Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра алгоритмических языков

Вересов Алексей Кириллович

Поддержка вычислительной модели Лиспа для языка Си

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:
к.ф.-м.н., доцент
А. В. Столяров

Москва, 2022

Аннотация

В работе рассматривается созданная автором реализация вычислительной модели языка Лисп, написанная на языке Си и предназначенная для применения в проектах, основным языком которых является Си. Реализация позволяет записывать S-выражения языка Лисп в форме суперпозиции вызовов функций Си; такая запись оказывается достаточно наглядной и лаконичной, не требуя при этом дополнительного препроцессирования исходного кода.

Оглавление

1 Введение

1.1 Мультипарадигмальное программирование

Различные задачи удобно решать разными методами, и для каждого языка набор предоставляемых методов отличается, порой существенно. При таких значительных отличиях языки относят к разным парадигмам, будь то логическое, объектно-ориентированное, функциональное или иное программирование. При этом под парадигмой понимается подход к решению задач определёнными методами.

Разные парадигмы показали себя эффективными в решении разных задач. Например, ООП успешно используется для построения графических интерфейсов, логическое программирование — в переборных задачах, а функциональное — в символьных вычислениях. Во многих проектах разные подзадачи наиболее очевидно решаются в разных парадигмах. Так, почти любой проект, использующий базы данных, сочетает в себе разные парадигмы для основной работы и для обращения к базе данных. Приведённый пример показывает необходимость сочетания различных парадигм в рамках одного проекта.

Это же наблюдение подтверждается статистикой использования языков программирования в проектах с открытым исходным кодом. Исследование [7] показало для языков с наибольшей популярностью повышенную долю многоязыковых проектов, превышающую 20%.

В статье [1] приведены и проанализированы следующие подходы к решению этой проблемы:

- применение нескольких систем программирования;
- создание нового языка;
- расширение существующего языка;
- непосредственная интеграция.

Из них остановимся на методе **непосредственной интеграции** — он означает моделирование средствами базового языка синтаксиса и семантики

альтернативных языков. В качестве достоинств этого метода авторы выделяют неизменность базового языка, лёгкость взаимодействия частей программы, написанных в разных парадигмах, и относительную простоту реализации.

1.2 Выбор языков для непосредственной интеграции

Язык **Си** является одним из самых популярных языков программирования [23, 25, 7], широко используется в системном, научном и прикладном программировании, а также в разработке встроенного программного обеспечения, обладает большим числом поддерживаемых различными компиляторами целевых платформ. Из этого следует особая привлекательность Си в качестве базового языка, подтверждаемая высокой долей многоязыковых проектов среди проектов на Си — свыше 9% [7].

Отметим, что язык препроцессора является отдельным от самого Си языком, в частности не учитывающим систему типов последнего. Это и другие свойства препроцессора обеспечили ему репутацию инструмента, активное использование которого приводит к повышенному числу ошибок [16]. В силу этого наблюдения предлагаемый метод не опирается на препроцессор.

Несмотря на высокую актуальность языка Си в качестве базового языка для непосредственной интеграции, существует относительно малое число исследований по теме. К их числу можно отнести библиотеку Cello [10], активно использующую препроцессор для предоставления модели динамического объектно-ориентированного программирования для Си. Непопулярность выбора языка Си в качестве базового можно объяснить, в частности, ограниченностью его возможностей. В языках с более гибким синтаксисом известен ряд проектов, например для C++: InteLib [2], Castor [17], LC++ [15], FC++ [14]; для Java известен проект поддержки функционального программирования Pizza [18]; для Haskell: Haskell Rules [12], исследование возможности встраивания вычислительной модели языка Пролог [22], а также постфиксных языков [19].

Языки семейства **Лисп** сильно отличаются от языка Си по своим возможностям и предоставляемой парадигме. Они известны своей гибкостью и динамической природой, тогда как Си известен своей эффективностью и близостью к машине. Это различие отражается и в области применения — языки семейства Лисп особенно активно используются в веб-разработке для написания серверной части приложений, а также для разработок в области искусственного интеллекта. На этом контрасте задача непосредственной интеграции Лиспа в Си выглядит особо актуально и требует разработки

нового метода использования возможностей языка Си для обеспечения поддержки чужеродных ему выразительных конструкций.

Актуальность задачи дополнительно подтверждается высокой популярностью языков этого семейства. Несмотря на их отличия от языков доминирующей императивной парадигмы, около 2% программистов профессионально используют эти языки, что выше аналогичного показателя таких языков, как Хаскель, Паскаль, Erlang, Ада и Фортран [7, 23].

Ещё одна причина привлекательности диалектов Лиспа в качестве языков для непосредственной интеграции заключается в их минималистичном в своей основе синтаксисе. В языках этого семейства программы традиционно записываются как данные, а именно как данные, представляющие их синтаксическое древо. Причём для записи программ используется префиксная нотация, а скобки позволяют описывать списки произвольной длины и использовать функции с переменным числом параметров.

1.3 Постановка задачи

Целью работы является реализация вычислительной модели Лиспа в рамках языка Си методом непосредственной интеграции. Для достижения поставленной цели необходимо решить следующие задачи:

1. разработать способ представления данных Лиспа, удовлетворяющий требованиям наглядности и компактности их записи средствами языка Си;

2. реализовать библиотеку, исполняющую записанные разработанным способом программы на диалекте Лиспа;

3. на основе полученного исполнителя создать интерпретатор программ, записанных в классическом синтаксисе Лиспа;

4. используя для первичной раскрутки созданный интерпретатор, написать на избранном диалекте Лиспа самоприменимый транслятор, переводящий программы из традиционной записи Лиспа в запись на Си с использованием разработанного способа.

2 Вычислительная модель Лиспа

В Лиспе и его диалектах используется единое представление программ и данных — в виде S-выражений. S-выражение программы подаётся функции *вычисления*, возвращающей в качестве результата также S-выражение.

2.1 Представление данных

Изначально определённые создателем языка Лисп Джоном Маккарти [13], **S-выражения** используются как базовая структура данных в Лиспе и его потомках, а также в проектах, не связанных с Лиспом непосредственно, но заимствующих эту концепцию в силу её простоты и гибкости. Примерами могут служить WebAssembly, KiCad, Emacs, Racket, Scheme [11], Common Lisp [24], Curl [26] и другие.

S-выражение является **атомом** (идентификатором, константой некоторого типа и т.п.) или же **точечной парой** — упорядоченной парой S-выражений. Своё имя атом получил в силу того, что он не подлежит дальнейшему разбору в рамках считывания S-выражения, а точечная пара называется так из-за своей классической записи в скобках с разделяющей вложенные S-выражения точкой. Можно записать такое определение S-выражений в виде БНФ [5]:

```
<S-expression> ::= <atom> | <dotted pair>
<dotted pair>  ::= "(" <S-expression> "." <S-expression> ")"
```

Диалекты Лиспа поддерживают достаточно широкое множество типов атомов. При наличии таковых сущностей в языке к атомам относятся числовые, логические, символьные и строковые константы. Отдельным случаем атома является **символ** — наиболее близкой аналогией которому из более распространённых языков будет идентификатор — в частности, символы используются для именования переменных и вычислимых форм. Вызов формы с некоторым набором параметров приводит к её вычислению, что возвращает результат. Сами объекты форм также считаются атомами. Кроме того, некоторые диалекты поддерживают дополнительные структуры данных, такие как массивы, записи или хеш-таблицы; в модели S-выражений

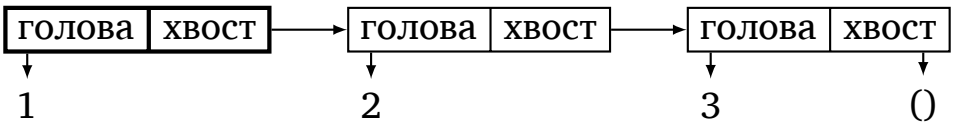


Рис. 2.1: Представление списка (1 2 3)

такие объекты тоже относятся к атомам. В диалекте Лиспа под названием Arc возможно установить произвольный символ в качестве типа объекта, в частности возможно динамически расширять набор типов атомов [21].

