Exercise 1

Codify the recursive-descent parser of the language defined by the following EBNF:

program → { *stat* }$^{+}$
stat → *if-stat* | *while-stat* | *assign-stat*
if-stat → **if** *cond* **then** *stat* { *else-if* } [**else** *stat*] **endif**
cond → **id** *relop* **num**
relop → **=** | **!=** | **>** | **<**
else-if → **elsif** *cond* **then** *stat*
while-stat → **while** *cond* **do** *stat*
assign-stat → **id := num**

Exercise 1

Codify the recursive-descent parser of the language defined by the following EBNF:

program → { *stat* }$^{+}$
stat → *if-stat* | *while-stat* | *assign-stat*
if-stat → **if** *cond* **then** *stat* { *else-if* } [**else** *stat*] **endif**
cond → **id** *relop* **num**
relop → = | != | > | <
else-if → **elsif** *cond* **then** *stat*
while-stat → **while** *cond* **do** *stat*
assign-stat → **id := num**

```
program()
{
  do
    stat()
  while(lookahead == IF ||
        lookahead == WHILE) ||
        lookahead == ID);
}

stat()
{
  if(lookahead == IF) if_stat();
  else if(lookahead == WHILE) while_stat();
  else if(lookahead == ID) assign_stat();
  else error();
}

if_stat()
{
  match(IF); cond(); match(THEN); stat();
  while(lookahead == ELSIF) else_if();
  if(lookahead == ELSE)
   {match(ELSE); stat();}
  match(ENDIF);
}
```

```
cond()
{ match(ID); relop(); match(NUM); }

relop()
{
  if(lookahead == EQ) match(EQ);
  else if(lookahead == NE) match(NE);
  else if(lookahead == GT) match(GT);
  else if(lookahead == LT) match(LT);
  else error();
}

else_if()
{match(ELSIF); cond(); match(THEN); stat();}

while_stat()
{
  match(WHILE); cond(); match(DO); stat();
}

assign_stat()
{
  match(ID); match(ASSIGN); match(NUM);
}
```

Exercise 2

Codify the recursive-descent parser of the language defined by the following EBNF:

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *assign-stat* | *if-stat*
def-stat → *type* **id** { **,** **id** }
type → **int** | **bool**
assign-stat → **id** **:=** *const*
const → **intconst** | **boolconst**
if-stat → **if** **id** **then** *block* [**else** *block*] **endif**
block → **begin** { *stat* **;** }$^{+}$ **end**

Exercise 2

Codify the recursive-descent parser of the language defined by the following EBNF:

program → { *stat* **;** }$^+$
stat → *def-stat* | *assign-stat* | *if-stat*
def-stat → *type* **id** { **,** **id** }
type → **int** | **bool**
assign-stat → **id** **:=** *const*
const → **intconst** | **boolconst**
if-stat → **if** **id** **then** *block* [**else** *block*] **endif**
block → **begin** { *stat* **;** }$^+$ **end**

```
parse()
{ next(); program(); }

program()
{ do
  { stat(); match(SEMICOLON); }
  while(look==INT || look==BOOL ||
        look==ID || look==IF);
}

stat()
{ if(look==INT || look==BOOL) def_stat();
  else if(look==ID) assign_stat();
  else if(look==IF) if_stat();
  else error();
}

def_stat()
{ type(); match(ID);
  while(look==COMMA)
  { match(COMMA); match(ID); }
}

type()
{ if(look==INT) match(INT);
  else if(look==BOOL) match(BOOL);
  else error(); }
```

```
assign_stat()
{ match(ID); match(ASSIGN); const();
}

const()
{ if(look==INTCONST) match(INTCONST);
  else if(look==BOOLCONST) match(BOOLCONST);
  else error();
}

if_stat()
{ match(IF); match(ID); match(THEN); block();
  if(look==ELSE)
  { match(ELSE); block(); }
  match(ENDIF);
}

block()
{ match(BEGIN);
  do
  { stat();
    match(SEMICOLON);
  } while(look==INT || look==BOOL || look==ID || look==IF);
  match(END);
}
```

Exercise 3

Codify the recursive-descent parser of the language defined by the following EBNF:

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *select-stat*
def-stat → **def id (** *attr-def* { **,** *attr-def* } **)**
attr-def → **id :** *domain*
domain → **integer** | **string** | **bool**
select-stat → **select** *id-list* **from** *id-list* [**where** *predicate*]
id-list → **id** { **,** **id** }
predicate → *condition* { (**and** | **or**) *condition* }
condition → *comparison* | *membership*
comparison → *elem comp-op elem*
elem → **id** | **intconst** | **strconst** | **boolconst**
comp-op → = | > | <
membership → **in (***elem* **,** *select-stat* **)**

Exercise 3

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *select-stat*
def-stat → **def id (** *attr-def* { **,** *attr-def* } **)**
attr-def → **id :** *domain*
domain → **integer** | **string** | **bool**
select-stat → **select** *id-list* **from** *id-list* [**where** *predicate*]

id-list → **id** { **,** **id** }
predicate → *condition* { (**and** | **or**) *condition* }
condition → *comparison* | *membership*
comparison → *elem comp-op elem*
elem → **id** | **intconst** | **strconst** | **boolconst**
comp-op → = | > | <
membership → **in (***elem* **,** *select-stat* **)**

```
parse(){ next(); program(); }

program()
{ do
  { stat(); match(SEMICOLON); }
    while(look==DEF || look==SELECT);
}

stat()
{ if(look==DEF) def_stat();
  else if(look==SELECT) select_stat();
  else error();
}

def_stat()
{ match(DEF); match(ID); match(LPAR);
  attr_def();
  while(look==COMMA)
  { match(COMMA); attr_def(); }
  match(RPAR);
}

attr_def()
{ match(ID); match(COLON); domain(); }

domain()
{ if(look==INTEGER || look==STRING || look==BOOL)
    next();
  else error();
}

select_stat()
{ match(SELECT); id_list();
  match(FROM); id_list();
  if(look==WHERE){match(WHERE); predicate();}
}
```

```
id_list()
{ match(ID);
  while(look==COMMA){ match(COMMA); match(ID);}
}

predicate()
{ condition();
  while(look==AND || look==OR)
  { next(); condition(); }
}

condition()
{ if(look==ID || look==INTCONST ||
     look==STRCONST || look==BOOLCONST) comparison();
  else if(look==IN) membership();
  else error();
}

comparison(){ elem(); comp_op(); elem(); }

elem()
{ if(look==ID || look==INTCONST ||
     look==STRCONST || look==BOOLCONST) next();
  else error();
}

comp_op()
{ if(look==EQUAL || look==GREATER || look==LESS)
    next();
  else error();
}

membership()
{  match(IN); match(LPAR); elem();
   match(COMMA); select_stat(); match(RPAR);
}
```

Exercise 4

Codify the recursive-descent parser of the language defined by the following EBNF:

program → { *stat* **;** }
stat → *def-table* | *join-op*
def-table → **table id is** *attr-decl* { **,** *attr-decl* } **end**
attr-decl → **id :** *domain*
domain → **int** | **string**
join-op → **join id** { **, id** } [**suchthat** *criterion*]
criterion → *cond* { (**and** | **or**) *cond* }
cond → *attr* (= | > | <) *attr*
attr → **id** [**. id**]

Exercise 4

program → { *stat* **;** }
stat → *def-table* | *join-op*
def-table → **table id is** *attr-decl* { **,** *attr-decl* } **end**
attr-decl → **id :** *domain*

domain → **int** | **string**
join-op → **join id** { **, id** } [**suchthat** *criterion*]
criterion → *cond* { (**and** | **or**) *cond* }
cond → *attr* (= | > | <) *attr*
attr → **id** [**. id**]

```
parse()
{ next();
  program();
}

program()
{ while(look==TABLE || look==JOIN)
  { stat();
    match(SEMICOLON);
  }
}

stat()
{ if(look==TABLE) def_table();
  else join_op();
}

def_table()
{ next(); match(ID); match(IS); attr_decl();
  while(look==COMMA){next();attr_decl();}
  match(END);
}

attr_decl()
{ match(ID); match(COLON); domain(); }
```

```
domain()
{ if(look==INT || look==STRING) next();
  else error();}

join_op()
{ next(); match(ID);
  while(look==COMMA){ next(); match(ID); }
  if(look==SUCHTHAT){next(); criterion();}
}

criterion()
{ cond();
  while(look==AND || look==OR)
  { next(); cond(); }
}

cond()
{ attr();
  if(look==EQUAL ||look==GREATER ||look==LESS)
  { next(); attr(); }
  else error();
}

attr()
{ match(ID);
  if(look==DOT){next(); match(ID);}
}
```

Exercise 5

Given a language defined by the grammar G specified by the following BNF:

program → *stat-list*
stat-list → *stat* **;** *stat-list* | **ε**
stat → *def-stat* | *if-stat* | **display**
def-stat → **id :** *type*
type → **int** | **string**
if-stat → **if** *expr* **then** *stat* **else** *stat*
expr → **boolconst**

- Build the LL(1) parsing table relevant to G;
- Based on this parsing table, codify a recursive-descent parsing of G.

