California State University, Fullerton
Dept of Computer Science

CPSC 323 Compilers & Languages (Spring 2024)

ASSIGNMENT 2

Please answer the questions appropriately, and if you believe a diagram is required, draw one.

If you want to solve the problems, do so while documenting your steps.

NOTE: Please avoid writing single-word responses or just the single solution to a problem. Please submit a PDF document.

---

1. What is Context Free Grammar? Give an example.

Context-Free Grammar is a formal grammar, the syntax and structure of a formal language that can be described using context free grammar, it can also be a set of recursive rules used to generate patterns of strings, it is also grammar that satisfies certain properties, the also describe formal languages, context free grammar generates a context free language, it is also an entire differently form of defining a class of languages, context free grammars are also known as Backus-Naur Form,, which is a natural notation for describing syntax, CFG also describes the syntax of programming languages, the syntax of a programming language with a minor exceptions are described by context free grammars. Attribute grammars can be described with a context free grammar. CFG also describes regular languages but not all possible languages, its also a concatenation of two simpler languages, it consists of a set of productions that you use to replace a variable by a string of variables and terminals. The language would be context-free if there is a CFG for it. CFG is also used to define context-free languages, written are a class of formal languages with certain characteristics, including a set of terminals, non-terminals,start symbol, and production rules, CFG is a set of recursive rules(or productions) used to describe patterns of stirngs in a language

Example of context free grammer

R1) S - > a B
R2) S - > bA
R3) A - > a
R4) A - > aS
R5) A - > bAA
R6) B - > b
R7) B - > bS
R8) B - > a BB
Simple String: ab, ba, abab, aaabbb etc

Another example of context free grammar

R1) S --> s1 | s2
R2) S --> s1 s2
R3) S --> | s1 S

a CFG is a 4-tuple of G=(T,N,S,R), where
-T =a finite state of terminals
-N= a finite state of non-terminals(variables and N & T are disjoint
-R = a finite state of rules(productions) where each has the from A -->a, A e (N U T)*
-S = a unique start symbol of strings where S is non-terminal symbol of the first rule in set R

2. Explain the fundamental differences between top-down and bottom-up parsers in the context of syntax analysis.

The difference between the top-down and bottom up parsers in the context of syntax analysis is that top-down parsing starts from the top of the tree and continues to parse down until all the tokens are matched, if we want to construct a top-down parser we need to eliminate left-recursion and remove back-tracking, top down parser starts from the root of the parse tree with the start symbol and recursively works its way down to the leaves of the trees with and which are the terminals by using the rewriting rules of a formal grammar, top down parsers are usually less efficient than bottom up parsers for parsing complex grammars, there are some backtracking issues such as ambiguity or non-deterministic grammars, top down parsers are well suited for LL grammars, where LL stands for left-to-right, left most derivation, top down parsers start from the symbol and work their way down to the input tokens, the idea of top down parsers is that each non-terminal has an associated parsing procedure that can recognize any sequence of tokens generated by a non terminal , whereas the bottom up parser starts from the input tokens which are known as the terminal symbols and attempt to construct the parse tree in a bottom up fashion, it starts with the input tokens and combines them to form higher level constructs until it reaches the start symbol, the parsing techniques the bottom up parsing uses include shift reduce parsing like LR parsing, LR parsing, which is left-to-right, rightmost derivation, and SLR parsing(Simple LR). These techniques use a stack to keep track of partially constructs and apply reduction rules to form higher-level constructs, bottom up parsers are more efficient than top-down parsers and can handle a wider range of grammars including ambiguous and left recursive grammars, which require more sophisticated algorithms and parsing tables. The bottom up parsers are more versatile and can handle a wider range of grammars, including LR(0), SLR(1), LR(1), and LALR(1) grammars, bottom up parsers start from the input tokens and build up towards to start symbol, bottom up parser is about finding the handles and reduce until the top of the tree is found. Bottom up parser recognizes the smallest set of CFL, it uses precedence table, stack and driver to recognize a sentence. Overall in some cases top down is better, whereas bottom up is between depending on many factors such as grammar complexity, efficiency requirements, and types of grammars being parsed, there are some disadvantages of top-down parser, which means we have to remove the left recursion and change the grammar and it'll be harder to generate the intermediate code, we have to look at the RMD to see which reduction we need in order to perform a bottom up parser,