Левое подвыражение точечной пары будем называть **головным** или **головой**, а правое — **хвостовым** или **хвостом**. Исторически им соответствуют названия CAR и CDR.

Списком в терминологии S-выражений называется *пустой список* или же точечная пара, хвост которой является списком. Можем записать это определение в виде БНФ:

```
<list> ::= <empty list> | "(" <S-expression> "." <list> ")"
```

Так как **пустой список**, по определению точечной пары, должен быть S-выражением и не является точечной парой, то он является атомом. Этот атом обычно записывают в виде `NIL` или же `()`.

Распространена сокращённая запись списка, при которой все элементы списка, кроме оконечного пустого списка, указываются через пробел и оборачиваются в скобки. Например, следующие записи эквивалентны, то есть порождают в памяти один и тот же объект, изображённый на рисунке 2.1:

```
(1 2 3)
(1 . (2 . (3 . ())))
```

Точечный список — S-выражение, структурно идентичное списку, но заканчивающееся не пустым списком, а другим атомом [2]. Для таких конструкций также предусмотрена сокращённая запись, обобщающая запись точечной пары, при которой последовательность S-выражений в скобках имеет перед последним своим элементом точку. При этом образуемое S-выражение можно описать как список L, построенный из всех элементов кроме стоящего за точкой, а этот последний элемент заменяет пустой список в конце списка L. Приведём пример эквивалентных записей, аналогичный примеру сокращённой записи списков, но где вместо оконечного пустого списка стоит атом 4:

```
(1 2 3 . 4)
(1 . (2 . (3 . 4)))
```

Отметим, что детали синтаксиса записи S-выражений в различных диалектах Лиспа могут существенно отличаться — вплоть до, например, изменения скобок с круглых на фигурные [26].

2.2 Исполнение программ

Программа на Лиспе представляет собой последовательность *вычисляемых* (evaluated) S-выражений, а выполнение программы состоит в *вычислении* этих выражений.

Пусть вычисляется некоторое S-выражение, тогда по определению оно может быть точечной парой или атомом, при этом атом может быть одного из множества типов.

Особо выделим возможность атома быть **символом**, при вычислении которого возвращается ассоциированное с ним значение. Языки семейства Лисп обычно допускают большую гибкость в записи символов, позволяя использовать в них такие спецзнаки, как `+`, `-`, `?` и др.

К записываемым явно **константам** обычно относят числа (`1`, `9.99`, `73`, …), текстовые символы (`#\a`, `#\@`, …) и строки (`"sunrise"`, `"Dijkstra"`, …). Атомы константных типов вычисляются сами в себя. Отметим, что символ, ассоциированным значением которого является он сам, также вычисляется сам в себя и часто тоже называется константой.

Некоторые атомарные значения не могут быть записаны явно, а являются результатом получения значения предопределённого символа или результатом вычисления формы. Вычисление этих атомов варьируется в различных диалектах Лиспа и их реализациях. Так, в описании языка Scheme вычисление функционального атома остаётся неопределённым [11], а реализации разнятся в своём поведении — MIT Scheme вычисляет такой атом в самого себя, а Chicken Scheme выдаёт ошибку.

В случае точечной пары она в основном рассматривается как список, при этом первый элемент обозначает вызываемую форму, а последующие — её аргументы. Для записи вызовов в Лиспе используется префиксная нотация, при которой $2 + 2$ становится:

```
(+ 2 2)
```

Наличие скобок позволяет использовать в Лиспе вариадические формы. Так, функция сложения обычно определена не для двух, а для произволь-

ного числа аргументов, и сумму чисел от трёх до семи можно записать так:

```
(+ 3 4 5 6 7)
```

Формы делятся на функции, макросы и спецформы. При вызове **функции** вычисляются все аргументы, после чего исполняется сама функция и результат этого исполнения возвращается в качестве результата вызова. При вызове **макроса** аргументы макроса обрабатываются им не будучи предвычисленными, а результат работы макроса подвергается вычислению в контексте вызова. К **спецформам** относят все остальные случаи.

Также формы классифицируются на **встроенные** в систему и **пользовательские**, то есть определяемые в ходе исполнения. Среди встроенных форм, как следствие, должны присутствовать такие, что позволяют определять пользовательские формы. При этом обычно пользовательская форма определяется двумя частями — своим *заголовком*, то есть перечнем формальных параметров, и *телом*, то есть S-выражением. В случае определения пользовательской функции фактические аргументы будут вычислены в контексте вызова, после чего подставлены вместо соответствующих формальных в тело функции, которое затем будет исполнено в контексте определения функции. В случае макросов аналогично подставляются в тело фактические параметры вместо формальных, но в том виде, в каком они присутствовали в вызове, после чего в контексте определения макроса вычисляется полученное подстановками тело, а результат этого вычисления вычисляется ещё раз, но уже в контексте вызова.

Отметим, что ни при перечислении параметров форм, ни при установке значений символам не указываются типы. Это происходит оттого, что Лисп — язык с динамической типизацией, и тип является свойством не имени связанного со значением, но самого этого значения. Как следствие, в языках семейства Лисп обычно доступны функции, проверяющие тип переданного им значения.

3 Использование средств языка Си

3.1 Поддержка динамической типизации

Лисп и другие языки, которым необходимо динамически определять информацию о типах, будь то сам тип — как, например, в Smalltalk, или же адреса функций, реализующих методы — как в C++ при использовании ключевого слова `virtual`, вынуждены так или иначе хранить такую информацию во время исполнения своих программ.

Несмотря на отсутствие поддержки со стороны языка Си, в нём возможна реализация этого механизма несколькими способами. Сформулируем идею требуемого объекта: это должна быть сущность, хранящая информацию о своём типе, а также интерпретируемые в соответствии с этим типом данные объекта. Наивной реализацией типа подобного объекта будет:

```
1 struct object {
2     enum objtype { objtype_1, /* ... */ objtype_n } type;
3     union objdata {
4         objdata_1 data_1;
5         /* ... */
6         objdata_n data_n;
7     } data;
8 };
```

Такая реализация не позволяет непосредственный доступ к данным объектов, что затрудняет взаимодействие с объектами при помощи стандартных средств языка Си. Кроме того, при добавлении нового типа пользователь будет вынужден изменять код системы, реализующей динамическую типизацию таким способом, все типы при этом должны быть известны уже на момент компиляции.

Решим проблему добавления новых типов через `void *`. Указатели этого типа могут приводиться к любому другому, за исключением указателей на функцию [4], в силу чего позволяют пользователю вводить новые типы без необходимости изменять код системы динамической типизации. Эта же техника успешно применяется при построении систем с функциями

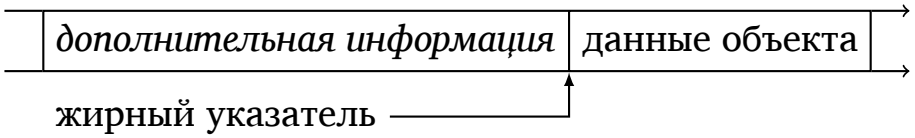


Рис. 3.1: Расширение объекта техникой жирного указателя

обратного вызова, где пользовательские данные обычно передаются именно через `void *`. Из недостатков такого подхода можно назвать необходимость разыменования указателя даже в случае данных простого типа, как `int`.

Теперь решим проблему опосредованного доступа к данным объекта. Предыдущее решение подразумевает, что все данные объектов доступны по указателю. Одна из техник неявного расширения указателя дополнительной информацией известна в англоязычной литературе под названием **fat pointers** [8], что дословно можно перевести как **жирные указатели**.

