Topic 10 – Context-free Grammars

Key Ideas

- context-free grammars (CFG's)
- terminals and non-terminals
- production rules and derivations
- leftmost and rightmost derivations
- ambiguous grammars
- implementing associativity and precedence

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 3.1 to 3.4.

What is Next?

What is Missing

- We now have the ability to recognize all the tokens in our programming language.
- Analogy: we can *recognize the words* (i.e. tokens), now we need to
    - *recognize the sentences* (we'll call this step *parsing*)
    - *recognize the meaning of sentences* (we'll do this later on)

What is Next?

Recall: Basic Compilation Steps

The steps in translating a program from a high level language to an assembly language program are:

WLP4 text file
↓ 1. *scanning*: identify the tokens *Done*
WLP4 tokens
↓ 2. *syntactic analysis*: parse *Next*
parse tree
↓ 3. *semantic analysis*: create a symbol table and perform type checking *Later*
⋮
↓ 4. *code generation* *Later*
MIPS Assembly Language

What is Next?

Recall: Staging

- different stages check for different types of errors
- can improve error messages
- simplifies compiler code (more modular)
- *Syntax:* structure / format of the sequence of tokens
  - Valid C++: a += b;
  - Not valid C++: + a =; b
- *Semantics:* meaning
  - Does that function have the right number of arguments?
  - Does that function have the right type of arguments?
  - What is that variable's type?

Motivation for CFG's

Current Challenge

- Check if the syntax of a program is correct.
- *Key Problem: we need something more powerful than regular expressions / DFAs / NFAs.*
- I.e. given $\Sigma = \{a, b\}$, it must have the ability to recognize the language $\mathcal{L} = \{w$: number of a's in w = the number of b's in w$\}$.
- E.g. in programming you must be able to recognize

balanced parentheses

(( ) (( )))

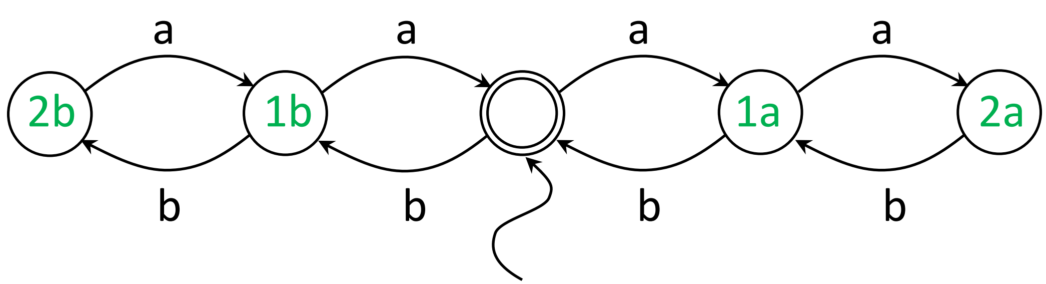
balanced braces

```
{
   {
      {
      }
   }
}
```

Motivation for CFG's

Current Challenge

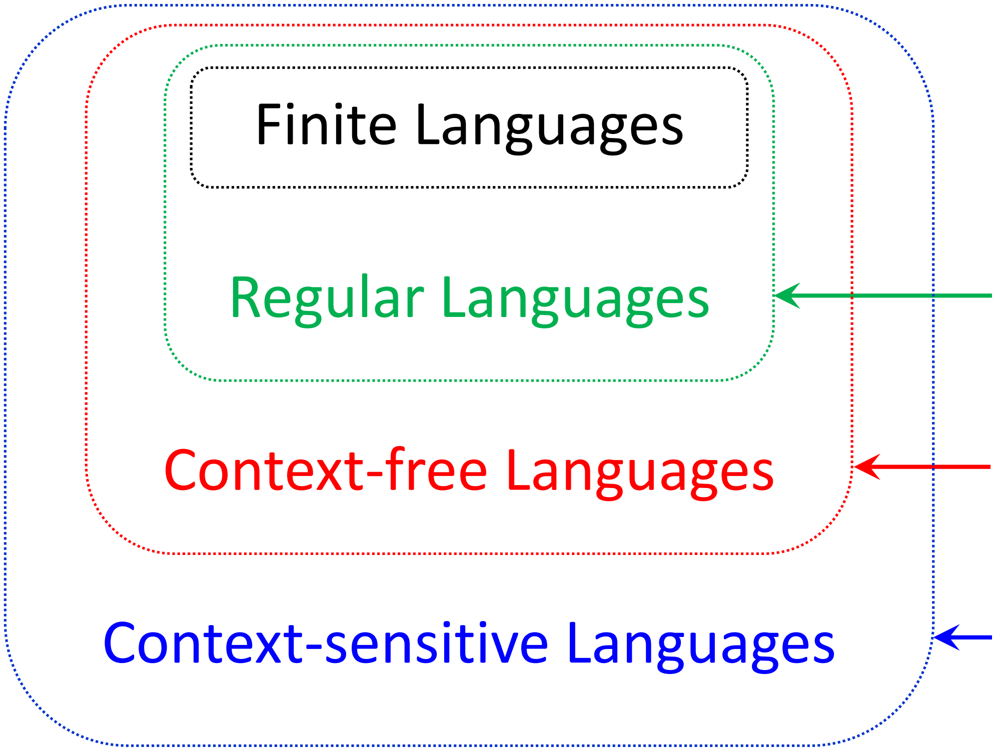
- Create a DFA that recognizes the language $\mathcal{L}$ = {*w*: number of *a*'s in *w* = the number of *b*'s in w} over alphabet Σ ={*a*, *b*}.
- Easy if the maximum number of a's and b's is fixed, say 2.



