_2, e_2)]\!]_{\gamma,\Gamma}(\mu)$
$$= \begin{cases} \mu & \text{if } \mathcal{E}[\![e_2]\!](\gamma,\mu) \neq 0 \\ \mathcal{L}[\![(e_1, s_1, s_2, s_2)]\!]_{\gamma,\Gamma}(\mu'') & \text{if } \mathcal{E}[\![e_2]\!](\gamma,\mu) = 0 \wedge \mathcal{E}[\![e_1]\!](\gamma,\mu') = 0 \\ & \quad \text{where } \mu'' = \mathcal{S}[\![s_1]\!]_{\gamma,\Gamma}(\mu'),\ \mu' = \mathcal{S}[\![s_2]\!]_{\gamma,\Gamma}(\mu) \end{cases}$$

$\mathcal{S}[\![\textbf{if}\ e_1\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ e_2]\!]_{\gamma,\Gamma}(\mu)$
$$= \begin{cases} \mu' & \text{if } (\mathcal{E}[\![e_1]\!](\gamma,\mu) \neq 0)\ \wedge\ (\mathcal{E}[\![e_2]\!](\gamma,\mu') \neq 0) \\ & \quad \text{where } \mu' = \mathcal{S}[\![s_1]\!]_{\gamma,\Gamma}(\mu) \\ \mu' & \text{if } (\mathcal{E}[\![e_1]\!](\gamma,\mu) = 0)\ \wedge\ (\mathcal{E}[\![e_2]\!](\gamma,\mu') = 0) \\ & \quad \text{where } \mu' = \mathcal{S}[\![s_2]\!]_{\gamma,\Gamma}(\mu) \end{cases}$$

$\mathcal{S}[\![\textbf{call}\ q\texttt{(}y_1\texttt{,}\ \ldots\texttt{,}\ y_n\texttt{)}]\!]_{\gamma,\Gamma}(\mu) = \mathcal{S}[\![s]\!]_{\gamma'',\Gamma}(\mu)$
where
$\mu(\gamma(this)) = \langle c,\ \gamma'\rangle \quad \Gamma(c) = \Big(\ fields,\ methods\ \Big)$
$\Big(\ \textbf{method}\ q\texttt{(}t_1\ z_1\texttt{,}\ \ldots\texttt{,}\ t_n\ z_n\texttt{)}\ s\ \Big)\ \in\ methods$
$\gamma'' = \gamma'\big[\ this \mapsto \gamma(this),\ z_1 \mapsto \gamma(y_1), \ldots,\ z_k \mapsto \gamma(y_k)\ \big]$

$\mathcal{S}[\![\textbf{uncall}\ q\texttt{(}y_1\texttt{,}\ \ldots\texttt{,}\ y_n\texttt{)}]\!]_{\gamma,\Gamma}(\mu) = \mu'$
where $\mathcal{S}[\![\textbf{call}\ q\texttt{(}y_1,\ \ldots\texttt{,}\ y_n\texttt{)}]\!]_{\gamma,\Gamma}(\mu') = \mu$

$\mathcal{S}[\![\textbf{call}\ x_0\ \texttt{::}\ q\texttt{(}y_1\texttt{,}\ \ldots\texttt{,}\ y_n\texttt{)}]\!]_{\gamma,\Gamma}(\mu) = \mathcal{S}[\![s]\!]_{\gamma'',\Gamma}(\mu)$
*where*
$l = \gamma(x_0) \quad l' = \mu(l) \quad \mu(l') = \langle c, \gamma'\rangle \quad \Gamma(c) = \Big(\ fields,\ methods\ \Big)$
$\Big(\ \textbf{method}\ q\texttt{(}t_1\ z_1\texttt{,}\ \ldots\texttt{,}\ t_n\ z_n\texttt{)}\ s\ \Big)\ \in\ methods$
$\gamma'' = \gamma'\big[\ this \mapsto \gamma(x_0),\ z_1 \mapsto \gamma(y_1), \ldots,\ z_n \mapsto \gamma(y_k)\ \big]$