3. Eliminate left recursion from:

   $E \rightarrow Ea \mid b$

   $A \rightarrow c \mid d$

left recurive productions

E --> Ea
A--> c

For E:

E --> Ea | b

E --> bE'

E' --> aE' | €

For A:

A --> c | d
remains unchanged since there is no left recursion

Without the left recursion

E --> bE'
E' --> aE' | €
A --> c | d

4. Draw LL (1) parsing table for the following grammar. Also check if the input "acdb" can be parsed or not?

$S \rightarrow aABb$ R1

$A \rightarrow c \mid \varepsilon$ R2

$B \rightarrow d \mid \varepsilon$ R3

1. There is no left recursion in the grammar

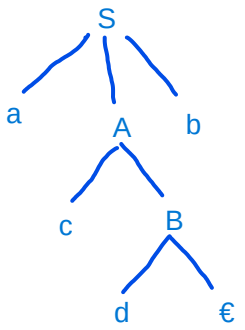
First(S) = {a}
First(A) = {c, €}
First(B) = {d, €}

Follow(S) = {$}
Follow(A) = First(B) U Follow(B) = {d, b} since there is a epsilon
Follow(B) = {b}

	a	b	c	d	$
S	S --> aABb				
A		A --> €	A --> c	A--> €	
B		B -> €		B-> d	

We can successfully parse input "acdb" based on the stack since another action is allowed to happen after putting that input, the stack we wrote shows that we can parse through it since its apart of the input and output and we can replace and shift various things as part of our actions, the parsing process secufessuly consues the input "acdb" and ends with an empty stack, which indicates the input can be parsed using the given LL(1) grammar

Stack	Input/ Output	Action
$0	acdb$	push s onto the stack
$0S	acdb$	S->aABb
$0bBAa	acdb$	pop a
$0bBA	cdb$	A --> c
$0bBc	cdb$	pop c
$0bB	db$	B --> d
$0bd	db$	pop d
$0b	b$	pop b
$0	$	accept



5. What is LR parsing? Discuss the differences between LR (0), SLR (1), CLR (1) and LALR (1) parsing tables.

LR parsing is a type of bottom-up parsing techniique used in the construction of a compiler during the syntax analysis phase and process any context-free grammars, it stands for left to right, rightmost derivation parsing, which it builds parse trees from the input string by starting from the left and replaced the rightmost non-terminal symbols with their productions until the string is reduced to the start symbol, it also reads input text from left to right without backing up, it produces right most derivation in reverse, they can parse a strictly large class of grammars, the LR parsing is difficult to produce the parse table by hand for a given grammar,

The differences between LR(0),SLR(1),CLR(1), and LALR(1) parsing tables are that the LR(0) parsing table has left to right, rightmost derivation with 0 lookahead, LR(0) is the simplest form of LR parsing table, it only uses the current state and the next input symbol(lookahead) to determine which action will take place next, LR(0) parsing tables can handle only a limited amount of grammars because they lack lookahead information, leading to conflicts and ambiguity in more complex grammars, the set on the right has a shift reduce conflict, it is the easiest to learn, these parsing tables are to weak to be of practical use because of a very limited set of grammars, even small additions to a LR(0) grammar can introduce conflicts, there is no lookahead. SLR(1) parsing tables, which is a simple LR with a lookahead of 1, this parsing table are a improvement over the LR(0) tables by using a one-symbol look ahead to solve parsing conflicts, it is simpler and more efficient than LR(0), CLR(1), and LALR(1) parsing tables, the SLR(1) parsing tables can handle a wider range of grammars than LR(0), but may still have conflicts in more complex grammars, any grammar that can be parsed with LR(0) parsing table can be parsed with a SLR(1) parser, the SLR(1) parsing tables can parse a larger number of grammars. CLR(1) parsing table is a canonical LR with a lookahead of 1, the CLR(1) parsing table are much more powerful than SLR(1) and LR(0) tables as they use a canonical collection of LR(1) items to construct the parsing table, the CLR(1) parsing table can handle a larger class of grammars compared to SLR(1) tables, including most of the LR(1) grammars, constructing the CLR(1) parsing tables can be more complex and resource intensive, the CLR(1) parsing table produces more number of states as compared to the SLR(1) parsing, for the input string writes a context free grammar, checks the ambiguity of the grammar, adds a augment production in the given grammar, creates a canonical collection of LR(0) items, draws a data flow diagram, and constructs a CLR(1) parsing table. The LALR(1) parsing table, is known as a look ahead LR with a lookahead of 1, the LALR(1) parsing tables are a compromise between the simplicity of SLR(1) tables and the power of CLR(1) tables, they use less verbose representation of LR(1) items compared to a CLR(1) table, resulting in smaller parsing tables and less memory requirements, the LALR(1) parsing tables can handle a wide range of grammars more than the other tables previously mentioned and are commonly used in practice due to their balance of effiency and power, we also attempt to reduce the number of states in the LALR(1) by merging similiar states that the LR(1) or CLR(1) parser created,the LALR(1) parser is the most powerful parser which can handle large classes of grammar, the LR parsing tables vary in complexity and power, with LR(0) being the simplest but limited in capacity, SLR(1) offering most improved handling of grammar, CLR(1) providing greater generalbilitty at the cost of compelxity, and LALR(1) shrinking a balance between effiency and parsing power

