Compiler Project 2 Tutorial

Goal

- Simple, create an AST of the code
  - How do we demonstrate this?
    - Easiest way: print the code back in a consistent format

- Break this into 3 parts
  - Create production rules
  - Make objects for each non-terminal
  - Make printing methods for each non-terminal

Getting Started - Scanner Class

- In the previous project the Lexer class was given and ran the lexer

- We need a new class now to run the scanner

ScannerTest.java

- Implement generated parser class
- Parse out the top of our tree
- Print output

```
Lexer lexer = new Lexer(reader);
Parser parser = new parser(lexer); //generated class
Program program = null;
try {
  program = (Program)parser.parse().value;
  System.out.print(program.toString(0));
}
catch (Exception e) {
  e.printStackTrace();
}
}
```

ScannerTest.java

- Need to create Program object
  - Should inherit from a Token class so we can insure methods exist
- Program should be the root of our tree.
- Feel free to copy this file from the repository.

```
    Lexer lexer = new Lexer(reader);
    parser parser = new parser(lexer); //generated class
    Program program = null;
    try {
      program = (Program) parser.parse().value;
      System.out.print(program.toString(0));
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Token.java

- Use this as the interface
- Use tabs for easy to read output
- ALL objects we create will extend this

```
abstract class Token {

 protected String getTabs(int t)
 {
   String tabs = "";
   for (int i = 0; i < t;i++)
   tabs = tabs + "\t";
   return tabs;
 }

 public String toString(int t)
 {
   return "";
 }
}
```

grammar.cup - Changes

- Previously we put terminals in our cup file out of necessity
  - Time to add some context by specifying a type when needed
  - There may be less obvious implementations to help clean things up

```
…
terminal ID; //4
…
terminal NUMBERLIT; //22
…
terminal STRINGLIT; //27
```

```
…
terminal String ID; //4
…
terminal int NUMBERLIT; //22
…
terminal String STRINGLIT; //27
```

How to start implementing productions

- There is no right way to implement your grammar
  - The easiest way will be to go from the bottom up
    - Since project 2 is broken into 2 parts, we are heavily incentivising taking this approach
    - However you don’t need to be completely meticulous and can put together several parts at a time

- Flow is as follows
  - We start with the Program production rules, initially it is empty
  - We add in new production rules to test
  - We make them viable rules from Program
  - We test
  - We add more

https://github.com/pattersonzUTD/UTDLang–

```
Script := Routines Main
Routines := Routine Routines
    | λ
Routine := ID Ins Outs start Body end
Ins := in : ID ( Type ) Ins
    | λ
Outs := out : ID ( Type ) Outs
    | λ
Type := number
    | string
    | flag
Main := main start Body end
Body := Variables Statements
Variables := Variable Variables
    | λ
Variable := ID ( Type ) :)
Statements := Statement Statements
    | λ
Statement := ID <- Expression :)
    | read ( ID ) :)
    | write ( ID ) :)
    | call ID ( IdList ) :)
    | when FlagExpression do Statements done :)
IdList := ID IdList
    | λ
Expression := NumericalExpression
    | StringExpression
    | FlagExpression
NumericalExpression := NUMBER
    | ID
    | NumericalExpression + NumericalExpression
    | NumericalExpression - NumericalExpression
    | NumericalExpression * NumericalExpression
    | NumericalExpression / NumericalExpression
StringExpression := STRING
    | ID
    | StringExpression + StringExpression
FlagExpression := up
    | down
    | ID
    | flip FlagExpression
    | FlagExpression + FlagExpression
    | FlagExpression * FlagExpression
    | NumericalExpression ? NumericalExpression
```

https://github.com/pattersonzUTD/UTDLang–

```
Script := Routines Main
Routines := Routine Routines
    | λ
Routine := ID Ins Outs start Body end
Ins := in : ID ( Type ) Ins
    | λ
Outs := out : ID ( Type ) Outs
    | λ
Type := number
    | string
    | flag
Main := main start Body end
Body := Variables Statements
Variables := Variable Variables
    | λ
Variable := ID ( Type ) :)
Statements := Statement Statements
    | λ
Statement := ID <- Expression :)
    | read ( ID ) :)
    | write ( ID ) :)
    | call ID ( IdList ) :)
    | when FlagExpression do Statements done :)

IdList := ID IdList
    | λ
Expression := NumericalExpression
    | StringExpression
    | FlagExpression
NumericalExpression := NUMBER
    | ID
    | NumericalExpression + NumericalExpression
    | NumericalExpression - NumericalExpression
    | NumericalExpression * NumericalExpression
    | NumericalExpression / NumericalExpression
StringExpression := STRING
    | ID
    | StringExpression + StringExpression
FlagExpression := up
    | down
    | ID
    | flip FlagExpression
    | FlagExpression + FlagExpression
    | FlagExpression * FlagExpression
    | NumericalExpression ? NumericalExpression
```

https://github.com/pattersonzUTD/UTDLang–

