LangLang Developer Manual

George Tucker - 27535592 - gkjt1g15@soton.ac.uk

April 29, 2017

1 Introduction

LangLang is a small, imperative, weakly typed language for handling basic manipulations of regular languages. It is very easy to learn, with only 3 primitive types and 9 command symbols. Input is read on stdin before the program is executed, where it can then be accessed like a buffer, more in section 2.3.

2 The Basics

2.1 Hello World

So we are very proud to say that LangLang is capable of outputting strings to Standard Output. Yes, a great feat, we know, we know! It works as you would expect, as shown in the following code snippet:

print "hello";

So lets break this down:

The most familiar part of this command is the simple string. In LangLang, strings are surrounded by double quotes, and contain lowercase English characters only (so no spaces at the moment).

The print command comes in two flavours, with one argument or with two. In this case we are using 1 argument, which will just print out all of the argument, whatever it is. It will take any of our primitive types, or you can pass it a variable name and it will print out the value of that variable. See the reference for the print command in section 3.1.

Finally, you will notice the semicolon on the end of the line, which is rather important in LangLang. Every statement must be terminated by a semicolon, and not using one will cause a Syntax error on the beginning of the next statement (*which may be an empty token in some cases!*).

2.2 Language Manipulation

This is LangLang's *raison d'etre*, regular language manipulation. LangLang provides several domain specific functions for working with regular languages, detailed in section 3. For this section I will use the following code:

```
a = {"h", "e", "l", "l", "o"};
b = {"w", "o", "r", "l", "d"};
print ((a+b) U a) 5;
```

Outputs: ed, el, eo, er, ew

This snippet demonstrates variable assignment, instantiating languages within a program. Assignment takes a variable name and binds a value to it, in this case a language, but could be a string, integer, or the output of a call to a built in function. We then use the concatenate operator on a and b, then take the union of the result of that function with a, and print the result, limiting the output to the first 5 elements. More about each function can be seen in section 3, but here you can see how easy it is to store data, perform operations, and output. Here we use the brackets to define evaluation order, as union binds more strongly than concatenation.

2.3 Using Input

LangLang reads input from STDIN before running the programme, and must get some input (what use is the program if it isn't using some input, right?). Input is separated by newlines and each line can either be an integer or a language. Languages are surrounded by curly braces and contain groups of lowercase English letters separated by a comma, with spaces around these commas ignored. These are read into a single buffer, where they can be read with *readint* and *readlang*. These return the next item in the buffer, so you must be aware of the order in which you call these and the number of calls you make to these functions.

2.4 Errors

LangLang has a very simple error system, syntax errors and type errors.

Syntax errors are for when you type something in that is not a valid LangLang token, and can occur in both the input and the program. The error message for a syntax error contains the line and column number for where the token that cause the error was, and the token that caused the error. Note that the token may be a newline or absence of a different token (ie. End of file before a semicolon on a previous statement), however it will still tell you where it is so you can easily see the issue. Syntax errors occur during lexing and parsing of the program, so they are always the first to be thrown. If the program and the input both pass the lexer and parser without issue the program moves on to the type checker.

```
print +;
```

Outputs:

```
Syntax error on token "+" (1, 7) of program.spl
```

Type errors occur when the expression is made up of valid tokens, but one or more of the tokens are the wrong type for the function. In this case a message describing how the offending function should be used is provided, so you can find the problem and fix your program. Type checking occurs between parsing of the program and executing the program, and is only run on the program, so errors due to calling read functions in the wrong order are not caught.

```
print 1 + "b";
```

Outputs:

```
Type Error: Concatenate must be between Languages, Strings, or both
```

3 Builtin Functions

LangLang provides a number of built in functions, detailed below. Many of these functions can be nested without brackets, although we recommend using brackets to keep the meaning of the program clear.

3.1 Print

```
print OBJECT (optional LIMIT) -> unit
```

Prints a representation of OBJECT to STDOUT. If OBJECT is a language, LIMIT can be used to print only a number of elements from the language. This does not affect the underlying data. All print calls end with a newline.

See section 2.1 for an example of using print.

3.2 Union

```
LANG U LANG -> language
```

Return the union of two languages

3.3 Intersection

```
LANG I LANG -> language
```

Returns the intersection of two languages. Intersection is left and right associative, so calls can be chained into one line with no issue.

3.4 Concatenation

```
FIRST + SECOND -> language
```

Concatenates every item of the language FIRST with every item of SECOND. Either parameter may be a string, which is treated as a single word set. This operation is not commutative, but is associative and may be safely chained.

3.5 Powerset

```
LANG ^ LIMIT -> language
```

Get the first LIMIT elements of the powerset of LANG. In practice, this is just the powerset of the first element of LANG.

```
print {"a", "b", "c"} ^ 2;
```

Outputs: $\{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$

3.6 Kleene's Star

```
ALPHABET * LENGTH -> language
```

Returns the set of elements of length LENGTH from the language over the set of symbols ALPHABET, which is also a language.

```
print {"a", "b", "c"} * 5;
```

Outputs: $\{:, a, aa, aaa, aaaa\}$

All elements of ALPHABET must be single characters, passing anything longer will cause it to only use the first character of each element in all but the first non empty element.

3.7 Read Integer

```
readint -> integer
```

Gets the next item from the input buffer if it is an integer, else will throw an error.

3.8 Read Language

```
readlang -> language
```

Gets the next item from the input buffer if it is a language, else will throw an error.