$\mathcal{S}[\![\textbf{call}\ x\texttt{[}e\texttt{]}\ \texttt{::}\ q\texttt{(}y_1\texttt{,}\ \ldots\texttt{,}\ y_n\texttt{)}]\!]_{\gamma,\Gamma}(\mu) = \mathcal{S}[\![s]\!]_{\gamma'',\Gamma}(\mu)$
*where*
$v = \mathcal{E}[\![e]\!](\gamma,\mu) \quad l = \mu(\gamma(x))\texttt{[}v\texttt{]} \quad l' = \mu(l)$
$\mu(l') = \langle c,\ \gamma'\rangle,\ \Gamma(c) = \Big(\ fields,\ methods\ \Big)$
$\Big(\ \textbf{method}\ q\texttt{(}t_1\ z_1\texttt{,}\ \ldots\texttt{,}\ t_n\ z_n\texttt{)}\ s\ \Big)\ \in\ methods$
$\gamma'' = \gamma'\big[\ this \mapsto \gamma(x_0),\ z_1 \mapsto \gamma(y_1), \ldots,\ z_n \mapsto \gamma(y_k)\ \big]$

図 6 意味関数 $\mathcal{S}$ の定義 1([16] から一部改変)

$\mathcal{S}[\![\textbf{uncall } y \texttt{ :: } q\texttt{(}y_1\texttt{, } \ldots\texttt{, } y_n\texttt{)}]\!]_{\gamma,\Gamma}(\mu) = \mu'$
where $\mathcal{S}[\![\textbf{call } y \texttt{ :: } q\texttt{(}y_1\texttt{, } \ldots\texttt{, } y_n\texttt{)}]\!]_{\gamma,\Gamma}(\mu') = \mu$

$\mathcal{S}[\![\textbf{construct } c\ x \quad s \quad \textbf{destruct } x]\!]_{\gamma,\Gamma}(\mu) = \mu''[l'_1 \mapsto 0, \ldots, l'_n \mapsto 0]$
where
$\Gamma(c) = \Big( \overbrace{\{\langle t_1, f_1\rangle, \ldots, \langle t_n, f_n\rangle\}}^{fields}, \mathit{methods} \Big) \quad \gamma' = [f_1 \mapsto l'_1, \ldots, f_n \mapsto l'_n]$
$\{ l, l', l'_0, \ldots, l'_n \} \cap \mathrm{dom}(\mu) = \emptyset \quad \mu' = \mu\,[l'_1 \mapsto 0, \ldots, l'_n \mapsto 0, l \mapsto l', l' \mapsto \langle c, \gamma'\rangle]$
$\mu'' = \mathcal{S}[\![s]\!]_{\gamma[x\mapsto r]}(\mu') \quad \mu''(l'_1) = 0, \ldots, \mu''(l'_n) = 0$

$\mathcal{S}[\![x\texttt{[}e_1\texttt{]} \odot\texttt{=}e_2]\!]_{\gamma,\Gamma}(\mu) = \mu[l' \mapsto w']$
where $v_1 = \mathcal{E}[\![e_1]\!](\gamma,\mu),\ v_2 = \mathcal{E}[\![e_2]\!](\gamma,\mu),\ l' = \mu(\gamma(x))\texttt{[}v_1\texttt{]},\ w = \mu(l'),\ w' = [\![\odot]\!](w, v_2)$

$\mathcal{S}[\![\textbf{new } c\ x]\!]_{\gamma,\Gamma}(\mu) = \mu\Big[l \mapsto l_0,\ l_0 \mapsto \langle c, \gamma'\rangle,\ l_1 \mapsto 0, \ldots, l_m \mapsto 0\Big]$
where
$\Gamma(c) = \Big( \overbrace{\{\langle t_1, f_1\rangle, \ldots, \langle t_m, f_m\rangle\}}^{fields}, \mathit{methods} \Big) \quad \gamma' = [f_1 \mapsto l_0, \ldots, f_m \mapsto l_0]$
$\gamma(x) = l \quad \{ l_0, \ldots, l_m \} \cap \mathrm{dom}(\mu) = \emptyset$

$\mathcal{S}[\![\textbf{new } c\ x\texttt{[}e\texttt{]}]\!]_{\gamma,\Gamma}(\mu) = \mu\Big[l \mapsto l_0,\ l_0 \mapsto \langle c, \gamma'\rangle,\ l_1 \mapsto 0, \ldots, l_m \mapsto 0\Big]$
where
$\Gamma(c) = \Big( \overbrace{\{\langle t_1, f_1\rangle, \ldots, \langle t_m, f_m\rangle\}}^{fields}, \mathit{methods} \Big) \quad \gamma' = [f_1 \mapsto l_0, \ldots, f_m \mapsto l_0]$
$v = \mathcal{E}[\![e]\!](\gamma,\mu) \quad l = \mu(\gamma(x))[v] \quad \{l_0, \ldots, l_m\} \cap \mathrm{dom}(\mu) = \emptyset$

