Лабораторная работа № 5. Интерпретатор стекового языка программирования

18 ноября 2024 г.

Александр Яннаев, Матвей Хромов, ИУ9-11Б

Цель работы

Познакомиться с интерпретацией низкоуровневого кода на примере **FORTH**-подобного языка программирования.

Условие задачи

Реализуйте интерпретатор стекового языка программирования, описание которого представлено ниже. Интерпретатор должен вызываться как процедура `(interpret program stack)` которая принимает программу на исходном языке `program` и начальное состояние стека данных `stack` и возвращает его состояние после вычисления программы. Программа на исходном языке задана вектором литеральных констант, соответствующих словам исходного языка. Исходное и конечное состояния стека данных являются списком, голова которого соответствует вершине стека.

Примеры вызова интерпретатора (здесь и далее в примерах код на исходном языке выделен синим цветом):

```
(interpret #(   define abs
                  dup 0 <
                  if neg endif
                end
                abs    ) ; программа
           '(-9))         ; исходное состояние стека
  (9)
```

При реализации интерпретатора избегайте императивных конструкций, используйте модель вычислений без состояний. Для хранения программы и состояния интерпретатора **запрещается** использовать глобальные переменные. Перечисленные ниже встроенные слова обязательны для реализации и будут проверены сервером тестирования. # Описание языка

Язык, интерпретатор которого следует реализовать, является видоизмененным ограниченным подмножеством языка **Forth**.

В нашем языке операции осуществляются с целыми числами. Используется постфиксная запись операторов. Все вычисления осуществляются на стеке данных. Стек данных является глобальным. При запуске интерпретатора стек может быть инициализирован некоторыми исходными данными или быть пустым.

Программа на исходном языке представляет собой последовательность слов. Интерпретатор анализирует слова по очереди. Если слово является целым числом, то оно число помещается на вершину стека данных. В противном случае слово интерпретируется как оператор (процедура). Если в программе уже встретилось определение этого слова (статья), то выполняется код этого определения. В противном случае слово рассматривается как встроенное в интерпретатор и выполняется соответствующей процедурой интерпретатора. Затем осуществляется возврат из процедуры (переход к слову, следующему за последним вызовом). Выполнение программы заканчивается, когда выполнено последнее слово.

Процедуры (операторы) снимают свои аргументы с вершины стека данных и кладут результат вычислений также на вершину стека данных.

Ввод-вывод или какое-либо взаимодействие с пользователем не предусматривается.

Например:

```
(interpret #(2 3 * 4 5 * +) '()) ▯ (26)
```

Встроенные слова

Ниже представлен список встроенных слов с кратким описанием их значений. Состояние стека до и после интерпретации каждого слова показаны с помощью схем — стековых диаграмм. Порядок, в котором элементы были помещены в стек, отражен в индексах элементов. Например, программа: `1 2 3` может быть показана стековой диаграммой `() → (1 2 3)`

Внимание! В нашем интерпретаторе в качестве стека используется список. Голова этого списка является вершиной стека, поэтому вершина стека в этих диаграммах находится *слева*! Такая запись отличается от традиционных стековых диаграмм, принятых, например, в языке Forth, в которых голова стека записывается *справа*.

Арифметические операции

`+ (n2 n1) → (сумма)` - Сумма `n1` и `n2` `- (n2 n1) → (разность)` - Разность: `n1 - n2` `* (n2 n1) → (произведение)` - Произведение `n2` на `n1` `/ (n2 n1)`

`→ (частное)` - Целочисленное деление `n1` на `n2` `mod (n2 n1) → (остаток)` - Остаток от деления `n1` на `n2` `neg n. → (-n)` - Смена знака числа

Операции сравнения

`= (n2 n1) → (флаг)` - Флаг равен `-1`, если `n1 = n2`, иначе флаг равен `0` `> (n2 n1) → (флаг)` - Флаг равен `-1`, если `n1 > n2`, иначе флаг равен `0` `< (n2 n1) → (флаг)` - Флаг равен `-1`, если `n1 < n2`, иначе флаг равен `0` Таким образом, булевы значения представлены с помощью целых чисел: `-1` соответствует значению «истина», `0` — значению «ложь».

