Improved String Support for the Amy Compiler

Computer Language Processing 2022 - Final Report

Suhas Shankar Edouard Michelin

EPFL

suhas.shankar@epfl.ch edouard.michelin@epfl.ch

1. Introduction

The Amy compiler consists of six stages transforming Amy source files into WebAssembly bytecode files. Amy is a simple functional programming language that could be simply described as a subset of Scala. The language originally offers basic programming language features like function definition, some primitive types (Int, Boolean, String, Unit), algebraic data types definition, control statements (IF-THEN-ELSE, MATCH-CASE (pattern matching)), basic binary operations ($+$, $-$, $<$, $>=$, etc...) and a standard built-in library of functions.

During the first part of this project, we had to implement the different phases of the compilation pipeline. We started by the first step which is the lexical analyzer (Lexer) which takes as input a source file and outputs its conversion in a list of tokens that will be passed to the parser. The parser takes as input the list of tokens produced by the lexer, parses it with an $LL(1)$ version of the Amy grammar and outputs a nominal Abstract Syntax Tree (AST), which concludes the syntactical analysis of an Amy program. The AST is then passed along to the provided Name Analyzer which either rejects an invalid programs that do not follow Amy's naming conventions or transforms the AST into a symbolic tree by generating unique identifiers for every name and populates the symbol table containing a mapping from identifiers to informations on the pointed object. Next stage we had to implement the Type Checker which first generates constraints on types in the program and secondly checks that every constraints is satisfied. This stage concludes the front-end of the compiler so at this point, the program is considered to be correct from a compiling perspective. Subsequently, we implemented the Code Generator which takes as input the AST produced by the front-end phases and outputs a list of instruction objects that the provided Code Printer will use to generate WebAssembly code.

Now we could argue that in its current state, the Amy programming language lacks a lot of features that are usually present in functional languages or that can be useful to almost every programmer like higher-order functions, tuples, the char type and a better support for operations on strings. Based on our personnal experience and a general observation of what programmers globally use and need, we chose to improve string support for Amy and add a new Char type (for characters) which will be useful for some of the built-in functions that we added.

2. Examples

In the introduction, we stated that the implemented extension directly aims to answer programmers' need in real world applications, so it is only normal that we demonstrates some of these real world use cases. The first example will demonstrate a simple case where the program asks for a user input which a certain format is expected for; let's say that a program named NamePrinter asks for the user's last name and prints it back to the console. Of course in a real application this would only be a tiny component that could for example be used in a registration form.

```
object NamePrinter

  fn capitalize(str : String) : String = {
    val length : Int(32) = StringImpl.length(str);
    if (length == 0) {
      ""
    } else {
      if (length == 1) {
        StringImpl.toUpperCase(str)
      } else {
        val firstLetter : Char = StringImpl.charAt(str, 0);
```

```
        val rest : String =
        StringImpl.substring(str, 1, length);
        val upper : String =
          StringImpl.toUpperCase(
            Std.charToString(firstLetter));
        upper ++ rest
      }
    }
  }

  Std.printString("What is your last name?");
  val lastName : String = Std.readString();
  val formatted : String = capitalize(
    StringImpl.strip(lastName));
  Std.printString(
    "Your last name is: '" ++ formatted ++ "'")

end NamePrinter
```

Here, a user running the program and giving input
`> " shankar  "`
(note the leading whitespace and trailing whitespaces) will read as output
`> "Your last name is: 'Shankar'".`

Another example that will this time make use of escape sequences. It is not rare that a programmer needs to make use of escape sequences, whether it is for characters that would otherwise be forbidden (i.e.: backslash, single quote) or for detecting/replacing a newline or for any other reason. However when an escape sequence is not handled, the behavior differs from language to language. As Amy is inspired by Scala, escaped characters that should not be escaped generate an error. The following code:

```
object UnauthorizedEscapeSequence
  // code...
  val noGood : Char = '\p'
  // code...
end UnauthorizedEscapeSequence
```

therefore produces the output:
`> Invalid escape character`

For our last example, we will demonstrate how to add a new line in a string. As there is currently no support for escape sequences for the `String` type, we will have to transform a literal `Char` to a `String` and concatenate them.

```
object StringNewLine
  Std.printString(
    "Hello " ++
    Std.charToString('\n') ++
    "World!")
end StringNewLine
```

3. Implementation

The implemented features can be grouped into two main sections. The first one being the addition of the character type and the second one being the new library of built-in functions for strings.

