CSC 448: Compilers

Lecture 4

Joseph Phillips
De Paul University

2015 April 22

Reading

- Charles Fischer, Ron Cytron, Richard LeBlanc Jr. “Crafting a Compiler” Addison-Wesley. 2010.
  - Chapter 4: Grammars and Parsing

Topics:

- Introduction
- Grammars and Parsing

Context-Free Grammars

- A language G is formally defined to have:
  - Terminal language, Σ:
    - Lexemes produced by scanner
    - Signified by lowercase and punctuation
    - {`a`, `b`, `c`, `if`, `then`, `(`, `;`}.
    - Also includes `$`, the end-of-input symbol.

  - Non-Terminal language, *N*:
    - The “variables” of the language: can stand for multiple value
    - Signified by uppercase
    - {`A`, `B`, `C`, `Prefix`}

Context-Free Grammars

- A language G is formally defined to have:
  - Start symbol, S.
    - In *N*.
    - Originates all derivations
  - Production rules:
    - Have form A -> $X_1 .. X_m$, where:
    - A is a symbol in *N*.
    - X_i is in either *N* or Σ.

Derivations:

- If $A \rightarrow \gamma$ is a production, then
  - $\alpha A\beta \Rightarrow \alpha\gamma\beta$ is one step of a derivation
  - $\alpha A\beta \Rightarrow^{+} \alpha\gamma\beta$ is one or more steps
  - $\alpha A\beta \Rightarrow^{*} \alpha\gamma\beta$ is zero or more steps

- SF(G): the sentential forms of the context-free grammar G:
  - $SF(G) = \{S \Rightarrow^{*} \beta\}$
  - $L(G) = \{w \in \Sigma^{*} | \Sigma \Rightarrow^{+} w\}$

Example:

- G = [Σ={a,b},*N*={S,A,B},S,{S->aA; A->bB; B->λ}]

- Therefore,
  - SF(G) = {S, aA, abB, ab}
  - L(G) = {ab}

Leftmost Derivation $=>_{lm}$

- Example G=[{f,v,+}, {E,P,T},E,rules=
  - E -> P (E)
  - E -> v T
  - P -> f
  - P -> λ
  - T -> + E
  - T -> λ
-]
- Expand the *left*most non-terminal:
  - E $=>_{lm}$ **P**(E)
  - $=>_{lm}$ f(**E**)
  - $=>_{lm}$ f(v **T**)
  - $=>_{lm}$ f(v + **E**)
  - $=>_{lm}$ f(v + v **T**)
  - $=>_{lm}$ f(v + v)

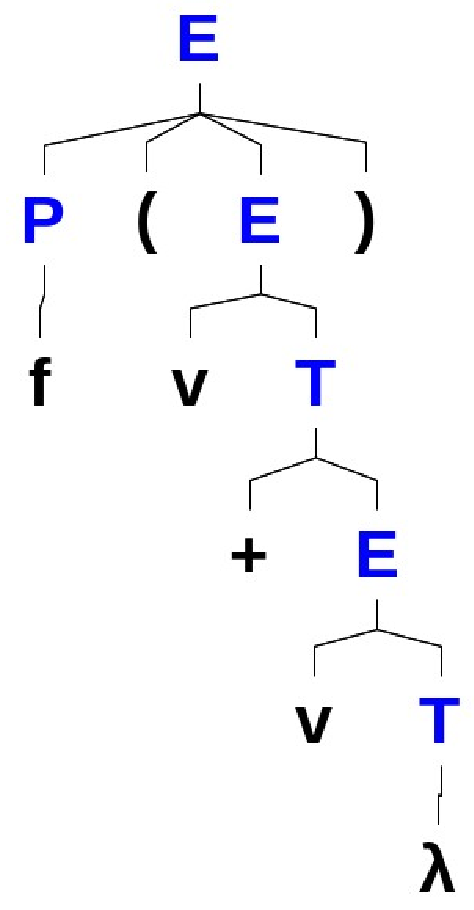
Rightmost Derivation $\Rightarrow_{rm}$

- Example G=[{f,v,+},{E,P,T},E,rules=
  - E -> P (E)
  - E -> v T
  - P -> f
  - P -> λ
  - T -> + E
  - T -> λ
-]

- Expand the *right*most non-terminal:
  - E $=>_{rm}$ P(**E**)
  - $=>_{rm}$ P(v **T**)
  - $=>_{rm}$ P(v + **E**)
  - $=>_{rm}$ P(v + v **T**)
  - $=>_{rm}$ **P**(v + v)
  - $=>_{rm}$ f(v + v)

