コンパイラ理論 2 言語理論

櫻井彰人

言語理論:

- 言語を定義する方法(いくつかある):
  - 文法(生成規則)
  - オートマトン
  - 既知の言語間の演算
- これらの間には対応関係がある
- コンパイラの設計に使用

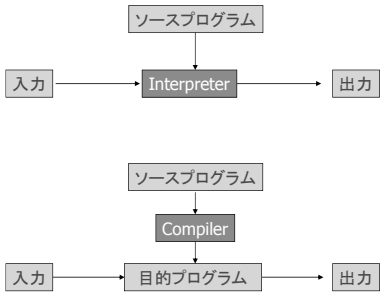
言語の定義方法

- どうやって定義するか
- 定義方法は使いやすくあるべし, i.e.:
  - 定義は有限の長さ
  - 与えられた文字列がその言語に属するか否かを調べるアルゴリズムが存在する必要がある
  - さらに、その意味が一意に抽出できるアルゴリズムが必要
- 広く使われている方法は、(生成規則を用いた)文法記述
- 自然言語の記述には、更なる工夫が必要

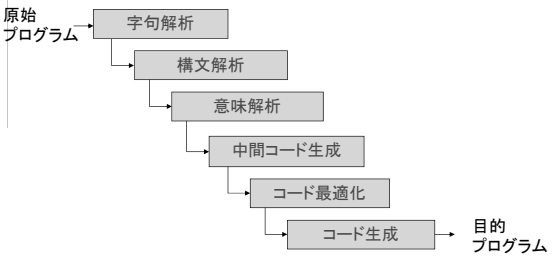
プログラムの構文と意味

- 構文 (syntax)
  - プログラムを書くのに用いる記号(達)
- 意味 (semantics)
  - プログラムが実行されるときに発生する行動
- プログラミング言語の実装
  - 構文 → 意味
  - プログラムの構文を機械命令列に変換する。この機械命令列を実行すると、行動の正しい系列が出現するような変換である

Interpreter と Compiler



コンパイルの典型的な流れ



構文を簡単に

- 文法
  e ::= n | e+e | e–e
  n ::= d | nd
  d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- 式
  e → e–e → e–e+e → n–n+n → nd–d+d → dd–d+d
  → … → 27 – 4 + 3

文法は言語を定める
式は、生成規則を順に適用することによって導出される

ご存じですね？

言語

記号を有限個並べて得られる系列

記号列

制約：文法という

言語

自然言語研究がきっかけ

文法の定義方法

- 「文」は「主部」と「述部」からなる
- 「主部」は「名詞句」と「が」からなる
- 「名詞句」は「名詞」か「修飾句」を一個以上並べたものに「名詞」をつけたもの

- <文> = <主部> <述部>
- <主部> = <名詞句> が

種類が異なることに注意

- <名詞句> = <名詞> | <修飾句並び> <名詞>
- <修飾句並び> = <修飾句> | <修飾句並び> <修飾句>

形式的には:

生成規則 production rules :

- 終端記号 terminal symbols, またはアルファベット alphabet
- 非終端記号 nonterminal symbols
- 文法を記述するための記号
- S → NP VP. NP → N | A NP. …

文法の書き方 (生成方向)

$A \Rightarrow X_1\ X_2 \ldots X_m$

書き換え規則
生成規則
BNF (Backus Naur form,
Backus normal form)

文法の書き方 (解析方向)

$A \Leftarrow X_1\ X_2 \ldots X_m$

解析方向：使用することはまれ

同一記号の書換え

$A \Rightarrow X_1\ X_2\ \ldots.\ X_m$

$A \Rightarrow Z_1\ Z_2\ \ldots.\ Z_m$

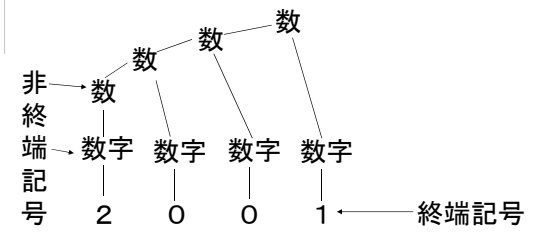
$A \Rightarrow X_1\ X_2\ \ldots.\ X_m \mid Z_1\ Z_2\ \ldots.\ Z_m$

