Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатики и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

Р А С Ч Ё Т Н О - П О Я С Н И Т Е Л Ь Н А Я З А П И С К А

к курсовому проекту по дисциплине

Конструирование компиляторов

на тему

Компилятор подмножества языка Scheme в подмножество языка Forth

Студент _____ **Мокров А.М.**
(Подпись, дата) **ИУ9-91** (Фамилия И.О., группа)

Руководитель курсового проекта _____ **Дубанов А.В.**
(Подпись, дата) (Фамилия И.О.)

Москва, 2015

Содержание

Введение

На сегодняшний день использование языков низкого уровня для создания программ, которые отвечают современным требованиям, зачастую связано с большими трудозатратами, поэтому для большинства практических задач решение инструментами языка более высокого уровня является наиболее оптимальным. Общепринятой практикой считается первичная реализация компилятора языка C с последующим созданием на его базе компиляторов более высокого уровня.

Целью данной работы является создание компилятора подмножества высокоуровневого языка Scheme в язык Forth при условиях наличия на машине только лишь интерпретатора Forth, то есть целевой язык является языком реализации.

1. Обзор исходного и целевого языков

Появление языков более высокого уровня было вызвано потребностью создания более сложных программ в кратчайшие сроки, необходимостью упрощения общих принципов разработки, возможностью реализации общепринятых алгоритмов и подходов как базовых функций языка.

В качестве исходного языка компилятора был выбран Scheme, который был создан Гаем Стилом и Джеральдом Сассменом в 1975 году как диалект языка Lisp. Lisp является функциональным языком высокого уровня, в то же время диалект языка основан на S-выражениях [1], что значительно упрощает процесс построения структуры программы в процессе синтаксического анализа. Реализация компилятора Lisp-подобного языка, который включает все преимущества представления и обработки данных, позволяет построить в дальнейшем с его помощью компиляторы языков более высокого уровня, а также создавать высокоэффективные программы без использования в цикле разработки компиляторов языка C.

Целевой язык Forth является одним из первых конкатенативных языков программирования, он был Чарльзом Х. Муром в конце 1960-х годов, такой выбор языка основывается на простоте реализации, которой способствует использование стековой нотации, постфиксная запись, представление исходного кода как последовательности лексем, что сводит реализацию компилятора языка Forth к почти тривиальной задаче [2]. В проекте использован интерпретатор Gforth.

2. Обзор синтаксиса языка Scheme

С целью корректного определения требуемого подмножество языка, необходимо изучить его структуру и основные преимущества реализованных возможностей [3].

Как уже упоминалось ранее, программа на языке Scheme строится на S-выражениях, то есть состоит из списков, заключенных в скобки, элементы которого разделены пробелами. Вызов функции записывается в виде «(f x y z)», где «f» – имя функции, а «x», «y», «z» – передаваемые аргументы. Однострочный комментарий начинается с точки запятой и продолжается до конца строки. Далее рассмотрим базовые элементы языка:

Числа:

```
99999999999999999999 ; integers
#b111 ; binary => 7
#o111 ; octal => 73
#x111 ; hexadecimal => 273
3.14 ; reals
6.02e+23
1/2; rationals
1+2i; complex numbers
```

Булева алгебра:

```
#t ; => true
#f ; => false
```

Символы и строки:

```
#\A ; => #\A
```

также символ можно задать кодом символа:

```
#\x03BB ; => #\λ
```

строки, как и в многих языках, заключаются в двойные кавычки:

```
"Hello, \"world\"!"
```

Для печати строки используется функция «display».

Списки задаются следующими способами:

```
(list 1 2 3 4 5 (+ 1 2 3)) ; => (1 2 3 4 5 6)
(quote (1 2 3 4 5 (+ 1 2 3))) ; => (1 2 3 4 5 (+ 1 2 3))
'(1 2 3 4 5 (+ 1 2 3)) ; => (1 2 3 4 5 (+ 1 2 3))
```

Переменные:

Переменные могут быть объявлены, с помощью «define», переопределены с помощью «set!», также имеется функция «let», которая позволяет объявлять сразу группу локальных переменных.

```
(let
  ((a 1) (b 2) (c 3) (d 4))
  (set! d 6)
  (+ a b c d))
```

Имя переменной может содержать любые символы за исключением: «()[]{}”,’`;#/\».