При использовании жирных указателей подразумевается, что дополнительная информация об объекте находится перед основными данными, а сам жирный указатель содержит адрес этих основных данных, как на рисунке 3.1. Рассмотрим для примера расширение строк языка Си дополнительной информацией об их длине по технике жирных указателей. Запишем обобщённую реализацию жирных указателей:

```
1  extra *obj_extra(void *obj)
2  {
3      return (extra *)obj - 1;
4  }
5  void *obj_new(extra *info, int size)
6  {
7      extra *res = malloc(sizeof(extra) + size);
8      memcpy(res, info, sizeof(extra));
9      return res + 1;
10 }
11 void obj_free(void *obj)
12 {
13     free((extra *)obj - 1);
14 }
```

Здесь типом `extra` обозначен тип дополнительной информации, размещаемой перед основными данными. Поскольку в рассматриваемом примере требуется расширить строки информацией об их длине, будем считать, что `extra` задано синонимом `int`, используя `typedef`. Теперь опишем создание требуемой строки:

```
char *fatstr_new(const char *cstr)
{
    int len = strlen(cstr);
    char *res = obj_new(&len, len + 1);
    memcpy(res, cstr, len + 1);
    return res;
}
```

Приведём также пример использования полученной системы:

```
int main()
{
    char *fatstr = fatstr_new("Hello, World!");
    puts(fatstr);
    printf("%ld = %d\n", strlen(fatstr), *obj_extra(fatstr));
    printf("second character is '%c'\n", fatstr[1]);
    obj_free(fatstr);
    return 0;
}
```

Как видно на этом примере, использование жирных указателей позволяет расширить доступный по указателю объект произвольной информацией, сохраняя частичную совместимость с оригинальными функциями над объектами такого типа. Существенное ограничение на такие функции — требование инвариантности дополнительной информации объекта относительно их применения. Так, при расширении строк информацией об их длине оригинальные функции, меняющие длину поданной им строки, нарушат согласование дополнительной информации и данных в объекте, вследствие чего возможны ошибки. Как правило, несложно модифицировать такие оригинальные функции для работы с расширенными объектами или же написать функцию-обёртку, восстанавливающую после использования оборачиваемой функции согласованность дополнительной информации и данных изменяемого объекта.

Для реализации динамической типизации с помощью приведённой техники достаточно задать `extra` синонимичным типу, описывающему всю требуемую от типа информацию. В случае S-выражений можно оставить `int` и завести перечисление (`enum`) всех требуемых типов: точечной пары, данные для которой будут парой жирных указателей на головное и хвостовое подвыражения; пустого списка, данные которого не рассматриваются программой; а также всех остальных требуемых типов атомов — будь то целые числа или символы, данные при этом должны интерпретироваться соответственно. Однако без поддержки сборки мусора исполнение про-

грамм на Лиспе приведёт к утечке памяти, а потому станет невозможно использовать разработанную систему на практике. Таковая поддержка потребует расширить хранимую дополнительную информацию об объекте. В разработанной системе для сборки мусора используется алгоритм пометок (*mark-and-sweep*), описанный в соответствующем разделе. При этом итоговый тип дополнительной информации имеет вид:

```
typedef struct extra {
    int mark;
    int type;
} extra;
```

При известном `extra` можно конкретизировать и упростить `obj_new`:

```
void *obj_new(int type, int size)
{
    extra *res = malloc(sizeof(extra) + size);
    res->mark = 0;
    res->type = type;
    return res + 1;
}
```

3.2 Поддержка функциональных объектов

При описании исполнения программ на Лиспе мы выделили встроенные формы, которые должны быть также доступны как S-выражения. Чтобы упростить расширение системы новыми встроенными формами со стороны пользователя, можно представить встроенные формы как функции Си. В наиболее общем виде встроенная форма принимает и возвращает S-выражение, где принимаемое S-выражение соответствует хвосту вычисляемой точечной пары, у которой головой оказалась встроенная форма. Тогда можно так описать тип указателя на функцию, реализующую встроенную форму:

```
typedef void *(*embedded_form)(void *tail);
```

Скрытые при обычном использовании системы детали её реализации определим в отдельном заголовочном файле — представляется целесообразным использовать открывающиеся при этом возможности в рамках описания встроенных форм.

Однако указатель функционального типа не может быть приведён к типу `void *`, так как не является указателем на объект [4]. Поэтому непосредственное использование определённых таким образом функций в качестве S-выражений невозможно.

Решением этой проблемы может служить организация жирного указателя на указатель на функцию:

```
1 embedded_form *new_embedded_form(embedded_form f)
2 {
3     embedded_form *res = obj_new(type_embedded_form, sizeof(f));
4     *res = f;
5     return res;
6 }
```

Пусть задана функция `f` со следующим заголовком:

```
void *f(void *tail);
```

— тогда создание соответствующего S-выражения и его явный вызов можно записать так:

```
1 void *sexpr = new_embedded_form(f);
2 (*(embedded_form *)sexpr)(tail);
```

3.3 Поддержка сокращённой записи списков

Язык Си позволяет передавать в функцию произвольное число параметров — такие функции называются **вариадическими**. В их заголовках после некоторого количества обычных параметров ставится многоточие:

```
void variadic_function(int arg_1, /* ... */ int arg_n, ...);
```

Возможно также определить функцию, безразличную к своим аргументам — в этом случае в заголовке функции список аргументов оставляют пустым. Однако, без знаний о реализации вызова функции в конкретной системе получить доступ к аргументам в таком случае невозможно, поэтому в дальнейшем тексте рассматривать их не будем.

Например, вариадической является стандартная функция `printf`. Первый аргумент этой функции указан в её заголовке явно, а остальные передаются через многоточие. Чтобы получить к ним доступ, используется функционал стандартной библиотеки языка Си, описанный в заголовоч-

ном файле stdarg.h. Стандартная библиотека не предоставляет средств для определения типов и количества фактически переданных аргументов, но поддерживает считывание очередного аргумента определённого типа через макрос va_arg. Функция printf использует этот подход, разбирая форматную строку, переданную первым аргументом. При встрече сочетания символов, обозначающих команду печати данных указанного типа, происходит считывание очередного аргумента и его печать в соответствии с типом. Таким образом информация о типах и количестве аргументов извлекается из первого аргумента, то есть форматной строки. Приведём пример вызова printf:

```
printf("Coordinates: (%d, %d)\n", x, y);
```

Используем возможность вариадических функций принимать произвольное число аргументов для построения функции, строящей списки в соответствии с их сокращённой записью в традиционном синтаксисе Лиспа. При считывании очередного аргумента вариадической функции требуется указать его тип. В рамках разработанной системы представления S-выражений все они обладают одним типом — void *. Это значит, что требуется передать только информацию о числе аргументов, но не об их типах.

По аналогии с определением функции printf можно передавать число аргументов в качестве первого аргумента функции построения списков, которую обозначим за L:

```
L(n, obj_1, obj_2, /* ... */ obj_n);
```

При использовании такого подхода необходимо производить дополнительные изменения кода при добавлении элементов в список. Эти изменения требуют найти начало списка, что может быть затруднительно при большом количестве элементов в нём, а кроме того требуют помнить количество добавленных элементов и складывать его с уже имеющимся размером. Использование предложенной записи списков в сравнении с использованием классической существенно повысило бы трудоёмкость написания кода и вероятность допустить ошибку, отчего можно считать, что необходимость такого рода изменений неприемлема.

Решением может служить введение маркера конца, помимо скобки, закрывающей список аргументов функции. В силу описанных ограничений языка Си и его стандартной библиотеки, маркер конца сам должен быть аргументом функции, а именно — последним из них.

Поскольку было решено обрабатывать S-выражения через жирные указатели, разумно использовать в качестве маркера конца списка нулевой указатель, ведь ни один объект в предлагаемой системе не будет иметь

такое значение. При этом отметим, что `NIL`, представляемый атомом особого типа «*пустой список*», сам является объектом, и потому его указатель не является нулевым.

