Tristan Programming Language

From start to finish

Tristan B. V. Kildaire

December 20, 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

2.3. Specified behaviour

TODO: Insert ramblings here about underspecified behaviour and how they plague C and how we easily fix this in tlang

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, $A_2 \dots A_n$ 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,

```
1 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:

```
1 /* Generate and add the token */
2 currentToken ~= "'";
3 currentTokens ~= new Token(currentToken, line, column);
4
5 /* Flush the token */
6 currentToken = "";
7 column += 2
8 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. Parsing

Once we have generated a list of tokens (instances of `Token`) from the `Lexer` instance we need to turn these into a structure that represents our program's source code *but* using in-memory data-structures which we can traverse and process at a later stage.

5.1. Overview

The `Parser` class contains several methods for parsing different sub-structures of a TLang program and returning different data types generated by these methods. The parser has the ability to move back and forth between the token stream provided and fetch the current token (along with analysing it to return the type of symbol the token represents - known as the `SymbolType`).

For example, the method `parseIf()` is used to parse if statements, it is called on the occurence of the token of `if`. This method returns an instance of type `IfStatement`. Then there are methods like `parseBody()` which is responsible for creating several sub-calls to methods such as `parseIf()` and building up a list of `Statement` instances (the top-type for all parser nodes).

The entry point to call is `parse()` which will return an instance of type `Module`.

5.2. API

The API exposed by the parser is rather minimal as there isn't much to a parser than controlling the token stream pointer (the position in the token stream), fetching the token and acting upon the type or value of said token. Therefore we have the methods summarised below:

1. `nextToken()`
    a) Moves the token pointer to the next token
2. `previousToken()`
    a) Moves the token pointer to the previous token
3. `getCurrentToken()`
    a) Returns the current `Token` instance at the current token pointer position
4. `hasTokens()`

a) Returns `true` if there are tokens still left in the stream (i.e. `tokenPtr < tokens.length`), `false` otherwise

5.3. Initialization

The initialization of the parser is rather simple, an instance of the `Parser` class must be instantiated, along with this the following arguments must be provided to the constructor:

1. `Token[] tokens`
   a) This is an array of `Token` to be provided to the parser for parsing. This would have been derived from the `Lexer` via its `performLex()` and `getTokens()` call.

A new instance woud therefore be created with something akin to:

```
// Tokenize the following program
string sourceCode = "int i = 2;"
Lexer lexer = new Lexer(sourceCode);
lexer.performLex();

// Extract tokens and pass to the lexer
Token[] tokens = lexer.getTokens();
Parser parser = new Parser(tokens);
```

5.4. Symbol types

The token stream is effectively a list of instances of `Token` which consist just of the token itself as a string and the coordinates of the token (where it occurs). However, some tokens, despite being different strings, can be of the same type or *syntactical grouping*. For example one would agree that both tokens `1.5` and `25.2` are both different tokens but are both floating points. This is where the notion of symbol types comes in.

The enum `SymbolType` in `parsing/symbols/check.d` describes all of the available *types* of tokens there are in the grammar of the Tristan programming language like so:

```
public enum SymbolType {
  LE_SYMBOL,
  IDENT_TYPE,
  NUMBER_LITERAL,
  CHARACTER_LITERAL,
  STRING_LITERAL,
  SEMICOLON,
  LBRACE,
  ...
}
```

Given an instance of `Token` one can pass it to the `getSymbolType(Token)` method which will then return an enum member from `SymbolType`. When a token has no associated symbol type then `SymbolType.UNKNOWN` is returned. Now for an example:

```
// Create a new token at with (0, 0) as coordinates
Token token = new Token("100", 0, 0);

// Get the symbol type
SymbolType symType = getSymbolType(token);
assert(symType == SymbolType.NUMBER_LITERAL);
```

This assertion would pass as the symbol type of such a token is a number literal.

5.4.1. API

The API for working with and using `SymbolTypes` is made available within the `parsing/data/check.d` and contains the following methods:

1. `isType(string)`
   a) Returns `true` if the given string (a token) is a built-in type
   b) Built-in type strings would be: `byte, ubyte, short, ushort, int, uint, long, ulong, void`
2. `getSymbolType(Token)`
   a) Returns the `SymbolType` associated with the given `Token`
   b) If the token is not of a valid type then `SymbolType.UNKNOWN` is returned
3. `getCharacter(SymbolType)`
   a) This performs the reverse of `getSymbolType(Token)` in the sense that you provide it a `SymbolType` and it will return the corresponding string that is of that type.
   b) This will work only for back-mapping a sub-section of tokens as you won't get anything back if you provide `SymbolType.IDENT_TYPE` as there are infinite possibiltiies for that - not a fixed token.

5.5. How to parse

The basic flow of the parser involves the following process:

1. Firstly you need an entry point, this entry point for us is the `parse()` method which will return an instance of `Module` which represents the module - the TLang program.
2. Every `parseX()` method gets called by another such method dependent on the current symbol (and sometimes a lookahead)
   a) For example, sometimes when we come across `SymbolType.IDENTIFIER` we call `parseName()` which can then either call `parseFuncCall()`, `parseTypedDeclaration()` or `parseAssignment()`. This requires a lookahead to check what follows the identifier because just by itself it is too ambuguous grammatically.

b) After determining what comes next the token is pushed back using `previousToken()` and then we proceed into the correct function

c) Lookaheads are rare but they do appear in situations like that

3. The `parseX()` methods return instances of `Statement` which is the top type for all parser-generated nodes or *AST nodes*.