3.1 The Char type

3.1.1 Theoretical Background

The idea behind the implementation of the character type comes from the fact that a character actually is an integer that is translated to different symbols depending on the character set that is used.

3.1.2 Implementation Details

In order to add this new primitive type, roughly every stage of the pipeline had to be modified.

Lexer For the first phase of the compilation, we had to add a new token to be parsed by the lexical analyzer which we named $CharLitToken$. We also added "$Char$" to the list of primitive type names and added a rule in the lexer so that literal characters could be properly parsed. For literal characters declaration, we chose the usual delimiter for chars, i.e.: the single quote. The grammar is the following:

```
EXPR ::= Delimiter ~ CHAR ~ Delimiter
CHAR ::= CharLit |
    (EscapeChar ~ ESCAPE_SEQUENCE)
ESCAPE_SEQUENCE ::= EscapeChar |
                    Delimiter |
                    EscapeSequence
```

where $EscapeChar$, $Delimiter$ and $EscapeSequence$ are terminals. Which in silex syntax gives:

```
elem('\'') ~
(
  elem(c => c != '\\' && c != '\n' && c != '\'') |
  (elem('\\') ~ oneOf(Character.escapeCharacters))) ~
elem('\'')
```

This pattern allows to retrieve any character, including the single quote character, the backslash (escape char) and other escape sequence (if escaped). An escaped character that should not be escaped will generate an

"Invalid escape character" error, as it is the case in Scala.

Parser In this part we had to add two new node types for the AST which are $CharType$ and $CharLiteral$. The former being for type declarations while the latter is for character values themselves. For the parser itself, we just needed to add a rule to parse $CharLitToken$ from the list of tokens as a $CharLiteral$ in the AST, which contains informations about whether the character is escaped or not.

Name Analyzer Here we needed to add support for our new type in the type discovering and module reconstructions phases. The modifications were trivial as literal values have no name, therefore the changes only resulted in transforming a $CharType$ from the nominal tree to a $CharType$ in the symbolic tree, without affecting the symbolic table.

Type Checker For the last part of the front-end, we generated a constraint for char literals (which was obviously directly a top level constraint).

Code Generator In the presence of a $CharLiteral$, a value is returned depending on whether the character is escaped or not, which is then directly pushed on the stack as a 32-bit constant integer, in order to preserve compatibility with character sets that require more than 8 bits (and at most 32-bit) per value, such as UTF-8.

Notes In order to offer compatibility with different character sets, we modified (1) the `parseArgs` method in the entry file, (2) the `Context` case class in the utility (`utils`) folder and (3) the `run` method in the lexer so that (1) the command line parameter `--charset=CHARSET` is correctly parsed, (2) its value is registered in the context and (3) it is used as a parameter for the Silex method `Source.fromFile` that will be used to read the content of files that will then be passed along to the lexer. This way, the charset chosen by default by `Source.fromFile` no longer relies on the one used by the JVM. However, the re-encoding is done via a javascript function which uses UTF-16. The result might then no display properly in the console, depending on the input.
N.B.$_1$: the charset parameter defaults to UTF-8.
N.B.$_2$: the charset parameter applies to all files given as input.
N.B.$_3$: the charset parameter only takes values accepted by the Java Platform's implementation of the user, see [Oracle].