Question 6 work

E --> T + E
E --> T
T --> id

6. Check whether the Grammar is LR (0) or not?
   $E \rightarrow T + E \mid T$
   $T \rightarrow id$

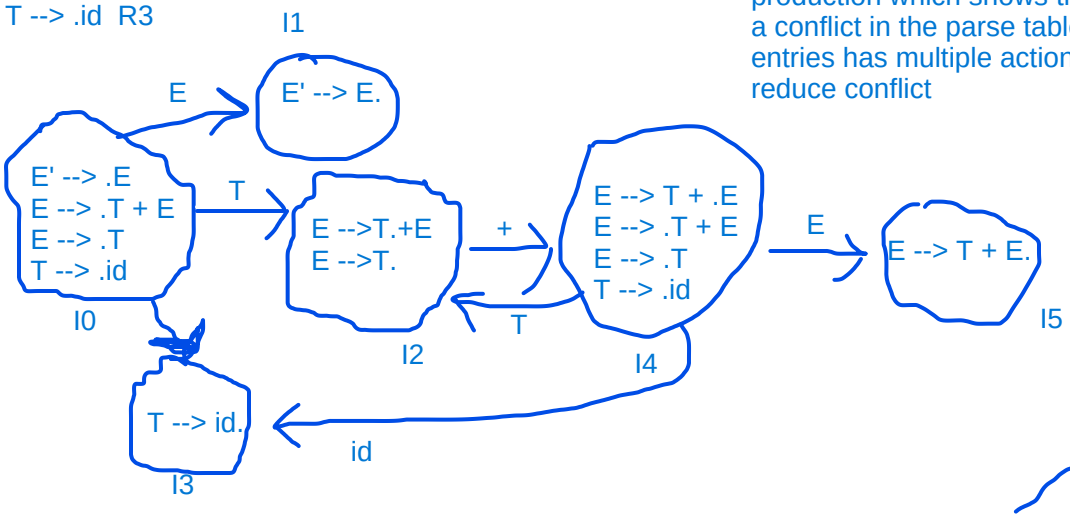
It is not considered a LR(0) parser because the parse table has a shift-reducing conflict in the terminal at input symbol "+" at state I2, in which it shifts 4 and reduces 2, it is also not LR(0) becuase it is left recursive, in which E --> T + E introduces the left recursion because E appears as the firs symbol on the right side of the production which shows that its not LR(0), there's a conflict in the parse table, in which one of the entries has multiple actions , there is a shift reduce conflict

