アンダースタンディング・コンピュテーション　第 2 章
プログラム意味論

岡本 吉央
okamotoy@uec.ac.jp

電気通信大学

2019 年 5 月 31 日

最終更新：2019 年 5 月 31 日　17:41

スケジュール

2. プログラム意味論 (5 月)
3. 有限オートマトン
4. プッシュダウン・オートマトン
5. チューリング機械
6. ラムダ計算
7. 万能性
8. 決定可能性
9. 抽象解釈／静的意味論

目次

プログラミング言語の 2 つの側面

構文論 (syntax)

プログラムがどのように見えるかを記述

構文論 (文法) は第 4 章で扱う

意味論 (semantics)

プログラムが何を意味するかを記述

自然言語にも，構文論と意味論はある

代表的な 3 種類のプログラム意味論

- 操作的意味論 (operational semantics)
  - スモールステップ意味論
  - ビッグステップ意味論
- 表示的意味論 (denotational semantics)
- 公理的意味論 (axiomatic semantics)

注：他にも意味論はある (ありえる)

ここからの話

時間の都合上，スモールステップ意味論に限定

- 教科書では，ビッグステップ意味論と表示的意味論も扱う

形式意味論の応用

- プログラミング言語の意味にあいまいさのない仕様を与えること
- 言語の特性，プログラムの特性を証明すること
- 1 つの言語におけるプログラム間の等価性を証明すること
- プログラムを安全に変換する方法を見つけること
- など

目次

次の機能を持つプログラミング言語を考える

$a + b$	2 つの数値 a, b を足す
$a * b$	2 つの数値 a, b を掛ける
$a < b$	2 つの数値 a, b の大小を比較する
$x = a$	変数 x に a を代入する
$A; B$	式 A, B を続けて実行する
`if` (A) { B } `else` { C }	条件分岐を行う
`while` (A) B	反復を行う

抽象機械 (abstract machine)

プログラムを実行する抽象的な装置 (理想的なコンピュータ)

スモールステップ意味論では

- 構文を小さなステップで繰り返し簡約
- それによって，プログラムを評価

ステップを経るたびに，プログラムはそれが意味するものへ近づく

SIMPLE において，足し算「$a + b$」 は次のように簡約される

1. a が簡約できるとき，a を簡約する
2. a が簡約できず，b が簡約できるとき，b を簡約する
3. a も b も簡約できないとき，$a + b$ に簡約する

例：

$$
\begin{aligned}
3 + 4 &\rightsquigarrow 7 \\
2 + 8 &\rightsquigarrow 10 \\
(3 + 4) + (2 + 8) &\rightsquigarrow 7 + (2 + 8) \\
&\rightsquigarrow 7 + 10 \\
&\rightsquigarrow 17
\end{aligned}
$$

スモールステップ意味論：SIMPLE における足し算 (2)

Ruby で実行した様子

```
>> e = Add.new(Add.new(Number.new(3),
                       Number.new(4)),
                Add.new(Number.new(2),
                       Number.new(8)))
=> <<3 + 4 + 2 + 8>>
>> Machine.new(e).run
3 + 4 + 2 + 8
7 + 2 + 8
7 + 10
17
=> nil
```

形式的な書き方

e_1 が e_1' に簡約できるとき，$e_1 + e_2$ は $e_1' + e_2$ に簡約される

$$\frac{\langle e_1, \sigma \rangle \rightsquigarrow_e e_1'}{\langle e_1 + e_2, \sigma \rangle \rightsquigarrow_e e_1' + e_2}$$

v_1 は簡約できないとする
e_2 が e_2' に簡約できるとき，$v_1 + e_2$ は $v_1 + e_2'$ に簡約される

$$\frac{\langle e_2, \sigma \rangle \rightsquigarrow_e e_2'}{\langle v_1 + e_2, \sigma \rangle \rightsquigarrow_e v_1 + e_2'}$$

n_1, n_2 は数であり，$n = n_1 + n_2$ を満たすとする
$n_1 + n_2$ は n に簡約される

$$\frac{}{\langle n_1 + n_2, \sigma \rangle \rightsquigarrow_e n} \text{ if } n = n_1 + n_2$$

SIMPLE における掛け算と大小比較 (1)

