Introduction

SPL (Stream Processing Language) is a programming language specifically designed for performing operations on potentially unbounded sequences of data.

Structure

SPL takes an input, process, output approach to stream processing which is encapsulated by the `whileInput` block. This loops while an input stream is present and allows the program to take in an input, perform some operations on it and produce an output.

It's for this reason that programs written in SPL employ the following structure:

```
definitions
whileInput {
  operations
}
```

All single line statements are closed with a `;` and you can write a single line comment using `--`.

Variables

Variables are defined by providing an identifier followed by an integer value and a semi colon.

```
identifier = value : int;
```

They are bound throughout program execution and so their value can be updated inside the `whileInput` loop.

> It is possible to define variables within the `whileInput` loop but this is not conventional as it reduces readability. That said, defining variables inside means that their values are set every time it loops, which may be desirable for certain applications.

Input

To read each element of our input streams we use:

```
input[ index : int ]
```

Multiple Inputs

SPL supports multiple input streams so the index is used to determine which stream we are reading from - indexing starts from 0. For example: `input [0]` will fetch the next element of the first input stream.

Process

Mathematical Operators

SPL provides a wide range of mathematical operators to process the input

Operator	Description	Example
`+`	Addition	2 + 2 = 4
`-`	Subtraction	2 - 2 = 0
`*`	Mutliplication	2 * 2 = 4
`/`	*Integer* Division	4 / 3 = 1
`%`	Modulo	8 % 3 = 2

> Brackets can also be used to group arithmetic expression to make their evaluation order explicit. For example `(1+2)*3 = 9` while `1+2*3 = 7`

Conditional Statements

SPL also provides an if statement:

```
if (condition) {
} else {
}
```

If the condition evaluates to true the code between the first curly braces will run otherwise the code between the second curly braces is executed. This enables conditional behaviour based on the value of the input.

Comparators

The condition will evaluate to true or false depending on the following comparators

Operator	Description	Example
`==`	Equal to	`1 == 1 = true`, `1 == 0 = false`
`<`	Less than	`0 < 1 = true`, `1 < 0 = false`
`>`	Greater than	`1 > 0 = true`, `0 > 1 = false`
`<=`	Less than or Equal to	`1 <= 1 = true`, `1 <= 0 = false`
`>=`	Greater than or Equal to	`1 >= 1 = true`, `0 >= 1 = false`
`!=`	Not Equal to	`1 != 0 = true`, `1 != 1 = false`

Output

In order to output to standard out we use:

```
output( statement : int )
```

The statement inside the braces can be anything of type integer, such as: an integer itself - `output(1)`, a variable - `output(var)` or a calculation - `output(1 + 2)` or `output(var1 + var2)`

> There should be one output statement per program within the `whileInput` loop. This means for each input an output will be produced

Multiple Outputs

We can also output multiple values (therefore creating multiple streams). To do this simply comma seperate statements inside the braces. For example `output(1, 2)`

Type Checking

SPL employs a type checker that ensures statements are well typed. All arithmetic and comparator operations are checked to ensure both arguments are integers. The input statements stream index and any value passed to the output function must also be integers.

Error Messages

If any of the type checks are violated a type error will display which includes the name of the operation that threw the error as well as it's position within the program. There are also some more specific error messages such as:

- When the input stream index is larger than number of streams provided
- When an identifier is used when the variable has not been defined

Acknowledgements

In the creation of SPL we drew inspiration from Python and Java. We took the variable definitions from Python, the line endings from Java and the rest of the syntax was constructed from a combination of the two. Including the use of standard braces for the output statement, as you are expected to provide a list of arguments, while the input method uses square braces as it expects an index. This makes SPL intuitive to learn if you have programming experience.

Appendix

Problem Solutions

1

```
x = 0;
whileInput {
  output(x);
  x = input[0];
}
```

2

```
whileInput {
  output(input[0], input[0]);
}
```

3

```
whileInput {
  output(input[0] + 3 * input[1]);
}
```

4

```
t = 0;
whileInput {
  t = t + input[0];
  output(t);
}
```

5

```
p1 = 0;
p2 = 0;
v = 0;
whileInput {
  v = p1 + p2 + input[0];
  output(v);
  p2 = p1;
  p1 = v;
}
```

6

```
prev = 0;
whileInput {
  output(input[0], prev);
  prev = input[0];
}
```

7

```
whileInput {
  output(input[0] - input[1], input[0]);
}
```

8

```
prev = 0;
whileInput {
  output(input[0] + prev);
  prev = input[0];
}
```

9

```
prev = 0;
add = 0;
whileInput {
  output(prev + add + input[0]);
  prev = prev + add + input[0];
  add = add + input[0];
}
```

10

```
p1 = 0;
p2 = 0;
v = 0;
whileInput {
  v = p2 + input[0];
  output(v);
  p2 = p1;
  p1 = v;
}
```

Syntax Highlighting

Iro

In order to implement syntax highlighting for SPL we used a tool called Iro (https://eeyo.io/iro/). It is a markup language that allows you to generate sytnax highlighting grammars for all mainstream text editors and IDEs.

