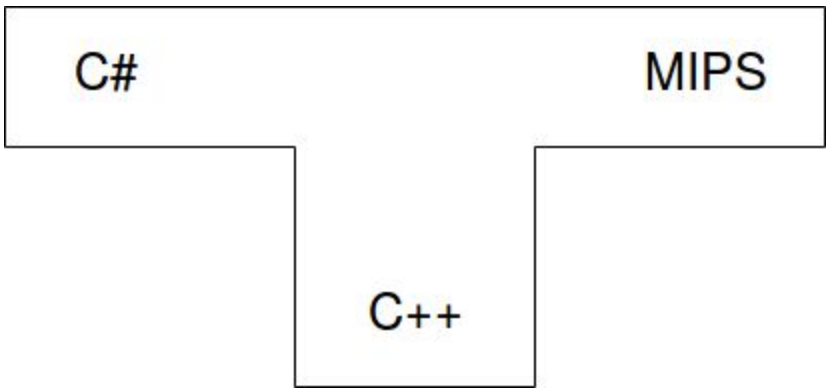
CS335: Assignment Zero

Amrit Singhal - 150092 - amrits@iitk.ac.in
Siddharth Agrawal - 150716 - sidarth@iitk.ac.in
Shobhit Rastogi - 150690 - shobhitr@iitk.ac.in

T-diagram of our compiler:



EBNF of our compiler language:

***(Note:** The following EBNF has been written in the standard ISO format for EBNF syntax, which follows the following conventions:*

1. *[ ] : Optional, may have zero or one occurrence.*
2. *, : Just a separator for the different elements in the description, anything to be actually included is written in single quotes(‘ ‘).*
3. *? ? : A placeholder for the description given within the marks.)*

start = compilation_unit;

(*B.1 Lexical grammar*)
input = [input_section];
input_section = input_section_part | input_section, input_section_part;
input_section_part = [input_elements], new_line | pp_directive;

input_elements = input_element | input_elements, input_element;
input_element = whitespace | comment | ?any token?;

(*B.1.1 Line terminators*)
new_line = ?Carriage return character (U+000D)? | ?Line feed character (U+000A)? | ?Carriage return character (U+000D) followed by line feed character (U+000A)? | ?Next line character (U+0085)? | ?Line separator character (U+2028)? | ?Paragraph separator character (U+2029)?;

(*B.1.2 Comments*)
comment = single_line_comment;
single_line_comment = '//', [input_characters];
input_characters = input_character | input_characters, input_character;
input_character = ?Any Unicode character except a new_line_character?;
new_line_character = ?Carriage return character (U+000D)? | ?Line feed character (U+000A)? | ?Next line character (U+0085)? | ?Line separator character (U+2028)? | ?Paragraph separator character (U+2029)?;

(*B.1.3 White space*)
whitespace = ?Any character with Unicode class Zs? | ?Horizontal tab character (U+0009)? | ?Vertical tab character (U+000B)? | ?Form feed character (U+000C)?;

(*B.1.4 Tokens*)
token = identifier | keyword | integer_literal | character_literal | string_literal | operator_or_punctuator;

(*B.1.5 Unicode character escape sequences*)
unicode_escape_sequence = '\u', hex_digit, hex_digit, hex_digit, hex_digit | '\U', hex_digit, hex_digit, hex_digit, hex_digit, hex_digit, hex_digit, hex_digit, hex_digit;

(*B.1.6 Identifiers*)
identifier = available_identifier | '@', identifier_or_keyword;
available_identifier = ?An identifier_or_keyword that is not a keyword?;
identifier_or_keyword = identifier_start_character, [identifier_part_characters];
identifier_start_character = letter_character | '_';
identifier_part_characters = identifier_part_character | identifier_part_characters, identifier_part_character;
identifier_part_character = letter_character | decimal_digit_character | connecting_character | combining_character | formatting_character;
letter_character = ?A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl? | ?A unicode_escape_sequence representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl?;
combining_character = ?A Unicode character of classes Mn or Mc? | ?A unicode_escape_sequence representing a character of classes Mn or Mc?;
decimal_digit_character = ?A Unicode character of the class Nd? | ?A unicode_escape_sequence representing a character of the class Nd?;
connecting_character = ?A Unicode character of the class Pc? | ?A unicode_escape_sequence representing a character of the class Pc?;

formatting_character = ?A Unicode character of the class Cf? | ?A unicode_escape_sequence representing a character of the class Cf?;

