**Scanner**

Character Literal:

Characters of the following types are handled according to C++ standard:

* char
* char16\_t
* char32\_t
* wchar\_t

The ***wchar\_t*** size as in C++ standard is implementation defined, and is set to 32 bits in the implementation of this compiler.

A character literal that does not begin with ***L***, ***u8***, ***u***, or ***U*** which contains a single character that is not representable in ***char*** has a value of equal to its lower 1 byte.

A multi-character literal that does not begin with ***L***, ***u8***, ***u***, or ***U*** which contains multiple character literals representable in ***char*** character set has integer type and has a value equal to its characters considered as a digit in base 256.

A multi-character literal that begins with ***L*** prefix which contains multiple character literals representable in ***wchar\_t*** character set has integer type and has a value equal to its first character.

A single character literal that begins with ***u8*, *u****,*or ***U*** prefix which contains a character literal that is not representable in its character set regarding to its prefix is an ill-formed character and will cause the scanner to return the symbol ***InvalidCharacter***.

A multi-character literal that begins with ***u8*, *u****,*or ***U*** prefix which contains multiple character literals is an ill-formed character and will cause the scanner to return the symbol ***InvalidCharacter***.

A multi-character literal that begins with ***L*** prefix or without any prefix which contains a character literal that is not representable in its character set regarding to its prefix is an ill-formed character and will cause the scanner to return the symbol ***InvalidCharacter***.

An octal number character can at most have 3 octal digits.

An octal number character with digit out of range [0, 7] will cause scanner to return the symbol ***InvalidOctalCharacterLiteral***.

An escape sequence ***\u*** that is not followed with exactly 4 hexadecimal digits or an escape sequence ***\U*** that is not followed with exactly 4 hexadecimal digits will cause the scanner to return the symbol ***InvalidUniversalUnicodeCharacterLiteral***.

A hexadecimal character can at most have 8 hexadecimal digits.

An escape sequence \x that is not followed by hexadecimal digits will cause the scanner to return the symbol **InvalidHexadecimalCharacterLiteral**.

An unbound character literal will cause the scanner to return the symbol ***CharacterMissingEndIndicator***.

If the escape sequence doesn’t fit in any of the ones defined by C++ standard the scanner will return the symbol ***InvalidEscapeSequence***.

Upon reading a valid character literal scanner will return the symbol ***CharacterConstant*** and the character itself and the size and type is accessible via scanner methods.

Upon reading a valid multi-character character literal scanner will return the symbol IntegerConstant and the value is written to token as a string in base 10 and is accessible via scanner methods.

String Literal:

Strings of the following types are handled according to C++ standard:

* char\*
* char16\_t\*
* char32\_t\*
* wchar\_t\*

Each character inside a string is treated like the aforementioned character literals.

An unbound string causes the scanner to return the symbol ***StringMissingEndIndicator***.

Upon reading a valid sting literal the scanner will return the symbol ***StringConstant*** and the value string type and size are accessible via scanner methods.

Integer Literal:

Strings of the following types are handled according to C++ standard:

* int
* long
* long long
* unsigned int
* unsigned long
* unsigned long long

An integer literal type is indicated by its size. The type of an integer is the smallest signed and if not unsigned type that can store the value of the integer literal.

In this implementation ***int*** and ***long*** are 4 bytes and ***long long*** is 8 bytes, respectively the unsigned counterparts.

The sign of the integer literal is decided later at parser.

Upon reading a valid integer literal the scanner will return the symbol IntegerConstant and the value and size of the literal is accessible using scanner methods.

Real number literals:

There are two types of real number literals:

* float
* double

A real number that is not a scientific representation can take 3 forms:

* *decimal-sequence* . *decimal-sequence*
* *decimal-sequence* .
* . *decimal-sequence*

A scientific representation has two forms:

* *real-number* [eE] [+-](optional) *decimal-sequence*

A postfix [fF] defines a float type and no postfix or postfix of [dD] defines a double type.

Identifier:

An identifier consist of alphabets, underscore and digits. An identifier doesn’t start with digit and does not end in underscore.

Upon reading a valid identifier scanner returns the symbol ***Identifier***.

An identifier that is the same as a keyword, is invalid.

The string of the identifier is accessible via scanner methods.

Operators and Keywords:

Upon reading an operator or keyword corresponding symbol is returned by scanner.

Comments:

There are two types of comments:

* Single line comments
* Comment blocks

A single line comment starts by ***%%*** and ends by end of line or end of file.

A comment block starts by ***%&*** and ends by ***&%***.

**Parser**

Grammar

The following grammar is represented in EBFN notation. The red terms are terminals.

MAIN = Dcls | Includes, [Dcls];

Includes = Include, {Include};

Dcls = Dcl, {Dcl};

Include = include, string\_literal, semicolon;

Dcl = FuncDclP | VarDclP | RecordDcl | Type, id, (VarDclWOTypeIdAF | FuncDclWOTypeIdAF);

VarDcl = VarInitDclP | (Type, id, (VarInitDclWOTypeIdAF | semicolon));

VarInitDcl = ((const, Type) | Type), id, assignment, Expr, semicolon;

RecordDcl = record, id, curlybracesopen, VarDcl, {VarDcl}, curlybracesclose;

FunDcl = FuncExternDcl | (Type, id, FuncInternDclWOTypeIdAF);

FuncInternDcl = Type, id, FuncInternDclWOTypeIdAF;

Block = curlybraceopen, {VarDclIdP | StatementP | (id, (VarDclWOIdAFP | StatementWOIdAFP | colon | (bracketopen, (VarDclWOIdAFWOBSAF | StatementWOIdAFWOBSAF))))}, curlybraceclose;

Statement = StatementP | (id, StatementWOIdAF);

