Tristan Programming Language

From start to finish

Tristan B. V. Kildaire

December 12, 2022

Dedicated to Gisele, Clint and Vaughan for whomst have always cared the most for me even when faced with adversity

Contents

Haha, this page is left intentionally blank because I want to waste paper.

Part I.

Introduction

1. Why in the lords name are you doing this?

Despite my eagerness to jump directly into the subject matter at hand I think believe there is something of even greater importance. Despite there being a myriad of reasons I embarked upon this project something more important than the stock-and-standard “I needed it to solve a problem of mine” reasoning comes to mind. There is indeed a better reason for embarking on something that the mere *technical requirement* thereof - I did this **because I can**. This sentiment is something that I really hold dear to my heart despite it being a seemingly obvious one. Of course you can do what you want with your code - it’s a free country. One would not be wrong to make such a statement but mention your ideas online and you get hounded down by others saying “that’s dumb, just use X” or “your implementation will be inefficient”. These statements are not entirely untrue but they miss the point that this is an exercise in scientific thinking and an artistic approach at it in that as well.

I would not want to see the naysayers put anyone down from doing something they have always dreamed of, that is why I have done this. I am aware of the risks and the downfalls of having grandeause expectations but luckily I do not require the external feedback of the mass - just some close few friends who can appreciate my work and join the endeavor with me.

Don’t let people stop you, you only have one life - take it by the horns and fly

2. Aims

A programming language normally has an aim, a *purpose of existence*, to put it in a snobbish way that a white male like me would. It can range from solving a problem in a highly specific domain (such are Domain Specific Language (TODO: add citation)) to trying to solving various problems spread across several different domains, a *general purpose* programming language. This is where I would like to place Tristan - a language that can support multiple paradigms of programming - whether this be object-oriented programming with the usage of *classes* and *objects* or functional programming with techniques such as map and filter.

Tristan aims to be able to support all of these but with certain limits, this is after all mainly an imperative language with those paradigms as *"extra features"*. Avoiding feature creep in other systems-levels languages such as C++ is something I really want to stress about the design of this language, I do not want a big and confusing mess that has an extremely steep learning curve and way too many moving parts.

One should not require the knowledge of more than two different paradigms in order to understand the usage of a standard library function as an example. If a user is looking at the documentation of a given function call then at most the amount of concepts required to understand it should be two, for example a *templatised* and *object-based* function would be the upper bound on concepts allowed.

2.1. Paradigms

Tristan is a procedural programming language that supports object-oriented programming and templates.

2.1.1. Object-oriented programming

Object orientation allows the programmer to create user-defined types which encapsulate both data fields and methods which act upon said data. Tristan supports:

1. Class-based object orientation
    a) Classes as the base of user-defined types and objects are instances of these types
    b) Single inheritance hierachy
    c) Runtime polymorhpism

2. Interfaces
    a) Multiple inheritance
    b) Runtime polomprhism (thinking\hyperref{})

It is with this lean approach to object orientation that we keep things simple enough (only single inheritance) but with enough power to model the real world in code (by supporting interfaces).

2.1.2. Templating

Templating, otherwise known as *generics*, is a mechanism by which a given body of code which contains a type specifier such as variable declarations or function definitions can have their said type specifiers parameterized. The usage of this can be illustrated in the code below, where we want to define a method `sum(a, b)` which returns the summation of the two inputs. We define version that works for integral types (`int`) and a version that works for decimal types (`float`):

```
// Integral summation function
int sum(int a, int b)
{
  return a+b;
}

// Decimal summation function
float sum(float a, float b)
{
  return a+b;
}
```

Being a small example we can reason about the easiness of simply defining two versions of the `sum(a, b)` method for the two types, but after some time this can either get overly repetitive if we have to do this for more methods of a similar structure *or* when more types are involved. This is where templating comes in, we can write a more general version of the same function and let the compiler generate the differently typed versions dependent on what *type parameter* we pass in.

A templatised version of the above `sum(a, b)` function is shown below:

```
// Templatised function
template T
{
  T sum(T a, T b)
  {
    return a+b;
  }
}

```

```
// Integral version
sum!(int)(1,2)

// Decimal version
sum!(float)(1.0,2.0)
```

The way this works is that whenever you call the function `sum(a, b)` you will have to provide it with the specific type you want generated for that function.