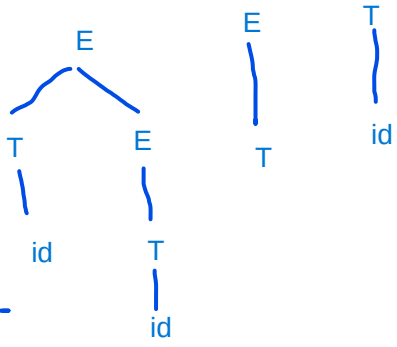
Step 1 Augment the given grammar

E' --> E

Step 2 draw canonical collection of LR(0) items

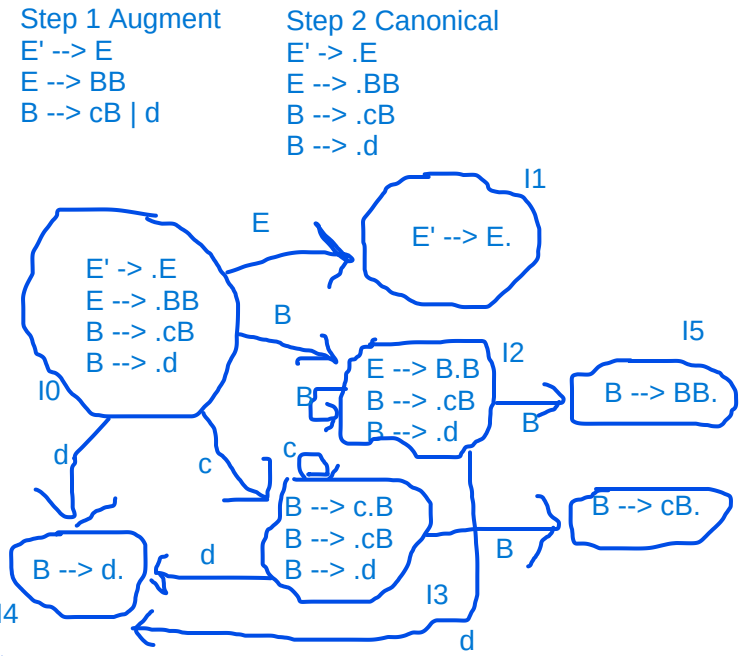
E' --> .E
E --> .T + E R1
E --> .T R2
T --> .id R3





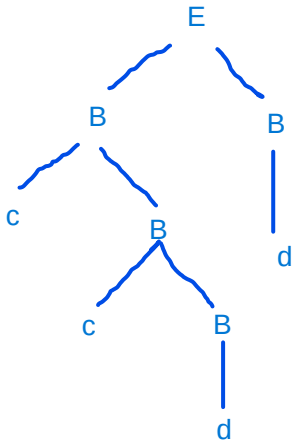
state	Terminal			Nonterminal	
	id	+	$	E	T
I0	S3			1	2
I1			accep -ted		
I2	R2	S4/R2	R2		
I3	R3	R3	R3		
I4	S3			5	2
I5	R1	R1	R1		

7. Construct the LR (0) parsing table for the given production rules. Also draw the data flow diagram along with the stack implementation for input "ccdd$".
   E –> BB
   B –> cB | d



state	Terminal			Non Terminal	
	c	d	$	E	B
I0	S3	S4		1	2
I1			acce		
I2	S3	S4			5
I3	S3	S4			5
I4	R3	R3	R3		
I5	R1	R1	R1		
I6	R2	R2	R2		

Stack	I/0	Action
$0	ccdd$	shift 3
$0c3	cdd$	shift 3
$0c3c3	dd$	shift 4
$0c3c3d4	d$	shift 4
$0c3c3d4	d$	reduce 3
$0c3c3B6	d$	reduce 2
$0c3B6	d$	reduce 2
$0B2	d$	shift 4
$0B2d4	$	reduce 3
$0B2B5	$	reduce 1
$0E1	$	accept



B c.B R1
B .cB R1
E BB. R2

8. Construct the SLR (1) parsing table for the given production rules. Also draw the data flow diagram along with the stack implementation for input "abab$".

