この本の目的

- semantics (意味論) で使われる考え方や method (方法) の記述すること
- これら (↑) の考え方や method をわかりやすく解説すること (アプリケーション作ってみるよ)
- 様々な method の関係性を研究すること

Specify : 詳しく記述する　　illustrate : わかりやすく説明する
application : アプリケーション, 適用

ちょっと注意事項

- semantics は詳しく、明確に、厳格に記述する必要が有る
- 記号は大体、普段数学で使うのと同じものです

※ syntax と semantics を明確に区別して読み進めましょう (次のスライドからあるよ)

Represent : 表す　implementation : 実装, (たまに、実行)　analysis : 解析

Syntax と Semantics

Syntax : grammatical structure(構文,文法)

parser に書く内容のこと

Semantics : the meaning (意味)

意味とはなんぞや？→次のページ

Syntax analysis : 構文解析

Semantics の3つの意味

- Operational semantics : どう計算するか
- Denotational semantics :

  どんな結果が出るか
- Axiomatic semantics :

  実行するのに必要な条件は何か

Execute : 実行する　Construct : 構文 (syntax に従って書かれたもの)

Effect : 影響, エフェクト (実行してくと、それぞれお互いに影響し合ってく感じ)

Statement : 命令文　Sequence : 並び

例

z := x; x := y; y := z;

(x と y の中身を入れ替えるプログラムです)

Assign : 与える, 譲渡する, 代入する

Operational semantics

z:=x; x:=y; y:=z; をどうやって計算するか

→ 個々の命令文を、左から右に一つずつ実行していく

(:= は代入を表す)

A variable followed by ':=' ∽ : (大抵) 命令文の先頭の文字のこと　z:=x; なら z

長くてよくわかんないんだよ。。。

Structual operational semantics

- プログラムをどうやって実行するかを抽象的に表す。(small-step semantics)
- レジスタとかアーキテクチャとかを考えてないあたりが抽象的
- これを図で表したものを derivation sequence という (P3 上の表)

Abstraction : 抽象　abstract : 抽象的　derivation : 導出

Derivation sequence : 訳せない。。。

Natural semantics

- プログラムによって、state (環境) がどう変化していくかを記述している、らしい
- どうして state がそんな変化をするかはあまりきにしていない (big-step semantic)
- どのように state が変化していくかを表したものを derivation tree (導出木) という (P.3 下の図)

abbreviation : 省略

Denotational semantics

- プログラム全体を、個々の命令文の関数合成と考える (P.4 真ん中の式)
- 例えば、z := x; は、z の中身を変化させる関数だと考える
- とっても数学です、プログラムはその対象です
- どうやって計算するかは特に考えてない、結果が全て (Chapter 5 で意味がわかるのでせう)

Reason : 推論する　functional composition : 関数合成　apply : 適用する

Denotational & Operational

- プログラムの推論 (?) をするには Denotational semantics が有効
- つくった言語を implement (実装) するには Operational semantics が有効
- Operational と Denotational が等価であるととても嬉しい → Chapter 4.3　楽しみだなー

Equivalent : 同等な　be adapt to ~ : 構成される

Axiomatic semantics

- プログラムの部分的正当性を Precondition (前提条件) と Postcondition (後置条件) を満たすかどうかで考える
- { precondition } construct { postcondition }

  で表される
- ただし、プログラムがちゃんと終了するかを示すものではない(だから「部分的」なのだ)