$\mathcal{S}[\![\textbf{delete } c\ y]\!]_{\gamma,\Gamma}(\mu) = \mu'$
where $\mathcal{S}[\![\textbf{new } c\ y]\!]_{\gamma,\Gamma}(\mu') = \mu$

$\mathcal{S}[\![\textbf{new } a\texttt{[}e\texttt{]}\ x]\!]_{\gamma,\Gamma}(\mu) = \mu\big[l \mapsto \{l'_1, \ldots, l'_n\},\ l'_1 \mapsto 0, \ldots, l'_n \mapsto 0\big]$
where $n = \mathcal{E}[\![e]\!](\gamma,\mu),\ l = \gamma(x),\ \{ l'_1, \ldots, l'_n \} \cap \mathrm{dom}(\mu) = \emptyset$

$\mathcal{S}[\![\textbf{delete } a\texttt{[}e\texttt{]}\ x]\!]_{\gamma,\Gamma}(\mu) = \mu'$
where $\mathcal{S}[\![\textbf{new } a\texttt{[}e\texttt{]}\ x]\!]_{\gamma,\Gamma}(\mu') = \mu$

$\mathcal{S}[\![\textbf{copy } c\ x\ x']\!]_{\gamma,\Gamma}(\mu) = \mu[l' \mapsto v]$
where $\gamma(x) = l,\ \gamma(x') = l',\ \mu(l) = v$

$\mathcal{S}[\![\textbf{uncopy } c\ x\ x']\!]_{\gamma,\Gamma}(\mu) = \mu'$
where $\mathcal{S}[\![\textbf{copy } c\ x\ x']\!]_{\gamma,\Gamma}(\mu') = \mu$

$\mathcal{S}[\![\textbf{local } c\ x = e_1 \quad s \quad \textbf{delocal } x = e_2]\!]_{\gamma,\Gamma}(\mu) = \mathit{remove}(r, \mu')$
where
$v_1 = \mathcal{E}[\![e_1]\!](\gamma,\mu),\ v_2 = \mathcal{E}[\![e_2]\!](\gamma,\mu),\ r \notin \mathrm{dom}(\mu)$
$\mu' = \mathcal{S}[\![s]\!]_{\gamma[x\mapsto r],\Gamma}(\mu[r \mapsto v_1]),\ \mu'' = \mu'[r \mapsto v_2],\ \mu''(x) = v_2$

図 7 意味関数 $\mathcal{S}$ の定義 2([16] から一部改変)

クラス環境が生成される. 次に, クラス環境から main メソッドが含まれているクラス $c$ を求める. そして, クラス名 $c$ のクラスの全てのフィールドにそれぞれロケーションの値を 1 から順に対応付け, 識別子 this を最後のロケーションの値 (i+1) に対応付けた環境 $\gamma$ を生成する. 次に, フィールドに対応する全てのロケーションに 0 を, ロケーション i+1 にロケーションの値 i+2 を, ロケーションの値 i+2 にクラス $c$ のオブジェクトを対応づけたストア $\mu$ を生成する. 最後に, 環境 $\gamma$ とストア $\mu$ の下で main メソッドの本体 $s$ を実行し, 環境 $\gamma$ と $s$ 実行後のストア $mu$ の組を返す.

### 2.5 可逆プログラミング言語の処理系

プログラミング言語の処理系とは, プログラミング言語で書かれたプログラムの実行や変換をするためのプログラムである. プログラムを逐次解析しながら実行するプログラムをインタプリタ, プログラムを機械語やその他のプログラミング言語に変換するプログラムをコンパイラまたはトランスレータという. 現在, 可逆プログラミング言語の処理系として, インタープリタは,Janus インタプリタ,PISA インタプリタの PHPISA や Pendulum VM,R-WHILE インタプリタなどが実装されている. コンパイラは,ROOPL を低水準言語の PISA に変換する ROOPLtoPISA,R 言語を Pendulum アセンブリ言語に変換する R コンパイラなどがある. 【以下も要チェック】