4. When you are about to parse a sub-section (like an if statement) of a bigger syntax group (like a body) you leave the *offending token* as the current token, then you call the parsing method (in this case `parseIf()`) and let it handle the call to `nextToken()` - this is simply the structure of parsing that TLang follows.

5. Upon exiting a `parseX()` method you call `nextToken()` - this determines whether this method would continue parsing or not - if not then you return and the caller will continue with that current token and move on from there.

5.5.1. Example of parsing if-statements

We will now look at an example of how we deal with parsing if statements in our parser, specifically within the `parseBody()`. The beginning of this method starts by moving us off the offending token that made us call `parseBody()` (hence the call to `nextToken()`). After which we setup an array of `Statement` such that we can build up a *body* of them:

```
gprintln("parseBody(): Enter", DebugType.WARNING);

Statement[] statements;

/* Consume the `{` symbol */
nextToken();
```

Now we are within the body, as you can imagine a body is to be made up of several statements of which we do not know how many there are. Therefore we setup a loop that will iterate till we run out of tokens:

```
while (hasTokens())
{
  ...
}
```

Next thing we want to do if grab the current token and check what type of symbol it is:

```
while (hasTokens())
{
  /* Get the token */
  Token tok = getCurrentToken();
  SymbolType symbol = getSymbolType(tok);
  gprintln("parseBody(): SymbolType=" ~ to!(string)(symbol));

  ...
}
```

Following this we now have several checks that make use of `getSymbolType(Token)` in order to determine what the token's type is and then in our case if the token is `"if"` then we will make a call to `parseIf()` and append the returned Statement-sub-type to the body of statements (`Statement[]`):

```
while(hasTokens())
{
  ...

  /* If it is a branch */
  else if (symbol == SymbolType.IF)
  {
    statements ~= parseIf();
  }

  ...
}
```

6. Dependency idk

TODO: Add lexer information here

7. 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 simply 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

7.1. Queues

There are several notable queues that the `CodeEmitter` class contains, these are as follows:

1. `initQueue`
    a) Despite its name this holds instructions for doing memory allocations for static entities (**not** initialization code for said entities)
2. `globalsQueue`
    a) This queue holds instructions for the globals executions. This includes things such as global variable declarations and the sorts.
3. Function definitions map
    a) This is a string-to-queue map which contains the code queues for every function definition.

Along with these queues there are some methods used to amniopulate and use them, these are:

1. `selectQueue(QueueType, string)`
   a) Select the type of queue: `ALLOC_QUEUE` (for the `initQueue`), `GLOBALS_QUEUE` (for `globalsQueue` and `FUNCTION_DEF_QUEUE` (for the function definitions queue)
   b) For function definitions, the optional string argument (second argument) must specify the name of the function definition you would wish to use. An invalid name will throw an error.
   c) This automatically calls `resetCursor()`.
2. `nextInstruction()`
   a) Moves the cursor to the next instruction. Throws an exception if out of bounds.
3. `previousInstruction()`
   a) Moves the cursor to the previous instruction. Throws an exception if out of bounds.
4. `resetCursor()`
   a) Resets the position of the instruction pointer to 0.
5. `getCurrentInstruction()`
   a) Retrieves the current instruction at the cursor.

7.2. 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.

7.2.1. Code emit example: Variable declarations

The example below is the code used to transform the in-memory representation of a variable declaration, known as the `VariableDeclaration` instruction, into the C code to be emitted:

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

```
6
7   Variable typedEntityVariable = cast(Variable)context.tc.getResolver().
      resolveBest(context.getContainer(), varDecInstr.varName);
8
9   string renamedSymbol = SymbolMapper.symbolLookup(typedEntityVariable);
10
11  return varDecInstr.varType~" "~renamedSymbol~";";
12 }
```

What we have here is some code which will extract the name of the variable being declared via \texttt{varDecInstr.varName} which is then used to lookup the parser node of type `Variable`. The `Variable` object contains information such as the variable's type and also if a variable assignment is attached to this declaration or not.

TODO: Insert code regarding assignment checking

Right at the end wwe then build up the C variable declaration with the line:

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

7.2.2. Symbol renaming

In terms of general code emitting we could have simply decided to use the TLang-esque symbol name structure where entities are seperated by periods such as `simple_module.x` where `simple_module` is a container-type such as a `module` and `x` is some entity within it, such as a variable. However, what we have decided to do in the emitter process, specifically in `DGen` - our C code emitter - is to actually rename these symbols to a hash, wherever they occur.

The renaming mechanism is hanlded by the `symbolLookup(Entity)` method from the `SymbolMapper` class. This method takes in a single argument:

1. `entity`
    a) This must be a type-of `Entity`, this is the entity of which the symbol renaming should be applied on.

This allows one do then translate the symbol name with the following usage. In this case we want to translate the symbol of the entity named `x` which is container in the module-container named `simple_variables_decls_ass`. Therefore we provide both peices of information into the function `symbolLookup`:

```
1 // The relative container of this variable is the module
2 Container container = tc.getModule();
3
4 // Lookup a variable named "x"
5 string varLookup = "x"
6
```

```
// The Variable (type-of Entity)
Variable variable = cast(Variable)tc.getResolver().resolveBest(context.
    getContainer(), varLookup);

// Symbol map
string renamedSymbol = SymbolMapper.symbolLookup(variable);

// renamedSymbol == t_c326f89096616e69e89a3874a4c7f324
```

The resulting hash is generated by resolving the absolute path name of the entity provided, applying an md5 hash to this name and then pre-pending a `t_` to the name. Therefore for the above code we will have `simple_variables_decls_ass.x` mapped to a symbol name of `t_c326f89096616e69e89a3874a4c7f324` to be emitted into the C code file.