と記述

文法例1

数 ⇒ 数　数字　｜　数字

数字 ⇒ 0｜1｜2｜3｜4｜5｜6｜7｜8｜9



文法例2

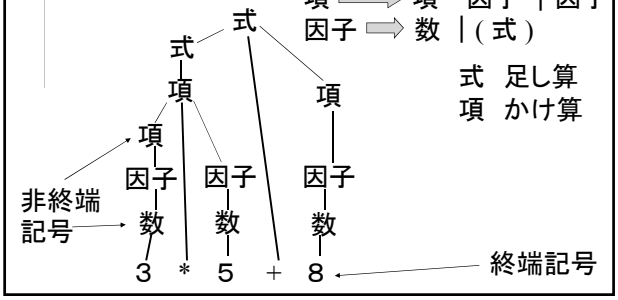
式 ⇒ 式 + 項　｜　項

項 ⇒ 項 * 因子 ｜因子

因子 ⇒ 数 ｜(式)

式　足し算

項　かけ算



文法例3

s　文
n　名詞
v　動詞
p　助詞
pp　後置詞句
vp　動詞句

s ⇒ pp vp

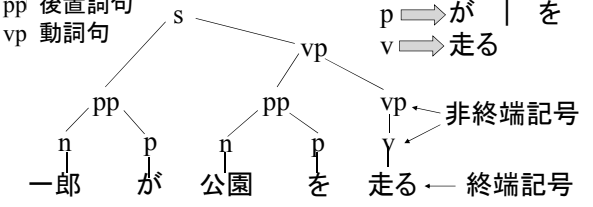
vp ⇒ pp vp｜v

pp ⇒ n p

n ⇒ 一郎｜公園

p ⇒ が　｜　を

v ⇒ 走る



チョムスキー階層

3型　正規文法　　受理：有限状態オートマトン

$A \Rightarrow a$　　$A \Rightarrow a\ B$

2型　文脈自由文法　　受理：プッシュダウン・オートマトン

$A \Rightarrow X_1\ X_2\ \ldots X_m$

1型　文脈依存文法　　受理：線型有界オートマトン

$Z_1 Z_2 \ldots Z_n \Rightarrow X_1\ X_2\ \ldots X_m$　　$n \leqq m$

0型　句構造文法　　受理：チューリング機械

右辺・左辺とも任意

a　終端記号

A,B　非終端記号

X,Z　どちらか

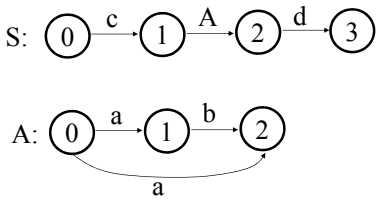
有限状態オートマトン

S ⇒ cAd

A ⇒ ab ｜a

S,A 非終端記号

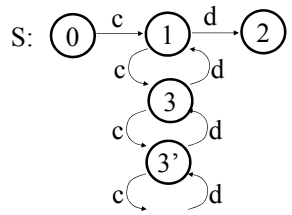
a,b,c,d 終端記号



決定性プッシュダウンオートマトン

S ⇒ cSd | cd

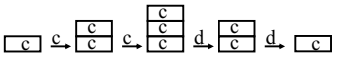
S,A 非終端記号
c,d 終端記号



後で、d との対応させるために c をとっておく

⇓

（一般には）スタック



正規文法で記述できない言語の例

S ⇒ (S) S ｜ε　文脈自由文法では記述可

とか

S ⇒ (S) ｜ε　文脈自由文法では記述可

((( )))
((( )) (((((( ))))) ()))

