CS323: Compilers

Spring 2023

Assignment 1

Assignment 1 – Q1 (6 mins)

Write a regular expression that matches different names of a *harvest festival* celebrated across India.

- Your expression must match at least one name attributed to the festival from the states of North, South, East, and West India
- Try to maximize the number of strings that your regular language/set contains.

(assume that the regular language is over English alphabets. and use the notations that we discussed in class).

Assignment 1 – Q2 (12 mins)

For the string `–(id+id)+id` show the sequence of derivations in:

a) Bottom-up parsing,

b) Recursive-descent parsing

```
The Grammar:
A -> B
A -> B+A
B -> -B
B -> id
B -> (A)
```

Hint: right-most derivation in reverse for bottom-up parsing.
Try all productions in that order for recursive descent parsing

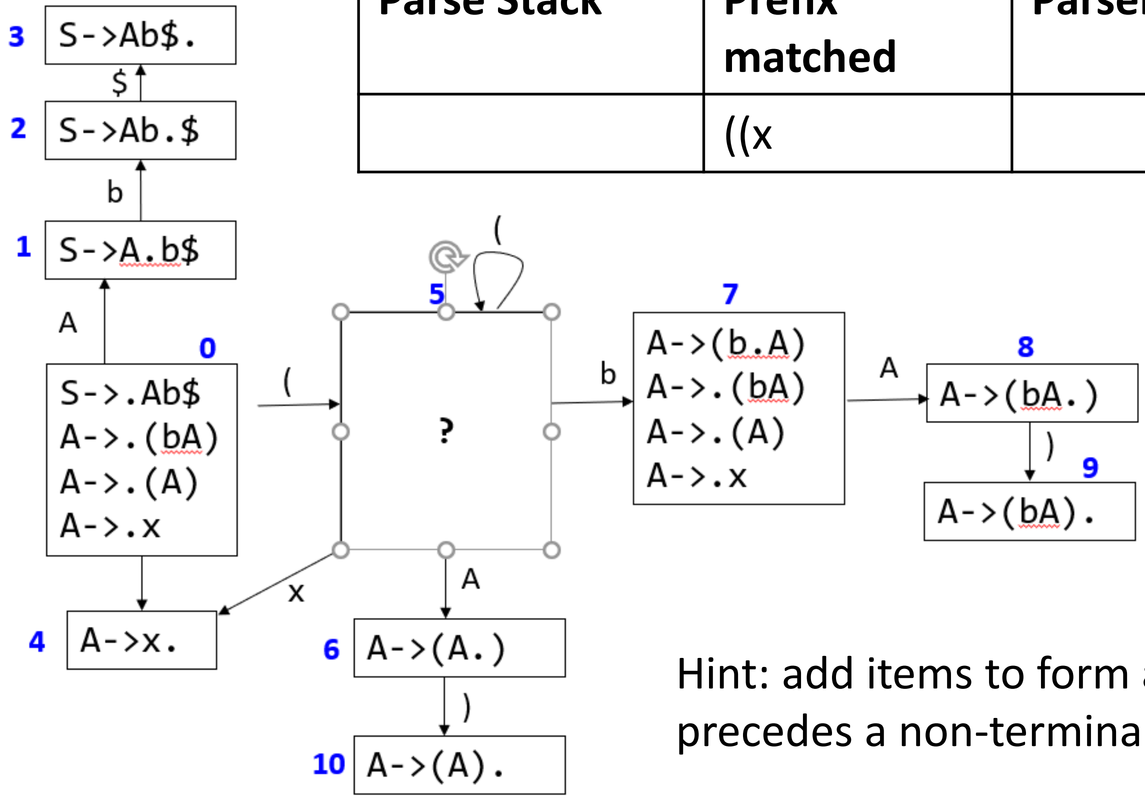
Assignment 1 – Q3 (8 mins)

The Grammar:

```
S -> Ab$
A -> (bA)
A -> (A)
A -> x
```

1. Complete the CFSM (fill state 5)
2. Fill the table and and add new entries if needed

Parse Stack	**Prefix matched**	**Parser Action**
	((x	



Hint: add items to form a closure if . precedes a non-terminal.

