Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

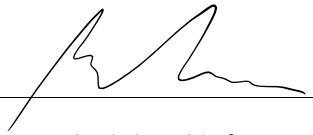
Last name:	Beardsmore	Student ID:	15504319
Other name(s):	Connor		
Unit name:	Programming Languages	Unit ID:	COMP2007
Lecturer / unit coordinator:	Stefan Prandl	Tutor:	Stefan Prandl
Date of submission:	03/11/2017	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature:  Date of signature: 03/11/2017

(By submitting this form, you indicate that you agree with all the above text.)

PL200 Report

Bison and Flex Parser

Connor Beardsmore - 15504319

Curtin University
Science and Engineering
Perth, Australia
November 2017

EBNF Specification

The full EBNF specification for *QUENYALGOL* is as listed below.
This EBNF follows the ISO BNF standard.

⟨*ident*⟩ ::= [a..z] { ⟨*ident*⟩ }

⟨*inumber*⟩ ::= [0..9] { ⟨*number*⟩ }

⟨*id_num*⟩ ::= [⟨*ident*⟩ | ⟨*number*⟩]

⟨*term*⟩ ::= ⟨*id_num*⟩ { ('`*`' | '`/`') ⟨*id_num*⟩ }

⟨*expression*⟩ ::= ⟨*term*⟩ { ('`+`' | '`-`') ⟨*term*⟩ }

⟨*statement_loop*⟩ ::= ⟨*statement*⟩ { '`;`' ⟨*statement*⟩ }

⟨*compound_statement*⟩ ::= '`BEGIN`' ⟨*statement_loop*⟩ '`END`'

⟨*for_statement*⟩ ::= '`FOR`' ⟨*ident*⟩ '`:=`' ⟨*expression*⟩ '`DO`' ⟨*statement_loop*⟩ '`END`' '`FOR`'

⟨*do_statement*⟩ ::= '`DO`' ⟨*statement_loop*⟩ '`WHILE`' ⟨*expression*⟩ '`END`' '`DO`'

⟨*while_statement*⟩ ::= '`WHILE`' ⟨*expression*⟩ '`DO`' ⟨*statement_loop*⟩ '`END`' '`WHILE`'

⟨*if_statement*⟩ ::= '`IF`' ⟨*expression*⟩ '`THEN`' ⟨*statement*⟩ '`END`' '`IF`'

⟨*procedure_call*⟩ ::= '`CALL`' ⟨*ident*⟩

⟨*assignment*⟩ ::= ⟨*ident*⟩ '`:=`' ⟨*expression*⟩

⟨*statement*⟩ ::= ⟨*assignment*⟩
| ⟨*procedure_call*⟩
| ⟨*if_statement*⟩
| ⟨*while_statement*⟩
| ⟨*do_statement*⟩
| ⟨*for_statement*⟩
| ⟨*compound_statement*⟩

⟨*implementation_part*⟩ ::= ⟨*statement*⟩

⟨*function_declaration*⟩ ::= '`FUNCTION`' ⟨*ident*⟩ '`;`' ⟨*block*⟩ '`;`'

⟨*procedure_declaration*⟩ ::= '`PROCEDURE`' ⟨*ident*⟩ '`;`' ⟨*block*⟩ '`;`'

⟨*specification_part*⟩ ::= {}
| '`CONST`' ⟨*constant_declaration*⟩
| '`VAR`' ⟨*variable_declaration*⟩
| ⟨*procedure_declaration*⟩
| ⟨*function_declaration*⟩

⟨*block*⟩ ::= ⟨*specification_part*⟩ ⟨*implementation_part*⟩

⟨*implementation_unit*⟩ ::= '`IMPLEMENTATION`' '`OF`' ⟨*ident*⟩ ⟨*block*⟩ '`.`'

⟨*range*⟩ ::= ⟨*number*⟩ '`..`' ⟨*number*⟩

⟨*array_type*⟩ ::= '`ARRAY`' ⟨*ident*⟩ '`[`' ⟨*range*⟩ '`]`' '`OF`' ⟨*type*⟩

⟨*range_type*⟩ ::= '`[`' ⟨*range*⟩ '`]`'

⟨*enumerated_type*⟩ ::= '`{`' ⟨*ident*⟩ { '`,`' ⟨*ident*⟩ } '`}`'

