構文法
プログラム言語論

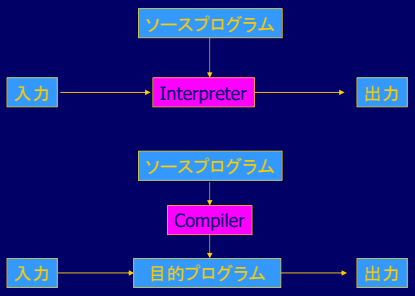
櫻井彰人

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

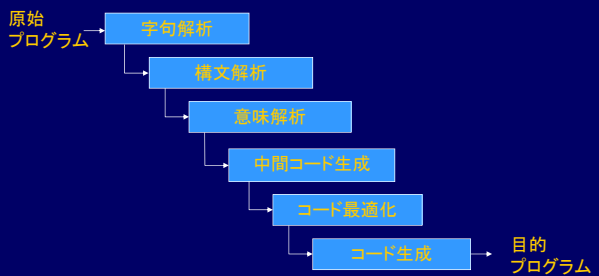
- 処理(翻訳・実行)方式の違い



言語と処理(翻訳・実行)方式

- 言語と処理とは、本来、独立な概念のはず
  - すなわち、どの言語にも interpreter があり、compiler があってよい
  - おまけに、多くの場合、compiler があれば interpreter は不要である(interpreter は、実行が遅いため)
  - さらに、compilerを作るのは難しくはない。
- では、interpreter は不要か？
- いや、必要！
  - compile できない(compileしても意味がない)機能をもつ言語がある
    - マクロ機能がこれに相当する。たとえば、LISP
  - compileする時間がもったいない利用環境の言語がある
    - たとえば、JavaScript
  - interactive性を重視する利用環境の言語がある

コンパイルの典型的な流れ



詳細については、コンパイラの本をよむこと

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 – 復習 かな？

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>

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

$A \rightarrow X_1\ X_2 \ldots X_m$

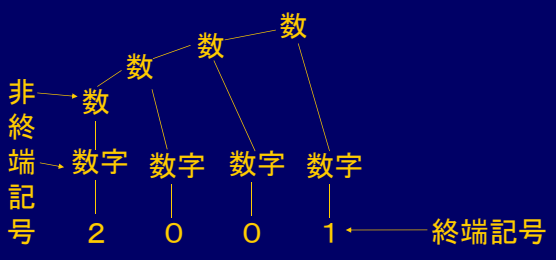
$A \rightarrow Z_1\ Z_2 \ldots Z_m$

$A \rightarrow X_1\ X_2 \ldots X_m \mid Z_1\ Z_2 \ldots Z_m$

と記述

文法例1

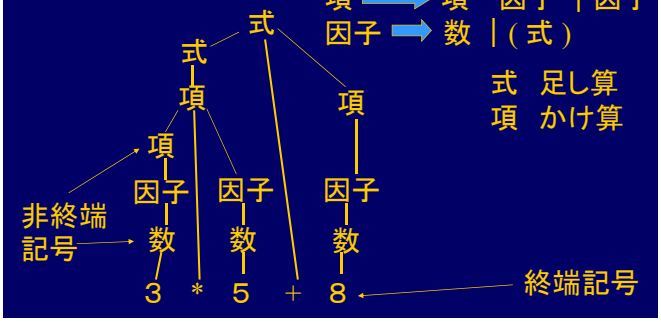
数 → 数　数字　|　数字
数字 → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



文法例2

式 ⟹ 式 + 項 ｜ 項
項 ⟹ 項 * 因子 ｜因子
因子 ⟹ 数 ｜(式)

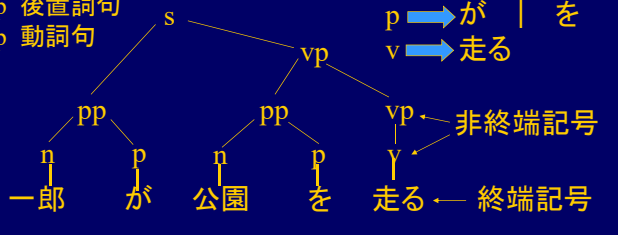
式　足し算
項　かけ算



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

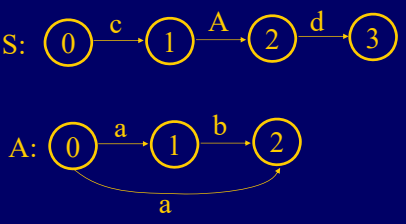
3型　正規文法　　受理：有限状態オートマトン
A ⟹ a　　A ⟹ a B
2型　文脈自由文法　　受理：プッシュダウン・オートマトン
A ⟹ X_1 X_2 …X_m
1型　文脈依存文法　　受理：線型有界オートマトン
$Z_1Z_2…Z_n$ ⟹ X_1 X_2 …X_m　　n ≦ m
0型　句構造文法　　受理：チューリング機械
右辺・左辺とも任意

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

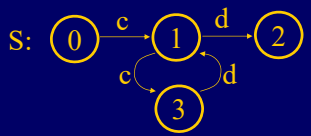
S,A 非終端記号
a,b,c,d 終端記号



決定性プッシュダウンオートマトン

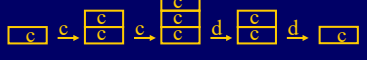
S ⟹ cSd ｜ cd

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

$L = \{ a^n b^m c^n d^m \mid n,m \geqq 1 \}$　　u^k uのk回並び

aa bbbb cc dddd

$L=\{ wcw \mid w$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

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

入力
一郎が公園を走る

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 – 復習 ここまで かな？

深さ優先＋最右

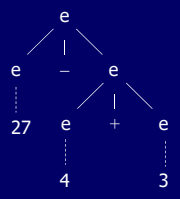
s ⇒ pp vp
vp ⇒ pp vp ｜ v
pp ⇒ n p
n ⇒ 一郎 ｜ 公園
p ⇒ が ｜ を
v ⇒ 走る

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