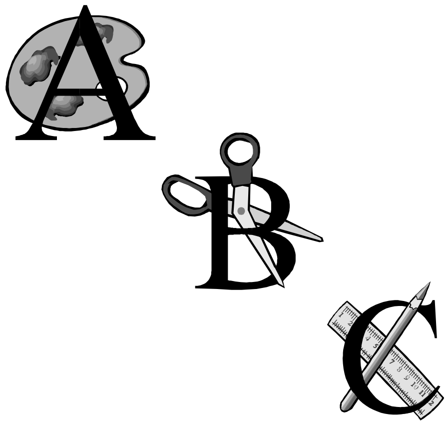
Lecture T3: Grammar



Review of Formal Languages

Alphabet = finite set of symbols.
- E.g., binary alphabet = {0, 1}

String = finite sequence of symbols from the alphabet.
- E.g., 011100101001 is a string over binary alphabet.

Language = (potentially infinite) set of strings over an alphabet.
- E.g., strings having same number of 0's and 1's:
  L = {01, 10, 1001, 011100101001, . . . }

Language recognition. (e.g., FSA)
- Is 011100101001 a string in language L?
- All computational problems can be expressed in this way.

Language generation. (e.g., RE)
- Set of rules for producing strings.

Why Learn Grammar?

Concrete applications:
- Better understanding of what computers can do.
- Compiler implementation.
- Natural language recognition / translation (linguistics).
- Models of physical world.

Grammar

Generates strings in language by a process of replacing symbols.
- Similar to regular expressions.

Four elements.
- Terminal symbols:
  characters in alphabet - denote by `0` or `1` for binary alphabet.
- Nonterminal symbols:
  local variables for internal use - denote by `<name>`.
- Start symbol: one special nonterminal.
  (analogous to start state in FSA)
- Production rules:
  replacement rules - denote by `<A> c` ⇒ `<D> b <B>`

A Familiar Example (abbreviated)

Terminals: horse, dog, cat, saw, heard, the

Nonterminals: <sentence>, <subject>, <verb>, <object>

Start symbol: <sentence>

Production rules:

```
<sentence> ⇒ <subject> <verb> <object>
<subject>  ⇒ the horse
<subject>  ⇒ the dog
<subject>  ⇒ the cat
<object>   ⇒ the horse
<object>   ⇒ the dog
<object>   ⇒ the cat
<verb>     ⇒ saw
<verb>     ⇒ heard
```

Some strings:

```
the horse saw the dog
the dog heard the cat
the cat saw the horse
```


Generating a String in Language

Start with the start symbol.

<sentence>

Generating a string in language:

<sentence>

Generating a String in Language

Start with the start symbol.

Use any applicable production rule.

<sentence> ⇒ <subject> <verb> <object>

Generating a string in language:

<sentence> ⇒ <subject> <verb> <object>

Generating a String in Language

Start with the start symbol.

Use any applicable production rule.

<subject> ⇒ the horse

Generating a string in language:

```
<sentence>  ⇒ <subject> <verb> <object>
            ⇒ the horse <verb> <object>
```

Generating a String in Language

Start with the start symbol.

Use any applicable production rule.

```
<object> ⇒ the dog
```

Generating a string in language:

```
<sentence>   ⇒ <subject> <verb> <object>
             ⇒ the horse <verb> <object>
             ⇒ the horse <verb> the dog
```


Generating a String in Language

Start with the start symbol.

Use any applicable production rule.

```
<verb> ⇒ saw
```

Generating a string in language:

```
<sentence>   ⇒ <subject> <verb> <object>
             ⇒ the horse <verb> <object>
             ⇒ the horse <verb> the dog
             ⇒ the horse saw the dog
```

one string in language

The C Language Grammar (abbreviated)

Terminals:

- `if do while for switch break continue typedef struct return main int long char float double void static ;( ) a b c A B C 0 1 2 + * - / _ # include += ++ ...`

Nonterminals:

- `<statement> <expression> <C source file> <identifier> <digit> <nondigit> <identifier> <selection-statement> <loop-statement>`

Start symbol: `<C source file>`

A string:

```
#include <stdio.h>
int main(void) {
   printf("Hello World!\n");
   return 0;
}
```