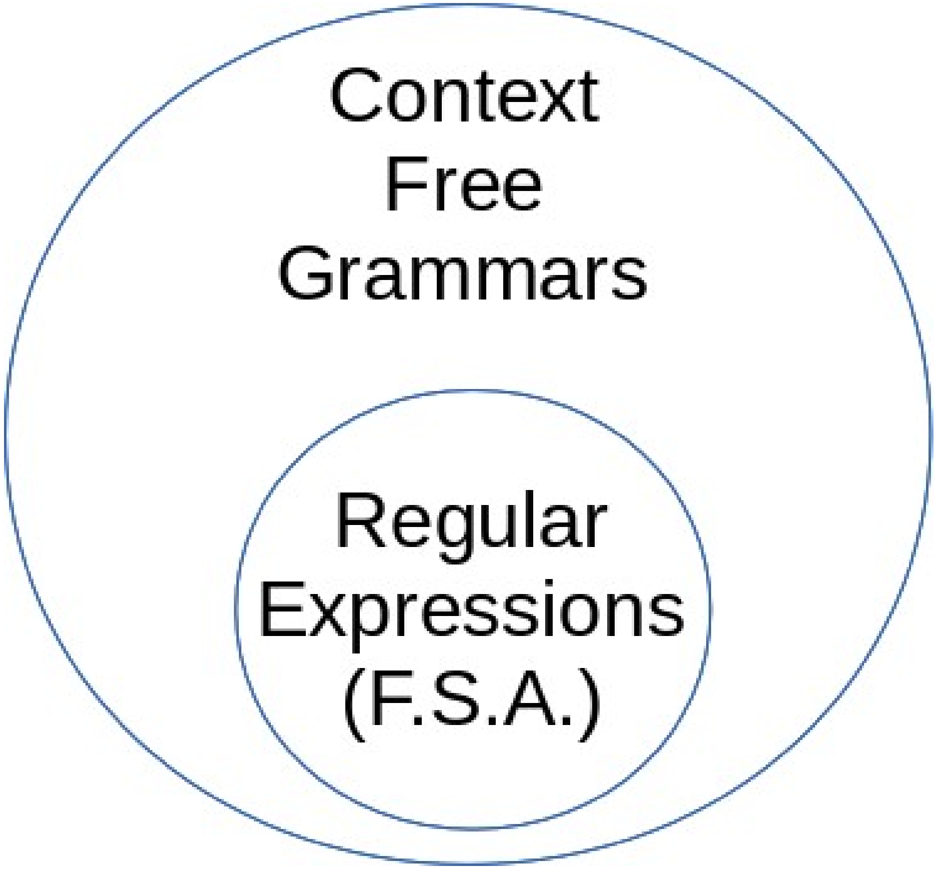
Parse Trees:

- Graphically outline which productions were used and how non-terminals map to other non-terminals and terminals



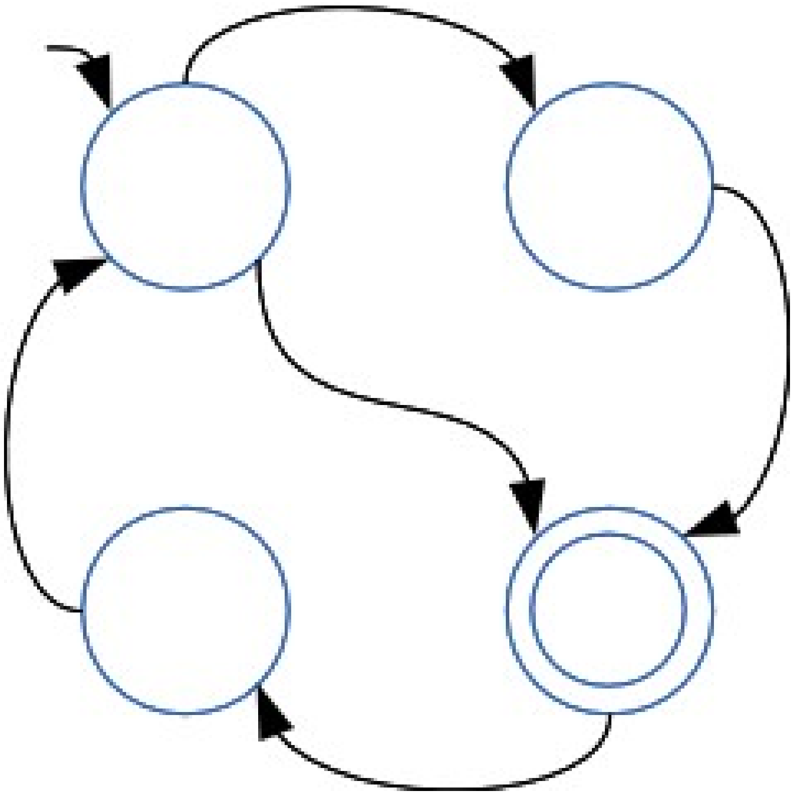
CFGs vs. Regular Expressions

- Cpntext-Free Grammars are more expressive than Regular Expressions
- Language that is CFG but not Reg Expr:
  - {wcw^R|w in (a|b)*}
  - Generable with grammar rules: S -> aSa; S -> bSb; S -> c