- ROOPL to PISA コペンハーゲン大 3 年生
- Pendulum VM (PISA インタープリタ) コペンハーゲン大 3 年生
- PISA オンラインインタプリタ PHPISA フロリダ大学学部生
- pendvm MIT の学生
- phpisa MIT の学生
- R コンパイラ MIT の学生
- R-WHIILE コペンハーゲン大の学生ら
- Janus ブレーメン大学の研究者やコペンハーゲン大の学生
- Syrec ブレーメン大学の研究者
- Bob コペンハーゲン大
- 量子プログラミング

## 3 目的

可逆な計算機の実現には, 可逆プログラミング言語は重要であり, また, 可逆言語のインタープリタは, 可逆アルゴリズムを開発する際に助けとなる. 我々の知る限りでは, 現段階では,ROOPL++ や JOULE といった可逆オブジェクト指向プログラミング言語は開発されているが, インタープリタは実装されていない. したがって, 可逆オブジェクト指向プログラミング言語を試すことができないというのが現状である. そのため, 本研究では ROOPL++ のインタープリタを実装し, そして, 誰でも試せるようにオンラインインタープリタを実装することを目的とする.

## 4 実装

今回は関数型言語の OCaml を用いてインタープリタの実装を行う. 実装の手順は以下の通りである.

1. データ型の定義
2. 字句解析器, 構文解析器の実装
3. inverter の実装
4. インタープリタの実装
5. オンラインインタープリタの実装

### 4.1 データ型の定義

初めに, 木構造でプログラムを処理するために, データ型を定義しなければならない.OCaml では,type 宣言を使うことでデータ型を定義できる.2 章の図 1 に従ってデータ型を定義する. データ型の定義の一部を図 11 に示す.

exp 型は ROOPL++ の構文の $e$,obj 型は $y$,stm 型は $s$,mDecl 型は $m$,cDecl 型は $cl$,prog 型は $prog$ に対応している. 例えば,stm 型のデータ構成子 Swap は, 引数に 2 つの obj 型を持つことを意味している. つまり,

$$x \text{ <=> } y$$

は,

```
Swap(("x",None),("y",None))
```

と表現される. ここで,obj 型や cDecl 型に出現する option 型は, データの有無を表すことのできる型であり,OCaml に標準で定義されているデータ型である. 定義は次のようになっている.

```
type 'a option = None | Some of 'a
```

データが必要ない場合は, データ構成子 None を使い, データが有る場合は,Some を使う. 例えば,

$$x[0] \text{ <=> } x[1]$$

は,

```
Swap(("x",Some(Const 0)),("x",Some(Const 1)))
```

と表現される. このように option 型を使うことによって,1 つの構成子で変数の場合と配列の場合を表現できる.

なお, 構文 $s$ の文の列 $s\ s$ は, リストで実現するため, データ構成子を定義していない.

### 4.2 字句解析器, 構文解析器の実装

プログラムを構文木にするためには字句解析器と構文解析器が必要である. 今回は, これらの実装のために ocamllex と ocamlyacc というツールを使う. このツールは, 専用の定義ファイルから字句解析器と構文解析器の役割を果たす OCaml プログラムを出力するツールである.

$$\mathcal{P}[\![cl_1 \ldots cl_n]\!] = (\gamma, \mu')$$
where
$$\Gamma = \mathrm{gen}(cl_1,\ \ldots,\ cl_n) \qquad \Gamma(c) = \Big(\ \overbrace{\{\langle t_1,\ f_1\rangle,\ \ldots,\ \langle t_i,\ f_i\rangle\}}^{fields},\ methods\ \Big)$$
$$\Big(\ \textbf{method main()}\ s\ \Big)\ \in\ methods \qquad \gamma = [f_1\ \mapsto\ 1,\ \ldots,\ f_i\ \mapsto\ i,\ this\ \mapsto\ i+1]$$
$$\mu = [1\ \mapsto\ 0,\ \ldots,\ i\ \mapsto\ 0,\ i+1\ \mapsto\ i+2,\ i+2\ \mapsto\ \langle c,\ \gamma\rangle] \qquad \mu' = \mathcal{S}[\![s]\!]_{\gamma,\Gamma}(\mu)$$

図 8 意味関数 $\mathcal{P}$ の定義([16] から一部改変)

$$\mathrm{gen}\Big(\overbrace{cl_1,\ ...,\ cl_n}^{p}\Big) = \overbrace{\Big[\alpha(cl_1) \mapsto \beta(cl_1),\ ...,\ \alpha(cl_n) \mapsto \beta(cl_n)\Big]}^{\Gamma}$$

$$\alpha\Big(\textbf{class}\ c\ \cdots\Big) = c \qquad \beta(cl) = \Big(\mathrm{fields}(cl),\ \mathrm{methods}(cl)\Big)$$

