Higher-Order Grammar のススメ

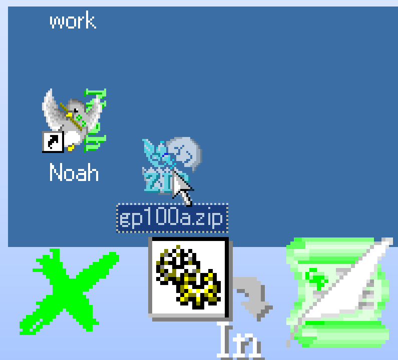
at ngcom (Sep. 11 2011)

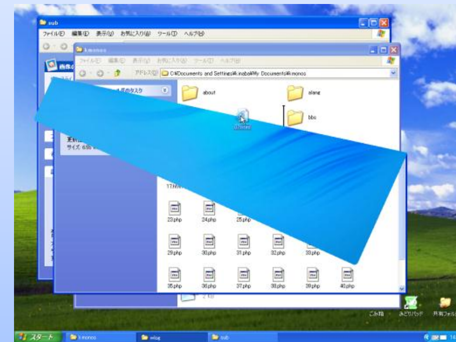
稲葉 一浩

自己紹介

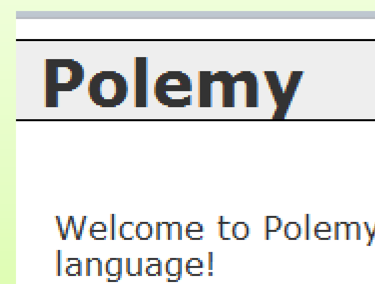
稲葉 一浩
http://www.kmonos.net/
Twitter: @kinaba

1998～ ：Windows用ツール作り



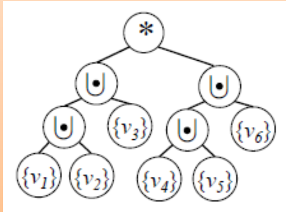


2002～ : プログラミング言語いじり趣味など



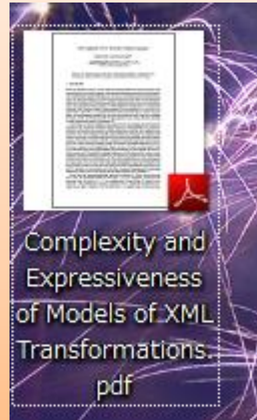
自己紹介

2004 ～ : 東大
<XML処理の理論的基礎>

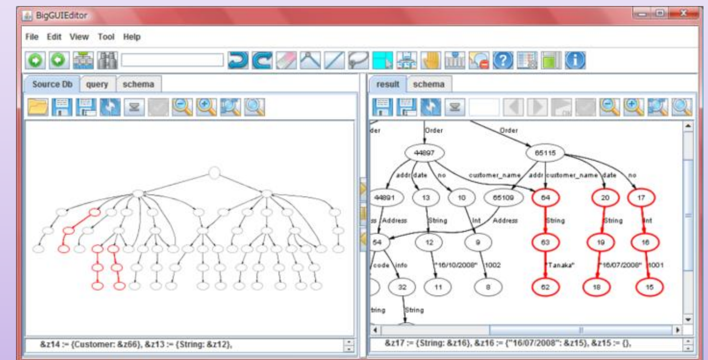


MTran

Back To [Publications]



2009 ～ : NII <双方向グラフ変換>



2011 ～ : Google <Chrome OS>

[chrome]

Revision 100000

Jump to revision: 100000

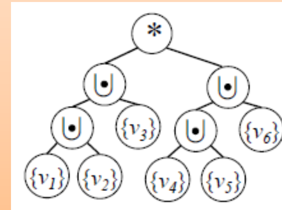
Author: kinaba@chromium.org

Date: Wed Sep 7 20:20:46 2011

今日のお話

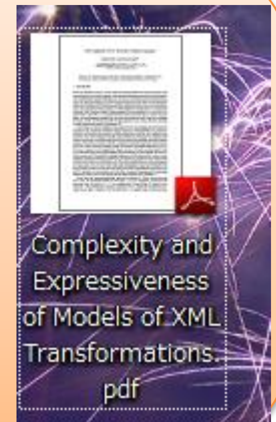
- この頃やった話 →
  - Inaba and Maneth, “The Complexity of Tree Transducer Output Languages”, FSTTCS 2008
  - Inaba, “Complexity and Expressiveness of Models of XML Translations”, PhD Thesis, 2009
- の一部
  （の一部（に関連する））
  話

2004 〜 : 東大
<XML処理の理論的基礎>



MTran

Back To [Publications]



注意事項

- XML一切関係ない
- いまのところ純粋に理論の話です

Chomsky 階層 [1956]

Type-0 文法

A B ::= C “+” D A E

B A ::= C C C

(左辺に2文字以上あっていい文法)

文脈依存文法

文脈自由文法

E ::= T | E “+” T

T ::= “(“ E “)” | “0” | “1”

正規文法

(a|b)*c(d|e*|f)

Chomsky 階層 [1956]