CFGs vs. Regular Expressions

- Cpntext-Free Grammars are more expressive than Regular Expressions
- Regular Expressions are recognized by Finite State Automata (FSA)

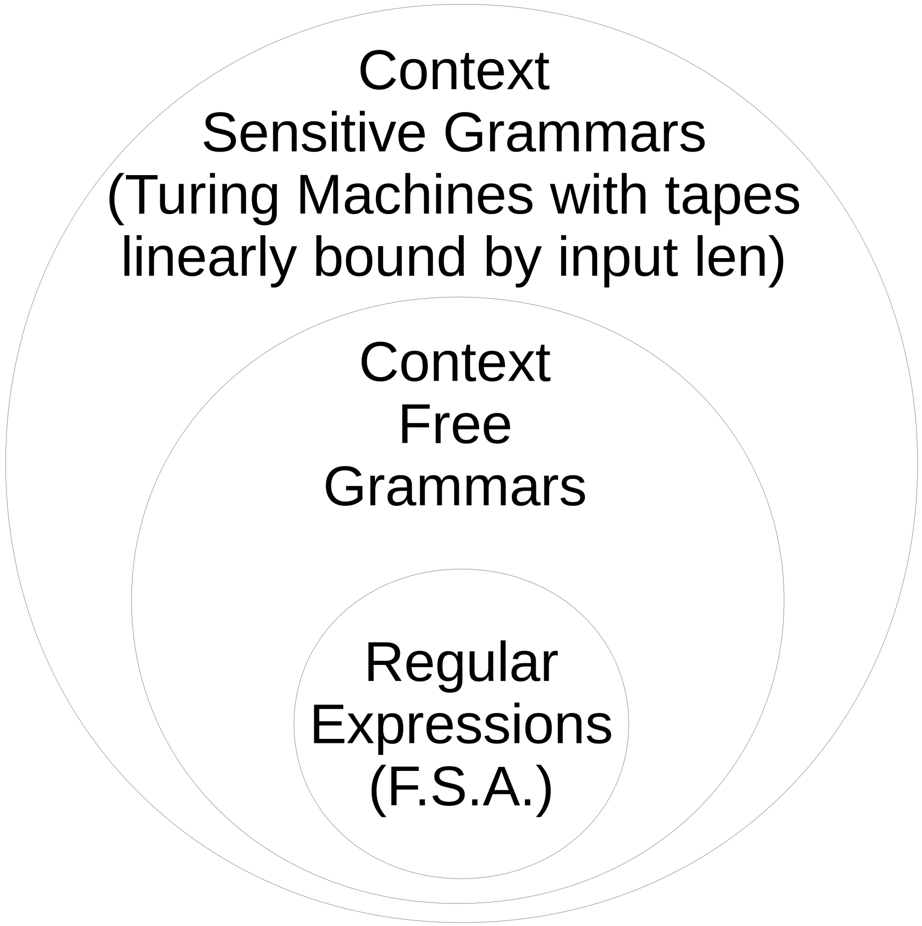


CFGs vs. Regular Expressions

- Regular Expressions are recognized by CFGs with productions restricted to either:
  - A -> a B, or
  - C -> d
  - Example: $(ab)^*$`bab`:
    - `S -> a b S`
    - `S -> b a b`
  - or:
    - `S -> a T`
    - `S -> b U`
    - `T -> b S`
    - `U -> a V`
    - `V -> b`

Context Free Grammars vs. Context-Sensitive Grammars

- Context-***Free*** Grammars are less expressive than Context-***Sensitive*** Grammars
- Language that is CSG but not CFG:
  - $\{a^N b^N c^N : N \geq 1\}$

