Aurora Compiler Documentation

Release alpha-0.1.0

Preston Hager

Apr 03, 2020

Contents

CHAPTER 1

Introduction

1.1 Contents

1. The Introduction contains a list of contents and other useful information.
2. The Plan describes the process of writing this compiler.
3. The Lexer, Parser, and Generator pages describe the function for each aspect respectively.
4. The Subset pages describe subsets of the language that have been implemented through the development of the language.
5. The Syntax has an in depth description of the syntax and grammar of the Aurora.
6. The Libraries page has information about how to create, install, publish, and include libraries in the Aurora language.

CHAPTER 2

Plan

2.1 What is “this” exactly?

The Aurora Compiler is just yet another compiler. It’s goal is to help me (the author) practice and learn writing compilers. Eventually this will lead to making a compiler in either C or Assembly for my OS, Startaste.

2.2 What technology is used?

This compiler is written entirely in Python, and compiles to assembly. GitHub is used for version control and releases, and Read the Docs is used for documentation.

2.3 What are the features?

The Aurora Compiler includes a Lexer, or Tokenizer, which creates an array (list) of tokens for the parser. The next item is a parser, it will put the tokens into a logical order for the generator. Next up, the generator. This will create Assembly code based on what the parser outputs and then the compiler can run an assembler such as NASM on the file to assemble it to bytecode.

CHAPTER 3

Lexer/Tokenizer

3.1 What does this do?

The lexer or tokenizer (hereby referred to as “the lexer”), will read the given content — usually a file — character by character without backtracking. The lexer detects spaces, separators, or other key characters to *tokenize* the content. These tokens are in a array (list), with the token type, the value, and the position(s). The parser will take these tokens next. For more see the Parser page, or **‘Syntax page.‘_**

3.1.1 Input & Output

The input is the actual code. This is in an ASCII or Unicode encoding, basically plain text. The output is a list of tokens, which is usually shown as a table in human-friendly formats. A token is yet another list, or rather a tuple, containing the token name, token value, and position. The following is an example of an input and output for the Lexer.

Input:

```
print>"Somewhere over the rainbow!";
```

Output:

Token ID	Token Value	Position
WORD	print	1, 1
FUNC	>	1, 6
STRING_DEF	“	1, 7
WORD	Somewhere	1, 8
SPACE		1, 16
WORD	over	1, 17
SPACE		1, 21
WORD	the	1 22
SPACE		1, 25
WORD	rainbow!	1 26
END_STRING_DEF	“	1, 34
ENDLINE	;	1, 35

CHAPTER 4

Parser

4.1 What does this do?

The Parser takes the array (list) of tokens generated by the Lexer and turns it into an Abstract Syntax Tree (AST). An AST is a visual representation of how code generation might be completed. This means it takes the ordered list of tokens and turns it into something a Generator can understand and create code out of.

4.1.1 Input & Output

The input is a list of tokens generated by the Lexer. These tokens can also be displayed in a human-friendly format of a table. The output is an Abstract Syntax Tree (AST), using Python classes. The following is an example of input and output of the Parser.

Input:

Token ID	Token Value	Position
WORD	print	1, 1
FUNC	>	1, 6
STRING_DEF	“	1, 7
WORD	Somewhere	1, 8
SPACE		1, 16
WORD	over	1, 17
SPACE		1, 21
WORD	the	1 22
SPACE		1, 25
WORD	rainbow!	1 26
END_STRING_DEF	“	1, 34
ENDLINE	;	1, 35

Output:

```
<ASTNode(1) FUNCTION: [
  <ASTValue NAME: "print">,
  <ASTNode(1) ARGUMENTS: [
    <ASTValue STRING: "Somewhere over the rainbow!">
  ]>
]>
```

CHAPTER 5

Generator

5.1 What does this do?

The Generator is the next, and the near last step of the compiler. It creates Assembly code based on the AST that the parser generates. This is quite hard because you're taking abstract ideas and concepts and putting them into assembly code.

5.1.1 Input & Output

The input is an Abstract Syntax Tree (AST) from the Parser. The output is assembly code that can be assembled using something like NASM.

Input:

```
<ASTNode(1) FUNCTION: [
  <ASTValue NAME: "print">,
  <ASTNode(1) ARGUMENTS: [
    <ASTValue STRING: "Somewhere over the rainbow!">
  ]>
]>
```

Output:

```
mov [_aurora_arg_buffer+4], DWORD _aurora_[hex_string]_string_0
call print

_aurora_arg_buffer times 32 dq 0
_aurora_[hex_string]_string_0 db "Somewhere over the rainbow!", 0
```

CHAPTER 6

Subset 1

The first subset of the Aurora Language. The following will describe what it includes, and what the end goal of the subset is. The first subset is simply a way of measuring when part of the compiler is finished. It includes all the functions that will be compile-abled within the first subset. Future subsets will be describe as the language develops further.

6.1 Features

The first subset will include function calls. Includes, A.K.A. imports. And variable definitions for numbers and strings.