⟨*basic_type*⟩ ::= ⟨*ident*⟩
| ⟨*enumerated_type*⟩
| ⟨*range_type*⟩

⟨*type*⟩ ::= ⟨*basic_type*⟩
| ⟨*array_type*⟩

⟨*variable_declaration*⟩ ::= ⟨*ident*⟩ '`:`' ⟨*ident*⟩ { '`,`' ⟨*ident*⟩ '`:`' ⟨*ident*⟩ } '`;`'

⟨*constant_declaration*⟩ ::= ⟨*ident*⟩ '`=`' ⟨*number*⟩ { '`,`' ⟨*ident*⟩ '`:`' ⟨*number*⟩ } '`;`'

⟨*formal_parameters*⟩ ::= '`(`' ⟨*ident*⟩ { '`;`' ⟨*ident*⟩ } '`)`'

⟨*type_declaration*⟩ ::= '`TYPE`' ⟨*ident*⟩ '`:`' ⟨*type*⟩ '`;`'

⟨*function_interface*⟩ ::= '`FUNCTION`' ⟨*ident*⟩ [⟨*formal_parameters*⟩]

⟨*procedure_interace*⟩ ::= '`PROCEDURE`' ⟨*ident*⟩ [⟨*formal_parameters*⟩]

⟨*declaration_unit*⟩ ::= '`DECLARATION`' '`OF`' ⟨*ident*⟩ ['`CONST`' ⟨*constant_declaration*⟩] ['`VAR`' ⟨*variable_declaration*⟩] [⟨*type_declaration*⟩] [⟨*procedure_interface*⟩] [⟨*function_interface*⟩]

⟨*basic_program*⟩ ::= ⟨*declaration_unit*⟩ ⟨*implemenetation_unit*⟩

Parser Implementation

Lex

The lexical analyser code within *lexxy.l* defines a total of 43 tokens used in the *QUENYARGOL* language. Upon seeing any of the tokens within the language, the analyser will print the lexeme found and return it for use by the Yacc component. This is performed by the use of *TOKEN_MACRO*, to increase code reuse and keep the token rules simple.

All tokens are both preceded and succeeded by the underscore character, to avoid any conflicts with in-built keywords such as *BEGIN*. To avoid issues with the semicolon character being misinterpreted in the code, the token for this takes the form *_SEMICOLON_*, as is the case with *_DOUBLE_DOT_*.

The tokens for *_NUMBER_* and *_IDENT_* both use Unix style regular expressions to define their form. To ignore all whitespace in the code such as newlines and tabs, a similar regular expression is also used. The final token in the code is the *empty* rule, used for when no token is matched. Upon reaching this rule, a lexer error is printed to standard error.

For both simplicity and regularity, it was decided that identifiers would be limited to only lowercase letters while uppercase characters were reserved for the languages keywords. In compliance with this limitation, it is also not valid to mix numbers into identifiers for the simplicity of the parser.

Yacc

The Yacc parser defines a total of 44 grammar rules. Building the parser through Yacc was relatively simple overall. Converting the EBNF specification rules into a valid Yacc format required only small syntactical changes. A *GRAMMAR_MACRO* - similarly to the one used in the lex code - is utilized to print code whenever the parser matches a given rule.

Extra rules were required within the Yacc code to enable optional terms in rules and the use of repeating loops in terms. The optional rules use the *or* rule with an empty token to enable them to be neglected if required, such as in the *specification_part* rule. The looping rules such as *constant_loop* and *statement_loop* allow for infinite loops of declarations seperately by a single token. The extra rules were required over the EBNF specification in this set to allow for this recursive-like functionality.

