Parser Combinators

# Overview

All parsers take the same basic format. They are first initialised with a value, for example the character to find.

The parse method is then called with the text to match against. The parse method then returns

* Was the string successfully parsed?
* What was passed?
* What unparsed bits of the string remain?

# Base Parsers

First you will create some base parsers on which we can build all the other parsers.

## StartsWithChar

Write a parser which checks to see if a string starts with a given character.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Initialisation** | **Input** | **Result** | **Parsed** | **Remaining** |
| A | ABC | Success | A | BC |
| A | AAAA | Success | A | AAA |
| A | BCD | False | - | BCD |
| A |  | False | - |  |

## StartsWithAnyChar

Write a parser which checks to see if a string starts with any of the given characters.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Initialisation** | **Input** | **Result** | **Parsed** | **Remaining** |
| A,B | ABC | Success | A | BC |
| A,B | BCD | Success | B | CD |
| A,B | CDEF | False | - | CDEF |

Use this parser to create an **isLowerCaseCharacter** parser which checks if a string starts with a lower case letter.

Use this parser to create an **isUpperCaseCharacter** parser which checks if a string starts with an upper case letter.

Use this parser to create an **isDigit** parser which checks if a string starts with a numeric digit.

# Compound Parsers

We can now combine the base parsers to make more interesting parsers

## Either

Write a parser which is initialised with 2 existing parsers. The parser returns true if either parser matches the text.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Initialisation** | **Input** | **Result** | **Parsed** | **Remaining** |
| StartsWithChar A, StartsWithChar B | ABC | Success | A | BC |
| StartsWithChar A, StartsWithChar B | BAC | Success | B | AC |
| StartsWithChar A, StartsWithChar B | CBA | False | - | CBA |

Use this parser to create an **isCharacter** parser from your **isLowerCaseCharacter** and **isUpperCaseCharacter** parsers

If your language supports operating overloading, then try to redefine this function as |. I.e.

**isCharacter** = **isLowerCaseCharacter | isUpperCaseCharacter**

## Many\*

Write a parser which repeatedly applies a parsers to an input string until the parser returns false.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Initialisation** | **Input** | **Result** | **Parsed** | **Remaining** |
| StartsWithChar A | AAAB | Success | AAA | B |
| StartsWithChar A | CAAAB | True |  | CAAAB |
| StartsWithChar A | StartsWithChar B | ABABAABBC | True | ABABAABB | C |

Use this parser to create an **IsInteger** parser*. (i.e. 1234.12 would be parsed as 1234)*

## OneOrMany+

Similar to the many parser, but the parser must find at least one match.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Initialisation** | **Input** | **Result** | **Parsed** | **Remaining** |
| StartsWithChar A | AAAB | Success | AAA | B |
| StartsWithChar A | CAAAB | False | - | CAAAB |

## Then

Write a parser which is initialised with 2 parsers. These parsers are applied in order and the result is only true if both parses succeed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Initialisation** | **Input** | **Result** | **Parsed** | **Remaining** |
| StartsWithChar A,  StartsWithChar B | ABA | Success | AB | A |
| StartsWithChar A,  StartsWithChar B | AC | False | - | AC |
| StartsWithChar A,  StartsWithChar B | BA | False | - | BA |

Use this parser to create an **IsDecimal** parser. *. (i.e. 1234.12 would be parsed as 1234.12)*

If your language supports operating overloading, then try to redefine this function as +. I.e.

**isDecimal** = **isInteger + isPeriod + isInteger**

Amend your parser so that it caters for both 1234.45 and 1234

# Exercise 1

Use your parsers to parse the following string

**Print ("Hello World");**

There can be optional whitespace (spaces, tabs) between the print and open bracket.

(To identify **Print** You may find it easy to create a new base parser called **StartsWithText**)

# Exercise 2

A create a parser which matches **Out** but not **Output**

There are a few possible solutions, I would however avoid implementing an “And” parser.

# Exercise 3

At this point we can now parse a piece of text but it would be more useful if we could extract values from it.

For example from the string **Print ("Hello Word");** we would want to know that the Print command is being called, with a parameter of “Hello World”. Often we would not be interested in the trivial such as the whitespace, brackets and semi-colon.

Action - Extend the definition of your parser so that a mapping function can be applied to the parsed value.

Then the Print parser would function as follows

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Result** | **Parsed** | **Remaining** |
| Print ("Hello Word"); | Success | A print command object, which is passed “Hello World” to its constructor |  |

In C# the code would look along the following lines

class PrintCommand : Command

{

private string printWhat;

public PrintCommand(string printWhat)

{

this.printWhat = printWhat;

}

}

var printCommand = (print + whitespace + openBracket + stringLit + closeBracket).Apply((\_,\_,\_,d,\_) => new PrintCommand(d));

# Exercise 4 (Optional)

Write a parser to identify variable declarations in the syntax of a language of your choice. Use the mapping function to populate a “declaration object” with name, type and value properties.

For example:

Dim name As String = “David”

int age = 40;

# Exercise 5

At this point we are now able to parse some text and then turn the recognised values into an abstract syntax tree (AST). The final step is now to walk down this tree to generate some code.

Write parsers to recognise the following Domain-specific language (DSL)

* Move Up *distance*
* Move Down *distance*
* Move Left *distance*
* Move Right *distance*

Map the results of these parsers into MoveUp, MoveDown, MoveLeft and MoveRight objects

Walk the generated AST to generate commands for drawing onto the screen in a language of your choice

In C# this might look like

if (cmd is MoveDown)

{

var command = (MoveDown)cmd;

output.AppendLine($"newX = X;");

output.AppendLine($"newY = Y + {command.Distance};");

output.AppendLine($"g.DrawLine(Pens.Blue, X, Y, newX, newY);");

output.AppendLine($"X = newX;");

output.AppendLine($"Y = newY;");

}

My program

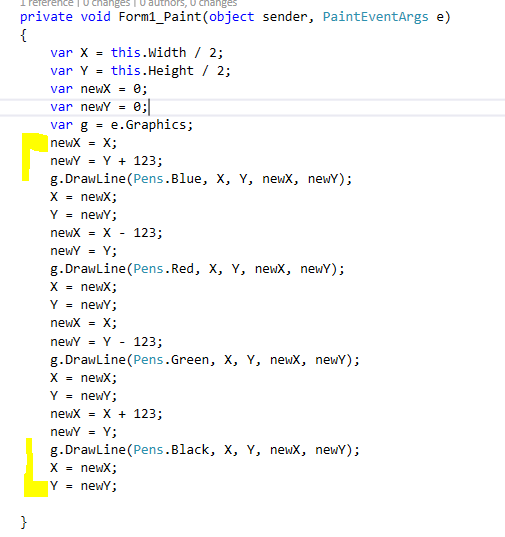
Move Down 123

Move Left 123

Move Up 123

Move Right 123

Then generated the following C# code



Which produced