Any call to any function name can be called. This is really the base of Aurora. Just specify the function name, use the > character, and follow it with any arguments passed to the function. A function is closed by either the < or `;` tokens. For example the following will call a function called `greet` with an argument of `"Aurora"`: `greet>"Aurora";`. This does the same: `greet>"Aurora"<;`.

Thirdly variable definitions of strings will be possible. This means that variables with the content of a string may be defined, and called upon. For example if the variable `String:  hello = "Salutations to you!";` may be defined and then later called with `println>hello;`. This will in turn print the line `Salutations to you!` with a newline at the end.

Number variables are another variable that can be defined. This can be done by specifying the `Number` variable. For example, `Number:  age = 15;` will define a variable named `age` with a value of `15`.

More on specific syntaxes can by found on the Syntax page.

CHAPTER 7

Subset 2

The second subset of the Aurora Language. The following will describe what it includes, and what the end goal of the subset is. Much like the first subset, the second is a way of measuring what has been completed.

7.1 Features

The second subset will include function definitions, math operations, and variable assignment.

A function may be defined by the programmer, to take in arguments and return a value. When these arguments are defined by variable in the function they are called parameters. More on how to define a function in the Syntax page.

Math is an extremely important aspect of programming. Now you can add, subtract, multiply, and divide numbers. For example to define a number variable with the product of 3 and 5 it would be `Number: age = 3 * 5;`. This method also works with variables. For example: `Number: old = age * 4;` would assign the value of 4 times `age` to `old`.

Variable assignment means you don't have to redefine a variable each time you want to assign a new value. So if we want to multiply `age` by 4 without reassigning it to `old`, but keeping it in `age` we can write: `age = age * 4;`. When adding or subtracting, you can do it without referencing the variable twice by either the += or -= operations. This cannot be done with multiplication nor division. Additionally, the ++ and -- operations will add and subtract 1 respectively.

More on specific syntaxes can by found on the Syntax page.

CHAPTER 8

Syntax

8.1 Definitions

The basis of all programming languages is definitions and math. To define a variable in Aurora is simple. You must know the *type*, the *name*, and a *value*. Definition is then as follows.

```
[Type]: [Name] = [Value]
```

There are many types, and one can also create their own variable type. Names consist of standard ASCII characters, underscores, and digits; a name must also not start with a digit (number). A value is constricted to that variable's type. For example if a variable's type is `Number` then it must be an integer or float (a number). If the type is `String` then the value must be an array of characters defined by `"[Value]"`. Notice the double quotation marks on either side of the value.

A pointer can be "defined" by using square brackets (`[` and `]`). So `[video_memory] = byte ascii>"L";` will assign the value of `byte ascii>"L"` to the memory at the location of `video_memory`. `video_memory` is most likely around `0x8b000` and will manipulate the screen somehow.

A list of types with their expected values can be found on the Types page.

8.2 Variable Manipulation

Variables can be manipulated by the = token, or assigner. This can only be done if the variable is already defined though. So the following code would throw an error that `age` isn't defined, but the `greeting` variable would work fine.

```
String: greeting = "Hello world!";
greeting = "Somewhere over the rainbow!";
age = 42;
```

There are also four more assignment tokens that can be used. The += and -= will add or subtract that value from the variable. The ++ and -- will add or subtract exactly 1 from the value of the variable. These operations do not work on strings, only numbers.

The size of the value might matter sometimes. The default is a `double word`, or 32-bit, value. You may also specify a `word`, 16-bit value, or `byte`, 8-bit value. Use these keywords in front of the value, after the assignment token. For example: `money = word 65536;` will assign the maximum value to `money`. Thus, `money = byte 260;`, which is 4 over the maximum of a byte (256) and will either fail or wrap around to 4. Needless to say it will do some wacky things, and is generally avoided by using double words unless needed.

8.3 Math

The other basic of programming languages is math. Basic level math consists of addition, subtraction, multiplication, division, powers, grouping, and order of operations. These are all included in Aurora and can be used whenever numbers are involved. As such the addition, subtraction, multiplication, division, and powers are used whenever the following characters are used in that context, respectively. `+`, `-`, `*`, `/`, `^`. Groups are defined by opening and closing parentheses (`()`). For more complex math, there is a built-in math library with more functions.

8.4 Functions

Functions are a large part of most programming languages. For older more retro languages, labels are usually used, these however, can become confusing and over crowded. Functions take an input and return an output based on those inputs. Some functions might also have no input, or no output.

8.4.1 Defining a Function

Functions are defined similarly to variables. The following format can be used, where anything in the square brackets is replaced with their described values.

```
func: [Function Name]>[Argument 1]::[Type], [Argument 2]::[Type...] => [Return Type];
end;
```

The Function Name is used whenever the function is called or invoked. The Argument(s) are required parameters to execute the function. And the Return Type is the type of variable returned by the function. This may be Void, saying that the function will not return a specific type. Also note the `end` tag after the function definition. This is required so that aurora can tell when to stop executing code from the function. It will also stop executing code if a value is returned from the function.