```
Script := Routines Main
Routines := Routine Routines
    | λ
Routine := ID Ins Outs start Body end
Ins := in : ID ( Type ) Ins
    | λ
Outs := out : ID ( Type ) Outs
    | λ
Type := number
    | string
    | flag
Main := main start Body end
Body := Variables Statements
Variables := Variable Variables
    | λ
Variable := ID ( Type ) :)
Statements := Statement Statements
    | λ
Statement := ID <- Expression :)
    | read ( ID ) :)
    | write ( ID ) :)
    | call ID ( IdList ) :)
    | when FlagExpression do Statements done :)
IdList := ID IdList
    | λ
Expression := NumericalExpression
    | StringExpression
    | FlagExpression
NumericalExpression := NUMBER
    | ID
    | NumericalExpression + NumericalExpression
    | NumericalExpression - NumericalExpression
    | NumericalExpression * NumericalExpression
    | NumericalExpression / NumericalExpression
StringExpression := STRING
    | ID
    | StringExpression + StringExpression
FlagExpression := up
    | down
    | ID
    | flip FlagExpression
    | FlagExpression + FlagExpression
    | FlagExpression * FlagExpression
    | NumericalExpression ? NumericalExpression
```

https://github.com/pattersonzUTD/UTDLang–

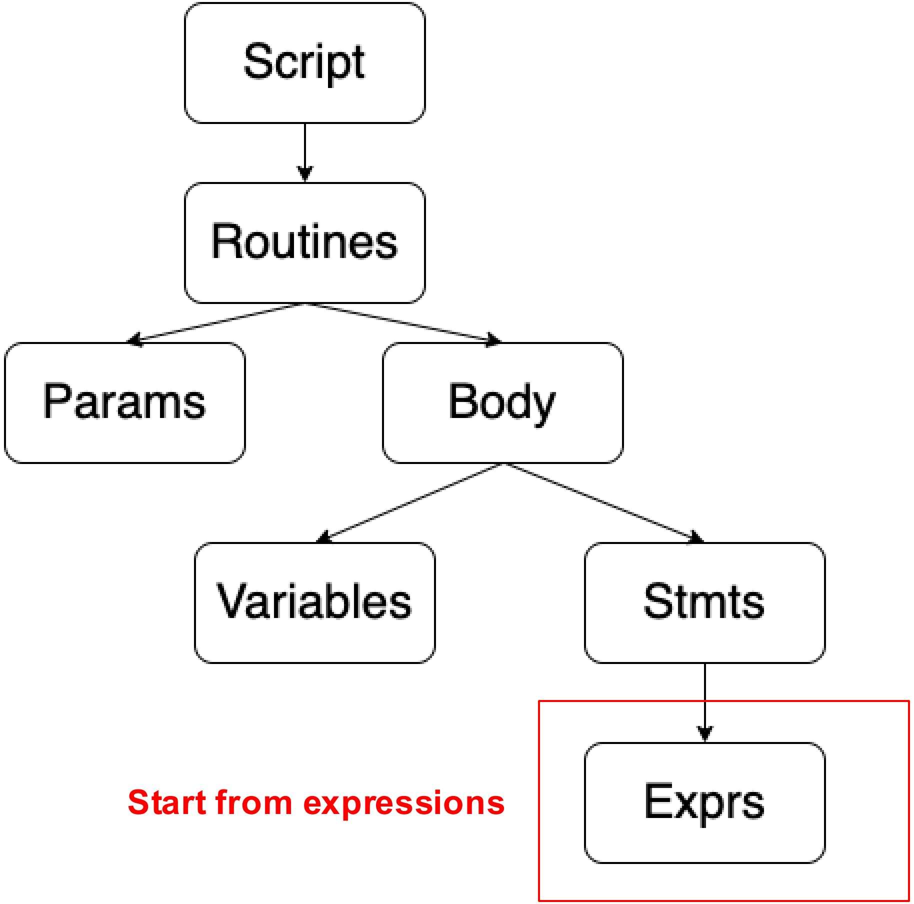
```
Script := Routines Main
Routines := Routine Routines
    | λ
Routine := ID Ins Outs start Body end
Ins := in : ID ( Type ) Ins
    | λ
Outs := out : ID ( Type ) Outs
    | λ
Type := number
    | string
    | flag
Main := main start Body end
Body := Variables Statements
Variables := Variable Variables
    | λ
Variable := ID ( Type ) :)
Statements := Statement Statements
    | λ
Statement := ID <- Expression :)
    | read ( ID ) :)
    | write ( ID ) :)
    | call ID ( IdList ) :)
    | when FlagExpression do Statements done :)
```

```
IdList := ID IdList
    | λ
Expression := NumericalExpression
    | StringExpression
    | FlagExpression
NumericalExpression := NUMBER
    | ID
    | NumericalExpression + NumericalExpression
    | NumericalExpression - NumericalExpression
    | NumericalExpression * NumericalExpression
    | NumericalExpression / NumericalExpression
StringExpression := STRING
    | ID
    | StringExpression + StringExpression
FlagExpression := up
    | down
    | ID
    | flip FlagExpression
    | FlagExpression + FlagExpression
    | FlagExpression * FlagExpression
    | NumericalExpression ? NumericalExpression
```

https://github.com/pattersonzUTD/UTDLang–

```
Script := Routines Main
Routines := Routine Routines
    | λ
Routine := ID Ins Outs start Body end
Ins := in : ID ( Type ) Ins
    | λ
Outs := out : ID ( Type ) Outs
    | λ
Type := number
    | string
    | flag
Main := main start Body end
Body := Variables Statements
Variables := Variable Variables
    | λ
Variable := ID ( Type ) :)
Statements := Statement Statements
    | λ
Statement := ID <- Expression :)
    | read ( ID ) :)
    | write ( ID ) :)
    | call ID ( IdList ) :)
    | when FlagExpression do Statements done :)
```

```
IdList := ID IdList
    | λ
Expression := NumericalExpression
    | StringExpression
    | FlagExpression
NumericalExpression := NUMBER
    | ID
    | NumericalExpression + NumericalExpression
    | NumericalExpression - NumericalExpression
    | NumericalExpression * NumericalExpression
    | NumericalExpression / NumericalExpression
StringExpression := STRING
    | ID
    | StringExpression + StringExpression
FlagExpression := up
    | down
    | ID
    | flip FlagExpression
    | FlagExpression + FlagExpression
    | FlagExpression * FlagExpression
    | NumericalExpression ? NumericalExpression
```



Expression := NumericalExpression
| StringExpression
| FlagExpression
NumericalExpression := NUMBER
| ID
| NumericalExpression + NumericalExpression
| NumericalExpression - NumericalExpression
| NumericalExpression * NumericalExpression
| NumericalExpression / NumericalExpression
StringExpression := STRING
| ID
| StringExpression + StringExpression
FlagExpression := up
| down
| ID
| flip FlagExpression
| FlagExpression + FlagExpression
| FlagExpression * FlagExpression
| NumericalExpression ? NumericalExpression

Conflicts and Ambiguity

- Our grammar is ambiguous. The grammar enforces some of the language's type checking. While parsing there is no easy way to identify if an ID is a certain type. Therefore if we see "ID +" there is no way to tell what should be allowed. (Code can be found in example repository at release Part2_Error1)
  - This particular reason doesn't exist in your grammar, however parts of your grammar are ambiguous and will require you to work the production rules around.

Warning : *** Reduce/Reduce conflict found in state #17 between flagExpression ::= ID (*) and numericalExpression ::= ID (*) under symbols: {PLUS, MULTIPLY}
Resolved in favor of the second production.
Warning : *** Reduce/Reduce conflict found in state #7 between flagExpression ::= ID (*) and numericalExpression ::= ID (*) under symbols: {EOF, PLUS, MULTIPLY}
Resolved in favor of the second production.