S –> AS | b
A –> SA | a

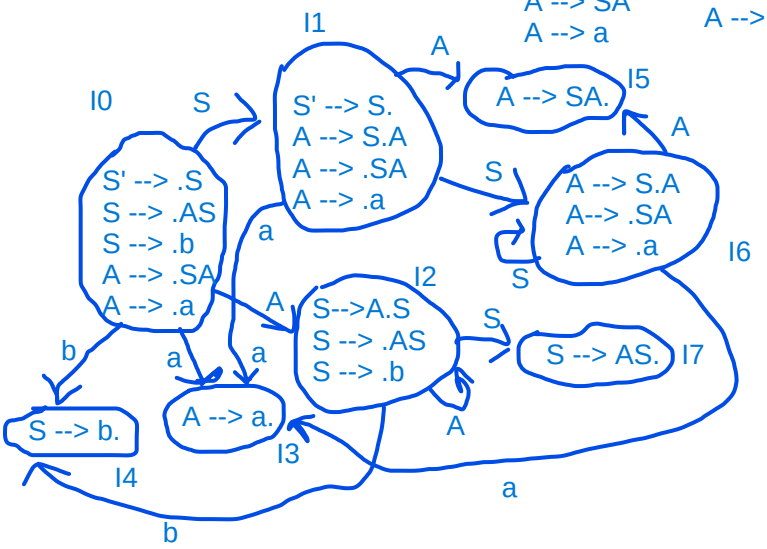
1. Augment

S' --> S
S --> AS
S --> b
A --> SA
A --> a

2. Canonical

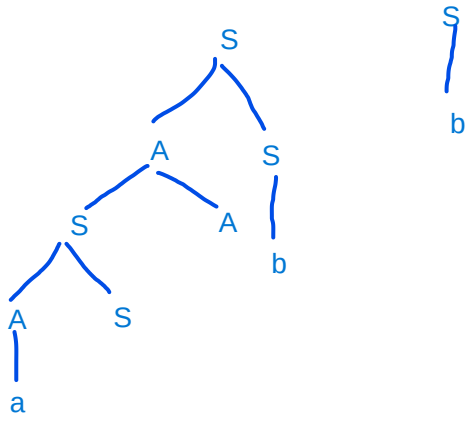
S' --> .S
S --> .AS R1
S --> .b R2
A --> .SA R3
A --> .a R4

follow(S) = {$, a,b}
follow(A) = {a, b}
first(S) = {a,b}
first(A) = {a,b}
follow(S) = first(A) U follow(S)
follow(A) = first(S) U follow(A)
first(S) = first(A) U first(S)
first(A) = first(S) U first(S)



stack	I/O	Action
$0	abab$	shift 3
$0a3	bab$	reduce 4
$0A4	bab$	shift 4
$0A4b4	ab$	reduce 2
$0A4S2	ab$	reduce 1
$0S1	ab$	shift 3
$0S1a3	b$	reduce 4
$0S1A4	b$	shift 4
$0S1A4b4	$	reduce 2
$0S1A4S2	$	reduce 1
$0S1S1	$	accept

state	Terminal			Non-Terminal	
	a	b	$	S	A
I0	s3	s4		1	2
I1	s3		acept	6	5
I2		s4		7	2
I3	R4	R4			
I4	R2	R2	R2		
I5	R3	R3			
I6	s3			6	5
I7	R1	R1	R1		



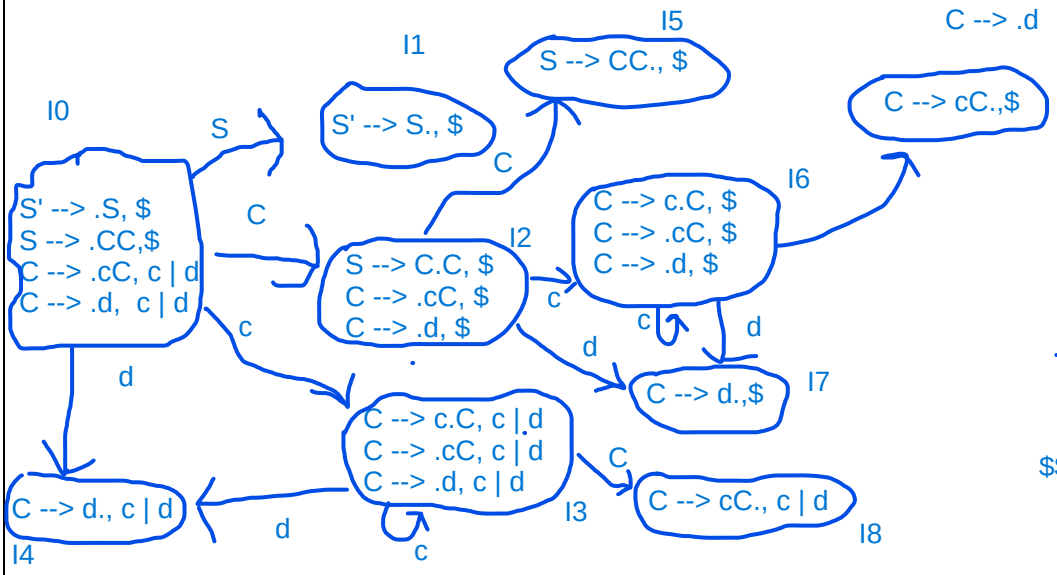
9. Construct the CLR (1) parsing table for the given production rules. Also draw the data flow diagram along with the stack implementation for input "dd$".