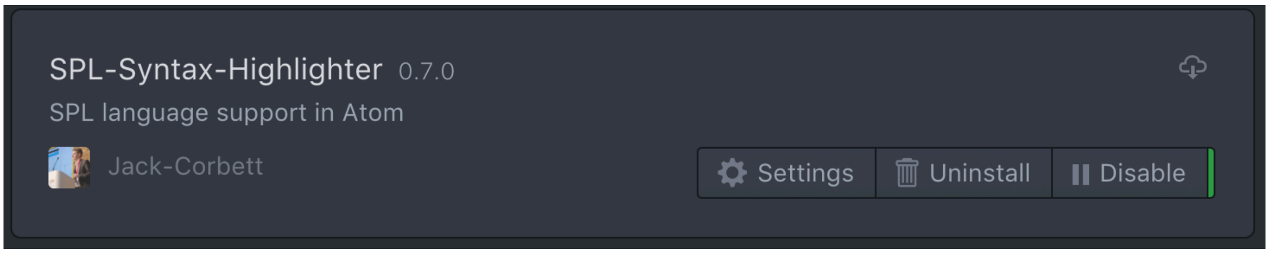
```
#################################################################
## Iro
################################################################
##
## * Press Ctrl + '+'/'-' To Zoom in
## * Press Ctrl + S to save and recalculate...
## * Documents are saved to web storage.
## * Only one save slot supported.
## * Matches cannot span lines.
## * Unicode chars must be defined in \u0000 to \uffff format.
## * All matches must be contained by a single group ( ... )
## * Look behinds not permitted, (?<= or (?<!
## * Look forwards are permitted (?= or (?!
## * Constants are defined as __my_const = (......)
## * The \= format allows unescaped regular expressions
## * Constants referenced by match \= $${__my_const}
## * Constants can reference other constants
## * You are free to delete all the default scopes.
## * Twitter : ainslec , Web: http://eeyo.io/iro
##
################################################################

name                   = spl
file_extensions []     = spl;

################################################################
## Constants
################################################################

__MY_CONSTANT \= (\b[a-z][a-zA-Z0-9]*)

################################################################
## Styles
################################################################

styles [] {

.comment : style {
   color                 = grey
   italic                = true
   ace_scope             = comment
   textmate_scope        = comment
   pygments_scope        = Comment
}

.keyword : style {
   color                 = cyan
   ace_scope             = keyword
   textmate_scope        = keyword
   pygments_scope        = Keyword
}

.method : style {
   color                 = light_blue
   ace_scope             = entity.name.type
   textmate_scope        = entity.name.type
   pygments_scope        = Operator
}

.structure : style {
   color                = violet_red
   ace_scope            = entity.name.function
   textmate_scope       = entity.name.function
   pygments_scope       = Generic
}

.boolean : style {
   color                = light_green
   ace_scope            = string
   textmate_scope       = string
   pygments_scope       = Literal
}

.numeric : style {
   color                 = gold
   ace_scope             = constant.numeric
   textmate_scope        = constant.numeric
   pygments_scope        = Number
}

.punctuation : style {
   ace_scope             = punctuation
   textmate_scope        = punctuation
   pygments_scope        = Punctuation
}

}

#################################################
## Parse contexts
```

```
##################################################

contexts [] {

###############################################
## Main Context - Entry point context
###############################################

main : context {

   : include "numeric" ;

   : inline_push {
      regex          \= (\{)
      styles []       = .punctuation;
      : pop {
         regex       \= (\})
         styles []    = .punctuation;
      }
      : include "main" ;
   }

   : pattern {
      regex          \= (;)
      styles []       = .punctuation;
   }

   : inline_push {
      regex          \= (\()
      styles []       = .punctuation;
      : pop {
         regex       \= (\))
         styles []    = .punctuation;
      }
      : include "main" ;
      : pattern {
         regex       \= (,)
         styles []    = .punctuation;
      }
   }

   : pattern {
      regex          \= (input|output)
      styles []       = .method;
   }

   : pattern {
      regex          \= (whileInput|if|else)
      styles []       = .structure;
```

```
    }

    : pattern {
       regex          \= (true|false)
       styles []       = .boolean;
    }


    : pattern {
       regex          \= $${__MY_CONSTANT}
       styles []       = .keyword;
    }

    : pattern {
       regex          \= (--.*)
       styles []       = .comment;
    }
}

#################################################
## End of Contexts
#################################################

###########################################
## Numeric Context
###########################################

numeric : context {
    : pattern {
       regex          \= (\b\d+)
       styles []       = .numeric;
    }
}
}
```

We decided to use this to write a syntax highlighting plugin for Atom.

