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 *QUENYARGOL* 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⟩ ';'
```

$\langle$*function_interface*$\rangle$ ::= ‘`FUNCTION`’ $\langle$*ident*$\rangle$ [$\langle$*formal_parameters*$\rangle$]

$\langle$*procedure_interace*$\rangle$ ::= ‘`PROCEDURE`’ $\langle$*ident*$\rangle$ [$\langle$*formal_parameters*$\rangle$]

$\langle$*declaration_unit*$\rangle$ ::= ‘`DECLARATION`’ ‘`OF`’ $\langle$*ident*$\rangle$ [‘`CONST`’ $\langle$*constant_declaration*$\rangle$] [‘`VAR`’ $\langle$*variable_declaration*$\rangle$] [$\langle$*type_declaration*$\rangle$] [$\langle$*procedure_interface*$\rangle$] [$\langle$*function_interface*$\rangle$] ‘`DECLARATION`’ ‘`END`’

$\langle$*basic_program*$\rangle$ ::= $\langle$*declaration_unit*$\rangle$ $\langle$*implemenetation_unit*$\rangle$

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 parser. 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 also 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 (Levine, Mason, and Brown 1992). Upon reaching this rule, a lexer error is printed to standard error.

For both simplicity and the regularity principle (Sebesta 2016), 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 the optional terms 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 "y.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(']')

```

```
"{"          TOKEN_MACRO('{')
"}"          TOKEN_MACRO('}')

"("          TOKEN_MACRO('(')
")"          TOKEN_MACRO(')')

"."          TOKEN_MACRO('.')
","          TOKEN_MACRO(',')
"="          TOKEN_MACRO('=')

"*"          TOKEN_MACRO('*')
"/"          TOKEN_MACRO('/')
"+"          TOKEN_MACRO('+')
"-"          TOKEN_MACRO('-')
":"          TOKEN_MACRO(':')

[0-9]+          TOKEN_MACRO_VARIABLE(_NUMBER_)
[a-z]+          TOKEN_MACRO_VARIABLE(_IDENT_)
[ \t\n\r]+          // Ignore whitespace

.                   {
                        // Empty rule
                        fprintf(stderr, "LEXER ERROR: unexpected token - '%s' at '%d'\n
    ", yytext, *yytext);
                        exit(1);
                    }

%%

/*************************************************************************/
/* USER ROUTINES */

int yywrap(void) { return 1; }

/*************************************************************************/
```

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 "y.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_
    _VAR_
    _WHILE_

%start basic_program
%%

/***************************************************************************/
/* GRAMMAR RULES - TKN_PRIMARY */

basic_program:
    declaration_unit implementation_unit
    { GRAMMAR_MACRO(basic_program) };

/***************************************************************************/
/* DECLARATION UNIT */

opt_constant_declaration:
    _CONST_ constant_declaration
    { GRAMMAR_MACRO(opt_constant_declaration) }
    | {};

opt_variable_declaration:
    _VAR_ variable_declaration
    { GRAMMAR_MACRO(opt_variable_declaration) }
    | {};

opt_type_declaration:
    type_declaration
    { GRAMMAR_MACRO(opt_type_declaration) }
    | {};

opt_procedure_interface:
    procedure_interface
    { GRAMMAR_MACRO(opt_procedure_interface) }
    | {};

opt_function_interface:
    function_interface
    { GRAMMAR_MACRO(opt_function_interface) }
    | {};

opt_formal_parameters:
    formal_parameters
    { GRAMMAR_MACRO(opt_formal_parameters) }
    | {};

declaration_unit:
    _DECLARATION_ _OF_ _IDENT_
        opt_constant_declaration
        opt_variable_declaration
        opt_type_declaration
        opt_procedure_interface
        opt_function_interface
    _DECLARATION_ _END_
    { GRAMMAR_MACRO(declaration_unit) };

/***************************************************************************/
/* DECLARATIONS AND INTERFACES */

procedure_interface:
    _PROCEDURE_ _IDENT_
        opt_formal_parameters
    { GRAMMAR_MACRO(procedure_interface) };

function_interface:
    _FUNCTION_ _IDENT_
        opt_formal_parameters
    { GRAMMAR_MACRO(function_interface) };

type_declaration:
    _TYPE_ _IDENT_ ':' type _SEMICOLON_
    { GRAMMAR_MACRO(type_declaration) };

ident_loop_semicolon:
    _IDENT_
    | ident_loop_semicolon _SEMICOLON_ _IDENT_
    { GRAMMAR_MACRO(ident_loop_semicolon) };

formal_parameters:
    '(' ident_loop_semicolon ')'
    { GRAMMAR_MACRO(formal_parameters) };

constant_loop:
    _IDENT_ '=' _NUMBER_
    | constant_loop ',' _IDENT_ '=' _NUMBER_
    { GRAMMAR_MACRO(constant_loop) };

constant_declaration:
    constant_loop _SEMICOLON_
    { GRAMMAR_MACRO(constant_declaration) };

variable_loop:
    _IDENT_ ':' _IDENT_
    | variable_loop ',' _IDENT_ ':' _IDENT_
    { GRAMMAR_MACRO(variable_loop) };

variable_declaration:
    variable_loop _SEMICOLON_
    { GRAMMAR_MACRO(variable_declaration) };

