CSE 30151 Theory of Computing
Fall 2017
Project 4-Combinator Project

Version 1: Nov. 20, 2017

1 Overview

At this point you understand computations that happen as planned series of individual steps where various memory structures: states, stacks, tapes, etc are modified as a fundamental part of computing. The goal of this project is to introduce you to a different model of computation - combinators - where computation is just as powerful, but there is less required sequentialism and no explicit sense of reading and (re)writing to "memory".

Completion of this project will involve two parts. First is writing and running a program that evaluates an expression consisting solely of combinators. The second part is developing some interesting combinator expressions that relate to real computations.

As before, you are free to use C, C++, or Python. The latter in particular has proven in the past as perhaps the most efficient way to get decent working code (the overall goal). If you use Python, feel free to use the *time*, *csv*, *sys*, *itertools*, and *copy* modules. Use of any other modules requires preapproval of the instructor.

Since this is an extra credit project, only one student per project submission is permitted.

For questions, TA Jessica Cioffi is particularly knowledgeable about implementation issues, and both TA Josh Siva and Prof. Kogge can answer more theoretical questions.

2 Background

Most of today's programming languages are **imperative**, that is they specify a precise order for the computation, and rely on "side-effects" to perform computations. All automtata we have studied have sets of transitions that must be tailored to the specific program. In addition, the PDA has its stack, the TM has its tape, and modern computer languages have their "memory',' all of which have "variables" that are explicitly defined by, and then changed by, the program over time.

This need not be the case. There are many languages that are **declarative**: they define the problem and let the details of sequencing to find the answer up to the underlying mechanisms. Most database, logic, and graph languages have this property. In addition, such languages tend not to compute by side-effects; they do not explicitly name and (re)modify memory. They are often called **single-assignment** languages.

Perhaps even more important, such declarative languages also elevate functions to "first-class object" status where a function can be passed around like any other object: be provided as an argument to some other function, and be delivered as a result from such function evaluations. Such languages are often called **functional**.

2.1 Expressions

Most computation today revolves around "evaluation" of **expressions**: turning a string that identifies some function and some objects serving as arguments into a string representing some other object. Such expressions may be in an **infix** notation where the function is between operands (as in "12+34"), **postfix** notation where the function is after a set of operands (as in "12 34 +" as used in stack machines and generations of HP calculators), and in **prefix** notation where the function is before a set of operands (as in "(+ 12 34)" as in Scheme and when calling a user-defined function "f(x,y)" in C).

When an expression is complex, with lots of embedded sub-expressions, there is typically an evaluation order that directs the order in which the sub-expressions are to be evaluated. The result of one sub-evaluation becomes an argument to the next evaluation, as in $(12 + 34) * (56 + 78) \;\; => \;\; 46 * (56 + 78) \;\; => \;\; 46 * 134 \;\; => \;\; 7504$.

What makes functional languages different is that the result of one sub-evaluation is often itself a function that at the next step receives a set of arguments to be evaluated accordingly. Functions can thus be used for anything a traditional "object" can - indeed as we will see with combinators we need nothing other than functions.

3 Combinators

Loosely speaking, combinators form a functional language including expressions built from a set of very simple functions that do nothing more than "reorder" their arguments - whatever they are. This latter point is particularly important. An argument may be a function itself, and after evaluation of the sub-expression it may become the function for the next evaluation.

As was shown in the class lecture, a very minimal set of combinators (3 of them) are capable of doing absolutely anything a Turing Machine can do, making them equal in power to TMs. In fact, this minimal set needs nothing else - no "numbers," no "math," no "variables," no "booleans" or "comparisons," or anything else. In fact, even one of the three functions is redundant!

The chapter referenced on the "Links" page of the class web site has a detailed discussion of combinators.

3.1 Combinator Expressions

A simple CFL syntax for a combinator expression is as follows:

$$\begin{aligned} <caf> &\rightarrow \;\; <constant> \; \mid (<caf> \;\; <caf>^*) \\ <constant> \;\; &\rightarrow <combinator> \\ <combinator> &\rightarrow S \mid K \mid I \end{aligned}$$

Here a $<caf>$ stands for a **constant applicative form**, and has two forms: a **constant** that represents some basic object, and an expression in prefix form as discussed above that consists of a series of caf >s inside a set of "()". This latter form represents an expression that can be "evaluated" into something simpler: the result of using the first $< caf >$ as a function to be applied to the series of $< caf >$s that follow.