Exercise 5

program → *stat-list*
stat-list → *stat* **;** *stat-list* | **ε**
stat → *def-stat* | *if-stat* | **display**
def-stat → **id :** *type*
type → **int** | **string**
if-stat → **if** *expr* **then** *stat* **else** *stat*
expr → **boolconst**

FIRST(*program*) = *FIRST*(*stat-list*) = { **id**, **if**, **display**, **ε** }
FIRST(*stat*) = { **id**, **if**, **display** }
FIRST(*type*) = { **int**, **string** }
FIRST(*def-stat*) = { **id** }
FIRST(*if-stat*) = { **if** }
FIRST(*expr*) = { **boolconst** }
FOLLOW(*program*) = *FOLLOW*(*stat-list*) = { **$** }

	id	**if**	**display**	**int**	**string**	**boolconst**	**$**
program	*stat-list*	*stat-list*	*stat-list*				*stat-list*
stat-list	*stat* **;** *stat-list*	*stat* **;** *stat-list*	*stat* **;** *stat-list*				ε
stat	*def-stat*	*if-stat*	**display**				
def-stat	**id :** *type*						
type				**int**	**string**		
if-stat		**if** *expr* **then** *stat* **else** *stat*					
expr						**boolconst**	

Exercise 5 (ii)

	id	if	display	int	string	boolconst	$
program	*stat-list*	*stat-list*	*stat-list*				*stat-list*
stat-list	*stat* **;** *stat-list*	*stat* **;** *stat-list*	*stat* **;** *stat-list*				ε
stat	*def-stat*	*if-stat*	**display**				
def-stat	**id :** *type*						
type				**int**	**string**		
if-stat		**if** *expr* **then** *stat* **else** *stat*					
expr						**boolconst**	

```
parse()
{ next();
  program();
}

program()
{ if(look==ID||look==IF||look==DISPLAY||look==EOF)
    stat_list();
  else error();
}

stat_list()
{ if(look==ID||look==IF||look==DISPLAY)
  { stat(); match(SEMICOLON); stat_list(); }
  else if(look==EOF)
    ;
  else error();
}

stat()
{ if(look==ID) def_stat();
  else if(look==IF) if_stat();
  else if(look==DISPLAY) next();
  else error();
}
```

```
def_stat()
{ match(ID); match(COLON); type(); }

type()
{ if(look==INT||look==STRING) next();
  else error();
}

if_stat()
{ match(IF); expr(); match(THEN);
  stat(); match(ELSE); stat();
}

expr()
{ match(BOOLCONST);
}
```

Exercise 6

Codify the recursive-descent parser of the language defined by the following EBNF:

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *procedure-call*
def-stat → **id** { **,** **id** } **:** *type*
type → **int** | **string** | **bool** | *structured-type*
structured-type → **matrix** **[** **intconst** { **,** **intconst** } **]** **of** *type*
procedure-call → **call** **id** **(** [*parameters*] **)**
parameters → *param* { **,** *param* }
param → **intconst** | **stringconst** | **boolconst** | **id**

Exercise 6

Codify the recursive-descent parser of the language defined by the following EBNF:

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *procedure-call*
def-stat → **id** { **,** **id** } **:** *type*
type → **int** | **string** | **bool** | *structured-type*
structured-type → **matrix [intconst** { **,** **intconst** } **]** **of** *type*
procedure-call → **call id (** [*parameters*] **)**
parameters → *param* { **,** *param* }
param → **intconst** | **stringconst** | **boolconst** | **id**

```
parse(){next(); program();}

program()
{ do
  { stat(); match(SEMICOLON); }
   while (look==ID || look==CALL);
}

stat()
{ if(look==ID) def_stat();
  else if(look==CALL) procedure_call();
  else error();
}

def_stat()
{ match(ID);;
  while{look==COMMA) {next(); match(ID);}
  match(COLON);
  type();
}

type()
{ if(look==INT || look==STRING ||look==BOOL) next();
  else if(look==MATRIX) structured_type();
  else error();
}
```

```
structured_type()
{ match(MATRIX); match(LBRACK); match(INTCONST);
  while(look==COMMA){next(); match(INTCOONST);}
  match(RBRACK); match(OF); type();
}

procedure_call()
{ match(CALL); match(ID); match(LPAR);
  if(look==INTCONST || look==STRINGCONST||
     look==BOOLCOSNT || look==ID)
    parameters();
  match(RPAR);
}

parameters()
{ param();
  while{look==COMMA) {next(); param();}
}

param()
{ if(look==INTCONST || look==STRINGCONST||
     look==BOOLCOSNT || look==ID) next();
  else error();
}
```

Exercise 7

Specify the LL(1) parsing table of the language L defined by the following BNF:

program → *def-list*
def-list → *def* **;** *def-list* | **ε**
def → *type-def* | *function-def*
type-def → **type id =** *domain*
domain → **int** | **string** | **[** *domain* **]**
function-def → **function id (** *param-list* **) :** *domain*
param-list → *param* **,** *param-list* | **ε**
param → **id :** *domain*

Based on the parsing table, codify a recursive-descent parser of L.

Exercise 7

program → *def-list*
def-list → *def* **;** *def-list* | **ε**
def → *type-def* | *function-def*
type-def → **type id =** *domain*
domain → **int** | **string** | **[** *domain* **]**
function-def → **function id (** *param-list* **) :** *domain*
param-list → *param* **,** *param-list* | **ε**
param → **id :** *domain*

FIRST(*program*) **=** *FIRST*(*def-list*) = { **type**, **function**, **ε** }
FIRST(*def*) = { **type**, **function** }
FIRST(*type-def*) = { **type** }
FIRST(*domain*) = { **int**, **string**, **[** }
FIRST(*function-def*) = { **function** }
FIRST(*param-list*) = { **id**, **ε** }
FIRST(*param*) = { **id** }
FOLLOW(*program*) **=** *FOLLOW*(*def-list*) = { **$** }
FOLLOW(*param-list*) **=** { **)** }

	type	**function**	**int**	**string**	**[**	**)**	**id**	**$**
program	*def-list*	*def-list*						*def-list*
def-list	*def* **;** *def-list*	*def* **;** *def-list*						ε
def	*type-def*	*function-def*						
type-def	**type id** = *domain*							
domain			**int**	**string**	**[** *domain* **]**			
function-def		**function id (** *param-list* **) :** *domain*						
param-list						ε	*param* **,** *param-list*	
param							**id :** *domain*	

Exercise 7 (ii)

	type	function	int	string	[	)	id	$
program	*def-list*	*def-list*						*def-list*
def-list	*def* **;** *def-list*	*def* **;** *def-list*						ε
def	*type-def*	*function-def*						
type-def	**type id** = *domain*							
domain			**int**	**string**	**[** *domain* **]**			
function-def		**function id (** *param-list* **) :** *domain*						
param-list						ε	*param* **,** *param-list*	
param							**id :** *domain*	

```
parse()
{ next(); program(); }

program()
{ if(look==TYPE || look==FUNCTION || look==EOF)
    def_list();
  else error();
}

def_list()
{ if(look==TYPE || look==FUNCTION)
  { def(); match(SEMICOLON); def_list(); }
  else if(look==EOF)
    ;
  else error();
}

def()
{ if(look==TYPE) type_def();
  else if(look==FUNCTION) function_def();
  else error();
}
```

```
type_def()
{ match(TYPE); match(ID); match(EQUAL); domain(); }

domain
{ if(look==INT || look==STRING) next();
  else if(look==SQUARE_LEFT)
  { next(); domain(); match(SQUARE_RIGHT); }
  else error();
}

function_def()
{ match(FUNCTION); match(ID); match(ROUND_LEFT);
  param-list(); match(ROUND_RIGHT); match(COLON);
  domain();
}

param_list()
{ if(look==ID)
  { param(); match(COMMA); param_list(); }
  else if(look==ROUND_RIGHT)
    ;
  else error();
}

param(){ match(ID); match(COLON); domain(); }
```