```
NumericalExpression := NUMBER
    | ID
    | NumericalExpression + NumericalExpression
    | NumericalExpression - NumericalExpression
    | NumericalExpression * NumericalExpression
FlagExpression := up
    | down
    | ID
    | flip FlagExpression
    | FlagExpression + FlagExpression
    | FlagExpression * FlagExpression
    | NumericalExpression ? NumericalExpression
```

```
Expression := NumericalExpression
    | StringExpression
    | FlagExpression
NumericalExpression := NUMBER
    | ID
    | NumericalExpression + NumericalExpression
    | NumericalExpression - NumericalExpression
    | NumericalExpression * NumericalExpression
    | NumericalExpression / NumericalExpression
StringExpression := STRING
    | ID
    | StringExpression + StringExpression
FlagExpression := up
    | down
    | ID
    | flip FlagExpression
    | FlagExpression + FlagExpression
    | FlagExpression * FlagExpression
    | NumericalExpression ? NumericalExpression
```

```
binaryExpression ::= unaryExpression
        |
        binaryExpression + binaryExpression
        |
        binaryExpression:b1 - binaryExpression:b2
        |
        binaryExpression:b1 * binaryExpression:b2
        |
        binaryExpression:b1 / binaryExpression:b2
        |
        binaryExpression:b1 ? binaryExpression:b2

unaryExpression ::= operandExpression
        |
        FLIP binaryExpression
operandExpression ::= ID
        |
        UP
        |
        DOWN
        |
        NUMBERLIT
        |
        STRINGLIT
```

grammar.cup - First Rules

- Let's start with Operand Expressions
- We need to turn this into production rules
  - Production Rules have the following format

```
nonTerminal ::= production rule
        {: JavaCode semantic action :}
        | alternative production rule(s)
        {:corresponding JavaCode semantic action:}
        ; //end rule for non terminal with a ;

//Each nonTerminal will have a value, this assigned with the
following  RESULT = …
```

```
operandExpression ::= ID
             |
             UP
             |
             DOWN
             |
             NUMBERLIT
             |
             STRINGLIT
```

grammar.cup - First Rules

- Let’s start with Operand Expressions

```
operandExpression ::= ID
              |
              UP
              |
              DOWN
              |
              NUMBERLIT
              |
              STRINGLIT
```

```
operandExpression ::= UP //We can associate tokens with a variable
            {: RESULT = new OperandExpr(true); :} //we can then use reference these in the code
            |  ID:i
            {: RESULT = new OperandExpr(i, “var”); :}
            ;

 //This is good enough for now, let’s see if we can print out this code.
```

grammar.cup - First Rules

```
operandExpression ::= ID
                |
                UP
                |
                DOWN
                |
                NUMBERLIT
                |
                STRINGLIT
```

```
//Last step, gotta create non terminal objects and specify types
…
terminal UP; //31

non terminal Program program;
non terminal OperandExpr operandExpression;
```

```
//Need to make this a possible from the top of the program!
        program ::= operandExpression:o
        {:RESULT = new Program(o); :}
        ;
        operandExpression ::= UP //We can associate tokens with a variable
        {: RESULT = new OperandExpr(true); :} //we can then use reference these in the code
        |  ID:i
        {: RESULT = new OperandExpr(i, “var”); :}
        ;

//This is good enough for now, let’s see if we can print out this code.
```

First Rules - creating more files

- We have now referenced two new types that don’t exist
  - OperandExpr - operand expressions
  - Program - our program base
- Let’s create these files

```java
class OperandExpr extends Token {
  String type;
  String value;

  public OperandExpr(boolean f) {
    value = String.valueOf(f);
    type = "flag";
  }

  public OperandExpr(String s, String t) {
    value = s;
    type = t;
  }

  public String toString(int t) {
    return getTabs(t) + type + ":" + value;
  }
}
```

```java
class Program extends Token {
  private OperandExpr operandExpr;
  //Constructor
  public Program(OperandExpr o) {
    operandExpr = o;
  }

  @Override
  public String toString(int t) {
    return "Program:\n" + operandExpr.toString(t+1) + "\n";
  }
}
```

First Rules - creating ... fi...

```
//Don't forget to update the makefile!!!
FILE=         Lexer.java parser.java
sym.java\
  LexerTest.java ScannerTest.java Token.java Program.java OperandExpr.java
...
sampleFile.utd: all
    $(JAVA) -cp $(CP) ScannerTest sampleFile.utd > sampleFile-
    output.txt  cat sampleFile.utd
    cat -n sampleFile-output.txt
```

- We have now referenced
  - OperandExpr - operand ex
  - Program - our program ba
- Let's create these files

```java
class OperandExpr extends Token {
  String type;
  String value;

  public OperandExpr(boolean f) {
    value = String.valueOf(f);
    type = "flag";
  }

  public OperandExpr(String s, String t) {
    value = s;
    type = t;
  }

  public String toString(int t) {
    return getTabs(t) + type + ":" + value;
  }
}
```

```java
class Program extends Token {
  private OperandExpr operandExpr;
  //Constructor
  public Program(OperandExpr o) {
    operandExpr = o;
  }

  @Override
  public String toString(int t) {
    return "Program:\n" + operandExpr.toString(t+1) + "\n";
  }
}
```

Examples:

If any weird error occur, try make clean and re-run it.

Sample 1:

true

```
java -cp .:../java-cup-11b.jar ScannerTest sampleFile_test.utd > sampleFile_test-output.txt
cat sampleFile_test.utd
truecat -n sampleFile_test-output.txt
     1  Program:
     2          var:true
```

```
java -cp .:../java-cup-11b.jar ScannerTest sampleFile_test.utd >
sampleFile_test-output.txt
cat sampleFile_test.utd
applecat -n sampleFile_test-output.txt
     1  Program:
     2          var:apple
```

Sample 2:

apple

Complete OperandExpression