3.2 Basic Combinators

For the simplest version, the original set of $< constant >$s are the three combinators denoted "S," "K," and "I." If we let c_i represent the i'th $< caf >$ in the list inside the () after the first one, we can define the operation of the three combinators as follows:

$$(S\ c_1\ c_2\ c_3\ c_4...) => (c_1\ c_3\ (c_2\ c_3)\ c_4...)$$
$$(K\ c_1\ c_2\ c_3\ ...)\ => (c_1\ c_3\ ...)$$
$$(I\ c_1\ c_2\ ...) => (c_1\ c_2\ ...)$$

Applying any of these rules to an expression is called a **reduction**. In all cases, what starts out in an "argument" position becomes the "function" at the next step (e.g.c_1).

S is known as the **distributed application** combinator as it moves its first argument into the function position, makes its second argument the first argument to the new function, and sets up a third argument as a sub-expression again using a copy of the second argument as a function applied to the third argument.

The K combinator (for **constant** or **kill**) makes the first argument the new function, and deletes the second argument entirely.

The I combinator (for **identity**) simply promotes its first argument into the function position.

Several notes:

- The arguments beyond the ones needed by the evaluation need not be present.
- When an evaluation leaves a single combinator in (), as in (I), then the () may be deleted.
- When a $< caf >$ is of the form $((c_1\ c_2\ c_k)\ c_{k+1}\ ...)$, the set of () around the function can be eliminated, yielding $(c_1\ c_2\ ...\ c_k\ c_{k+1}\ ...)$
- The expression $(S\ K\ K\ c_1\ c_2\ ...) => (K\ c_1\ (K\ c_1)\ c_2\ ...) => (c_1\ c_2\ ...)$ has the same effect as I. Thus we could delete I by replacing it by (SKK).

3.3 Evaluation Order

There is no required order to evaluating a $< caf >$ - any time there is a $(< combinator >\ c_1\ c_2\ ...)$ where there are sufficient arguments for the combinator in the function position, then the string within the "()" can be replaced by the one resulting from the application of the combinator to its arguments. In fact, very often there are many such places in a long string for which such reductions can be made. It is perfectly permissible to do all such operations "in parallel." The answer will be the same as if they were done one at a time, in any order.

For simplicity, however, a **normal order evaluation** always starts at the leftmost such possible reduction, performs that one, and then looks for the next leftmost.

3.4 Booleans

Most programming languages have objects that represent "true" and "false". In combinators we want to find an expression that has the same effect as a boolean. A very natural approach is to see how we can duplicate their effect in such forms as if-then-else, which as an expression may take the form ($<$ *boolean* $>$ $<$ *expr*1 $>$ $<$ *expr*2 $>$. If the *boolean* is true, we should return *expr*1; if false we should return *expr*2. Consider the following:

- $(K\ e_1\ e_2\ e_3\ ...)\ => (e_1\ e_3\ ...)$
- $((S\ K)\ e_1\ e_2\ e_3\ ...)\ =>\ (S\ K\ e_2\ (e_1\ e_2)\ e_3\ ...)\ =>\ (K\ e_2\ (e_1\ e_2)\ e_3\ ...)\ =>\ (e_2\ e_3\ ...)$

Thus for all practical purposes we can replace "true" by K and "false" by (SK). Then any function that takes these as arguments and returns one of them as a result is a boolean function as we normally think about it.

3.5 Numbers

As discussed in class, combinator expressions can represent numbers, but in a different sense than we are used to. Assume all we have is "0" and a *successor* function s which if given an integer k then $s(k)$ returns k+1. Then a perfectly good standin for k is the function that applies s to 0 k times. Alternatively, we can make the integer k a function that, if given a "0" as an argument, returns the integer k.

This has the interesting effect of allowing us to do things like represent the sum of two integers k and j as the function $k(j)$ - a function which if given 0 will apply the successor function k+j times, i.e. $k = s(s(...s(0)...)$ with k applications of s.

The bottom of page 306 in the referenced book has an expression for s that has the correct properties.