- *Impossible if the number of a's and b's is arbitrary.*
- DFAs are good for tracking a finite number of things, e.g. strings with 3 b's in a row.
- The number of nested parentheses is unbounded.
- We need an unbounded stack to track if the number of left and right parentheses are equal.

The Compiler

Recall: Chomsky Hierarchy



Steps in Compiling

1. lexical analysis: find each lexical element
2. syntactic analysis recognize with 1 stack
3. check the semantics
4. code generation

Example – Simple Sentence

Example

English has rules that guide the format of a sentence

(R1)	<sentence>	⇒	<subj phrase> <verb phrase>
(R2)	<subj phrase>	⇒	<article> <noun>
(R3)	<verb phrase>	⇒	<verb>
(R4)	<article>	⇒	the
(R5)	<noun>	⇒	dog
(R6)	<verb>	⇒	barks

The rules have two types of components

1. *terminal / token*: visible in the output e.g. the, dog, barks
2. *non-terminal / variable*: abstract component
   - does not literally appear in the output
   - notation: angle brackets < ... > or capital letters, e.g. <sentence>, <subj phrase>, <verb phrase>

Specification Components

Production Rules

- possible expansion of a non-terminal into zero or more terminals and/or non-terminals
- more than one rule per non-terminal is possible

Derivation of the sentence "The dog barks."

<sentence>

⇒ <subj phrase> <verb phrase> (R1)

⇒ <article> <noun> <verb phrase> (R2)

⇒ <article> <noun> <verb > (R3)

⇒ the <noun> <verb > (R4)

⇒ the dog <verb> (R5)

⇒ the dog barks (R6)

Derivation

How to Derive a String

- i.e. how to recognize if a string is part of the language
- apply the production rules to generate a valid string
  - beginning with *start symbol*
- repeatedly replace one *non-terminal* using one rule
- continue until there are no more *non-terminals*
- resulting sequence of *terminals* is a *syntactically correct* string

Formal Definition

- *language of a CFG*: set of all valid strings (sequences of characters) that can be derived from the *start symbol*

Example CFG

Typical CS241 Example

G:	(R1)	S	$\rightarrow$	*aSb*	// "*aSb*" is *concatenation*
	(R2)	S	$\rightarrow$	D	// 2 rules with S on LHS is *union*
	(R3)	D	$\rightarrow$	*c*D	// D on both sides is *recursion*
	(R4)	D	$\rightarrow$	ε	

- the word *accb* is in the language generated by the grammar G, i.e. L(G), since we can *derive accb* from G.
- derivation:

$$\mathrm{S} \underset{\mathrm{R1}}{\Rightarrow} a\mathrm{S}b \underset{\mathrm{R2}}{\Rightarrow} a\mathrm{D}b \underset{\mathrm{R3}}{\Rightarrow} ac\mathrm{D}b \underset{\mathrm{R3}}{\Rightarrow} acc\mathrm{D}b \underset{\mathrm{R4}}{\Rightarrow} accb$$

General Approach

Differences compared to Regular Languages

- *Context-free languages* are built from:
  - finite sets
  - concatenation
  - union
  - recursion

  (finite sets, concatenation, union:) *same* as Regular Languages

  (recursion:) *new* replaces repetition
- Recognizers for Regular Languages use a finite amount of memory
- Recognizers for context-free languages use a finite amount of memory plus one (unbounded) stack

Example CFG

Unpacking the Example

- *G* is a context-free grammar
- *L(G)* is the language (set of words) specified by G
- *a word*: a sequence of terminals (or tokens) that can be derived by the CFG
- *a derivation*: a sequence of rewriting steps from the start symbol until there are no more non-terminals.
- *Production Rules (a.k.a. Rewrite Rules)* capture
  - union
  - concatenation
  - recursion (which is strictly more powerful than repetition)

Example CFG

Unpacking the Example

- *N* is a finite set of non-terminals
  - they *may not appear* at the end of the derivation
- *T* is a finite set of terminals
  - they *may appear* at the end of the derivation
- *P* is a finite set of production rules in the form $\alpha \to \beta$ where
  - α is a not-terminal, i.e. $\alpha \in N$
  - β is a repetition of terminals and non-terminals, i.e. $\beta \in (N \cup T)^*$
- *S* is the start symbol, $S \in N$
  - by convention it is on the LHS of the first rule.

Example – Simple Sentence

Unpacking the Example

- N = {S, D}, i.e. the non-terminals
- T = {a, b, c} i.e. the terminals
- P = set of production rules in the form $\alpha \rightarrow \beta$, i.e.

$$
\begin{aligned}
S &\rightarrow a\,S\,b \\
S &\rightarrow D \\
D &\rightarrow cD \\
D &\rightarrow \varepsilon
\end{aligned}
$$

  - rules all have elements of N on the LHS and elements of (N U T)* on the RHS
- S is the start symbol, $S \in N$ and by convention it on the LHS of the first rule.

Example CFG

More Examples

G: (R1) $S \rightarrow a\, S\, b$

(R2) $S \rightarrow D$

(R3) $D \rightarrow cD$

(R4) $D \rightarrow \varepsilon$

- derive: aaabbb

- derive: ccc

Another Example CFG

Balanced Parentheses

- Task: Create a CFG that access accepts words with balanced parentheses
- Example words: ε, (), (()), ()(), (() ()), ...

(R1) S $\Rightarrow$

(R2) $\Rightarrow$

(R3) $\Rightarrow$

Another Example CFG

Balanced Parentheses

- Derive (( )):

- Derive (( ) ( )):