```
operandExpression ::= ID:i
            {: RESULT = new OperandExpr(i, "var"); :}
            |
            UP
            {: RESULT = new OperandExpr(true); :}
            |
            DOWN
            {: RESULT = new OperandExpr(false); :}
            |
            NUMBERLIT:n
            {: RESULT = new OperandExpr(n); :}
            |
            STRINGLIT:s
            {: RESULT = new OperandExpr(s, "strLit"); :}
```

Revisit OperandExpression

```java
class OperandExpr extends Token {
  String type;
  String value;

  public OperandExpr(int n) {
    value = String.valueOf(n);
    type = "number";
  }

  public OperandExpr(boolean f) {
    value = String.valueOf(f);
    type = "flag";
  }
```

```java
  public OperandExpr(String s, String t) {
    value = s;
    type = t;
  }

  public String toString(int t) {
    return getTabs(t) + type + ":" + value;
  }
}
```

Updated Expression Grammar

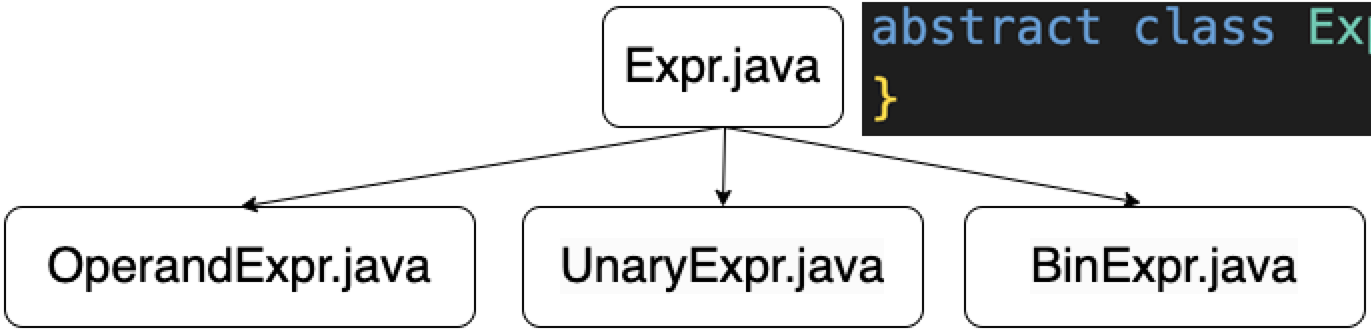
```
program ::= binaryExpression:e
  {: RESULT = new Program(e); :}
  ;

binaryExpression ::= unaryExpression:u
                 {: RESULT = u; :}
                 |
                 binaryExpression:b1 PLUS binaryExpression:b2
                 {: RESULT = new BinExpr(b1, "+", b2); :}
                 |
                 binaryExpression:b1 MINUS binaryExpression:b2
                 {: RESULT = new BinExpr(b1, "-", b2); :}
                 |
                 binaryExpression:b1 MULTIPLY binaryExpression:b2
                 {: RESULT = new BinExpr(b1, "*", b2); :}
                 |
                 binaryExpression:b1 DIVIDE binaryExpression:b2
                 {: RESULT = new BinExpr(b1, "/", b2); :}
                 |
                 binaryExpression:b1 QUESTION binaryExpression:b2
                 {: RESULT = new BinExpr(b1, "?", b2); :}
                 ;
```

```
unaryExpression ::= operandExpression:o
                    {: RESULT = o; :}
                    |
                FLIP binaryExpression:b
                    {: RESULT = new UnaryExpr("flip", b); :}
                    ;

operandExpression ::= ID:i
                 {: RESULT = new OperandExpr(i, "var"); :}
                 |
                 UP
                 {: RESULT = new OperandExpr(true); :}
                 |
                 DOWN
                 {: RESULT = new OperandExpr(false); :}
                 |
                 NUMBERLIT:n
                 {: RESULT = new OperandExpr(n); :}
                 |
                 STRINGLIT:s
                 {: RESULT = new OperandExpr(s, "strLit"); :}
                 ;
```

Create Expression Objects




```
abstract class Expr extends Token {
}
```

Create Expression Objects

```java
class OperandExpr extends Expr {
  String type;
  String value;

  public OperandExpr(int n) {
    value = String.valueOf(n);
    type = "number";
  }

  public OperandExpr(boolean f) {
    value = String.valueOf(f);
    type = "flag";
  }

  public OperandExpr(String s, String t) {
    value = s;
    type = t;
  }
```

```java
abstract class Expr extends Token {
}
```

BinExpr.java

Create Expression Objects

```java
abstract class Expr extends Token {
}
```

```java
class OperandExpr extends Expr {
  String type;
  String value;

  public Ope
    value =
    type = '
  }

  public Ope
    value =
    type = '
  }

  public Ope
    value =
    type = t;
  }
```

```java
class UnaryExpr extends Expr {
  String  operator;
  Expr rhs;

  public UnaryExpr(String op, Expr e2) {
    operator = op;
    rhs = e2;
  }

  public String toString(int t) {
    return getTabs(t) + operator + "(" + rhs.toString(t:0) + ")";
  }
}
```

Create Expression Objects

```
class OperandExpr extends Expr {
  String type;
  String value;

  public Op
    value =
    type =
  }

  public Op
    value =
    type =
  }

  public Op
    value =
    type =
  }
```

```
class UnaryExpr extends Expr {
  String  operator;
```

```
class BinExpr extends Expr {
  Expr lhs;
  String  operator;
  Expr rhs;

  public BinExpr(Expr e1, String op, Expr e2) {
    lhs = e1;
    operator = op;
    rhs = e2;
  }

  public String toString(int t) {
    return getTabs(t) + "(" + lhs.toString(t:0) + " " + operator + " " + rhs.toString(t:0) + ")";
  }
}
```

```
abstract class Expr extends Token {
}
```

C

Shift/Reduce Conflict

make parserD.java 2> dump-output.txt

```
Warning : *** Shift/Reduce conflict found in state #52
between binaryExpression ::= binaryExpression PLUS binaryExpression (*)
and     binaryExpression ::= binaryExpression (*) MINUS binaryExpression
under symbol MINUS
Resolved in favor of shifting.

Warning : *** Shift/Reduce conflict found in state #52
between binaryExpression ::= binaryExpression PLUS binaryExpression (*)
and     binaryExpression ::= binaryExpression (*) MULTIPLY binaryExpression
under symbol MULTIPLY
Resolved in favor of shifting.