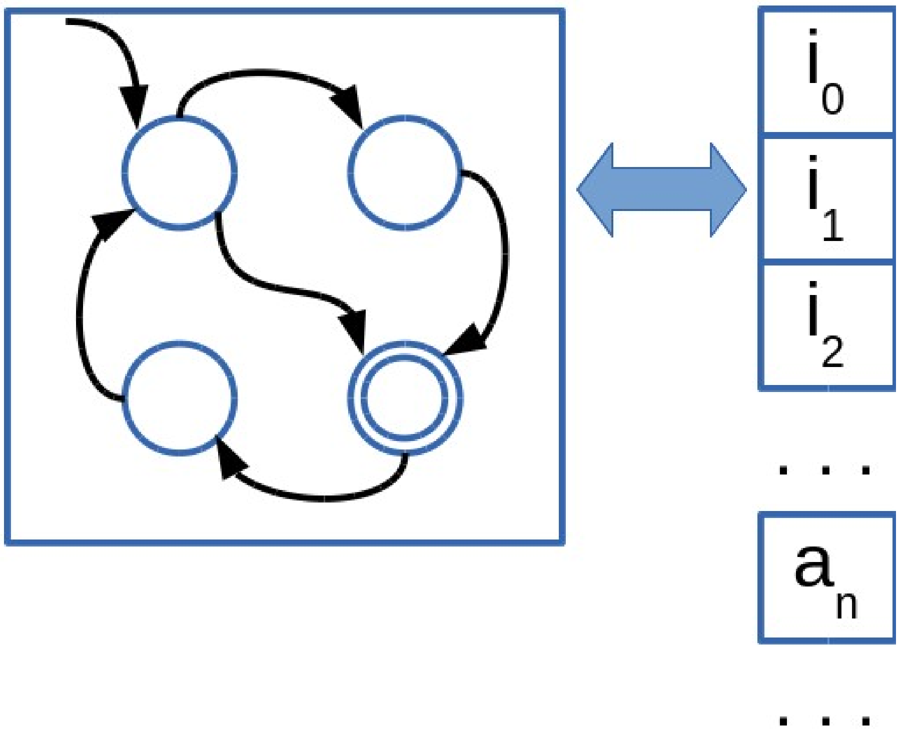


Context Free Grammars vs. Context-Sensitive Grammars

- Context-***Free*** Grammars are less expressive than Context-***Sensitive*** Grammars
- Context-Sensitive Grammars are:
  - Recognized by Turing Machines whose tape's length is O(inputLen)
  - Has productions of form αAβ -> αγβ where:
    - α γ β can each have terminals and non-terminals
    - More general than CFG's productions A -> γ because production only applicable in "context" of "α~β"

What's a Turing Machine?

- Turing machines have:
  - An input tape, divided into cells, which has an end on one side but is infinite on the other
  - Initially the n-leftmost cells hold the input
- Operation:
  - Depending on state (Q) and tape symbol (Γ):
    - Change state
    - Overwrite current symbol
    - Move tape head Left or Right



What's a Turing Machine?

- More formally:
- M = (Q,Γ,*B*,Σ,δ,q_0,F)
  - Q is finite set of states
  - Γ allowed tape symbol
  - *B* a symbol in Γ (the blank)
  - Σ input symbols, a subset of Γ (not including *B*)
  - δ next move function: (QxΓ) -> (QxΓx{L,R})
  - q_0 is start state
  - F set of final states (in Q)
- Operation:
  - Depending on state (Q) and tape symbol (Γ):
    - Change state
    - Overwrite current symbol
    - Move tape head Left or Right

Example Turing Machine

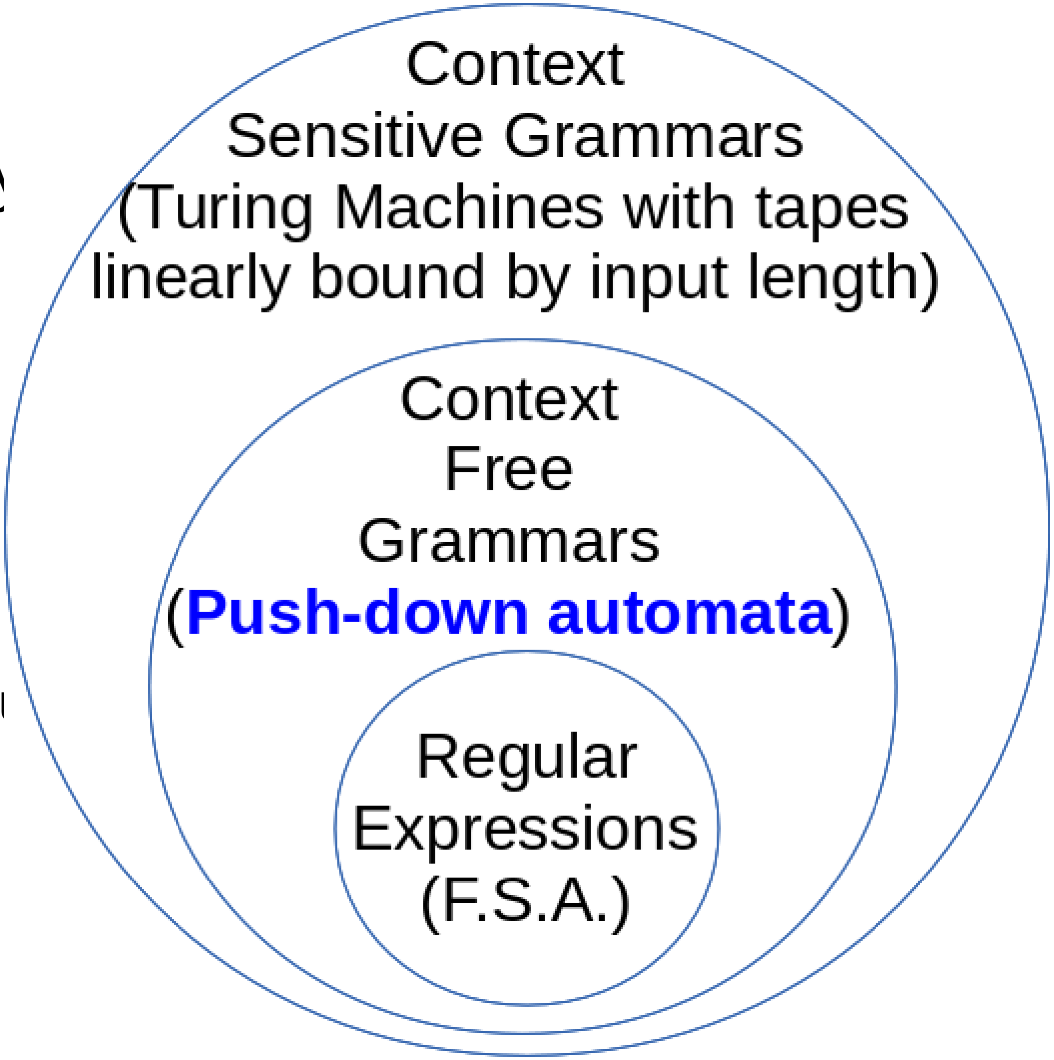
Regular Expressions have Finite State Automata,