(*B.1.7 Keywords*)

keyword = 'base' | 'bool' | 'break' | 'case' | 'catch' | 'char' | 'class' | 'const' | 'continue' | 'default' | 'do' | 'else' | 'false' | 'finally' | 'for' | 'foreach' | 'goto' | 'if' | 'in' | 'int' | 'long' | 'namespace' | 'new' | 'null' | 'object'| 'params' | 'private' | 'protected' | 'public' | 'ref' | 'return' | 'string' | 'struct' | 'switch' | 'this' | 'throw' | 'true' | 'try' | 'typeof' | 'uint' | 'ulong' | 'using' | 'void' | 'while';

(*B.1.8 Literals*)

literal = boolean_literal | integer_literal | character_literal | string_literal | null_literal;
boolean_literal = 'true' | 'false';
integer_literal = decimal_integer_literal | hexadecimal_integer_literal;
decimal_integer_literal = decimal_digits, [integer_type_suffix];
decimal_digits = decimal_digit | decimal_digits, decimal_digit;
decimal_digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
integer_type_suffix = 'U' | 'u' | 'L' | 'l' | 'UL' | 'Ul' | 'uL' | 'ul' | 'LU' | 'Lu' | 'lU' | 'lu';
hexadecimal_integer_literal = '0x', hex_digits, [integer_type_suffix] | '0X', hex_digits, [integer_type_suffix];
hex_digits = hex_digit | hex_digits, hex_digit;
hex_digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f';
character_literal = "'", character, "'";
character = single_character | simple_escape_sequence | hexadecimal_escape_sequence | unicode_escape_sequence;
single_character = ?Any character except ' (U+0027), \ (U+005C), and new_line_character?;
simple_escape_sequence = " \’ “ | ‘ \” ‘ | ' \\ ' | ' \0 ' | ' \n ' | ' \r ' | ' \t ';
hexadecimal_escape_sequence = '\x', hex_digit, [hex_digit], [hex_digit], [hex_digit];
string_literal = regular_string_literal | verbatim_string_literal;
regular_string_literal = '"', [regular_string_literal_characters], '"';
regular_string_literal_characters = regular_string_literal_character | regular_string_literal_characters, regular_string_literal_character;
regular_string_literal_character = single_regular_string_literal_character | simple_escape_sequence | hexadecimal_escape_sequence | unicode_escape_sequence;
single_regular_string_literal_character = ?Any character except " (U+0022), \ (U+005C), and new_line_character?;
verbatim_string_literal = '@” ', [verbatim_string_literal_characters], ‘ “ ‘;
verbatim_string_literal_characters = verbatim_string_literal_character | verbatim_string_literal_characters, verbatim_string_literal_character;
verbatim_string_literal_character = single_verbatim_string_literal_character | quote_escape_sequence;
single_verbatim_string_literal_character = ?any character except "?;
quote_escape_sequence = ‘ “ “ ‘;
null_literal = 'null';

(*B.1.9 Operators and punctuators*)

operator_or_punctuator = '{' | '}' | '[' | ']' | '(' | ')' | '.' | ',' | ':' | ';' | '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '!' | '~' | '=' | '<' | '>' | '?' | '??' | '::' | '++' | '--' | '&&' | '||' | '->' | '==' | '!=' | '<=' | '>=' | '+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' | '<<' | '<<=' | '=>';
right_shift = '>>';
right_shift_assignment = '>>=';

(*B.1.10 Pre_processing directives*)
pp_directive = pp_declaration | pp_line ;
conditional_symbol = ?Any identifier_or_keyword except true or false?;
pp_declaration = [whitespace], '#', [whitespace], 'define', whitespace, conditional_symbol, pp_new_line | [whitespace], '#', [whitespace], 'undef', whitespace, conditional_symbol, pp_new_line;
pp_new_line = [whitespace], [single_line_comment], new_line;

(*B.2.1 Basic concepts*)
namespace_name = namespace_or_type_name;
type_name = namespace_or_type_name;
namespace_or_type_name = identifier, [type_argument_list] | namespace_or_type_name, '.', identifier, [type_argument_list] | qualified_alias_member;

(*B.2.2 Types*)
type = value_type | reference_type | type_parameter;
value_type = struct_type;
struct_type = type_name | simple_type | nullable_type;
simple_type = numeric_type | 'bool';
numeric_type = integral_type | 'decimal';
integral_type = 'int' | 'uint' | 'long' | 'ulong' | 'char';
nullable_type = non_nullable_value_type, '?';
non_nullable_value_type = type;
reference_type = class_type | interface_type | array_type;
class_type = type_name | 'object' | 'dynamic' | 'string';
interface_type = type_name;
type_argument_list = '<', type_arguments, '>';
type_arguments = type_argument | type_arguments, ',', type_argument;
type_argument = type;

(*B.2.3 Variables*)
variable_reference = expression;

