Syntactic Analysis

Alternative Parsing Algorithms
Classification of Grammars
Beyond Syntax

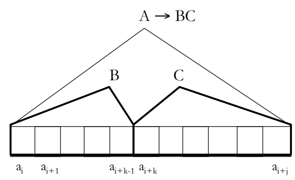
Alternative Parsing Algorithms

- LL and LR algorithms
  - Low Space and Time Complexity
  - Very Practical, i.e. there are Tools for their construction
  - In same cases, however, cannot cope with CFLs…
- More Generic Parsing Alternatives
  - More Lookahead…
  - Higher Complexity (space and Time)
- Today:
  - Cocke-Young-Kasami (CYK)
  - Early

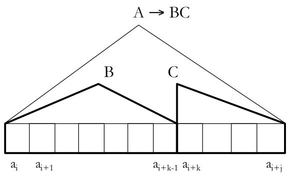
Cooke-Young-Kasami (CYK)

- Dynamic Programming Technique
  - Uses a Triangular Table, $O(n^2)$ for n input tokens
  - Requires Grammar to be in some specific form (*not a big deal*):
    - No ε-productions
    - Chomsky-Normal-Form (CNF): at most two (2) symbols per production
  - Handles Ambiguity very well…
- Recurrence Relation
  - The Production A → BC can derive the input string $s_{i,j}$ (input starting at index i with length j) if there exists a k, s.t., B can derive $s_{i,k}$ and C can derive $s_{i+k,j-k}$ with $1 \leq k < j$
  - Input string is in L(G) iff S can derive $s_{1,n}$

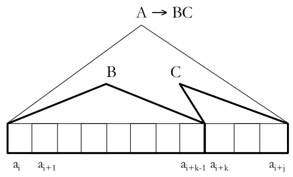
Cooke-Young-Kasami (CYK)



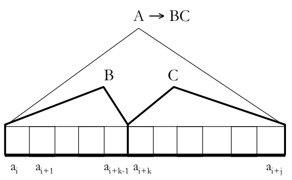
Cooke-Young-Kasami (CYK)



Cooke-Young-Kasami (CYK)



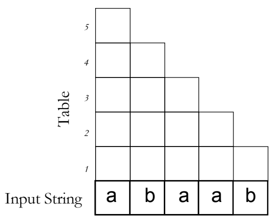
Cooke-Young-Kasami (CYK)



- **Algorithm:**
  - Start with initialization
    - Fill table T with $t_{i1} = \{A \mid A \rightarrow a_i \in P\}$ for each i.
  - For each "level" of the table
    - $t_{ij} = \{ A \mid$ for some k, $1 \leq k < j$, $A \rightarrow BC \in P$, s.t. $B \in t_{ik}$ and $C \in t_{i+k,j-k}\}$
  - Accept Input iff $S \in t_{1n}$

CYK Example

G = {S,{A,S},{a,b}, P}
P:
- S → AA
- S → AS
- S → b
- A → SA
- A → AS
- A → a



- **Algorithm:**
  - Start with initialization
    - Fill table T with $t_{i1} = \{A \mid A \rightarrow a_i \in P\}$ for each i.
  - For each "level" of the table
    - $t_{ij} = \{ A \mid$ for some k, $1 \leq k < j$, $A \rightarrow BC \in P$, s.t. $B \in t_{ik}$ and $C \in t_{i+k,j-k}\}$
  - Accept Input iff $S \in t_{1n}$

CYK Example

G = {S,{A,S},{a,b}, P}
P:
- S → AA
- S → AS
- S → b
- A → SA
- A → AS
- A → a

Table

5					
4					
3					
2					
1	A	S	A	A	S
Input String	a	b	a	a	b

- **Algorithm:**
  - Start with initialization
    - Fill table T with $t_{i,1}$ = {A | A → a_i ∈ P} for each i.
  - For each "level" of the table
    - t_{ij} = { A | for some k, 1 ≤ k < j, A → BC ∈ P, s.t. B ∈ t_{ik} and C ∈ $t_{i+k,j-k}$}
  - Accept Input iff S ∈ t_{1n}