Context *Sensitive* Grammars have finite tape-length Turing Machines,

Context *Free* Grammars have . . .

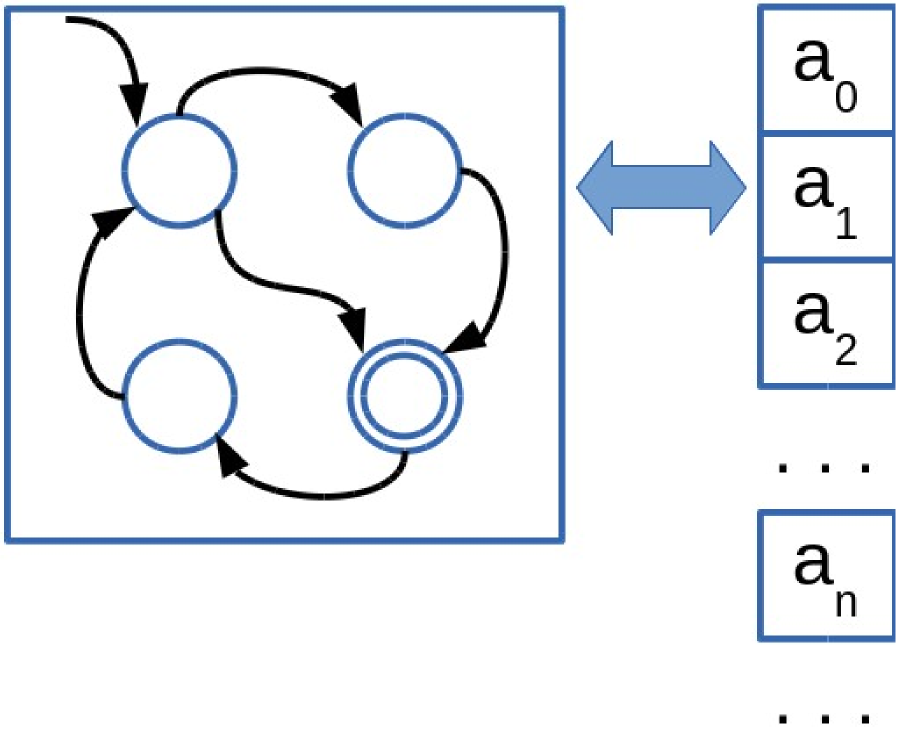
Push-Down Automata!

- Not an arbitrary tape, but a stack
- Stops and accepts when stack empt
- Pop stack symbol, then push either 0,1 or 2 symbols
- Similar, but less general to Turing Machine
  - Turing Machine can go left/right, PDA only has push pop
  - Turing Machine has input on tape, outp to tape. PDA only accepts with empty stack



Push-Down Automata!

