Version 1.0

The ‘cs2dart’ Compiler

Architecture/Design Document

Table of Contents

Change History

Version: v1.1
Modifier: Matthew Fuller, Pablo Lepe, Dominic Ciliberto, Kevin Zayala, John Aquino
Date: May 12th, 2020
Description of Change: Improved formal syntax definition, added more code snippets, and added more limitations.

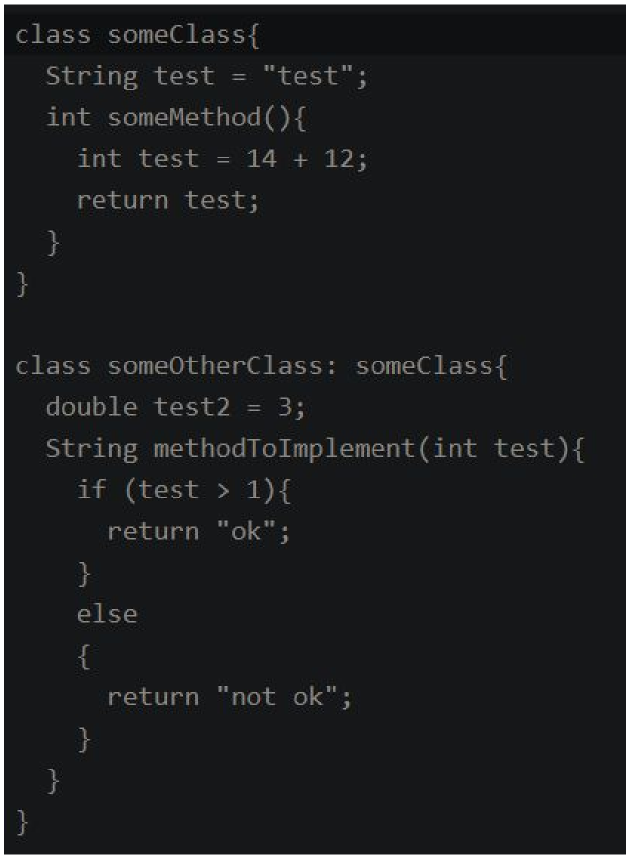
Version: v1.0
Modifier: Matthew Fuller, Pablo Lepe, Dominic Ciliberto, Kevin Zayala, John Aquino
Date: May 12th, 2020
Description of Change: Initial version

1 Why this language, and why this language design

Our compiler named cs2dart facilitates the interconnections of our components. The input language named ‘B Flat’ is just a subset of C#. The target language is Dart, but the output language is just as limited by the input language. Thus we call our compiler’s output language Start.

We wanted this language design because it is very similar to languages that we were using for our other school projects. Both C# and Dart are being utilized for these aforementioned projects. Also Dart is an advanced language that offers flexibility, and C# is a popular language at the moment. In order to offer a suitable alternative for web development and mobile development, cs2dart was created to incorporate Dart to those with C# skills. Additionally, concepts involving OOP are just easy to communicate with other programmers. So we thought that it would be natural to make an OOP language.

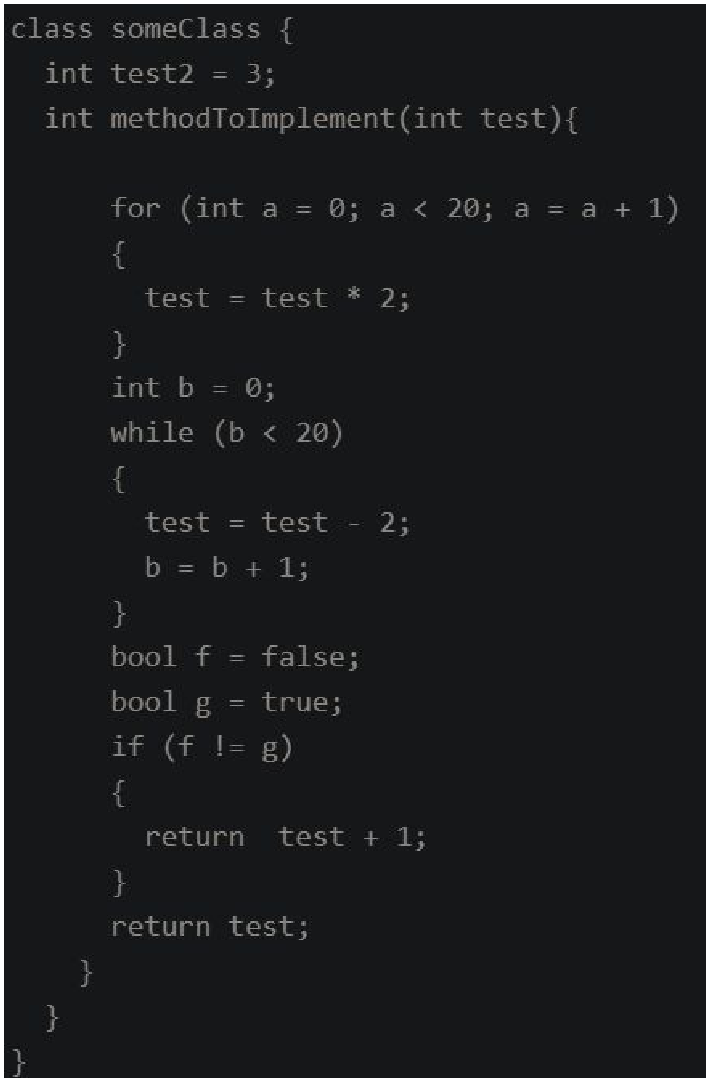
2 Code snippets in your language highlighting features and edge cases, along with relevant explanations



```
class someClass{
  String test = "test";
  int someMethod(){
    int test = 14 + 12;
    return test;
  }
}

class someOtherClass: someClass{
  double test2 = 3;
  String methodToImplement(int test){
    if (test > 1){
      return "ok";
    }
    else
    {
      return "not ok";
    }
  }
}
```

