Computational Models

- Constructive (algorithmic) vs Non-constructive Mathematics
- Various models
  - Semi-Thue systems: string => string
    - Thue systems: symmetric (a=>b implies b=>a)
    - *Term rewriting systems, Post Production systems*
    - *Grammars* (adds terminals/non-terminals)
    - lambda calculus (beta rule: substitution, alpha rule)
  - *Turing Machines, counter machines*
  - *Deductive Logic* (Peano, Russell/Whitehead, ...)
  - Recursive function theory (Kleene/Rosser)
- Church-Turing Hypothesis: All are equivalent!
- But what about models based on biological processes, quantum processes?

Panini? (term rewriting systems + grammars)

Term rewriting systems

- Post production systems studied mathematically in 20's
  - Shown to be equivalent to Turing machines
- Chomsky hierarchy (add terminals/non-)
  - Regular grammars: *A*=>*aB* or *A*=>*a* (FSMs)
  - Context free grammars: *A*=>*string* (NDPDA)
  - Context sensitive grammars: *x A y*=>*x string y*
    - (Linear Bounded nondet automaton)
  - Unrestricted grammars: *string* => *string* (TMs)
- *Pānini? Sanskrit*
  - generation (Pānini, past scholars) vs recognition (now)
    - *Posit non-terminals, etc. or not?*

Context-Free vs Context-Sensitive Grammars

- Earlier examples: Context-Free L-systems
- Context-Sensitive L-system:

$$\begin{aligned} \omega &: \ baaaaaaaa \\ p_1 &: \ b < a \ \rightarrow b \\ p_2 &: \ \ \ \ b \ \rightarrow a \end{aligned}$$

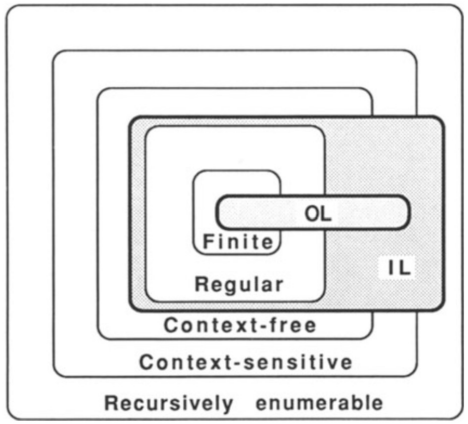


Figure 1.2: Relations between Chomsky classes of languages and language classes generated by L-systems. The symbols OL and IL denote language classes generated by context-free and context-sensitive L-systems, respectively.

- Simultaneous appl of the 2 rules: (signal ”b” going to the right)

$$\begin{aligned} &baaaaaaaa \\ &abaaaaaaa \\ &aabaaaaaa \\ &aaabaaaaa \\ &aaaabaaaa \\ &\ldots \end{aligned}$$

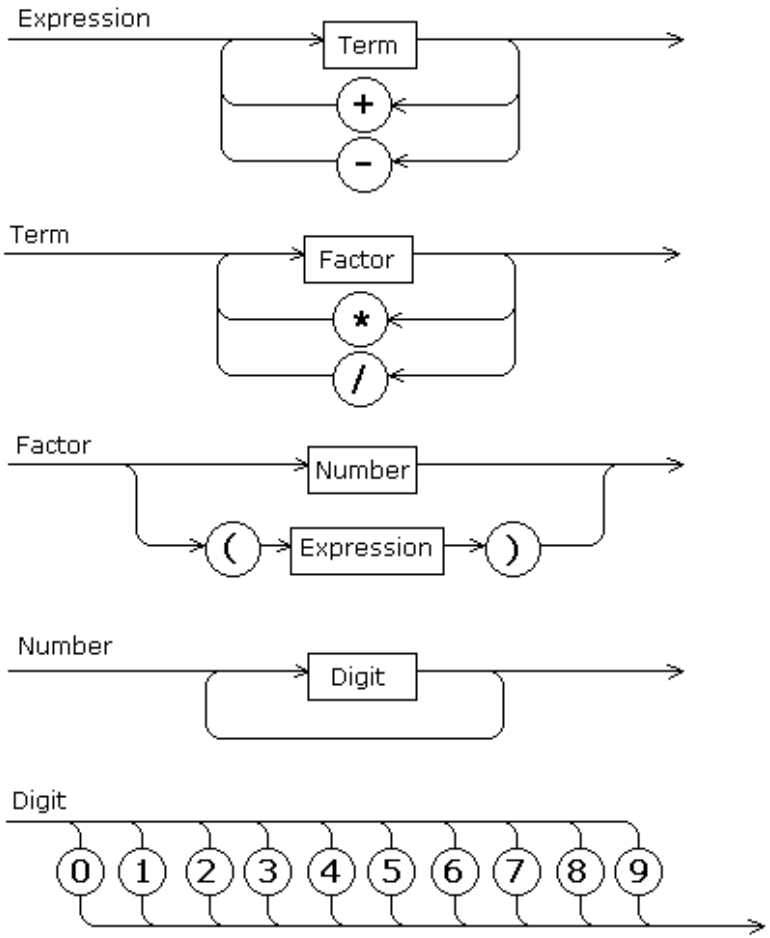
Qn: Are Hilbert curves CS or CF?