Warning : *** Shift/Reduce conflict found in state #52
between binaryExpression ::= binaryExpression PLUS binaryExpression (*)
and     binaryExpression ::= binaryExpression (*) DIVIDE binaryExpression
under symbol DIVIDE
Resolved in favor of shifting.
```

Sample:

12 * 13 + 2

Precedence

```
non terminal Program program;
non terminal BinExpr binaryExpression;
non terminal UnaryExpr unaryExpression;
non terminal OperandExpr operandExpression;

precedence left PLUS, MINUS; //After non terminals and before production rules we can specify
precedence left MULTIPLY, DIVIDE;//A resolution order can be given, resolved from bottom to
top

program ::= binaryExpression:e
  {: RESULT = new Program(e); :}
  ;

binaryExpression ::= unaryExpression:u
                 {: RESULT = u; :}
                 |
………
```

Precedence Highlight

cat sampleFile2.utd

"string" * up / down ? var + 8

cat -n sampleFile2-output.txt

—---------------Question mark has highest precedence

```
precedence left PLUS, MINUS;
precedence left MULTIPLY, DIVIDE;
precedence left QUESTION;
```

1 Program:

2 (((strLit:"string" * flag:true) / (flag:false ? var:var)) + number:8)

—---------------Question mark has lowest precedence

1 Program:

2 (((strLit:"string" * flag:true) / flag:false) ? (var:var + number:8))

```
precedence left QUESTION;
precedence left PLUS, MINUS;
precedence left MULTIPLY, DIVIDE;
```

Results

java -cp .:./java-cup-11b.jar ScannerTest sampleFile1.utd > sampleFile1-output.txt cat sampleFile1.utd

varId + 5 * flip 6 / 7 - 2

cat -n sampleFile1-output.txt

1 Program:

2   (var:varId + (number:5 * flip(((number:6 / number:7) - number:2))))

java -cp .:./java-cup-11b.jar ScannerTest sampleFile2.utd > sampleFile2-output.txt cat sampleFile2.utd

"string" * up / down ? var + 8

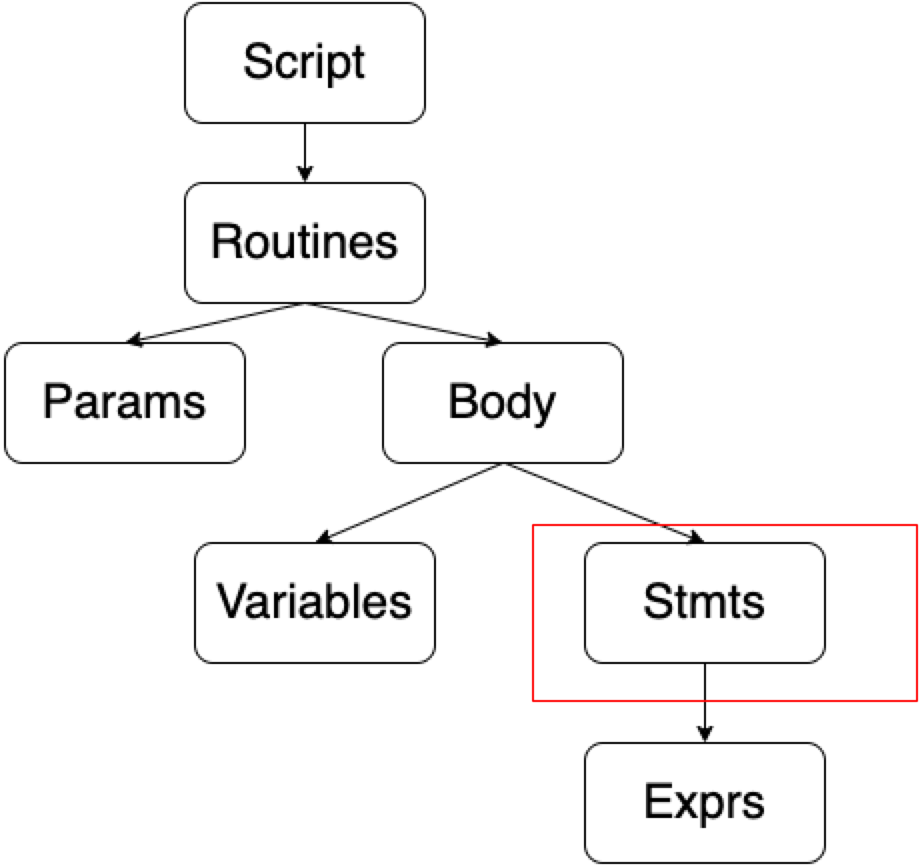
cat -n sampleFile2-output.txt

1 Program:

2   (((strLit:"string" * flag:true) / (flag:false ? var:var)) + number:8)

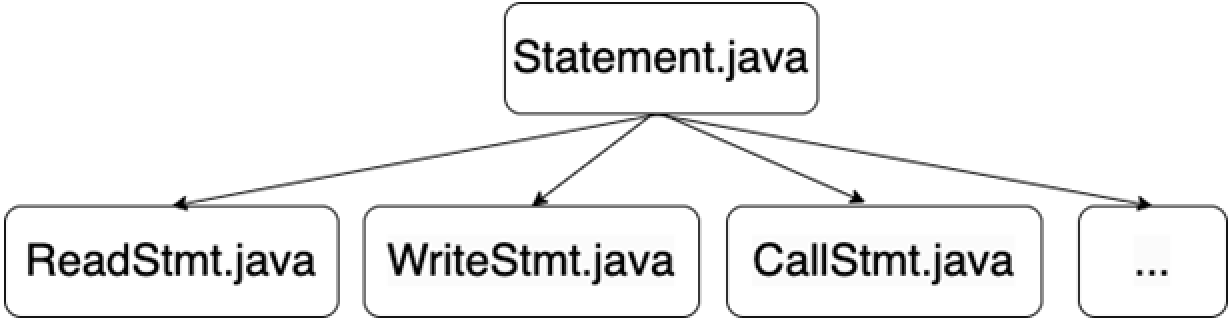
Adding All Statements

Program := Statements
Statements := Statement Statements
| λ
Statement := ID <- Expression :)
| read (ID) :)
| write (ID) :)
| call ID (IdList) :)
| when FlagExpression do Statements done :)
IdList := ID IdList
| λ



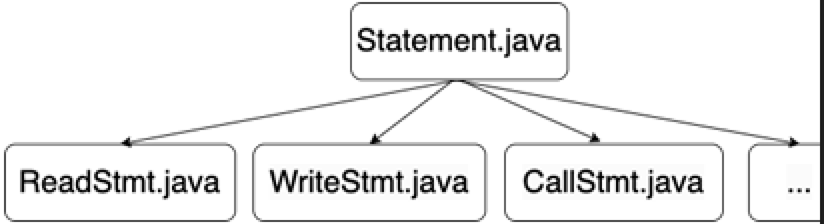
Adding All Statements

Program := Statements
Statements := Statement Statements
| λ
Statement := ID <- Expression :)
| read (ID) :)
| write (ID) :)
| call ID (IdList) :)
| when FlagExpression do Statements done :)
IdList := ID IdList
| λ



Adding All Statements

Program := Statements
Statements := Statement Statements
| λ
Statement := ID <- Expression :)
| read (ID) :)
| write (ID) :)
| call ID (IdList) :)
| when FlagExpression do Statements done :)
IdList := ID IdList
| λ




