Primer

C# Programming Language

C# as a language greatly resembles Java and, to a much lesser extent, C++. Most C# code should therefore not look entirely too alien, but there are a few functionalities of the language that are in use in the compiler that some of you may not be familiar with. Here is a list of such functionalities:

- **Type Inference**:

  Type inference in C# is realized through the `var` keyword, much like `auto` in C++.

  ```
  var x = 5;
  var list = new List<int>();
  ```

- **Tuples**:

  C# tuples are a quick way to aggregate variables without creating a new `struct` or `class`, or to return multiple values from functions. They are formed with parentheses:

  ```
  public (int, string) Foo() {
      return (5, "Hello");
  }

  public void Bar() {
      var (num, str) = Foo();
  }
  ```

- **Nullable Types**:

  New in C# 8.0, nullable reference types are simply the addition of `?` at the end of a type, like so:

  ```
  public void Foo(Bar? bar){
      // ...
  }
  ```

  It is a way to explicitly mark function parameters, variables, class members, etc as being possibly `null`. By extension, types that don't end with `?` are assumed to not be `null`.

- **LINQ**:

  LINQ is a set of methods available on all of C#'s collections. They are analogous to the `fold`, `map`, `filter`, etc, functions founds in functional programming languages. As such, lambda (anonymous) functions are used extensively in conjunction with those methods. Here is an example of a LINQ expression:

  ```
  var list = new List<int>{1, 2, 3, 4};
  list = list.Select(n => n + 1).ToList(); // list now contains [2, 3, 4, 5]
  ```

- **String Interpolation**:

  These are simply a convenient way of creating a string that contains the value of some variables or expression, like so:

  ```
  var n = 5;
  Console.WriteLine($"n is equal to {n}");
  ```

- **Null Propagation**:

  In C#, an alternative to consecutive null checks is null propagation. It looks like this:

  ```
  List<int> list = null;
  var result = list?.Select(n => n + 1)?.ToList(); // result is null because
    list is null
  ```

  Another feature often used in conjunction with null propagation is null coalescing:

  ```
  result ?? new List<int>()
  ```

  This simply returns `result` if it is not `null`, or a new list.

  Finally, there is the null coalesce assignment, which assigns the value only if the left hand side is null:

  ```
  result ??= new List<int>()
  ```

- **Type Casts**:

  While C# supports the traditional type casting syntax of C (`(NewType) variable`), it also possesses two extra operators, `as` and `is`.

  `variable is Type` simply returns a boolean if the variable or expression is castable into the desired type, while `variable as Type` performs the cast but returns `null` on failure. The two can be combined to check if a cast is possible and perform that cast at the same time, like so:

  ```
  if (variable is List<int> listVariable){
      listVariable.Add(5);
  }
  ```

  An extension of this can be used for the `case` value in a `switch` statement.

- **Expression Bodied Functions**:

  This is simply a feature that allows the expression of single line functions or properties in a more succinct manner:

  ```
  public override string ToString() => this.Name;
  // equivalent to
  // public override string ToString()
  // {
  //    return this.Name;
  // }
  ```

- **Extension Methods**:

These are simply syntax sugar that allows us to call `static` methods as if they were a member method of a type through the `this` keyword. For example:

```
public static bool IsPositive(this int n) => n >= 0;
```

can be called either like this: `5.IsPositive()` or like this: `IsPositive(5)`.

- **Variadic Functions**:

  Variadic functions are functions that accept a variable number of arguments. An example would be C's `printf` function. C# possesses a form of variadic functions through the `params` keyword:

  ```
  void Foo(int n, params string[] strings) { /* ... */ }
  ```

  This keyword allows us to call `Foo` like so:

  ```
  Foo(5);
  Foo(5, "first");
  Foo(5, "first", "second");
  ```

  It is essentially just syntax sugar for:

  ```
  Foo(5, new string[]{ /* ... */ });
  ```

ANTLR

ANTLR is a parser generator written in Java that is capable of producing parsers in a number of languages from a relatively simple grammar file.

An ANTLR grammar (.g4) file is comprised of rules, which are essentially patterns that the generated parser will be able to recognize. There are two types of rules, lexer rules that start with an uppercase character, and parser rules that start with a lowercase character. Lexer rules simply exist to define the kinds of symbols that are recognized by the parser, while parser rules define the syntax and structure of the language.

Rules can be grouped using parentheses and can have multiple alternative forms separated by the `|` character. In addition, much like in regular expressions, the `?`, `+`, and `*` symbols can be used to express "optional", "one or more times", and "zero or more times", respectively.

It is possible to create a name for the different alternatives of a parser rules by adding `# SomeName` before the `|` character:

```
valueType
    : 'bool'  # BoolType
    | 'int8'  # IntegralType
    | 'uint8' # IntegralType
    // ...
    ;
```

In this example, a new `BoolType` class is created in the parser which inherits from the base class of the `valueType` rule. This allows us to then type cast the parent rule into `BoolType`, which may prove to be convenient. This has the advantage of reducing the number of redundant parser rules that exist solely to name a group of tokens. However,

naming any of the alternatives to a rule forces us to name all of them.

LLVM

LLVM is a compiler backend infrastructure, which takes an Intermediate Representation of the code as input and produces optimized machine code for the desired infrastructure.

- **Basic blocks (important)**:

  Basic Blocks are the LLVM objects you will interact with the most in the code generator. A basic block is essentially just a label that can be branched to, but they have some restrictions on the kinds of instructions they can contain and where they can occur. It is *highly* recommended to get familiar with them. Reading llvm.org is a good start. They are also conceptually the same as the basic blocks of chapter 9 in the course notes.

LLVMSharp

The C# library we use for bindings to the LLVM API, used for code generation.

Link: https://github.com/microsoft/LLVMSharp

Tools

Visual Studio Code

Extensions: * ANTLR4 grammar syntax support - Github

Jetbrains

Extensions: * ANTLR v4 grammar plugin - Github

Resources

- Compiler explorer

  The `clang` compiler can be used with the `-emit-llvm` flag to see how C or C++ code gets transformed into LLVM IR.

- Mapping High Level Constructs to LLVM

- Swift programming language

  Swift's compiler being based on LLVM, it can be useful at times to see their implementation of different features. They use LLVM's C++ API, but it should be close enough to the C# API to be intelligible.