Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего профессионального образования

«Уральский федеральный университет
имени первого Президента России Б.Н. Ельцина»

Институт естественных наук и математики
Кафедра высокопроизводительных компьютерных технологий

Развитие конструктора компиляторов LiME

Допустить к защите:
Зав. кафедрой,
кандидат технических наук
Созыкин Андрей Владимирович

Магистерская диссертация
студента 2 курса
Куклина Ильи Юрьевича

Научный руководитель:
Бахтерев Михаил Олегович

Екатеринбург

2017 г

ОГЛАВЛЕНИЕ

Введение

Еще в конце 80-х годов XX века стало понятно, что для того, чтобы удовлетворять текущие потребности в вычислительной мощности, недостаточно просто поднимать тактовую частоту процессоров, опираясь на незыблемую фоннеймановскую архитектуру: с каждым разом шаг будет уменьшаться и, рано или поздно, тактовая частота достигнет своего предела. Возникла настоятельная потребность дополнить и расширить фоннеймановские принципы, научиться строить иные системы, позволяющие продолжать увеличивать производительность без увеличения тактовой частоты. Так появились параллельные вычислительные системы, а способы их построения и использования стали называться технологиями параллельной обработки данных.

Но и эта технология стала приближаться к своему пределу. Решающую роль в производительности стала играть не эффективность и количество функциональных устройств процессора, а сложность и эффективность схем передачи данных между ними, то есть то, что называется архитектурой процессора. Одним из направлений поиска методов увеличения эффективности этих схем передачи данных являются поиски новых, неклассических архитектур, с нестандартной организацией интерфейса между вычислительными устройствами и программистом (то есть, нестандартными ISA). Например, архитектуры EPIC и EDGE. Одной из таких попыток является архитектура процессора российской разработки, MultiClet, и ее последняя реализация, MultiClet R1, на которой хотелось бы остановиться поподробнее.

MultiClet

Мультиклеточное ядро – это группа идентичных процессорных блоков (2 и более), объединенных полносвязной однонаправленной коммутационной средой. Процессорный блок в мультиклеточной архитектуре называется клеткой. Набор команд, который она может выполнять определяется конкретной реализацией и не зависит от архитектуры. Особенности взаимодействия процессорных блоков между собой вытекают из представления алгоритма.

Любую формулу можно представить в виде ярусно-параллельной формы (ЯПФ). Рассмотрим простой пример: $g = e * (a + b) + (a - c) * f$, в ЯПФ это может выглядеть так:

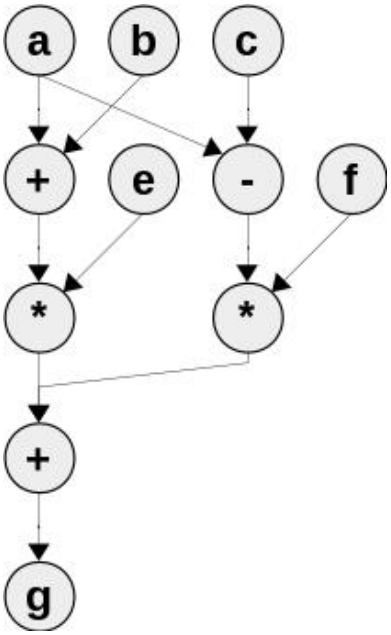

Рис 1. Пример ЯПФ

В общем случае, для выполнения операций узлы, расположенные на i-ом ярусе, могут использовать результаты, полученные на ярусах от 1 до (i–1)-го. Из этого следует, что команды, находящиеся на одном ярусе, независимы. Любой алгоритм – это множество формул, которое разделено на подмножества – линейные участки, связанные между собой операторами передачи управления. Под линейным участком (ЛУ) понимается такое подмножество формул, которое вычисляется тогда и только тогда, когда управление передано на данный

линейный участок. Внутри линейного участка, информационно несвязанные формулы могут выполняться в любом порядке.