Логические операции

`not (n) → (результат)` - НЕ `n` `and (n2 n1) → (результат)` - `n2` И `n1` `or (n2 n1) → (результат)` - `n2` ИЛИ `n1` Эти операции также должны давать правильный результат, если в одном или обеих операндах «истина» представлена любым ненулевым целым числом.

Операции со стеком

При выполнении вычислений на стеке часто возникает необходимость изменять порядок следования элементов, удалять значения, копировать их и т.д. Для этого реализуйте следующие операции: `drop (n1) → ()` - Удаляет элемент на вершине стека `swap (n2 n1) → (n1 n2)` - Меняет местами два элемента на вершине стека `dup (n1) → (n1 n1)` - Дублирует элемент на вершине стека `over (n2 n1) → (n1 n2 n1)` - Копирует предпоследний элемент на вершину стека `rot (n3 n2 n1) → (n1 n2 n3)` - Меняет местами первый и третий элемент от головы стека `depth (…) → (n …)` - Возвращает число элементов в стеке перед своим вызовом

Управляющие конструкции

`define _word_ () → ()` - Начинает словарную статью — определение слова `_word_` `end () → ()` - Завершает статью `exit () → ()` - Завершает выполнение процедуры (кода статьи) `if (флаг) → ()` - Если флаг не равен `0`, то выполняется код в теле `if..endif`, иначе выполнение кода до `endif` пропускается `endif () → ()` - Завершает тело `if`

Пусть слово `define _word_` начинает определение слова `_word_`. В теле определения (словарной статьи) следуют слова, которые надо вычислить, чтобы вычислить слово `word`. Статья заканчивается словом `end`. Определенное таким образом слово может быть использовано в программе так же, как и встроенное. Например, унарный декремент может быть определен, а затем использован так:

```
(interpret #(   define -- 1 - end
                5 -- --      ) '())
  (3)
```

Завершить выполнение процедуры до достижения её окончания `end` можно с помощью слова `exit`.

В статьях допускаются рекурсивные определения. Вложенные словарные статьи не допускаются.

Конструкции `if`…`endif` не должны быть вложенными (в ЛР). В программах ниже даны примеры их использования.