Exercise 8

Trace the LL(1) parsing of the sentence **ba(b)** relevant to the language defined by the following grammar:

$A \rightarrow A\mathbf{a}B \mid B$

$B \rightarrow \mathbf{(}A\mathbf{)} \mid \mathbf{b}$

Exercise 8

Trace the LL(1) parsing of the sentence **ba(b)** relevant to the language defined by the following grammar:

$A \rightarrow A\mathbf{a}B \mid B$

$B \rightarrow \mathbf{(}A\mathbf{)} \mid \mathbf{b}$

$A \rightarrow BR$

$R \rightarrow \mathbf{a}BR \mid \varepsilon$

$B \rightarrow \mathbf{(}A\mathbf{)} \mid \mathbf{b}$

$FOLLOW(R) = FOLLOW(A) = \{ \mathbf{)}, \mathbf{\$} \}$

$FISRT(BR) = \{ \mathbf{(}, \mathbf{b} \}$

$FIRST(\mathbf{a}BR) = \{ \mathbf{a} \}$

$FISRT(\varepsilon) = \{ \varepsilon \}$

$FISRT(\mathbf{(}A\mathbf{)}) = \{ \mathbf{(} \}$

$FISRT(\mathbf{b}) = \{ \mathbf{b} \}$

	a	**(**	**)**	**b**	**$**
A		BR		BR	
R	$\mathbf{a}BR$		ε		ε
B		$\mathbf{(}A\mathbf{)}$		**b**	

Stack	*Input*	*Action*
A$	ba(b)$	$A \rightarrow BR$
BR$	ba(b)$	$B \rightarrow \mathbf{b}$
bR$	ba(b)$	match
R$	a(b)$	$R \rightarrow \mathbf{a}BR$
aBR$	a(b)$	match
BR$	(b)$	$B \rightarrow \mathbf{(}A\mathbf{)}$
(A)R$	(b)$	match
A)R$	b)$	$A \rightarrow BR$
BR)R$	b)$	$B \rightarrow \mathbf{b}$
bR)R$	b)$	match
R)R$	)$	$R \rightarrow \varepsilon$
)R$	)$	match
R$	$	$R \rightarrow \varepsilon$
$	$	accept

Exercise 9

Outline the LL(1) parsing of the sentence **bba** relevant to the language defined by the following grammar:

$A \rightarrow A$ **a** | B **b**

$B \rightarrow$ **c** A **d** | **b**

Exercise 9

Outline the LL(1) parsing of the sentence **bba** relevant to the language defined by the following grammar:

$A \rightarrow A$ **a** $\mid B$ **b**

$B \rightarrow$ **c** A **d** $\mid$ **b**

$A \rightarrow B$ **b** R

$R \rightarrow$ **a** $R \mid \varepsilon$

$B \rightarrow$ **c** A **d** $\mid$ **b**

$FOLLOW(R) = FOLLOW(A) =$ { **$, d** }

$FISRT(B\mathbf{b}R) =$ { **c, b** }

$FISRT(\mathbf{a}R) =$ { **a** }

$FISRT(\mathbf{c}A\mathbf{d}) =$ { **c** }

	a	**b**	**c**	**d**	**$**
A		$B\mathbf{b}R$	$B\mathbf{b}R$		
R	**a**R			ε	ε
B		**b**	**c**A**d**		

Stack	*Input*	*Action*
A$	bba$	$A \rightarrow B\mathbf{b}R$
BbR$	bba$	$B \rightarrow$ **b**
bbR$	bba$	match
bR$	ba$	match
R$	a$	$R \rightarrow$ **a**R
aR$	a$	match
R$	$	$R \rightarrow \varepsilon$
$	$	accept

Exercise 10

After specifying the LL(1) parsing table of the language **L** defined by the following BNF, codify a recursive-descent parser of **L**.

program → *type-def*
type-def → **id** **:** *type*
type → **int** | **string** | *ptr-type* | *rec-type* | *vect-type*
ptr-type → **^** *type*
rec-type → **record** **(** *type-def-list* **)**
type-def-list → *type-def* *type-def-list* | **ε**
vect-type → **vector** **[** **intconst** **]** **of** *type*

Exercise 10

After specifying the LL(1) parsing table of the language **L** defined by the following BNF, codify a recursive-descent parser of **L**.

program → *type-def*
type-def → **id** **:** *type*
type → **int** | **string** | *ptr-type* | *rec-type* | *vect-type*
ptr-type → **^** *type*
rec-type → **record** **(** *type-def-list* **)**
type-def-list → *type-def* *type-def-list* | **ε**
vect-type → **vector** **[** **intconst** **]** **of** *type*

FOLLOW(*type-def-list*) = { **)** }

	id	int	string	^	record	vector	)
program	*type-def*						
type-def	**id :** *type*						
type		**int**	**string**	*ptr-type*	*rec-type*	*vect-type*	
ptr-type				**^** *type*			
rec-type					**record** **(** *type-def-list* **)**		
type-def-list	*type-def* *type-def-list*						**ε**
vect-type						**vector** **[** **intconst** **]** **of** *type*	

Exercise 10 (ii)

	id	int	string	^	record	vector	)
program	*type-def*						
type-def	**id :** *type*						
type		**int**	**string**	*ptr-type*	*rec-type*	*vect-type*	
ptr-type				^ *type*			
rec-type					**record (***type-def-list***)**		
type-def-list	*type-def type-def-list*						**ε**
vect-type						**vector [intconst] of** *type*	

```
parse()
{ next(); program(); }

program()
{ if(look==ID) type_def();
  else error();
}

type_def()
{
  match(ID); match(COLON); type();
}

type()
{ if(look==INT||look==STRING) next();
  else if(look==HAT) ptr_type();
  else if(look==RECORD) rec_type();
  else if(look==VECTOR) vect_type();
  else error();
}
```

```
ptr_type()
{ match(HAT); type();}

rec_type()
{ match(RECORD); match(LPAR);
  type_def_list(); match(RPAR);
}

type_def_list()
{ if(look==ID)
    {type_def(); type_def_list();}
  else if(look==RPAR)
     ;
  else error();
}

vect_type()
{
  match(VECTOR); match(LBRACKET);
  match(INTCONST); match(RBRACKET); match(OF);
  type();
}
```

Exercise 11

Codify the recursive-descent parser of the language defined by the following EBNF:

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *assign-stat* | *if-stat*
def-stat → **def id** { **,** **id** } **:** *type*
type → **int** | **string** | **bool** | *record-type*
record-type → **record** **(** **id :** *type* { **,** **id :** *type* } **)**
assign-stat → **id :=** (**id** | *const*)
const → (**intconst** | **strconst** | **boolconst**)
if-stat → **if** *cond* **then** *stat* [*elsif-part*] **otherwise** *stat*
elsif-part → { **elsif** *cond* **then** *stat* }$^{+}$
cond → **id =** *const*

Exercise 11

program → { *stat* **;** }$^+$
stat → *def-stat* | *assign-stat* | *if-stat*
def-stat → **def id** { **,** **id** } **:** *type*
type → **int** | **string** | **bool** | *record-type*
record-type → **record** **(** **id :** *type* { **,** **id :** *type* } **)**

assign-stat → **id :=** (**id** | *const*)
const → (**intconst** | **strconst** | **boolconst**)
if-stat → **if** *cond* **then** *stat* [*elsif-part*] **otherwise** *stat*
elsif-part → { **elsif** *cond* **then** *stat* }$^+$
cond → **id =** *const*

```
parse(){ next(); program(); }

program()
{ do {
    stat();
    match(SEMICOLON);
  } while(look==DEF || look==ID || look==IF);
}

stat()
{ if(look==DEF) def_stat();
  else if(look==ID) assign_stat();
  else if(look==IF) if_stat();
  else error();
}

def_stat()
{ match(DEF); match(ID);
  while(look==COMMA){next(); match(ID);}
  match(COLON); type();
}

type()
{ if(look==INT || look==STRING || look==BOOL) next();
  else if(look==RECORD) record_type();
  else error();
}

record_type()
{ match(RECORD); match(LPAR);
  match(ID); match(COLON); type();
  while(look==COMMA){
    next(); match(ID); match(COLON); type();}
  match(RPAR);
}
```

```
assign_stat()
{  match(ID);
   match(ASSIGN);
   if(look==ID) next();
   else const();
}

const(){if(look==INTCONST ||
           look==STRCONST ||
           look==BOOLCONST)
          next();
        else error();}

if_stat()
{ match(IF); cond(); match(THEN); stat();
  if(look==ELSIF) elsif_part();
  match(OTHERWISE); stat();
}

elsif_part()
{ do
  { match(ELSIF); cond(); match(THEN); stat(); }
  while(look==ELSIF);
}

cond(){ match(ID); match(EQUAL); const(); }
```

