CENSORED

9/28/18

COMP 3220

HW1

1.)

I.) <S> -> <A> a <B> b

-> ba <B> b

-> baab

**Explanation:** So, baab is obtainable with the given grammar.

II.) <S> -> <A> a <B> b

-> <A> ba <B> b

-> <A> bba <B> b

-> bbba <B> b

-> bbbaab

**Explanation:** So, bbbab is not obtainable with the given grammar.

III.) <S> -> <A> a <B> b

-> <A> ba <B> b

-> bba <B> b

-> bbaa <B> b

-> bbaaa <B> b

-> bbaaaab

**Explanation:** So, bbaaaa is not obtainable with the given grammar.

IV.) <S> -> <A> a <B> b

-> <A> ba <B> b

-> bba <B> b

-> bbaab

**Explanation:** So, bbaab is obtainable with the given grammar.

2.) Token | Lexeme

= equals

+ addition

\* multiplication

( left parenthesis

) right parenthesis

3.) <assign>

/ \

<id> = <expr>

/ / | \

B = <id> + <expr>

/ \

B + ( <expr> )

/ | \

( <id> + <expr> )

/ \

( C + ( <expr> ) )

/ | \

( <id> \* <expr> ) )

/ \

( A \* <id> ) )

\

A ) )

**Explanation:** So, B = B + (C + (A \* A) ) can be obtained from the given grammar.

4.) S -> Aa | Bb

A -> CA’ | SbA’ // Redirects other statements to A prime

A’ -> aA’ | bCA’ | Ɛ // Uses recursion with A prime until epsilon (exit) is selected

B -> bb

C -> c

**Explanation:** So, all the other grammar rules stay the same aside from the rules for A. If choices are selected that do not include A naturally, it redirects to A prime. A prime allows the other options that redirected to A to remain used within the grammar rules for A prime until epsilon (exit) is selected, which terminates the recursive loop.

5.) Pairwise disjointness test:

A -> aBc | ac | a

FIRST(aBc) = a

FIRST(ac) = a

FIRST(a) = a

So, it fails the pairwise disjointness test.

B -> b | aB

FIRST(b) = b

FIRST(aB) = a

So, they do not intersect and it passes the pairwise disjointness test.

Left Factoring:

A -> aA’

A’ -> Bc | c | Ɛ

B -> b | aB

**Explanation:** So, the grammar for A is split into A and A prime. When A is called upon, it outputs an ‘a’ and links to the grammar rules for A prime. A prime then has three options: output ‘Bc’, output ‘c’, or epsilon (exit) is selected and the recursive loop is terminated. When ‘Bc’ is the output, it then goes to the B loop which does not need to be separated because it’s already in right-recursive form.

6.) r0: S’ -> E$

r1: E -> E + T

r2: E -> E \* T

r3: E -> T

r4: T -> (E)

r5: T -> id

**Explanation:** So, we break up the rules so that there aren’t any || options in each rule. By doing this, we expand the 2 given rules into 5 rules. The first rule (r0) is the one that tells the compiler to accept our input when it is an E.

7.) Given: (id + id) \* id & a parse table

|  |  |  |
| --- | --- | --- |
| Stack | Input | Action |
| 0 | (id + id) \* id $ | S4 |
| 0(4 | id + id) \* id $ | S5 |
| 0(4id5 | + id) \* id $ | R6 |
| 0(4F3 | + id) \* id $ | R4 |
| 0(4T2 | + id) \* id $ | R2 |
| 0(4E8 | + id) \* id $ | S6 |
| 0(4E8+6 | id) \* id $ | S5 |
| 0(4E8+6id5 | ) \* id $ | R6 |
| 0(4E8+6F3 | ) \* id $ | R4 |
| 0(4E8+6T9 | ) \* id $ | R1 |
| 0(4E8)11 | \* id $ | R5 |
| 0F3 | \* id $ | R4 |
| 0T2 | \* id $ | S7 |
| 0T2\*7 | id $ | S5 |
| 0T2\*7id5 | $ | R6 |
| 0T2\*7F10 | $ | R3 |
| 0T2 | $ | R2 |
| 0E1 | $ | Accept! |

**Explanation:** So, we perform a bottom-up parse on the given string and find that the compiler accepts the string with the given rules. When finding the stack, input, and action at each step, we can determine the grammar rules that allow the compiler to interpret the string. With the given parse table, we can find R1 – R6 using this new table. When the compiler is at state 4 and the id is found, we go to state 5 and find that the only output is R6 (rule 6). We then determine that rule 6 must be where F changes the state to 3. At state 3, rule 4 is the only option (and coming from state 4, the next option is T) so we go to state 2. Now, because the compiler isn’t receiving the \* symbol, we know that the next rule is rule 2. Since we are still technically in state 4, the last option is that rule 2 is when E is equal to T and changes the state to 8. We now have 3 out of 6 of the rules, so we keep working the string down to eventually discover what the other 3 rules are. When we hit 0(4E8+6T9 in the stack, we also finally hit rule 1 in the parse table. We need to allow the compiler to receive the right parenthesis (which is in state 8), so we know that the stack has to be combined so that 0(4E8 is the only thing in the stack. Given this (and the fact that the stack only has E + T, excluding the numbers and left parenthesis), we know we have to have the rule E -> E + T to effectively combine the two things in the stack and remain at state 8. We do the same thing for the other two rules and eventually get the compiler to accept our string input.

8.) Grammar Rules:

R0: S’ -> E$

R1: E -> E + T

R2: E -> T

R3: T -> T \* F

R4: T -> F

R5: F -> (E)

R6: F -> id

Rightmost derivation:

<S> -> E

-> T

-> T \* F

-> T \* id

-> F \* id

-> (E) \* id

-> (E + T) \* id

-> (E + F) \* id

-> (E + id) \* id

-> (T + id) \* id

-> (F + id) \* id

-> (id + id) \* id

**Explanation:** So, the above grammar rules dictate that id be assigned to F, that T be assigned F, and E be assigned T. We also have E -> E + T, T -> T \* F, and F -> (E). With the given rules, we use the rightmost derivation method to produce the given string and therefore prove that the bottom-up parse method used in question 7 correctly finds all of the handles for the input string.