The C Language: Identifiers

Production rules:

```
<identifier> ⇒ <nondigit>
             ⇒ <identifier> <nondigit>
             ⇒ <identifier> <digit>

<nondigit>   ⇒ a | b | . . . | Y | Z | _

<digit>      ⇒ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Some identifiers:

```
x
f
temp
temp1
done
_CanStartWithUnderscoreButNot7
```

The C Language: Expressions

Production rules:

```
<expression>  ⇒ <identifier>
              ⇒ <constant>
              ⇒ <cond-expression>
              ⇒ <assign-expression>

<cond-expression>    ⇒ <expression>  > <expression>
                     ⇒ <expression> != <expression>

<assign-expression>  ⇒ <expression>  = <expression>
                     ⇒ <expression> += <expression>
```

Some expressions:

- `x`
- `x > 4`
- `done != 1`
- `x = y = z = 0`
- `x += 2.0`

This grammar also considers 4 = x a valid expression.

The C Language: Statements

Production rules:

```
<statement>           ⇒ <select-statement>
                      ⇒ <loop-statement>
                      ⇒ <compound-statement>
                      ⇒ <express-statement>

<select-statement>    ⇒ if (<expression>)
                      ⇒ if (<expression>)<statement>
                        else <statement>

<loop-statement>      ⇒ while (<expression>) <statement>
                      ⇒ do <statement> while (<expression>)

<express-statement> ⇒ <expression> ;
```

A statement:

```
while(done != 1)
   if (f(x) > 4.0)
      done = 1;
   else
      x += 2.0;
```


Grammars

In principle, could write out the grammar for English language.

In practice, need to write out grammar for C.

- Compiler check to see if your program is a valid "string" in the C language.
- The C Standard formalizes what it means to be a valid ANSI C program using grammar (see K+R, Appendix A13).
- Compiler implementation: simulate FSA and PDA machines to recognize valid C programs.

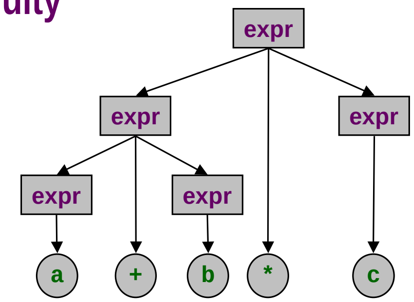
Ambiguity

Production rules:

```
<expr> ⇒ <expr> + <expr>
       ⇒ <expr> * <expr>
       ⇒ a | b | c
```

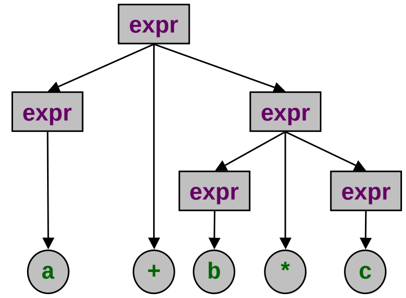
An ambiguous expression:

- `a + b * c`



Two different derivations (parse trees).

- `(a + b) * c`
- `a + (b * c)`



Postorder traversals of parse trees:

```
b a + c *
a c b * +
```

Ambiguity

Need more refined grammar:

```
<expr>    ⇒ <expr> + <T>
          ⇒ <T>
<T>       ⇒ <T> * <P>
          ⇒ <P>
<P>       ⇒ ( <expr> )
          ⇒ a | b | c
```

No ambiguous expressions.

- `a + b * c`
- `(a + b) * c`

Type III Grammar (Regular)

Limit production rules to have exactly one nonterminal on LHS and at most one nonterminal and terminal on RHS:

```
<A> ⇒ <B> a
<A> ⇒ <A> b
<B> ⇒ c
<C> ⇒ ε
```

Example:

```
<A> ⇒ <B> 0        Start = <A>
<B> ⇒ <A> 1
<A> ⇒ ε
```

Strings generated:

```
ε, 10, 1010, 101010, 10101010, …
```

Grammar GENERATES language = set of all strings derivable from applying production rules.

Type II Grammar (Context Free)

Limit production rules to have exactly one nonterminal on LHS, but anything on RHS.