```java
import java.util.List;
import java.util.LinkedList;

class StatementList extends Token {
  private List<Statement> statements;

  public StatementList() {
    statements = new LinkedList<Statement>();
  }

  public StatementList prependStatement(Statement s) {
    statements.add(index:0,s);
    return this;
  }

  public String toString(int t) {
    String ret = "";
    for (Statement s : statements) {
      ret += s.toString(t);
    }
    return ret;
  }
}
```

All Statements

```
java -cp .:../java-cup-11b.jar ScannerTest sampleFile1.utd > sampleFile1-output.txt  cat sampleFile1.utd
a <- 5 :) b <- a + 10 :) c <- up:)d<-flip
c:)  write(a):)read(b):)
call e(a b c):)
call f():)
when c ? d do a <- 20:) done:)
cat -n sampleFile1-output.txt
     1 Program:
     2         a <- number:5 :)
     3         b <- (var:a + number:10) :)
     4         c <- flag:true :)
     5         d <- flip(var:c) :)
     6         write(a) :)
     7         read(b) :)
     8         call e(a b c ) :)
     9         call f() :)
    10         when (var:c ? var:d) do
    11                 a <- number:20 :)
    12         done :)
```

There unfortunately there is not time to go over everything, you can find a checkpoint for my code here: https://github.com/pattersonzUTD/UTDLang--/releases/tag/Part2_AllStatements_Checkpoint

Add in Declarations

Program := Body

Body := Variables Statements

Variables := Variable

Variables

| λ

Variable := ID (Type)

:) Type := number

| string

| flag

Easy Right?

Updated Grammar

```
program ::= body:b
       {: RESULT = new Program(b); :} ;
body ::= variables:v statements:s
     {: RESULT = new Body(v,s); :} ;
variables ::= variable:v variables:vs
          {: RESULT = vs.prependVariable(v); :}
          | {: RESULT = new VariableList(); :} ;
variable ::= ID:i LPAREN type:t RPAREN SMILE
        {: RESULT = new Variable(i,t); :} ;
Type ::= NUMBER:n
     {: RESULT = n; :}
     |              STRING:s
     {: RESULT = s; :}
     | FLAG:f
     {: RESULT = f; :} ;
statements ::= statement:s statements:ss
     {: RESULT = ss.prependStatement(s); :}
     | {: RESULT = new StatementList(); :};
```

Results…. Kind of

Warning : * Shift/Reduce conflict found in state**
#4 between variables ::= (*)
and   variable ::= (*) ID LPAREN type RPAREN
SMILE under symbol ID
Resolved in favor of shifting.

Warning : * Shift/Reduce conflict found in state**
#0 between variables ::= (*)
and   variable ::= (*) ID LPAREN type RPAREN
SMILE under symbol ID
Resolved in favor of shifting.

Project Checkpoint: https://github.com/pattersonzUTD/UTDLang--/releases/tag/Part2_ShiftReduce

Solving a Conflict
Step 1: Identify the Cause

Warning : * Shift/Reduce conflict found in state**

#4 between variables ::= (*)

and   variable ::= (*) ID LPAREN type RPAREN

under symbol ID

Take note

State Number (4)

Symbol (ID)

Solving a Conflict
Step 1: Identify the Cause

State Number (4)

Symbol (ID)

Now we use the command

make parserD.java

This will print a list of all our states anc

and symbols. If this is too long to scro

make parserD.java 2> dump-ou

And then look at that file

```
transition on routines to state [4]
transition on program to state [3]
transition on routine to state [2]
transition on ID to state [1]

-------------------
lalr_state [1]: {
  [ins ::= (*) IN COLON ID LPAREN type RPAREN ins , {START OUT }]
  [routine ::= ID (*) ins outs START body END , {MAIN ID }]
  [ins ::= (*) , {START OUT }]
}
transition on ins to state [75]
transition on IN to state [74]

-------------------
lalr_state [2]: {
  [routines ::= routine (*) routines , {MAIN }]
  [routines ::= (*) routine routines , {MAIN }]
  [routine ::= (*) ID ins outs START body END , {MAIN ID }]
  [routines ::= (*) , {MAIN }]
}
transition on routines to state [73]
transition on routine to state [2]
transition on ID to state [1]
```

Solving a Conflict
Step 1: Identify the Cause

State Number (4)

Symbol

(ID)

Scroll to the State in Question

lalr_state [4]: {
  [variable ::= (*) ID LPAREN type RPAREN SMILE , {EOF READ WRITE CALL WHEN ID
  }] [variables ::= (*) , {EOF READ WRITE CALL WHEN ID }]
  [variables ::= variable (*) variables , {EOF READ WRITE CALL WHEN ID
  }] [variables ::= (*) variable variables , {EOF READ WRITE CALL WHEN
  ID }]
}

transition on variables to state [56]

transition on variable to state [4]

transition on ID to state [2]

Solving a Conflict
Step 1: Identify the Cause

State Number (4)

Symbol

(ID)

Recall we had a shift reduce on ID
This means we need to look for a spot where we could shift or reduce on ID

[variable ::= (*) ID LPAREN type RPAREN SMILE , {EOF READ WRITE CALL WHEN ID }]
[variables ::= (*) , {EOF READ WRITE CALL WHEN ID }]

Now we ask:

"Why does variables reduce on ID?

Check your grammar:

variables Is followed by statements in body | body ::= variables:v statements:s
statements can start with statement | statements ::= statement:s statements:ss
statement can start with ID | statement ::= ID:i ASSIGN binaryExpression:b SMILE

Solving a Conflict
Step 2: Lift Production Rules

- If we make the language no longer reduce in that spot, it turns to a Shift/Shift, which is not a conflict.
  - To do this we take non-terminals in our rules and expand them by their possible definitions