図 9 クラス環境生成関数 gen の定義 ([16] より転載)

$$\mathrm{fields}(cl) = \begin{cases} \eta(cl) & \text{if } cl \sim\ [\textbf{class}\ c\ \cdots] \\ \eta(cl) \cup \mathrm{fields}\Big(\alpha^{-1}(c')\Big) & \text{if } cl \sim [\textbf{class}\ c\ \textbf{inherits}\ c'\ \cdots] \end{cases}$$

$$\mathrm{methods}(cl) = \begin{cases} \delta(cl) & \text{if } cl \sim [\textbf{class}\ c\ \cdots] \\ \delta(cl) \uplus \mathrm{methods}\Big(\alpha^{-1}(c')\Big) & \text{if } cl \sim [\textbf{class}\ c\ \textbf{inherits}\ c'\ \cdots] \end{cases}$$

$$A\ \uplus B\ \stackrel{def}{=}\ A \cup \left\{ m \in B \;\middle|\; \nexists\ m'\Big(\zeta(m') = \zeta(m) \wedge m' \in A\Big) \right\}$$

$$\zeta\Big(\textbf{method}\ q\ (\cdots)\ s\Big) = q \qquad \eta\Big(\textbf{class}\ c\ \cdots\ \overbrace{t_1 f_1\ \cdots\ t_n f_n}^{fs}\ \cdots\Big) = fs$$

$$\delta\Big(\textbf{class}\ c\ \cdots \overbrace{\textbf{method}\ q_1\ (\cdots)\ s_1\ \cdots\ \textbf{method}\ q_n\ (\cdots)\ s_n}^{ms}\ \cdots\Big) = ms$$

図 10 fields と methods の定義 ([16] より転載)

#### 4.2.1 字句解析器

字句解析器とは, プログラムを字句の並びに変換するためのプログラムである. 今回使用する字句解析器を生成するツール ocamllex は, 字句や規則が定義されたファイルを入力として受け取る. 字句は正規表現で定義される. 定義ファイルの一部を図 12 に示す.

ファイルの先頭では, 字句のデータ型を定義する. 今回は, 他のファイルで定義したデータ型を使うので,open を用いてデータ型を読み込んでいる. 次の let 宣言では, 後に使う正規表現に名前をつけている. そして,rule token から始まる定義が規則の定義である. ここでは, 正規表現を定義し, カギカッコ内でそれにマッチした時に返す式を記述する.token は関数であり, まず, 文字を Lexing.lexbuf から読み,rule にある正規表現にマッチさせる. そして, そのマッチした正規表現に対応する式を評価し, 関数の結果としてその値を返す. 例えば, 文字列から整数の 1 を読み込んだとすると, 正規表現 digit+ にマッチする. そして,digit+ に対応するカギカッコ内の式を評価し, その値を返す. この式では, まず,

```
let str = Lexing.lexeme lexbuf in
```

より, 変数 str にマッチした文字列の 1 が束縛される.

```
Lexing.lexeme
```

は, マッチした文字列をそのまま返す関数である. そして,

```
type id = string
type typeId = id
type methodId = id

type exp =
  | Const of int
  | Var of id
  | ArrayElement of id * exp
  | Nil
  | Binary of binOp * exp * exp

type obj = id * exp option

type stm =
  | Assign of obj * modOp * exp
  | Swap of obj * obj
  | Conditional of exp * stm list * stm list * exp
  | Loop of exp * stm list * stm list * exp
  | ObjectBlock of typeId * id * stm list
  | LocalBlock of dataType * id * exp * stm list * exp
  | LocalCall of methodId * obj list
  | LocalUncall of methodId * obj list
  | ObjectCall of obj * methodId * obj list
  | ObjectUncall of obj * methodId * obj list
  | ObjectConstruction of typeId * obj
  | ObjectDestruction of typeId * obj
  | CopyReference of dataType * obj * obj
  | UncopyReference of dataType * obj * obj
  | ArrayConstruction of (typeId * exp) * id
  | ArrayDestruction of (typeId * exp) * id
  | Skip

type decl = Decl of dataType * id

type mDecl = MDecl of methodId * decl list * stm list

type cDecl = CDecl of typeId * typeId option * decl list * mDecl list

type prog = Prog of cDecl list
```

図 11　データ型の定義の一部

```
CONST (int_of_string str)
```

により,

```
CONST(1)
```

が返され,1 つの字句となる. ここで, 関数

```
int_of_string
```

は, 文字列の数字を整数に変換する関数である. また, スペースの部分では,

```
token lexbuf
```

により, 値を返さずに読み飛ばしている.