(*B.2.4 Expressions*)
argument_list = argument | argument_list, ',', argument;
argument = [argument_name], argument_value;
argument_name = identifier, ':';
argument_value = expression | 'ref', variable_reference ;

primary_expression = primary_no_array_creation_expression | array_creation_expression;
primary_no_array_creation_expression = literal | simple_name | parenthesized_expression | member_access | invocation_expression | element_access | this_access | base_access | post_increment_expression | post_decrement_expression | object_creation_expression | anonymous_object_creation_expression | typeof_expression | default_value_expression;
simple_name = identifier, [type_argument_list];
parenthesized_expression = '(', expression, ')';
member_access = primary_expression, '.', identifier, [type_argument_list] | predefined_type, '.', identifier, [type_argument_list] | qualified_alias_member, '.', identifier;
predefined_type = 'bool' | 'char' | 'decimal' | 'double' | 'float' | 'int' | 'long' | 'object' | 'string' | 'uint' | 'ulong' | 'ushort';
invocation_expression = primary_expression, '(', [argument_list], ')';
element_access = primary_no_array_creation_expression, '[', expression_list, ']';
expression_list = expression | expression_list, ',', expression;
this_access = 'this';
base_access = 'base', '.', identifier | 'base', '[', expression_list, ']';
post_increment_expression = primary_expression, '++';
post_decrement_expression = primary_expression, '--';
object_creation_expression = 'new', type, '(', [argument_list], ')', [object_or_collection_initializer] | 'new', type, object_or_collection_initializer;
object_or_collection_initializer = object_initializer | collection_initializer;
object_initializer = '{', [member_initializer_list], '}' | '{', member_initializer_list, ',', '}';
member_initializer_list = member_initializer | member_initializer_list, ',', member_initializer;
member_initializer = identifier, '=', initializer_value;
initializer_value = expression | object_or_collection_initializer;
collection_initializer = '{', element_initializer_list, '}' | '{', element_initializer_list, ',', '}';
element_initializer_list = element_initializer | element_initializer_list, ',', element_initializer;
element_initializer = non_assignment_expression | '{', expression_list, '}';
array_creation_expression = 'new', non_array_type, '[', expression_list, ']', [rank_specifiers], [array_initializer] | 'new', array_type, array_initializer | 'new', rank_specifier, array_initializer;
anonymous_object_creation_expression = 'new', anonymous_object_initializer;
anonymous_object_initializer = '{', [member_declarator_list], '}' | '{', member_declarator_list, ',', '}';
member_declarator_list = member_declarator | member_declarator_list, ',', member_declarator;
member_declarator = simple_name | member_access | base_access | identifier, '=', expression;
typeof_expression = 'typeof', '(', type, ')' | 'typeof', '(', unbound_type_name, ')' | 'typeof', '(', 'void', ')';
unbound_type_name = identifier, [generic_dimension_specifier] | identifier, '::', identifier, [generic_dimension_specifier] | unbound_type_name, '.', identifier, [generic_dimension_specifier];
generic_dimension_specifier = '<', [commas], '>';
commas = ',' | commas, ',';
default_value_expression = 'default', '(', type, ')';
unary_expression = primary_expression | '+', unary_expression | '-', unary_expression | '!', unary_expression | '~', unary_expression | pre_increment_expression | pre_decrement_expression | cast_expression;
pre_increment_expression = '++', unary_expression;

pre_decrement_expression = '--', unary_expression;
cast_expression = '(', type, ')', unary_expression;
multiplicative_expression = unary_expression | multiplicative_expression, '*', unary_expression | multiplicative_expression, '/', unary_expression | multiplicative_expression, '%', unary_expression;
additive_expression = multiplicative_expression | additive_expression, '+', multiplicative_expression | additive_expression, '-', multiplicative_expression;
shift_expression = additive_expression | shift_expression, '<<', additive_expression | shift_expression, right_shift, additive_expression;
relational_expression = shift_expression | relational_expression, '<', shift_expression | relational_expression, '>', shift_expression | relational_expression, '<=', shift_expression | relational_expression, '>=', shift_expression;
equality_expression = relational_expression | equality_expression, '==', relational_expression | equality_expression, '!=', relational_expression;
and_expression = equality_expression | and_expression, '&', equality_expression;
exclusive_or_expression = and_expression | exclusive_or_expression, '^', and_expression;
inclusive_or_expression = exclusive_or_expression | inclusive_or_expression, '|', exclusive_or_expression;
conditional_and_expression = inclusive_or_expression | conditional_and_expression, '&&', inclusive_or_expression;
conditional_or_expression = conditional_and_expression | conditional_or_expression, '||', conditional_and_expression;
null_coalescing_expression = conditional_or_expression | conditional_or_expression, '??', null_coalescing_expression;
conditional_expression = null_coalescing_expression | null_coalescing_expression, '?', expression, ':', expression;
assignment = unary_expression, assignment_operator, expression;
assignment_operator = '=' | '+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' | '<<=' | right_shift_assignment;
expression = non_assignment_expression | assignment;
non_assignment_expression = conditional_expression;
constant_expression = expression;
boolean_expression = expression;

