Parser

Parser:

In this part of the project, you are going to make a parser which can parse the language that will be discussed in this document. You have been taught pgen and you'll be given links to the latest version of the project. You are also able to use CUP parser in order to connect JFlex that you have been taught before, and which you used in the first phase of your project.

Reserved Words :

Begin, bool, break, case, char, const, continue, default, double, else, end, extern, false, function, float, for, if, int, long , return, record, sizeof, string, switch, true

Numbers :

Numbers are either integers that can be either 32 bits or 64 bits or they are real numbers which can be written like 1., .1, 1.1, 1e-2. Keep in mind that in the next phase you should be able to do arithmetic function with all the possible mixes of this numbers.

Comments :

Comments can be either in one line which will start like: ## or it can expand into multiple lines like /# … #/ .

Symbols :

==	equal
!=	Not equal
<=	Less equal
<	less
>	greater
>=	Greater equal
=	assignment
not	not
~	Bitwise negation
&	Arithmatic and

and	Logical and
|	Arithmetic or
^	Logical/Arithmetic Xor
*	Production
+	add
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
/	Div
%	mod
{ }	Opening and Closing Curly Braces
( )	Opening and Closing Parenthesis
.	dot
,	comma
:	colon
;	Semicolon
[ ]	Opening and Closing Brace

Strings:

Strings are made up of Ascii characters. You can also use Unicode. The longest Strings can have a length of 60000. You should also be able to read the strings and characters which you implemented in your last project.

Variables:

Variables have the same rules of variables in c++.

Grammer:

⟨*program*⟩ → {⟨*var_dcl*⟩ * | ⟨*func_extern*⟩ * |⟨*struct_dec*⟩ * } +

⟨*func_extern*⟩ → ⟨*func_dcl*⟩ |⟨*extern_dcl*⟩

⟨*func_dcl*⟩ → ***function*** ⟨*type*⟩ ***id*** ([⟨*arguments*⟩]) ; | ***function*** ⟨*type*⟩ ***id*** (⟨*arguments*⟩]) ⟨*block*⟩

⟨*extern_dcl*⟩ → ***extern*** ⟨*type*⟩ ***id*** ;

⟨*arguments*⟩ → ⟨*type*⟩ ***id*** [{ '[' ']' } +] [, ⟨*arguments*⟩]

⟨*type*⟩ → ***int*** | ***bool*** | ***float*** | ***long*** | ***char*** | ***double*** | ***id*** | ***string*** | ***void*** | ***auto***

⟨*struct_dec*⟩ → ***record id begin*** ⟨*var_dcl*⟩ + ***end record*** ;

⟨*var_dcl*⟩ → [***const***] ⟨*type*⟩ ⟨*var_dcl_cnt*⟩ [,⟨*var_dcl_cnt*⟩] * ;

⟨*var_dcl_cnt*⟩ → ⟨*variable*⟩ [= {⟨*expr*⟩}]

⟨*block*⟩ → ***begin*** { ⟨*var_dcl*⟩ | ⟨*statement*⟩ } * ***end***

⟨*statement*⟩ → ⟨*assignment*⟩ ;

| ⟨*method_call*⟩ ;

| ⟨*cond_stmt*⟩

| ⟨*loop_stmt*⟩

| ***return*** [⟨*expr*⟩];

| ***break*** ;

| ***continue*** ;

⟨*assignment*⟩ → ⟨*variable*⟩ {= | += | −=| *= | / =} ⟨*expr*⟩

⟨*method_call*⟩ → ***id*** ([⟨*parameters*⟩])

⟨*parameters*⟩ → ⟨*expr*⟩

| ⟨*expr*⟩, ⟨*parameters*⟩

⟨*cond_stmt*⟩ → ***if*** (⟨*expr*⟩ ⟨*block*⟩ [***else*** ⟨*block*⟩]

| ***switch*** (***id***) ***of*** : ***begin*** [{***case int_const*** : ⟨*block*⟩}] * ***default***: ⟨*block*⟩ ***end***

⟨*loop_stmt*⟩ → ***for*** ([⟨*assignment*⟩] ; ⟨*expr*⟩ ; [⟨*assignment*⟩ | ⟨*expr*⟩]) ⟨*block*⟩

| ***repeat*** ⟨*block*⟩ ***until*** (⟨*expr*⟩);

| ***foreach***(***id in id***) ⟨*block*⟩

⟨*expr*⟩ → ⟨*expr*⟩ ⟨*binary_op*⟩ ⟨*expr*⟩

| (⟨*expr*⟩)

| ⟨*method_call*⟩

| ⟨*variable*⟩

| ⟨*const_val*⟩

| − ⟨*expr*⟩

| ~ ⟨*expr*⟩

| ***sizeof***(⟨*type*⟩)

⟨*variable*⟩ → ***id*** [{ '[' ⟨*expr*⟩ ']' } +]

| ⟨*variable*⟩ . id

|~⟨*variable*⟩

| − −⟨*variable*⟩

| + +⟨*variable*⟩

|⟨*variable*⟩ − −

|⟨*variable*⟩ + +

⟨*binary_op*⟩ → ⟨*arithmatic*⟩

| ⟨*conditional*⟩

⟨*arithmatic*⟩ → +

| −

| *

| /

| %

| &

| '|'

| ^

⟨*conditional*⟩ → ==

| ! =

| >=

|<=

| <

| >

| and

| or

| not

⟨*const_val*⟩ → ***int_const***

| ***real_const***

| ***char_const***

|***bool_const***

| ***string_const***

| ***long_cons***t

Notice :

Keep in mind that PGEN is a LL(1) so it is a top-down parser, and there will be more conflicts that need to be resolved, and in contrast CUP is an LALR(1) and is a bottom-up parser and even though you will see fewer conflicts you will probably need to implement an abstract tree.

Also, the Implementation of Foreach is optional and will provide you extra marks.