Exercise 12

Check whether the following BNF is LL(1):

program → *stat*
stat → *def-stat* | *assign-stat*
def-stat → **relation id :** *rel-type*
rel-type → **[** *attr-list* **]**
attr-list → *attr-def attr-list* | **ε**
attr-def → **id :** *type*
type → *atomic-type* | *rel-type*
atomic-type → **int** | **string**
assign-stat → **id :=** *const*
const → **intconst** | **strconst**

Exercise 12

Check whether the following BNF is LL(1):

program → *stat*
stat → *def-stat* | *assign-stat*
def-stat → **relation id :** *rel-type*
rel-type → **[** *attr-list* **]**
attr-list → *attr-def attr-list* | **ε**
attr-def → **id :** *type*
type → *atomic-type* | *rel-type*
atomic-type → **int** | **string**
assign-stat → **id :=** *const*
const → **intconst** | **strconst**

FOLLOW(*attr-list*) = { **]** }

	relation	**id**	**[**	**]**	**int**	**string**	**intconst**	**strconst**
program	*stat*	*stat*						
stat	*def-stat*	*assign-stat*						
def-stat	**relation id :** *rel-type*							
rel-type			**[** *attr-list* **]**					
attr-list		*attr-def attr-list*		**ε**				
attr-def		**id :** *type*						
type			*rel-type*		*atomic-type*	*atomic-type*		
atomic-type					**int**	**string**		
assign-stat		**id :=** *const*						
const							**intconst**	**strconst**

Exercise 13

Codify the recursive-descent parser of the language defined by the following EBNF:

program → **program id** [*var-section*] *body* **.**
var-section → **var** { *decl-list* **;** }$^{+}$
decl-list → **id** { **,** **id** } **:** *type*
type → **int** | **real** | **string**
body → **begin** *block* **end**
block → { *stat* **;** }$^{+}$
stat → *assign-stat* | *if-stat*
assign-stat → **id** **:=** (*const* | **id**)
const → **intconst** | **realconst** | **stringconst**
if-stat → **if** *cond* **then** *block* { **elsif** *cond* **then** *block* } [**otherwise** *block*] **endif**
cond → **id** (= | **!=** | >| < | >=| <=) (*const* | **id**)

Exercise 13

program → **program id** [*var-section*] *body* **.**
var-section → **var** { *decl-list* **;** }$^{+}$
decl-list → **id** { **,** **id** } **:** *type*
type → **int** | **real** | **string**
body → **begin** *block* **end**
block → { *stat* **;** }$^{+}$

```
parse(){ next(); program(); }

program()
{ match(PROGRAM); match(ID);
  if(look==VAR)
    var_section();
  body();
  match(DOT);
}

var_section()
{ match(VAR);
  do {decl_list; match(SEMICOLON);}
  while(look==ID);
}

decl_list()
{ match(ID);
  while(look==COMMA){next(); match(ID);}
  match(COLON); type();
}

type()
{ if(look==INT || look==REAL || look==STRING) next();
  else error();
}

body(){ match(BEGIN); block(); match(END); }

block()
{ do {stat(); match(SEMICOLON);}
  while(look==ID || look==IF);
}
```

Exercise 13 (ii)

stat → *assign-stat* | *if-stat*
assign-stat → **id** := (*const* | **id**)
const → **intconst** | **realconst** | **stringconst**
if-stat → **if** *cond* **then** *block* { **elsif** *cond* **then** *block* } [**otherwise** *block*] **endif**
cond → **id** (= | != | >| < | >=| <=) (*const* | **id**)

```
stat()
{ if(look==ID) assign_stat();
  else if(look==IF) if_stat();
  else error();
}

assign_stat()
{ match(ID); match(ASSIGN);
  if(look==INTCONST || look==REALCONST || look==STRCONST) const();
  else match(ID);
}

const()
{ if(look==INTCONST || look==REALCONST || look == STRCONST) next();
  else error();
}

if_stat()
{ match(IF); cond(); match(THEN) block();
  while(look==ELSIF){next(); cond(); match(THEN); block();}
  if(look==OTHERWISE) {next(); block();}
  match(ENDIF);
}

cond()
{ match(ID);
  if(look==EQ || look==NE || look==GT || look ==LT || look==GE || look==LE) next();
  else error();
  if(look==INTCONST || look==REALCONST || look==STRCONST) const();
  else match(ID);
}
```

Exercise 14

After specifying the LL(1) parsing table of the language defined by the following BNF, check whether this grammar is LL(1).

program → def-list
def-list → def def-list | **ε**
def → **type id =** dom
dom → simple-dom | struct-dom
simple-dom → **int** | **string** | **id**
struct-dom → tuple-dom| list-dom | func-dom
tuple-dom → **(** dom-list **)**
dom-list → dom dom-list | **ε**
list-dom → **[** dom **]**
func-dom → **(** dom **:** dom map-list **)**
map-list → **:** dom map-list | **ε**

Exercise 14

program → def-list
def-list → def def-list | **ε**
def → **type id =** dom
dom → simple-dom | struct-dom
simple-dom → **int** | **string** | **id**
struct-dom → tuple-dom| list-dom | func-dom

tuple-dom → **(** dom-list **)**
dom-list → dom dom-list | **ε**
list-dom → **[** dom **]**
func-dom → **(** dom **:** dom map-list **)**
map-list → **:** dom map-list | **ε**

	type	**int**	**string**	**id**	**(**	**)**	**[**	**:**	**$**
program	def-list								def-list
def-list	def def-list								**ε**
def	**type id =** dom								
dom		simple-dom	simple-dom	simple-dom	struct-dom		struct-dom		
simple-dom		**int**	**string**	**id**					
struct-dom					tuple-dom func-dom		list-dom		
tuple-dom					**(** dom-list **)**				
dom-list		dom dom-list	dom dom-list	dom dom-list	dom dom-list	**ε**	dom dom-list		
list-dom							**[** dom **]**		
func-dom					**(** dom **:** dom map-list **)**				
map-list						**ε**		**:** dom map-list	

Exercise 15

After specifying the LL(1) parsing table of the language defined by the following BNF, check whether this grammar is LL(1).

program → stat-list
stat-list → stat **;** stat-list | **ε**
stat → def-stat | assign-stat
def-stat → id-list **:** type
id-list → **id ,** id-list | **id**
type → **int** | record
record → **(** attr-list **)**
attr-list → attr **,** attr-list | **ε**
attr → **id :** type
assign-stat → **id :=** expr
expr → term **+** expr | term
term → factor ***** term | factor
factor → **num** | **id** | **(** expr **)**

Exercise 15

program → stat-list
stat-list → stat **;** stat-list | **ε**
stat → def-stat | assign-stat
def-stat → id-list **:** type
id-list → **id ,** id-list | **id**
type → **int** | record
record → **(** attr-list **)**
attr-list → attr **,** attr-list | **ε**
attr → **id :** type
assign-stat → **id :=** expr
expr → term **+** expr | term
term → factor ***** term | factor
factor → **num** | **id** | **(** expr **)**

	id	**int**	**(**	**)**	**num**	**$**
program	stat-list					stat-list
stat-list	stat ; stat-list					ε
stat	def-stat assign-stat					
def-stat	id-list **:** type					
id-list	**id ,** id-list **id**					
type		**int**	record			
record			**(** attr-list **)**			
attr-list	attr **,** attr-list			ε		
attr	**id :** type					
assign-stat	**id :=** expr					
expr	term **+** expr term		term **+** expr term		term **+** expr term	
term	factor ***** term factor		factor ***** term factor		factor ***** term factor	
factor	**id**		**(** expr **)**		**num**	

Exercise 16

Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase terminates with EOF.