Функции:

Для создания функций используется «lambda» или «define».

```
((lambda (x) (+ x x)) 1)
(define (plus x) (+ x x))
```

Если в процессе выполнения функции было получено несколько значений, то функция возвращает самое последнее, то есть при вызове функции «x», определенной следующим образом:

```
(define (x) 1 2 3)
```

функция вернёт значение «3». Однако, в таком случае рекомендуется использовать функцию «begin»:

```
(define (x) (begin (display 1) 3))
```

которая вернет также «3», но не нарушит целостности стека возврата в процессе выполнения.

Управление потоком:

Для выполнения условных переходов в Scheme используется функция «if»:

```
(if #t ; условие
  1 ; если истина
  0); если ложь
```

Для множества последовательны условий применяется конструкция «cond»:

```
(cond
  ((> 1 1) "wrong!")
  ((< 1 1) "wrong!")
  ((= 1 1) "ok")
  (else "wrong!"))
```

Циклы реализуются преимущественно с помощью хвостовой рекурсии.

Данное описание помогает более чётко определить список необходимых к реализации типов и функций.

3. Выбор подмножества языка Scheme

Перед основной стадией разработки компилятора языка Scheme необходимо выбрать его подмножество, реализация компилятора для которого является необходимой для решения поставленной задачи.

Как уже упоминалось в предыдущей главе, структура языка основана на cons-ячейках, что даёт высокий приоритет реализации списков и операций над ними, кроме того необходимо сохранить функциональную парадигму исходного языка.

После рассмотрения основных преимуществ языка Scheme, а также обзора существующих компиляторов минимального его подмножества [4, 5], для реализации были выбраны следующие функции:

- определения: lambda, define (без определения через «lambda»);
- условные операторы и логические операции: if, or, and, not, boolean?;
- числа: number?, real?, integer?, +, –, *, /, <, <=, >, >=, =, zero?, positive?, negative?, abs, quotient, remainder, max, min;
- символы: char?, char->integer, integer-char;
- списки: car, cdr, cons, null?, list, list?, length, map, for-each, apply («map», «for-each» только для унарных функций, определенных с помощью «lambda»; «apply» только для списков);
- строки: string?, make-string, string-length, string-ref, string-set!, string->list, list->string;
- ввод/вывод (только stdout): display, newline.
- другие: begin, eq?.

Реализация данных функций включает в себя поддержку следующих типов:

- целые числа с поддержкой шестнадцатеричного, десятичного, двоичного представлений (ввод);
- числа с плавающей запятой;

- символы (без поддержки ввода с помощью кода символа);
- строки;
- булевы значения;
- списки.

Поддержка обыкновенных дробей и комплексных чисел не представлена, так как в Forth нет аналогичных типов данных, а реализация вычислительных методов такого типа с нуля является трудоёмкой задачей и не является основной целью данного курсового проекта.

Полный список поддерживаемых функций представлен в приложении 1 в алфавитном порядке.

4. Создание компилятора

Компилятор представляет собой программу, которая выполняет компиляцию исходного кода из входного файла, результирующий код записывается в выходной файл, имена файлов задаются как аргументы входной строки.

Процесс создания компилятора можно разбить на два основных этапа [6]:

1. разработка лексического и синтаксического анализаторов;
2. разработка семантического анализатора и генератора кода.

4.1. Разработка лексического и синтаксического анализаторов

На данном этапе в процессе компиляции происходит получение последовательности токенов и построение дерева разбора [7].

Перед выполнением лексического анализа необходимо считать входной поток, считывание проводится базовыми средствами Forth посимвольно без использования буфера, полученные данные объединяются в лексемы, выполняется определение соответствующих им лексических доменов, также выполняется определение текущей позиции в файле.