Пусть функция, создающая список, именуется `L`, тогда использование разработанного решения будет выглядеть так:

```
L(obj_1, obj_2, /* ... */ obj_n, 0);
```

Что соответствует следующему S-выражению, записанному на языке Лисп:

```
(obj_1
 obj_2
 ; ...
 obj_n)
```

Теперь продумаем поддержку сокращённой записи **точечных списков**. Пусть идентификатор `o` используется как обозначение точки, тогда для построения точечного списка можно использовать такой вызов:

```
L(obj_1, obj_2, /* ... */ o, obj_n, 0);
```

— ноль в этом случае избыточен, так как точка в записи списка может появиться только один раз, перед последним элементом. Однако для единообразия записи предлагается оставлять ноль, играющий в таком случае вместе с закрывающей скобкой роль закрывающей скобки в рамках классического синтаксиса Лиспа.

Чтобы реализовать такое поведение, достаточно гарантировать уникальность адреса, связанного с `o`. Это свойство можно будет использовать при реализации функции `L`: встретив среди аргументов этот адрес, функция извлечёт очередной аргумент, вставит его в хвост последней точечной пары конструируемого списка и завершит работу.

Уникальным является, например, адрес глобальной переменной — достаточно присвоить его в указатель `o`. Однако, так как сама переменная в описываемом случае не имеет никакого применения, следует завести её в качестве статической и оставить доступной только в рамках библиотеки, предоставляющей Лисп-вычислитель.

4 Лисп-вычислитель

В этой главе описан реализуемый Лисп-вычислитель, который мы будем называть CSX — от *C S-expressions*.

4.1 Система типов

По определению S-выражений требуется наличие типа точечной пары, обозначенного в CSX символом `pair`. Данными этого типа, доступными по жирному указателю, является структура из двух указателей на S-выражения, `head` и `tail`. Так, если эти идентификаторы связаны с соответствующими головным и хвостовым подвыражениями точечной пары, её создание можно записать как `L(head, o, tail, 0)`.

Отдельным типом, состоящим ровно из одного значения и потому не имеющим данных, было решено сделать пустой список, доступный по символу `nil`. Получить его в рамках программы на Си можно, например, вызовом `L(0)`.

Для поддержки вычислений имеется тип `int`, обозначающий целые числа. Его данные имеют тип `int` языка Си, а создание S-выражения этого типа происходит вызовом функции `I`, например `I(37)`.

Имеется тип `str`, данные которого есть оканчивающаяся нулём строка, а именно динамический массив языка Си, состоящий из значений типа `char`. При создании строки её содержимое копируется в выделенную на куче область памяти. Примером вызова конструктора этого типа может служить `T("wind")`.

Имеется тип `symbol`, представляющий символы Лиспа. Его данными, так же как и у `str`, является строка, однако принципиально отличается её способ хранения и создания. Система организует таблицу символов, представляющую собой массив указателей на все доступные на данный момент в системе символы, как на рисунке 4.1. При создании очередного символа, например вызовом `S("a")`, происходит поиск по таблице, при этом переданная конструктору строка сравнивается со строкой очередного символа. Если требуемый символ был найден в таблице, то создание символа не происходит, а возвращается указатель из таблицы на уже су-

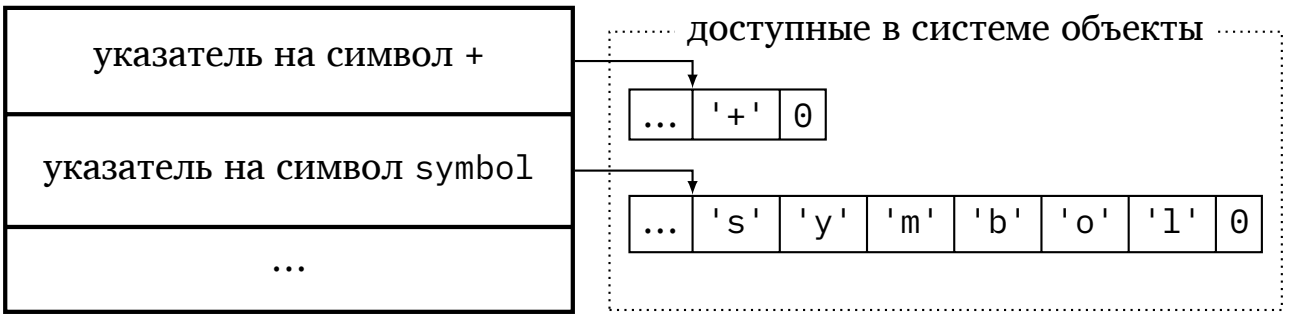


Рис. 4.1: Организация таблицы символов

ществующий. Если же символ с требуемым именем не оказался доступен в системе, то происходит создание нового символа и занесение указателя на него в таблицу. Такое решение позволяет производить сравнение символов по их указателям, то есть независимо от длины соответствующих строк. Это свойство значительно ускоряет поиск значений символов.

Имеется тип `embedded_form`, которым обладают все встроенные формы CSX и данными которого является указатель на функцию языка Си принимающую и возвращающую S-выражение.

Последние два типа отражают два варианта определения пользовательской формы. Это типы `lambda` и `macro`, соответствующие функции и макросу. Данные этих типов устроены одинаково: это тройки из списка формальных параметров, S-выражения тела и контекста определения. Контекст при этом также является S-выражением, а именно списком фреймов. Фрейм в свою очередь тоже список, но состоящий из точечных пар, где голова — это некоторый символ, а хвост — его определение, то есть S-выражение, на которой символ будет заменён при вычислении. Можем сформулировать тип данных обоих этих типов так:

```
1 struct form_data {
2     void *params;
3     void *body;
4     void *context;
5 };
```

Явные конструкторы значений этих типов, доступные из Си, не предусмотрены — для создания таковых значений следует воспользоваться встроенными формами, доступными по символам `lambda` и `macro`.

4.2 Модель исполнения

Запишем вычисление и вывод суммы 37 и 73 с использованием CSX:

```
printf("%d\n", *(int *)E(L(S("+"), I(37), I(73), 0)));
```

— здесь `E` является функцией, принимающей S-выражение, вычисляющей его и возвращающей полученный результат.

Опишем, как происходит вычисление объектов каждого из типов.

Вычисление объектов типа `str`, `int`, `base`, `lambda`, `macro` и `nil` возвращает их самих без изменений.

Для объектов типа `symbol` возвращается связанное с символом в текущем контексте значение. Контекст при этом является списком из фреймов. Фрейм представляется списком точечных пар, где голова — это символ, а хвост — соответствующее значение. Такая организация позволяет легко организовать лексическое связывание, тем самым не допуская возникновения *фунарг-проблемы* [3].

Объекты типа `cons` воспринимаются как *вызовы форм*. При этом головное подвыражение пары всегда вычисляется, и в зависимости от типа полученного выражения дальнейшее вычисление организуется по-разному.

Если получен объект типа `base`, то хвостовое подвыражение вызова передаётся в качестве аргумента функции языка Си, доступной по указателю в данных объекта встроенной формы. Результат вызова этой функции является результатом вызова встроенной формы.

Если вызывается `lambda`, то вычисляется каждый элемент хвостового подвыражения, воспринимаемого как список. Элементы полученного списка сопоставляются с соответствующими символами из списка формальных параметров. Список формальных параметров может быть точечным, тогда все оставшиеся фактические аргументы, кроме сопоставленных, формируют список, который будет связан с символом, являющимся последним элементом точечного списка формальных параметров. Такая реализация позволяет описывать вариадические пользовательские функции. После сопоставления аргументов сохранённый контекст функции расширяется фреймом из полученных сопоставлений и происходит исполнение тела функции в полученном контексте. Полученный результат возвращается непосредственно.

Если вызывается `macro`, то происходит процесс, аналогичный вызову `lambda`, с той разницей, что список фактических параметров сопоставляется с формальными без вычисления, но результат вычисления тела в расширенном аргументами контексте вычисляется повторно, уже в контексте вызова.

