INCA-Lang

Идеята ни беше за имплементация на простичък, интерпретиран език с поддръжка на ламбда изрази и рекурсия, който може да се използва за по-ясно и кратко описание на базови алгоритми.

*INCA* на пръв поглед изглежда като JavaScript, но е различен. Първо, няма statement-и, само изрази. Един израз връща стойност и може да се използва вместо всеки друг израз.

Функциите се въвеждат с ключовата дума func. След ключовата дума трябва да има списък с имена на променливи, разделени със запетаи, както в JavaScript - това са имената на аргументите. Тялото на функцията е единичен израз, но може да бъде последователност, опакована в {…}. Няма return (няма statement-и) - последният израз, оценяван във функция, дава стойността, която се връща на повикващия.

Няма var, използваме func и декларираме променливите като аргументи. Не се изискват скоби около условието на if-а, но не е грешка, ако ги добавим.

Цялата програма се анализира така, сякаш е вградена във скоби, затова трябва да поставяме точка и запетая след всеки израз. Последният израз може да бъде изключение.

Oписание на input stream-a

Това е най-малката част. Ще създадем stream обект, който осигурява операции за четене на символи от низ. Имаме 4 метода:

peek() — връща следващата стойност без да я премахва от потока.

next() — връща следващата стойност като я премахва от потока.

eof() — връща true ако няма повече стойности в потока.

croak(msg) — връща съобщение за грешка.

Описание на AST

Парсерът ще изгради структура, която точно представя семантиката на програмата. AST е обикновен JavaScript обект, който има свойство type, определящо какъв тип възел е той, и допълнителна информация, в зависимост от конкретния тип.

Накратко:

num { type: "num", value: NUMBER }

str { type: "str", value: STRING }

bool { type: "bool", value: true or false }

var { type: "var", value: NAME }

func { type: "func", vars: [ NAME... ], body: AST }

call { type: "call", func: AST, args: [ AST... ] }

if { type: "if", cond: AST, then: AST, else: AST }

assign { type: "assign", operator: "=", left: AST, right: AST }

binary { type: "binary", operator: OPERATOR, left: AST, right: AST }

prog { type: "prog", prog: [ AST... ] }

Например:

Numbers ("num")

255.5 → { type: "num", value: 255.5 }

Strings ("str")

"Hello World!" → { type: "str", value: "Hello World!" }

func (x) 10 → { type: "func", vars: [ "x" ],body:

{ type: "num", value: 10 }}

Описание на lexer-a

Lexer-ът работи с входен поток от символи и връща stream обект със същия интерфейс, но стойностите, върнати от peek () / next (), ще бъдат символи. Този обект с две свойства: тип и стойност. Ето няколко примера:

{type: "punc", value: "("} - пунктуация: скоби, запетая, точка и запетая и т.н.

{type: "num", value: 5} - числа

{type: "str", value: "Hello, world!" } - низове

{type: "kw", value: "func"} - ключови думи

{type: "var", value: "a"} - идентификатори

{type: "op", value: "! ="} - операции

Шпациите и коментарите се прескачат, не се връщат символи.

За да напишем лексъра, трябва да разгледаме по-отблизо синтаксиса на нашия език. Идеята е, че в зависимост от текущия знак (както се връща от input.peek ()) можем да решим какъв токен да чете:

Първо, прескачаме шпациите.

Ако input.eof () след това върнете null.

Ако е цитат, прочитаме низ.

Ако е цифра, ще продължим да четем число.

Ако е буква, тогава прочитаме идентификатор или ключова дума.

Ако е един от препинателните знаци, връщаме знак за пунктуация.

Ако е оператор, връщаме маркера за оператор.

Ако е нищо от горното, грешка с input.croak ().

Описание на парсър

Парсерът работи върху поток от символи, вместо да се занимава с отделни символи. В него изпозлваме множество helper функции с цел да запазим сложността на кода по-ниска.

Видът на парсъра е recursive descent parser, тъй като този вид парсъри са едни от най-лесните за писане.

Този вид парсъри са изградени от набор от взаимно рекурсивни процедури (или нерекурсивен еквивалент), където всяка такава процедура изпълнява една от не-крайните елементи на граматиката. Така структурата на получената програма тясно отразява тази на граматиката, която тя разпознава.

Интерпретатор

Състои се от структура за правилното поддържане на околната среда(environment) - структура, притежаваща връзката на променливите с техните стойности. Тя ще бъде предадена като аргумент за нашата функция за оценка.

Всеки път, когато влезем в "ламбда" функция, трябва да създадем нов обхват (scope) с нови променливи (аргументи на функцията) и да ги инициализираме със стойности, предавани по време на изпълнение. Ако аргумент засенчва променлива от външния обхват, трябва да внимаваме да възстановим предишната стойност, когато напускаме функцията.

Най-простият начин за осъществяване на това е използването на наследството на прототипа на JavaScript. Когато влезем във функция, ще създадем нова среда, зададем прототипа му на външната (родителска) среда и ще оценим тялото на функцията в новата. По този начин, когато излезем, не е нужно да правим нищо - външният env вече ще съдържа всички сенки.

Накратко това е нашият език. Не е задължително да е добър. Синтаксисът изглежда сладък, но има своите капани. Има много липсващи функции, обекти и масиви, които естествено с допълнителна работа могат да бъдат добавени.