Type-0 文法

文脈依存文法

GOOD　表現力高い

BAD　文法として、書きにくい

BAD　Parsing等の計算量が重い

文脈自由文法

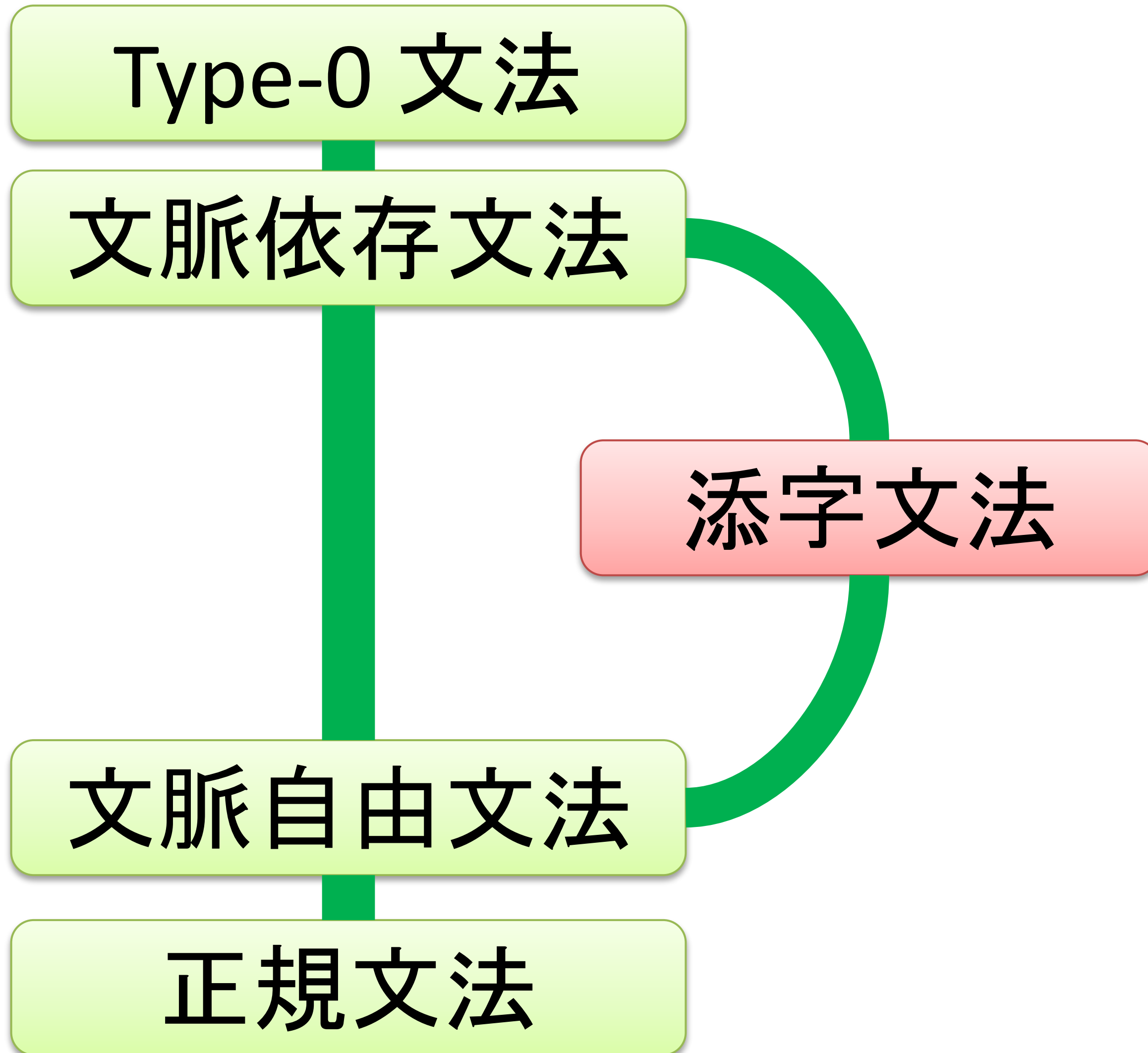
正規文法

GOOD　使いやすい (yacc, grep, ...)

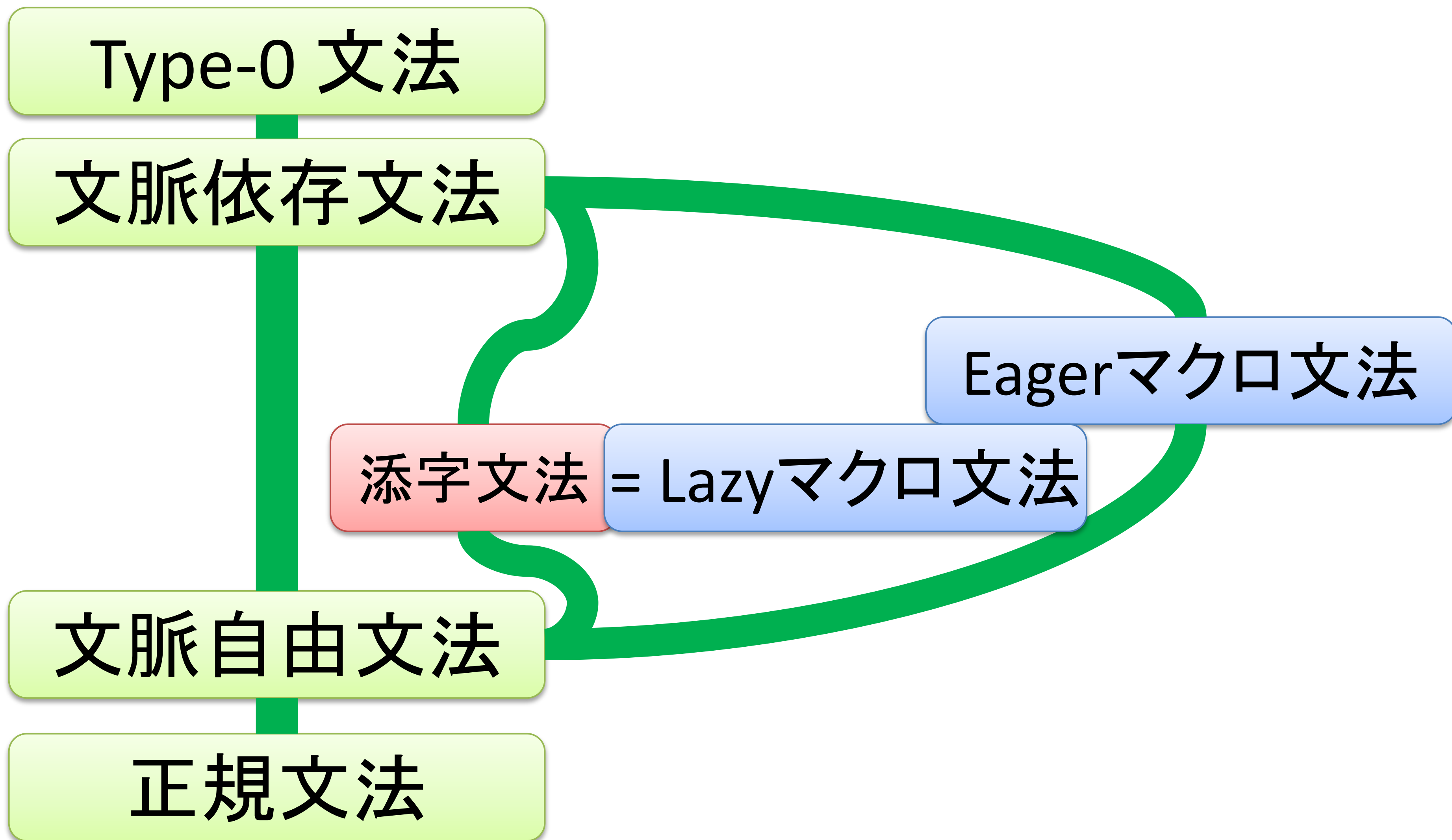
BAD　表現力低め

色々な拡張文法

Aho [1968]