$$S \rightarrow A\ (\mathbf{,}\ B)^{+}\ \mathbf{c}\ B \mid \mathbf{z}\ A$$
$$A \rightarrow \mathbf{b}\ (S^{*}\ \mathbf{a})^{+}\ [\ \mathbf{f}\ A\ ]\ \mathbf{c} \mid B$$
$$B \rightarrow \mathbf{a}\ A^{+} \mid \mathbf{c}$$

Exercise 16

Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase terminates with EOF.

$$S \rightarrow A\ (,\ B)^{+}\ \mathbf{c}\ B \mid \mathbf{z}\ A$$
$$A \rightarrow \mathbf{b}\ (S^{*}\ \mathbf{a})^{+}\ [\ \mathbf{f}\ A\ ]\ \mathbf{c} \mid B$$
$$B \rightarrow \mathbf{a}\ A^{+} \mid \mathbf{c}$$

```
parse()
{
  next();
  S();
  if(look!=EOF)
    error();
}

S()
{
  if(look=='a' || look=='b' || look=='c')
  {  A();
     do
     { match{',');
       B();
     } while(look==',');
     match('c');
     B();
   }
   else if(look=='z')
   { next();
     A();
   }
   else error();
}
```

```
A()
{
   if(look=='b')
   { next();
     do
     { while(look=='a' || look=='b' || look=='c' || look=='z')
         S();
       match('a');
     } while(look=='a' || look=='b' || look=='c' || look=='z');
     if(look=='f')
     { next();
       A();
     }
     match('c');
   }
   else if(look=='a' || look=='c')
     B();
}

B()
{
   if(look=='a')
   { next();
     do
       A();
     while(look=='a' || look=='b' || look=='c');
   }
   else
     match('c');
}
```

Exercise 17

After specifying the parsing table of the language defined by the following BNF, determine whether such BNF is LL(1):

program → **module id is** var-decl **body** body-decl **end**
var-decl → var-list var-decl | var-list
var-list → id-list **:** type **;**
id-list → **id ,** id-list | **id**
type → **integer** | **string** | rec-type | vec-type
rec-type → **record (** attr-list **)**
attr-list → attr-decl **,** attr-list | attr-decl
attr-decl → **id :** type
vec-type → **vector [intconst] of** type
body-decl → assign body-decl | assign
assign → lhs **:=** rhs **;**
lhs → **id** | lhs [**intconst**] | lhs **. id**
rhs → **intconst** | **strconst**

Exercise 17

	module	id	integer	string	record	vector	intconst	strconst
program	**module id is** …							
var-decl		var-list var-decl var-list						
var-list		id-list **:** type **;**						
id-list		**id ,** id-list **id**						
type			**integer**	**string**	rec-type	vec-type		
rec-type					**record (** attr-list **)**			
attr-list		attr-decl , attr-list attr-decl						
attr-decl		id **: type**						
vec-type						**vector [intconst] of** type		
body-decl		assign body-decl assign						
assign		lhs **:=** rhs **;**						
lhs		**id** lhs [**intconst**] lhs **. id**						
rhs							**intconst**	**strconst**

program → **module id is** var-decl **body** body-decl **end**
var-decl → var-list var-decl | var-list
var-list → id-list **:** type **;**
id-list → **id ,** id-list | **id**
type → **integer** | **string** | rec-type | vec-type
rec-type → **record (** attr-list **)**
attr-list → attr-decl **,** attr-list | attr-decl
attr-decl → **id :** type
vec-type → **vector [intconst] of** type
body-decl → assign body-decl | assign
assign → lhs **:=** rhs **;**
lhs → **id** | lhs [**intconst**] | lhs **. id**
rhs → **intconst** | **strconst**

Exercise 18

Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase terminates with EOF.

$$S \rightarrow \mathbf{a}\ \{\mathbf{,}\ A\}^{+}\ \mathbf{b}\ [\ \mathrm{c}\ B\ ]\ |\ \mathbf{b}\ \{B\}$$
$$A \rightarrow \mathbf{a}\ \{\{S\}\ \mathbf{d}\}^{+}\ |\ \mathbf{b}$$
$$B \rightarrow \mathbf{c}\ [\ A\ ]\ |\ S$$

Exercise 18

Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase terminates with EOF.

$$S \rightarrow \mathbf{a}\ \{, A\}^{+}\ \mathbf{b}\ [\ \mathrm{c}\ B\ ]\ |\ \mathbf{b}\ \{B\}$$
$$A \rightarrow \mathbf{a}\ \{\{S\}\ \mathbf{d}\}^{+}\ |\ \mathbf{b}$$
$$B \rightarrow \mathbf{c}\ [\ A\ ]\ |\ S$$

```
parse()
{ next();
  S();
  if(look!=EOF)
    error();
}

S()
{ if(look=='a')
  {  next();
     do
     { match(',');
       A();
     } while(look==',');
     match('b');
     if(look=='c')
     { next();
       B();
     }
  }
  else if(look=='b')
   { next();
     while(look=='a' || look=='b' || look=='c')
       B();
   }
   else error();
}
```

```
A()
{ if(look=='a')
  { next();
    do
    { while(look=='a' || look=='b')
        S();
      match('d');
    } while(look=='a' || look=='b' || look=='d');
  }
  else
    match('b');
}

B()
{
   if(look=='c')
   { next();
     if(look=='a' || look=='b')
       A();
   }
   else if(look=='a' || look=='b')
     S();
}
```

Exercise 19

Given the following grammar **G** in BNF notation,

$$A \rightarrow A\ \mathbf{b}\ B \mid B$$
$$B \rightarrow A\ \mathbf{c} \mid \mathbf{b}$$

we ask to:

- Transform **G** into a non left-recursive grammar **G'** (equivalent to **G**);
- Based on the parsing table, determine whether **G'** is LL(1).

Exercise 19

$A \rightarrow A\ \mathbf{b}\ B \mid B$
$B \rightarrow A\ \mathbf{c} \mid \mathbf{b}$

⇨

$A \rightarrow B\ A'$
$A' \rightarrow \mathbf{b}\ B\ A' \mid \boldsymbol{\varepsilon}$
$B \rightarrow A\ \mathbf{c} \mid \mathbf{b}$

⇨

$A \rightarrow B\ A'$
$A' \rightarrow \mathbf{b}\ B\ A' \mid \boldsymbol{\varepsilon}$
$B \rightarrow B\ A'\ \mathbf{c} \mid \mathbf{b}$

⇨

$A \rightarrow B\ A'$
$A' \rightarrow \mathbf{b}\ B\ A' \mid \boldsymbol{\varepsilon}$
$B \rightarrow \mathbf{b}\ B'$
$B' \rightarrow A'\ \mathbf{c}\ B' \mid \boldsymbol{\varepsilon}$

$FIRST(B\ A') = \{\ \mathbf{b}\ \}$
$FIRST(A'\ \mathbf{c}\ B') = \{\ \mathbf{b}, \mathbf{c}\ \}$
$FIRST(A') = \{\ \mathbf{b}, \boldsymbol{\varepsilon}\ \}$
$FIRST(A) = FIRST(B) = \{\ \mathbf{b}\ \}$
$FIRST(B') = \{\ \mathbf{b}, \mathbf{c}, \boldsymbol{\varepsilon}\ \}$

	b	**c**	**$**
A	$A \rightarrow B\ A'$		
A'	$A' \rightarrow \mathbf{b}\ B\ A'$	$A' \rightarrow \boldsymbol{\varepsilon}$	$A' \rightarrow \boldsymbol{\varepsilon}$
B	$B \rightarrow \mathbf{b}\ B'$		
B'	$B' \rightarrow A'\ \mathbf{c}\ B'$ $B' \rightarrow \boldsymbol{\varepsilon}$	$B' \rightarrow A'\ \mathbf{c}\ B'$ $B' \rightarrow \boldsymbol{\varepsilon}$	$B' \rightarrow \boldsymbol{\varepsilon}$

⇨ **Not LL(1)!**

$FOLLOW(A) = \{\ \mathbf{\$}\ \}$
$FOLLOW(A') = \{\ \mathbf{\$}, \mathbf{c}\ \}$
$FOLLOW(B') = FOLLOW(B) = \{\ \mathbf{b}, \mathbf{c}, \mathbf{\$}\ \}$

Exercise 20

Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase terminates with EOF.

program→ { *stat* }$^+$
stat → (*def-stat* | *if-stat* | *loop-stat* | *assign-stat* | **break**) **;**
def-stat → *type* **id** { **,** **id** }
type → **int** | **bool** | **struct (** *attr-list* **)** | **vector [intconst] of** *type*
attr-list → **id :** *type* { **,** **id :** *type* }
if-stat → **if** *expr* **then** { *stat* }$^+$ [**else** { *stat* }$^+$] **endif**
loop-stat → **loop** { *stat* }$^+$ **endloop**
assign-stat → **id :=** *expr*
expr → **id** | **boolconst** | **intconst**