文脈自由文法で記述できない言語の例

L = { $a^n b^m c^n d^m$ | n,m ≧1}　　u^k uのk回並び

aa bbbb cc dddd

L={ wcw | wは(a|b)* }

aabbcaabb

aacaa

構文解析の手法

- 下向き　vs. 上向き
  - 文法項目をまとめてより上位の項目に
    - あらゆる纏め方を考える
  - 実際に生成して同じものができるか？
- 深さ優先　vs. 広さ優先
  - 候補生成の順番：縦方向、横方向
- 最左　vs. 最右
  - 左端（初め）からか、右端からか
    - 左から2番目、ということも考えられるが

下向き　解析例1

文法

S ⇒ cAd

A ⇒ ab ｜a

S,A 非終端記号
a,b,c,d 終端記号

入力 cad

S ⇒ cAd ⇒cabd　失敗

⇒cad　成功　バックトラック

解析例2

下向き＋縦＋最左

入力

一郎が公園を走る

s ⇒ pp vp
vp ⇒ pp vp ｜ v
pp ⇒ n p
n ⇒ 一郎 ｜ 公園
p ⇒ が ｜ を
v ⇒ 走る

s⇒ pp vp
⇒n p vp
⇒一郎 p vp
⇒一郎 が vp
⇒一郎 が pp vp
⇒一郎 が n p vp
⇒一郎 が 一郎 p vp
⇒一郎 が 公園 p vp
⇒一郎 が 公園 が vp
⇒一郎 が 公園 を vp
.....
⇒一郎 が 公園 を 走る

上向き型

S ⇨ aABe
A ⇨ Abc ｜b
B ⇨ d

最右＋深さ優先

入力　abbcde ➡ aAbcde
➡ aAde
➡ aABe
➡ S

解析例2

深さ優先＋最右

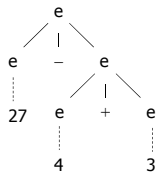
s ⇨ pp vp
vp ⇨ pp vp ｜v
pp ⇨ n p
n ⇨ 一郎｜公園
p ⇨ が ｜ を
v ⇨ 走る

一郎が公園を走る	pp pp 走る
nが公園を走る	pp pp v
n p 公園を走る	pp pp vp
pp 公園を走る	pp s 失敗
pp n を走る	pp vp 成功
pp n p 走る	s

構文解析木

- 導出過程を表現した木

e → e–e → e–e+e → n–n+n → nd–d+d → dd–d+d
→ … → 27 – 4 + 3



木は、括弧付けされた式を表すと考えられる

構文解析

- 式が与えられたとき、構文木を作成すること
- 曖昧性があることもある
  - 式　27 – 4 + 3　に二通りの構文解析方法がありうる
  - 問題となるのは：　27 – (4 + 3) ≠ (27 – 4) + 3
- 曖昧性を解消する方法
  - 手順で
    - 纏める順序は、* が + より先
    - 3*4 + 2 は (3*4) + 2 と解析
  - 結合性(associativity)
    - 等しい優先順序の演算は、左(または右)から括弧でくくる
    - 3 – 4 + 5 は (3 – 4) + 5 と解析

詳細はコンパイラの本等を

文法の性質

- 文法は、多くの場合、生成規則で記述される
- 文法が異なっても、生成する言語は同じときがある。同一の言語を生成する文法は等価 *equivalent* であるという
- ある文法では、生成規則の適用順序や適用規則が異なるにも関わらず、同じ文が生成されることがある。曖昧 ambiguous な文、曖昧 ambiguous な文法という

Chomsky 階層 再登場

- Chomsky 階層（文法規則のパターンに対する制限の強弱 type-0: 弱い～type-3:強い）:
  - type-0
  - type-1: 文脈依存 context dependent grammars
  - type-3: 正規 regular grammars
- 文法規則のパターンに対する制限が弱いほど、文法的に正しい文と正しくない文の違いが微妙になる
  - type-2: 文脈自由 context-free grammars