Fischer [1968]



マクロ文法とは

- 文脈自由文法の "非終端記号" が パラメータを取る文法
  - 例1: インデントでブロックを表現 (Python風)

```
BLOCK(x)        ::= STATEMENT(x)  |  STATEMENT(x) BLOCK(x)
STATEMENT(x)::= S_IF(x)   | S_WHILE(x)  |  S_EXPR(x)
S_IF(x)         ::= x  if       EXPR  ¥n  BLOCK(x  ¥t)
S_WHILE(x)      ::= x  while  EXPR   ¥n  BLOCK(x  ¥t)
S_EXPR(x)       ::= x  E
E               ::= E + E  |  E * E  |   …
PROGRAM         ::= BLOCK()
```

マクロ文法とは

- 文脈自由文法の “非終端記号” が パラメータを取る文法
  - 例2: 正規表現の後方参照的な

```
/<([a-z]+)>…</¥1>/

XML          ::= ELEMENT( [a-z]+ )
ELEMENT(tag) ::=  < tag > … </ tag >
```

マクロ文法とは

- Parser Generator の実装があるかどうかは知らない
- 論文はあった
  - “index長に依存した長さの先読みを行う構文解析器生成系“ 東達軌, 山口文彦, 山崎克典 (東京理科大), 2008
    - インデントによるブロック表現に特化した、特殊な添字文法を扱う
    - http://www.nue.riec.tohoku.ac.jp/ppl2008/program.html

4.7　実装

　提案する構文解析器生成系は D 言語を用いて実装を行った。gentoo linux(kernel 2.6.16) gcc の D 言語フロントエンド gdc ver. 0.24 でコンパイルしたものによる動作を確認している。

Eager と Lazy (遅延評価)

AORB	::= a | b
PAIR(*x*)	::= < *x* , *x* >
S	::= **PAIR(AORB)**

- **S** にマッチする文字列を選びなさい(複数選択可)
  - “<a,a>”
  - “<a,b>”
  - “<b,a>”
  - “<b,b>”

Eager と Lazy

AORB	::= a | b
PAIR(*x*)	::= < *x* , *x* >
S	::= **PAIR(AORB)**

- S にマッチする文字列を選びなさい(複数選択可)
  - “<a,a>” Eager Lazy
  - “<a,b>” Lazy
  - “<b,a>” Lazy
  - “<b,b>” Eager Lazy

Eager と Lazy

AORB ::= a | b
PAIR(*x*) ::= < *x* , *x* >
S ::= **PAIR(AORB)**

- S
- PAIR(AORB)
- PAIR(a | b)
- PAIR(a) または PAIR(b)
- <a,a> または <b,b>

- S
- PAIR(AORB)
- < AORB , AORB >
- < (a|b) , AORB >
- < a, AORB > または < b, AORB>
- …
- <a,a> <a,b> <b,a> <b,b>

Eager と Lazy

Type-0 文法

文脈依存文法

どちらにも
そっちでしか
書けない文法がある

Eagerマクロ文法

添字文法 = Lazyマクロ文法

文脈自由文法

正規文法

※ 正確には、文法用語では、
・Lazy のことを OI
・Eager のことを IO
と呼びます。

Eager と Lazy の使い分け

- どちらでも変わらない (インデントは tab 1 固定)

```
PROGRAM      ::= BLOCK()
BLOCK(x)     ::= S(x) |  S(x) BLOCK(x)
S(x)         ::= x  if  EXPR  ¥n  BLOCK(x NEST)
NEST         ::= ¥t
```

- Eager (任意個のspace。ただし同じブロックは同じ深さで)

```
NEST         ::= “ ” | “ ” NEST
```

- Lazy (インデントは tab 1 または space 8 を同一視)

```
NEST         ::= ¥t | “        ”
```

他の例 (Lazy)

- データ型の宣言

```
data List a = Nil | Cons a (List a)
```

```
List(a) ::= Nil | Cons(a, List(a))
Bool    ::= False | True
```

- **List(Bool)** にマッチするデータ
  - Nil
  - Cons(False, Nil)
  - Cons(True, Cons(False, Nil))
  - …

Polymorphic Recursion

```
data Pair a b   = P a b
data Pow2Seq a = S a
               | B (Pow2Seq (Pair a a))
```

