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:
    \[ L = \{01, 10, 1001, 011100101001, \ldots \} \]

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:

\[ <\text{sentence}> \Rightarrow <\text{subject}> <\text{verb}> <\text{object}> \]
\[ <\text{subject}> \Rightarrow \text{the horse} \]
\[ <\text{subject}> \Rightarrow \text{the dog} \]
\[ <\text{subject}> \Rightarrow \text{the cat} \]
\[ <\text{object}> \Rightarrow \text{the horse} \]
\[ <\text{object}> \Rightarrow \text{the dog} \]
\[ <\text{object}> \Rightarrow \text{the cat} \]
\[ <\text{verb}> \Rightarrow \text{saw} \]
\[ <\text{verb}> \Rightarrow \text{heard} \]

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.

\[ \text{<sentence>} \Rightarrow \text{<subject>} \text{<verb>} \text{<object>} \]

Generating a string in language:

\[ \text{<sentence>} \Rightarrow \text{<subject>} \text{<verb>} \text{<object>} \]
Generating a String in Language

Start with the start symbol.

Use any applicable production rule.

\[ \text{<subject>} \Rightarrow \text{the horse} \]

Generating a string in language:

\[ \text{<sentence>} \Rightarrow \text{<subject>} \text{<verb>} \text{<object>} \]

\[ \Rightarrow \text{the horse} \text{<verb>} \text{<object>} \]
Generating a String in Language

Start with the start symbol.

Use any applicable production rule.

\[ \text{<object> } \Rightarrow \text{ the dog} \]

Generating a string in language:

\[ \text{<sentence> } \Rightarrow \text{ <subject> <verb> <object> } \]
\[ \Rightarrow \text{ the horse <verb> <object> } \]
\[ \Rightarrow \text{ the horse <verb> the dog} \]
Generating a String in Language

Start with the start symbol.

Use any applicable production rule.

\[<\text{verb}> \Rightarrow \text{saw}\]

Generating a string in language:

\[<\text{sentence}> \Rightarrow <\text{subject}> <\text{verb}> <\text{object}>\]
\[\Rightarrow \text{the horse} <\text{verb}> <\text{object}>\]
\[\Rightarrow \text{the horse} <\text{verb}> \text{the dog}\]
\[\Rightarrow \text{the horse saw the dog}\]

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

Production rules:

\[
\begin{align*}
\langle \text{identifier} \rangle & \Rightarrow \langle \text{nondigit} \rangle \\
& \quad \Rightarrow \langle \text{identifier} \rangle \ \langle \text{nondigit} \rangle \\
& \quad \Rightarrow \langle \text{identifier} \rangle \ \langle \text{digit} \rangle \\
\langle \text{nondigit} \rangle & \Rightarrow a \ | \ b \ | \ . \ . \ . \ | \ Y \ | \ Z \ | \ _ \\
\langle \text{digit} \rangle & \Rightarrow 0 \ | \ 1 \ | \ 2 \ | \ 3 \ | \ 4 \ | \ 5 \ | \ 6 \ | \ 7 \ | \ 8 \ | \ 9
\end{align*}
\]

Some identifiers:

\[
\begin{align*}
x \\
f \\
temp \\
templ \\
temp1 \\
done \\
_CanStartWithUnderscoreButNot7
\end{align*}
\]
The C Language: Expressions

Production rules:

\[ \langle \text{expression} \rangle \Rightarrow \langle \text{identifier} \rangle \]
\[ \Rightarrow \langle \text{constant} \rangle \]
\[ \Rightarrow \langle \text{cond-expression} \rangle \]
\[ \Rightarrow \langle \text{assign-expression} \rangle \]

\[ \langle \text{cond-expression} \rangle \Rightarrow \langle \text{expression} \rangle \ > \ <\text{expression}> \]
\[ \Rightarrow \langle \text{expression} \rangle \ !\ = \ <\text{expression}> \]

\[ \langle \text{assign-expression} \rangle \Rightarrow \langle \text{expression} \rangle \ = \ <\text{expression}> \]
\[ \Rightarrow \langle \text{expression} \rangle \ += \ <\text{expression}> \]

Some expressions:

- \( x \)
- \( x > 4 \)
- \( \text{done} \ !\ = \ 1 \)
- \( x = y = z = 0 \)
- \( x += 2.0 \)

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:

\[ \text{<expr>} \Rightarrow \text{<expr>} + \text{<expr>} \]
\[ \Rightarrow \text{<expr>} \ast \text{<expr>} \]
\[ \Rightarrow a \mid b \mid c \]

An ambiguous expression:
- \( a + b \ast c \)

Two different derivations (parse trees).
- \( (a + b) \ast c \)
- \( a + (b \ast c) \)

Postorder traversals of parse trees:
- \( b a + c \ast \)
- \( a c b \ast + \)
Ambiguity

Need more refined grammar:

\[
\begin{align*}
\langle \text{expr} \rangle & \quad \Rightarrow \quad \langle \text{expr} \rangle + \langle T \rangle \\
& \quad \Rightarrow \quad \langle T \rangle \\
\langle T \rangle & \quad \Rightarrow \quad \langle T \rangle \times \langle P \rangle \\
& \quad \Rightarrow \quad \langle P \rangle \\
\langle P \rangle & \quad \Rightarrow \quad ( \langle \text{expr} \rangle ) \\
& \quad \Rightarrow \quad a \mid b \mid c
\end{align*}
\]

No ambiguous expressions.