掛け算と大小比較も，足し算と同様に簡約を行うことで評価する

$$
\begin{aligned}
(3 * (2 + 1)) < (2 * 4) &\rightsquigarrow (3 * 3) < (2 * 4) \\
&\rightsquigarrow 9 < (2 * 4) \\
&\rightsquigarrow 9 < 8 \\
&\rightsquigarrow \mathsf{false}
\end{aligned}
$$

SIMPLE における掛け算と大小比較 (2)

Ruby で実行した様子

```
>> e = LessThan.new(Multiply.new(Number.new(3),
                                 Add.new(Number.new(2),
                                         Number.new(1))),
                    Multiply.new(Number.new(2),
                                 Number.new(4)))
=> <<3 * 2 + 1 < 2 * 4>>
>> Machine.new(e).run
3 * 2 + 1 < 2 * 4
3 * 3 < 2 * 4
9 < 2 * 4
9 < 8
false
=> nil
```

SIMPLE における変数 (1)

変数を使うために「環境 (environment)」を用いる

- 環境は，変数とその値を表として持っている (つまり，写像)

例： $\mathtt{x} : 2,\ \mathtt{y} : \mathsf{false}$

SIMPLE において，変数は次のように簡約される

1. 変数 x の値が環境で与えられているとき，x をその値に簡約する

例：$\mathtt{x} : 2$，$\mathtt{y} : 4$ のとき

$$\begin{aligned} \mathtt{x} * \mathtt{y} &\rightsquigarrow 2 * \mathtt{y} \\ &\rightsquigarrow 2 * 4 \\ &\rightsquigarrow 8 \end{aligned}$$

SIMPLE における変数 (2)

Ruby で実行した例

```
>> exp = Multiply.new(Variable.new(:x), Variable.new(:y))
=> <<x * y>>
>> env = { x: Number.new(2), y: Number.new(4) }
=> {:x=><<2>>, :y=><<4>>}
>> Machine.new(exp, env).run
x * y
2 * y
2 * 4
8
=> nil
```

式と文の違い

- 式 (expression)：評価されると別の式を生成する　　$(1 + 2 \rightsquigarrow 3)$
- 文 (statement)：評価されると新たな環境を生成する

SIMPLE における文

- `do-nothing` 文：何もしない文 (最も単純な文)
- 代入文
- シーケンス文
- 条件文 (`if-else` 文)
- 反復文 (`while` 文)

スモールステップ意味論で，`do-nothing` 文は，簡約できない唯一の文

SIMPLE において，代入文 $\mathtt{x} = \mathtt{a}$ は次のように簡約される

1. $\mathtt{a}$ が簡約できるとき，$\mathtt{a}$ を簡約する
2. $\mathtt{a}$ が簡約できないとき，$\mathtt{x} : \mathtt{a}$ となるように環境を変え，
   代入文 $\mathtt{x} = \mathtt{a}$ を do-nothing 文に簡約する

例：

$$\begin{aligned} \mathtt{x} = 2 + 3,\ \{\ \} &\rightsquigarrow \mathtt{x} = 5,\ \{\ \} \\ &\rightsquigarrow \text{do-nothing},\ \{\mathtt{x} : 5\} \end{aligned}$$

$$\begin{aligned} \mathtt{x} = \mathtt{x} * 3,\ \{\mathtt{x} : 4\} &\rightsquigarrow \mathtt{x} = 4 * 3,\ \{\mathtt{x} : 4\} \\ &\rightsquigarrow \mathtt{x} = 12,\ \{\mathtt{x} : 4\} \\ &\rightsquigarrow \text{do-nothing},\ \{\mathtt{x} : 12\} \end{aligned}$$

SIMPLE における文：代入文 (2)

Ruby で実行した例　　(このように実行させるために，コードは修正した)

```
>> stmt = Assign.new(Variable.new(:x),
                     Multiply.new(Variable.new(:x),
                                  Number.new(3)))
=> <<x = x * 3>>
>> env = { x: Number.new(4) }
=> {:x=><<4>>}
>> Machine.new(stmt, env).run
x = x * 3, {:x=><<4>>}
x = 4 * 3, {:x=><<4>>}
x = 12, {:x=><<4>>}
do-nothing, {:x=><<12>>}
=> nil
```

SIMPLE における文：シーケンス文 (1)