Syntax of expressions

(proposed)
Pāṇini Backus form

Backus Normal Form (BNF)
Backus Naur Form

E ::= T | E + T | E - T
T ::= F | T * F | T / F
F ::= (E) | Num | Var



Extended BNF

Usage	Notation
definition	=
concatenation	,
termination	;
alternation	|
optional	[...]
repetition	{ ... }
grouping	(...)
terminal string	" ... "
terminal string	' ... '
comment	(* ... *)
special sequence	? ... ?
exception	-

Context Free Grammar for Expressions

Simple (but “ambiguous”)

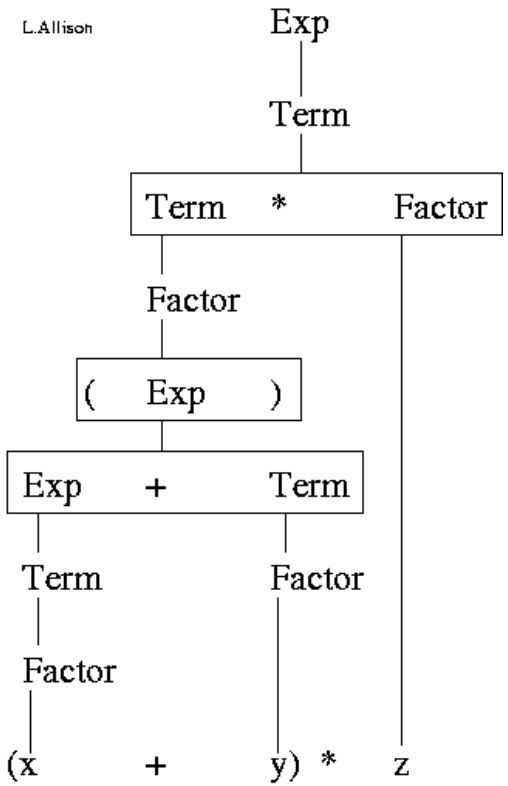
- E -> E op E
- E-> (E)
- op -> + | * | - | /
- E -> value

Better:

- E-> T
- E-> T+E
- E-> T-E
- T-> value| value*T | value/T

Or:

- E-> T
- E-> E+T
- E-> E-T
- T-> value| value*T | value/T



Ambiguity in C

statement = ... | selection-statement

selection-statement = ... | IF (expression) statement | IF (expression) statement ELSE statement

if (a) **if** (b) s; **else** s2;

if (a) { **if** (b) s; **else** s2; } // C: An else is associated with the lexically nearest preceding if that is allowed by the syntax.

Or?

if (a) { **if** (b) s; } **else** s2;

Can remove ambiguity with

statement: … | open_statement ; | closed_statement ;

open_statement: IF '(' expression ')' statement ; | IF '(' expression ')' closed_statement ELSE open_statement ;

closed_statement: … | IF '(' expression ')' closed_statement ELSE closed_statement ;

In K&R 2nd ed:

"With one further change, namely deleting the production *typedef-name: identifier* and making *typedef-name* a terminal symbol, this grammar is acceptable to the YACC parser-generator. It has only one conflict, generated by the `if-else` ambiguity."

Other problems

(A) *B

Is this A mult B
or
(typecast to A) (deref B)?

```
{
  T (x); ... // is this a type decl? Or, a function call? Context sensitive!
}
```

C is a context sensitive language! Have to weave lexing and parsing
Same with decl matching: $a^n b^n$ *vs* $a^n b^n c^n$ *vs* $a^n b^n c^n d^n$

Solution: Use symbol table (that keeps info about identifiers).
Symbol table is part of parsing analysis that is now interwoven with lexical analysis

When an identifier is seen by lexical analysis, looks up symbol table to see identifier's type.
The lexical analysis uses this info to return different ids such as type decl or function call

Another example:

```
typedef struct { ... } A;
int main() {
  int A = 10; // is this a redefinition?
  // if A is known to be a typedef, OK!
  int B = 20;
    ...
  int C = (A)*B;
}
```

$S \rightarrow aSXY | abc$
$cX \rightarrow Xc$
$bX \rightarrow bb$
$cY \rightarrow cc$
produces $a^n b^n c^n$
(CS grammar)

Parsing Expressions

- 1 pass or multiple pass?
- Ambiguity Resolution
- Shift Reduce conflicts
  - 2+3*5
- Reduce Reduce conflicts
  - Dangling if stmt
- Right vs Left Associativity
  - left-associativity with left-recursive production (C arith)
  - right-associativity with right-recursive production (exponentiation)

Some examples of Languages vs Grammars

Grammar for $a^n b^n$:

S => epsilon

S => aSb

Requires a stack for recognition

match a

match S // recursive call

match b

S => abc | aAbc
A => abC | aAbC
Cb => bC
Cc => cc

Grammar for properly bracketed expressions with (,)

Grammar for properly bracketed expressions with (,), [,]

Grammar for $a^n b^n c^n$

Grammar for $a^{2\hat{}n}$

Grammar for 0,1,10,11,101,1000, 1101, … (virahānka in binary)