認識機械との対応

- 文の認識機械（与えられた文字列が文か文でないかを判定する機械）の複雑さは、文法の階層によって異なる:
  - type-0 の言語を認識するには、Turing機械が必要（なこともある）
  - 文脈依存言語を認識するには、線形有界オートマトン linearly bounded automata が十分、必要（なこともある）
  - 文脈自由言語を認識するには、プッシュダウンオートマトン pushdown automata が十分、必要（なこともある）
  - 正規言語を認識するには、有限オートマトン finite automata で十分

有限オートマトン finite automata

- 有限オートマトンは, 5つ組

M = (Q, Σ, δ, q_0, F), ただし

1. Q：状態 states の有限集合
2. Σ:（許される）入力記号 acceptable input symbols の有限集合
3. δ: 遷移関数 transition function
4. $q_0 \in Q$：初期状態 initial state
5. F：終了状態 final states の集合

正規文法と有限オートマトン

- 正規文法で定義できる言語と、有限オートマトンで定義できる言語とは一致する。すなわち、この2つの定式化は等価である
- 以下に述べる、正規文法で定義される言語の性質は、いずれもテストすることができる:
  - 2言語の等価性
  - 定義した言語が空かどうか
  - 所与の文字列が、所与の言語の要素かどうか
- 残念なことに、正規文法で定義できる言語というのは、非常に限られた言語だけである

文脈自由文法の性質

- コンパイラ言語の構文解析は、所与の文字列が、文脈自由言語 context-free grammar, or cfg に従っているかどうかをチェックすることと考えられる
- 以下のような（他にもたくさん）話題がある
  - 曖昧性の問題
  - 演算子の優先順位
  - 文法の変換

文法の変換 transformations

- 文脈自由文法を、任意の形式に変換するような一般的アルゴリズムは存在しない
- しかし、任意の文脈自由文法は、Chomsky 標準形に変換することができる

  A→BC　(A, B, C∈N)

  A→a　　(A∈N, a∈T)
- グライバッハ Greibach 標準形もよく知られている

Backus-Naur form

- **::=**　左辺は右辺で定義される
- **< >**　範疇の名称は角括弧angle brackets で囲む
- 例:

```
<program> ::=
    program
        <declaration_sequence>
    begin
        <statements_sequence>
     end ;
```

Extended Backus-Naur form

- N. Wirth が Pascal と Modula-2 を定義するのに使用
- 1981年、British Standards Institute が標準化
- 記号の追加
  - **|**　　または
  - *****　　Kleene の星印
  - **( )**　メタレベルの括弧
- 比較

BNF と EBNF の比較

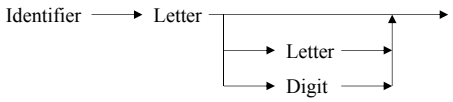
BNF

⟨digit⟩ → 0
⟨digit⟩ → 1
…
⟨digit⟩ → 9
⟨unsigned_integer⟩ → ⟨digit⟩
⟨unsigned_integer⟩ → ⟨digit⟩ ⟨unsigned_integer⟩

EBNF

⟨digit⟩ → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨unsigned_integer⟩ → ⟨digit⟩ ⟨digit⟩*

図式表現



3章　ステップ1：問題の把握

- BNFと文法
- BNFとEBNF
- 言語仕様
- プログラムと言語仕様との関係

「コンパイラ入門 C#で学ぶ理論と実践」より

3.2 BNF（Backus Naur Form)

- BNF
  - 「文法」を記述する表記法
  - コンピュータ言語を表す為に使われることが多い
- 英文法
  - 単語と単語の構成・関係を表す
  - 5文型は単語の品詞から英文の型を表現している
- プログラム言語の文法
  - プログラムの最小構成要素の構成・関係を表す
  - 変数、キーワード、オペレータなどの関係
  - 代入文の①abc=123、②123=abc、どちらが正しい？

