

CS5000: Lecture 15

Vladimir Kulyukin

Department of Computer Science

Utah State University

Outline

- Top-Down Parsing
- Recursive-Descent Parsing
- Bottom-Up Parsing
- PDAs, DPDAs, and DCFLs

Top-Down Parsing

A Context-Free Language

- $L = \{xcx^R \mid x \text{ in } \{a, b\}^*\}$
- Here is a grammar for L:
 - $S \rightarrow aSa$
 - $S \rightarrow bSb$
 - $S \rightarrow c$

A Stack Machine for L

	read	pop	push
1.	ϵ	S	aSa
2.	ϵ	S	bSb
3.	ϵ	S	c
4.	a	a	ϵ
5.	b	b	ϵ
6.	c	c	ϵ

Nondeterminism

- Consider the input abbcbbba.
- There are three possible moves that our stack machine can make from the initial configuration:
 1. $(abbcbbba, S) \rightarrow (abbcbbba, aSa)$
 2. $(abbcbbba, S) \rightarrow (abbcbbba, bSb)$
 3. $(abbcbbba, S) \rightarrow (abbcbbba, c)$

Look-Ahead Rules

- When **S** is on top of the stack and the next input symbol is **a**, pop **S** and push **aSa**.
- When **S** is on top of the stack and the next input symbol is **b**, pop **S** and push **bSb**.
- When **S** is on top of the stack and the next input symbol is **c**, pop **S** and push **c**.

Look-Ahead Table

	a	b	c	\$
S	$S \rightarrow aSa$	$S \rightarrow bSb$	$S \rightarrow c$	

\$ is the end of input symbol.

Table[A][a] gives a production to use when **A** is on top of the stack and **a** is the next symbol in the input.

Since **Table[S][\$]** is empty, when the input string has been read and **S** is on top of the stack, there is no way to proceed.

Predictive Parsing

```
void predictiveParser(input, table, S) {
    push S onto the empty stack;
    while ( stack is not empty ) {
        A = the top symbol on the stack;
        c = the current input symbol;
        if ( A is terminal ) {
            if ( A != c ) return failure;
            pop A;
            advance input to the next symbol;
        }
        else {
            if ( table[A][c] is empty ) return failure;
            pop A;
            push the right-hand side of table[A][c];
        }
    }
    if ( input's end is not reached ) return failure;
}
```

LL(1) Grammars

- The predictive parser on the previous slide is an example of LL(1) parsers.
- The first L means left-to-right scan of the input.
- The second L means that the parse follows the order of the leftmost derivation.
- 1 means that one look-ahead symbol is required.

Recursive-Descent Parsing

A Context-Free Language

- $L = \{xcx^R \mid x \text{ in } \{a, b\}^*\}$
- Here is a grammar for L:
 - $S \rightarrow aSa$
 - $S \rightarrow bSb$
 - $S \rightarrow c$

Recursive-Descent Parsing

- An LL(1) parser can be implemented recursively.
- One function is required for every non-terminal symbol in the grammar.

Input Matcher

```
void match(x) {  
    c = current symbol in input;  
    if ( c != x ) return failure;  
    advance input to the next symbol;  
}
```

Recursive-Descent Parser

```
void parse_S() {  
    c = current symbol in input;  
    if ( c == 'a' ) {  
        match('a'); parse_S(); match('a');  
    }  
    else if ( c == 'b' ) {  
        match('b'); parse_S(); match('b');  
    }  
    else if ( c == 'c' ) {  
        match('c');  
    }  
    else  
        return failure;  
}
```

Table-Driven vs. Recursive-Descent

- Both are top-down parsers.
- Table-driven parser uses an explicit stack.
- Recursive-descent parser also uses a stack but implicitly.

Bottom-Up Parsing

Shift-Reduce Parsing

- Shift-Reduce parsing is based on two operations:
 - **Shift:** Push the current input symbol onto the stack and advance to the next input symbol;
 - **Reduce:** There is a string on top of the stack that is the right hand side of some production. Pop it off the stack and replace it with the nonterminal left-hand side symbol of that production.