Source Code

lexxy.l

```
/*****************************************************************************
* FILE: lexxy.l
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: PL200
* PURPOSE: Lex file for tokenizing a file
*   LAST MOD: 13/04/07
*   REQUIRES: stdio.h, yaccy.tab.h
*****************************************************************************/

/* DEFINITIONS */

%{
#include <stdio.h>
#include "yaccy.tab.h"

//MACRO to output the token found and return the token to yacc
#define TOKEN_MACRO(TYPE) { \
        printf("LEXER FOUND TOKEN: %s\n", yytext); \
        return TYPE; \
    }

//MACRO to output the token found, its value, and return the token to yacc
#define TOKEN_MACRO_VARIABLE(TYPE) { \
        printf("LEXER FOUND TOKEN: " #TYPE "with value: %s\n", yytext); \
        return TYPE; \
    }
%}

/*****************************************************************************/
/* RULES */

%%

"ARRAY"                 TOKEN_MACRO(_ARRAY_)
"BEGIN"                 TOKEN_MACRO(_BEGIN_)
"CALL"                  TOKEN_MACRO(_CALL_)
"CONST"                 TOKEN_MACRO(_CONST_)
"DECLARATION"           TOKEN_MACRO(_DECLARATION_)
"DO"                    TOKEN_MACRO(_DO_)
"END"                   TOKEN_MACRO(_END_)
"END DO"                TOKEN_MACRO(_END_DO_)
"END FOR"               TOKEN_MACRO(_END_FOR_)
"END IF"                TOKEN_MACRO(_END_IF_)
"END WHILE"             TOKEN_MACRO(_END_WHILE_)
"FOR"                   TOKEN_MACRO(_FOR_)
"FUNCTION"              TOKEN_MACRO(_FUNCTION_)
"IF"                    TOKEN_MACRO(_IF_)
"IMPLEMENTATION"        TOKEN_MACRO(_IMPLEMENTATION_)
"OF"                    TOKEN_MACRO(_OF_)
"PROCEDURE"             TOKEN_MACRO(_PROCEDURE_)
"THEN"                  TOKEN_MACRO(_THEN_)
"TYPE"                  TOKEN_MACRO(_TYPE_)
"VAR"                   TOKEN_MACRO(_VAR_)
"WHILE"                 TOKEN_MACRO(_WHILE_)
":="                TOKEN_MACRO(_ASSIGNMENT_)
";"                 TOKEN_MACRO(_SEMICOLON_)
".."              TOKEN_MACRO(_DOUBLE_DOT_)

"["                 TOKEN_MACRO('[')
"]"               TOKEN_MACRO(']')

```

```
62 "{"              TOKEN_MACRO('{')
63 "}"              TOKEN_MACRO('}')
64
65 "("              TOKEN_MACRO('(')
66 ")"              TOKEN_MACRO(')')
67
68 "."              TOKEN_MACRO('.')
69 ","              TOKEN_MACRO(',')
70 "="              TOKEN_MACRO('=')
71
72 "*"              TOKEN_MACRO('*')
73 "/"              TOKEN_MACRO('/')
74 "+"              TOKEN_MACRO('+')
75 "-"              TOKEN_MACRO('-')
76 ":"              TOKEN_MACRO(':')
77
78 [0-9]+           TOKEN_MACRO_VARIABLE(_NUMBER_)
79 [a-z]+           TOKEN_MACRO_VARIABLE(_IDENT_)
80 [ \t\n\r]+           // Ignore whitespace
81
82 .                    {
83                          // Empty rule
84                          fprintf(stderr, "LEXER ERROR: unexpected token - '%s' at '%d'\n
       ", yytext, *yytext);
85                          exit(1);
86                      }
87
88 %%
89
90 /*****************************************************************************/
91 /* USER ROUTINES */
92
93 int yywrap(void) { return 1; }
94
95 /*****************************************************************************/
```