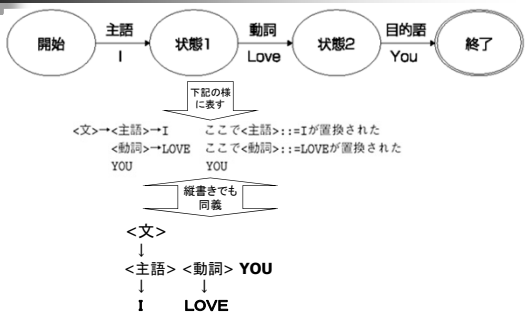
3.2 BNFの定義

- BNF
  - ターミナル（終端記号）
  - ノンターミナル（非終端記号）で< >と表記する
  - 左辺と右辺はターミナルとノンターミナルの集合体
  - 左辺 ::= 右辺
    - 本書では左辺はノンターミナルだけに制限する
- 例題
  - ＜文＞　　::= ＜主語＞ ＜動詞＞ ＜目的語＞
  - ＜主語＞　　::= I
  - ＜動詞＞　　::= Love
  - ＜目的語＞　　::= You
- 「::=」は置き換えるという意味、以後「→」を使用

3.2 BNFの定義

- BNFが出来ること
  - 文字列が文法に合致しているかどうかを「識別」できる
  - 置き換えのステップを「導出」と呼ぶ
- 例題
  - ステップ1～先頭から開始される～
    ＜文＞
  - ステップ2～＜文＞は＜主語＞＜動詞＞＜目的語＞によって置き換えられる～
    ＜主語＞＜動詞＞＜目的語＞
  - ステップ3～＜主語＞は I によって置き換えられる～
    I ＜動詞＞＜目的語＞
  - ステップ4～＜動詞＞は Love によって置き換えられる～
    I Love ＜目的語＞
  - ステップ5～＜目的語＞は You によって置き換えられる～
    I Love You
  - ステップ6～＜文＞は **I Love You** に変換換された～

3.2 BNFの定義



3.2 BNF例～数値の識別

- 数値
  - 1文字以上の数字
- BNF
  - <number> → <digit> <number1>
  - <number1> → ε
    | <digit> <number1>
  - <digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - 注：| （OR）は選択を示す（次スライド参照）
- 識別できる数値の例
  - 1、1、2、3、5、8、13、21・・・
- 識別できない数値の例
  - -123（マイナスは定義されていない）、abc(数字ではない)

3.2 BNF例～数値の識別

- BNF
  - <number> → <digit> <number1>
  - <number1> → ε| <digit> <number1>
  - <digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- BNF（上記の記述と同義）
  - <number> → <digit> <number1>
  - <digit> → 0
  - <digit> → 1
  - <digit> → 2
  - <digit> → 3
  - <digit> → 4
  - <digit> → 5
  - <digit> → 6
  - <digit> → 7
  - <digit> → 8
  - <digit> → 9
  - <number1> → ε
  - <number1> → <digit> <number1>

3.2 BNF例～数値の識別

- 構文解析木
  - 識別に到るBNF（右辺と左辺）を表したもの
  - 前ページの例題でアニメーションが付いた部分を並べたもの
- 1を識別した場合の構文解析木
  - <number> → <digit> → **1**
    <number1> → ε
- 13の識別した場合の構文解析木
  - <number> → <digit> → **1**
    <number1> → <digit> → **3**
    <number1> → ε
- ε（エプシロン）の意味
  - 該当するBNFやシンボルが「選択・識別」されなかった意味
  - 選択・識別する物が無いと明確に示す

3.2 BNF例～数値の識別

- 構文解析木の表記について
  - トポロジーが一致していればどんな形式でもかまわない
  - 下記の解析木は同じ意味
- 横書きの場合（13を識別した場合）
  - <number> → <digit> → 1
    <number1> → <digit> → 3
    <number1> → ε
- 縦書きの場合（13を識別した場合）