4.3 Сборка мусора

Ресурсы компьютера конечны, в частности ограничена оперативная память, доступная программе. Однако классический Лисп и разработанный вычислитель не предоставляют программисту способов явного контроля потребления памяти — во многом это ограничение обусловлено чрезмерной трудностью ручного управления памятью в рамках вычислительной модели Лиспа. Разрешить это противоречие призван *сборщик мусора*, изначально придуманный создателем Лиспа Джоном Маккарти [13]. Это неявная часть программы, отвечающая за освобождение памяти, занятой недоступными из остальной программы объектами.

Существуют разные подходы к организации сборки мусора. Доминирующими являются алгоритмы, основанные на подсчёте ссылок (*reference counting*) и алгоритме пометок (*mark-and-sweep*).

При **подсчёте ссылок** каждый объект снабжается информацией об имеющихся на него ссылках из других объектов. Уменьшение счётчика ссылок до нуля означает отсутствие к нему доступа из системы, в силу чего занимаемая им память может быть освобождена. При этом для всех объектов, на которые ссылался освобождаемый, следует уменьшить счётчик ссылок и проверить, не стал ли он равен нулю. При создании нового объекта, до тех пор пока он непосредственно достижим, его счётчик ссылок устанавливается в единицу. При нарушении непосредственной достижимости объекта счётчик декрементируется. Если созданный объект содержит ссылки на уже присутствовавшие в системе объекты, их счётчики инкрементируются.

Важными достоинствами подсчёта ссылок являются простота реализации и незамедлительное освобождение памяти недоступных объектов, что позволяет обеспечить предсказуемую отзывчивость программы. Существенным недостатком является возможность утечки памяти ввиду неспособности наивной реализации распознавать циклы ссылок, то есть в случае, когда объект прямо или косвенно ссылается сам на себя. Известны варианты алгоритма [6, 20], при которых происходит распознавание циклов, однако они значительно усложняют реализацию и обладают собственными недостатками.

Алгоритм пометок является изначально представленным в рамках Лиспа алгоритмом сборки мусора. Алгоритм периодически вызывается системой для очистки от накопленных недостижимых объектов. Все объекты системы снабжаются битом пометки об их достижимости — при запуске алгоритма все объекты помечаются как недостижимые. Затем для каждого непосредственно достижимого объекта вызывается функция пометки, устанавливающая бит в состояние достижимости. Для каждой ссылки, хранящейся в помечаемой объекте, рекурсивно вызывается функция по-

метки. Когда все объекты помечены, все объекты системы проверяются на состояние бита пометки и освобождаются, если остались недостижимыми.

В отличие от алгоритма подсчёта ссылок, алгоритм пометок способен распознавать циклы. Однако алгоритм пометок требует остановки системы на время своей работы, которое порой может быть существенным. Это приводит к проблемам с отзывчивостью у программ, использующих этот сборщик мусора. Эдсгер Дейкстра, Лесли Лэмпорт и другие в работе [9] описывают алгоритм трёхцветных пометок, лишённый этого недостатка и работающий параллельно с управляемой им программой.

В рамках поставленной задачи наиболее целесообразным представляется использование наивного алгоритма пометок, который и был реализован в CSX. Непосредственно достижимыми объектами системы считаются объекты стека временных объектов вызовов форм, а также текущий контекст. При этом используется свойство определения контекста, по которому он является S-выражением, что позволяет не обходить входящие в него объекты явно, но положиться на рекурсию функции пометки. В случае CSX она носит имя `deepmark`, а рекурсивный вызов происходит только для типов `pair`, `lambda` и `macro`:

```
1  static void deepmark(void *p)
2  {
3      if (mark(p)) return;
4      setmark(p);
5      if (type(p) == type_pair) {
6          deepmark(head(p));
7          deepmark(tail(p));
8      } else if (type(p) == type_lambda ||
9                 type(p) == type_macro) {
10         form_data *fd = p;
11         deepmark(fd->params);
12         deepmark(fd->body);
13         deepmark(fd->context);
14     }
15 }
```

Как было упомянуто, полный проход алгоритма пометок может занимать существенное время, а потому его исполнение при каждом изменении системы вызвало бы заметные задержки в работе программы. Решением может служить введение некоторого ограничения — в случае CSX было принято решение запускать алгоритм сборки мусора только тогда, когда объём занимаемой объектами системы оперативной памяти превышает

объём объектов, оставшихся после прошлого прохода алгоритма, более чем в два раза. Единственно возможным местом запуска сборщика мусора является начало функции вычисления S-выражения.

4.4 Встроенные формы

Укажем встроенные формы по символам, через которые они доступны.

`cons`: функция, возвращающая точечную пару из двух своих аргументов.

`car`: функция, ожидающая единственным своим аргументом точечную пару и возвращающая её головное подвыражение.

`cdr`: функция, ожидающая единственным своим аргументом точечную пару и возвращающая её хвостовое подвыражение.

`set-car!`: функция, принимающая первым аргументом точечную пару и изменяющая её, устанавливая в качестве головного подвыражения второй аргумент.

`set-cdr!`: функция, принимающая первым аргументом точечную пару и изменяющая её, устанавливая в качестве хвостового подвыражения второй аргумент.

`str`: функция, принимающая символ или последовательность чисел. Если аргументом является символ, то возвращает соответствующую ему строку, если передана последовательность чисел, то каждое число обрезается до байта и из них формируется новая строка.

`symbol`: функция, принимающая единственным своим аргументом строку и возвращающая соответствующий её символ.

`len`: функция, принимающая единственным аргументом строку или список и возвращающая их длину.

`define`: спецформа, устанавливающая символу значение. При этом первый аргумент (символ) не вычисляется, а второй вычисляется в контексте вызова и используется в качестве устанавливаемого значения.

`defined?`: функция, принимающая единственным аргументом символ. Если этот символ определён, то возвращается единица, иначе — пустой список. Эта функция полезна при расширении системы, если заранее неизвестен набор встроенных форм.

`quote`: спецформа, возвращающая свой единственный аргумент невычисленным.

`eqv?`: функция, принимающая произвольное число аргументов. Если аргументы разного типа, возвращается пустой список. Если в качестве аргументов поданы числа и среди них есть хотя бы два неравных, то вернётся пустой список. Для других типов аргументов пустой список вернётся при наличии объектов с неравными представляющими их жирными указателями. Во всех остальных случаях возвращается единица.

`type`: функция, возвращающая символ, соответствующий типу её единственного аргумента.

`begin`: функция, возвращающая свой последний аргумент.

`if`: спецформа, принимающая произвольное число аргументов. Если передан только один аргумент, то он вычисляется в контексте вызова, и полученный результат возвращается. Иначе первый аргумент всё равно вычисляется в контексте вызова, и если получен пустой список, то второй аргумент пропускается, а все оставшиеся подаются в спецформу рекурсивно. Если же первый аргумент вычислился не в пустой список, то возвращается результат вычисления второго аргумента в контексте вызова — остальные аргументы при этом игнорируются.

`eval`: спецформа, вычисляющая свой аргумент дважды.

`exit`: функция без аргументов, завершающая исполнение программы.

`write-char`: функция, принимающая единственным аргументом число и выводящая текстовый символ, код которого равен переданному числу.

`read-char`: функция без аргументов, возвращающая в виде числа код считанного из ввода текстового символа.

`context`: функция без аргументов, возвращающая текущий контекст.

`new-context`: функция без аргументов, создающая и возвращающая новый контекст, в котором присутствуют все встроенные в CSX определения символов.

`+`, `*`: функции от набора чисел, возвращают их сумму и произведение соответственно.

`<`, `>`: функции от последовательности чисел, возвращают единицу, если переданная последовательность упорядочена соответственно по возрастанию и по убыванию. В противном случае возвращают пустой список.

`neg`: функция отрицания числа.

`div`: функция от двух чисел, возвращает целую часть от деления первого аргумента на второй.

`mod`: функция от двух чисел, возвращает остаток от деления первого аргумента на второй.