#### 4.2.2　構文解析器

構文解析器とは, 字句の列を構文木に変換するためのものプログラム. 今回使用する構文解析器を生成するツール ocamlyacc は, 評価規則を記述した専用のファイルを入力として受け取る. 構文評価規則の定義ファイルの一部を図に示す.

まず, ファイルの先頭では, 字句解析器と同じデータ型を使うために,open でファイルを読み込んでいる. また, 後に使う関数も定義している. 次に,%token で字句解析器で使った字句を宣言する. 次の %left,%nonassoc 及び %right では, 演算子の優先順位や結合の規則を定義したものである.%left は左結合,%right は右結合,%nonassoc は結合なしである. 下の行に記述された演算子ほど結合の優先順位が高い. 次の %start では, この文法のエントリポイントの記号を定義している. そして, 次の %type でエントリポイントの型を指定している. 最後に, 文法を定義する. 左辺にあるのが非終端記号である. 右辺では, 特定の字句列にマッチした場合の評価規則を定義している.

#### 4.2.3　inverter の実装

inverter は, プログラムを逆変換するためのプログラムである. これは,ROOPL++ の構文 uncall を実装するために必要である. プログラムの文や式の inverter の定義を図 13 に示す.

実装するには,OCaml 上で関数 $\mathcal{I}$ を定義すればよい. プログラムの文はリストに格納されているので, 要素を先頭から取り出し, パターンマッチングを用いて, 図 13 に従って変換すればよい. また, 逆実行する際には元のプログラムを逆から実行するので, 変換された文は, 元のリストとは逆の順に格納する.

#### 4.2.4　インタープリタの実装

インタープリタは,2.4 章の意味領域, 意味関数を実装することで実現する. まず, 式 Expressions を評価する関数 eval_exp を定義する.eval_exp の型は,

```
Syntax.exp ->
```

```
{
open Parser
open Syntax
}

let space = [' ' '\t' '\n' '\r']
let digit = ['0'-'9']
let alpha = ['A'-'Z' 'a'-'z' '_']
let alnum = digit | alpha | '\''

rule token = parse
  (* 定数 *)
  | digit+
    { let str = Lexing.lexeme lexbuf in
      CONST (int_of_string str) }

  (* コメント *)
  | "//" [^'\n']* { token lexbuf }

  (* 演算子 *)
  | '*'        { MUL }
  | '/'        { DIV }
  | '%'        { MOD }
  | '+'        { ADD }
  | '-'        { SUB }
  | '<'        { LT  }
  | "<="       { LE  }
  | '>'        { GT  }
  | ">="       { GE  }
  | '='        { EQ  }

  (* 括弧 *)
  | '('        { LPAREN }
  | ')'        { RPAREN }
  | '['        { LBRA }
  | ']'        { RBRA }

  (* キーワード *)
  | "class"     { CLASS }
  | "inherits"  { INHERITS }
  | "method"    { METHOD }
  | "call"      { CALL }
  | "uncall"    { UNCALL }
  | "construct" { CONSTRUCT }
  | "destruct"  { DESTRUCT }
  | "skip"      { SKIP }
  | "from"      { FROM }
  | "do"        { DO }
  | "loop"      { LOOP }
  | "until"     { UNTIL }

  (* 変数 *)
  | alpha alnum*
    { ID (Lexing.lexeme lexbuf) }

  (* スペース *)
  | space+    { token lexbuf }

  | eof { EOF }

  | _
    {
      let message = Printf.sprintf
        "unknown token %s near characters %d-%d"
        (Lexing.lexeme lexbuf)
        (Lexing.lexeme_start lexbuf)
        (Lexing.lexeme_end lexbuf)
      in
      failwith message
    }
```