Используются следующие теги токенов: левая скобка, правая скобка, булевы данные, целое число, вещественное число, символ, строка, идентификатор. Пробельные символы учитываются при неоднозначности представления последовательности токенов.

Лексическую структуру выбранного подмножества языка можно представить следующим образом:

```
bracket-left ::= '('
bracket-right ::= ')'
boolean :: = '#' 't' | '#' 'f'
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
     | '9'
float ::= ('+' | '-')? digit+ '.' digit+ (('e' | 'E') (('+' |
     '-')? digit+)?)?
integer-decimal ::= ('+' | '-')? digit+
digit-binary ::= '0' | '1'
```

```
integer-binary ::= '#' 'b' ('+' | '-')? digit-binary+
digit-hexadecimal ::= 'A' | 'a' | 'B' | 'b' | 'C' | 'c' | 'D'
     | 'd' | 'E' | 'e' | 'F' | 'f' | digit
integer-hexadecimal ::= '#' 'x' ('+' | '-')?
     digit-hexadecimal+
integer ::= integer-decimal | integer-binary |
     integer-hexadecimal
character-letter ::= любой непробельный символ
character ::= '#' '\' character-letter
string-letter ::= любой символ, кроме '"' и '\'
string-element ::= string-letter | '\' '"' | '\' '\'
string ::= '"' string-element* '"'
letter ::= 'A' | ... | 'Z' | 'a' | ... | 'z'
letter-special ::= '!' | '$' | '%' | '&' | '*' | '/' | ':' |
     '<' | '=' | '>' | '?' | '^' | '_' | '~' | '.' | '@'
letter-idetifier ::= '+' | '-' | '*' | '/' | ...
identifier-head :: = letter | letter-special |
     letter-identifier
identifier-tail :: = identifier-head | digit
identifier ::= identifier-head identifier-tail*
```

В связи со сложностью хранения больших структур данных средствами Forth [8] процесс компиляции выполняется в один проход, а фазы работают параллельно.

Одновременно с лексическим анализом выполняется синтаксический анализ полученных токенов методом рекурсивного спуска. На данном этапе проводится проверка токена на соответствие синтаксической структуре языка.

Синтаксическую структуру языка можно описать следующим образом:

```
program ::= expression*
expression ::= boolean | integer | float | character | string
     | function | identifier
function ::= bracket-left identifier expression*
     bracket-right
```

Если в процессе синтаксического анализа была обнаружена ошибка в исходном коде, выводится сообщение об ошибке с текущей позицией во входном файле, процесс компиляции завершается, восстановление при ошибках не производится.

4.2. Разработка семантического анализатора и генератора кода

Контроль соответствия типов будет осуществляться в процессе выполнения сгенерированного кода на языке Forth, таким образом

семантический анализ сводится к контролю области определения функций и количества переданных аргументов. Так как компилятор устроен так, что фазы компиляции выполняются параллельно, вышеупомянутые проверки будут выполнятся одновременно с генерацией кода, способ их осуществления будет объяснен при описании структуры генератора кода.

Для реализации возможностей Scheme на языке Forth был создан интерфейс, который позволит выполнять код, сгенерированный данным компилятором. На базе полученного интерфейса в процессе выполнения скомпилированного кода интерпретатором Forth будет выполнятся проверка соответствия типов, корректность аргументов функции.

Для реализации поддержки базовых типов подмножества языка каждому элементу на стеке предшествует тег, определяющий его тип (вершина стека – справа):

- (-1 @boolean) – булевы данные («-1» соответствует значению «true», «0» – значению «false»);
- (12 @integer) – целое число;
- (f: 1.2e0 @float) – вещественное число, где «f:» стек чисел с плавающей запятой;
- (65 @character) – символ;
- (addr u @string) – строка длины «u», начало которой расположено по адресу «addr»;
- (addr-tail addr-head @list) – список, по адресу «addr-head» расположен элемент, представляющий голову списка, список, который является хвостом списка расположен по адресу «addr-tail».