```
variables ::= variable:v variables:vs
        |
        ;
variable ::= ID:i LPAREN type:t RPAREN SMILE
        ;
```

Solving a Conflict
Step 2: Lift Production Rules

- If we make the language no longer reduce in that spot, it turns to a Shift/Shift, which is not a conflict.
  - To do this we take non-terminals in our rules and expand them by their possible definitions

```
variables ::= ID:i LPAREN type:t RPAREN SMILE variables:vs
        |
        ;
```

Solving a Conflict
Step 2: Lift Production Rules

- If we make the language no longer reduce in that spot, it turns to a Shift/Shift, which is not a conflict.
  - To do this we take non-terminals in our rules and expand them by their possible definitions

```
variables ::= ID:i LPAREN type:t RPAREN SMILE variables:vs
        |
        ;
```

- We do this for all points of conflict, and continue to do so until there are no more.
  - Statement -> statements
  - Statements & variables -> body

- WARNING: This gets verbose.

Solving a Conflict
Step 2: Lift Production Rules

Note: We can temporarily make all types of non terminals strings and return empty strings. (Change Program to take a string)

```
Body ::= variables:v statements:s
            {: RESULT = “”; :};
variables ::= ID:i LPAREN type:t RPAREN SMILE variables:vs
            {: RESULT = “”; :}
         |              ;
statements ::= ID:i ASSIGN binaryExpression:b SMILE statements:ss
         {: RESULT = “”; :}
         |
         READ LPAREN ID:i RPAREN SMILE statements:ss
         {: RESULT = “”; :}
         |
         WRITE LPAREN ID:i RPAREN SMILE statements:ss
         {: RESULT = “”; :}
         |
         CALL ID:i LPAREN idList:is RPAREN SMILE statements:ss
         {: RESULT = “”; :}
         |
         WHEN binaryExpression:b DO statements:s DONE SMILE statements:ss
         {: RESULT = “”; :}
         |
         {: RESULT = “”; :};
```

Solving a Conflict
Step 2: Lift Production Rules

```
non terminal Program program;
non terminal Expr binaryExpression;
non terminal Expr unaryExpression;
non terminal OperandExpr operandExpression;
non terminal String statements;
//non terminal Statement statement;
non terminal IDList idList;
non terminal String type;
//non terminal Variable
variable;  non terminal String
variables;  non terminal String
body;
```

Solving a Conflict
Step 2: Lift Production Rules

```
Warning : *** Shift/Reduce conflict found in state #62
  between variables ::= (*)
  and          variables ::= (*) ID LPAREN type RPAREN SMILE
  variables  under symbol ID
  Resolved in favor of shifting.
```

Rats… We need to continue flattening
How do we flatten an empty production rule?
Or recursive rules?

Solving a Conflict
Step 2: Lift Production Rules

```
body ::= variables:v statements:s
        ;

variables ::= ID:i LPAREN type:t RPAREN SMILE
variables:vs
        |
        ;
```

Solving a Conflict
Step 2: Lift Production Rules

```
body ::= ID:i LPAREN type:t RPAREN SMILE variables:vs
statements:s
        |
        statements:s
        ;
```

Hmm… This isn't quite right. We can't reference variables.
Replace with call to body

Solving a Conflict
Step 2: Lift Production Rules

```
body ::= ID:i LPAREN type:t RPAREN SMILE body:b
statements:s
        |
        statements:s
        ;
```

How about now?

Solving a Conflict
Step 2: Lift Production Rules

```
body ::= ID:i LPAREN type:t RPAREN SMILE body:b ID:i ASSIGN binaryExpression:b SMILE body:b
    {: RESULT = ""; :}
    |       ID:i LPAREN type:t RPAREN SMILE body:b READ LPAREN ID:i RPAREN SMILE body:b
    {: RESULT = ""; :}
    |       ID:i LPAREN type:t RPAREN SMILE body:b WRITE LPAREN ID:i RPAREN SMILE body:b
    {: RESULT = ""; :}
    |       ID:i LPAREN type:t RPAREN SMILE body:b CALL ID:i LPAREN idList:is RPAREN SMILE body:b
    {: RESULT = ""; :}
    |       ID:i LPAREN type:t RPAREN SMILE body:b WHEN binaryExpression:b DO statements:s DONE
    SMILE body:b
    {: RESULT = ""; :}
    |       ID:i LPAREN type:t RPAREN SMILE body:b
    {: RESULT = ""; :}
    |       ID:i ASSIGN binaryExpression:b SMILE body:b
    {: RESULT = ""; :}
    | READ LPAREN ID:i RPAREN SMILE body:b
    {: RESULT = ""; :}
    |       WRITE LPAREN ID:i RPAREN SMILE body:b
    {: RESULT = ""; :}
    |       CALL ID:i LPAREN idList:is RPAREN SMILE body:b
    {: RESULT = ""; :}
    |       WHEN binaryExpression:b DO statements:s DONE SMILE body:b
    {: RESULT = ""; :}
    |
    {: RESULT = ""; :};
```

Results

```
body ::= ID:i LPAREN type:t RPAREN SMILE body:b ID:i ASSIGN binaryExpression:b SMILE
body:b
 {: RESULT = ""; :}
|
ID:i ASSIGN binaryExpression:b SMILE body:b
    {: RESULT = ""; :}
```

Notice how we have production rules for body as follow ups for body? We can cut those out. By just putting body at the end since they are redundant interpretations.

Results