```
<A> ⇒ b <B> <C> a <C>
<A> ⇒ <A> b c a <A>
```

Example:

```
<PAL> ⇒ 0 <PAL> 0        Start = <PAL>
      ⇒ 1 <PAL> 1
      ⇒ 0
      ⇒ 1
      ⇒ ε
```

Strings generated:

```
ε, 1, 0, 101, 001100, 111010010111, …
```

Language generated:

Type II Grammar (Context Free)

Example:

```
<S>  ⇒ ( <S> )        Start = <S>
     ⇒ { <S> }
     ⇒ [ <S> ]
     ⇒ <S> <S>
     ⇒ ε
```

Strings generated:

```
ε, (), ()[()], (([]{()})[](())), …
```

Language generated:

Type I Grammar (Context Sensitive)

Add production rules of the type:

```
[A] <B> [C] ⇒ [A] a [C]
```

where [A] and [C] represent some fixed sequence of nonterminals and terminals.

```
<A> <B> <C> ⇒ <A> b <C>
<A> hi <B> <C> <D> ⇒ <A> hip <C> <D>
```


Type 0 Grammar (Recursive)

No limitation on production rules: at least one nonterminal on LHS.

Example:

```
Start = <S>
<S> ⇒ <S> <S>          <A><B> ⇒ <B><A>
<S> ⇒ <A> <B> <C>      <B><A> ⇒ <A><B>
<A> ⇒ a                <A><C> ⇒ <C><A>
<B> ⇒ b                <C><A> ⇒ <A><C>
<C> ⇒ c                <B><C> ⇒ <C><B>
<S> ⇒ ε
```

Strings generated:

```
ε, abc, aabbcc, cabcab, acacacacacacbbbbbb, ...
```

Language generated:

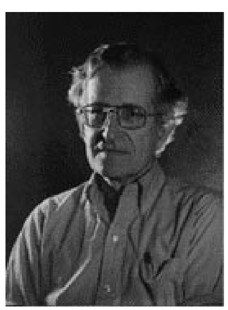
Chomsky Hierarchy

Powerful Machines ↓

Type	Machine	Grammar
III	FSA	regular
II	NPDA	context free
I	LBA	context sensitive
0	TM	recursive

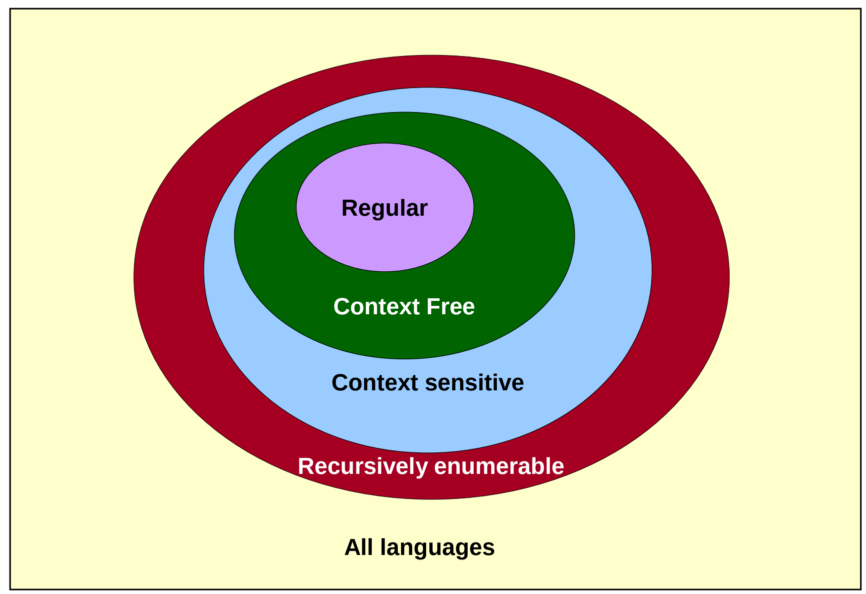
Expressive languages ↓

Essential one-to-one correspondence between machines and languages.



Noam Chomsky

Chomsky Hierarchy



FSA and Type III Grammar Equivalence

FSA's and Type III grammar are equally powerful.

- **Given an FSA, can construct Type III grammar to generate same language.**
- **Given Type III language, can construct FSA that accepts same language.**

Proof idea:

FSA	Type III Grammar
Start state	Start symbol
States	Nonterminals
Transition arcs	Production rules: <A> $\Rightarrow$ <B> a
Accept state	Production rules: <A> $\Rightarrow$ a

Compilers and Grammar

Compiler: translates program from high-level language to native machine language.

- **C $\Rightarrow$ TOY**

Three basic phases.

- **Lexical analysis (tokenizing).**
  - convert input into "tokens" or terminal symbols
  - # include <stdio.h> int main (void) { printf ("Hello World!\n") ; return 0 ; }
  - implement with FSA
  - Unix program lex

Note: as specified, grammar for <identifier> is not Type III. Easy exercise: make Type III.

Compilers and Grammar

Compiler: translates program from high-level language to native machine language.

- **C $\Rightarrow$ TOY**

Three basic phases.

- **Lexical analysis (tokenizing).**
- **Syntax analysis (parsing).**
  - implemented using pushdown automata since C language is (almost) completely described with context-free grammar
  - Unix program yacc

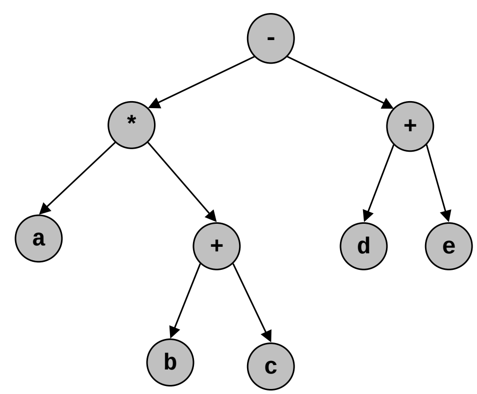
Compilers and Grammar

Compiler: translates program from high-level language to native machine language.

- **C $\Rightarrow$ TOY**

Three basic phases.

- **Lexical analysis (tokenizing).**
- **Syntax analysis (parsing).**
- **Code generation.**
  - parse tree gives structure of computation
  - traverse tree in postorder and create native code



Parse tree for expression:
(a * (b + c)) - (d + e)

Other Exotic Forms of Grammar

Lindenmayer systems:

- **Apply production rules SIMULTANEOUSLY.**
- **Falls in between Chomsky hierarchy levels.**

Example:

- **Production rules:**

