Assignment 3

CS314

blw89
Section 1

1 Problem — LL(1) Recursive Descent Parsing

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

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

In simple terms, a grammar is LL(1) if and only if, for all rules that have different derivations, have different FIRST sets and consequently, deterministic. The intersection of the different derivations of a rule must be the empty set in order for our grammar to be LL(1).

We must prove this for each rule that has different derivations, e.g.:

1. <morestmts>
2. <stmt>
3. <expr>
4. <var>
5. <digit>

For FIRST(<morestmts>), we have FIRST+(; <stmtlist>) and FIRST+(ϵ).

FIRST+(; <stmtlist>) = {;}
FIRST+(ϵ) = $\{\epsilon\} \cap \mathrm{FOLLOW}(\mathrm{morestmts}) = \{\epsilon, \mathrm{end}\}$
The intersection of these two sets is $\emptyset$.

For FIRST(<stmt>), we have FIRST+(<assign>), FIRST+(<ifstmt>), FIRST+(<repeatsmt>), FIRST+(<block>)

FIRST+(<assign>) = FIRST(<assign>) = {a,b,c}
FIRST+(<ifstmt>) = FIRST(<ifstmt>) = {if}
FIRST+(<repeatstmt>) = FIRST(<repeatstmt>) = {repeat}
FIRST+(<block>) = FIRST(<block>) = {prog}
The intersection of these 5 sets is ∅.

For FIRST(<expr>), we have FIRST+(+ <expr> <expr>), FIRST+(- <expr> <expr>), FIRST+(* <expr> <expr>), FIRST+(<var>), FIRST+(<digit>)

FIRST+(+ <expr> <expr>) = {+}
FIRST+(- <expr> <expr>) = {-}
FIRST+(* <expr> <expr>) = {*}
FIRST+(<var>) = {a, b, c}
FIRST+(<digit>) = {0, 1, 2}
The intersection of these 5 sets is ∅.

For FIRST(<var>), we have FIRST+(a), FIRST+(<b>), FIRST+(<c>).

FIRST+(a) = {a}
FIRST+(b) = {b}
FIRST+(c) = {c}
The intersection of these 3 sets is ∅.

For FIRST(<digit>), we have FIRST+(<0>), FIRST+(<1>), FIRST+(<2>).

FIRST+(0) = {0}
FIRST+(1) = {1}
FIRST+(2) = {2}
The intersection of these 3 sets is ∅.

■ since all of the intersections of the sets are ∅, we know that the grammar above is LL(1).

2. Show the LL(1) parse table.

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

```
// Important pattern to follow: when in a call, first validate
// after validating, then get next token and make call as needed.
// The method call that spawned the call will get the next token.

main {
        token := next_token();
        if (<program>() && token == eof)
                print("accept");
        else
                print("error");
}

bool <program> {
        if (token != "prog") return false;
    token := next_token();
    if (!<block>()) return false;
    token := next_token();
    return (token == ".")
}

bool <block> {
    if (token != "begin") return false;
    token := next_token();
    if (!<stmtlist>()) return false;
    token := next_token();
```

```
    return (token == "end")
}

bool <stmtlist> {
    if (!<stmt>()) return false;
    token := next_token();
    return <morestmts>();
}

bool <morestmts> {
    if (token == ";"){
        token := next_token();
        return <stmtlist>();
    } else if (token == "end"){
        return true;
    }
    return false;
}

bool <stmt> {
    return <assign>() || <ifstmt>() || <repeatstmt>() || <block>();
}

bool <assign> {
    if (!<var>()) return false;
    token := next_token();
    if (token != "=") return false;
    token := next_token();
    return <expr>();
}

bool <ifstmt> {
    if (token != "if") return false;
    token := next_token();
    if (<testexpr>()) return false;
    token := next_token();
    if (token != "then") return false;
    token := next_token();
    if (!<stmt>()) return false;
    token := next_token();
    if (token != "else") return false;
    token := next_token();
    return <stmt>();
```

```
}

bool <repeatstmt> {
    if (token != "repeat") return false;
    token := next_token();
    if (!<stmt>()) return false;
    token := next_token();
    if (token != "until") return false;
    token := next_token();
    return <testexpr>();
}

bool <testexpr> {
    if (!<var>()) return false;
    token := next_token();
    if (token != "<") return false;
    token := next_token();
    if (token != "=") return false;
    token := next_token();
    return <expr>();
}


bool <expr> {
    switch(token){
        case "+":
        case "-":
        case "*":
            token := next_token();
            if (!<expr>()) return false;
            token := next_token();
            return <expr>();
    }
    return <var>() || <digit>();
}

bool <var> {
    return (token == "a" || token == "b" || token == "c");
}

bool <digit> {
    return (token == 0 || token == 1 || token == 2);
}
```

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.
```

```
main {
        token := next_token();
    int numOperators = <program>();
        if (token == eof)
                print("accept");
        printf("%d binary operators.",numOperators);
        else
                print("error");
}

int <program> {
        if (token != "prog") return error;
    token := next_token();
    int block_ops = <block>();
    if (block_ops == error) return error;
    token := next_token();
    if (token != ".") return error;
    return block_ops;
}
```

```
int <block> {
    if (token != "begin") return error;
    token := next_token();
    int stmtlist_ops = <stmtlist>();
    if (stmtlist_ops == error) return error;
    token := next_token();
    if (token != "end") return error;
    return stmtlist_ops;
}

int <stmtlist> {
    int stmt_ops = <stmt>();
    token := next_token();
    return stmt_ops + <morestmts>();
}

int <morestmts> {
    if (token == ";"){
        token := next_token();
        return <stmtlist>();
    } else if (token == "end"){
        return true;
    }
    return error;
}

int <stmt> {
    switch (token){
        case "a":
        case "b":
        case "c":
            return <assign>();
        case "if":
            return <ifstmt>();
        case "repeat":
            return <repeatstmt>();
        case "begin":
            return <block>();
        default: return error;
    }
}

int <assign> {
```

```
    if (!<var>()) return error;
    token := next_token();
    if (token != "=") return error;
    token := next_token();
    return 1 + <expr>();
}

int <ifstmt> {
    if (token != "if") return error;
    token := next_token();
    int testexpr_ops = <testexpr>();
    if (testexpr_ops == error) return error;
    token := next_token();
    if (token != "then") return error;
    token := next_token();
    int stmt_ops = <stmt>();
    token := next_token();
    if (token != "else") return error;
    token := next_token();
    return testexpr_ops + stmt_ops + <stmt>();
}

int <repeatstmt> {
    if (token != "repeat") return error;
    token := next_token();
    int stmt_ops = <stmt>();
    if (stmt_ops == error) return error;
    token := next_token();
    if (token != "until") return error;
    token := next_token();
    return stmt_ops + <testexpr>();
}

int <testexpr> {
    if (!<var>()) return error;
    token := next_token();
    if (token != "<") return error;
    token := next_token();
    if (token != "=") return error;
    token := next_token();
    return <expr>();
}
```

```
int <expr> {
    switch(token){
        case "+":
        case "-":
        case "*":
            token := next_token();
            int expr_ops = <expr>();
            if (expr_ops == error) return error;
            token := next_token();
            return expr_ops + <expr>();
    }
    return <var>() || <digit>();
}

int <var> {
    return (token == "a" || token == "b" || token == "c");
}

int <digit> {
    return (token == 0 || token == 1 || token == 2);
}
```