S –> CC R1
C –> cC | d R2 | R3

Step 1 : Augment
S' --> S
S --> CC
C --> cC
C --> d

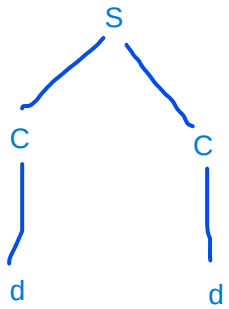
Step 2: Canonical

S' --> .S
S --> .CC
C --> .cC
C --> .d



stack	I/O	action
$0	dd$	shift 4
$0d4	d$	reduce 3
$0C3	d$	shift 4
$$0C3d4	$	reduce 3
$0C3C5	$	reduce 2
$0S1	$	accept

state	Terminal			Non-Terminal	
	c	d	$	S	C
I0	S3	S4		1	2
I1			accept		
I2	S6	S7			5
3	S3	S4			8
I4	R3	R3			
I5			R1		
I6	S6	S7			9
I7			R3		
I8	R2	R2			
I9			R2		



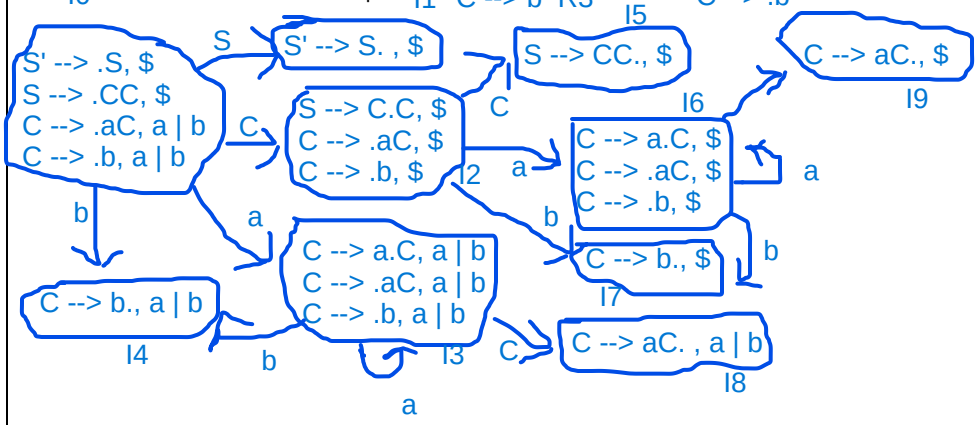
10. Construct the LALR (1) parsing table for the given production rules. Also draw the data flow diagram along with the stack implementation for input "bb$".

S –> CC

C –> aC | b

Step 1 : Augment
S' --> S
S --> CC R1
C --> aC R2
C --> b R3

Step 2 : Canonical
S' --> .S
S --> .CC
C --> .aC
C --> .b



I3 - I6
I4 - I7
I8 - I9

We combine states i3 and i6
i4 and i7
i8 and i9

State	Terminal			Non - Terminal	
	a	b	$	S	C
I0	S3, S6	S4, S7		1	2
I1			accept		
I2	S3, S6	S4, S7			5
I3, I6	S3, S6	S4, S7			8,9
I4, I7	R3	R3	R3		
I5			R1		
I8, I9	R2	R2	R2		

Stack	I/O	Action
$0	bb$	shift 4
$0b4	b$	reduce 3
$0C8	b$	shift 4
$0C8b4	$	reduce 3
$0C8C8	$	reduce 1
$0S1	$	accept