Если список состоит из одного элемента, то значение «addr-tail» равно нулю, а если список является пустым, то и «addr-tail», и «addr-head» равны нулю.

Для реализации поддержки чисел с плавающей запятой использован базовый тип Forth для вещественных чисел, для приведения к такому типу число

представляется в экспоненциальной записи. Так как в Forth для значений с плавающей запятой используется отдельный стек, это необходимо учесть при реализации функций, работающих с этим типом данных.

В процессе компиляции все функции и их вызовы переименовываются, для того, чтобы не возникало противоречий с функциями языка Forth. Для базовых функций языка Scheme выполняется добавление нижнего подчеркивания к началу названия функции, то есть все вызовы «+» переименовываются в «_+». Полученная функция «_+», как и другие основные функции, реализована в написанном интерфейсе, при её вызове перед сложением выполняется вычисление аргументов, проверка типов полученных аргументов, их количество. Информация о списке базовых функций и количестве их аргументов занесена в область данных компилятора.

В процессе генерации кода передаваемые аргументы функций записываются в обратном порядке, что реализуется за счет использования моментальной генерации простейших деревьев разбора в конечный код и последующей их конкатенации в обратном порядке. Таким образом выполняется преобразование префиксной нотации языка Scheme в постфиксную нотацию языка Forth, то есть вызов функции также переносится в конец, когда уже все её аргументы вызываемой функции загружены на стек.

Для того, чтобы при выполнении полученного кода на языке Forth не происходило моментальное вычисление функции, а это необходимо при использовании постфиксной нотации для функций, аргументы которых могут оказаться невычислимы (например, «if»), вызов функции преобразуется в токен выполнения и добавляется на стек. Таким образом, вызов функции представлен последовательностью вида: (xt n @function), где «@function» – тег, определяющий, что данные на стеке представляют вызов функции, «n» – количество аргументов функции, «xt» – адрес токена выполнения данной функции, который генерируется выполнением кода: «' f», «f» – название

рассматриваемой функции. С целью поддержания целостности данных для функций «newline» и «display» языка Scheme также был добавлен тип данных «@unspecified», который они и возвращают.

Так как вычисление функций не выполняется сразу, выполнение начинается вызовом «eval», который при нахождении на вершине стека тега «@function» выполняет функцию по адресу токена, следующему за этим тегом, иначе – стек не изменяется.

Таким образом, следующий код на Scheme:

```
(+ 1 2 (+ 2 3.0))
```

Будет преобразован в следующий код на конечном языке:

```
require new-api.fs

3.0000000000000000E0 @float 2 @integer 2 ' _+ @function 2
@integer 1 @integer 3 ' _+ @function eval
bye
```

Здесь файл «new-api.fs» содержит определения тегов, а также реализацию всех функций интерфейса.

Функция «eval» вызывается при каждом возвращении на нулевой уровень глубины, также она вызывается внутри функций интерфейса для подсчета значений аргументов, при отсутствии необходимости подсчета значения аргумента вызывается функция «drop-typed», которая позволяет очистить дерево выполнения, находящееся в данный момент на вершине стека.

Для удобства манипулирования с переданными аргументами функций существует возможность упаковки данных и размещению по адресу после вычисления, а также последующей распаковки (функции «pack» и «unpack» соответственно).

Далее представлена реализация функции «_+» из файла «new-api.fs»:

```
: 2_+
  eval pack { x }
  eval pack { y }
  x unpack
  case
    @integer of
      y unpack
```

```
      case
        @integer of
          + @integer
        endof
        @float of
          i>f fswap f+ @float
        endof
        s" +: unexpected type" error-internal
      endcase
    endof
    @float of
      y unpack
      case
        @integer of
          i>f f+ @float
        endof
        @float of
          f+ @float
        endof
        s" +: unexpected type" error-internal
      endcase
    endof
    s" +: unexpected type" error-internal
  endcase
;

: _+
  { n }
  eval
  n 1 ?do
    2_+
  loop
;
```

Forth не позволяет контролировать правила видимости идентификаторов на уровне языка, а также не позволяет определять новые слова (функции) внутри определений слов. Таким образом, подобный контроль выполняется на этапе семантического анализа и генерации кода. Для этого необходимо отдельно обрабатывать вызовы функции «define» исходного языка. В процессе компиляции вызовов данных функций дерево разбора, представляющее тело определяемой с помощью рассматриваемых примитивов функции, сразу преобразуется в код на конечном языке и сохраняется в памяти, из общего дерева разбора данное поддерево удаляется. В случае определения функции внутри функции выполняются аналогичные операции.