```
0    ⇒ 1 [ 0 ] 1 [ 0 ] 0
1 [  ⇒ 1 1 [
```

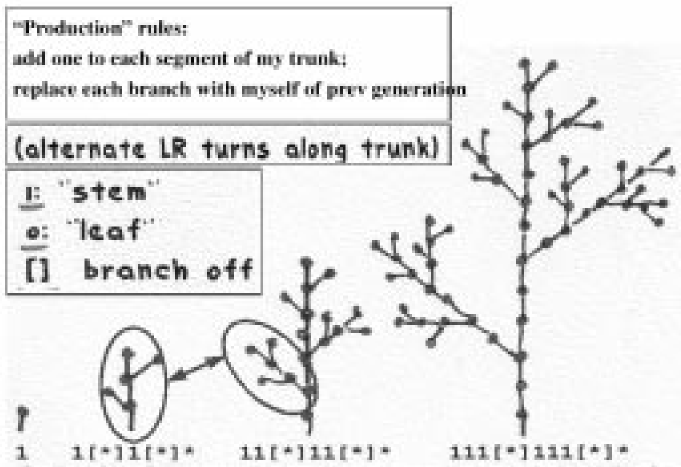
- **Start with 10. At stage i, apply rules to each symbol in string from stage i-1.**

-
```
10 ⇒ 1  [    0    ] 1  [    0    ]     0
   ⇒ 11 [1[0]1[0]0] 11 [1[0]1[0]0] 1[0]1[0]
   ⇒ 111 [*] 111 [*] *
```

*** denotes copy of previous string**

Other Exotic Forms of Grammar

Visualize in 2D:



What's Ahead?

Last 3 lectures developed formal method for studying computation.

Now, we get to use it!

3 of the most important ideas in computer science ahead.

- **Lecture T4: what can be computed?**
- **Lecture T5: designing high-performance algorithms?**
- **Lecture T6: why we can't solve problems like the TSP?**