Language Design Proposal

Student Name(s): Dominic Ciliberto, John Aquino, Kevin Zelaya, Matthew Fuller, Pablo Lepe
Language Name: B ♭ (B Flat)

Compiler Implementation Language and Reasoning: Our group will implement the compiler in Dart. We want to use this language for three strong reasons.

1. The language is syntactically and semantically similar to Java and C#. We all have experience with both of these languages.
2. Most of us have experience with Dart since we, with the exception of John, are utilizing the language to implement our senior design project.
3. We can utilize very robust tools on Dart's SDK, API, and package website. Some of these tools are useful for lexical and syntactic analysis, parsing, and CLI applications.

Target Language: Dart

Language Description: B ♭ (B Flat), is a subset of C# that does not support the reserved keyword 'var'. Most functionality involving error handling is also largely ignored under the assumption that error checking and exception handling would be too difficult to implement in such a short time. There are many other limitations that will be considered for our language. These limitations are necessary. This is due in part that a small five man team cannot possibly consider, cover, design, implement for, and test every possible feature that exists in C#, let alone the effort of translating these features from C# to Dart.

Fundamentally, the main object of the language, B ♭ (B Flat), is to be a statically typed subset of C#. This flagship design choice of our group's language fulfills the proposal requirement where the language needs to be static. B Flat should also compile into Microsoft's dotnet compiler since B Flat is itself is a subset of C#. Utilizing a compiler (transpiler) that we will implement using Dart, B Flat should then be translated into Dart code. To be more specific the output language of Dart code is of our group's own arbitrary namesake and definitions, Start (Statically Typed Dart). Start is a subset of Dart and is equivalent to B Flat. Since Start is a subset of Dart all syntactically correct instances of Start should be able to run in any Dart (>= v.2.7.1) interpreter, virtual machine, or compiler that Google has developed. There is nothing special about Start, it simply exists as an abstraction of the subset of Dart that will be the compiler's output. To reiterate and put succinctly, Start is a simplified Dart. These namesakes are in place as an attempt to provide clarity between all of the abstract concepts within the project.

Our compiler is aptly named 'cs2dart', which coincides with the naming scheme of Google's new AOT compiler 'dart2native'. The name of the compiler is kept as 'cs2dart' because this idea as a

whole intrigues me (Matthew) and could be expanded to support much more aspects of each language at a later date (This was the ultimate goal of the compiler, at least as a cool idea). Additionally, all B Flat files will have a file extension of '.cs', and all Start files will have a file extension of '.dart'. Finally, since B Flat is a subset of C# a large part of B Flat is derived from the current specification of C# (C# Specification 6.0): https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction

Planned Restrictions: Here is a list of all of the restrictions of B Flat (a.k.a. features of C# that will not carry over to B Flat):

1. Lexical Structure
    a. No restrictions.
    b. Specific escape sequences for all of the hex and binary representations, such as '\xXXXXXXX' and '\uXXXXXXX' will not be supported as well.
2. Types
    a. The common notion of the 'var' keyword will not be supported. All variables, objects, et cetera (anything involving space on the stack and heap), will have to be statically declared and/or initialized.
    b. Referring to restriction number one, there would naturally be no type inference in B Flat. Especially since the implementation of type inference would probably prove to be too involved and centered around the concept of ambiguous types.
3. Variables
    a. Variables will be supported in B Flat.
4. Conversions
    a. Conversions will be supported in B Flat.
5. Expressions
    a. Expressions will be supported in B Flat.
6. Statements
    a. Statements will be supported in B Flat.
7. Namespaces
    a. The concept of a 'namespace' from C# will not be supported in B Flat. The equivalent of a namespace in Dart (Start) is a manually prefixed import call to a Dart library.
8. Classes
    a. Classes will be supported in B Flat.
9. Structs
    a. Structs will not be supported in B Flat.
10. Arrays
    a. Arrays will not be supported in B Flat.
11. Interfaces
    a. Interfaces will not be supported in full for B Flat.
12. Enums

    a. Enums will not be supported in B Flat.
13. Delegates
    a. Delegates will not be supported in B Flat.
14. Exceptions
    a. Exceptions will not be supported in B Flat solely for the sake of time.
15. Attributes
    a. Attributes will not be supported in B Flat. Metadata about methods and classes is really not important in a language like this. Fundamentally it is just syntactic sugar for programmers that want to give additional information about the code that they write in their programs.
16. Unsafe code
    a. Unsafe code will not be supported in B Flat. It's basically C#'s way to support pointers within itself. The challenge of B Flat is to support subtyping, inheritance, and generics. We assume that we will not have the time to support pointers in B Flat.
17. Documentation comments
    a. The concept of C#'s 'doc comment's tags' will not be supported in B Flat. The documentation comments that exist in C# ('///' or '/** … **/') are perfectly valid as this special type of comment also exists in Dart (Start) and would be a perfectly reasonable pattern to translate.

Syntax: Below is the syntax of the B ♭ (B Flat) language, which is a subset of C#. The syntax of B Flat will be represented in EBNF. Most of the production rules listed below will coincide with C#'s 6.0 specification to ensure syntactic accuracy with C#'s actual syntax rules. This design decision was made so that the 'cs2dart' compiler project would be easier to develop and improve in the future.