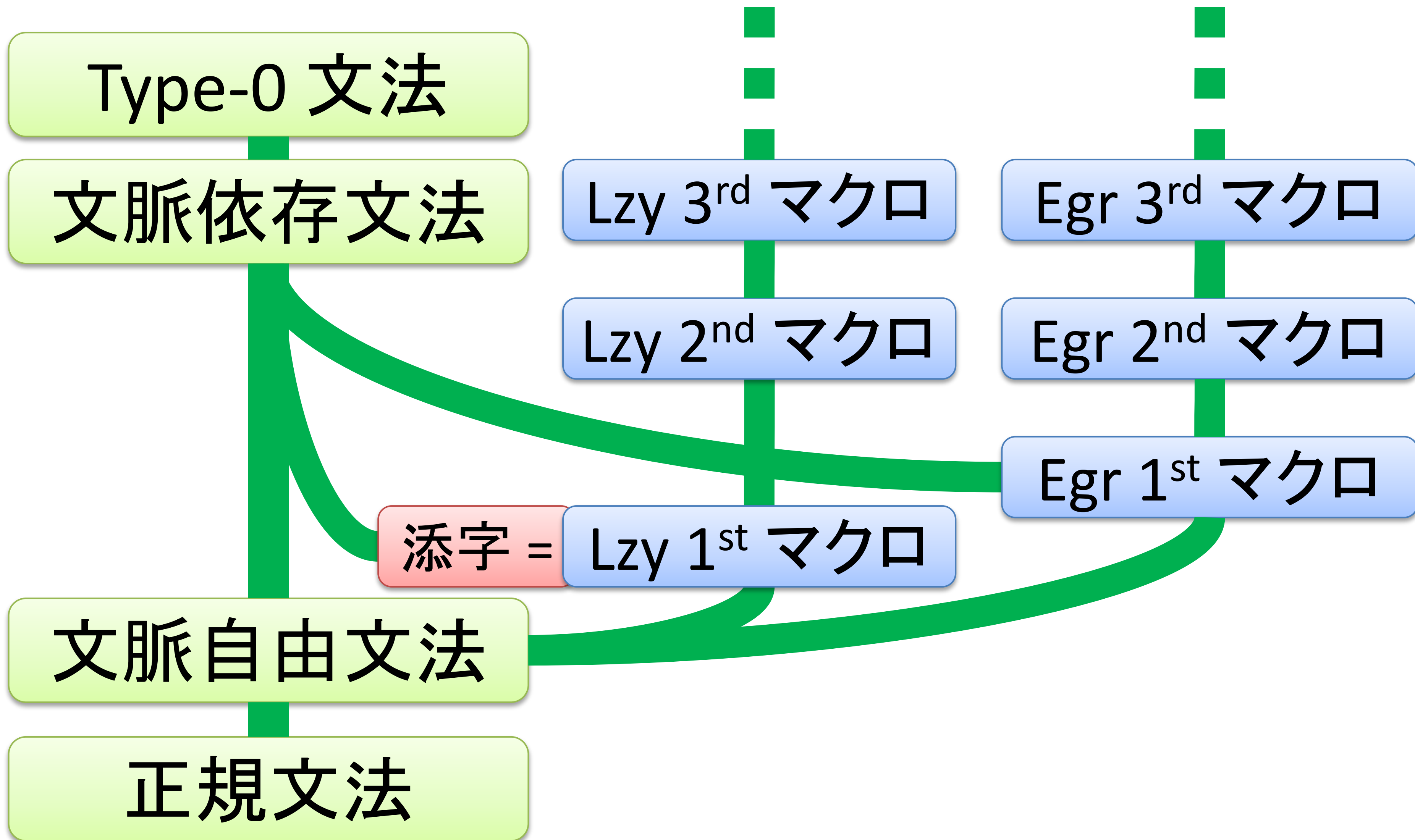
Pair(*a*,*b*)   ::= P(*a*,*b*)
Pow2Seq(*a*) ::= S(*a*)
                | B(**Pow2Seq**(**Pair**(*a*,*a*)))

- **Pow2Seq(Bool)** にマッチするデータ
  - S(False)
  - B(S(P(False,True)))
  - B(B(S(P(True,True),P(True,False))))
  - B(B(B(S(…8個…))))

「Polymorphic Recursion で表せる制約」＝「CbNマクロ文法で表せる制約」

ここから本題

高階マクロ文法 [Engelfriet&Schmidt 1977]



マクロ文法の「パラメタ」を高階にしてみよう！

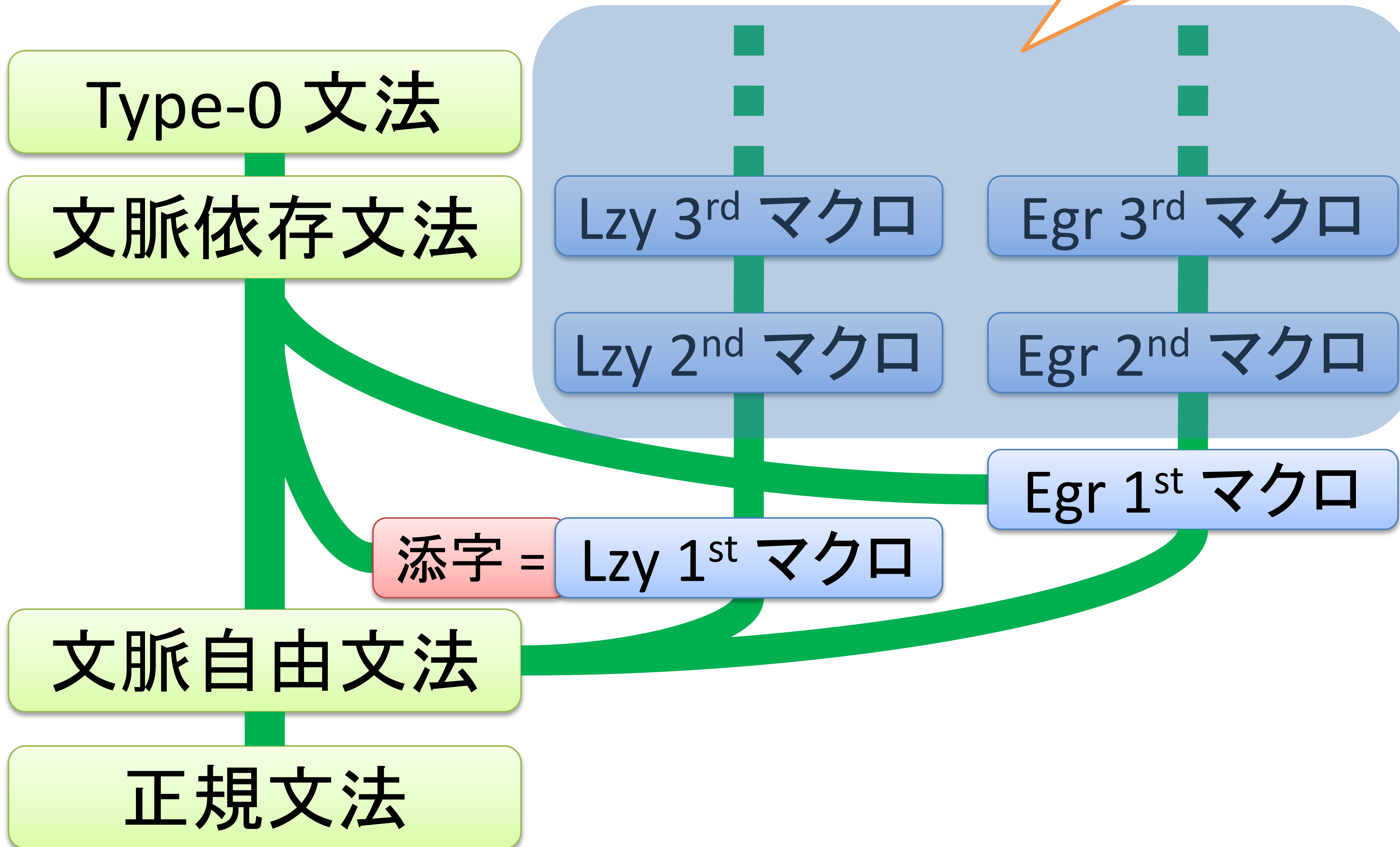
```
TYPE1(h, b) ::= h { b }
TYPE2(h, b) ::= h begin b end

SS(t) ::= | S(t) SS(t)
 S(t)       ::= IF(t) | WHILE(t) | UBE
 IF(t)      ::= t(if E, SS(t))
 WHILE(t) ::= t(while E, SS(t))
 UBE        ::= beginend S(TYPE2)

PROGRAM ::= SS(TYPE1)
```

```
if x<3 {
  beginend while x<100 begin
    if x%2==0 begin ... end
  end
  while x<200 { ... }
}
```