. \( a + b \times c \)
. \( (a + b) \times c \)
Type III Grammar (Regular)

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

\[
\begin{align*}
<A> & \Rightarrow <B> \ a \\
<A> & \Rightarrow <A> \ b \\
<B> & \Rightarrow c \\
<C> & \Rightarrow \varepsilon \\
\end{align*}
\]

Example:

\[
\begin{align*}
<A> & \Rightarrow <B> \ 0 \quad \text{Start} = <A> \\
<B> & \Rightarrow <A> \ 1 \\
<A> & \Rightarrow \varepsilon \\
\end{align*}
\]

Strings generated:

\[
\varepsilon, \ 10, \ 1010, \ 101010, \ 10101010, \ldots
\]

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.

\[ <A> \Rightarrow b <B> <C> a <C> \]
\[ <A> \Rightarrow <A> b c a <A> \]

Example:

\[ <\text{PAL}> \Rightarrow 0 <\text{PAL}> 0 \quad \text{Start} = <\text{PAL}> \]
\[ \Rightarrow 1 <\text{PAL}> 1 \]
\[ \Rightarrow 0 \]
\[ \Rightarrow 1 \]
\[ \Rightarrow \varepsilon \]

Strings generated:

\[ \varepsilon, 1, 0, 101, 001100, 111010010111, \ldots \]

Language generated:
Type II Grammar (Context Free)

Example:

\[<S> \Rightarrow ( <S> ) \quad \text{Start} = <S>\]
\[\Rightarrow \{ <S> \}\]
\[\Rightarrow [ <S> ]\]
\[\Rightarrow <S> <S>\]
\[\Rightarrow \varepsilon\]

Strings generated:

\[\varepsilon, ( ), ( ) [ ( )], ( ( [ ] \{ ( ) \}) \{ ] ( ( ) ) )], \ldots\]

Language generated:
Type I Grammar (Context Sensitive)

Add production rules of the type:

\[ [A] \ <B> \ [C] \Rightarrow [A] \ a \ [C] \]

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

\[ <A> \ <B> \ <C> \Rightarrow <A> \ b \ <C> \]
\[ <A> \ hi \ <B> \ <C> \ <D> \Rightarrow <A> \ hip \ <C> \ <D> \]
Type 0 Grammar (Recursive)

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

Example:

\[
\begin{align*}
\text{Start} & = <S> \\
<S> & \Rightarrow <S> <S> \quad <A><B> \Rightarrow <B><A> \\
<S> & \Rightarrow <A> <B> <C> \quad <B><A> \Rightarrow <A><B> \\
<A> & \Rightarrow a \quad <A><C> \Rightarrow <C><A> \\
<B> & \Rightarrow b \quad <C><A> \Rightarrow <A><C> \\
<C> & \Rightarrow c \quad <B><C> \Rightarrow <C><B> \\
<S> & \Rightarrow \varepsilon
\end{align*}
\]

Strings generated:

\[
\varepsilon, \ abc, \ aabbcc, \ cabcab, \ acacacacacacacacacacbbbbbb, \ ...
\]

Language generated:
Chomsky Hierarchy

<table>
<thead>
<tr>
<th>Type</th>
<th>Machine</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>III</td>
<td>FSA</td>
<td>regular</td>
</tr>
<tr>
<td>II</td>
<td>NPDA</td>
<td>context free</td>
</tr>
<tr>
<td>I</td>
<td>LBA</td>
<td>context sensitive</td>
</tr>
<tr>
<td>0</td>
<td>TM</td>
<td>recursive</td>
</tr>
</tbody>
</table>

Expressive languages

Powerful Machines

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

Noam Chomsky
Chomsky Hierarchy

- Regular
- Context Free
- Context sensitive
- Recursively enumerable
- All languages
**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:**

<table>
<thead>
<tr>
<th>FSA</th>
<th>Type III Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start state</td>
<td>Start symbol</td>
</tr>
<tr>
<td>States</td>
<td>Nonterminals</td>
</tr>
<tr>
<td>Transition arcs</td>
<td>Production rules: $&lt;A&gt; \Rightarrow &lt;B&gt; a$</td>
</tr>
<tr>
<td>Accept state</td>
<td>Production rules: $&lt;A&gt; \Rightarrow a$</td>
</tr>
</tbody>
</table>
Compilers and Grammar

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

- C ⇒ 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.

- \( C \Rightarrow \text{TOY} \)

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 \texttt{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:

\[
( a * ( b + c ) ) - ( d + e )
\]
Lindenmayer systems:
  - Apply production rules SIMULTANEOUSLY.
  - Falls in between Chomsky hierarchy levels.

Example:
  - Production rules:
    0 \Rightarrow 1 \ [ \ 0 \ ] \ 1 \ [ \ 0 \ ] \ 0
    1 \ [ \ \Rightarrow 1 \ 1 \ [
  - Start with 10. At stage i, apply rules to each symbol in string from stage i-1.

  10 \Rightarrow 1 \ [ \ 0 \ ] \ 1 \ [ \ 0 \ ] \ 0
  \Rightarrow 11 \ [1[0]1[0]0] \ 11 \ [1[0]1[0]0] \ 1[0]1[0]
  \Rightarrow 111 \ [*] \ 111 \ [*] \ *

* denotes copy of previous string
Other Exotic Forms of Grammar

Visualize in 2D:

“Production” rules:
add one to each segment of my trunk;
replace each branch with myself of prev generation

(alternate LR turns along trunk)

1: "stem"
0: "leaf"
[] branch off

1 1[*]1[*]* 11[*]11[*]* 111[*]111[*]*
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?