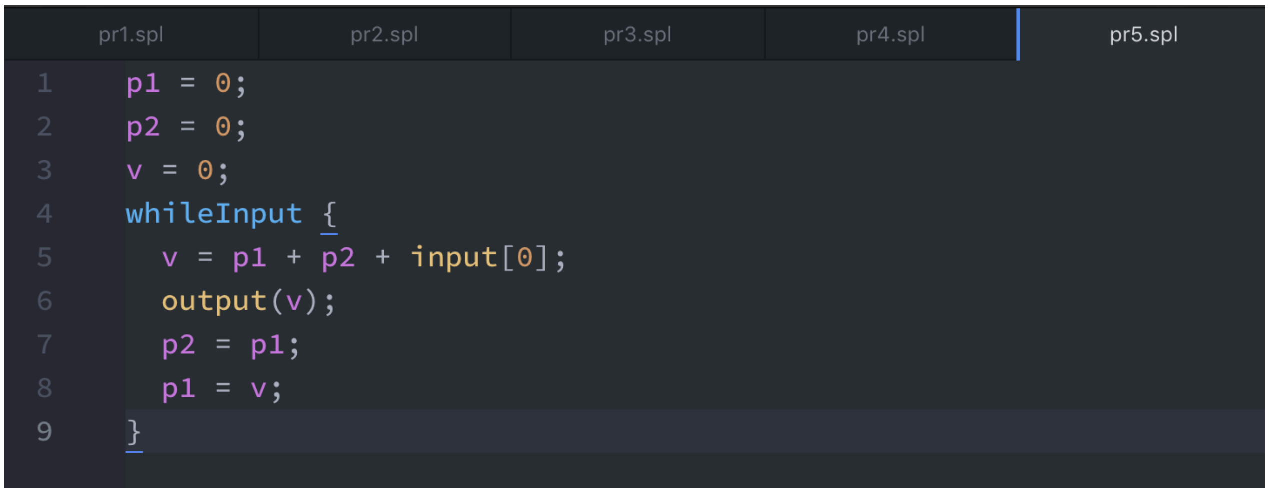
Atom


SPL-Syntax-Highlighter 0.7.0

SPL language support in Atom

Jack-Corbett

You can install the Atom package here: https://atom.io/packages/spl-syntax-highlighter which will automatically highlight any `.spl` files.



```
p1 = 0;
p2 = 0;
v = 0;
whileInput {
  v = p1 + p2 + input[0];
  output(v);
  p2 = p1;
  p1 = v;
}
```

```
'fileTypes' : [
  'spl'
]
'name' : 'SPL'
'patterns' : [
  {
    'include' : '#main'
  }
]
'scopeName' : 'source.spl'
'uuid' : ''
'repository' : {
  'main' : {
    'patterns' : [
      {
        'include' : '#numeric'
      }
      {
        'begin' : '(\\{)'
        'beginCaptures' : {
          '1' : {
            'name' : 'punctuation.spl'
          }
        }
        'patterns' : [
          {
            'include' : '#main__1'
          }
        ]
        'end' : '(\\})'
        'endCaptures' : {
          '1' : {
            'name' : 'punctuation.spl'
          }
        }
      }
      {
        'match' : '(;)'
        'name' : 'punctuation.spl'
      }
      {
        'begin' : '(\\()'
        'beginCaptures' : {
          '1' : {
            'name' : 'punctuation.spl'
          }
        }
        'patterns' : [
          {
            'include' : '#main__2'
          }
        ]
        'end' : '(\\))'
        'endCaptures' : {
          '1' : {
            'name' : 'punctuation.spl'
          }
        }
      }
      {
        'match' : '(input|output)'
        'name' : 'entity.name.type.spl'
      }
      {
        'match' : '(whileInput|if|else)'
        'name' : 'entity.name.function.spl'
      }
      {
        'match' : '(true|false)'
        'name' : 'string.spl'
      }
      {
        'match' : '(\\b[a-z][a-zA-Z0-9]*)'
        'name' : 'keyword.spl'
      }
      {
        'match' : '(--.*)'
        'name' : 'comment.spl'
      }
    ]
  }
  'main__1' : {
    'patterns' : [
      {
        'include' : '#main'
      }
    ]
  }
  'main__2' : {
    'patterns' : [
      {
        'include' : '#main'
      }
      {
        'match' : '(,)'
        'name' : 'punctuation.spl'
      }
    ]
  }
  'numeric' : {
    'patterns' : [
      {
        'match' : '(\\b\\d+)'
        'name' : 'constant.numeric.spl'
      }
    ]
  }
}
```