Preparations

Given the grammar from the assignment specifications I eliminated Left recursion and some left factoring to receive the below productions. As we will see later however we need to make a lot more changes to the grammar in order for Javacc to utilize it:

Block ::= { DeclSet StmtSet } $$$
DeclSet ::= Decl DeclSet'
DeclSet' ::= Decl DeclSet' | ϵ
Decl ::= **int** Vars **;** |**bool** Vars **;**
Vars ::= **id** Vars'
Vars' ::= Vars" Vars' | ϵ
Vars" ::= **,** **id**

StmtSet ::= Stmt StmtSet'
StmtSet' ::= **;** Stmt StmtSet' | ϵ
Stmt ::= Assn
Assn ::= **id** Assn'
Assn' ::= **:=** Assn"
Assn" ::= Arith |Comp |Cond
Arith ::= Term Arith'
Arith' ::= **+** Term Arith' |- Term Arith' | ϵ
Term ::= Factor Term'
Term' ::= $\star$ Term Arith' | ϵ
Factor ::= **id** |**num** |**(** Arith **)**
Comp ::= Arith Comp'
Comp' ::= **=** Arith |> Arith |< Arith
Cond ::= Comp Cond'
Cond' ::= -> Arith Cond"
Cond" ::= **:** Arith

Below are the rules I used to represent the tokens of the grammar.

```
TOKEN :
{
    <INT : "int">
|   <BOOL : "bool">
|   <NUM : (["0"-"9"])+>
|   <ID : (["A"-"Z","a"-"z"])+>
|   <ASSIGN : ":=">
|   <LPAR : "{">
|   <RPAR : "}">
|   <LBRAC : "(">
|   <RBRAC : ")">
|   <SUM : "+">
|   <SUB : "-">
|   <PROD : "*">
|   <EQU : "=">
|   <GRTN : ">">
|   <LSTN : "<">
|   <IF : "->">
|   <ELSE : ":">
|   <SEMIC : ";">
|   <COMMA : ",">
}
```

Task 4 & 5 - *Grammar amendments*

Getting a working Abstract syntax tree

There isn't much to say for the basics of how the grammar.jjt file that I made works. I simply followed the first case study we did in class with regards to format of the file.

However in order to get javacc to correctly run my grammar I've had to make several changes to it. At first I needed to change the `Assn` part of the grammar since `Assn'' := Arith | Comp | Cond` produces multiple ways of starting the production with **id, num** and **(**....(I've highlighted changes in italics and with a lighter font!)

Assn ::= **id** Assn'
Assn' ::= **:=** *Term* Assn"
Assn" ::= *Arith Assn'''*
Assn"' ::= *Comp Cond* | ϵ
Arith ::= Term *Arith*
Arith' ::= **+** Arith' |- Arith' | ϵ
Term ::= Factor Term'
Term' ::= $\star$ Arith' | ϵ
Factor ::= **id** | **num** | **(** *Arith'* **)**
Comp ::= **=** *Arith'* |> *Arith'* |< *Arith'*
Cond ::= -> *Arith'* Cond'
Cond' ::= **:** *Arith'*

A helping hand

Moving forward from here we will want to generate code for specific parts of the input file. In order to do this we will need to alter the grammar even further. Notably in the case where we have a production like A ::= αB | ϵ we need to create a new production, say A', that says:

A ::= A'B | ϵ
A' ::= α

This is important so that the parser doesn't generate code where it uses epsilon as a terminal. We avoid this by making a new `Node` on A'. Utilizing the above rule we end up with a sort of super-grammar with no look ahead and we'll use this as the grammar for Task 5 of the assignment.

continued on next page...

Working with Arithmetic instructions

Implementing the previously mentioned rule the following is the grammar amendments for generating arithmetic code. (Again all amendments are in italics with a lighter font)

Assn ::= **id** Assn'
Assn' ::= **:=** Term Assn"
Assn" ::= Arith Assn"'
Assn"' ::= *Comp* | ϵ
Arith ::= *SUMArith* | *SUBArith* | *PRODArith*
SUMArith ::= **+** *Arith'*
SUBArith ::= **-** *Arith'*
PRODArith ::= $\star$ *Arith'*
Arith' ::= *Term Arith*
Term ::= Factor *TermArith*
TermArith ::= *Term Arith* | ϵ
Factor ::= *IDFactor* | *NUMFactor* | **(** Arith' **)**
IDFactor ::= **id**
NUMFactor ::= **num**

You'll notice that I have a production called `TermArith` that's very similar to `Arith'`. The only difference here is that `TermArith` is nullable whereas `Arith'` isn't. This is simply so that I can have an **id**, **num** or **(** Arith' **)** by itself.

