COMP 430 Language Documentation

Thad House

**Why this language, and why this language design**

This language was arbitrarily designed as a language using syntax already common, and learning how that gets parsed and mapped to lower level code. It was designed to be simpler then a more common language that runs on the CLR, in order to make implementation easier. For instance, generics was not included at all, as the runtime support for that is very complicated. Inheritance could have been implemented without too much trouble, however to save time it was skipped as well. This also avoided the need to support casting, greatly simplifying the language as well.

**Code Snippets**

Found at the CodeSnippets.md file located next to this document.

**Known limitations**

Static variables are not supported.

The namespace syntax can be a bit confusing to remember, especially when calling static methods. However, it greatly simplified the parsing.

Inheritance would be the next thing to add, along with casting.

Methods taking the same parameters, but returning different things will emit, and is supported by the runtime, but calling these methods is undefined behavior which one will be called.

Any identifier with a number causes a parsing failure, due to being tokenized incorrectly.

No current way to emit a library rather then an executable.

No way to get length of an array

Taking a delegate on a static instance will not error, but will fail to run

**Knowing what you know now, what would you do differently**

I would probably do the parser in a different way. It was a bit confusing, and had multiple different ways to handle recursive expressions. The tokenizer worked very well, but would need to forward line and column information to make debugging parsing issues much easier. The code generator is one of the better parts, especially with the abstraction. Supporting multiple backends is awesome, and with a bit more work compiling to completely separate language from runtime, and allow targeting things like the JVM

C# is an awesome language and .NET is an awesome runtime, so I likely wouldn’t change anything about my editor, build environment, or target language. I’m very comfortable in them, and have a lot of experience with the internals and limitations of both.

**Compiling, Running**

**Compiling the compiler**

1. Install .NET Core 3.1 at minimum. <https://dotnet.microsoft.com/download/dotnet-core/3.1>
2. Clone repo
3. CD into CompilerEXE directory
4. dotnet build

**Running the compiler**

1. Follow Compile Steps, stay in CompilerEXE directory
2. dotnet run Program.net
3. The output file will be an executable placed in the CompilerEXE directory.

Additional options can be passed after run.

1. Multiple files can be passed to compile. Any extension that is not .dll or .exe is accepted as a file to attempt to compile.
2. By default, the program name will equal the name of the first compilation file passed. This can be changed with --program-name <program-name>
3. Libraries can be added to be linked to by passing them as normal files, except they must have a .dll or .exe extension.

**Running the compiled program**

1. If on windows, the executable can just be ran from command line by running the compiled exe name.
2. If on Linux or Mac, the latest version of Mono 6 is required to run the executable. .NET Core cannot run the outputted exe. Then, run mono Program.exe

**Syntax**

variable is a variable

classname is the name of a class. Can contain letters, numbers, :: or \_

methodname is the name of a method

fieldname is the name of a field

int, bool, void, and string are aliases to the BCL System.Int32, System.Boolean, System.Void, and System.String types

auto is the inferred type keyword

null is the null reference keyword

method is a keyword

field is a keyword

constructor is a keyword

intconst is an integer constant

stringconstant is a string constant

true is the true constant

false is the false constant

static is the keyword to make a class item static

entrypoint is the keyword to mark a method as the program entry point.

type ::= int | bool | void | string | object | classname

op ::= + | - | \* | / | < | > | == | “||” | && | >= | <=

callparam ::= methodcall | variable | ref variable

methodcall ::= exp.methodname(callparam\*) | new classname(callparam\*) | newarr classname(exp)

fielddec ::= field type variable; | field type variable = exp;

exp ::= variable | intconst | stringconstant | true | false | null | this | exp op exp | methodcall | variable[exp] | exp.variable | exp.methodname | classname.methodname

vardec ::= type variable

paramdec := type var | ref type var

stmt ::= vardec; | vardec = exp; | auto variable = exp; | while (exp) { stmt\* } | break; | { stmt\*} | if (exp) { stmt\* } else { stmt\* } | return exp; | return; | exp = exp;

methoddef ::= method type methodname(paramdec\*) { stmt\* } | method static type methodname(paramdec\*) { stmt\* } | method entrypoint void methodname() { stmt\* }

classdef ::= class classname { fielddec\* constructor(paramdec\*) { stmt\* }\* methoddef\* }

fpdef ::= delegate type classname(paramdec\*);