SICPy §4

Khooi Xin Zhe, Ang Jun Jie

19 April, 2021

A SICPy[1] program is a *module*, defined using Backus-Naur Form[2] as follows:

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

[1] SICPy is an adaptation of Source - the official language of the textbook Structure and Interpretation of Computer Programs, JavaScript Adaptation.

[2] We adopt Henry Ledgard's BNF variant that he described in A human engineered variant of BNF, ACM SIGPLAN Notices, Volume 15 Issue 10, October 1980, Pages 57-62. In our grammars, we bold and underline keywords, [ ] for optional syntaxes, italics for syntactic variables, ϵ for nothing, $x|y$ for x or y, and x... for zero or more repetitions of x.

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