3.6 Supercombinators

While S, K, and I are sufficient to compute anything, there are a few additional **supercombinators** that can be constructed from S, K, and I and that greatly simplify computations. Page 307 in the book chapter referenced on the Links page of the class web site lists many of them that can greatly reduce the execution time of a combinator expression. As with optimized Turing Machines, such as multi-tape, they can compute nothing beyond what S,K,I can.

Fig. 11-12 on page 310 of the referenced book shows that the combinator expression $(S\ B)$ when used in the function position of an expression is equivalent to the successor function.

3.7 Recursion

Recursion is at the heart of many programming languages. The Y supercombinator provides an elegant expression of this capability as follows:

$(Y\ e1\ e2)\ =>\ (e_1\ (Y\ e_1)\ e2)$

If we assume that e_1 is a function of two arguments, the first of which is a copy of itself and the second an argument, then Y applied to e_1 and some argument e_2 will put e_1 in the function position with a copy of itself as its first argument. Fig. 11-14 on page 3-12 in the reference book gives an example of factorial using this.

3.8 Built-ins

Again as shown in class, S, K, and I are sufficient to represent numbers, math, comparisons, etc., it often gets very boring to represent "+" for example as a long SKI string. Thus for simplicity, many people also allow basic math operators such as "+", "-", "*", "/," "==," "¿," ... to appear as $< constant >$s in a $< caf >$. Basic numbers are also often allowed as $< constant >$s. Thus a $< caf >$ of (* (+ 12 34) (+ 56 78)) reduces down to its equivalent number.

4 The Project Simulator

The goal of this project is to design a simulator which if given a $< caf >$ as a string as an input, shows all reductions and delivers what is the final answer. For ease of evaluation, each reduction should be presented on a separate line starting with a =>.

The input may be from a file or keyboard. If from a file, each line may thus represent a new string to execute.

It is strongly suggested for your own sanity that you implement a normal order evaluation, where each reduction is the leftmost one that can be made. If you use some other order, just document.

At a minimum the simulator should be capable of dealing with just S, K, and I. More extra credit points will be given if additional features are added, such as some of the super-combinators (especially B and Y), and/or simple builtins.

Also, it is strongly suggested that you allow lower case letters in expressions as standins for other expressions, like the c_is in the above definitions. Such lower case letters would be moved around as any other expression when used as arguments, but evaluation using them stops when they end up in a function position. Thus for example entering "$(Sabcd))$" into your simulator may return $(ac(bc)d)$.

5 Test Files

Under the Projects tab of the class website https://www3.nd.edu/ kogge/courses/cse30151-fa17/index.html there is a directory called Project4. Within this directory there are several test files. Each such file has five lines of sample input, a line of "—" and five lines of what your simulator should output for each.

Note that each string here has no outermost "()". You are free to add a set of "()" for clarity.

5.1 Student-supplied Expressions

You should also create some of your own functions in combinators and show that they work. Particularly straightforward examples are functions that perform AND, OR, or XOR on booleans.

6 Documentation

A key part of what you submit is a file that includes the following in this order:

1. Approximately how much time was spent in total on the project, and how much by each student.
2. A description of how you managed the code development and testing. Use of github in particular is a particularly strong suggestion to simplifying this process, as is some sort of organized code review.
3. The language you used, and a list of libraries you invoked.
4. The format of both input and output.
5. A description of what combinators your code is capable of executing.
6. A description of the key data structures you used and how you found the next expression to reduce.
7. A description of how you tested it and what functions you wrote.

7 Submission

- Each student submitting a project should have in their own Sakai directory for this course a directory called Project4, where all submissions should go.
- When you are ready to submit, you will place copies of all code and test machine output in the designated common directory.
  - Your code should be runnable on any of the studentnn.cse.nd.edu machines so that if there is an issue the graders can run the code themselves.
  - If you wrote in a compiled language like C++, include all needed source files (excepting standard libraries), a make file, and a compiled executable. The source code is there to allow the graders to look at the code for comments and to resolve any discrepancies that may arise in looking at your results.
  - If you wrote in a language like Python make sure your code is compatible with one of the versions supported on the studentnn.cse.nd.edu machines, again to allow the graders to check something if there is an issue.
- Also in the same directory as the code you should place an output file for each of the test files that you ran.

8 Grading

Grading of the project will be at the instructor's discretion, and will be factored into the student's grade after all other work assignments have been considered, and an initial course grade developed. Typically, a well-executed project will boost a student's letter grade one or more places.