Ayan Ahmad K19002255 21-11-2021

6CCS3CFL-CW3

Q1

Grammer:

$Stmts ::= Stmt; Stmts \mid Stmt$
$Block ::= \{Stmts\} | Stmt$

$AExp ::= Te + AExp \mid Te - AExp \mid Te$
$Te ::= Fa * Te \mid Fa \ / \ Te | Fa \ \% \ Te \mid Fa$
$Fa ::= (AExp) \mid num \mid id$

$Stmt := skip|$
$id := \ Aexp|$
$write\ (id) \mid$
$write\ (string) \mid$
$write\ id \mid$
$write\ string \mid$
$read\ (id) \mid$
$read\ id \mid$
$if\ BExp\ then\ Block\ else\ Block \mid$
$while\ BExp\ do\ Block$
Type equation here.
$BExp := AExp == \ AExp|$
$AExp \ != AExp|$
$AExp < AExp|$
$AExp \leq AExp|$
$AExp > AExp|$
$AExp \geq AExp|$
$(BExp\ ) \ \&\& \ BExp|$
$(BExp) \ || \ BExp|$
$true|$
$false|$
$(BExp)$

For making my parser recognise tokens, I had to change the implementation of a few essential functions.

I introduced a few new implicit definitions which correspond to
`Parser[List[Token], Token]`
And also introduced some atomic parsers for some specific tokens.

Boolen Expressions

ayan.ahmad@kcl.ac.uk

```scala
// boolean expressions with some simple nesting
lazy val BExp: Parser[List[Token], BExp] =
   (AExp ~ T_OP("==") ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("==", x, z) } ||
   (AExp ~ T_OP("!=") ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("!=", x, z) } ||
   (AExp ~ T_OP("<") ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("<", x, z) } ||
   (AExp ~ T_OP(">") ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop(">", x, z) } ||
   (AExp ~ T_OP(">=") ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop(">=", x, z)} ||
   (AExp ~ T_OP("<=") ~ AExp).map[BExp]{ case x ~ _ ~ z => Bop("<=", x, z)} ||
   (T_LPAREN_N ~ BExp ~ T_RPAREN_N ~ T_OP("&&") ~ BExp).map[BExp]{ case _ ~ y ~ _ ~ _ ~ v => And(y, v) } ||
   (T_LPAREN_N ~ BExp ~ T_RPAREN_N ~ T_OP("||") ~ BExp).map[BExp]{ case _ ~ y ~ _ ~ _ ~ v => Or(y, v) } ||
   (T_KWD("true").map[BExp]{ _ => True }) ||
   (T_KWD("false").map[BExp]{ _ => False }) ||
   (T_LPAREN_N ~ BExp ~ T_RPAREN_N).map[BExp]{ case _ ~ x ~ _ => x }
```

Arithmetic Expressions

```scala
// arithmetic expressions
lazy val AExp: Parser[List[Token], AExp] =
    (Te ~ T_OP("+") ~ AExp).map[AExp]{ case x ~ _ ~ z => Aop("+", x, z) } ||
    (Te ~ T_OP("-") ~ AExp).map[AExp]{ case x ~ _ ~ z => Aop("-", x, z) } ||
    Te

lazy val Te: Parser[List[Token], AExp] =
    (Fa ~ T_OP("*") ~ Te).map[AExp]{ case x ~ _ ~ z => Aop("*", x, z) } ||
    (Fa ~ T_OP("/") ~ Te).map[AExp]{ case x ~ _ ~ z => Aop("/", x, z) } ||
    (Fa ~ T_OP("%") ~ Te).map[AExp]{ case x ~ _ ~ z => Aop("%", x, z) } ||
    Fa

lazy val Fa: Parser[List[Token], AExp] =
    (T_LPAREN_N ~ AExp ~ T_RPAREN_N).map{ case _ ~ y ~ _ => y } ||
    IdParserToken.map(Var) ||
    NumParserToken.map(Num)
```