未解決（だった）問題



Type-0 文法

文脈依存文法

Eager なら Yes
[Maneth 02]

Lzy 3rd マクロ

Egr 3rd マクロ

Lzy 2nd マクロ

Egr 2nd マクロ

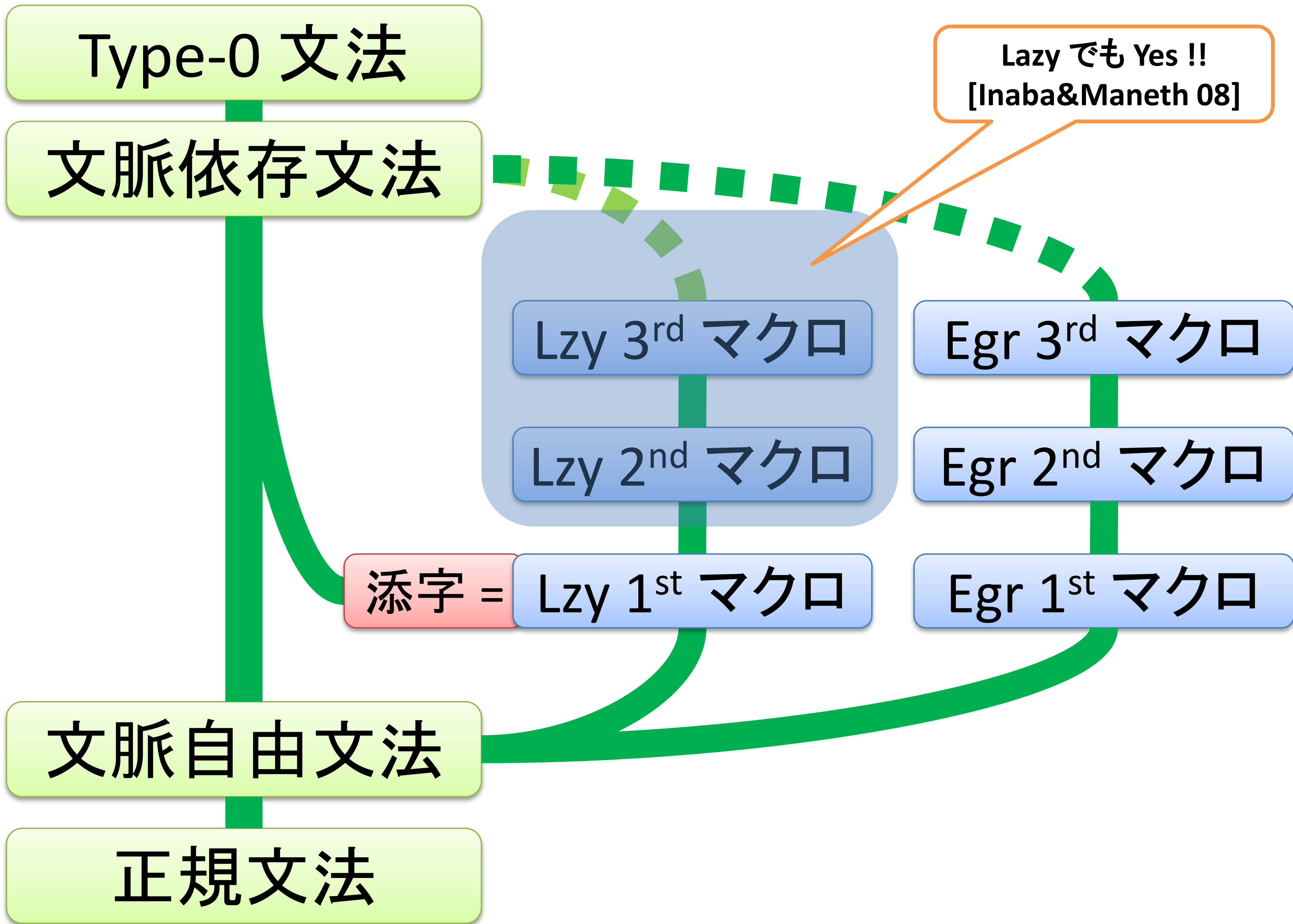
添字 =

Lzy 1st マクロ

Egr 1st マクロ

文脈自由文法

正規文法



Type-0 文法
文脈依存文法
Lazy でも Yes !!
[Inaba&Maneth 08]
Lzy 3rd マクロ
Egr 3rd マクロ
Lzy 2nd マクロ
Egr 2nd マクロ
添字 =
Lzy 1st マクロ
Egr 1st マクロ
文脈自由文法
正規文法

別の見方：構文解析の計算量

「こいつらの NSPACE(n) ⊆ EXPTIME 以下の構文解析アルゴリズムは作れるか？」

Type-0 (Turingマシン)

文脈依存 NSPACE(n)