A function's parameters and return type may also be nothing. Parameters may include optional or predefined values. The following is an example of a function taking no arguments.

```
func: foo_bar> => Number;
    return 32;
end;
```

This function, named "foo_bar", can be called using `foo_bar>;` and will return the number `32`. The following is an example of a function with one required, and one optional argument.

```
func: rainbows>colors::String{}, pretty::Number=1;
    for>Number: i=0, i?<len>colors<, i++;
      print>colors{i}+"-";
    println>"";
    if>pretty ?= 1;
```

(continues on next page)

(continued from previous page)

```
      println>"It's a pretty rainbow.";
    else;
      println>"It's just a rainbow.";
end;
```

8.4.2 Calling a Function

A defined function may be called as well. If it isn't defined then it cannot be called. For example in a completely empty file the function `bar_bar` does not exist. But, if it is defined prior to calling it, such as in the following code, then it may be called.

```
func: bar_bar> => Void;
    println>"foo foo";
end;

bar_bar>;
```

8.4.3 Including Functions

In an empty aurora file, no functions are defined except one, `include`. The include function can "import" or "include" other functions into the file. This helps with freeing space, and making things look nicer. There are also predefined libraries which can be imported without any extra installation. The full list can be found on the Libraries page, however the most basic ones are the I/O library by name `io`, and the String library by name `string`.

In most of the previous example, the function `print` or `println` is called. However, to do this the following line must be added to the top of the page, `include>io;`. This will "import" or "include" the I/O library so that the functions `print` and `println` can be called later in the code.

CHAPTER 9

Libraries

There are three different types of libraries. Predefined libraries, are already defined by the aurora language, and are included without having to do extra work. User libraries, are defined by you, the programmer, and can be included in any local file. Community libraries, are any user libraries that has been made by another programmer, or found from another source, such as the internet.

9.1 Predefined Libraries

The following is a list of predefined libraries, their purpose, and short list of their functions. More for each library can be found in their documentation.

- I/O Library - used for input and output operations. `io_in`, `io_out`.
- String Library - used for string operations. `ascii`, `len`.

9.2 User Libraries

A programmer may also define their own library for use in local code. To create a library, a folder with the library name must be present. For example if the library `greetings` was to be defined then a folder with the name `greetings` will be created. The folder must be located in the `bin/libraries` folder of the aurora installation directory. The `greetings` folder must contain an `config.json` file and any functions defined in the library in one or more files. The config file will contain configurations for each function, what file they're in, and what the call name is. The following might be an example of the `greeting` library.

```
{
  "bye": {
    "file": "farewells.aurora",
    "function": "bye"
  },
  "hi": {
    "file": "salutations.aurora",
```

(continues on next page)

(continued from previous page)

```
    "function": "hi"
  },
  "hello": {
    "file": "salutations.aurora",
    "function": "hello"
  }
}
```

The functions are usually in in alphabetical order. The structure could be defined as so:

```
{
"function_name": {
    "file": "filename.aurora",
    "function": "function_name_in_file"
}
}
```

The `salutations.aurora` file then might look like so:

```
include>io;

func: hello>String name;
    print>"Hello, ";
    print>name;
    println>"!";
end;

func: hi> => Void;
    println>"Hi there.";
end;
```

And the `farewells.aurora` file might look as follows:

```
include>io;

func: bye>String name;
    print>"Good bye, ";
    print>name;
    println>".";
end;
```

This will all create a library that can be called with `include>greetings;` and then the `hello`, `hi`, and `bye` functions can be called.

9.3 Community Libraries

The community can also put together libraries. To install on of these download the library and put it in the `bin/libraries` folder like you would if you were creating a library. Then you can use any of those functions in your aurora files after including them.

CHAPTER 10

I/O Library

The Input/Output library, or IO for short, is used for simple input and output functions. This is a built-in library included with the compiler and can be used by any Aurora program.

The following are the functions of the library and a description of how to use them.

10.1 IO In

10.1.1 Syntax

`io_in>location;.`

10.1.2 Usage

Used to access the IO ports. The returned value is the value in the location of "location".

10.2 IO Out

10.2.1 Syntax

`io_out>location, value;.`

10.2.2 Usage

Also used to access the IO ports. The location, "location" will be passed the value of "value".

CHAPTER 11

String Library

The String library is used for string manipulation. This is a built-in library included with the compiler and can be used by any Aurora program.

The following are the functions of the library and a description of how to use them.

11.1 ASCII

11.1.1 Syntax

`ascii>char;.`

11.1.2 Usage

Takes a `String` and returns the numerical ASCII value of the first character of the string.

11.2 Length

11.2.1 Syntax

`len>string;.`

11.2.2 Usage

Returns the length of the passed in string. To do this, the string must be null-terminated which automatically happens when defining strings in Aurora.