yaccy.y

```
/*****************************************************************************
* FILE: yaccy.y
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: PL200
* PURPOSE: Yacc file for parser generation
*   LAST MOD: 27/09/17
*   REQUIRES: stdio.h, yaccy.tab.h
*****************************************************************************/

/* DEFINITIONS */

%{
#include <stdio.h>
#include "yaccy.tab.h"

int yylex();
int yyparse();

void yyerror(const char* msg) {
        fprintf(stderr, "yyeror: %s\n", msg);
   }

int main(void) {
       yyparse();
       return 0;
}

//MACRO to output the grammar matched
#define GRAMMAR_MACRO(TYPE) { \
        printf("\tYACC MATCHED RULE: " #TYPE "\n"); \
    }

//Enables more in-depth error messages from Yacc
#define YYERROR_VERBOSE
%}

/*****************************************************************************/

%token
    _ASSIGNMENT_
    _ARRAY_
    _BEGIN_
    _CALL_
    _CONST_
    _DECLARATION_
    _DO_
    _DOUBLE_DOT_
    _END_
    _END_DO_
    _END_FOR_
    _END_IF_
    _END_WHILE_
    _FOR_
    _FUNCTION_
    _IDENT_
    _IF_
    _IMPLEMENTATION_
    _NUMBER_
    _OF_
    _PROCEDURE_
    _SEMICOLON_
    _THEN_
    _TYPE_
```

```
64      _VAR_
65      _WHILE_
66
67  %start basic_program
68  %%
69
70  /*************************************************************************/
71  /* GRAMMAR RULES - TKN_PRIMARY */
72
73  basic_program:
74      declaration_unit implementation_unit
75      { GRAMMAR_MACRO(basic_program) };
76
77  /*************************************************************************/
78  /* DECLARATION UNIT */
79
80  opt_constant_declaration:
81      _CONST_ constant_declaration
82      { GRAMMAR_MACRO(opt_constant_declaration) }
83      | {};
84
85  opt_variable_declaration:
86      _VAR_ variable_declaration
87      { GRAMMAR_MACRO(opt_variable_declaration) }
88      | {};
89
90  opt_type_declaration:
91      type_declaration
92      { GRAMMAR_MACRO(opt_type_declaration) }
93      | {};
94
95  opt_procedure_interface:
96      procedure_interface
97      { GRAMMAR_MACRO(opt_procedure_interface) }
98      | {};
99
100 opt_function_interface:
101     function_interface
102     { GRAMMAR_MACRO(opt_function_interface) }
103     | {};
104
105 opt_formal_parameters:
106     formal_parameters
107     { GRAMMAR_MACRO(opt_formal_parameters) }
108     | {};
109
110 declaration_unit:
111     _DECLARATION_ _OF_ _IDENT_
112         opt_constant_declaration
113         opt_variable_declaration
114         opt_type_declaration
115         opt_procedure_interface
116         opt_function_interface
117     _DECLARATION_ _END_
118     { GRAMMAR_MACRO(declaration_unit) };
119
120 /*************************************************************************/
121 /* DECLARATIONS AND INTERFACES */
122
123 procedure_interface:
124     _PROCEDURE_ _IDENT_
125         opt_formal_parameters
126     { GRAMMAR_MACRO(procedure_interface) };
127
128 function_interface:
```

```
129     _FUNCTION_ _IDENT_
130         opt_formal_parameters
131     { GRAMMAR_MACRO(function_interface) };
132
133 type_declaration:
134     _TYPE_ _IDENT_ ':' type _SEMICOLON_
135     { GRAMMAR_MACRO(type_declaration) };
136
137 ident_loop_semicolon:
138     _IDENT_
139     | ident_loop_semicolon _SEMICOLON_ _IDENT_
140     { GRAMMAR_MACRO(ident_loop_semicolon) };
141
142 formal_parameters:
143     '(' ident_loop_semicolon ')'
144     { GRAMMAR_MACRO(formal_parameters) };
145
146 constant_loop:
147     _IDENT_ '=' _NUMBER_
148     | constant_loop ',' _IDENT_ '=' _NUMBER_
149     { GRAMMAR_MACRO(constant_loop) };
150
151 constant_declaration:
152     constant_loop _SEMICOLON_
153     { GRAMMAR_MACRO(constant_declaration) };
154
155 variable_loop:
156     _IDENT_ ':' _IDENT_
157     | variable_loop ',' _IDENT_ ':' _IDENT_
158     { GRAMMAR_MACRO(variable_loop) };
159
160 variable_declaration:
161     variable_loop _SEMICOLON_
162     { GRAMMAR_MACRO(variable_declaration) };
163
164 /************************************************************************/
165 /* TYPES */
166
167 type:
168     basic_type
169     { GRAMMAR_MACRO(type) }
170     | array_type
171     { GRAMMAR_MACRO(type) };
172
173 basic_type:
174     _IDENT_
175     { GRAMMAR_MACRO(basic_type) }
176     | enumerated_type
177     { GRAMMAR_MACRO(basic_type) }