CYK Example

G = {S,{A,S},{a,b}, P}
P:
- S → AA
- S → AS
- S → b
- A → SA
- A → AS
- A → a

Table

5					
4					
3					
2	S	A	S	S	
1	A	S	A	A	S
Input String	a	b	a	a	b

- **Algorithm:**
  - Start with initialization
    - Fill table T with $t_{i,1}$ = {A | A → a_i ∈ P} for each i.
  - For each "level" of the table
    - t_{ij} = { A | for some k, 1 ≤ k < j, A → BC ∈ P, s.t. B ∈ t_{ik} and C ∈ $t_{i+k,j-k}$}
  - Accept Input iff S ∈ t_{1n}

CYK Example

G = {S,{A,S},{a,b}, P}
P:
- S → AA
- S → AS
- S → b
- A → SA
- A → AS
- A → a

Table

5					
4					
3	S				
2	S	A	S	S	
1	A	S	A	A	S
Input String	a	b	a	a	b

- **Algorithm:**
  - Start with initialization
    - Fill table T with $t_{i,1}$ = {A | A → a_i ∈ P} for each i.
  - For each "level" of the table
    - t_{ij} = { A | for some k, 1 ≤ k < j, A → BC ∈ P, s.t. B ∈ t_{ik} and C ∈ $t_{i+k,j-k}$}
  - Accept Input iff S ∈ t_{1n}

CYK Example

G = {S,{A,S},{a,b}, P}
P:
- S → AA
- S → AS
- S → b
- A → SA
- A → AS
- A → a

Table

5					
4					
3	S,A				
2	S	A	S	S	
1	A	S	A	A	S
Input String	a	b	a	a	b

- **Algorithm:**
  - Start with initialization
    - Fill table T with $t_{i,1}$ = {A | A → a_i ∈ P} for each i.
  - For each "level" of the table
    - t_{ij} = { A | for some k, 1 ≤ k < j, A → BC ∈ P, s.t. B ∈ t_{ik} and C ∈ $t_{i+k,j-k}$}
  - Accept Input iff S ∈ t_{1n}

CYK Example

G = {S,{A,S},{a,b}, P}
P:
- S → AA
- S → AS
- S → b
- A → SA
- A → AS
- A → a

Table					
5					
4					
3	S,A	S	S,A		
2	S	A	S	S	
1	A	S	A	A	S
Input String	a	b	a	a	b

- **Algorithm:**
  - Start with initialization
    - Fill table T with $t_{i1} = \{A \mid A \to a_i \in P\}$ for each i.
  - For each "level" of the table
    - $t_{ij} = \{ A \mid$ for some k, $1 \leq k < j$, $A \to BC \in P$, s.t. $B \in t_{ik}$ and $C \in t_{i+k,j-k}\}$
  - Accept Input iff $S \in t_{1n}$

CYK Example

G = {S,{A,S},{a,b}, P}
P:
- S → AA
- S → AS
- S → b
- A → SA
- A → AS
- A → a

Table					
5					
4	S,A	S,A			
3	S,A	S	S,A		
2	S	A	S	S	
1	A	S	A	A	S
Input String	a	b	a	a	b

- **Algorithm:**
  - Start with initialization
    - Fill table T with $t_{i1} = \{A \mid A \to a_i \in P\}$ for each i.
  - For each "level" of the table
    - $t_{ij} = \{ A \mid$ for some k, $1 \leq k < j$, $A \to BC \in P$, s.t. $B \in t_{ik}$ and $C \in t_{i+k,j-k}\}$
  - Accept Input iff $S \in t_{1n}$

CYK Example

G = {S,{A,S},{a,b}, P}
P:
- S → AA
- S → AS
- S → b
- A → SA
- A → AS
- A → a

Table					
5	S,A				
4	S,A	S,A			
3	S,A	S	S,A		
2	S	A	S	S	
1	A	S	A	A	S
Input String	a	b	a	a	b

- **Algorithm:**
  - Start with initialization
    - Fill table T with $t_{i1} = \{A \mid A \to a_i \in P\}$ for each i.
  - For each "level" of the table
    - $t_{ij} = \{ A \mid$ for some k, $1 \leq k < j$, $A \to BC \in P$, s.t. $B \in t_{ik}$ and $C \in t_{i+k,j-k}\}$
  - Accept Input iff $S \in t_{1n}$