Lzy 3rd マクロ

Egr 3rd マクロ

Lzy 2nd マクロ

Egr 2nd マクロ

Egr 1st マクロ ⊆ P

添字 = Lzy 1st マクロ ⊆ NP完全

文脈自由 ⊆ P

正規文法 DSPACE(1)

2k

※ kth マクロ文法の 2^2^…^n 時間構文解析はわりと簡単に作れる。「も少しマシになりませんか？」

定理 [Inaba 09] [Inaba&Maneth 08]

- Higher-Order Macro Grammar は
  - DSPACE(n)
    - ➔ O(n) の空間計算量で構文解析できる
    - ➔ せいぜい指数時間で構文解析できる
    - ➔ (理論上は)文脈依存文法に書き直せる
  - NP 完全
    - ➔ (P=NP でない限り) 多項式時間で解析するのは難しい
    - ➔ 文脈依存言語の全てを書けるわけではない

**「NP完全なので難しいと言えば難しいけど
そこまで絶望的に難しいわけではない！」**

証明の方針

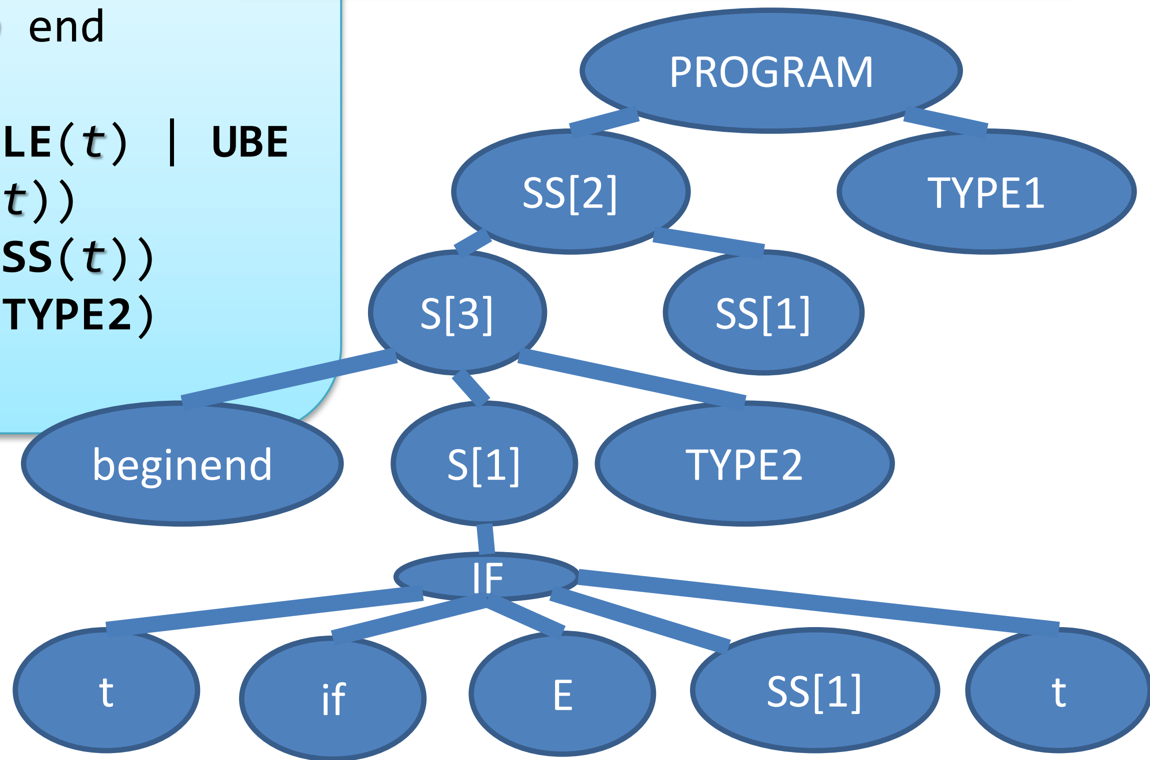
おおざっぱに言うと

- 「構文木のプリティプリンタ」を考える
- 構文解析とはその逆計算であると考える
- 頑張って（計算量爆発しないように）逆計算

高階マクロ文法の構文木

- (Eagerの場合)普通の文脈自由文法と同じ
  - 「どの構文規則を使ったか」をノードにした木

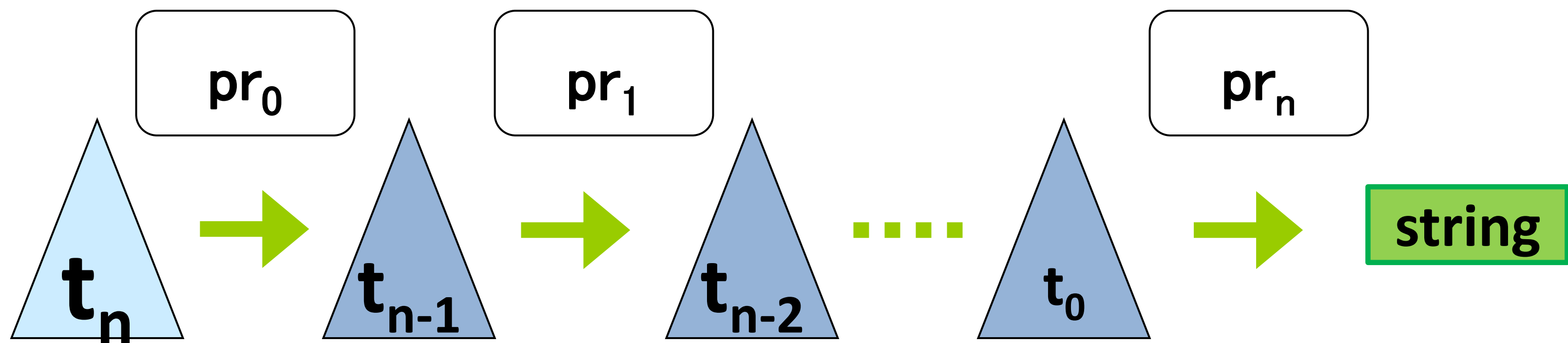
```
TYPE1(h, b) ::= h { b }
TYPE2(h, b) ::= h begin b end
SS(t) ::= | S(t) SS(t)
 S(t)       ::= IF(t) | WHILE(t) | UBE
 IF(t)      ::= t(if E, SS(t))
 WHILE(t) ::= t(while E, SS(t))
 UBE        ::= beginend S(TYPE2)
PROGRAM ::= SS(TYPE1)
```