シーケンス文「A; B」は A の実行に続いて B を実行する

SIMPLE において，シーケンス文 A; B は次のように簡約される

1. A が簡約できるとき，A を簡約する
2. A が簡約できないとき (つまり，do-nothing であるとき)
   A; B を B に簡約する

例：

$$
\begin{aligned}
\mathtt{x} = \mathtt{x} + 3; \mathtt{x} = \mathtt{x} + \mathtt{x},\ \{\mathtt{x} : 2\} &\rightsquigarrow \mathtt{x} = 2 + 3; \mathtt{x} = \mathtt{x} + \mathtt{x},\ \{\mathtt{x} : 2\} \\
&\rightsquigarrow \mathtt{x} = 5; \mathtt{x} = \mathtt{x} + \mathtt{x},\ \{\mathtt{x} : 2\} \\
&\rightsquigarrow \mathtt{do\text{-}nothing}; \mathtt{x} = \mathtt{x} + \mathtt{x},\ \{\mathtt{x} : 5\} \\
&\rightsquigarrow \mathtt{x} = \mathtt{x} + \mathtt{x},\ \{\mathtt{x} : 5\} \\
&\rightsquigarrow \mathtt{x} = 5 + \mathtt{x},\ \{\mathtt{x} : 5\} \\
&\rightsquigarrow \mathtt{x} = 5 + 5,\ \{\mathtt{x} : 5\} \\
&\rightsquigarrow \mathtt{x} = 10,\ \{\mathtt{x} : 5\} \\
&\rightsquigarrow \mathtt{do\text{-}nothing},\ \{\mathtt{x} : 10\}
\end{aligned}
$$

SIMPLE における文：シーケンス文 (2)

```
>> stmt = Sequence.new(Assign.new(Variable.new(:x),
                                  Add.new(Variable.new(:x),
                                          Number.new(3))),
                       Assign.new(Variable.new(:x),
                                  Add.new(Variable.new(:x),
                                          Variable.new(:x))))
=> <<x = x + 3; x = x + x>>
>> env = { x: Number.new(2) }
=> {:x=><<2>>}
>> Machine.new(stmt, env).run
x = x + 3; x = x + x, {:x=><<2>>}
x = 2 + 3; x = x + x, {:x=><<2>>}
x = 5; x = x + x, {:x=><<2>>}
do-nothing; x = x + x, {:x=><<5>>}
x = x + x, {:x=><<5>>}
x = 5 + x, {:x=><<5>>}
x = 5 + 5, {:x=><<5>>}
x = 10, {:x=><<5>>}
do-nothing, {:x=><<10>>}
=> nil
```

条件文 `if (A) B else C` の簡約

1. `A` が簡約できるとき，`A` を簡約する
2. `A` が true であるとき，この文を `B` に簡約する
3. `A` が false であるとき，この文を `C` に簡約する

$$
\begin{aligned}
\texttt{if}\ (2+3<4)\ 5\ \texttt{else}\ 7 &\rightsquigarrow \texttt{if}\ (5<4)\ 5\ \texttt{else}\ 7 \\
&\rightsquigarrow \texttt{if}\ (\mathsf{false})\ 5\ \texttt{else}\ 7 \\
&\rightsquigarrow 7
\end{aligned}
$$

SIMPLE における文：条件文 (2)

```
>> stmt = If.new(LessThan.new(Add.new(Number.new(2),
                                      Number.new(3)),
                           Number.new(4)),
                 Number.new(5), Number.new(7))
=> <<if (2 + 3 < 4) { 5 } else { 7 }>>
>> Machine.new(stmt).run
if (2 + 3 < 4) { 5 } else { 7 },
if (5 < 4) { 5 } else { 7 },
if (false) { 5 } else { 7 },
7,
=> nil
```