CYK Example

G = {S,{A,S},{a,b}, P}
P:
- S → AA
- S → AS
- S → b
- A → SA
- A → AS
- A → a

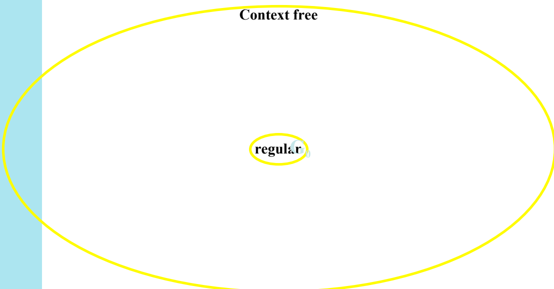
Table					
5	S,A				
4	S,A	S,A			
3	S,A	S	S,A		
2	S	A	S	S	
1	A	S	A	A	S
Input String	a	b	a	a	b

- **Algorithm:**
  - Start with initialization
    - Fill table T with $t_{i1} = \{A \mid A \to a_i \in P\}$ for each i.
  - For each "level" of the table
    - $t_{ij} = \{ A \mid$ for some k, $1 \leq k < j$, $A \to BC \in P$, s.t. $B \in t_{ik}$ and $C \in t_{i+k,j-k}\}$
  - Accept Input iff $S \in t_{1n}$

Early Algorithm

Classification of Grammars

Classification of Grammars



Regular Grammars

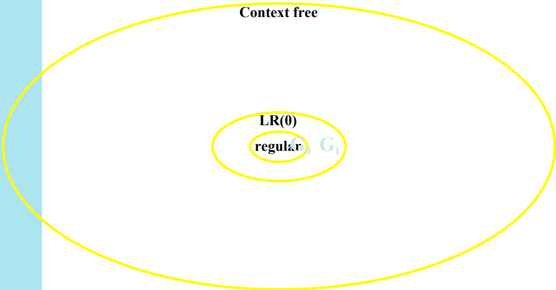
- A grammar that can be expressed using a regular expression is a regular grammar
- Example Language:
  - Zero or more left parentheses followed by zero or more right parentheses
- $G_0 = \{ \; (^a \;)^b \mid a, b >= 0 \; \}$
- Grammar

  S → X Y $

  X → (X | ε

  Y →) Y | ε

Classification of Grammars

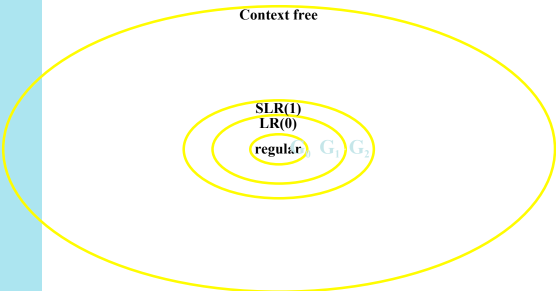


LR(0) Grammars

- A grammar that can create a LR(0) parse table without any shift/reduce or reduce/reduce conflicts
- Example language:
  - One or more open parentheses followed by matching # of close parentheses
- $G_1 = \{\ (^n\ )^n \mid n > 0\ \}$
- The grammar

```
<S> → <X> $
<X> → ( <X> )     | ( )
```

Classification of Grammars

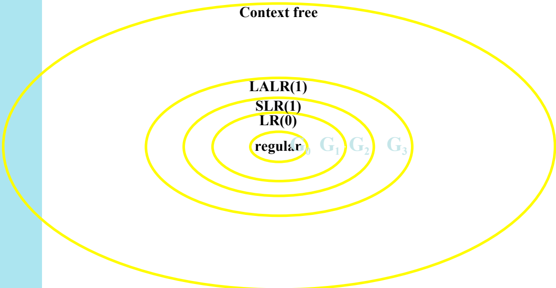


SLR(1) Grammars

- A grammar that can create a SLR(1) parse table without any shift/reduce or reduce/reduce conflicts
- Example language:
  - Zero or more open parentheses followed by matching # of close parentheses
- $G_2 = \{\ (^n\ )^n \mid n >= 0\ \}$
- The grammar

```
<S> → <X> $
<X> → ( <X> )     | ε
```