Мультиклеточный процессор оперирует структурами, которые называются параграфами. Параграф – это информационно замкнутая последовательность команд. Параграф является аналогом команды, после выполнения которой, изменяется состояние процессора и/или систем в его составе (регистров, шин, ячеек памяти, каналов ввода/вывода и т.д.). Т.е. для мультиклеточного ядра параграф – это команда. Основная особенность мультиклеточной архитектуры в том, что она непосредственно реализует ЯПФ представления алгоритма.

Команды именуются, и при этом имена должны идентифицировать собственно команду, а не ее местоположение или другие особенности реализации. Доступ к результатам может осуществляться по именам как команд-источников, так команд-приемников. В первом случае используется широковещательная рассылка с последующим поименным отбором, при которой в поле операнда команды-приемника задают имя команды-источника. Во втором – выполняется поименная рассылка, при которой в команде-источнике задают имя команды-приемника результата. Теоретически узлы ЯПФ можно поименовать любым образом. Единственное требование – однозначность. Для этого в мультиклеточном процессоре команды, при выборке из памяти, получают тег (метку) – таким образом им и их результатам дается локальное имя – и далее взаимодействие между командами организовывается через теги. Данные можно получать из любой команды с тегом меньше своего. Номер тега требуемого результата задается разницей значений тега команды и тега требуемого результата. Например, если в операнде указано `@5`, а тег команды равен `7`, то в качестве операнда используется результат команды с тегом `2`. Все результаты выполнения команд поступают в коммутационную среду, из которой по тегу отбираются требуемые. Таким образом в мультиклеточном процессоре

используется широковещательная рассылка результатов с последующим их поименным отбором.

Из такого понимания алгоритма вытекают многие другие свойства мультиклеточной архитектуры:

1. Независимость от количества клеток.

Алгоритм не зависит от количества клеток. Информационные связи между командами указаны явно и нет необходимости знать о количестве процессорных блоков, когда пишется программа. Команды просто ждут готовности своих операндов и после этого уходят на исполнение и им не важно кто и когда им подготовит операнды.

2. Динамическое распределение вычислительных ресурсов.

Т.к. код не зависит от количества клеток, которым он будет исполняться, появляется способность мультиклеточного ядра к распределению своих вычислительных ресурсов во время работы, при этом управление происходит программно. Ядро может быть разбито на группы из 1 или более клеток: каждая группа может исполнять разный код.

3. Все команды, готовые к выполнению, выполняются одновременно.

Все, что в данный момент может быть выполнено – будет выполнено без каких-либо специальных указаний со стороны программиста, т.е. распараллеливание происходит аппаратно, основное условие – команда должна получить все данные для исполнения.

4. Масштабируемость, нет ограничений на количество клеток.

Архитектура не ограничивает количество клеток. Дело стоит за технологиями. Последняя реализация, MultiClet R1, имеет в ядре 4 клетки.

Зачем нужен новый компилятор

На данный момент существует уже множество компиляторов под всевозможные системы. Некоторые из них, например LLVM, изначально созданы модульными, позволяя вносить изменения только одной часть компилятора, не вдаваясь в подробности остальных. Казалось бы: нужно всего лишь поменять backend, т.е. сделать кодогенератор для нового процессора, и все готово. Но, к сожалению, вся проблема кроется именно в инновационности таких архитектур, как MultiClet или TrueNorth от IBM: они неклассические, а значит компиляторы, изначально предусмотренные для классических процессоров будут слабо справляться с поставленной задачей. Однако в компании «Мультиклет» все же сделали попытку адаптировать LLVM для своей архитектуры, столкнувшись со следующими проблемами:

1. LLVM ориентирован на генерацию кода для регистровых машин, коей MultiClet не является. Роль сверхбыстрой памяти в мультиклеточном процессоре выполняет коммутатор, который обладает несколько другими свойствами, нежели набор регистров:
    a. Результат каждой команды автоматически кладется в коммутатор; другие команды могут ссылаться на результат предыдущих команд. Ввиду того, что коммутатор конечный (в последней реализации, может содержать не более 63х последних результатов), результаты команд не могут храниться продолжительное время, в отличие от значения регистра, которое будет в храниться, пока не будет перезаписано.
    b. Ячейки коммутатора именуются не статично, а относительно текущей команды: если какая-либо команда обращалась к результату одной из предыдущих команд через `@2`, то следующая команда по той же метке `@2` будет обращаться к результату уже другой команды.