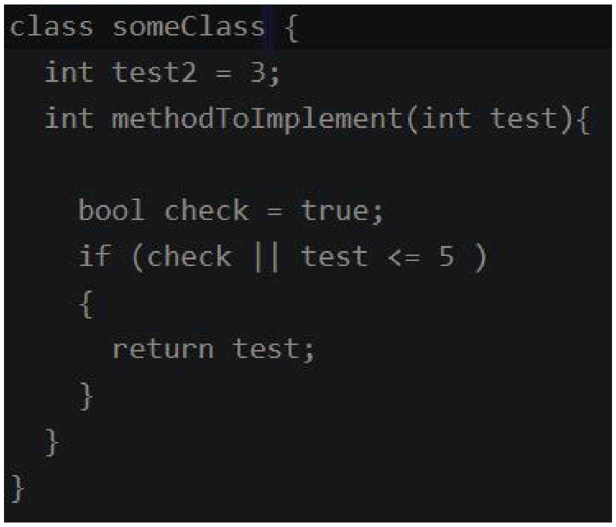
This is a representation of what is capable in our language. In this specific example, we define the classes called someClass and someOtherClass. someOtherClass extends the someClass class, and both class declarations contain method declarations as well as its respectable statements. There are also variable declarations such as “double test2 = 3;” as well as an expression/declaration “int test = 14 + 12;”. An “if” statement exists in methodToImplement() as well, which has the expression “test > 1”. The ‘return’ jump statement also exists in this snippet as an example of usage.



```
class someClass {
  int test2 = 3;
  int methodToImplement(int test){

      for (int a = 0; a < 20; a = a + 1)
      {
        test = test * 2;
      }
      int b = 0;
      while (b < 20)
      {
        test = test - 2;
        b = b + 1;
      }
      bool f = false;
      bool g = true;
      if (f != g)
      {
        return  test + 1;
      }
      return test;
    }
  }
}
```

This snippet shows the ability to use ‘for’ iterators, ‘while’ iterators as well as boolean expressions.



```
class someClass {
  int test2 = 3;
  int methodToImplement(int test){

    bool check = true;
    if (check || test <= 5 )
    {
      return test;
    }
  }
}
```

This snippet shows how to implement a conditional 'or' expression.

3 Known limitations

Here is a list of unsupported feature in our language:

1. Type Inference
    a. Due to the lack of the 'var keyword', naturally there is 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.
2. Namespaces
    a. The concept of a 'namespace' from C# is not supported in B Flat. The equivalent of a namespace in Dart (Start) is a manually prefixed import call to a Dart library.
3. Structs
4. Enums
5. Generics
    a. This was a feature we eventually removed due to time restraints.
6. Arrays
7. Delegates
8. Exceptions
9. Attributes
    a. Attributes are not 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.
10. Unsafe code:
    a. Unsafe code is not supported in B Flat. It's basically C#'s way to support pointers within itself. The challenge of B Flat is to support subtyping and inheritance.

11. Multiple variable declarations in a row ex: int x,y,z
12. Multiple adds or multiplications in a single statement must be wrapped in parends ex:(1+2)+3; | (1*2)*3

4 Knowing what you know now, what would you do differently?

A one to one conversion C# to Dart is simply not possible for a small time frame of one semester. Also programming a compiler was not as simple as we expected. We would have not underestimated the project's scope or have scaled down sooner than later. Overall it was a very humbling project to attempt and complete. Fundamentally if we had more time and the knowledge that we have now, we would have our cake and ate it too.

As far as the project itself, it would have been better to implement parser output as fields instead of elements of a list, and to correctly expect outputs properly in the parser functions.

5 How do I compile your compiler?

Since the implementation language of the compiler is an interpreted language, the main file does not need to be compiled. Please refer to the header below to run the main file of the compiler.

6 How do I run your compiler?

To run the compiler first the Dart SDK must be installed. Instructions on how to do this can be found at dart.dev/get-dart. After installing the Dart SDK run in the terminal the command in the same directory where the compiler's main.dart is located. Also include the filename you are trying to compile as a command line argument.

For instance, below is an example of how to run the compiler in the terminal.

```
> dart main.dart <path/to/compile/file.cs>
```

7 Formal syntax definition

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 ::= type identifier local_variable_initializer?

local_variable_declaration_initializer ::= ‘=’ expression

local_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