プリティプリンタ

- 単に構文木を interpret する関数
- [Theorem: Engelfriet&Vogler 1988] 高階関数を使わず
  - n階関数だけを実行してn-1階関数のみに落とす
  - n-1階関数だけ実行してn-2階関数のみに落とす
  - …
  - 1階関数を実行して普通のCFGの構文木に落とす

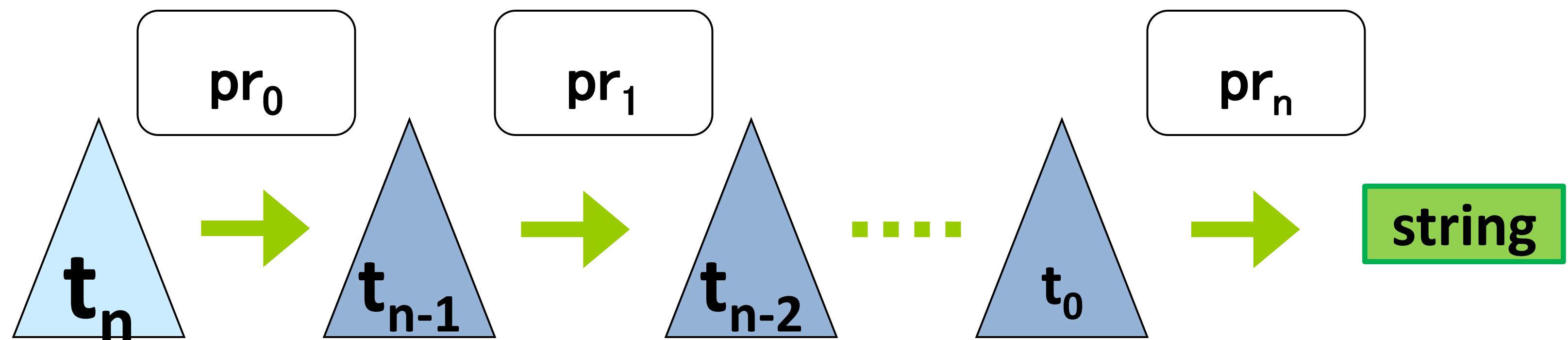
という n 個の一階関数の合成で書ける



計算量

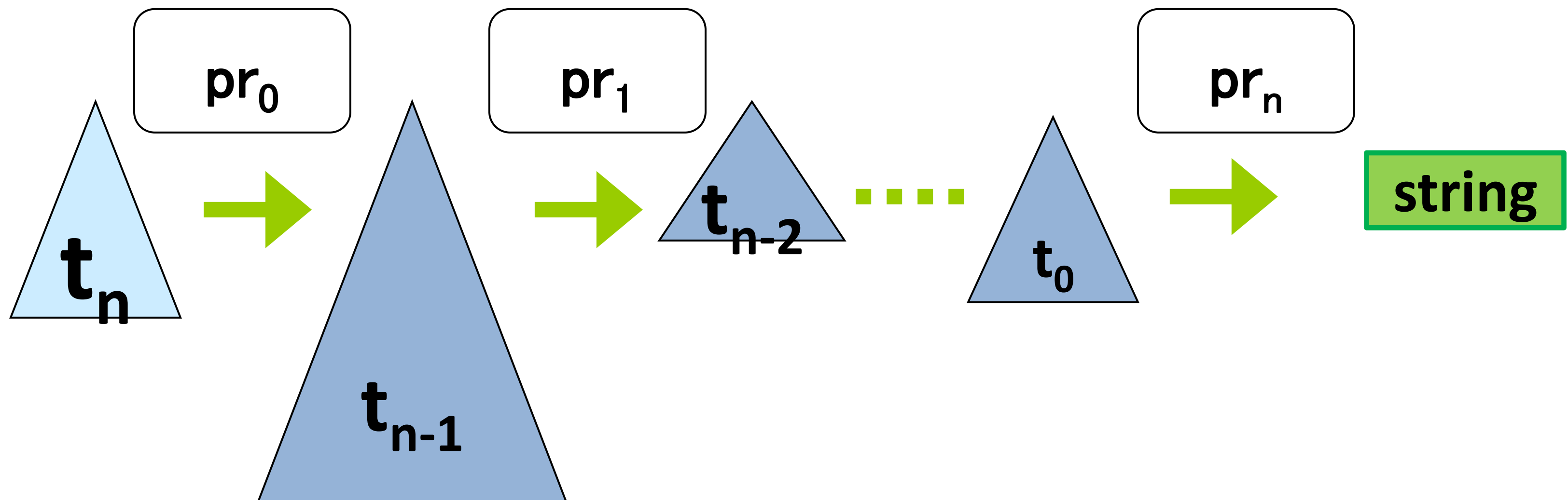
- 1段階 pretty print 関数 pr_i の入出力テスト問題
  - Input: tree x and tree y
  - Output: “f(x) = y” ?

が 時間計算量 $O(\ t(|x|+|y|)\ )$
空間計算量 $O(\ s(|x|+|y|)\ )$で解けるなら
（非決定性TMを使えば）全体はその総和で解ける