Example

Input	Stack
<u>a</u> bbcbba\$	ϵ

SHIFT

Example

Input	Stack
a <u>b</u> bc b ba\$	a

SHIFT

Example

Input	Stack
ab <u>b</u> cbba\$	ba

SHIFT

Example

Input	Stack
abb <u>c</u> bba\$	bba

REDUCE by $S \rightarrow c$

Example

Input	Stack
abbc <u>b</u> ba\$	Sbba

SHIFT

Example

Input	Stack
abbc <u>b</u> a\$	bSbba

REDUCE by $S \rightarrow bSb$

Example

Input	Stack
abbc <u>b</u> a\$	Sba

SHIFT

Example

Input	Stack
abbcbb <u>a</u> \$	bSba

REDUCE by $S \rightarrow bSb$

Example

Input	Stack
abbc <u>b</u> a\$	Sa

SHIFT

Example

Input	Stack
abbc <u>b</u> ba\$	aSa

REDUCE by $S \rightarrow aSa$

Example

Input	Stack
abbc <u>b</u> ba\$	S

S is on top of the stack.
the end of the input has been reached.
So we are DONE!!!

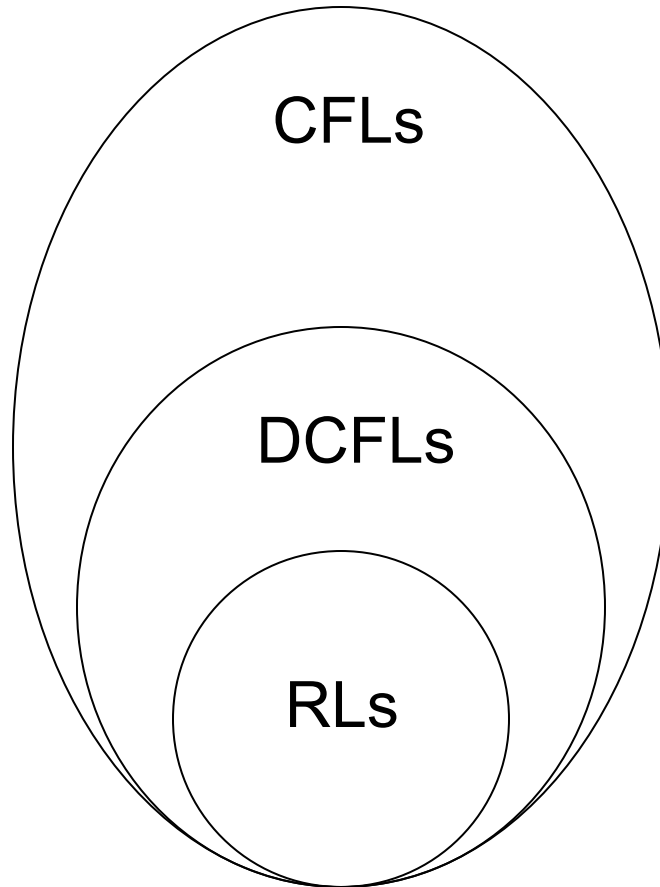
PDAs, DPDAs, and DCFLs

- PDAs stand for push-down automata.
- They are the same as stack machine and accept the same class of languages: context-free languages.
- (q, r, a, Z, x) : when in state q , the input symbol is a , and Z is on top of the stack, pop Z off the stack, move x on top of the stack and go to state r .

PDAs, DPDAs, and DCFLs

- DPDAs are deterministic PDAs.
- DCFLs are deterministic context-free languages.
- A DCFL is a CFL accepted by some DPDA.
- DCFLs are a proper subset of CFLs.

RLs, DCFLs, and CFLs



Recommended Reading

- Sections 10.8, 10.9.