Exercise 20

```
parse(){next(); program();}

program()
{  do
     stat()
   while(look==INT || look==BOOL || look==STRUCT || look==VECTOR || look==IF ||
         look==LOOP || look==ID || look==BREAK);
   match(EOF);
}

stat()
{  if(look==INT || look==BOOL || look==STRUCT || look==VECTOR) def-stat();
   else if(look==IF) if-stat();
   else if(look==LOOP) loop-stat();
   else if(look==ID) assign-stat();
   else if(look==BREAK) next();
   else error();
   match(SEMICOLON);
}

def-stat()
{  type(); match(ID);
   while(look==COMMA){next(); match(ID);}
}

type()
{  if(look==INT || look==BOOL) next();
   else if(look==STRUCT){next(); match(LPAR); attr-list(); match(RPAR);}
   else if(look==VECTOR){next(); match(LQPAR); match(INTCONST); match(RQPAR); match(OF); type();}
   else error();
}

attr-list()
{  match(ID); match(COLON); type();
   while(look==COMMA){next(); match(ID); match(COLON); type();}
}
```

program → { *stat* }$^{+}$
stat → (*def-stat* | *if-stat* | *loop-stat* | *assign-stat* | **break**) **;**
def-stat → *type* **id** { **,** **id** }
type → **int** | **bool** | **struct (** *attr-list* **)** | **vector [** **intconst** **]** **of** *type*
attr-list → **id** **:** *type* { **,** **id** **:** *type* }

Exercise 20 (ii)

```
if-stat()
{  match(IF); expr(); match(THEN);
   do
      stat();
   while(look==INT || look==BOOL || look==STRUCT || look==VECTOR || look==IF ||
         look==LOOP || look==ID || look==BREAK);
   if(look==ELSE)
   {  next();
      do
         stat();
      while(look==INT || look==BOOL || look==STRUCT || look==VECTOR || look==IF ||
            look==LOOP || look==ID || look==BREAK);
   }
   match(ENDIF);
}

loop-stat()
{  match(LOOP);
   do
      stat()
   while(look==INT || look==BOOL || look==STRUCT || look==VECTOR || look==IF ||
            look==LOOP || look==ID || look==BREAK);
   match(ENDLOOP);
}

assign-stat()
{  match(ID); match(ASSIGN); expr(); }

expr()
{  if(look==ID || look==BOOLCONST ||look==INTCONST)
     next();
   else
     error();
}
```

if-stat → **if** *expr* **then** { *stat* }$^+$ [**else** { *stat* }$^+$] **endif**
loop-stat → **loop** { *stat* }$^+$ **endloop**
assign-stat → **id** **:=** *expr*
expr → **id** | **boolconst** | **intconst**

Exercise 21

Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase terminates with EOF.

$$S \rightarrow \mathbf{a}\ A \mid \mathbf{b}\ B \mid \mathbf{c}$$
$$A \rightarrow \mathbf{b}\ \{B\ \mathbf{b}\}^{+}\ [\mathbf{c}\ B]\ \mathbf{a}$$
$$B \rightarrow \mathbf{d}\ (A \mid B \mid \mathbf{c}\ )$$

We assume the following auxiliary functions (whose code is not required):

- `match(symb)`: checks the equality of the lookahead symbol and `symb`, and reads the next input symbol;
- `next()`: reads the next input symbol.

Exercise 21

Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase terminates with EOF.

$S \rightarrow \mathbf{a}\ A \mid \mathbf{b}\ B \mid \mathbf{c}$
$A \rightarrow \mathbf{b}\ \{B\ \mathbf{b}\}^{+}\ [\mathbf{c}\ B]\ \mathbf{a}$
$B \rightarrow \mathbf{d}\ (A \mid B \mid \mathbf{c})$

```
parse(){next(); S(); match(EOF);}

S()
{  if(lookahead=='a'){next(); A();}
   else if(lookahead=='b'){next(); B();}
   else match('c');
}

A()
{
   match('b');
   do {B(); match('b');} while(lookahead=='d');
   if(lookahead=='c'){next(); B();}
   match('a');
}

B()
{
   match('d');
   if(lookahead=='b') A();
   else if(lookahead=='d') B();
   else match('c');
}
```

Exercise 22

Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase terminates with EOF.

program→ {*stat*}$^+$
stat → (*var-decl* | *proc-decl* | *if-stat* | *while-stat* | *assign-stat* | *call*) **;**
var-decl → *type* **id** {**,** **id**}
type → **int** | **string** | **bool**
proc-decl → **procedure id (** **id** {**,** **id**} **)** {*stat*}$^+$ **end**
if-stat → **if** *expr* **then** {*stat*}$^+$ { **elsif** *expr* **then** {*stat*}$^+$ } [**else** {*stat*}$^+$] **end**
expr → **id** | **boolconst** | **intconst** | **strconst**
while-stat → **while** *expr* **do** {*stat*}$^+$ **end**
assign-stat → **id =** *expr*
call → **call id (** *expr* {**,** *expr*} **)**

Exercise 22

Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase terminates with EOF.

```
parse(){next(); program();}

program()
{  do stat()
   while(look==INT || look==STRING || look==BOOL || look==PROCEDURE || look==IF ||
         look==WHILE || look==ID || look==CALL);
   match(EOF);
}

stat()
{  if(look==INT || look==STRING || look==BOOL) var-decl();
   else if(look==PROCEDURE) proc-decl();
   else if(look==IF) if-stat();
   else if(look==WHILE) while-stat();
   else if(look==ID) assign-stat();
   else if(look==CALL) call();
   else error();
   match(SEMICOLON);
}

var-decl(){type(); match(ID); while(look==COMMA){next(); match(ID);}}

type()
{if(look==INT || look==BOOL || look==STRING) next(); else error();}

proc-decl()
{  match(PROCEDURE); match(ID); match(LEFT); match(ID);
   while(look==COMMA){next(); match(ID);}
   match(RIGHT);
   do stat()
   while(look==INT || look==STRING || look==BOOL || look==PROCEDURE || look==IF ||
         look==WHILE || look==ID || look==CALL);
   match(END);
}
```

$program \rightarrow \{stat\}^+$
$stat \rightarrow (var\text{-}decl \mid proc\text{-}decl \mid if\text{-}stat \mid while\text{-}stat \mid assign\text{-}stat \mid call)$ **;**
$var\text{-}decl \rightarrow type$ **id** {**,** **id**}
$type \rightarrow$ **int** | **string** | **bool**
$proc\text{-}decl \rightarrow$ **procedure id (** **id** {**,** **id**} **)** $\{stat\}^+$ **end**

Exercise 22 (ii)

```
if-stat()
{  match(IF); expr(); match(THEN);
   do stat()
   while(look==INT || look==STRING || look==BOOL || look==PROCEDURE || look==IF || look==WHILE || look==ID || look==CALL);
   while(look==ELSIF)
   {  next(); expr(); match(THEN);
      do stat()
      while(look==INT || look==STRING || look==BOOL || look==PROCEDURE ||
            look==IF || look==WHILE || look==ID || look==CALL);
   }
   if(look==ELSE)
   { next();
     do stat()
     while(look==INT || look==STRING || look==BOOL || look==PROCEDURE || look==IF || look==WHILE || look==ID || look==CALL);
     match(END);
   }
}

expr()
{  if(look==ID || look==BOOLCONST ||look==INTCONST ||look==STRCONST) next();
   else error();
}

while-stat()
{ match(WHILE); expr(); match(DO);
  do stat()
     while(look==INT || look==STRING || look==BOOL || look==PROCEDURE ||
           look==IF || look==WHILE || look==ID || look==CALL);
  match(END);
}

assign-stat()
{  match(ID); match(ASSIGN); expr(); }

call()
{  match(CALL); match(ID); match(LEFT); expr();
   while(look==COMMA {next(); expr();}
   match(RIGHT);
}
```

if-stat → **if** *expr* **then** $\{stat\}^+$ { **elsif** *expr* **then** $\{stat\}^+$ } [**else** $\{stat\}^+$] **end**
expr → **id** | **boolconst** | **intconst** | **strconst**
while-stat → **while** *expr* **do** $\{stat\}^+$ **end**
assign-stat → **id** = *expr*
call → **call id (** *expr* {, *expr*} **)**

Exercise 23

Given the following BNF:

$$S \rightarrow S\ \mathbf{a}\ A \mid \mathbf{b}$$
$$A \rightarrow (\ S\ ) \mid S\ \mathbf{b}$$

we ask to:

- Trace the LL(1) parsing relevant to the phrase **babb**.
- Outline the syntax tree of the given phrase based on the rewriting actions generated at the previous point.