Реализация

```
(define (interpret program stack)
  (define (execute words idx stack return-stack dictionary)
    (if (>= idx (vector-length words))
        stack
        (let ((word (vector-ref words idx)))
          (cond
            ((integer? word)
             (execute words (+ idx 1) (cons word stack) return-stack dictionary))

            ; arithmetic ops
            ((eq? word '+)
             (execute-binop words idx stack return-stack dictionary +))
            ((eq? word '-)
             (execute-binop words idx stack return-stack dictionary -))
            ((eq? word '*)
             (execute-binop words idx stack return-stack dictionary *))
            ((eq? word '/)
             (execute-binop-checked words idx stack return-stack dictionary quotient))
            ((eq? word 'mod)
             (execute-binop-checked words idx stack return-stack dictionary modulo))
            ((eq? word 'neg)
             (execute-unop words idx stack return-stack dictionary -))

            ; logic ops
            ((eq? word '=)
             (execute-binop words idx stack return-stack dictionary
                            (lambda (a b) (if (= a b) -1 0))))
            ((eq? word '<)
             (execute-binop words idx stack return-stack dictionary
                            (lambda (a b) (if (< a b) -1 0))))
            ((eq? word '>)
             (execute-binop words idx stack return-stack dictionary
                            (lambda (a b) (if (> a b) -1 0))))
            ((eq? word 'not)
             (execute-unop words idx stack return-stack dictionary
```

```
                (lambda (a) (if (zero? a) -1 0))))
((eq? word 'and)
 (execute-binop words idx stack return-stack dictionary
                (lambda (a b) (if (and (not (zero? a)) (not (zero? b)))
                                  -1 0))))
((eq? word 'or)
 (execute-binop words idx stack return-stack dictionary
                (lambda (a b) (if (or (not (zero? a)) (not (zero? b)))
                                  -1 0))))

; stack ops
((eq? word 'drop)
 (execute-stack-op words idx stack return-stack dictionary
                   (lambda (s) (cdr s))))
((eq? word 'swap)
 (execute-stack-op words idx stack return-stack dictionary
                   (lambda (s) (cons (cadr s) (cons (car s) (cddr s))))))
((eq? word 'dup)
 (execute-stack-op words idx stack return-stack dictionary
                   (lambda (s) (cons (car s) s))))
((eq? word 'over)
 (execute-stack-op words idx stack return-stack dictionary
                   (lambda (s) (cons (cadr s) s))))
((eq? word 'rot)
 (execute-stack-op words idx stack return-stack dictionary
                   (lambda (s) (cons (caddr s)
                                     (cons (cadr s)
                                           (cons (car s) (cdddr s)))))))
((eq? word 'depth)
 (execute-stack-op words idx stack return-stack dictionary
                   (lambda (s) (cons (length s) s))))

; ctrl funcs
((eq? word 'define)
 (let ((name (vector-ref words (+ idx 1))))
   (let loop ((i (+ idx 2)))
     (if (or (>= i (vector-length words)) (eq? (vector-ref words i) 'end))
         (execute words (+ i 1) stack return-stack
                  (cons (cons name (+ idx 2)) dictionary))
         (loop (+ i 1))))))
((eq? word 'end)
 (if (null? return-stack)
     stack
     (execute words (car return-stack) stack
              (cdr return-stack) dictionary)))
((eq? word 'exit)
```

```
            (if (null? return-stack)
                stack
                (execute words (car return-stack) stack
                         (cdr return-stack) dictionary)))
            ((eq? word 'if)
             (if (zero? (car stack))
                 (let loop ((i (+ idx 1)))
                   (if (or (>= i (vector-length words)) (eq? (vector-ref words i) 'endif))
                       (execute words (+ i 1) (cdr stack) return-stack dictionary)
                       (loop (+ i 1))))
                 (execute words (+ idx 1) (cdr stack) return-stack dictionary)))
            ((eq? word 'endif)
             (execute words (+ idx 1) stack return-stack dictionary))

            ; user funcs
            ((assoc word dictionary)
             (execute words (cdr (assoc word dictionary)) stack
                      (cons (+ idx 1) return-stack) dictionary))

            ; unknown word
            (else stack)))))

; helper functions
(define (execute-binop words idx stack return-stack dictionary op)
  (if (< (length stack) 2)
      stack
      (execute words (+ idx 1)
               (cons (op (cadr stack) (car stack)) (cddr stack))
               return-stack dictionary)))
(define (execute-binop-checked words idx stack return-stack dictionary op)
  (if (< (length stack) 2)
      stack
      (if (= (car stack) 0)
          #f
          (execute-binop words idx stack return-stack dictionary op))))
(define (execute-unop words idx stack return-stack dictionary op)
  (if (null? stack)
      stack
      (execute words (+ idx 1)
               (cons (op (car stack)) (cdr stack))
               return-stack dictionary)))
(define (execute-stack-op words idx stack return-stack dictionary op)
  (if (null? stack)
      stack
      (execute words (+ idx 1)
               (op stack)
```

```
                return-stack dictionary)))

  ; start execution
  (execute program 0 stack '() '()))


;; testing...

; arithmetic
(interpret #(2 3 +) '()) ; -> (5)
(interpret #(10 4 -) '()) ; -> (6)
(interpret #(3 7 *) '()) ; -> (21)
(interpret #(20 5 /) '()) ; -> (4)
(interpret #(17 5 mod) '()) ; -> (2)
(interpret #(7 neg) '()) ; -> (-7)

; logic
(interpret #(5 5 =) '()) ; -> (-1)
(interpret #(10 5 >) '()) ; -> (-1)
(interpret #(3 8 <) '()) ; -> (-1)
(interpret #(0 not) '()) ; -> (-1)
(interpret #(-1 0 and) '()) ; -> (0)
(interpret #(-1 -1 or) '()) ; -> (-1)

; stack
(interpret #(drop) '(42)) ; -> ()
(interpret #(swap) '(2 3)) ; -> (3 2)
(interpret #(dup) '(9)) ; -> (9 9)
(interpret #(over) '(1 2)) ; -> (2 1 2)
(interpret #(rot) '(1 2 3)) ; -> (3 2 1)
(interpret #(depth) '(1 1 1)) ; -> (3 1 1 1)

; global
(interpret #(   define abs
                    dup 0 <
                    if neg endif
                end
                 9 abs
                -9 abs      ) (quote ()))
;  □ (9 9)

(interpret #(   define =0? dup 0 = end
                define <0? dup 0 < end
                define signum
                    =0? if exit endif
                    <0? if drop -1 exit endif
```

```
                    drop
                    1
                end
                 0 signum
                -5 signum
                10 signum       ) (quote ()))
;  (1 -1 0)

(interpret #(   define -- 1 - end
                define =0? dup 0 = end
                define =1? dup 1 = end
                define factorial
                    =0? if drop 1 exit endif
                    =1? if drop 1 exit endif
                    dup --
                    factorial
                    *
                end
                0 factorial
                1 factorial
                2 factorial
                3 factorial
                4 factorial     ) (quote ()))
;  (24 6 2 1 1)

(interpret #(   define =0? dup 0 = end
                define =1? dup 1 = end
                define -- 1 - end
                define fib
                    =0? if drop 0 exit endif
                    =1? if drop 1 exit endif
                    -- dup
                    -- fib
                    swap fib
                    +
                end
                define make-fib
                    dup 0 < if drop exit endif
                    dup fib
                    swap --
                    make-fib
                end
                10 make-fib     ) (quote ()))
;  (0 1 1 2 3 5 8 13 21 34 55)

(interpret #(   define =0? dup 0 = end
```

```
                define gcd
                    =0? if drop exit endif
                    swap over mod
                    gcd
                end
                90 99 gcd
                234 8100 gcd    ) '())
;  (18 9)
```