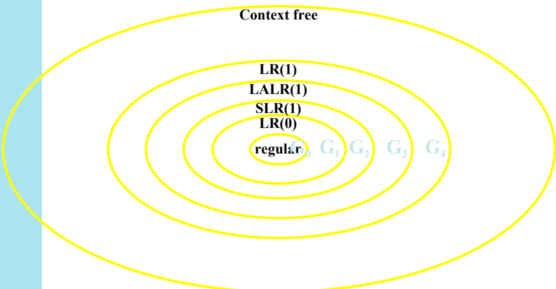
Classification of Grammars



LALR(1) Grammars

- A grammar that can create a LALR(1) parse table without any shift/reduce or reduce/reduce conflicts
- Example language:
  - ???
- G_3 = { ??? }
- The grammar

Classification of Grammars

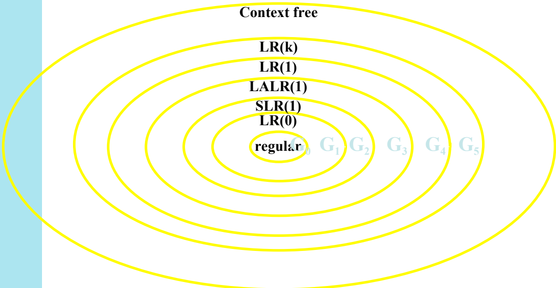


LR(1) Grammars

- A grammar that can create a LR(1) parse table without any shift/reduce or reduce/reduce conflicts
- Example language:
  - Zero or more open parentheses followed by matching # of close parentheses or single open parenthesis
- $G_4 = \{ (^n\ )^n \mid n >= 0 \} \cup \{ (\}$
- The grammar

<S> → <X> $
<X> → (| <Y>
<Y> → (<Y>) | ε

Classification of Grammars

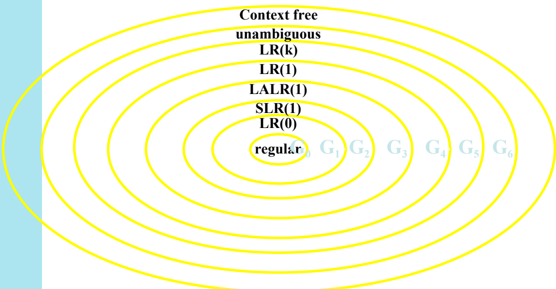


LR(k) Grammars

- A grammar that can create a LR(k) parse table without any shift/reduce or reduce/reduce conflicts
- Example language:
  - Zero or more open parentheses followed by matching # of close parentheses or a matching # of close brackets
- $G_5 = \{\ (^n\ )^n \mid n >= 0\ \} \cup \{\ (^n\ ]^n \mid n >= 0\ \}$
- The grammar

```
<S> → <X> $
<X>  → <Y> | <Z>
<Y> → ( <Y> )      | ε
<Z> → ( <Z> ]      | ε
```

Classification of Grammars

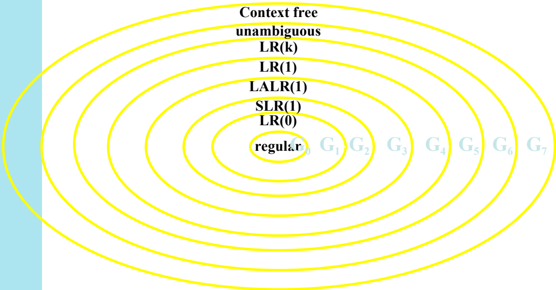


Unambiguous Grammars

- A grammar is unambiguous if and only if it has a unique rightmost derivation sequence (parse tree)
- Example:
- $G_6 = \{\ [(^n\ )^n \mid n >= 0\ \} \cup \{\ ](^n\ )^{2n} \mid n >= 0\ \}$
- The grammar

```
<S> → <X> $
<X>  → [ <Y> | ] <Z>
<Y> → ( <Y> )      | ε
<Z> → ( <Z> ))     | ε
```