5 Трансляция диалекта Лиспа в Си

5.1 Базовый транслятор

Рассмотрим сначала задачу трансляции отдельного S-выражения для последующего использования полученного кода в программе на Си. При этом предполагается, что заголовочные файлы библиотеки CSX уже подключены.

Если транслятор встретил открывающую скобку, то он выводит `L(`. Если встречена закрывающая скобка, то транслятор выводит `0)`, добавляя ещё запятую, если скобка вложенная. Если встречена отдельно стоящая точка, то транслятор выводит `o,`.

Если встречены кавычки, то транслятор выводит `T("`. Затем повторяет на вывод ввод до закрывающих кавычек с учётом возможного экранирования. При закрытии выводит `")`.

Если встречен знак минус или цифра, то транслятор выводит `I(` и повторяет встреченный символ и все последующие цифры до разделителя. При закрытии выводит `),`.

Если был встречен иной символ, то транслятор выводит `S("` и встреченный символ. Затем повторяет на вывод ввод, по необходимости экранируя символы до разделителя. При завершении выводится `"),`.

Можно модифицировать транслятор для поддержки более человекочитаемого вывода, а именно вывода с корректными отступами и переносами строк при объявлении каждого нового элемента списка. Такой вывод используется в итоговом трансляторе.

Описанный транслятор иллюстрирует успех в представлении S-выражений в языке Си, ведь их перевод в такое представление прост и нагляден. Рассмотрим пример перевода следующего S-выражения:

```
((lambda (p) (+ (car p) (cdr p))) (37 . 73))
```

— которое после обработки транслятором станет следующим кодом на Си:

```
L(L(S("lambda"),
    L(S("p"),
      0),
    L(S("+"),
      L(S("car"),
        S("p"),
        0),
      L(S("cdr"),
        S("p"),
        0),
      0),
    0),
  L(I(37),
    o,
    I(73),
    0),
  0)
```

5.2 Трансляция символов

Теперь рассмотрим отдельно проблему трансляции символов. Решить эту задачу необходимо для того, чтобы построить компилятор с возможностью явного использования из Си сущностей, определённых в рамках Лиспа. Таким образом можно будет избежать использования конструктора символов `S` при каждом использовании символа в коде на Си.

Так как набор разрешённых к использованию знаков не совпадает у идентификаторов Си и символов Лиспа, требуется предложить схему перевода. В работе [2] была предложена модель, использующая чувствительность к регистру Си++ для обозначения букв латинского алфавита верхним регистром, а для обозначения спецсимволов используется нижний регистр. При этом делается предположение о нечувствительности к регистру интегрируемого диалекта Лиспа. Опишем модель трансляции символов, не опирающуюся на это предположение.

Для наглядности представления каждую букву латинского алфавита будем переводить в неё саму. Тогда, согласно определению идентификатора Си [4], остаётся возможность использовать цифры и нижнее подчёркивание. Цифры будем также переводить в самих себя, ведь идентификаторы в Лиспе, как и в Си, не могут начинаться с цифры, но могут иметь их в дальнейшем. А оставшееся нижнее подчёркивание будем использовать в качестве символа экранирования — между двумя нижними подчёркиваниями

будет находиться имя используемого спецсимвола. В силу популярности использования дефиса в качестве разделителя частей символа в Лиспе, дефис будем по умолчанию обозначать двойным нижним подчёркиванием.

С учётом выработанных правил покажем пример трансляции следующего S-выражения:

```
(set-car! x (+ 37 GREEN))
```

— что будет скомпилировано в:

```
1 L(set__car_excl_,
2   x,
3   L(_plus_,
4     I(37),
5     GREEN,
6     0),
7   0)
```

Несложно заметить, что при использовании такой системы трудно будет избежать конфликта идентификаторов в коде на Си, в том числе конфликтов с ключевыми словами Си, такими как `if`. Решить проблему можно снабдив образуемые идентификаторы префиксом. Так как выбор префикса и именований спецсимволов в общем произвольный, следует предоставить возможность настройки из кода на Лиспе поведения транслятора.

Введём символ %%%, который определим в базовом контексте формой, игнорирующей свой вход. При этом транслятор, встретив первым S-выражением в файле вызов этого символа, считает каждый его аргумент как свою настройку.

Так как в начале файла используется базовый контекст, то использование в первом в файле S-выражении в качестве самой внешней формы той, что доступна по %%%, не может иметь другого эффекта, кроме как её игнорирования. Это свойство позволяет транслятору не транслировать эту конструкцию. При дальнейшем изменении значения этого символа разницы поведения интерпретатора и компилятора не возникнет, так как транслятор по-особому реагирует только на вхождение этого символа в качестве самого внешнего в первом S-выражении файла.

Такой формат настройки транслятора одновременно удобен и для реализации, и для использования. Так как в Лиспе данные и код записываются одинаково, то запись настроек не потребует со стороны пользователя изучения нового формата записи данных.

Для решения описанных проблем трансляции имён можно ввести ниже-следующие настройки.

`(declare-char <char> <image>)` позволяет установить для указанного кода символа `char` строку `image`, которая будет использована для подстановки внутри подчёркиваний при трансляции этого символа. Строка `image` содержит только буквы латинского алфавита и цифры.

`(declare-name <name> <image>)` позволяет установить для указанного символа непосредственное правило трансляции.

`(prefix <name>)` позволяет установить префикс для оттранслированных имён, по умолчанию используется `"csx__"`.

5.3 Многомодульная компиляция

Для компиляции многомодульной программы требуется определить правила получения полноценных файлов исходного кода на Си. Рассмотрим компиляцию файла `modulename.csx`. Имя файла до расширения (modulename) запомним в качестве имени модуля по умолчанию. Отметим, что имя модуля является настройкой компилятора, кроме того, можно настроить имена получаемых заголовочного файла и файла исходного кода, а также набор публичных символов. Опишем структуру заголовочного файла `modulename.h`:

```
#ifndef CSX_MODULE_INCLUDED_modulename
#define CSX_MODULE_INCLUDED_modulename
extern void *csx_public_symbol; /* ... */
void *csx_load_modulename();
#endif
```

Теперь рассмотрим структуру файла исходного кода `modulename.c`:

```
#include "modulename.h"
#include <csx.h>
#include "additional_header.h" /* ... */
extern void *csx__external__symbol; /* ... */
void *csx__public__symbol; /* ... */
static void *csx__local__symbol; /* ... */
void *csx_load_modulename()
{
    csx__public__symbol = S("public-symbol"); /* ... */
    csx__local__symbol = S("local-symbol"); /* ... */
    E(/* S-expression */); /* ... */
    return E(/* last S-expression */);
}
```

Для объявления публичных символов следует воспользоваться настройкой транслятора `public-symbols`, для указания альтернативного имени модуля — `module-name`, для указания имён загловочного и исходного файла на Си — `module-header` и `module-source` соответственно. Дополнительные заголовочные файлы можно указать используя `additional-headers`. Внешние символы перечисляются в `external-symbols`.

Объявление символов в виде S-выражений позволяет ускорить исполнение, так как избавляет от необходимости повторно находить символ в таблице символов. Кроме того, символ, доступный непосредственно в виде переменной языка Си, проще использовать при ручном написании кода с использованием S-выражений в рамках их представления в CSX.