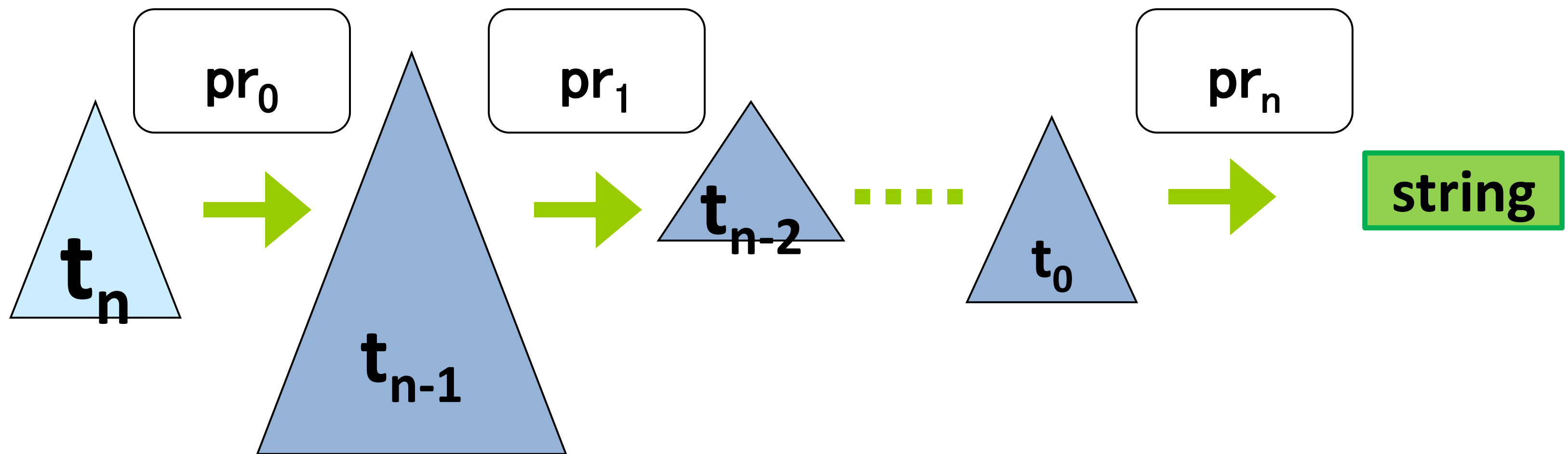
計算量

- 「１段階pretty print関数 pr_i の入出力テスト問題」を効率よく解きたい
  - ➔Theorem:　Eager なら　P ∩ DSPACE(n)
  - ➔Theorem:　Lazy なら　NP-complete ∩ DSPACE(n)
- 途中に出てくる木のサイズを確実に小さくしたい

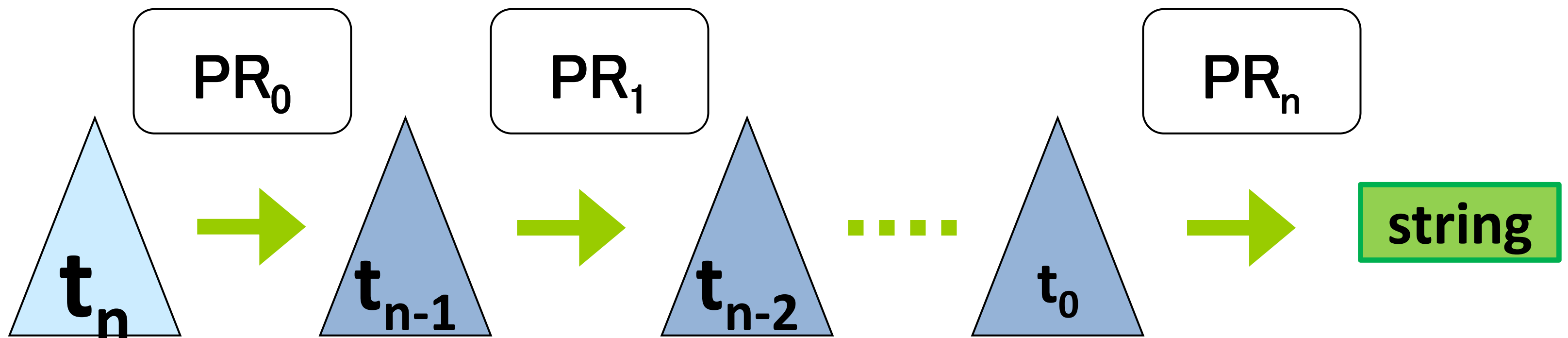


計算量

- 途中に出てくる木のサイズを小さくしたい



➔ Theorem: できる



ところで何に使うの？（応用）

- 文法やその理論の主な応用先
  1. 自然言語向けの、文の構文解析
  2. プログラミング向けの、ソース/データの構文解析
  3. プログラミング言語の挙動の静的解析, Verification
  4. バイオなど向けの、遺伝子情報などの解析

ところで何に使うの

高階関数や
オブジェクトを使う言語だと
必然的に高階になる

- 文法やその理論の主な応用先
  1. 自然言語向けの、文の構文解析
  2. プログラミング向けの、ソース/データの構文解析
  3. プログラミング言語の挙動の静的解析, Verification
  4. バイオなど向けの、遺伝子情報などの解析

文法を使った解析の例

- リソース使用法解析

```
FILE* fp = fopen(filename, “r”);
while( !feof(fp) )
  { ... fread( ... ); ...}
fclose(fp);
```

```
PROG ::= “fopen” LOOP “fclose”
LOOP ::= “feof”
       | “feof” “fread” LOOP
```

PROG ⊆ “fopen” (“feof” | “fwrite” | “fread” | “ftell”)* “fclose” ?

```
interface Stream { String readLine(); void close(); }
class FileStream extends Stream { ... }

void processFile( String name, Fun<void(Stream)> f ) {
  FileStream fp = new FileStream(name); f(fp); fp.close();
}
void echo( Stream input ) {
  while( String line = input.readLine() ) println(line);
  input.close();
}
void main() { processFile("foo.txt", echo); }
```

PROCESSFILE(*f*) ::= "fopen" *f*("fread", "fclose")
ECHO(*rd*, *cl*) ::= **LOOP**(*rd*, *cl*) *cl*
LOOP(*rd*, *cl*) ::= *rd* | *rd* **LOOP**(*rd*, *cl*)
MAIN() ::= **PROCESSFILE**(**ECHO**)

MAIN ⊆ "fopen" ("feof" | "fwrite" | "fread" | "ftell")* "fclose" ?

ところで何に使うの

こっちの応用
考えたい

- 文法やその理論の主な応用先
  1. 自然言語向けの、文の構文解析
  2. プログラミング向けの、ソース/データの構文解析
  3. プログラミング言語の挙動の静的解析, Verification
  4. バイオなど向けの、遺伝子情報などの解析

終

Type-0 文法

文脈依存文法

Lzy 3rd マクロ

Egr 3rd マクロ

Lzy 2nd マクロ

Egr 2nd マクロ

Lzy 1st マクロ

Egr 1st マクロ

文脈自由文法

正規文法