Определение функции с помощью «lambda» в свою очередь обрабатывается также как вызов «define», где название определяемой функции «lambda», и последующий ее вызов. Так как при определении функций с помощью «define» через «lambda» количество аргументов задано неявно, использование «lambda» в таких конструкция не поддерживается.

С целью соблюдения правил видимости функций, названия определяемых функций, в отличие от базовых функций, переименовываются не только посредством добавления нижнего подчеркивания, но ещё приписыванием порядкового номера в конце функции, то есть функция «f» будет переименована в «_f123», если до этого были переименованы уже 122 функции. Информация о переименовании сохраняется в памяти с учетом глубины текущей позиции. При повышении уровня вложенности данные о переименованиях остаются актуальными и позволяют корректно переименовать все вызовы определенных в данной области видимости функций. При понижении глубины вложенности выполняется удаление данных о переименованиях функций, определенных на более глубоком уровне. Использование данного метода также позволяет избежать конфликтов при переопределении функции, так как она получает новое имя, а данные о переименованиях обновляются таким образом, что последующие вызовы данной функции будут изменены на вызовы функции с новым именем. При определении функции также сохраняется количество необходимых аргументов, создание новых функций с неопределенным количеством аргументов не поддерживается.

Так как в процессе компиляции можно получить информацию о переименованных функциях и известны базовые функции, выполняется проверка существования данных функций в текущей области видимости, количество передаваемых аргументов, в случае их отсутствия или несоответствии количества аргументов выводится сообщение о возникшей ошибке.

Определения функций обрабатываются отдельно от общего вывода сгенерированного кода и добавляется в начало выходного файла.

Тело функций обрабатывается по общему правилу, в названия аргументов также добавляется нижнее подчеркивание, выполняется аналогичный названиям функций контроль видимости переменных. Так как функции можно рассматривать как подпрограммы, после каждого элемента нулевого уровня глубины относительно тела функции добавляется «eval». Стоит отметить, что рекомендуется использовать функцию «begin», которая возвращает результат выполнения только последней функции, иначе на стеке будут находится значения всех выполненных функций.

В качестве примера рассмотрим определение функции добавления единицы к переданному аргументу на языке Scheme:

```
(define (f x) (+ x 1))
```

Данный участок кода будет преобразован в следующий код на языке Forth:

```
: _f1 recursive drop eval pack { _x }  1 @integer _x unpack 2
['] _+ @function eval ;
```

5. Тестирование компилятора

Тестирование было проведено таким образом, что каждая реализованная функция была вызвана по меньшей мере один раз.

Далее представлен сгенерированный код и результаты его выполнения для некоторых примеров:

Scheme	Forth

(display
  (map
  (lambda (x)
  (begin (display x)(+ x 1)))
  (list 1 2 3 4)))
 	
Вывод: 1234(2 3 4 5)
	require new-api.fs
	: _lambda01 recursive drop eval pack { _x } 1 @integer _x unpack 2 ['] _+ @function _x unpack 1 ['] _display @function 2 ['] _begin @function eval ;
	4 @integer 3 @integer 2 @integer 1 @integer 4 ' _list @function 1 ' _lambda01 @function 2 ' _map @function 1 ' _display @function eval
	bye
	
	Вывод: 1 2 3 4 (2 3 4 5)

(display
  (list->string
  (cdr
  (string->list "HHello, World!"))))
 	
Вывод: Hello, World!
	require new-api.fs
	s\" HHello, World!" @string 1 ' _string->list @function 1 ' _cdr @function 1 ' _list->string @function 1 ' _display @function eval
	bye
	
	Вывод: Hello, World!