6 Эффективность решения

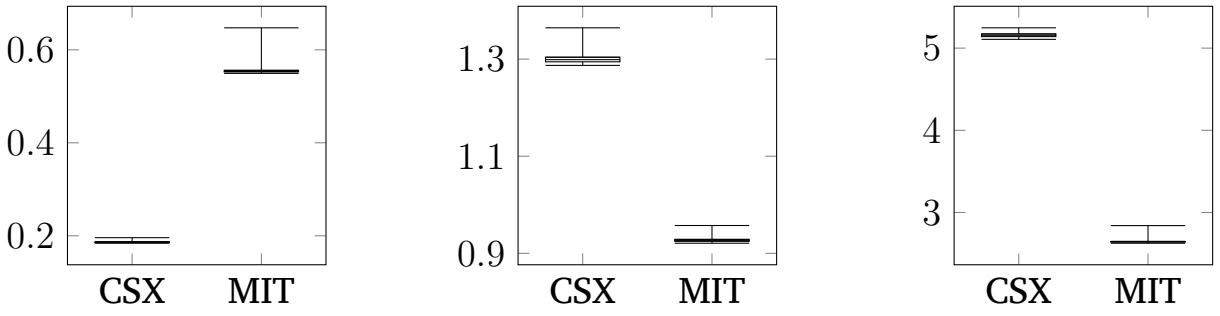
Оценим показатели эффективности разработанной системы, а именно скорость исполнения программ и потребление памяти в ходе их выполнения. Измерения будем приводить в сравнении с результатами исполнения того же кода на MIT Scheme — это потребует писать программы для оценки на подмножестве созданного диалекта Лиспа и диалекта, используемого в MIT Scheme.

Отметим, что данные для графиков потребления памяти получены посредством измерений одиночного исполнения программы, тогда как данные по скорости получены в результате запуска программы 1 000 раз для каждого исполнителя. Тестирование проводилось на компьютере с процессором Intel Pentium Silver N5000 и оперативной памятью объёмом в четыре гигабайта. В качестве операционной системы использовался Linux, компиляция производилась вызовом gcc с флагом `-O3`.

В качестве первого примера программы воспользуемся разработанным самоприменимым транслятором, написанным на требуемом подмножестве диалектов. В качестве тестовых данных на вход транслятора будем подавать его собственный исходный код. Выбор этой программы для примера обуславливается большей схожестью её кода с промышленным — как по размерам, так и по структуре — чем у более формальных примеров, таких как вычисление чисел Фибоначчи или факториала.

На рисунке 6.2 можно увидеть графики потребления транслятором оперативной памяти, в частности отчетливо видна работа сборщика мусора CSX. Рваный характер графика вызван принятым решением о вызове сборщика мусора, а именно что сборка мусора происходит только тогда, когда занимаемая объектами системы оперативная память по объёму превосходит объём оставленных после прошлой сборки мусора объектов более чем в два раза. Подчеркнём, что занимаемая системой CSX память при исполнении транслятора не превышает 205 килобайт, тогда как MIT Scheme требует более 277 мегабайт — то есть для запуска транслятора с помощью CSX требуется в тысячу раз меньше оперативной памяти, что может позволить использовать разработанную систему на большем числе устройств.

На рисунке 6.1а показаны диаграммы распределения времени исполнения, на которых изображены (сверху вниз): максимум, верхний квартиль,



(a) Транслятор (b) $Ack(3, 7)$ (c) $Ack(3, 8)$

Рис. 6.1: Распределения времени исполнения

медиана, нижний квартиль и минимум. Средним временем самоприменения транслятора для CSX является 186 миллисекунд, а для MIT Scheme — 554 миллисекунды.

В качестве второго примера воспользуемся программой вычисления функции Аккермана, также написанной на подмножестве избранных диалектов. Эта функция при наивной рекурсивной реализации известна быстрым ростом сложности, и в частности высокой глубиной рекурсии, в силу чего может служить удобным примером для проверки поведения системы при исполнении ресурсоёмких задач. Приведём определение функции Аккермана:

$$Ack(m,\ n) = \begin{cases} n + 1, & m = 0; \\ Ack(m - 1,\ 1), & m > 0,\ n = 0; \\ Ack(m - 1,\ Ack(m,\ n - 1)), & m > 0,\ n > 0. \end{cases}$$

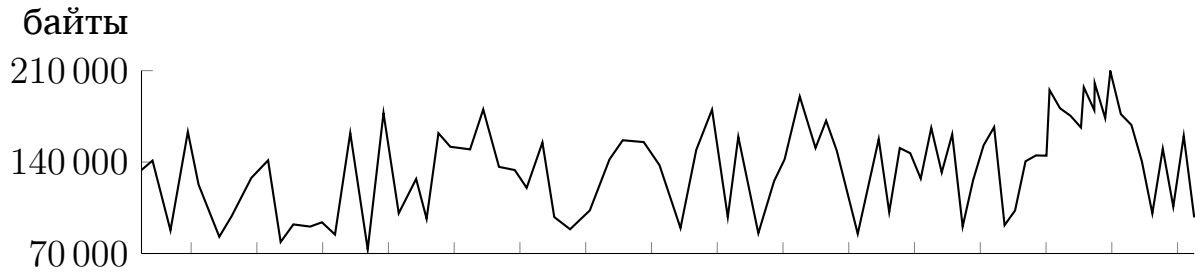
На рисунке 6.3 изображён график потребления памяти при вычислении $Ack(3, 7)$. Несмотря на повышение потребления по сравнению с примером исполнения транслятора, CSX всё ещё потребляет значительно меньше памяти, чем MIT Scheme.

На рисунке 6.1b показано распределение времени вычисления $Ack(3, 7)$; средним временем для CSX является 1300 миллисекунд, а для MIT Scheme — 927 миллисекунд. Видим, что при повышении алгоритмической сложности задачи преимущество CSX в скорости перед MIT Scheme пропало. Это наблюдение говорит о том, что CSX обладает крайне быстрым временем загрузки, но при этом не оптимизирует исполняемый код, в отличие от MIT Scheme. Однако даже при отсутствии оптимизаций, несмотря на достижение задачей времени исполнения, заметного пользователю, CSX остаётся не более чем в два раза медленней MIT Scheme.

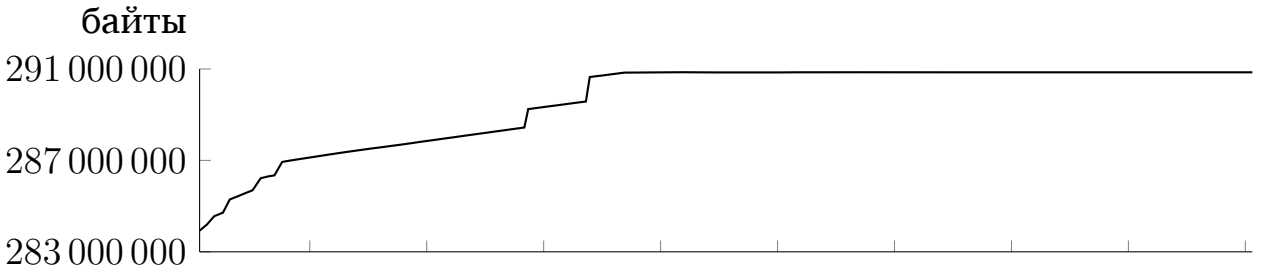
На рисунке 6.1c показаны аналогичные графики для $Ack(3, 8)$, среднее время исполнения системой CSX — 5157 миллисекунд, MIT Scheme справ-

ляется за 2644 миллисекунды. Время загрузки в рамках этого примера играет ещё меньшую роль, при этом соотношение скорости исполнения программ системами CSX и MIT Scheme слабо изменилось в сравнении с предыдущей задачей и остаётся ниже двойки.

Проведённые измерения свидетельствуют о приемлемых показателях потребления ресурсов компьютера разработанной системой CSX. Из полученных данных можно сделать вывод о перспективности построения оптимизирующего компилятора на основе предложенного подхода — при этом представляется возможным достичь скорости исполнения программ, продемонстрированной MIT Scheme.