Exercise 23

Given the following BNF:

$S \rightarrow S\ \mathbf{a}\ A \mid \mathbf{b}$
$A \rightarrow (\ S\ ) \mid S\ \mathbf{b}$

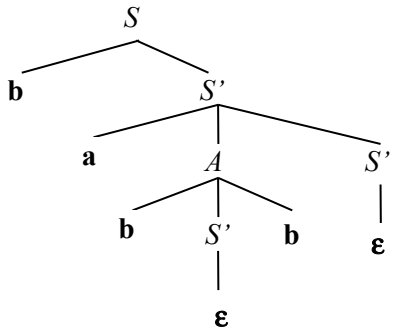
we ask to:

- Trace the LL(1) parsing relevant to the phrase **babb**.
- Outline the syntax tree of the given phrase based on the rewriting actions generated at the previous point.

$S \rightarrow\ \mathbf{b}\ S'$
$S' \rightarrow \mathbf{a}\ A\ S' \mid \varepsilon$
$A \rightarrow (\ S\ ) \mid \mathbf{b}\ S'\ \mathbf{b}$

FOLLOW(S') = {b,), $}

	a	b	(	)	$
S		$\mathbf{b}\ S'$			
S'	$\mathbf{a}\ A\ S'$	ε		ε	ε
A		$\mathbf{b}\ S'\ \mathbf{b}$	$(\ S\ )$		



Stack	*Input*	*Action*
S$	babb$	$S \rightarrow \mathbf{b}\ S'$
bS'$	babb$	match
S'$	abb$	$S' \rightarrow \mathbf{a}\ A\ S'$
aAS'$	abb$	match
AS'$	bb$	$A \rightarrow\ \mathbf{b}\ S'\ \mathbf{b}$
bS'bS'$	bb$	match
S'bS'$	b$	$S' \rightarrow\ \varepsilon$
bS'$	b$	match
S'$	$	$S' \rightarrow\ \varepsilon$
$	$	accept

Exercise 24

Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase ends with EOF.

program → {*stat*}
stat → (*def-stat* | *assign-stat* | *if-stat* | *while-stat*) **;**
def-stat → *type* **id** {**,** **id**}
type → **int** | **string** | **bool**
assign-stat → **id** **:=** *expr*
expr → **id** *op expr* | **(** *expr* **)** | **id** | **const**
op → **+** | **-** | ***** | **/** | **and** | **or**
if-stat → **if** *expr* **then** {*stat*}$^{+}$ [**else** {*stat*}$^{+}$] **endif**
while-stat → **while** *expr* **do** {*stat*}$^{+}$ **endwhile**

Exercise 24

```
parse()
{next(); program(); match(EOF);}

program()
{ while(look==INT || look== STRING || look==BOOL || look==ID|| look==IF || look==WHILE)
    stat();
}

stat()
{ if(look==INT || look== STRING || look==BOOL) def_stat();
  else if(look==ID) assign_stat();
  else if(look==IF) if_stat();
  else if(look==WHILE) while_stat();
  else error();
  match(SEMICOLON);
}

def_stat()
{ type(); match(ID);
  while(look==COMMA){next(); match(ID};}
}

type()
{ if(look==INT || look==STRING || look== BOOL) next();
  else error();
}

assign_stat()
{match(ID); match(ASSIGN); expr();}
```

program → {*stat*}
stat → (*def-stat* | *assign-stat* | *if-stat* | *while-stat*) **;**
def-stat → *type* **id** {**,** **id**}
type → **int** | **string** | **bool**
assign-stat → **id** := *expr*

Exercise 24 (ii)

```
expr()
{ if(look==ID){
    next();
    if(look==PLUS || look==MINUS || look==TIMES || look==SLASH || look==AND | look==OR)
      next();
      expr();
    }
  }
  else if(look == LEFT){next(); expr(); match(RIGHT);}
  else match(CONST);
}

if_stat()
{ match(IF); expr(); match(THEN);
  do stat();
  while(look==INT || look== STRING || look==BOOL || look==ID|| look==IF || look==WHILE);
  if(look==ELSE)
    do stat();
    while(look==INT || look== STRING || look==BOOL || look==ID|| look==IF || look==WHILE);
  match(ENDIF);
}

while_stat()
{ match(WHILE); expr(); match(DO);
  do stat();
  while(look==INT || look== STRING || look==BOOL || look==ID|| look==IF || look==WHILE);
  match(ENDWHILE);
}
```

expr → **id** *op expr* | **(** *expr* **)** | **id** | **const**
op → **+** | **-** | ***** | **/** | **and** | **or**
if-stat → **if** *expr* **then** $\{stat\}^+$ [**else** $\{stat\}^+$] **endif**
while-stat → **while** *expr* **do** $\{stat\}^+$ **endwhile**

Exercise 25

Given the following grammar **G** in BNF notation,

$$A \rightarrow A\ \mathbf{a}\ B \mid B$$
$$B \rightarrow A\ \mathbf{b} \mid \mathbf{a}$$

we ask to:

- Transform **G** into a non left-recursive grammar **G'** (equivalent to **G**);
- Generate the LL(1) parsing table of **G'**.
- Based on the parsing table, determine whether **G'** is LL(1).

Exercise 25

Given the following grammar **G** in BNF notation,

$$A \rightarrow A\ \mathbf{a}\ B \mid B$$
$$B \rightarrow A\ \mathbf{b} \mid \mathbf{a}$$

we ask to:

- Transform **G** into a non left-recursive grammar **G'** (equivalent to **G**);
- Generate the LL(1) parsing table of **G'**.
- Based on the parsing table, determine whether **G'** is LL(1).

$$A \rightarrow B\ A'$$
$$A' \rightarrow \mathbf{a}\ B\ A \mid \varepsilon$$
$$B \rightarrow A\ \mathbf{b} \mid \mathbf{a}$$

⇨

$$A \rightarrow B\ A'$$
$$A' \rightarrow \mathbf{a}\ B\ A' \mid \varepsilon$$
$$B \rightarrow B\ A'\ \mathbf{b} \mid \mathbf{a}$$

⇨

$$A \rightarrow B\ A'$$
$$A' \rightarrow \mathbf{a}\ B\ A' \mid \varepsilon$$
$$B \rightarrow \mathbf{a}\ B'$$
$$B' \rightarrow A'\ \mathbf{b}\ B' \mid \varepsilon$$

	a	**b**	**$**
A	$B\ A'$		
A'	$\mathbf{a}\ B\ A'$	ε	ε
B	$\mathbf{a}\ B'$		
B'	$A'\ \mathbf{b}\ B'$ ε	$A'\ \mathbf{b}\ B'$ ε	ε

$$FIRST(A) = FIRST(B) = \{\ \mathbf{a}\ \}$$
$$FIRST(A') = \{\ \mathbf{a}, \varepsilon\ \}$$
$$FIRST(B') = \{\ \mathbf{a}, \mathbf{b}, \varepsilon\ \}$$

$$FOLLOW(A) = \{\ \mathbf{\$}\ \}$$
$$FOLLOW(A') = \{\ \mathbf{b}, \mathbf{\$}\ \}$$
$$FOLLOW(B) = FOLLOW(B') = \{\ \mathbf{a}, \mathbf{b}, \mathbf{\$}\ \}$$

Exercise 26

Given the following grammar **G** in BNF notation,

$$A \rightarrow B\ \mathbf{a}\ C \mid C$$
$$B \rightarrow C\ \mathbf{b} \mid \mathbf{b}$$
$$C \rightarrow A\ \mathbf{b} \mid \mathbf{a}$$

we ask to:

- Transform **G** into a non left-recursive grammar **G'** (equivalent to **G**);
- Generate the complete LL(1) parsing table of **G'**.
- Based on the parsing table, establish whether G' is LL(1), providing relevant explanation.