Classification of Grammars

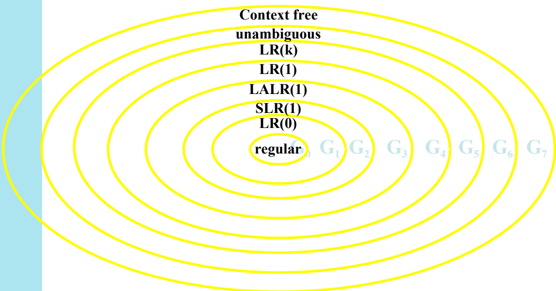


Ambiguous Grammars

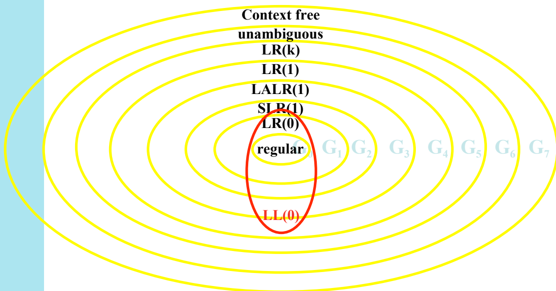
- A grammar is ambiguous if and only if it has more than one rightmost derivation sequence
- Example:
- $G_7 = \{ (^i \;)^j \; (^k \mid i = j \text{ or } j = k \}$
- The grammar

```
<S> → <X> $
<X> → <P> <Q> | <R> <S>
<P> → ( <P> )   | ε
<Q> → ( <Q>     | ε
<R> → ( <R>     | ε
<S> → ) <S> (   | ε
```

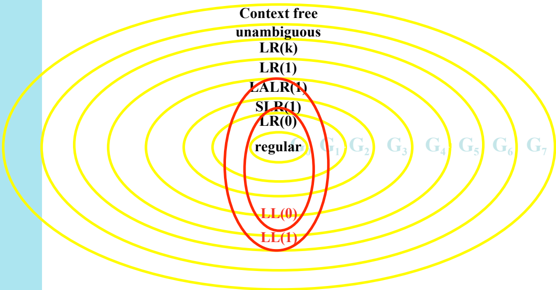
Classification of Grammars



Classification of Grammars



Classification of Grammars



Question

- What about the language?
- $G_8 = \{ (^i\ )^j\ (^k \mid i = j = k \}$

LR Languages

- A context-free language is an LR language if and only if it can be generated by an LR(k) grammar for some fixed k

LR Languages

- The set of LR languages are independent of the lookahead distance k
- Given any LR(k) grammar G_k, there exist a LR(0) grammar G_0 such that $L(G_k) = L(G_0)$
- For all the languages with SLR(1), LALR(1) and LR(1) grammars we looked at, we could have found a LR(0) grammar!!!

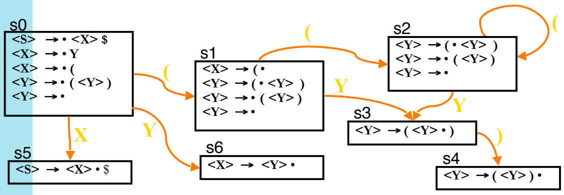
Example

- Language
  - Zero or more open parentheses followed by matching # of close parentheses
  - or a single open parenthesis
- LR(1) Grammar

  <S> → <X> $
  <X> → <Y>
  <X> → (
  <Y> → (<Y>)
  <Y> → ε
- Is there a LR(0) Grammar for this language?

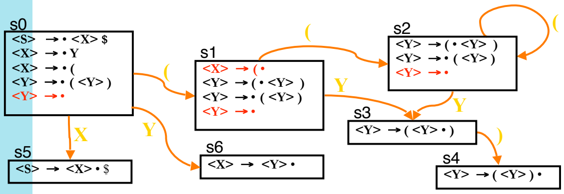
Expanded Example DFA

<S> → <X> $
<X> → <Y>
<X> → (
<Y> → (<Y>)
<Y> → ε



Expanded Example DFA

<S> → <X> $
<X> → <Y>
<X> → (
<Y> → (<Y>)
<Y> → ε



Example

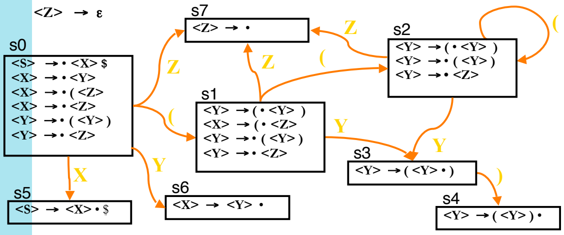
- Language
  - Zero or more open parentheses followed by matching # of close parentheses
  - or a single open parenthesis
- LR(1) Grammar