2.2. Systems-level access

Tristan does not shy away from features which give you access to system-level concepts such as memory addresses (via pointers), assembly (via the inline assembler) and so on. Such features are inherently unsafe but it is this sort of control that I wish to give the user, the balance between what the compiler should do and what the user should make sure they are doing is tipped quite heavily in favor of the latter in my viewpoint and hence we support such features as:

- Weak typing
  - By default this is not the behavior when using `cast()`
  - Casting to an incompatible type is allowed - even when a run-time type-check is invalid you can still force a cast with `castunsafe()`
  - The user should be able to do what *he* wants if requested
- Pointers
  - The mere *support* of pointers allowing one to take a memory-level view of objects in memory rather than the normal "safe access" means
- Inline assembly
  - Inserting of arbitrary assembler is allowed, providing the programmer with access to systems level registers, interrupts/syscall instructions and so on
- Custom byte-packing
  - Allowing the user to deviate from the normal struct packing structure in favor of a tweaked packing technique
  - Custom packing on a system that doesn't agree with the alignment of your data **is** allowed but the default is to pack accordingly to the respective platform

Part II.

Users guide

3. TODO Basics syntax

Part III.

Implementation

4. Lexical analysis

Lexical analysis is the process of taking a program as an input string A and splitting it into a list of n sub-strings A_1, $A_2 \ldots A_n$ called *tokens*. The length n of this list of dependent on several rules that determine how, when and where new tokens are built - this set of rules is called a *grammar*.

4.1. Grammar

The Tristan grammar is specified in EBNF below:

TODO: We need to derive a grammar/come up with one (and include explanations of EBNF).

4.2. Overview of implementation

The source code for the lexical analysis part of the compiler is located in `source/tlang/lexer.d` which contains two important class definitions:

- `Token` - This represents a token
  - Complete with the token string itself, `token`. Retrivebale with a call to `getToken()`
  - The coordinates in the source code where the token begins as `line` and `column`
  - Overrides equality (`opEquals`) such that doing,

    ```
    new Token("int") == new Token("int")
    ```

  - would evaluate to `true`, rather than false by reference equality (the default in D)
- `Lexer` - The token builder
  - `sourceCode`, the whole input program (as a string) to be tokenized
  - `position`, holds the index to the current character in the string array `sourceCode`
  - `currentChar`, the current character at index-`position`
  - Contains a list of the currently built tokens, `Token[] tokens`
  - Current line and column numbers as `line` and `column` respectively
  - A "build up" - this is the token (in string form) currently being built - `currentToken`

4.3. Implementation

The implementation of the lexer, the `Lexer` class, is explained in detail in this section. (TODO: constructor) The lexical analysis is done one-shot via the `performLex()` method which will attempt to tokenize the input program, on failure returning `false`, `true` otherwise. In the successful case the `tokens` array will be filled with the created tokens and can then later be retrieved via a call to `getTokens()`.

Example usage:
TODO

4.3.1. performLex()

TODO: This is going to change sometime soonish, so I want the final version of how it works here. I may as well, however, give a brief explanation as I doubt *much* will change - only specific parsing cases.

This method contains a looping structure which will read character-by-character from the `sourceCode` string and follow the rules of the grammar (TODO: add link), looping whilst there are still characters available for consumption (`position < sourceCode.length`).

We loop through each character and dependent on its value we start building new tokens, certain characters will cause a token to finish being built which will sometimes be caused by `isSpliter(character)` being `true`. A typical token building process looks something like the following, containing the final character to be tacked onto the current token build up, the creation of a new token object and the addition of it to the `tokens` list, finishing with flushing the build up string and incrementing the coordinates:

A typical token building procedure looks something like this:

```
/* Generate and add the token */
currentToken ~= "'";
currentTokens ~= new Token(currentToken, line, column);

/* Flush the token */
currentToken = "";
column += 2
position += 2;
```

4.3.2. Character and token availability

Helper functions relating to character and token availability.

hasToken()

Returns `true` if there is a token currently built i.e. `currentToken.length != 0`, `false` otherwise.

isBackward()

Returns `true` if we can move the character pointer backwards, `false` otherwise.

isForward()

Returns `true` if we can move the character pointer forward, `false` otherwise.

