SICPy Specifications

Khooi Xin Zhe, Ang Jun Jie

19 April, 2021

SICPy §0

module	::=	*statement* …	statement sequence
statement	::=	*expression*	expression statement
expression	::=	*number*	primitive number expression
	|	`True` | `False`	primitive boolean expression
	|	*string*	primitive string expression
	|	*expression binary-operator expression*	binary operator combination
	|	*unary-operator expression*	unary operator combination
	|	`(` *expression* `)`	parenthesised expression
binary-operator	::=	`+` | `-` | `*` | `/` | `%` | `==`	
	|	`>` | `<` | `>=` | `<=` | `and` | `or`	
unary-operator	::=	`not` | `+` | `-`	

SICPy §1

module	::=	*statement* ...	statement sequence
statement	::=	*name* `=` *expression*	assignment statement
	|	*function*	function declaration
	|	`return` *expression*	return statement
	|	*if-statement*	conditional statement
	|	*expression*	expression statement
function	::=	`def` *name* `(` *parameters* `)` `:`	
		statement ...	function declaration
parameters	::=	ϵ | *name* [`,` *name*] ...	function parameters
if-statement	::=	`if` *expression* `:`	
		statement ...	
	[[	`elif` *expression* `:`	
		statement ...] ...	
		`else` `:`	
		statement ...]	conditional statement
expression	::=	*number*	primitive number expression
	|	`True` | `False`	primitive boolean expression
	|	*string*	primitive string expression
	|	*name*	name expression
	|	*expression binary-operator expression*	binary operator combination
	|	*unary-operator expression*	unary operator combination
	|	*expression* `(` *expressions* `)`	function application
	|	`lambda` *name* [`,` *name*] ... `:` *expression*	lambda expression
	|	*expression* `if` *expression* `else` *expression*	conditional expression
	|	`(` *expression* `)`	parenthesised expression
binary-operator	::=	`+` | `-` | `*` | `/` | `%` | `==`	
	|	`>` | `<` | `>=` | `<=` | `and` | `or`	
unary-operator	::=	`not` | `+` | `-`	
expressions	::=	ϵ | *expression* [`,` *expression*] ...	argument expressions

SICPy §2

module	::=	*statement* …	statement sequence
statement	::=	*name* `=` *expression*	assignment statement
	|	*function*	function declaration
	|	`return` *expression*	return statement
	|	*if-statement*	conditional statement
	|	*expression*	expression statement
function	::=	`def` *name* `(` *parameters* `)` `:`	
		statement …	function declaration
parameters	::=	ϵ | *name* [`,` *name*] …	function parameters
if-statement	::=	`if` *expression* `:`	
		statement …	
	[[	`elif` *expression* `:`	
		statement …] …	
		`else` `:`	
		statement …]	conditional statement
expression	::=	*number*	primitive number expression
	|	`True` | `False`	primitive boolean expression
	|	`None`	primitive list expression
	|	*string*	primitive string expression
	|	*name*	name expression
	|	*expression binary-operator expression*	binary operator combination
	|	*unary-operator expression*	unary operator combination
	|	*expression* `(` *expressions* `)`	function application
	|	`lambda` *name* [`,` *name*] … `:` *expression*	lambda expression
	|	*expression* `if` *expression* `else` *expression*	conditional expression
	|	`(` *expression* `)`	parenthesised expression
binary-operator	::=	`+` | `-` | `*` | `/` | `%` | `==`	
	|	`>` | `<` | `>=` | `<=` | `and` | `or`	
unary-operator	::=	`not` | `+` | `-`	
expressions	::=	ϵ | *expression* [`,` *expression*] …	argument expressions