(define (prime? n)
  (define (help-prime n a)
  (if (not(or (eq? (remainder n a) 0)
  (> a (quotient n 2))))
  (help-prime n (+ 1 a))
  (not (eq? (remainder n a) 0))))
  (help-prime n 2))
(display (prime? 23))
 	
Вывод: #t
	require new-api.fs
	: _help-prime2 recursive drop eval pack { _n } eval pack { _a } 0 @integer _a unpack _n unpack 2 ['] _remainder @function 2 ['] _eq? @function 1 ['] _not @function _a unpack 1 @integer 2 ['] _+ @function _n unpack 2 ['] _help-prime2 @function 2 @integer _n unpack 2 ['] _quotient @function _a unpack 2 ['] _> @function 0 @integer _a unpack _n unpack 2 ['] _remainder @function 2 ['] _eq? @function 2 ['] _or @function 1 ['] _not @function 3 ['] _if @function eval ; : _prime?1 recursive drop eval pack { _n } (define)2 @integer _n unpack 2 ['] _help-prime2 @function eval ;
	23 @integer 1 ' _prime?1 @function 1 ' _display @function eval
	bye
	
	Вывод: #t

Заключение

В рамках курсового проекта был создан компилятор подмножества языка Scheme в язык Forth. Использование полноценного аналога данного компилятора позволит избежать использования на стадиях построения компиляторов высокоуровневых языков языка C, однако, в то же время прямая реализация языка Scheme на Forth является достаточно трудозатратной из-за абсолютно разной структуры этих языков. Таким образом, можно сделать вывод о том, что данная задача не является достаточно эффективным шагом построения каскада компиляторов.

В процессе разработки удалось выявить основные преимущества каждого из языков, что позволяет сделать вывод о возможных сферах применения каждого из них, а также возможности добавления преимуществ одного языка в другой и последующего создания нового языка.

Список литературы

1. Guide: PLT Scheme [Электронный ресурс] — Режим доступа: http://download.plt-scheme.org/doc/html/guide/index.html
2. Starting FORTH by Leo Brodie [Электронный ресурс] — Режим доступа: http://www.forth.com/starting-forth/index.html
3. Scheme — Formal syntax and semantics [Электронный ресурс] — Режим доступа: http://www.cs.indiana.edu/scheme-repository/R4RS/r4rs_9.html
4. Revised(5) Report on the Algorithmic Language Scheme [Электронный ресурс] — Режим доступа: http://people.csail.mit.edu/jaffer/r5rs_toc.html
5. Wilfred/Minimal-scheme · GitHub [Электронный ресурс] — Режим доступа: https://github.com/Wilfred/Minimal-scheme
6. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман Компиляторы. Принципы, технологии и инструментарий. — М.: Вильямс, 2014.
7. An Incremental Approach to Compiler Construction [Электронный ресурс] — Режим доступа: http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf
8. Gforth Manual [Электронный ресурс] — Режим доступа: http://www.delorie.com/gnu/docs/gforth/gforth_toc.html

Приложения

Приложение 1. Список реализованных функций языка Scheme

```
-
*
/
+
<
<=
=
>
>=
abs
acos
and
apply (только для списков)
asin
atan
begin
boolean?
car
cdr
char?
char<?
char<=?
char=?
char>?
char>=?
char->integer
char-upcase
cons (результат: список,
первый аргумент - голова,
второй - хвост)
cos
cosh
display
eq?
exp
expt
floor
for-each (только для унарных
функций)
if
integer?
integer->char
list
list?
list->string
log
make-string
make-string
map (только для унарных
функций)
max
min
negative?
newline
not
null?
number?
or
pi
positive?
quotient (только для целых
чисел)
real?
remainder (только для целых
чисел)
round
sin
sinh
string
string?
string<?
string<=?
string=?
string>?
string>=?
string->list
string-append
string-length
string-ref
string-set!
tan
tanh
zero?
```