Single Statements

Had to introduce extra cases of write since there were various different ways it was implemented in test programs.

```scala
// a single statement
/* we need 4 types of writes here
 write id
 write "string"
 write (id)
 write ("string")
*/
lazy val Stmt: Parser[List[Token], Stmt] =
  ((T_KWD("skip").map[Stmt]{_ => Skip }) ||
   (IdParserToken ~ T_OP(":=") ~ AExp).map[Stmt]{ case x ~ _ ~ z => Assign(x, z) } ||
   (T_KWD("write") ~ T_LPAREN_N ~ IdParserToken ~ T_RPAREN_N).map[Stmt]{ case _ ~ y ~ _ => WriteVar(y) } ||
   (T_KWD("write") ~ StrParserToken).map[Stmt]{ case _ ~ y => WriteStr(y) } ||
   (T_KWD("write") ~ IdParserToken).map[Stmt]{ case _~ y => WriteVar(y)} ||
   (T_KWD("write") ~ T_LPAREN_N ~ StrParserToken ~ T_RPAREN_N).map[Stmt]{ case _ ~ _ ~ y ~ _ => WriteStr(y) } ||
   (T_KWD("read") ~ IdParserToken).map[Stmt]{ case _ ~ y => Read(y) } ||
   (T_KWD("if") ~ BExp ~ T_KWD("then") ~ Block ~ T_KWD("else") ~ Block)
   .map[Stmt]{ case _ ~ y ~ _ ~ u ~ _ ~ w => If(y, u, w) } ||
   (T_KWD("while") ~ BExp ~ T_KWD("do") ~ Block).map[Stmt]{ case _ ~ y ~ _ ~ w => While(y, w) })
```

Compound Statements

```scala
// statements
lazy val Stmts: Parser[List[Token], Block] =
  (Stmt ~ T_SEMI ~ Stmts).map[Block]{ case x ~ _ ~ z => x :: z } ||
  (Stmt.map[Block]{ s => List(s) })
```

Blocks

```scala
// blocks (enclosed in curly braces)
lazy val Block: Parser[List[Token], Block] =
  ((T_LPAREN_C ~ Stmts ~ T_RPAREN_C).map{ case _ ~ y ~ _ => y } ||
   (Stmt.map(s => List(s))))
```

ayan.ahmad@kcl.ac.uk

Q2

Output of running the respective programs:

__________Fib __________

```
HashSet(List(WriteStr("Fib"), Read(n), Assign(minus1,Num(0)),
Assign(minus2,Num(1)),
While(Bop(>,Var(n),Num(0)),List(Assign(temp,Var(minus2)),
Assign(minus2,Aop(+,Var(minus1),Var(minus2))), Assign(minus1,Var(temp)),
Assign(n,Aop(-,Var(n),Num(1))))), WriteStr("Result"), WriteVar(minus2)))
```

__________Loops __________

```
HashSet(List(Assign(start,Num(100)), Assign(x,Var(start)),
Assign(y,Var(start)), Assign(z,Var(start)),
While(Bop(<,Num(0),Var(x)),List(While(Bop(<,Num(0),Var(y)),List(While(Bop(<,Nu
m(0),Var(z)),List(Assign(z,Aop(-,Var(z),Num(1))))), Assign(z,Var(start)),
Assign(y,Aop(-,Var(y),Num(1))))), Assign(y,Var(start)), Assign(x,Aop(-
,Var(x),Num(1)))))))
```

__________Primes __________

```
Set(List(Assign(end,Num(100)), Assign(n,Num(2)),
While(Bop(<,Var(n),Var(end)),List(Assign(f,Num(2)), Assign(tmp,Num(0)),
While(And(Bop(<,Var(f),Aop(+,Aop(/,Var(n),Num(2)),Num(1))),Bop(==,Var(tmp),Num
(0))),List(If(Bop(==,Aop(*,Aop(/,Var(n),Var(f)),Var(f)),Var(n)),List(Assign(tm
p,Num(1))),List(Skip)), Assign(f,Aop(+,Var(f),Num(1))))),
If(Bop(==,Var(tmp),Num(0)),List(WriteVar(n)),List(Skip)),
Assign(n,Aop(+,Var(n),Num(1)))))))
```