Partial : この場合は、終了するかは置いといて、(正しいかどうか)という感じ

(ついでに) total : 必ず終了する, 定義域全てに対して値を返すような

Pre とか Post とかって何？

独断と偏見で説明します。

たとえば2次方程式の解を、解の公式を使って求めるプログラムをつくったとき、プログラムがちゃんと動くには、a, b, c に実数が入っていて、

$$a \neq 0,\ b^2 - 4ac > 0$$

の2式が成り立ってる必要があります。これが precondition です。

Partial correctness property : (終了するかは考えない、プログラムの) 部分的正当性

Pre とか Post とかって何？

で、プログラムが終了(?) したときには、変数 x に何らかの実数 (求めた解) が入ってないと困りますね。これが Postcondition です。

ちなみに、何回か登場する assertion という言葉(日本語だと「表明」と言う) は、プログラムの中で、プログラマが「ここの時にはこの変数の値はこうなってるか調べてくれ～！」……というのをプログラム中に書いたりすること？とかまたはそこに書かれたことを言うようです。

Precondition とか Postcondition は (というか axiomatic semantics は) 周囲は考えず、ただ与えられた construct が正しいかどうかを判断するためのものなので、assertion とは大分意味合いが違います。

話は戻って Axiomatic

- プログラムの部分的正当性の証明を表したものを証明木という (P.6 上の図)
- ただし、{} の中身が同じだからといって、プログラムが同じ動きをするとは限らない (P.6 真ん中)
- Axiomatic semantics の利点は、プログラムの性質を簡単に証明できることにある

Specification : 仕様(書)

今後の展望

- 3つの semantics を While のために展開する
- 3つの semantics の弱点や利点を While を拡張することによって図解する
- While (の実装?) のために、3つの〜の関連性を証明する
- 〜のメリットを図解するために、意味記述アプリケーションの例を示す

3つの semantics はそれぞれに向き不向きがある。成果を競っているのではない。

While 言語

この本で使うプログラミング言語 (定義はP.7)

BNF : 前期でおなじみ、みんなよく使ってる、(文脈自由文法の) 構文の記述法

Syntactic category : 統語範疇 cf.戸次先生　例えば、前期での「項」とか「値継続」とか

Meta-variables : メタ変数　プログラムで使う変数ではなく、syntactic category を代表する変数のこと　数字なら n, (まだ出てこないけど)文字列は str とかはおなじみ

Range over : 動き回る, (表す, と考えてよいと思う)

Numeral : (syntactic category としての) 数

Arithmetic expressions : 計算すると (普通の) 数字になる表現 (プログラム)

Boolean expressions : 計算すると真偽値になる表現 (プログラム)

Statement : 命令文

While 言語

※この定義は抽象的なものなので、よく理解するには、実際に構文解析木 (syntax tree) を描いてみると良い。

※最左、最右の区別はない (文脈自由文法だから)

※ていうか、書かれているものが、どの syntactic category に属するかが問題なのである

※ () は必要に応じて多用してよい

※ でも面倒だから + - * and or not なんかは普段数学で使ってる結合規則を使う

Primed : ダッシュをつける　subscripted : 下付き文字をつける

string : (普通の意味で) 文字列　digit : (文字としての) 数字　Letter : 文字 (アルファベット)

Basis element : たとえば b の expression のなかで b を含まないもの

Composite element :たとえば b の expression のなかで b を含むもの

While 言語

- While 言語 (で書かれたプログラム) の意味は、Semantic function (意味関数, 後述) によって与えられる
- Semantic function のためにここまでいろいろ…以下略

Entity : もの (syntax 側のものを指しているらしい)　assume : 仮定する

Syntactic entity : (syntax に従って書かれたプログラムの)　argument : 引数

Assign : 譲渡する、代入する

Semantics function : 意味関数

　特にこの本では、P.7 の syntax で定義されたプログラム (syntactic entity) を受け取ってきて、その意味 (大体は、プログラムを計算した値) を返す関数のこと (あとで詳しく)

いよいよ Semantics

注意！！
ここからしばらく、numerals は2進法になります

他のとこでは10進法です。でも簡単にするために2進法にします。どう簡単なのかは、先に進めばわかります。

Binary system : 2進法

しばらく読むだけです

P.9 の式を参照

※ syntactic entity と semantics (meaning) の違いをはっきり読み分けてください (単語も違うものがあてられています)

