Assignment 3

CS314

1 Problem — LL(1) Recursive Descent Parsing

$$
\begin{aligned}
\text{<program>} &::= \text{prog<block>.} \\
\text{<block>} &::= \text{begin <stmtlist> end} \\
\text{<stmtlist>} &::= \text{<stmt> <morestmts>} \\
\text{<morestmts>} &::= \text{; <stmtlist>}|\epsilon \\
\text{<stmt>} &::= \text{<assign>}|\text{<ifstmt>}| \\
&\quad\ \ \text{<repeatstmt>}|\text{<block>} \\
\text{<assign>} &::= \text{<var> = <expr>} \\
\text{<ifstmt>} &::= \text{if <testexpr> then <stmt> else <stmt>} \\
\text{<repeatstmt>} &::= \text{repeat <stmt> until <testexpr>} \\
\text{<testexpr>} &::= \text{<var> <= <expr>} \\
\text{<expr>} &::= +\,\text{<expr> <expr>}| \\
&\quad\ \ -\,\text{<expr> <expr>}| \\
&\quad\ \ *\,\text{<expr> <expr>}| \\
&\quad\ \ \text{<var>}|\text{<digit>} \\
\text{<var>} &::= \text{a} \mid \text{b} \mid \text{c} \\
\text{<digit>} &::= 0 \mid 1 \mid 2
\end{aligned}
$$

1. Show that the grammar above is LL(1). Use a formal argument based on the definition of the LL(1) property.

> **A grammar is LL(1)** if and only if $(A ::= \alpha \text{ and } A ::= \beta)$ implies
>
> $FIRST^+(\alpha) \cap FIRST^+(\beta) = \emptyset$

FIRST(prog <block>) = {prog}
FIRST(begin <stmtlist> end) = {begin}
FIRST(<stmt> <morestmts>) = {a, b, c, if, repeat, begin}
FIRST(; <stmtlist> | ϵ) = {;, ϵ}
FIRST(<stmt>) = {a,b,c, if, repeat, begin}
FIRST(<var> = <expr>) = {a, b, c}
FIRST(<ifstmt>) = {if}
FIRST(<repeatstmt>) = {repeat}
FIRST(<var> <= <expr>) = {a,b,c}
FIRST(<expr>) = {+, -, *, a, b, c, 0, 1, 2}

FOLLOW(<program>) = {end}
FOLLOW(<block>) = {.}
FOLLOW(<stmtlist>) = {end}
FOLLOW(<morestmts>) = {end}
FOLLOW(<stmt>) = {;, else, until, ϵ, +, -, *, a, b, c, 0, 1, 2}
FOLLOW(<assign>) = {;, end}
FOLLOW(<ifstmt>) = {;, end}
FOLLOW(<repeatstmt>) = {;, end}
FOLLOW(<testexpr>) = {;, then, end}
FOLLOW(<expr>) = {+, -, *, a, b, c, 0, 1, 2}
FOLLOW(<var>) = {+, -, *, a, b, c, 0, 1, 2}
FOLLOW(<digit>) = {+, -, *, a, b, c, 0, 1, 2}

2. Show the LL(1) parse table.

	prog	.	begin	end	;	=	if	then	else	repeat	until	<=	=+	-	*	a	b	c	0	1	2	eof
<program>	prog <block>																					
<block>			begin <stmtlist> end																			
<stmtlist>			<stmt> <morestmts>				<stmt>< morestmts>			<stmt> <morestmts>						<stmt> <morestmts>	<stmt> <morestmts>	<stmt> <morestmts>				
<morestmts>					; <stmtlist>																	
<stmt>			<block>				<ifstmt>			<repeatment>						<assign>	<assign>	<assign>				
<assign>																<var> = <expr>	<var> = <expr>	<var> = <expr>				
<ifstmt>							if <testexpr> then															
<repeatstmt>										repeat <stmt> until												
<textexpr>																<var> = <expr>	<var> = <expr>	<var> = <expr>				
<expr>													=+ <expr> <expr>	- <expr> <expr>	* <expr> <expr>	<var>	<var>	<var>	<digit>	<digit>	<digit>	
<var>																a	b	c				
<digit>																			0	1	2	

3. Write a recursive descent parser for the above grammar imperative C-like pseudo code as used in class (see lecture 9).

```
main: {
        int num_operators = 0;
        token := next_token();
        if (<program>) && token == eof)
                print("accept");
        else
                print("error");
        printf("%d binary operators",num_operatorsz)
```

```
}

bool <program>:
        switch (token) {
                case prog:
                        token := next_token();
                        if (not <block>()) return false;
                        if (token == ".") {
                                token := next_token();
                        }
                        break;
                default: return false;
        }

bool <block>:
        switch (token){
                case begin:
                        token := next_token();
                        if (not <stmtlist>()) return false;
                        if (token == end) {
                                token := next_token();
                        }
                        break;
                default: return false;
        }

bool <stmtlist>:
        switch (token){
                case begin:
                        token := next_token();
                        if (not <stmt>()) return false;
                        if (not <morestmts>()) return false;
                        break;
                default: return false;
        }

bool <morestmts>:
        switch (token){
                case ";":
                        continue;
                case "epsilon":
                        break;
                default: return false;
```

```
        }

bool <stmt>:
        switch (token){
                case <assign>:
                        if (not <assign>()) return false;
                        break;
                case <ifstmt>:
                        if (not <ifstmt>()) return false;
                        break;
                case <repeatstmt>:
                        if (not <repeatstmt>()) return false;
                        break;
                case <block>:
                        if (not <block>()) return false;
                        break;
                default: return false;
        }

bool <assign>:
        switch (token){
                case <var>:
                        if (next_token() != "=") return false;
                        if (not <expr>()) return false;
                        break;
                default: return false;
        }

bool <ifstmt>:
        switch (token){
                case "if":
                        if (not <testexpr>()) return false;
                        if (next_token() != "then") return false;
                        if (not <stmt>()) return false;
                        if (next_token() != "else") return false;
                        if (not <stmt>()) return false;
                        break;
                default: return false;
        }

bool <repeatstmt>:
        switch (token){
                case "repeat":
```

```
                        if (not <stmt>()) return false;
                        if (next_token() != "until") return false;
                        if (not <textexpr>()) return false;
                        break;
                default: return false;
        }

bool <testexpr>:
        switch (token){
                case <var>:
                        if (not <var>()) return false;
                        if (next_token() != "<") return false;
                        if (next_token() != "=") return false
                        if (not <expr>()) return false;
                        num_operators++;
                        break;
                default: return false;
        }

bool <expr>:
        switch (token){
                case "+":
                case "-":
                case "*":
                        if (not <expr>()) return false;
                        if (not <expr>()) return false;
                        num_operators++;
                        break;
                case <var>:
                        if (not <var>()) return false;
                        break;
                case <digit>:
                        if (not <digit>()) return false;
                        break;
                default: return false;
        }

bool <var>:
        switch (token){
                case "a":
                case "b":
                case "c":
                        break;
```

```
                default: return false;
        }

bool <digit>:
        switch (token){
                case 0:
                case 1:
                case 2:
                        break;
                default: return false;
        }
```

4. Extend your recursive descent parser such that it prints the total number of binary operators (+, -, *, <=) in the program. For the program listed below, your parser should print '7 binary operators'

```
program
begin
        if b <= then
                begin
                        a = * a + b c
                end
                c = + a b
        else
                repeat
                begin
                        a = + a b;
                        c = - a 1
                end
                until a <= 1
end.
```