Тестирование

```
Language: R5RS; memory limit: 128 MB.
(5)
(6)
(21)
(4)
(2)
(-7)
(-1)
(-1)
(-1)
(-1)
(0)
(-1)
()
(3 2)
(9 9)
(2 1 2)
(3 2 1)
(3 1 1 1)
(9 9)
(1 -1 0)
(24 6 2 1 1)
(0 1 1 2 3 5 8 13 21 34 55)
(18 9)
```

Вывод

В ходе этой лабораторной работы мы реализовали интерпретатор для простого стекового языка программирования, похожего на **FORTH**. Главная задача заключалась в том, чтобы научить интерпретатор выполнять арифметические операции, работать с логическими значениями, изменять порядок элементов в стеке и выполнять пользовательские команды.

Нам удалось написать интерпретатор так, чтобы он читал программу,

анализировал команды одну за другой и правильно обрабатывал данные в стеке. Например, он может выполнять сложение, умножение, сравнения, дублировать элементы стека или определять свои команды через `define`.

Самым интересным было сделать программу чисто функциональной, то есть без всяких глобальных переменных и циклов. Вместо этого мы использовали рекурсию. Тесты показали, что всё работает как надо, даже сложные примеры вроде факториала или вычисления чисел Фибоначчи.

Мы довольны результатом, потому что интерпретатор полностью соответствует заданию, а еще это хороший опыт для понимания, как работают языки программирования «изнутри».