※意味関数の引数は二重角かっこでくくります

Number : (普通の意味で、10進法の) 数

Total function : 全ての定義域に対して値を返し、かつ終了する関数

clause : 節　bracket : 角かっこ (ただ単にかっこ, くくるみたいな意味でも使ってるもよう)

Corresponding : (厳密に1対1で) 対応する　application : 適用

Semantics clause : すみません、訳せません。P.10 に出てくるような定義の一行のことです

Equation : 方程式

$\mathcal{N}$を合成定義で書く

P.10 上半分くらい

・P.9 の numeral の定義に沿っているでしょう？

※ 二重角かっこの中身が syntax なもの
(業界では syntax は二重角かっこでくくる習慣があります)

※ ↑以外が semantics (meaning) 書いてある数字、記号は普通に演算できる

※一応、細字が syntactic entity で、太字が semantics らしい (よくわかんないなあ)

Compositional definition : 合成定義　Obtain : 得る
Subconstruct : construct の一部, 部分式

$\mathcal{N}$が well-defined である証明

P.10 真ん中～P.11真ん中

Well-defined : ちゃんと定義されている cf. 計算基礎論

Induction : 数学的帰納法

Base case : 数学的帰納法の最初のところ

Induction step : n’ で命題が成立するならば、n のとき～ のところ

Hypothesis : 仮説

Hold : 有効である, 成り立つ

Omit : 省く ex. The case ～ is similar and we omit the details.

Compositional definition

- syntactic category が basis element と composite element によって、(明確に)定義されているとき
- syntactic category の全ての element に対して、semantic clause を定義する。Composite element に対しては、その構造に見合った定義をする。

Immediate constituents of the element : Composite element の中に出てくる、自身の syntactic category と同じメタ変数 ……自分で書いてて何がなんだか

あれだ、b で言えば、b1 ^ b2 の b1 と b2 がそれだ

Compositional Definition

- composite element の定義は、部分式に自分と同じ syntactic category の式を含んでおり
- さらに**その部分式が自分よりも小さくなっている**

→ composite element を定義に従って分解していくと、**必ず** basis element が出てきて、**分解が止まる！**

→帰納法が使える（次スライド）

Immediate constituents of the element : Composite element の中に出てくる、自身の syntactic category と同じメタ変数 ……自分で書いてて何がなんだか

あれだ、b で言えば、b1 ^ b2 の b1 と b2 がそれだ

Structual Induction

- compositional definition の証明で有効
- まず、basis element で命題が成り立つことを証明する
- 次に、composite element で、immediate constituents of the elements では命題が成り立つという仮説 (induction hypothesis) を立て、証明をする

Structual Induction : 構造帰納法

※ このあとの numeral は全て10進法です。

Semantic function

プログラムの意味（値）を返す関数

→　変数の中身によって返ってくるものが変わる

Semantic function : 意味関数

State

- 変数をうけとると、その中身を返す関数
- “環境” とは違う (state は変数の中身が変わる)
- 状況 s に変数 x の中身をきくときは、

s x

で表す

- 他の表し方もあるにはあるが、めんどい

State : 状況, 状態

意味関数 $\mathcal{A}$

- expression と state によって、値が決まる

→ expression と state を受けとって、値を返す関数 ($\mathcal{A}$) を定義する

- p.9 の型の通り、$\mathcal{A}$ に [[AExp]] を渡すと、state を受け取ると値を返す関数が返ってきて、それに stateを渡すと、値が返ってくるような関数である
- 表記法も ↑ に従っているようだ

Functionality : 機能性, 型

意味関数 $\mathcal{A}$

- 具体的な (合成) 定義は p.10 上の表
- Compositional Definition に従って、＝の右側の $\mathcal{A}$ の引数は、左側で引数となっている syntactic entity の部分式 (immediate ⌣) となっている
- さらに、＝右側に書かれているものは、演算子なども全て semantic であることに着目されたし (二重角かっこ内は除く)

Sum : 和　product : 積　difference : 差