3.2 The Built-in Functions

3.2.1 Theoretical Background

String built-in functions are essential components of any developer's toolbox. The list of built-in functions that we wanted to implement was decided based on our own experience as programmers as well as data about commonly used string functions that we found online. An article in [Chr] stated that `std::string` accounted for half of all calls to the memory manager in Chromium. Thus, it becomes extremely important for us to keep runtime considerations when developing the functions.

3.2.2 Implementation Details

We implemented null-terminated strings. We also steered clear of any in-place operations and always returned a copy. The list of functions implemented can be found in Table: 1. We created a "StringImpl" library (anologous to `std::string`) that contains all string-related functions. The alternative to null-terminated strings were to treat strings as objects that contain the memory address as well as the length of the string. This would make taking slices more efficient. However, keeping track of all the variables pointing to a certain memory location was too cumbersome.

There were two main ways of writing the built-in functions.

1. Implement the function in the jsWrapper object in Module.scala
2. Write the webassembly code for the functions in Utils.scala

Simple functions that didn't need to write to the memoryBuffer (our heap) could be implemented through route 1. Examples include `length` and `charAt`. However, access to the current memory boundary can be done only through wasm instructions `GetGlobal(memoryBoundary)`. Hence, functions that needed to write to memory had to be implemented via route 2. Examples include `strip`, `substring` and `toLowerCase`.

Code Generator The only phase that was touched to implement the built-in functions was the codegen phase.

Functions implemented	
Function Signature	**Description**
`length(input: String): Int(32)`	Returns the length of the String
`equals(s1: String, s2: String): Boolean`	Returns if the two strings are equal
`charAt(input: String, index: Int(32)): Char`	Returns the character at the specified index. All strings are 0-indexed. Returns '$' if the index is not in bounds
`substring(input: String, start: Int(32), end: Int(32)): String`	Returns a copy of the substring from indices start to end
`strip(input: String): String`	Returns a copy of input with whitespaces trimmed (start and end)
`toLowerCase(input: String): String`	Returns a copy of input with letters converted to lowercase
`toUpperCase(input: String): String`	Returns a copy of input with letters converted to uppercase
`replace(input: String, c1: Char, c2: Char): String`	Returns a copy of input with 'c1' replaced by 'c2'
`indexOf(input: String, search: Char): Int(32)`	Returns the first index at which 'search' occurs. Returns -1 if it doesn't exist
`printChar(input: Char): Unit`	Prints the character to output
`charToString(input: Char): String`	Converts a character to a string

Table 1. String built-in functions

4. Possible Extensions

4.1 Optimizations for multi-concatenation (String Builder - String Interpolation)

Another task that programmers often do is concatenating multiple strings. For instance, displaying informations on users on a profile page, which could be of the form:

```
Hello {firstname} {lastname},
you have {friendsCount} friends
and {followersCount} followers.
It is {time.hours}:{time.minutes},
have a great day!.
```

It is important to provide programmers with easy and efficient ways to concatenate multiple strings (literal or not) with each other.

4.2 ToCharArray, Split

Being able to split a string into a list/array of strings/chars can be very important when it comes to, for instance, parsing an input. As it was a primary concern throughout our project, we obviously would have liked to implement such feature. However we would need arrays/lists of strings/chars in order to proceed. Whereas this would also be an interesting and very useful addition, it falls out of the scope of the project we chose to assess.

4.3 Extend escape sequence support to String

As seen in the examples, the only way to add an escape sequence (e.g.: new-line) in a string is to concatenate the string with a char transformed to a string via the `Std.charToString` function. The same process is required to have a string that contains a double quote character (the string delimiter). This is not practical and it would as such be a useful addition to implement support for strings.

4.4 Multi Line Stings

Following the extension stated above, support for multi-line strings would also be a useful feature to add.

References

Chromium developer forum. `http://bit.ly/chromium-dev`.

Oracle. Charset - oracle documentation. `http://bit.ly/oracle-doc-charset`.