<number>
↓
<digit> <number1>
↓　↓
1　<digit> <number1>
↓　↓
3　ε

3.2 BNF例～文字列の識別

- 文字列
  - 1文字以上の文字でプログラムでは識別子と呼ばれる
- BNF
  - <ident> → <letter> <ident1>
  - <ident1> → ε
    | <letter> <ident1>
  - <letter> → a | b | c | d | e | f | g | h | i | j | k | l | m |
  - n | o | p | q | r | s | t | u | v | w | x | y | z |
  - A | B | C | D | E | F | G | H | I | J | K | L | M |
  - N | O | P | Q | R | S | T | U | V | W | X | Y | Z
- 識別できる数値の例
  - a、ab, abc, xyz, Hello・・
- 識別できない数値の例
  - 123、abc123（数字は定義されていない）

3.2 BNF例～文字列の識別

- Abを識別した場合の構文解析木
  - ステップ1： **<ident>**
  - ステップ2： <ident>→**<letter>**
    <ident1>
  - ステップ3： <ident>→<letter>→**A**
    <ident1>
  - ステップ4： <ident>→<letter>→A
    <ident1>→**<letter>**
    <ident1>
  - ステップ5： <ident>→<letter>→A
    <ident1>→<letter>→**b**
    <ident1>
  - ステップ6： <ident>→<letter>→A
    <ident1>→<letter>→b
    <ident1>→**ε**

3.3 BNFとEBNF

- EBNF
  - Extended BNF（拡張されたBNF）
  - BNFよりコンパクトに記述できる
- （ ）によるグルーピング
  - まとめて処理できる
- ＊による0回以上の繰り返し
  - 再帰呼び出しを省略できる
- ＋による1回以上の繰り返し
  - 再帰呼び出しを省略できる
- [ ]により二者択一の選択
  - εを使わずに処理できる

3.3 EBNF～（ ）と＊

- BNF（識別子）
  - <ident> → <letter> <ident1>
  - <ident1> → ε
    → <lettter> <ident1>
    → <digit> <ident1>
  - <letter> → 前例と同じ（英小文字、英大文字）
  - <digit> → 前例と同じ（0～字、数値）
- 同等のEBNF
  - <ident> → <letter> (<letter> | <digit>)*
- （ ）の効果
  - (<letter> | <digit>)によって2つのノンターミナルがまとめて処理
- ＊の効果
  - (<letter> | <digit>)*により<ident1>の再帰呼び出しが不要

3.3 EBNF～[ ]

- BNF
  - <program> → MODULE <ident> ; <additional> BEGIN・・・END <ident> .
  - <additional>→ ε
    → <decllist>（変数定義が行われるノンターミナル）
- 同等のEBNF
  - <program> --> MODULE <ident> ; [<decllist>] BEGIN・・・END <ident> .
- 例題1～変数が無い場合は[ ]内が選択されなかった
  - MODULE PROGRAM;
  - ～ここに変数定義が無い～
  - BEGIN
    ・・・
- 例題2～変数がある場合は[ ]内が選択された
  - MODULE PROGRAM;
    VAR I, J, K : INTEGER;
  - BEGIN
    ・・・

3.3 EBNF～＋

- BNF
  - <decllist> → VAR <decilist1>
  - <decllist1> → <identlist> : <type> ; <decllist2>
  - <decllist2> → ε
    → <decilist1>
- 同等のEBNF
  - <decllist> -->VAR (<identlist> : <type> ;)+
- 例題1～<decllist>が＋により1回選択された場合（前ページの例題2）.
- 例題2～変数定義のラインは1回以上何回定義されてもよい（この例では3回）
  - MODULE PROGRAM;
    VAR I, J, K : INTEGER;
    VAR a, b : INTEGER;
    VAR z, z, x, y, z : INTEGER;
    ・・・
  - BEGIN
    ・・・