4.3.3. isNumericalStr()

This method is called in order to chck if the build up, `currentToken`, is a valid numerical string. If the string is empty, then it returns `false`. If the string is non-empty and contains anything *other* than digits then it returns `false`, otherwise is returns `true`.
TODO

4.3.4. isSpliter()

This method checks if the given character is one of the following:

- character == ';' || character == ',' || character == '(' || character == ')' || character == '[' || character == ']' || character == '+' || character == '-' || character == '/' || character == '%' || character == '*' || character == '&' || character == '{' || character == '}' || character == '=' || character == '|' || character == '^' || character == '!' || character == '\n' || character == '~' || character =='.' || character == ':';
- `; , ( ) [ ] + - / % * & { }`
- `=` | (TODO: make it texttt) `^ !`
  `n`(TODO: \n not appearing) `~ .`

Whenever this method returns `true` it generally means you should flush the current token, start a new token add the offending spliter token and flush that as well.

5. Code emit

The code emit process is the final process of the compiler whereby the `initQueue`, `codeQueue` and all assorted auxilllary information is passed to an instance of `CodeEmitter` (in the case of the C backend this is sub-typed to the `DGen` class) such that the code can be written to a file. At this stage all queues consist simpyl of instances of the `Instruction` class.

Our C backend or *custom code emitter*, `DGen`, inherits from the `CodeEmitter` class which specifies that the following methods must be overriden/implemented:

1. `emit()`
   a) Begins the emit process
2. `finalize()`
   a) Finalizes the emitting process (only to be called after the 'emit()' finishes)
3. `transform(Instruction instruction)`
   a) Transforms or emits a single Instruction and returns the transformation as a string

5.1. Custom code emits

We override/implement the `transform(Instruction instruction)` in `DGen` to work somewhat as a big if-statement that matches the different sub-types of Instructions that exist, then the respective code-emit (C code) is generated. This method has the potential to be recursive as some instructions contain nested instructions that must be transformed prior before the final transformation, in which case a recursive call to `transform(Instruction)` is made.

5.1.1. Code emit example: Variable declarations

For example, with the `VariableDeclaration` instruction we have such an emit which fetches the type of the variable being declared and its name, then concatenates these toghether with spaces and a trailing semi-colon:

```
/* VariableDeclaration */
else if(cast(VariableDeclaration)instruction)
{
  VariableDeclaration varDecInstr = cast(VariableDeclaration)instruction;
  Context context = varDecInstr.getContext();

  auto typedEntityVariable = context.tc.getResolver().resolveBest(context.getContainer(), varDecInstr.varName); //TODO: Remove `auto`
  string typedEntityVariableName = context.tc.getResolver().generateName(context.getContainer(), typedEntityVariable);

  //NOTE: We should remove all dots from generated symbol names as it won't be valid C (I don't want to say C because
  // a custom CodeEmitter should be allowed, so let's call it a general rule)
  //
  //simple_variables.x -> simple_variables_x
  //NOTE: We may need to create a symbol table actually and add to that and use that as these names               //could get out of hand (too long)
  // NOTE: Best would be identity-mapping Entity's to a name
  import compiler.codegen.mapper : SymbolMapper;
  string renamedSymbol = SymbolMapper.symbolLookup(context.getContainer(), varDecInstr.varName);

  return varDecInstr.varType~" "~renamedSymbol~";";
}
```

5.1.2. Symbol renaming

Of the many sub-types of the `Instruction` class, when it comes to time to `emit()` some code we do need to apply certain fixups to some of the symbol names (if any) which will appear in the final emitted code. Nmaely, we need to remove any periods in the name such that something like `simple_variables.x` becomes `simple_variables_x`. The reason for this is because when we emit C code using the DGen code emitter we will need to have valid C names which cannot contain characters such as periods.

The method used in order to do this is `symbolRename()` from the `mics.utils` module which takes in the symbol name and returns the *"fixed-up"* version.

6. Parsing

TODO: Add lexer information here

7. Dependency idk

TODO: Add lexer information here