178     | range_type
179     { GRAMMAR_MACRO(basic_type) };
180
181 ident_loop_comma:
182     _IDENT_
183     | ident_loop_comma ',' _IDENT_
184     { GRAMMAR_MACRO(ident_loop_comma) };
185
186 enumerated_type:
187     '{' ident_loop_comma '}'
188     { GRAMMAR_MACRO(enumerated_type) };
189
190 range_type:
191     '[' range ']'
192     { GRAMMAR_MACRO(range_type) };
193
```

```
194 array_type:
195     _ARRAY_ _IDENT_ '[' range ']' _OF_ type
196     { GRAMMAR_MACRO(array_type) };
197
198 range:
199     _NUMBER_ _DOUBLE_DOT_ _NUMBER_
200     { GRAMMAR_MACRO(range) };
201
202 /************************************************************************/
203 /* IMPLEMENTATION AND SPECIFICATION */
204
205 implementation_unit:
206     _IMPLEMENTATION_ _OF_ _IDENT_ block '.'
207     { GRAMMAR_MACRO(implementation_unit) };
208
209 block:
210     specification_part implementation_part
211     { GRAMMAR_MACRO(block) };
212
213 specification_part:
214     _CONST_ constant_declaration
215     { GRAMMAR_MACRO(specification_part) }
216     | _VAR_ variable_declaration
217     { GRAMMAR_MACRO(specification_part) }
218     | procedure_declaration
219     { GRAMMAR_MACRO(specification_part) }
220     | function_declaration
221     { GRAMMAR_MACRO(specification_part) }
222     | {}
223
224 procedure_declaration:
225     _PROCEDURE_ _IDENT_ _SEMICOLON_ block _SEMICOLON_
226     { GRAMMAR_MACRO(procedure_declaration) };
227
228 function_declaration:
229     _FUNCTION_ _IDENT_ _SEMICOLON_ block _SEMICOLON_
230     { GRAMMAR_MACRO(function_declaration) };
231
232 implementation_part:
233     statement
234     { GRAMMAR_MACRO(implementation_part) };
235
236 /************************************************************************/
237 /* STATEMENTS */
238
239 statement:
240     assignment
241     { GRAMMAR_MACRO(statement) }
242     | procedure_call
243     { GRAMMAR_MACRO(statement) }
244     | if_statement
245     { GRAMMAR_MACRO(statement) }
246     | while_statement
247     { GRAMMAR_MACRO(statement) }
248     | do_statement
249     { GRAMMAR_MACRO(statement) }
250     | for_statement
251     { GRAMMAR_MACRO(statement) }
252     | compound_statement
253     { GRAMMAR_MACRO(statement) };
254
255 assignment:
256     _IDENT_ _ASSIGNMENT_ expression
257     { GRAMMAR_MACRO(assignment) };
258
```

```
259 procedure_call:
260     _CALL_ _IDENT_
261     { GRAMMAR_MACRO(procedure_call) };
262
263 if_statement:
264     _IF_ expression _THEN_ statement _END_IF_
265     { GRAMMAR_MACRO(if_statement) };
266
267 while_statement:
268     _WHILE_ expression _DO_ statement_loop _END_WHILE_
269     { GRAMMAR_MACRO(while_statement) };
270
271 do_statement:
272     _DO_ statement_loop _WHILE_ expression _END_DO_
273     { GRAMMAR_MACRO(do_statement) };
274
275 for_statement:
276     _FOR_ _IDENT_ _ASSIGNMENT_ expression _DO_ statement_loop _END_FOR_
277     { GRAMMAR_MACRO(for_statement) };
278
279 compound_statement:
280     _BEGIN_ statement_loop _END_
281     { GRAMMAR_MACRO(compound_statement) };
282
283 statement_loop:
284     statement
285     | statement_loop _SEMICOLON_ statement
286     { GRAMMAR_MACRO(statement_loop) };
287
288 /*************************************************************************/
289 /* EXPRESSIONS, TERMS AND IDENTIFIERS */
290
291 expression :
292     expression_loop
293     { GRAMMAR_MACRO(expression) };
294
295 expression_loop:
296     term
297     | expression_loop
298     '+'
299     term
300     { GRAMMAR_MACRO(expression_loop) }
301     | expression_loop
302     '-'
303     term
304     { GRAMMAR_MACRO(expression_loop) };
305
306 term:
307     term_loop
308     { GRAMMAR_MACRO(term) };
309
310 term_loop:
311     id_num
312     | term_loop
313     '*'
314     id_num
315     { GRAMMAR_MACRO(term_loop) }
316     | term_loop
317     '/'
318     id_num
319     { GRAMMAR_MACRO(term_loop) };
320
321 id_num:
322     _IDENT_
323     { GRAMMAR_MACRO(id_num) }
```

```
324     | _NUMBER_
325     { GRAMMAR_MACRO(id_num) };
326 
327 /*************************************************************************/
```

References

Levine, John, Tony Mason, and Doug Brown. 1992. *Lex & Yacc*. 2nd. O'Reilly.

Sebesta, Robert W. 2016. *Concepts of Programming Languages*. 11th. USA: Addison-Wesley Publishing Company. ISBN: 0136073476.