(*B.2.5 Statements*)
statement = labeled_statement | declaration_statement | embedded_statement;
embedded_statement = block | empty_statement | expression_statement | selection_statement | iteration_statement | jump_statement | try_statement | using_statement;
block = '{', [statement_list], '}';
statement_list = statement | statement_list, statement;
empty_statement = ';';
labeled_statement = identifier, ':', statement;
declaration_statement = local_variable_declaration, ';' | local_constant_declaration, ';';
local_variable_declaration = local_variable_type, local_variable_declarators;
local_variable_type = type | 'var';

local_variable_declarators = local_variable_declarator | local_variable_declarators, ',',
local_variable_declarator;
local_variable_declarator = identifier | identifier, '=', local_variable_initializer;
local_variable_initializer = expression | array_initializer;
local_constant_declaration = 'const', type, constant_declarators;
expression_statement = statement_expression, ';';
statement_expression = invocation_expression | object_creation_expression | assignment |
post_increment_expression | post_decrement_expression | pre_increment_expression |
pre_decrement_expression;
selection_statement = if_statement | switch_statement;
if_statement = 'if', '(', boolean_expression, ')', embedded_statement | 'if', '(', boolean_expression, ')',
embedded_statement, 'else', embedded_statement;
switch_statement = 'switch', '(', expression, ')', switch_block;
switch_block = '{', [switch_sections], '}';
switch_sections = switch_section | switch_sections, switch_section;
switch_section = switch_labels, statement_list;
switch_labels = switch_label | switch_labels, switch_label;
switch_label ='case', constant_expression, ':' | 'default', ':';
iteration_statement = while_statement | do_statement | for_statement | foreach_statement;
while_statement = 'while', '(', boolean_expression, ')', embedded_statement;
do_statement = 'do', embedded_statement, 'while', '(', boolean_expression, ')', ';';
for_statement = 'for', '(', [for_initializer], ';', [for_condition], ';', [for_iterator], ')', embedded_statement;
for_initializer = local_variable_declaration | statement_expression_list;
for_condition = boolean_expression;
for_iterator = statement_expression_list;
statement_expression_list = statement_expression | statement_expression_list, ',', statement_expression;
foreach_statement = 'foreach', '(', local_variable_type, identifier, 'in', expression, ')', embedded_statement;
jump_statement = break_statement | continue_statement | goto_statement | return_statement |
throw_statement;
break_statement = 'break', ';';
continue_statement = 'continue', ';';
goto_statement = 'goto', identifier, ';' | 'goto', 'case', constant_expression, ';' | 'goto', 'default', ';';
return_statement = 'return', [expression], ';';
throw_statement = 'throw', [expression], ';';
try_statement = 'try', block, catch_clauses | 'try', block, finally_clause | 'try', block, catch_clauses,
finally_clause;
catch_clauses = general_catch_clause;
general_catch_clause = 'catch', block;
finally_clause = 'finally', block;
using_statement = 'using', '(', resource_acquisition, ')', embedded_statement;
resource_acquisition = local_variable_declaration | expression;

(*B.2.6 Namespaces;*)

compilation_unit = [using_directives], [namespace_member_declarations];
namespace_declaration = 'namespace', qualified_identifier, namespace_body, [';'];
qualified_identifier = identifier | qualified_identifier, '.', identifier;
namespace_body = '{', [using_directives], [namespace_member_declarations], '}';
using_directives = using_directive | using_directives, using_directive;
using_directive = using_alias_directive | using_namespace_directive;
using_alias_directive = 'using', identifier, '=', namespace_or_type_name, ';';
using_namespace_directive = 'using', namespace_name, ';';
namespace_member_declarations = namespace_member_declaration | namespace_member_declarations, namespace_member_declaration;
namespace_member_declaration = namespace_declaration | type_declaration;
type_declaration = class_declaration | struct_declaration;
qualified_alias_member = identifier, '::', identifier, [type_argument_list];