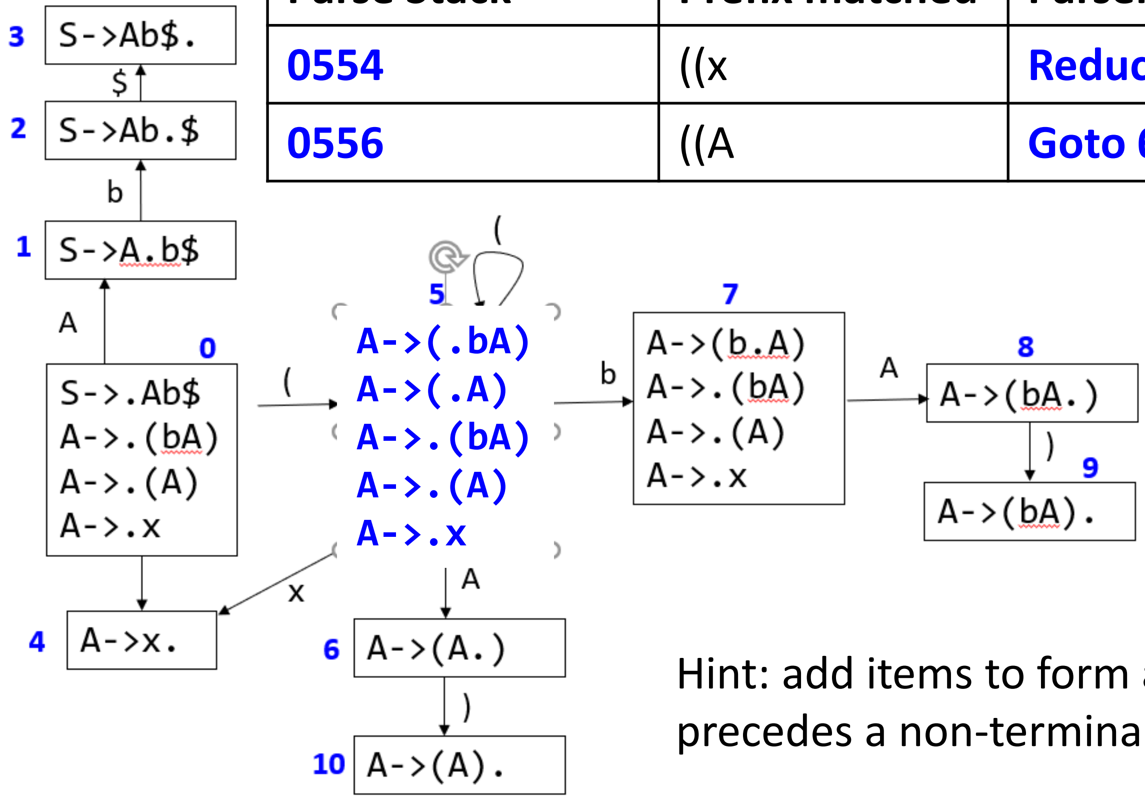
Assignment 1 – Q3 (answer)

The Grammar:

```
S -> Ab$
A -> (bA)
A -> (A)
A -> x
```

1. Complete the CFSM (fill state 5)
2. Fill the table and and add new entries if needed

Parse Stack	Prefix matched	Parser Action
0554	((x	**Reduce using rule A->x**
0556	((A	**Goto 6**



Hint: add items to form a closure if . precedes a non-terminal.

Assignment 1 – Q4 (12 mins)

- Draw the AST for the expression and generate 3-address code `a := b + c * d + 1 ;`
  - assume bison declarations:

```
%left *
%left +
```

Hint: + has higher priority than * and both operators are left associative. So, the resulting expression is treated as: `a := ((b + c) * (d + 1)) ;`

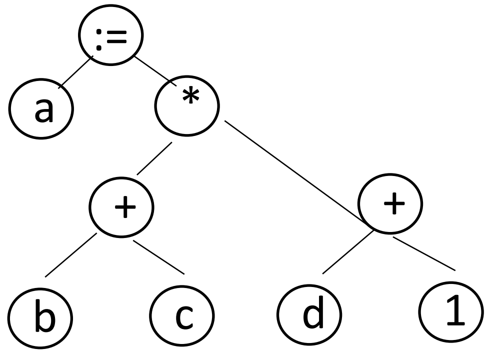
Assignment 1 – Q4 (answer)

