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, auto

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
&	Arithmetic and

and	Logical and
or	Logical or
|	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***;

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

⟨*var_dcl_cnt*⟩ → ***id*** [= {⟨*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*⟩

| ***not*** ⟨*expr*⟩

| ***sizeof***(⟨*type*⟩)

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

| ⟨*variable*⟩ ***.id***

| − −⟨*variable*⟩

| + +⟨*variable*⟩

|⟨*variable*⟩ − −

|⟨*variable*⟩ + +

⟨*binary_op*⟩ → ⟨*arithmatic*⟩

| ⟨*conditional*⟩

⟨*arithmatic*⟩ → +

| −

| *

| /

| %

| &

| '|'

| ^

⟨*conditional*⟩ → ==

| ! =

| >=

|<=

| <

| >

| and

| or

⟨*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.

Semantical Notes:

As you may have noticed this language shares a lot with c++. So if something is left unexplained here, please refer to c++ language rules.

Just like c++,every program's starting point is main function that has no input arguments and returns an integer (status code of the program). Missing main function will result to a semantical error.

Program's Structure

Types:

Each type is similar to c++ types. With one exception, in this language the key word to declare an structure is *record*. And remember that when declaring arrays, all dimensions are integers.

Functions:

- Each program should have exactly one *main* function
- Variables are passed to function using call-by-reference method. So changes inside the function will affect the variables outside. However every functions output is just value.
- In this language overloading is allowed.
- Local variable's names are not allowed to be the same as other function names or reserved words or variable types.
- In this language it is possible to use C standard library. These function's names are declared in the program using *extern* keyword.
- Passing parameters to functions should be based on C Standard

Variables:

- Variables are primitive or user-defined and it is possible to have arrays of these types. Variables are either local or global. Local variables are only accessible from their respective code block, but global variables are accessible from every block.
- Global variables cannot be initialized with values
- All instances of user-defined variables are accessed via *reference*
- If a variable type is *auto*, this variable's type is figured out after first value assignment to it. (Read more about *Type Inference*
- *Constant* variables must be initialized when declared and its value will not change during the life of the program. Constant variables cannot be global
- String variables are immutable and must be initialized when declared. A *record* type cannot contain string type. String is stored in the memory as a array of characters with an ending cell containing '\0'.

Other Notes :

- Assignment operator works like C++, however assignment does not work for array's themselves , but it works for their elements
- Every primitive type can be casted. When casting an integer to char, keep in mind that integer's least valuable byte will be transferred to char
- Allowed casting scenarios are (int, float) , (int, char), (int, double), (float, double) and vise versa. Booleans can be casted to everything (just like C++). other casting scenarios are treated as errors.
- When assigning two instances of a same record, right hand side's values is copied to left hand side's fields
- In conditional statements 0 is deemed false and 1 is deemed true
- Bitwise operations are applicable only to integers
- *Break* and *Continue* statements are usable only in loop bodies.
- In *switch-case* statements exactly one case statement should run and default statement is mandatory
- ++ and - - operators are only valid for char and integer data types. Both prefix and postfix types of these operators will **apply their change after the end of the statement**
- In this language len() will return a string's length (C standard library
- input and output functions are printf() and scanf() (C standard library
- **Having these functions are mandatory and they should be reference from C++ standard library when converting code to assembly**
- *Foreach* command is just used for arrays
- Memory allocation of all local variables should be done in memory stack

Grading:

- Scanner : 15 points
- Parser : 30 points
- Code Generation : 55 points