(*B.2.7 Classes;*)

class_declaration = [class_modifiers], 'class', identifier, [type_parameter_list], [class_body], [type_parameter_constraints_clauses], class_body, [';'];
class_modifiers = class_modifier | class_modifiers, class_modifier;
class_modifier = 'new' | 'public' | 'protected' | 'private' ;
type_parameter_list = '<', type_parameters, '>';
type_parameters = type_parameter | type_parameters, ',', type_parameter;
type_parameter = identifier;
class_base = ':', class_type;
type_parameter_constraints_clauses = type_parameter_constraints_clause | type_parameter_constraints_clauses, type_parameter_constraints_clause;
type_parameter_constraints_clause = 'where', type_parameter, ':', type_parameter_constraints;
type_parameter_constraints = primary_constraint | secondary_constraints | constructor_constraint | primary_constraint, ',', secondary_constraints | primary_constraint, ',', constructor_constraint | secondary_constraints, ',', constructor_constraint | primary_constraint, ',', secondary_constraints, ',', constructor_constraint;
primary_constraint = class_type | 'class' | 'struct';
secondary_constraints = type_parameter | secondary_constraints, ',', type_parameter;
constructor_constraint = 'new', '(', ')';
class_body = '{', [class_member_declarations], '}';
class_member_declarations = class_member_declaration | class_member_declarations, class_member_declaration;
class_member_declaration = constant_declaration | field_declaration | method_declaration | indexer_declaration | constructor_declaration | destructor_declaration | |type_declaration;
constant_declaration = [constant_modifiers], 'const', type, constant_declarators, ';';
constant_modifiers = constant_modifier | constant_modifiers, constant_modifier;
constant_modifier = 'new' | 'public' | 'protected' | 'private';
constant_declarators = constant_declarator | constant_declarators, ',', constant_declarator;

constant_declarator = identifier, '=', constant_expression;
field_declaration = [field_modifiers], type, variable_declarators, ';';
field_modifiers = field_modifier | field_modifiers, field_modifier;
field_modifier = 'new' | 'public' | 'protected' | 'private';
variable_declarators = variable_declarator | variable_declarators, ',', variable_declarator;
variable_declarator = identifier | identifier, '=', variable_initializer;
variable_initializer = expression | array_initializer;
method_declaration = method_header, method_body;
method_header = [method_modifiers], return_type, member_name, [type_parameter_list], '(',
[formal_parameter_list], ')', [type_parameter_constraints_clauses];
method_modifiers = method_modifier | method_modifiers, method_modifier;
method_modifier = 'new' | 'public' | 'protected' | 'private' ;
return_type = type | 'void';
member_name = identifier | interface_type, '.', identifier;
method_body = block | ';';
formal_parameter_list = fixed_parameters | fixed_parameters, ',', parameter_array | parameter_array;
fixed_parameters = fixed_parameter | fixed_parameters, ',', fixed_parameter;
fixed_parameter = [parameter_modifier], type, identifier, [default_argument];
default_argument = '=', expression;
parameter_modifier = 'ref' | 'this';
parameter_array = 'params', array_type, identifier;
constructor_declaration = [constructor_modifiers], constructor_declarator, constructor_body;
constructor_modifiers = constructor_modifier | constructor_modifiers, constructor_modifier;
constructor_modifier = 'public' | 'protected' | 'private';
constructor_declarator = identifier, '(', [formal_parameter_list], ')', [constructor_initializer];
constructor_initializer = ':', 'base', '(', [argument_list], ')' | ':', 'this', '(', [argument_list], ')';
constructor_body = block | ';';
destructor_declaration = '~', identifier, '(', ')', destructor_body;
destructor_body = block | ';';

(*B.2.8 Structs*)
struct_declaration = [struct_modifiers], 'struct', identifier, [type_parameter_list],
[type_parameter_constraints_clauses], struct_body, [';'];
struct_modifiers = struct_modifier | struct_modifiers, struct_modifier;
struct_modifier = 'new' | 'public' | 'protected' | 'private';
struct_body = '{', [struct_member_declarations], '}';
struct_member_declarations = struct_member_declaration | struct_member_declarations,
struct_member_declaration;
struct_member_declaration = constant_declaration | field_declaration | method_declaration | event_declaration
| constructor_declaration | type_declaration;

(*B.2.9 Arrays*)
array_type = non_array_type, rank_specifier;

non_array_type = type;
rank_specifier = '[', [dim_separator], ']';
dim_separator = ',';
array_initializer = '{', [variable_initializer_list], '}' | '{', variable_initializer_list, ',', '}';
variable_initializer_list = variable_initializer | variable_initializer_list, ',', variable_initializer;

Syntactic Rules deleted from our base language:

Preprocessing directives :

pp_expression = [whitespace], pp_or_expression, [whitespace];
pp_or_expression = pp_and_expression | pp_or_expression, [whitespace], '||', [whitespace], pp_and_expression;
pp_and_expression = pp_equality_expression | pp_and_expression, [whitespace], '&&', [whitespace], pp_equality_expression;
pp_equality_expression = pp_unary_expression | pp_equality_expression, [whitespace], '==', [whitespace], pp_unary_expression | pp_equality_expression, [whitespace], '!=', [whitespace], pp_unary_expression;
pp_unary_expression = pp_primary_expression | '!', [whitespace], pp_unary_expression;
pp_primary_expression = 'true' | 'false' | conditional_symbol | '(', [whitespace], pp_expression, [whitespace], ')';
pp_conditional = pp_if_section, [pp_elif_sections], [pp_else_section], pp_endif;
pp_if_section = [whitespace], '#', [whitespace], 'if', whitespace, pp_expression, pp_new_line, [conditional_section];
pp_elif_sections = pp_elif_section | pp_elif_sections, pp_elif_section;
pp_elif_section = [whitespace], '#', [whitespace], 'elif', whitespace, pp_expression, pp_new_line, [conditional_section];
pp_else_section = [whitespace], '#', [whitespace], 'else', pp_new_line, [conditional_section];
pp_endif = [whitespace], '#', [whitespace], 'endif', pp_new_line;
conditional_section = input_section | skipped_section;
skipped_section = skipped_section_part | skipped_section, skipped_section_part;
skipped_section_part = [skipped_characters], new_line | pp_directive;
skipped_characters = [whitespace], not_number_sign, [input_characters];
not_number_sign = ?Any input_character except #?;
pp_diagnostic = [whitespace], '#', [whitespace], 'error', pp_message | [whitespace], '#', [whitespace], 'warning', pp_message;
pp_message = new_line | whitespace, [input_characters], new_line;
pp_region = pp_start_region, [conditional_section], pp_end_region;
pp_start_region = [whitespace], '#', [whitespace], 'region', pp_message;
pp_end_region = [whitespace], '#', [whitespace], 'endregion', pp_message;
pp_line = [whitespace], '#', [whitespace], 'line', whitespace, line_indicator, pp_new_line;
line_indicator = decimal_digits, whitespace, file_name | decimal_digits | 'default' | 'hidden';
file_name = '"', file_name_characters, '"';
file_name_characters = file_name_character | file_name_characters, file_name_character;
file_name_character = ?Any input_character except "?;

pp_pragma = [whitespace], '#', [whitespace], 'pragma', whitespace, pragma_body, pp_new_line;
pragma_body = pragma_warning_body;
pragma_warning_body = 'warning', whitespace, warning_action | 'warning', whitespace, warning_action, whitespace, warning_list;
warning_action = 'disable' | 'restore';
warning_list = decimal_digits | warning_list, [whitespace], ',', [whitespace], decimal_digits;

Floating point:
floating_point_type = 'float' | 'double';

Lambda expressions:
lambda_expression = anonymous_function_signature, '=>', anonymous_function_body;
anonymous_function_signature = explicit_anonymous_function_signature | implicit_anonymous_function_signature;
explicit_anonymous_function_signature = '(', [explicit_anonymous_function_parameter_list], ')';
explicit_anonymous_function_parameter_list = explicit_anonymous_function_parameter | explicit_anonymous_function_parameter_list, ',', explicit_anonymous_function_parameter;
explicit_anonymous_function_parameter = [anonymous_function_parameter_modifier], type, identifier;
anonymous_function_parameter_modifier = 'ref' | 'out';
implicit_anonymous_function_signature = '(', [implicit_anonymous_function_parameter_list], ')' | implicit_anonymous_function_parameter;
implicit_anonymous_function_parameter_list = implicit_anonymous_function_parameter | implicit_anonymous_function_parameter_list, ',', implicit_anonymous_function_parameter;
implicit_anonymous_function_parameter = identifier;
anonymous_function_body = expression | block;

Query expressions:
query_expression = from_clause, query_body;
from_clause = 'from', [type], identifier, 'in', expression;
query_body = [query_body_clauses], select_or_group_clause, [query_continuation];
query_body_clauses = query_body_clause | query_body_clauses, query_body_clause;
query_body_clause = from_clause | let_clause | where_clause | join_clause | join_into_clause | orderby_clause;
let_clause = 'let', identifier, '=', expression;
where_clause = 'where', boolean_expression;
join_clause = 'join', [type], identifier, 'in', expression, 'on', expression, 'equals', expression;
join_into_clause = 'join', [type], identifier, 'in', expression, 'on', expression, 'equals', expression, 'into', identifier;
orderby_clause = 'orderby', orderings;
orderings = ordering | orderings, ',', ordering;
ordering = expression, [ordering_direction];
ordering_direction = 'ascending' | 'descending';
select_or_group_clause = select_clause | group_clause;
select_clause = 'select', expression;

group_clause = 'group', expression, 'by', expression;
query_continuation = 'into', identifier, query_body;