/**********************************************************************/
/* TYPES */

type:
    basic_type
    { GRAMMAR_MACRO(type) }
    | array_type
    { GRAMMAR_MACRO(type) };

basic_type:
    _IDENT_
    { GRAMMAR_MACRO(basic_type) }
    | enumerated_type
    { GRAMMAR_MACRO(basic_type) }
    | range_type
    { GRAMMAR_MACRO(basic_type) };

ident_loop_comma:
    _IDENT_
    | ident_loop_comma ',' _IDENT_
    { GRAMMAR_MACRO(ident_loop_comma) };

enumerated_type:
    '{' ident_loop_comma '}'
    { GRAMMAR_MACRO(enumerated_type) };

range_type:
    '[' range ']'
    { GRAMMAR_MACRO(range_type) };

```

```
array_type:
    _ARRAY_ _IDENT_ '[' range ']' _OF_ type
    { GRAMMAR_MACRO(array_type) };

range:
    _NUMBER_ _DOUBLE_DOT_ _NUMBER_
    { GRAMMAR_MACRO(range) };

/**********************************************************************/
/* IMPLEMENTATION AND SPECIFICATION */

implementation_unit:
    _IMPLEMENTATION_ _OF_ _IDENT_ block '.'
    { GRAMMAR_MACRO(implementation_unit) };

block:
    specification_part implementation_part
    { GRAMMAR_MACRO(block) };

specification_part:
    _CONST_ constant_declaration
    { GRAMMAR_MACRO(specification_part) }
    | _VAR_ variable_declaration
    { GRAMMAR_MACRO(specification_part) }
    | procedure_declaration
    { GRAMMAR_MACRO(specification_part) }
    | function_declaration
    { GRAMMAR_MACRO(specification_part) }
    | {}

procedure_declaration:
    _PROCEDURE_ _IDENT_ _SEMICOLON_ block _SEMICOLON_
    { GRAMMAR_MACRO(procedure_declaration) };

function_declaration:
    _FUNCTION_ _IDENT_ _SEMICOLON_ block _SEMICOLON_
    { GRAMMAR_MACRO(function_declaration) };

implementation_part:
    statement
    { GRAMMAR_MACRO(implementation_part) };

/**********************************************************************/
/* STATEMENTS */

statement:
    assignment
    { GRAMMAR_MACRO(statement) }
    | procedure_call
    { GRAMMAR_MACRO(statement) }
    | if_statement
    { GRAMMAR_MACRO(statement) }
    | while_statement
    { GRAMMAR_MACRO(statement) }
    | do_statement
    { GRAMMAR_MACRO(statement) }
    | for_statement
    { GRAMMAR_MACRO(statement) }
    | compound_statement
    { GRAMMAR_MACRO(statement) };

assignment:
    _IDENT_ _ASSIGNMENT_ expression
    { GRAMMAR_MACRO(assignment) };
```

```
procedure_call:
    _CALL_ _IDENT_
    { GRAMMAR_MACRO(procedure_call) };

if_statement:
    _IF_ expression _THEN_ statement _END_IF_
    { GRAMMAR_MACRO(if_statement) };

while_statement:
    _WHILE_ expression _DO_ statement_loop _END_WHILE_
    { GRAMMAR_MACRO(while_statement) };

do_statement:
    _DO_ statement_loop _WHILE_ expression _END_DO_
    { GRAMMAR_MACRO(do_statement) };

for_statement:
    _FOR_ _IDENT_ _ASSIGNMENT_ expression _DO_ statement_loop _END_FOR_
    { GRAMMAR_MACRO(for_statement) };

compound_statement:
    _BEGIN_ statement_loop _END_
    { GRAMMAR_MACRO(compound_statement) };

statement_loop:
    statement
    | statement_loop _SEMICOLON_ statement
    { GRAMMAR_MACRO(statement_loop) };

/************************************************************************/
/* EXPRESSIONS, TERMS AND IDENTIFIERS */

expression :
    expression_loop
    { GRAMMAR_MACRO(expression) };

expression_loop:
    term
    | expression_loop
    '+'
    term
    { GRAMMAR_MACRO(expression_loop) }
    | expression_loop
    '-'
    term
    { GRAMMAR_MACRO(expression_loop) };

term:
    term_loop
    { GRAMMAR_MACRO(term) };

term_loop:
    id_num
    | term_loop
    '*'
    id_num
    { GRAMMAR_MACRO(term_loop) }
    | term_loop
    '/'
    id_num
    { GRAMMAR_MACRO(term_loop) };

id_num:
    _IDENT_
    { GRAMMAR_MACRO(id_num) }
    | _NUMBER_
    { GRAMMAR_MACRO(id_num) };

/************************************************************************/
```

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.