図 12 字句定義ファイルの一部

$$
\begin{aligned}
&\mathcal{I}[\![\textbf{skip}]\!] = \textbf{skip} && \mathcal{I}[\![s_1\ s_2]\!] = \mathcal{I}[\![s_2]\!]\ \mathcal{I}[\![s_1]\!]\\
&\mathcal{I}[\![x\ \textbf{+=}\ e]\!] = x\ \textbf{-=}\ e && \mathcal{I}[\![x\ \textbf{-=}\ e]\!] = x\ \textbf{+=}\ e\\
&\mathcal{I}[\![x\ \textbf{\^{}=}\ e]\!] = x\ \textbf{\^{}=}\ e && \mathcal{I}[\![x\ \textbf{<=>}\ e]\!] = x\ \textbf{<=>}\ e\\
&\mathcal{I}[\![x[e_1]\ \textbf{+=}\ e_2]\!] = x[e_1]\ \textbf{-=}\ e_2 && \mathcal{I}[\![x[e_1]\ \textbf{-=}\ e_2]\!] = x[e_1]\ \textbf{+=}\ e_2\\
&\mathcal{I}[\![x[e_1]\ \textbf{\^{}=}\ e_2]\!] = x[e_1]\ \textbf{\^{}=}\ e_2 && \mathcal{I}[\![x[e_1]\ \textbf{<=>}\ e_2]\!] = x[e_1]\ \textbf{<=>}\ e_2\\
&\mathcal{I}[\![\textbf{new}\ c\ x]\!] = \textbf{delete}\ c\ x && \mathcal{I}[\![\textbf{copy}\ c\ x\ x']\!] = \textbf{uncopy}\ c\ x\ x'\\
&\mathcal{I}[\![\textbf{delete}\ c\ x]\!] = \textbf{new}\ c\ x && \mathcal{I}[\![\textbf{uncopy}\ c\ x\ x']\!] = \textbf{copy}\ c\ x\ x'\\
&\mathcal{I}[\![\textbf{call}\ q(\dots)]\!] = \textbf{uncall}\ q(\dots) && \mathcal{I}[\![\textbf{call}\ x :: q(\dots)]\!] = \textbf{uncall}\ x :: q(\dots)\\
&\mathcal{I}[\![\textbf{uncall}\ q(\dots)]\!] = \textbf{call}\ q(\dots) && \mathcal{I}[\![\textbf{uncall}\ x :: q(\dots)]\!] = \textbf{call}\ x :: q(\dots)\\
&\mathcal{I}[\![\textbf{if}\ e_1\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ e_2]\!] && = \textbf{if}\ e_2\ \textbf{then}\ \mathcal{I}[\![s_1]\!]\ \textbf{else}\ \mathcal{I}[\![s_2]\!]\ \textbf{fi}\ e_1\\
&\mathcal{I}[\![\textbf{from}\ e_1\ \textbf{do}\ s_1\ \textbf{loop}\ s_2\ \textbf{until}\ e_2]\!] && = \textbf{from}\ e_2\ \textbf{do}\ \mathcal{I}[\![s_1]\!]\ \textbf{loop}\ \mathcal{I}[\![s_2]\!]\ \textbf{until}\ e_1\\
&\mathcal{I}[\![\textbf{construct}\ c\ x\quad s\quad \textbf{destruct}\ x]\!] && = \textbf{construct}\ c\ x\quad \mathcal{I}[\![s]\!]\quad \textbf{destruct}\ x\\
&\mathcal{I}[\![\textbf{local}\ t\ x\ = e_1\quad s\quad \textbf{delocal}\ t\ x\ = e_2]\!] && = \textbf{local}\ t\ x = e_2\quad \mathcal{I}[\![s]\!]\quad \textbf{delocal}\ t\ x\ = e_1
\end{aligned}
$$

図 13 inverter の定義 ([16] から一部改変)

```
  (Syntax.id * Value.locs) list ->
  (Value.locs * Value.value) list -> Value.value
```

である. これは, 式 exp, 環境, 及びストアを受け取り値 value を返すという意味である. 環境は変数名 id とロケーション locs の組のリスト, ストアはロケーション locs と値 value の組のリストで表現する. 関数は図 4 の意味関数 $\mathcal{E}$ に従ってパターンマッチングを用いて定義する. 以下に整数, 変数, 配列及び nil の場合分けを示す.

```
let rec eval_exp exp env st =
  match exp with
  | Const(n) -> IntVal(n)
  | Var(x) -> lookup_st (lookup_envs x env) st
  | ArrayElement(id, e) ->
     let IntVal(index) = eval_exp e env st in
     let locs = lookup_envs id env in
     let LocsVec(lv) = lookup_st locs st in
     let locs2 = lookup_vec index lv in
     lookup_st locs2 st
  | Nil -> IntVal(0)
```