- More formally:
- M = (Q,Σ,Γ,q0,Z_0,F,δ)
  - Q is finite set of states
  - Σ finite input alphabet
  - Γ finite stack alphabet
  - q0 is start state (in Q)
  - Z_0 is start symbol in Γ
  - F final states (in Q)
  - δ next move function: (Qx(Σ∪{λ})xΓ) -> (QxΓ)



PDA Example (1)

- A machine that recognizes palindromes (with separating character ‘c’)
  - $\{wcw^R | w \text{ in } (a|b)^*\}$
  - Generatable with grammar rules S -> aSa; S -> bSb; S -> c
- Idea
  - Before we read **c**: (state = q_1)
    - When read **a** from input, push **a'** on stack
    - When read **b** from input, push **b'** on stack
  - Reading **c** means its time to pop (change state q_1 -> q_2)
  - After we read **c**: (state = q_2)
    - When read **a** from input, make sure pop **a'** from stack
    - When read **b** from input, make sure pop **b'** from stack

PDA Example (2)

- $M = (\{q_1,q_2\}, \{a,b,c\}, \{a',b',z'\},\delta,q_1,R,\emptyset\})$
  - $\delta(q_1,a,z') = \{(q_1,a'z')\}$ $\delta(q_1,b,z') = \{(q_1,b'z')\}$
  - $\delta(q_1,a,a') = \{(q_1,a'a')\}$ $\delta(q_1,b,a') = \{(q_1,b'a')\}$
  - $\delta(q_1,a,b') = \{(q_1,a'b')\}$ $\delta(q_1,b,b') = \{(q_1,b'b')\}$
  - $\delta(q_1,c,z') = \{(q_2,z')\}$
  - $\delta(q_1,c,a') = \{(q_2,a')\}$
  - $\delta(q_1,c,b') = \{(q_2,b')\}$
  - $\delta(q_2,a,a') = \{(q_2,\lambda)\}$ $\delta(q_2,b,b') = \{(q_2,\lambda)\}$
  - $\delta(q_2,\lambda,z') = \{(q_2,\lambda)\}$

PDA Example (3)

- Initially:
  - state = $\mathbf{q_1}$
  - input = **a**bcba
  - stack = [**z'**]
- Operation:
  - $\mathbf{q_1}$ **a**bcba [**z'**]
    - $\delta(\mathbf{q_1,a,z'}) = \{(q_1,a'z')\}$ $\therefore$ q_1 [a'z']
  - $\mathbf{q_1}$ a**b**cba [**a'**z']
    - $\delta(\mathbf{q_1,b,a'}) = \{(q_1,b'a')\}$ $\therefore$ q_1 [b'a'z']
  - $\mathbf{q_1}$ ab**c**ba [**b'**a'z']
    - $\delta(\mathbf{q_1,c,b'}) = \{(q_2,b')\}$ $\therefore$ q_2 [b'a'z']

- Operation, cont'd:
  - $\mathbf{q_2}$ abc**b**a [**b'**a'z']
    - $\delta(\mathbf{q_2,b,b'}) = \{(q_2,\lambda)\}$ $\therefore$ q_2 [a'z']
  - $\mathbf{q_2}$ abcb**a** [**a'**z']
    - $\delta(\mathbf{q_2,a,a'}) = \{(q_2,\lambda)\}$ $\therefore$ q_2 [z']
  - $\mathbf{q_2}$ abcba [**z'**]
    - $\delta(\mathbf{q_2,\lambda,z'}) = \{(q_2,\lambda)\}$
  - $\mathbf{q_2}$ abcba [ ]
    - ***Stop!*** ***Accept!***

Real-world PDAs

```
/* Just a question: */
int main ()
{
  printf("Have you seen any computer"
         " language like wcwR?\n"
        );
  return(EXIT_SUCCESS);
}
```

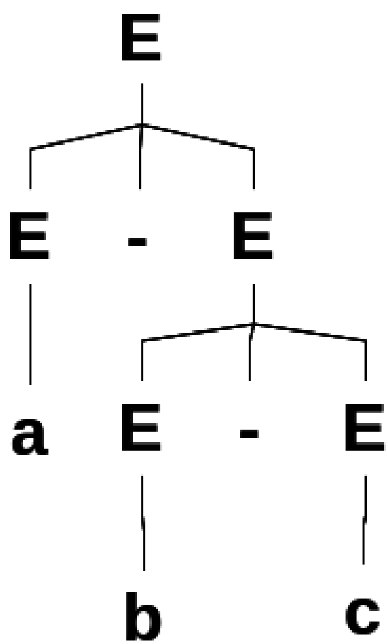
Properties of CFGs (1)

- Reduced Grammars:
  - Example:
    - **`S -> A`**
    - **`S -> B`**
    - **`A -> a`**
    - **`B -> B b`**
    - **`C -> c`**
  - Idea:
    - Remove non-terminals not reachable from **`S`** (*e.g*. **`C`**)
    - Remove non-terminals that never terminate (*e.g*. **`B`**)