  <S> → <X> $
  <X> → <Y>
  <X> → (
  <Y> → (<Y>)
  <Y> → ε
- LR(0) Grammar

  <S> → <X> $
  <X> → <Y>
  <X> → (<Z>
  <X> → <Z>
  <Y> → (<Y>)
  <Y> → <Z>
  <Z> → ε

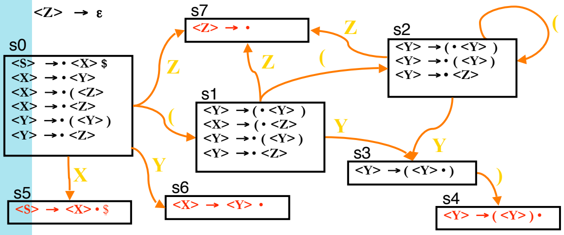
DFA of the Example

<S> → <X> $
<X> → <Y>
<X> → (<Z>
<X> → <Z>
<Y> → (<Y>)
<Y> → <Z>
<Z> → ε



DFA of the Example

<S> → <X> $
<X> → <Y>
<X> → (<Z>
<X> → <Z>
<Y> → (<Y>)
<Y> → <Z>
<Z> → ε



LR Languages

- The set of LR languages are independent of the lookahead distance k.
- Given any LR(k) grammar G_k, there exist a LR(0) grammar G_0 such that $L(G_k) = L(G_0)$
- For all the languages we looked at, we could have found a LR(0) grammar!!!
- But this can be very hard!!!

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
   int a, b, c, d;
{ … }
fee() {
   int f[3],g[0],
      h, i, j, k;
  char *p;
   fie(h,i,"ab",j, k);
   k = f * i + j;
   h = g[17];
   printf("<%s,%s>.\n",
      p,q);
   p = 10;
}
```

What is wrong with this program?
(let me count the ways …)

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
   int a, b, c, d;
{ … }
fee() {
   int f[3],g[0],
      h, i, j, k;
  char *p;
   fie(h,i,"ab",j, k);
   k = f * i + j;
   h = g[17];
   printf("<%s,%s>.\n",
      p,q);
   p = 10;
}
```

What is wrong with this program?
(let me count the ways …)

- declared g[0], used g[17]
- wrong number of args to fie()
- "ab" is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are "deeper than syntax"

To generate code, we need to understand its meaning !

Beyond Syntax

To generate code, the compiler needs to answer many questions

- Is "x" a scalar, an array, or a function? Is "x" declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of "x" does each use reference?
- Is the expression "x * y + z" type-consistent?
- In "*a*[i,j,k]", does *a* have three dimensions?
- Where can "z" be stored? *(register, local, global, heap, static)*
- In "f ← 15", how should 15 be represented?
- How many arguments does "fie()" take? What about "printf ()" ?
- Does "*p" reference the result of a "malloc()" ?
- Do "p" & "q" refer to the same memory location?
- Is "x" defined before it is used?

These cannot be expressed in a CFG

Beyond Syntax

These questions are part of context-sensitive analysis

- Answers depend on values, not parts of speech
- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
  - Context-sensitive grammars?
  - Attribute grammars? *(attributed grammars?)*
- Use *ad-hoc* techniques
  - Symbol tables
  - *Ad-hoc* code *(action routines)*

In scanning & parsing, formalism won; different story here.

Beyond Syntax

Telling the story

- The attribute grammar formalism is important
  - Succinctly makes many points clear
  - Sets the stage for actual, *ad-hoc* practice
- The problems with attribute grammars motivate practice
  - Non-local computation
  - Need for centralized information
- Some folks in the community still argue for attribute grammars
  - Knowledge is power
  - Information is immunization

We will cover attribute grammars, then move on to *ad-hoc* ideas