1 つ目は受け取った exp が整数にマッチ場合であり,value 型の整数の値 IntVal(n) を返す.2 つ目は変数にマッチした場合である. ここではまず, 関数 lookup_envs で受け取った環境 env から, マッチした変数名 x に対応するロケーションを求める. そして, 関数 lookup_st で求めたロケーションに対応する値を返す. 関数 lookup_env と lookup_st の定義を以下に示す.

```
let lookup_envs x env =
 try snd (List.find (fun (y,_) -> x = y) env)
 with Not_found ->
 failwith ("unbound variable: " ^ x)

let lookup_st x st =
 try snd (List.find (fun (y,_) -> x = y) st)
 with Not_found ->
 failwith ("unbound locations: " ^ (string_of_int x))
```

関数 lookup_envs は, 変数名と環境を受け取り, その環境の中から受け取った変数名に対応するロケーションを返す関数である. 関数 lookup_st は, ロケーションとストアを受け取り, そのストアの中から受け取ったロケーションに対応する値を返す関数である. 関数 List.find は, 第一引数に真偽を判定する関数, 第二引数にリストを取る.

## 参考文献


[1] Landauer, R.: Irreversibility and Heat Generation in the Computing Process, *IBM Journal of Research and Development*, Vol.5, No.3, pp.183–191 (1961).

[2] Wille, R. and Drechsler, R.: *Towards a Design Flow for Reversible Logic*, Springer (2010).

[3] Lutz, C.: Janus: a time-reversible language (1986). *Letter to R. Landauer.*

[4] Frank, M.P.: Reversibility for Efficient Computing, PhD Thesis (1999).

[5] Yokoyama, T., Axelsen, H.B. and Glück, R.: Towards a Reversible Functional Language, Vol.7165, Springer-Verlag, pp.14–29 (2012).

[6] Schultz, U.P.: Reversible Object-Oriented Programming with Region-Based Memory Management, *Reversible Computation* (Kari, J. and Ulidowski, I., Eds.), Cham, Springer-Verlag, pp.322–328 (2018).

[7] Haulund, T.: Design and Implementation of a Reversible Object-Oriented Programming Language, Master's thesis, University of Copenhagen (2017).

[8] Yokoyama, T. and Glück, R.: A Reversible Programming Language and Its Invertible Self-Interpreter, *Proceedings of the 2007 ACM SIG-*

*PLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New York, NY, USA, Association for Computing Machinery, p.144–153 (2007).

[9] James, R.P. and Sabry, A.: Theseus: A High Level Language for Reversible Computing (Work-in-progress report), *Conference on Reversible Computation* (2014).

[10] James, R.P. and Sabry, A.: Isomorphic Interpreters from Logically Reversible Abstract Machines, Vol.7581, Springer-Verlag, pp.57–71 (2013).

[11] Ulidowski, I., Phillips, I. and Yuen, S.: Reversing Event Structures, *New Generation Computing*, Vol.36, No.3, pp.281–306 (2018).

[12] Wille, R., Offermann, S. and Drechsler, R.: SyReC: A programming language for synthesis of reversible circuits, pp.184–189 (2010).

[13] Schultz, U.P. and Axelsen, H.B.: Elements of a Reversible Object-Oriented Language, *Reversible Computation* (Devitt, S. and Lanese, I., Eds.), Cham, Springer International Publishing, pp.153–159 (2016).

[14] Mezzina, C.A., Schlatte, R., Glück, R., Haulund, T., Hoey, J., Holm Cservenka, M., Lanese, I., Mogensen, T.Æ., Siljak, H., Schultz, U.P. and Ulidowski, I.: *Software and Reversible Systems: A Survey of Recent Activities*, pp.41–59, Springer International Publishing (2020).

[15] Yokoyama, T., Axelsen, H.B. and Glück, R.: Fundamentals of reversible flowchart languages, *Theoretical Computer Science*, Vol.611, pp.87–115 (2016).

[16] Cservenka, M.H.: Design and Implementation of Dynamic Memory Management in a Reversible Object-Oriented Programming Language, Master's thesis, University of Copenhagen (2018).