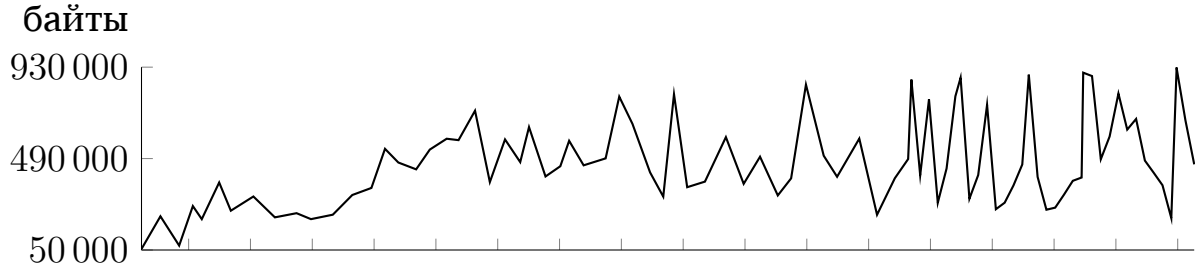


(a) CSX

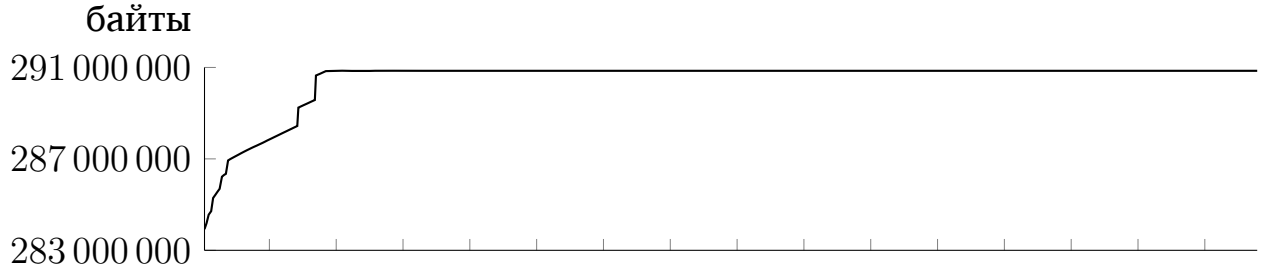


(b) MIT Scheme

Рис. 6.2: Потребление памяти в куче при трансляции



(a) CSX



(b) MIT Scheme

Рис. 6.3: Потребление памяти в куче при вычислении $Ack(3, 7)$

7 Заключение

7.1 Основные результаты

В рамках работы получены следующие основные результаты:

1. предложен подход к представлению S-выражений, допускающий их наглядную запись средствами языка Си в виде суперпозиции вызовов функций;
2. создана библиотека модулей Си, реализующая вычислительную модель Лиспа на основе S-выражений, представленных в соответствии с вышеупомянутым подходом;
3. создан интерактивный интерпретатор диалекта языка Лисп, а также самоприменимый транслятор, принимающий на вход текст на Лиспе в традиционном синтаксисе и создающий соответствующий модуль языка Си.

7.2 Перспективы

К перспективам дальнейшей работы можно отнести исследование возможности построения оптимизирующего транслятора, использующего предложенный метод. Желательно расширить возможности взаимодействия полученной системы с внешними средствами разработки и компонентами системы программирования, такими как отладчики и профилировщики. Актуальной выглядит задача поддержки исполнения параллельных программ в рамках предложенного подхода. Перспективно продолжить исследование расширением набора поддерживаемых системой языков, например такими языками, как Пролог, Рефал и Форт.

Литература

[1] И. Г. **Головин**, А. В. **Столяров**. *Объектно-ориентированный подход к мультипарадигмальному программированию.* Вестник МГУ, сер. 15 (ВМК), № 1, с. 46–50, 2002.

[2] А. В. **Столяров**. *Интеграция изобразительных средств альтернативных языков программирования в проекты на C++.* Деп. в ВИНИТИ РАН 06.11.2001 № 2319-В2001, Москва, 2001.

[3] А. В. **Столяров**. *Программирование: введение в профессию.* МАКС Пресс, Издание второе, том III: *Парадигмы*, с. 351–360, 2021.

[4] **ANSI X3.159-1989**. *Programming Language C*.

[5] John W. **Backus**. *The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference.* Proceedings of the International Conference on Information Processing, UNESCO, p. 125–132, 1959.

[6] David F. **Bacon**, V. T. **Rajan**. *Concurrent Cycle Collection in Reference Counted Systems.* Proceedings of the 15th European Conference on Object-Oriented Programming, p. 207–235, 2001.

[7] Tegawendé **Bissyandé**, Ferdian **Thung**, David **Lo**, Lingxiao **Jiang**, Laurent **Réveillère**. *Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects.* 2013 IEEE 37th Annual Computer Software and Applications Conference, p. 303–312, 2013.

[8] Walter **Bright**. *C's Biggest Mistake. //* Digital Mars, 2009. URL (date of access: 10.05.2022): https://www.digitalmars.com/articles/C-biggest-mistake.html

[9] Edsger W. **Dijkstra**, Leslie **Lamport**, A. J. **Martin**, C. S. **Scholten**, E. F. M. **Steffens**. *On-the-fly Garbage Collection: an Exercise in Cooperation.* Communication of the ACM, № 11, p. 966–975, 1978.

[10] Daniel **Holden**. *Cello library*. // Cello, 2015. URL (date of access: 10.05.2022):
https://libcello.org/

[11] R. **Kelsey**, W. **Clinger**, J. **Rees**. (eds.). *Revised^5 Report on the Algorithmic Language Scheme.* ACM SIGPLAN Notices, № 33(9), p. 26–76, 1998.

[12] Steve **Kollmansberger**, Martin **Erwig**. *Haskell Rules: Embedding Rule Systems in Haskell.* // Haskell Rules, 2006. URL (date of access: 10.05.2022):
https://web.engr.oregonstate.edu/~erwig/HaskellRules/

[13] John **McCarthy**. *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.* Communications of the ACM, № 3(4), p. 184–195, 1960.

[14] Brian **McNamara**, Yannis **Smaragdakis**. *Functional Programming in C++.* ICFP 2000. // FC++: The Functional C++ Library, 2012. URL (date of access: 10.05.2022):
https://people.cs.umass.edu/~yannis/fc++/

[15] Brian **McNamara**, Yannis **Smaragdakis**. *Logic Programming in C++ with the LC++ library.* // LC++: Logic Programming in C++, 2003. URL (date of access: 10.05.2022):
https://people.cs.umass.edu/~yannis/lc++/

[16] Flávio **Medeiros**, Márcio **Ribeiro**, Rohit **Gheyi**. *Investigating Preprocessor-Based Syntax Errors.* ACM SIGPLAN Notices, № 49(3), p. 75–84, 2014.

[17] Roshan **Naik**. *Blending the Logic Paradigm into C++.* // Castor: Logic paradigm for C++, 2008. URL (date of access: 10.05.2022):
http://www.mpprogramming.com/resources/CastorDesign.pdf

[18] Martin **Odersky**, Philip **Wadler**. *Pizza into Java: Translating theory into practice.* Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '97), p. 146–159, 1997.

[19] Chris **Okasaki**. *Techniques for embedding postfix languages in Haskell.* Proceedings of the 2002 ACM SIGPLAN Haskell Workshop, p. 105-113, 2002.

[20] Harel **Paz**, David F. **Bacon**, Elliot K. **Kolodner**, Erez **Petrank**, V. T. **Rajan**. *An efficient on-the-fly cycle collection.* ACM Transactions on Programming Languages and Systems, № 29(4), p. 20–es, 2007.

[21] Ken **Shirriff**. *Arc Programming Language Reference.* // Arc Programming Language, 2008. URL (date of access: 10.05.2022): https://arclanguage.github.io/ref/

[22] Michael **Spivey**, Silvija **Seres**. *Embedding PROLOG in HASKELL.* Haskell Workshop, p. 25–38, 1999.

[23] **Stack Overflow Developer Survey 2021**. // Stack Overflow, 2022. URL (date of access: 10.05.2022): https://insights.stackoverflow.com/survey/2021

[24] G. L. **Steele**. *Common Lisp the Language.* Digital Press, 2nd edition, 1990.

[25] **TIOBE Index for May 2022**. // TIOBE, 2022. URL (date of access: 10.05.2022): https://www.tiobe.com/tiobe-index/

[26] Steve **Ward**, Mat **Hostetter**. *Curl: A language for web content.* International Journal of Web Engineering and Technology, No 1(1), p. 41–62, 2003.