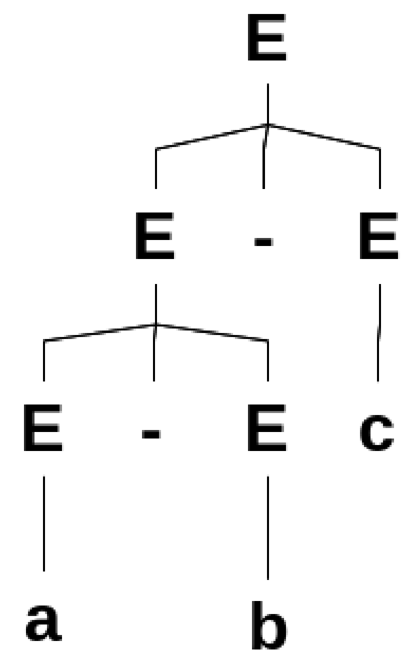
Properties of CFGs (2a)

- Ambiguity
  - When two or more parse trees exist
  - Example:
    - E -> E - E
    - E -> id
  - Common in natural languages like English:
    - “I saw the man with the telescope”

Properties of CFGs (2b)



a – (b – c)
= a – b + c



(a – b) – c
= a – b – c

Backus-Naur Form (BNF)

- Extends the way to define CFGs with:
  - Optional symbols, use square brackets:
    - A -> α [X_1,...X_m] β
  - Repeated symbols, use curly braces:
    - B -> γ {X_1,...X_n} δ
- How the grammar of computer languages is generally defined

The syntax of C in Backus-Naur Form

```
<translation-unit> ::= {<external-declaration>}*

<external-declaration> ::= <function-definition>
                         | <declaration>

<function-definition> ::= {<declaration-specifier>}* <declarator> {<declaration>}*
<compound-statement>

<declaration-specifier> ::= <storage-class-specifier>
                          | <type-specifier>
                          | <type-qualifier>

<storage-class-specifier> ::= auto
                            | register
                            | static
                            | extern
                            | typedef
```

The syntax of C in Backus-Naur Form

```
<type-specifier> ::= void
                   | char
                   | short
                   | int
                   | long
                   | float
                   | double
                   | signed
                   | unsigned
                   | <struct-or-union-specifier>
                   | <enum-specifier>
                   | <typedef-name>

<struct-or-union-specifier> ::= <struct-or-union> <identifier> { {<struct-declaration>}+ }
                              | <struct-or-union> { {<struct-declaration>}+ }
                              | <struct-or-union> <identifier>

<struct-or-union> ::= struct
                    | union
```

The syntax of C in Backus-Naur Form

```
<struct-declaration> ::= {<specifier-qualifier>}* <struct-declarator-list>

<specifier-qualifier> ::= <type-specifier>
                        | <type-qualifier>

<struct-declarator-list> ::= <struct-declarator>
                           | <struct-declarator-list> , <struct-declarator>

<struct-declarator> ::= <declarator>
                      | <declarator> : <constant-expression>
                      | : <constant-expression>

<declarator> ::= {<pointer>}? <direct-declarator>

<pointer> ::= * {<type-qualifier>}* {<pointer>}?

<type-qualifier> ::= const
                   | volatile

<direct-declarator> ::= <identifier>
                      | ( <declarator> )
                      | <direct-declarator> [ {<constant-expression>}? ]
                      | <direct-declarator> ( <parameter-type-list> )
                      | <direct-declarator> ( {<identifier>}* )
```

(It continues, just search for it online)

Parsers and Recognizers

- Top-down parsing
  - Start with S, expand tree downward
- LL
  - ***L***L: left-to-right input
  - L***L***: ***Left***most parse produced

- $E \Rightarrow_{lm} \mathbf{P}(E)$
- $\Rightarrow_{lm} f(\mathbf{E})$
- $\Rightarrow_{lm} f(v\ \mathbf{T})$
- $\Rightarrow_{lm} f(v + \mathbf{E})$
- $\Rightarrow_{lm} f(v + v\ \mathbf{T})$
- $\Rightarrow_{lm} f(v + v)$

Parsers and Recognizers

- Bottom-up parsing
  - Start with lexemes, build tree upward
- LR
  - ***L***R: left-to-right input
  - L***R***: ***Right***most parse produced

- $\Rightarrow_{rm}$ `f(v + v)`
- $\Rightarrow_{rm}$ `**P**(v + v)`
- $\Rightarrow_{rm}$ `P(v + v **T**)`
- $\Rightarrow_{rm}$ `P(v + **E**)`
- $\Rightarrow_{rm}$ `P(v **T**)`
- `E` $\Rightarrow_{rm}$ `P(**E**)`

Intro to building Parsers

- Idea:
  - Parser sees lexemes (“terminals”)
  - How does it know which productions to apply?
- It would be nice to know:
  - Can this non-terminal generate λ?
  - (For top-down): I just read this terminal, which productions can it start?
  - (For bottom-up): I just read this terminal, which productions can it end?