WORKING: Yes

Working with INT and BOOL

The amendments to the `Declset` productions are much simpler than the previous amendments. As shown below it simply involves creating different paths for both an INT and a BOOL so that they can be treated individually:

Block ::= { DeclSet StmtSet } $$$
DeclSet ::= Decl DeclSet'
DeclSet' ::= Decl DeclSet' |ϵ
Decl ::= **int** *INTVars* **;** | **bool** *BOOLVars* **;**
INTVars ::= *INTID INTVars'*
INTVars' ::= *INTVars''* *INTVars'* | ϵ
INTVars'' ::= **,** *INTID*
INTID ::= **id**
BOOLVars ::= *BOOLID BOOLVars'*
BOOLVars' ::= *BOOLVars''* *BOOLVars'* | ϵ
BOOLVars'' ::= **,** *BOOLID*
BOOLID ::= **id**
Vars ::= **id** Vars'
Vars' ::= Vars" Vars' | ϵ
Vars" ::= **,** **id**

WORKING: Yes

Working with IF and ELSE

Representing if and else conditions is also pretty easy as shown below

Assn"' ::= Comp | ϵ
⋮
Comp ::= *EQUComp Cond* | *GRTNComp Cond* | *LSTNComp Cond*
EQUComp ::= **=** Arith'
GRTNComp ::= > Arith'
LSTNComp ::= < Arith'
Cond ::= *IFCond ELSECond* | ϵ
IFCond ::= **->** *Arith'*
ELSECond ::= **:** *Arith'*

WORKING: Yes

Some sample outputs

Sample output:

```
input:  { bool tom; tom := 1+2 < 2+1 }
output:
[loadliteral, t0, 1, null]
[loadliteral, t1, 2, null]
[sum, t2, t1, t0]
[loadliteral, t3, 2, null]
[loadliteral, t4, 1, null]
[sum, t5, t4, t3]
[jumpless, LBL0, t2 , t5]
[jump, LBL1, null, null]
[label, LBL0, null, null]
[load, t6, TRUE, null]
[jump, LBL2, null, null]
[label, LBL1, null, null]
[load, t6, FALSE, null]
[label, LBL2, null, null]
[load, tom, t6, null]

input:  { int tom; dick := 2}
output:
[loadliteral, t0, 2, null]
[error, var used wasn't declared]
```

continued on next page...

input: { **int tom; tom := 2 < 4-> 3 : 1**}
output:
```
[loadliteral, t0, 2, null]
[loadliteral, t1, 4, null]
[jumpless, LBL0, t0 , t1]
[jump, LBL1, null, null]
[label, LBL0, null, null]
[loadliteral, t3, 3, null]
[jump, LBL2, null, null]
[label, LBL1, null, null]
[loadliteral, t3, 1, null]
[label, LBL2, null, null]
[load, tom, t3, null]
```

input:
{ **int tom, dick;**
int harry ; bool george;
tom := 5 - 4 - 3;
dick := (tom * 7) + 2;
george := tom < dick;
harry := tom < dick -> tom + 2 : tom - 2}
output:
```
[loadliteral, t0, 5, null]
[loadliteral, t1, 4, null]
[loadliteral, t2, 3, null]
[subtract, t3, t1, t2]
[subtract, t4, t0, t3]
[load, tom, t4, null]
[load, t5, tom, null]
[loadliteral, t6, 7, null]
[multiply, t7, t5, t6]
[loadliteral, t8, 2, null]
[sum, t9, t8, t7]
[load, dick, t9, null]
[load, t10, tom, null]
[load, t11, dick, null]
[jumpless, LBL0, t10 , t11]
[jump, LBL1, null, null]
[label, LBL0, null, null]
[load, t12, TRUE, null]
[jump, LBL2, null, null]
[label, LBL1, null, null]
[load, t12, FALSE, null]
[label, LBL2, null, null]
[load, george, t12, null]
[load, t13, tom, null]
[load, t14, dick, null]
[jumpless, LBL3, t13 , t14]
```

continued on next page...

```
[jump, LBL4, null, null]
[label, LBL3, null, null]
[load, t15, tom, null]
[loadliteral, t16, 2, null]
[sum, t20, t16, t15]
[jump, LBL5, null, null]
[label, LBL4, null, null]
[load, t18, tom, null]
[loadliteral, t19, 2, null]
[subtract, t20, t18, t19]
[label, LBL5, null, null]
[load, harry, t20, null]
```