Specific catch clauses:
specific_catch_clauses = specific_catch_clause | specific_catch_clauses, specific_catch_clause;
specific_catch_clause = 'catch', '(', class_type, [identifier], ')', block;

Properties:
property_declaration = [property_modifiers], type, member_name, '{', accessor_declarations, '}';
property_modifiers = property_modifier | property_modifiers, property_modifier;
property_modifier = 'new' | 'public' | 'protected' | 'private' | 'static' | 'virtual' | 'sealed' | 'override' | 'abstract';

Accessor
accessor_declarations = get_accessor_declaration, [set_accessor_declaration] | set_accessor_declaration, [get_accessor_declaration];
get_accessor_declaration = [accessor_modifier], 'get', accessor_body;
set_accessor_declaration = [accessor_modifier], 'set', accessor_body;
accessor_modifier = 'protected' | 'private';
accessor_body = block | ';';

Indexer:
indexer_declaration = [indexer_modifiers], indexer_declarator, '{', accessor_declarations, '}';
indexer_modifiers = indexer_modifier | indexer_modifiers, indexer_modifier;
indexer_modifier = 'new' | 'public' | 'protected' | 'private' | 'virtual' | 'sealed' | 'override' | 'abstract';
indexer_declarator = type, 'this', '[', formal_parameter_list, ']' | type, interface_type, '.', 'this', '[', formal_parameter_list, ']';

Some modifiers:

- *Static, Abstract, Virtual, Override, Sealed, Readonly have been removed.*

static_constructor_declaration = static_constructor_modifiers, identifier, '(', ')', static_constructor_body;
static_constructor_modifiers = 'static';
static_constructor_body = block | ';';

Multithreading removed:
lock_statement = 'lock', '(', expression, ')', embedded_statement;
yield_statement = 'yield', 'return', expression, ';' | 'yield', 'break', ';';

- Multidimensional arrays of **dimensionality > 2** have been removed.

Attributes:
global_attributes = global_attribute_sections;
global_attribute_sections = global_attribute_section | global_attribute_sections, global_attribute_section;

global_attribute_section = '[', global_attribute_target_specifier, attribute_list, ']' | '[',
global_attribute_target_specifier, attribute_list, ',', ']';
global_attribute_target_specifier = global_attribute_target, ':';
global_attribute_target = 'assembly' | 'module';
attributes = attribute_sections;
attribute_sections = attribute_section | attribute_sections, attribute_section;
attribute_section = '[', [attribute_target_specifier], attribute_list, ']' | '[', [attribute_target_specifier], attribute_list,
',', ']';
attribute_target_specifier = attribute_target, ':';
attribute_target = 'field' | 'event' | 'method' | 'param' | 'property' | 'return' | 'type';
attribute_list = attribute | attribute_list, ',', attribute;
attribute = attribute_name, [attribute_arguments];
attribute_name = type_name;
attribute_arguments = '(', [positional_argument_list], ')' | '(', positional_argument_list, ',', named_argument_list,
')' | '(', named_argument_list, ')';
positional_argument_list = positional_argument | positional_argument_list, ',', positional_argument;
positional_argument = attribute_argument_expression;
named_argument_list = named_argument | named_argument_list, ',', named_argument;
named_argument = identifier, '=', attribute_argument_expression;
attribute_argument_expression = expression;

Enums:
enum_declaration = [attributes], [enum_modifiers], 'enum', identifier, [enum_base], enum_body, [';'];
enum_base = ':', integral_type;
enum_body = '{', [enum_member_declarations], '}' | '{', enum_member_declarations, ',', '}';
enum_modifiers = enum_modifier | enum_modifiers, enum_modifier;
enum_modifier = 'new' | 'public' | 'protected' | 'internal' | 'private';
enum_member_declarations = enum_member_declaration | enum_member_declarations, ',',
enum_member_declaration;
enum_member_declaration = [attributes], identifier | [attributes], identifier, '=', constant_expression;

Interfaces:
interface_declaration = [attributes], [interface_modifiers], ['partial'], 'interface', identifier,
[variant_type_parameter_list], [interface_base], [type_parameter_constraints_clauses], interface_body, [';'];
interface_modifiers = interface_modifier | interface_modifiers, interface_modifier;
interface_modifier = 'new' | 'public' | 'protected' | 'internal' | 'private' | interface_modifier_unsafe;
variant_type_parameter_list = '<', variant_type_parameters, '>';
variant_type_parameters = [attributes], [variance_annotation], type_parameter | variant_type_parameters, ',',
[attributes], [variance_annotation], type_parameter;
variance_annotation = 'in' | 'out';
interface_base = ':', interface_type_list;
interface_body = '{', [interface_member_declarations], '}';