```
body ::= ID:i LPAREN type:t RPAREN SMILE body:b
    {: RESULT = ""; :}
    |
    ID:i ASSIGN binaryExpression:be SMILE body:b
    {: RESULT = ""; :}
    |
    READ LPAREN ID:i RPAREN SMILE body:b
    {: RESULT = ""; :}
    |
    WRITE LPAREN ID:i RPAREN SMILE body:b
    {: RESULT = ""; :}
    |
    CALL ID:i LPAREN idList:is RPAREN SMILE body:b
    {: RESULT = ""; :}
    |
    WHEN binaryExpression:b DO body:bb DONE SMILE body:bo //temporarily make the body body for simplicity. We will address this in a bit.
    {: RESULT = ""; :}
    |
    {: RESULT = ""; :}
    ;
```

MAKE SURE ALL TOKEN REFERENCES HAVE UNIQUE NAMES ie, binearyExpression, body and body can't all be named b

```
kisamefishfry@Lemillion:~/TA/UTDLang~$ make
java -jar jflex-full-1.8.2.jar tokens.jflex
Reading "tokens.jflex"
Constructing NFA : 190 states in NFA
Converting NFA to DFA :
.................................................................................
83 states before minimization, 76 states in minimized DFA
Writing code to "Lexer.java"
java -jar ./java-cup-11b.jar -interface < grammar.cup
Warning : LHS non terminal "Type" has not been declared
Warning : Terminal "FLAG" was declared but never used
Warning : Terminal "IN" was declared but never used
Warning : Terminal "END" was declared but never used
Warning : Terminal "COLON" was declared but never used
Warning : Terminal "STRING" was declared but never used
Warning : Terminal "NUMBER" was declared but never used
Warning : Terminal "MAIN" was declared but never used
Warning : Terminal "START" was declared but never used
Warning : Terminal "OUT" was declared but never used
Warning : Non terminal "statements" was declared but never used
Warning : Non terminal "variables" was declared but never used
------- CUP v0.11b 20160615 (GIT 4ac7450) Parser Generation Summary -------
  0 errors and 12 warnings
  32 terminals, 9 non-terminals, and 24 productions declared,
  producing 61 unique parse states.
  11 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
  Code written to "parser.java", and "sym.java".
---------------------------------------------------- (CUP v0.11b 20160615 (GIT 4ac7450))
javac -cp .:./java-cup-11b.jar Lexer.java
Note: Lexer.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
javac -cp .:./java-cup-11b.jar parser.java
javac -cp .:./java-cup-11b.jar LexerTest.java
javac -cp .:./java-cup-11b.jar ScannerTest.java
Note: ScannerTest.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
javac -cp .:./java-cup-11b.jar AssignmentStmt.java
javac -cp .:./java-cup-11b.jar ReadStmt.java
javac -cp .:./java-cup-11b.jar WriteStmt.java
javac -cp .:./java-cup-11b.jar CallStmt.java
javac -cp .:./java-cup-11b.jar WhenStmt.java
make: *** No rule to make target 'Type.class', needed by 'all'.  Stop.
```

SUCCESS

Final Step

- Does the new implementation cause any bugs?
    - Yes
    - We no longer enforce that all variables are declared first

- We need to be creative in our solution
    - Logic
        - A variables is followed by either a list of variables or statements
        - A statement can only be followed by a statement
            - **Note: There is no easy way to come up with these.**
                - **Solve your conflict first, and then I can help you here**

Final Solution

```
body ::= varAndStatementList:v
    {: RESULT = new Body(v); :}
    ;
varAndStatementList ::= variable:v varAndStatementList:isl
                {: RESULT = isl.prepend(v); :}
                |
                statement:s statements:ss
                {: RESULT = ss.prepend(s); :}
                |
                {: RESULT = new VarStmtList(); :}
                ;
variable ::= ID:i LPAREN type:t RPAREN SMILE
        {: RESULT = new Variable(i,t); :}
        ;
statements ::= statement:s statements:ss
          {: RESULT = ss.prepend(s); :}
          |
          {: RESULT = new VarStmtList(); :}
          ;
```

```java
import java.util.List;
import java.util.LinkedList;

class VarStmtList extends Token {
  private List<Statement> stmtVar;

  public VarStmtList() {
    stmtVar = new LinkedList<Statement>();
  }

  public VarStmtList prepend(Statement s) {
    stmtVar.add(index:0,s);
    return this;
  }

  public String toString(int t) {
    String ret = "";
    for (Statement s : stmtVar) {
      ret += s.toString(t);
    }
    return ret;
  }
}
```

Final Results

```
cat sampleFile2.utd
num (number) :)
guess (number) :)
guessPP (number) :)
approx (flag) :)
strTest (string) :)
     call nextGuess (num) :)
     call guessPP (num) :)
     when guess * guess ? value do
          nextGuess <- guess - 1 :)
          call sqrtLoop(value nextGuess result approx) :)
     done :)

     guessPP <- guess + 1 :)

     when value ? guess * guess * flip guessPP * guessPP ? value do
          call sqrtLoop(value guessPP result) :)
     done :)

     when value ? guess * guess * guessPP * guessPP ? value do
          result <- guess :)
          approx <- up :)
          when guess * guess ? value do
               approx <- down :)
          done :)
     done :)
```

Final Results

```
cat -n sampleFile2-output.txt
     1  Program:
     2          Body:
     3                  num (number) :)
     4                  guess (number) :)
     5                  guessPP (number) :)
     6                  approx (flag) :)
     7                  strTest (string) :)
     8                  call nextGuess(num ) :)
     9                  call guessPP(num ) :)
    10                  when (var:guess * (var:guess ? var:value)) do
    11                          nextGuess <- (var:guess - number:1) :)
    12                          call sqrtLoop(value nextGuess result approx ) :)
    13                  done :)
    14                  guessPP <- (var:guess + number:1) :)
    15                  when (((var:value ? var:guess) * var:guess) * flip((var:guessPP * (var:guessPP ? var:value))) do
    16                          call sqrtLoop(value guessPP result ) :)
    17                  done :)
    18                  when ((((var:value ? var:guess) * var:guess) * var:guessPP) * (var:guessPP ? var:value)) do
    19                          result <- var:guess :)
    20                          approx <- flag:true :)
    21                          when (var:guess * (var:guess ? var:value)) do
    22                                  approx <- flag:false :)
    23                          done :)
    24                  done :)
    25
```

End of Language

- Complete the rest of the grammar

- Release:
  - https://github.com/pattersonzUTD/UTDLang--/releases/tag/Part2_Complete