ConditionalStatement = (if, parenthesisopen, Expr, parenthesisclose, (Block | Statement), else, (Block | Statement)) | (switch, parenthesisopen, Variable, parenthesisclose, curlybracesopen, (curlybracesclose | (Cases, (curlybracesclose | (default, colon, (Statement | Block), curlybracesclose))) | (default, (curlybracesclose | (default, colon, (Statement | Block), curlybracesclose))));

Case = case, IntegerLiteral, colon, (Statement | Block);

Cases = Case, {Case};

LoopStatement = (((while, parenthesisopen, Expr, parenthesisclose) | (for, parenthesisopen, ((id, (VarInitDclWOIdAFP | bracketopen, ((VariableWOIdAFWOBSAF, assignment, Expr) | VarInitDclWOIdAFWOBSAF))) | VarInitDclidP | semicolon))), (Statement, Block)) | (do, Block, (while | until), parenthesisopen, Expr, parenthesisclose, semicolon);

FuncExternDcl = extern, Type, id, semicolon;

Arguments = SoloType, id, {bracketopen, bracketclose}, comma, Arguments;

Type = (id | SoloType), {bracketopen, bracketclose};

SoloType = bool | byte | char | double | float | int | long | string | void;

Expr = ((parenthesisopen, Expr, parenthesisclose) | (UnaryOpWOAddSub, Expr) | LiteralValP | FuncCallP | AssignmentP), [BinaryOp, Expr]) | (id, ((AssignmentWOIdAFWOV, [BinaryOp, Expr]) | (VariableWOIdAF [AssignmentWOIdAFWOV], [BinaryOp, Expr]) | (FuncCallWOIdAF, [BinaryOp, Expr]) | [BinaryOp, Expr])));

UnaryOp = logicalnot | bitwisenot | sub | add;

BinaryOp = equal | notequal | greaterequal | lessequal | greater | less | logicaland | bitwiseand | logicalor | bitwiseor | bitwisexor | multiplication | division | add | sub | mode;

Variable = id, VariableWOIdAF;

Assignment = (id, AssignmnetWOIdAF) | AssignmentP;

FuncCall = (id, FuncCallWOIdAF) | FuncCallP;

Parameters = Expr, [comma, Parameters];

IntegerLiteral = integer\_literal | ((add | sub), integer\_literal);

VarInitDclWOTypeIdAF = assignment, Expr, semicolon;

VarInitDclP = const, Type, id, assignment, Expr, semicolon;

FuncinternDclWOTypeIdAF = parenthesisopen, [Arguments] parenthesisclose, (semicolon | Block);

VarDclWOTypeIdAF = semicolon | VarInitDclWOTypeIdAF;

VarDclP = VarInitDclP;

FuncDclWOTypeIdAF = FuncInternDclWOTypeIdAF;

FuncDclP = FuncExternDcl;

StatementP = ConditionalStatment | LoopStatment | (((goto, id) | AssignmentP | FuncCallP | break | continue), semicolon) | (return, [Expr], semicolon);

StatmentWOIdAF = (bracketopen, StatementWOIdAFWOBSAF) | StatementWOIdAFP;

VariableWOIdAF = (bracketopen | VariableWOIdAFWOBSAF) | VariableWOIdAFP;

FuncCallWOIdAF = parenthesisopen, [Paramenters], parenthesisclose;

FuncCallP = sizeof, parenthesisopen, ((TypeIdP | (id, [(bracketopen, (VariableWOIdAFWOBSAF | TypeWOIdAFWOBSAF)) | VariableWOIdAFP])), parenthesisclose;

AssignmentWOIdAF = (bracketopen, AssignmentWOIdAFWOBSAF) | AssignmentWOIdAFP;

AssignmentP = (increment | decrement), Variable;

VarDclWOIdAF = (bracketopen, VarDclWOIdAFWOBSAF) | VarDclWOIdAFP;

VarDclP = (TypeIdP, id, (semicolon | VarInitDclWOTypeIdAF)) | VarInitDclP;

TypeIdP = SoloType, {bracketopen, bracketclose};

VarInitDclWOIdAF = (bracketopen, VarInitDclWOIdAFWOBSAF) | VarInitDclWOIdAFP;

VarInitDclidP = ((const, Type) | TypeIdP), id, assignment, Expr, semicolon;

VariableWOIdAFWOBSAF = Expr, bracketclose, {bracketopen, Expr, bracketclose}, [dot, Variable];

TypeWOIdAFWOBSAF = bracketclose, {bracketclose, bracketopen};

VarInitDclWOIdAFWOBSAF = TypeWOIdAFWOBSAF, id, assignment, Expr, semicolon;

VariableWOIdAFP = dot, Variable, {dot, Variable};

LiteralValP = string\_literal | character\_literal | true | false | integer\_literal | float\_literal | double\_literal;

AssignmentWOIdAFWOV = (assignment, Expr) | increment | decrement;

VarDclWOIdAFWOBSAF = TypeWOIdAFWOBSAF, id, (semicolon | VarinitDclWOTypeIdAF);

VarDclWOIdAFP = id, (semicolon | VarinitDclWOTypeIdAF);

StatementWOIdAFWOBSAF = AssignmentWOIdAFWOBSAF, semicolon;

StatementWOIdAFP = (AssignmentWOIdAFP | FuncCallWOIdAF), semicolon;

AssignmentWOIdAFP = AssignmentWOIdAFWOV;

AssignmentWOIdAFWOBSAF = VariableWOIdAFWOBSAF, AssignmentWOIdAFWOV;

VarInitDclWOIdAFP = id, assignment, Expr, semicolon;

Handling signed values

As mentioned before, the sign of the signed values is handles on parser. This is done on Expr grammar.