interface_member_declarations = interface_member_declaration | interface_member_declarations, interface_member_declaration;
interface_member_declaration = interface_method_declaration | interface_property_declaration | interface_event_declaration | interface_indexer_declaration;
interface_method_declaration = [attributes], ['new'], return_type, identifier, type_parameter_list, '(', [formal_parameter_list], ')', [type_parameter_constraints_clauses], ';';
interface_property_declaration = [attributes], ['new'], type, identifier, '{', interface_accessors, '}';
interface_accessors = [attributes], 'get', ';' | [attributes], 'set', ';' | [attributes], 'get', ';', [attributes], 'set', ';' | [attributes], 'set', ';', [attributes], 'get', ';';
interface_event_declaration = [attributes], ['new'], 'event', type, identifier, ';';
interface_indexer_declaration = [attributes], ['new'], type, 'this', '[', formal_parameter_list, ']', '{', interface_accessors, '}';

Delegates:

delegate_declaration = [attributes], [delegate_modifiers], 'delegate', return_type, identifier, [type_parameter_list], '(', [formal_parameter_list], ')', [type_parameter_constraints_clauses], ';';
delegate_modifiers = delegate_modifier | delegate_modifiers, delegate_modifier;
delegate_modifier = 'new' | 'public' | 'protected' | 'internal' | 'private' | delegate_modifier_unsafe;

Unsafe sections (i.e. data pointers):

class_modifier_unsafe = 'unsafe';
struct_modifier_unsafe = 'unsafe';
interface_modifier_unsafe = 'unsafe';
delegate_modifier_unsafe = 'unsafe';
field_modifier_unsafe = 'unsafe';
method_modifier_unsafe = 'unsafe';
property_modifier_unsafe = 'unsafe';
event_modifier_unsafe = 'unsafe';
indexer_modifier_unsafe = 'unsafe';
operator_modifier_unsafe = 'unsafe';
constructor_modifier_unsafe = 'unsafe';
destructor_declaration_unsafe = [attributes], ['extern'], ['unsafe'], '~', identifier, '(', ')', destructor_body | [attributes], ['unsafe'], ['extern'], '~', identifier, '(', ')', destructor_body;
static_constructor_modifiers_unsafe = ['extern'], ['unsafe'], 'static' | ['unsafe'], ['extern'], 'static' | ['extern'], 'static', ['unsafe'] | ['unsafe'], 'static', ['extern'] | 'static', ['extern'], ['unsafe'] | 'static', ['unsafe'], ['extern'];
embedded_statement_unsafe = unsafe_statement | fixed_statement;
unsafe_statement = 'unsafe', block;
type_unsafe = pointer_type;
pointer_type = unmanaged_type, '*' | 'void', '*';
unmanaged_type = type;
primary_no_array_creation_expression_unsafe = pointer_member_access | pointer_element_access | sizeof_expression;

unary_expression_unsafe = pointer_indirection_expression | addressof_expression;
pointer_indirection_expression = '*', unary_expression;
pointer_member_access = primary_expression, '->', identifier;
pointer_element_access = primary_no_array_creation_expression, '[', expression, ']';
addressof_expression = '&', unary_expression;
sizeof_expression = 'sizeof', '(', unmanaged_type, ')';
fixed_statement = 'fixed', '(', pointer_type, fixed_pointer_declarators, ')', embedded_statement;
fixed_pointer_declarators = fixed_pointer_declarator | fixed_pointer_declarators, ',', fixed_pointer_declarator;
fixed_pointer_declarator = identifier, '=', fixed_pointer_initializer;
fixed_pointer_initializer = '&', variable_reference | expression;
struct_member_declaration_unsafe = fixed_size_buffer_declaration;
fixed_size_buffer_declaration = [attributes], [fixed_size_buffer_modifiers], 'fixed', buffer_element_type,
fixed_size_buffer_declarators, ';';
fixed_size_buffer_modifiers = fixed_size_buffer_modifier | fixed_size_buffer_modifier,
fixed_size_buffer_modifiers;
fixed_size_buffer_modifier = 'new' | 'public' | 'protected' | 'internal' | 'private' | 'unsafe';
buffer_element_type = type;
fixed_size_buffer_declarators = fixed_size_buffer_declarator | fixed_size_buffer_declarator,
fixed_size_buffer_declarators;
fixed_size_buffer_declarator = identifier, '[', constant_expression, ']';
local_variable_initializer_unsafe = stackalloc_initializer;
stackalloc_initializer = 'stackalloc', unmanaged_type, '[', expression, ']';

Tools that will be used:

1. **Flex:** as a lexer generator
2. **Bison:** as a parser generator