TOKENS

token ::= identifier | keyword | integer_literal | real_literal | character_literal | string_literal | interpolated_string_literal | operator_or_punctuator

KEYWORDS

keyword ::= 'abstract' | 'as' | 'base' | 'bool' | 'break' | 'byte' | 'case' | 'catch' | 'char' | 'checked' | 'class' | 'const' | 'continue' | 'decimal' | 'default' | 'delegate' | 'do' | 'double' | 'else' | 'enum' | 'event' | 'explicit' | 'extern' | 'false' | 'finally' | 'fixed' | 'float' | 'for' | 'foreach' | 'goto' | 'if' | 'implicit' | 'in' | 'int' | 'interface' | internal' | 'is' | 'lock' | 'long' | 'namespace' | 'new' | 'null' | 'object' | 'operator' | 'out' | 'override' | 'params' | 'private' | 'protected' | 'public' | 'readonly' | 'ref' | 'return' | 'sbyte' | 'sealed' | 'short' | 'sizeof' | 'stackalloc' | 'static' | 'string' | 'struct' | 'switch' | 'this' | 'throw' | 'true' | 'unsafe' | 'ushort' | 'using' | 'virtual' | 'void' | 'volatile' | 'while'

IDENTIFIERS

identifier ::= Any string that is not a keyword or literal.

STATEMENTS

statement ::= declaration_statement | embedded_statement

declaration_statement ::= local_variable_declaration ‘;’ | local_const_declaration

local_variable_declaration ::= local_variable_type local_variable_initializer?

local_variable_type ::= type

const_declaration ::= ’const’ type identifier local_variable_initializer?

embedded_statement ::= block | empty_statement | expression_statement |
selection_statement | iteration_statement | jump_statement

block ::= ‘{‘ statement_list? ‘}’

statement_list ::= statement+

empty_statement ::= ’;’

expression_statement ::= statement_expression ‘;’

statement_expression ::= invocation_expression | object_creation_expression | assignment

selection_statement ::= if_statement

if_statement ::= 'if' '(' boolean_expression ')' embedded_statement | 'if' '(' boolean_expression ')'
embedded_statement 'else' embedded_statement

boolean_expression ::= expression

iteration_statement ::= while_statement | for_statment

while_statement ::= ’while’ ‘(‘ boolean_expression ‘)’ embedded_statement

for_statement ::= ’for’ ‘(‘ local_variable_declaration ‘;’ boolean_expression ‘;’
statement_expression ‘)’ embedded_statement

jump_statement ::= ‘return’ expression? ‘;’

EXPRESSIONS

expression ::= primary_expression | assignment_expression | additive_expression | multiplicative_expression | equality_expression | relational_expression | conditional_and_expression | conditional_or_expression

primary_expression ::= literal | simple_name | parenthesized_expression | member_access | invocation_expression | this_access | object_creation_expression | typeof_expression | unary

simple_name ::= identifier

parenthesized_expression ::= ’(‘ expression ‘)’

member_access ::= type ‘.’ identifier | this_access ‘.’ identifier | simple_name ‘.’ identifier ;

Invocation_expression ::= primary_expression '(' argument_list? ')'

argument_list ::= expression (‘,’ expression)*

this_access ::= 'this'

object_creation_expression ::= 'new' type '(' argument_list? ')'

typeof_expression ::= 'typeof' '(' type ')' | typeof' '(' identifier ')'

unary ::= ’false’ | ’true’

assignment_expression ::= primary_expression assignment_operator expression

assignment_operator ::= ‘=’ | ‘+=’ | ‘*=’ | ‘/=’ | ’%=’ ;

additive_expression ::= primary_expression ‘+’ primary_expression | primary_expression ‘-’ primary_express

mutliplicative_expression ::= primary_expression ‘*’ primary_expression | primary_expression ‘/’ primary_expression | primary_expression ‘%’ primary_expression

equality_expression ::= primary_expression ‘==’ primary_expression | primary_expression ‘!=’ primary_expression

relational_expression ::= primary_expression ‘<’ primary_expression | primary_expression ‘>’ primary_expression | primary_expression ‘<=’ primary_expression | primary_expression ‘>=’ primary_expression | primary_expression ‘is’ primary_expression

conditional_and_expression ::= primary_expression ‘&&’ primary_expression

conditional_or_expression ::= primary_expression ‘||’ primary_expression

CLASSES

class_declaration ::= ‘class’ identifier class_Base? class_Body ‘;’?

class_base ::= ‘:’ identifier

class_body ::= ‘{‘ (constant_declaration | local_variable_declaration | method_declaration | constructor_declaration)* ‘}’

NAMESPACE

namespace ::= class_declaration

NON-TRIVIAL FEATURES

Computation Abstraction Non-Trivial Feature: Handle for Inheritance

Non-Trivial Feature #2: Subtyping

Non-Trivial Feature #3: Generics (REQUIREMENT REMOVED THROUGH THE PERMISSION OF PROFESSOR)

Work Planned for Custom Component: Handle for Inheritance. Until the custom component deadline, generics will not be supported.