Can this non-terminal generate λ?

```
fnc derivesEmpty()
  foreach A in NonTerms() do
    symDerivEmp(A) := f
  foreach p in Productions() do
    ruleDerivEmp(p) := f
    count(p) := p.rhs.length()
    checkForEmpty(p)
  foreach X in toDoList
    toDoList -= {X}
    foreach p in productions with X in rhs
      count(p)--;
      checkForEmp(p)
```

Can this non-terminal generate λ?

```
fnc checkForEmpty(p)
  if (count(p) == 0)
    ruleDerivEmp(p) := t
    var := p.lhs
    if ( not symDerivEmp(var) )
      symDerivEmp(var) := t
      toDoList += { var }
```

Can this non-terminal generate λ?

- Example grammar:

  r1: S -> a
  r2: S -> Bb
  r3: B -> C
  r4: C -> λ

Can this non-terminal generate λ?

```
fnc derivesEmpty()
  foreach A in NonTerms() do
    symDerivEmp(A) := false
  foreach p in Productions() do
    ruleDerivEmp(p) := false
    count(p) := p.rhs.len()
    checkForEmpty(p)
  foreach X in toDoList
    toDoList -= {X}
    foreach p in productions with X in rhs
      count(p)--;
      checkForEmp(p)
```

Can this non-terminal generate λ?

r1: S -> a
r2: S -> Bb
r3: B -> C
r4: C -> λ

Prod or non-term	derives empty?	count
S	false	--
B	false	--
C	false	--
r1	false	1
r2	false	2
r3	false	1
r4	false	0

Can this non-terminal generate λ?

```
fnc derivesEmpty()
  foreach A in NonTerms() do
    symDerivEmp(A) := false
  foreach p in Productions() do
    ruleDerivEmp(p) := false
    count(p) := p.rhs.len()
    checkForEmpty(p)
  foreach X in toDoList
    toDoList -= {X}
    foreach p in productions with X in rhs
      count(p)--;
      checkForEmp(p)
```

Can this non-terminal generate λ?

- checkForEmpty(r1)
  - count(r1) > 0, so nothing to do
- `checkForEmpty(r2)`
  - count(r2) > 0, so nothing to do
- `checkForEmpty(r3)`
  - count(r3) > 0, so nothing to do
- `checkForEmpty(r4)`
  - count(r4) == 0, so:
  - derivesEmpty(r4) := true
  - var := C
  - derivesEmpty(C) := true
  - toDoList := {C}

Prod or non-term	derives empty?	count
S	false	--
B	false	--
C	true	--
r1	false	1
r2	false	2
r3	false	1
r4	true	0

Can this non-terminal generate λ?

```
fnc derivesEmpty()
  foreach A in NonTerms() do
    symDerivEmp(A) := false
  foreach p in Productions() do
    ruleDerivEmp(p) := false
    count(p) := p.rhs.len()
    checkForEmpty(p)
  foreach X in toDoList
    toDoList -= {X}
    foreach p in productions with X in rhs
      count(p)--;
      checkForEmp(p)
```

Can this non-terminal generate λ?

- X = C, toDoList = [ ]
- foreach production p with C on RHS
  - p := r3
  - count(r3) := count(r3) -1 = 0
  - checkForEmpty(r3)
    - count(r3) == 0, so:
    - derivesEmpty(r3) := true
    - var := B
    - derivesEmpty(B) := true
    - toDoList := {B}

Prod or non-term	derives empty?	count
S	**false**	**--**
B	**true**	**--**
C	**true**	**--**
r1	**false**	**1**
r2	**false**	**2**
r3	**true**	**0**
r4	**true**	**0**

Can this non-terminal generate λ?

- X = B, toDoList = [ ]
- foreach production p with B on RHS
  - p := r2
  - count(r2) := count(r2) -1 = 1
  - checkForEmpty(r2)
    - count(r2) > 0
    - so nothing to do

Prod or non-term	derives empty?	count
S	**false**	**--**
B	**true**	**--**
C	**true**	**--**
r1	**false**	**1**
r2	**false**	**1**
r3	**true**	**0**
r4	**true**	**0**

Can this non-terminal generate λ?

The final result:

Prod or non-term	derives empty?	count
S	**false**	**--**
B	**true**	**--**
C	**true**	**--**
r1	**false**	**1**
r2	**false**	**1**
r3	**true**	**0**
r4	**true**	**0**

References:

- Hopcroft, John; Ullman, Jeffery. “*Introduction to Automata Theory, Languages and Computation*.” Addison-Wesley. 1979.