__________Collatz __________

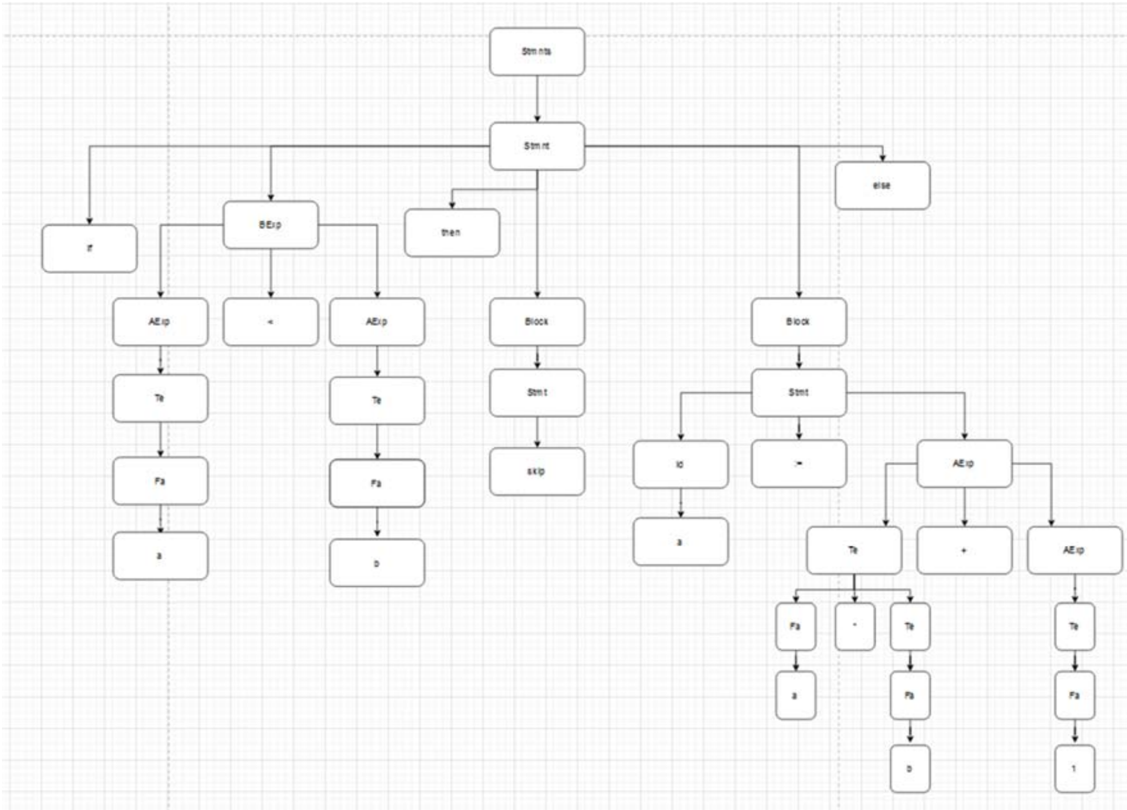
```
Set(List(Assign(bnd,Num(1)),
While(Bop(<,Var(bnd),Num(101)),List(WriteVar(bnd), WriteStr(": "),
Assign(n,Var(bnd)), Assign(cnt,Num(0)),
While(Bop(>,Var(n),Num(1)),List(WriteVar(n), WriteStr(","),
If(Bop(==,Aop(%,Var(n),Num(2)),Num(0)),List(Assign(n,Aop(/,Var(n),Num(2)))),Li
st(Assign(n,Aop(+,Aop(*,Num(3),Var(n)),Num(1))))),
Assign(cnt,Aop(+,Var(cnt),Num(1))))), WriteStr(" => "), WriteVar(cnt),
WriteStr("\n"), Assign(bnd,Aop(+,Var(bnd),Num(1)))))))
```

The tokenised output has \n and quotes since my implementation of the program manipulates strings in the eval_stmt function.

Output for (**if** (a **<** b) **then skip else** a **:=** a * b + 1)

Set(List(If(Bop(<,Var(a),Var(b)),List(Skip),List(Assign(a,Aop(+,Aop(*,Var(a),Var(b)),Num(1))))))

Parse Tree for (**if** (a **<** b) **then skip else** a **:=** a * b + 1) is denoted below.


Stmnts
Stmt
if
BExp
then
Block
else
AExp
<
AExp
Te
Fa
a
Te
Fa
b
Block
Stmt
skip
Block
Stmt
id
a
:=
AExp
Te
+
AExp
Fa
*
Te
a
Fa
b
Te
Fa
1

Q3

The Time measurements are outlined below.

1. **Fibonacci**

```
ibWaiting for User Input....

esult8Code Run Time: 8.0255445 s
```

The formatting of the code is not ideal since there are no newlines in the actual code of the program.

The write function doesn't use **println** instead it uses **print**.

2. **Three Nested Loops**

```
________________
Loop Program - start: 100
Code Run Time: 0.1543181 s
________________
Loop Program - start: 500
Code Run Time: 17.3906449 s
________________
Loop Program - start: 800
Code Run Time: 68.5276533 s
________________
```

Started off with 100 which took less than half a second.
Then 500 took 18 seconds to execute and lastly having a start value of 800 took just a little more than 1 minute.

ayan.ahmad@kcl.ac.uk

3. Factors

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
Code Run Time: 4.2165406 s
```

The formatting of this code is better since when the screenshot was taken, the write function was using **println** to print output. But since it is now using **print**, the new output is denoted below:

```
2357111317192329313741434753596167717379838997Code Run Time: 1.0296003 s
```

4. Collatz

```
50,425,1276,638,319,958,479,1438,719,2158,1079,3238,16
, => 118
98: 98,49,148,74,37,112,56,28,14,7,22,11,34,17,52,26,1
99: 99,298,149,448,224,112,56,28,14,7,22,11,34,17,52,2
100: 100,50,25,76,38,19,58,29,88,44,22,11,34,17,52,26,
Code Run Time: 0.3878173 s
```