SICPy §3

```
           module  ::=  statement ...                                    statement sequence
        statement  ::=  name = expression                                assignment statement
                     |  function                                         function declaration
                     |  return expression                                return statement
                     |  if-statement                                     conditional statement
                     |  while-statement                                  while statement
                     |  for-statement                                    for statement
                     |  expression                                       expression statement
                     |  break | pass | continue
         function  ::=  def name ( parameters ) :
                          statement ...                                  function declaration
       parameters  ::=  ϵ | name [ , name ] ...                          function parameters
     if-statement  ::=  if expression :
                          statement ...
                   [[   elif expression :
                          statement ...  ] ...
                        else :
                          statement ...  ]                               conditional statement
  while-statement  ::=  while expression :
                          statement ...                                  while statement
    for-statement  ::=  for expression in expression :
                          statement ...                                  for statement
       expression  ::=  number                                           primitive number expression
                     |  True | False                                     primitive boolean expression
                     |  None                                             primitive list expression
                     |  string                                           primitive string expression
                     |  name                                             name expression
                     |  expression binary-operator expression            binary operator combination
                     |  unary-operator expression                        unary operator combination
                     |  expression ( expressions )                       function application
                     |  lambda name [ , name ] ... : expression          lambda expression
                     |  expression if expression else expression         conditional expression
                     |  list-expression                                  list expression
                     |  expression [ expression ]                        list access
                     |  ( expression )                                   parenthesised expression
  list-expression  ::=  [ expressions ]                                  literal list expression
                     |  [ expression for expression in expression [ if expression ] ]  list comprehension expression
  binary-operator  ::=  + | - | * | / | % | ==
                     |  > | < | >= | <= | and | or
   unary-operator  ::=  not | + | -
      expressions  ::=  ϵ | expression [ , expression ] ...              argument expressions
```

SICPy §4

module	::=	*statement* …	statement sequence
statement	::=	*name* = *expression*	single assignment
	|	*name* [, *name*] … = *expression*	tuple assignment
	|	*function*	function declaration
	|	`return` *expression*	return statement
	|	*if-statement*	conditional statement
	|	*while-statement*	while statement
	|	*for-statement*	for statement
	|	*try-statement*	try statement
	|	*expression*	expression statement
	|	`break` | `pass` | `continue`	
function	::=	`def` *name* (*parameters*) :	
		statement …	function declaration
parameters	::=	ϵ | *name* [, *name*] …	function parameters
if-statement	::=	`if` *expression* :	
		statement …	
	[[	`elif` *expression* :	
		statement …] …	
		`else` :	
		statement …]	conditional statement
while-statement	::=	`while` *expression* :	
		statement …	while statement
for-statement	::=	`for` *expression* `in` *expression* :	
		statement …	for statement
try-statement	::=	`try` :	
		statement …	
		`except` *expression* :	
		statement …	
	[	`except` *expression* :	
		statement …] …	try statement
expression	::=	*number*	primitive number expression
	|	`True` | `False`	primitive boolean expression
	|	`None`	primitive list expression
	|	*string*	primitive string expression
	|	*name*	name expression
	|	*expression binary-operator expression*	binary operator combination
	|	*unary-operator expression*	unary operator combination
	|	*expression* (*expressions*)	function application
	|	`lambda` *name* [, *name*] … : *expression*	lambda expression
	|	*expression* `if` *expression* `else` *expression*	conditional expression
	|	*list-expression*	list expression
	|	{ *expression* : *expression* [, *expression* : *expression*] … }	literal dict expression
	|	(*tuple-expression*)	tuple expression
	|	*expression* `[` *expression* `]`	list/ dictionary access
	|	(*expression*)	parenthesised expression

list-expression	::=	`[` *expressions* `]`	literal list expression
	$\mid$	`[` *expression* `for` *expression* `in` *expression* [`if` *expression*] `]`	list comprehension expression
tuple-expression	::=	ϵ $\mid$ *expression* `,` *expressions*	*tuple expression*
binary-operator	::=	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `%` $\mid$ `==`	
	$\mid$	`>` $\mid$ `<` $\mid$ `>=` $\mid$ `<=` $\mid$ `and` $\mid$ `or`	
unary-operator	::=	`not` $\mid$ `+` $\mid$ `-`	
expressions	::=	ϵ $\mid$ *expression* [`,` *expression*] $\ldots$	argument expressions