反復文 while (A) B の簡約

if (A) { B; while (A) B } else do-nothing に簡約

$$
\begin{aligned}
& \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2,\ \{\mathtt{x} : 2\} \\
\rightsquigarrow\ & \texttt{if}\ (\mathtt{x} < 3)\ \{\mathtt{x} = \mathtt{x} + 2; \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2\ \}\ \texttt{else do-nothing}, \\
& \{\mathtt{x} : 2\} \\
\rightsquigarrow\ & \texttt{if}\ (2 < 3)\ \{\mathtt{x} = \mathtt{x} + 2; \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2\ \}\ \texttt{else do-nothing}, \\
& \{\mathtt{x} : 2\} \\
\rightsquigarrow\ & \texttt{if}\ (\texttt{true})\ \{\mathtt{x} = \mathtt{x} + 2; \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2\ \}\ \texttt{else do-nothing}, \\
& \{\mathtt{x} : 2\} \\
\rightsquigarrow\ & \mathtt{x} = \mathtt{x} + 2; \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2, \{\mathtt{x} : 2\} \\
\rightsquigarrow\ & \mathtt{x} = 2 + 2; \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2, \{\mathtt{x} : 2\} \\
\rightsquigarrow\ & \mathtt{x} = 4; \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2, \{\mathtt{x} : 2\}
\end{aligned}
$$

反復文 while (A) B の簡約

if (A) { B; while (A) B } else do-nothing に簡約

続き

$$
\begin{aligned}
&\rightsquigarrow \quad \texttt{do-nothing}; \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2, \{\mathtt{x} : 4\} \\
&\rightsquigarrow \quad \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2, \{\mathtt{x} : 4\} \\
&\rightsquigarrow \quad \texttt{if}\ (\mathtt{x} < 3)\ \{\mathtt{x} = \mathtt{x} + 2; \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2\}\ \texttt{else do-nothing}, \\
&\qquad \{\mathtt{x} : 4\} \\
&\rightsquigarrow \quad \texttt{if}\ (4 < 3)\ \{\mathtt{x} = \mathtt{x} + 2; \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2\}\ \texttt{else do-nothing}, \\
&\qquad \{\mathtt{x} : 4\} \\
&\rightsquigarrow \quad \texttt{if}\ (\mathsf{false})\ \{\mathtt{x} = \mathtt{x} + 2; \texttt{while}\ (\mathtt{x} < 3)\ \mathtt{x} = \mathtt{x} + 2\}\ \texttt{else do-nothing}, \\
&\qquad \{\mathtt{x} : 4\} \\
&\rightsquigarrow \quad \texttt{do-nothing}, \{\mathtt{x} : 4\}
\end{aligned}
$$

SIMPLE における文：反復文 (2)

```
>> stmt = While.new(LessThan.new(Variable.new(:x), Number.new(3)),
                    Assign.new(Variable.new(:x),
                               Add.new(Variable.new(:x), Number.new(2))))
=> <<while (x < 3) { x = x + 2 }>>
>> env = { x: Number.new(2) }
=> {:x=><<2>>}
>> Machine.new(stmt, env).run
while (x < 3) { x = x + 2 }, {:x=><<2>>}
if (x < 3) { x = x + 2; while (x < 3) { x = x + 2 } } else { do-nothing },
if (2 < 3) { x = x + 2; while (x < 3) { x = x + 2 } } else { do-nothing },
if (true) { x = x + 2; while (x < 3) { x = x + 2 } } else { do-nothing }, {
x = x + 2; while (x < 3) { x = x + 2 }, {:x=><<2>>}
x = 2 + 2; while (x < 3) { x = x + 2 }, {:x=><<2>>}
x = 4; while (x < 3) { x = x + 2 }, {:x=><<2>>}
do-nothing; while (x < 3) { x = x + 2 }, {:x=><<4>>}
while (x < 3) { x = x + 2 }, {:x=><<4>>}
if (x < 3) { x = x + 2; while (x < 3) { x = x + 2 } } else { do-nothing },
if (4 < 3) { x = x + 2; while (x < 3) { x = x + 2 } } else { do-nothing },
if (false) { x = x + 2; while (x < 3) { x = x + 2 } } else { do-nothing },
do-nothing, {:x=><<4>>}
=> nil
```

目次

個人プロジェクト案の例

- 加算，乗算，大小比較の適用順序をカッコを使って正しく表示するために，`to_s`, `inspect` の実装を修正する
- 減算，除算,「$<$」以外の不等号，論理演算 (AND, OR, NOT) の機能を SIMPLE に追加するために，
  それらの操作的意味論を考えて，実装する
- `treetop` を使って，パーサの実装を完成させる
  - そのために，まず SIMPLE の文法を定義する
- SIMPLE から Ruby 以外のプログラミング言語へのコンパイラを作る