Exercise 26

Given the following grammar **G** in BNF notation,

$$A \rightarrow B\ \mathbf{a}\ C \mid C$$
$$B \rightarrow C\ \mathbf{b} \mid \mathbf{b}$$
$$C \rightarrow A\ \mathbf{b} \mid \mathbf{a}$$

we ask to:

- Transform **G** into a non left-recursive grammar **G'** (equivalent to **G**);
- Generate the complete LL(1) parsing table of **G'**.
- Based on the parsing table, establish whether **G'** is LL(1), providing relevant explanation.

$$A \rightarrow B\ \mathbf{a}\ C \mid C$$
$$B \rightarrow C\ \mathbf{b} \mid \mathbf{b}$$
$$C \rightarrow B\ \mathbf{a}\ C\ \mathbf{b} \mid C\ \mathbf{b} \mid \mathbf{a}$$

⇨

$$A \rightarrow B\ \mathbf{a}\ C \mid C$$
$$B \rightarrow C\ \mathbf{b} \mid \mathbf{b}$$
$$C \rightarrow C\ \mathbf{b}\ \mathbf{a}\ C\ \mathbf{b} \mid \mathbf{b}\ \mathbf{a}\ C\ \mathbf{b} \mid C\ \mathbf{b} \mid \mathbf{a}$$

⇨

$$A \rightarrow B\ \mathbf{a}\ C \mid C$$
$$B \rightarrow C\ \mathbf{b} \mid \mathbf{b}$$
$$C \rightarrow \mathbf{b}\ \mathbf{a}\ C\ \mathbf{b}\ C' \mid \mathbf{a}\ C'$$
$$C' \rightarrow \mathbf{b}\ \mathbf{a}\ C\ \mathbf{b}\ C' \mid \mathbf{b}\ C' \mid \boldsymbol{\varepsilon}$$

$$FIRST(B) = FIRST(C) = \{\ \mathbf{a}, \mathbf{b}\ \}$$
$$FIRST(C') = \{\ \mathbf{b}, \varepsilon\ \}$$
$$FOLLOW(C') = FOLLOW(C) = \{\ \mathbf{b}, \$\ \}$$

	a	**b**	**$**
A	$B\ \mathbf{a}\ C$ C	$B\ \mathbf{a}\ C$ C	
B	$C\ \mathbf{b}$	$C\ \mathbf{b}$ $\mathbf{b}$	
C	$\mathbf{a}\ C'$	$\mathbf{b}\ \mathbf{a}\ C\ \mathbf{b}\ C'$	
C'		$\mathbf{b}\ \mathbf{a}\ C\ \mathbf{b}\ C'$ $\mathbf{b}\ C'$ ε	ε

Exercise 27

Given the following grammar G in BNF notation, we ask to transform G into an equivalent non left-recursive grammar G* and then, based on the complete parsing table, determine whether G* is LL(1).

$$S \rightarrow S\ \mathbf{a}\ T \mid T$$
$$T \rightarrow S\ \mathbf{b} \mid \mathbf{a} \mid \mathbf{b}$$

Exercise 27

Given the following grammar G in BNF notation, we ask to transform G into an equivalent non left-recursive grammar G* and then, based on the complete parsing table, determine whether G* is LL(1).

$S \rightarrow S\ \mathbf{a}\ T \mid T$
$T \rightarrow S\ \mathbf{b} \mid \mathbf{a} \mid \mathbf{b}$

⇨

$S \rightarrow T\ S'$
$S' \rightarrow \mathbf{a}\ T\ S' \mid \boldsymbol{\varepsilon}$
$T \rightarrow S\ \mathbf{b} \mid \mathbf{a} \mid \mathbf{b}$

⇨

$S \rightarrow T\ S'$
$S' \rightarrow \mathbf{a}\ T\ S' \mid \boldsymbol{\varepsilon}$
$T \rightarrow T\ S'\ \mathbf{b} \mid \mathbf{a} \mid \mathbf{b}$

⇨

$S \rightarrow T\ S'$
$S' \rightarrow \mathbf{a}\ T\ S' \mid \boldsymbol{\varepsilon}$
$T \rightarrow \mathbf{a}\ T' \mid \mathbf{b}\ T'$
$T' \rightarrow S'\ \mathbf{b}\ T' \mid \boldsymbol{\varepsilon}$

$FIRST(S') = \{\mathbf{a}, \boldsymbol{\varepsilon}\}$
$FIRST(\mathrm{T}') = \{\mathbf{a}, \mathbf{b}, \boldsymbol{\varepsilon}\}$
$FIRST(T\ S') = \{\mathbf{a}, \mathbf{b}\}$
$FIRST(S'\ \mathbf{b}\ T') = \{\mathbf{a}, \mathbf{b}\}$

$FOLLOW(S') = \{\$, \mathbf{b}\}$
$FOLLOW(T) = FOLLOW(T') = \{\mathbf{a}, \mathbf{b}, \$\}$

	a	**b**	**$**
S	$S \rightarrow T\ S'$	$S \rightarrow T\ S'$	
S'	$S' \rightarrow \mathbf{a}\ T\ S'$	$S' \rightarrow \boldsymbol{\varepsilon}$	$S' \rightarrow \boldsymbol{\varepsilon}$
T	$T \rightarrow \mathbf{a}\ T'$	$T \rightarrow \mathbf{b}\ T'$	
T'	$T' \rightarrow S'\ \mathbf{b}\ T'$ $T' \rightarrow \boldsymbol{\varepsilon}$	$T' \rightarrow S'\ \mathbf{b}\ T'$ $T' \rightarrow \boldsymbol{\varepsilon}$	$T' \rightarrow \boldsymbol{\varepsilon}$

Exercise 28

Given the following grammar G in BNF notation, we ask to transform G into an equivalent non left-recursive grammar G* and then, based on the complete parsing table, determine whether G* is LL(1).

$$A \rightarrow B\ \mathbf{a} \mid \boldsymbol{\varepsilon}$$
$$B \rightarrow C\ \mathbf{a}$$
$$C \rightarrow A\ \mathbf{b} \mid \boldsymbol{\varepsilon}$$

Exercise 28

Given the following grammar G in BNF notation, we ask to transform G into an equivalent non left-recursive grammar G* and then, based on the complete parsing table, determine whether G* is LL(1).

$$A \rightarrow B\,\mathbf{a} \mid \boldsymbol{\varepsilon}$$
$$B \rightarrow C\,\mathbf{a}$$
$$C \rightarrow A\,\mathbf{b} \mid \boldsymbol{\varepsilon}$$

⇨

$$A \rightarrow B\,\mathbf{a} \mid \boldsymbol{\varepsilon}$$
$$B \rightarrow C\,\mathbf{a}$$
$$C \rightarrow B\,\mathbf{a}\,\mathbf{b} \mid \mathbf{b} \mid \boldsymbol{\varepsilon}$$

⇨

$$A \rightarrow B\,\mathbf{a} \mid \boldsymbol{\varepsilon}$$
$$B \rightarrow C\,\mathbf{a}$$
$$C \rightarrow C\,\mathbf{a}\,\mathbf{a}\,\mathbf{b} \mid \mathbf{b} \mid \boldsymbol{\varepsilon}$$

⇨

$$A \rightarrow B\,\mathbf{a} \mid \boldsymbol{\varepsilon}$$
$$B \rightarrow C\,\mathbf{a}$$
$$C \rightarrow \mathbf{b}\,C' \mid C'$$
$$C' \rightarrow \mathbf{a}\,\mathbf{a}\,\mathbf{b}\,C' \mid \boldsymbol{\varepsilon}$$

$$FIRST(C') = \{\mathbf{a}, \boldsymbol{\varepsilon}\}$$
$$FIRST(C) = \{\mathbf{a}, \mathbf{b}, \boldsymbol{\varepsilon}\}$$
$$FIRST(B) = \{\mathbf{a}, \mathbf{b}\}$$

$$FOLLOW(A) = \{\$\}$$
$$FOLLOW(C) = FOLLOW(C') = \{\mathbf{a}\}$$

	a	**b**	**$**
A	$A \rightarrow B\,\mathbf{a}$	$A \rightarrow B\,\mathbf{a}$	$A \rightarrow \boldsymbol{\varepsilon}$
B	$B \rightarrow C\,\mathbf{a}$	$B \rightarrow C\,\mathbf{a}$	
C	$C \rightarrow C'$	$C \rightarrow \mathbf{b}\,C'$	
C'	$C' \rightarrow \mathbf{a}\,\mathbf{a}\,\mathbf{b}\,C'$ $C' \rightarrow \boldsymbol{\varepsilon}$		