Draw the AST for the expression and generate 3-address code `a := b + c * d + 1 ;`

- assume bison declarations:

```
%left *
%left +
```

Hint: + has higher priority than * and both operators are left associative. So, the resulting expression is treated as: a `:= ((b + c) * (d + 1)) ;`



Node a:
Temp: a
Type: l-value
Code: --

Node b:
Temp: b
Type: l-value
Code: --

Node c:
Temp: c
Type: l-value
Code: --

Node + (parent of b,c):
Temp: t1
Type: r-value
Code: ld b t2
ld c t3
add t2 t3 t1

Node d:
Temp: d
Type: l-value
Code: --

Node 1:
Temp: 1
Type: constant
Code: --

Node +(parent of d,1):
Temp: t4
Type: r-value
Code: ld d t5
add t5 1 t4

Node *:
Temp: t6
Type: r-value
Code: ld b t2
ld c t3
add t2 t3 t1
ld d t5
add t5 1 t4
mul t1 t4 t6

Node :=:
Temp: N/A
Type: N/A
Code: **ld b t2**
ld c t3
add t2 t3 t1
ld d t5
add t5 1 t4
mul t1 t4 t6
st t6 a

Acceptable if you just write the final answer shown in bold blue text. The order of traversal (postorder), generating code (left subtree followed by right subtree followed by self) must be adhered to. The order of generating temporaries and using them must be consistent.

Assignment 1 – Q5 (5 mins)

- Your language has a looping construct like C's **`do-while`** construct:

`do{S1;…;Sn;}while(cond1);` Statements $S_1 \ldots S_n$ are executed once before evaluating the condition `cond1`. The statements are executed repeatedly till the condition `cond1` becomes `false`.

- Pascal has the **`repeat-until`** construct:

`repeat{R1;…;Rn;}until(cond2);` Statements $R_1 \ldots R_n$ are executed once before evaluating the condition `cond2`. The statements are executed repeatedly till the condition `cond2`, becomes `true`.

- Now, you want to *remove* the `do-while` feature in your language and *introduce* a **`repeat-while`** construct:

`repeat{T1;…;Tn;}while(cond3);` Statements $T_1 \ldots T_n$ are executed once before evaluating the condition `cond3`. The statements are executed repeatedly till the condition `cond3` becomes `false`.

What phase(s) of the compiler you *must* change to implement the repeat-while construct? (explanation in support of your choices are welcome).

Assume keywords cannot be used as identifiers in your language

Assignment 1 – Q5 (answer)

- Your language has a looping construct like C's **`do-while`** construct:

`do{`S_1`;…;`S_n`;}while(`$cond_1$`);` Statements $S_1 … S_n$ are executed once before evaluating the condition $cond_1$. The statements are executed repeatedly till the condition $cond_1$ becomes `false`.

- Pascal has the **`repeat-until`** construct:

`repeat{`R_1`;…;`R_n`;}until(`$cond_2$`);` Statements $R_1 … R_n$ are executed once before evaluating the condition $cond_2$. The statements are executed repeatedly till the condition $cond_2$, becomes `true`.

- Now, you want to *remove* the `do-while` feature in your language and *introduce* a **`repeat-while`** construct:

`repeat{`T_1`;…;`T_n`;}while(`$cond_3$`);` Statements $T_1 … T_n$ are executed once before evaluating the condition $cond_3$. The statements are executed repeatedly till the condition $cond_3$ becomes `false`.

What phase(s) of the compiler you *must* change to implement the repeat-while construct? (explanation in support of your choices are welcome).

Assume keywords cannot be used as identifiers in your language

notice that meaning of do-while and repeat-while stays the same. Only the keyword has changed. At the least, the lexer must be modified. Parser may or may not be modified: You should remove the string "do" from the list of keywords in your lexer. In your lexer, you may return token DO when string "repeat" is seen in program text. This way, the parser need not be modified. If you want to make your compiler more readable, you return token REPEAT from lexer and then your parser has to declare %token REPEAT and hence, requires changes.

Marking criteria: -0.25 if parser is mentioned but no explanation is given. -0.25 is semantic routines or any other modules are mentioned.