Было принято решение считать ячейки коммутатора регистрами с ограниченным временем жизни, т. е. значения таких регистров актуальны

только в пределах одного базового блока, который в свою очередь является параграфом, и, более того, их актуальность в пределах одного базового блока зависит от области видимости результатов, предоставляемой коммутатором. Фреймворк LLVM предоставляет следующий набор аллокаторов регистров: *fast*, *basic*, *greedy*, *pbqp*. В виду особенностей (наличие коммутатора) мультиклеточной архитектуры из представленных выше аллокаторов правильный ассемблерный код может быть сгенерирован только при использовании *fast* аллокатора, который распределяет регистры на уровне базовых блоков, что в рамках принятой концепции "базовый блок есть параграф" и необходимо. Так как *fast* аллокатор является аллокатором по умолчанию для сборок с поддержкой возможности отладки, он не осуществляет никаких оптимизаций. Для устранения данного факта был разработан свой собственный аллокатор *multiclet*, который также содержит некоторые дополнительные архитектурно-зависимые оптимизации.

2. Необходимо было описать процесс выборки и преобразования инструкций промежуточного представления LLVM IR, представленного в виде направленного ациклического графа (DAG), в соответствующие инструкции, явно поддерживаемые целевой машиной. При реализации данного этапа выявилась необходимость специальной обработки инструкций, которые отображаются в инструкции *setXX* целевой машины. Дело в том, что инструкции *setXX* установки значения регистра (**не** ячейки коммутатора; регистры в мультиклеточном процессоре являются, фактически, особыми ячейками памяти, используемыми для управления самим процессом и различной периферией) выполняются по завершению параграфа, т. е., в предположении, что параграф есть базовый блок - по завершению базового блока, поэтому данные инструкции должны разбивать этот блок, что замедляет скорость работы кода.

3. В мультиклеточной архитектуре имеются лишь инструкции установки адреса следующего выполняемого параграфа, а сама передача управления осуществляется по окончанию текущего параграфа. Поэтому на данном проходе, во-первых, в базовый блок, в котором отсутствует инструкция передачи управления (в этом случае предполагается, что выполняется следующий расположенный в памяти базовый блок), добавляется инструкция безусловной передачи управления на следующий базовый блок, а во-вторых, в базовом блоке, в котором имеется одна условная инструкция передачи управления, непосредственно за которой следует одна безусловная инструкция передачи управления, безусловная инструкция передачи управления заменяется на условную инструкцию передачи управления с противоположным условием.

Несмотря на все это, удалось сделать рабочую версию компилятора. Однако, из-за множества попыток реализовать особенности неклассической архитектуры, сильно пострадали возможности компилятора оптимизировать код.

Например, на некотором этапе была проведена проверка производительности процессора MultiClet R1 при вычислении криптографических функций. Самым большим замедляющим фактором оказалась не архитектура и не частота процессора, а низкая оптимальность запускаемого кода. Одним из первых протестированных алгоритмов был SHA-256. Реализация основной вычислительной функции `sha256_transform` показана в приложении А. Как можно заметить, код содержит как циклы, так и просто последовательные операции.

Здесь необходимо напомнить, что наиболее быстро мультиклеточный процессор выполняет код внутри линейного участка, используя сверхбыструю память. То есть вычисления внутри параграфа, обменивающиеся данными через коммутатор, выполняется быстро; все остальное: переходы между параграфами,

запись во внутренюю память процессора и в регистры выполняется несравнимо медленнее.

В приложении Б показана начальная часть скомпилированного с максимальным уровнем оптимизации (-O3) кода функции `sha256_transform`. Первым делом бросаются в глаза недочеты, незначительно замедляющих код, например как почти пустой первый параграф (в исходном коде параграф начинается с конструкции «`метка:`» и заканчивается ключевым словом «`complete`»), который можно легко объединить со вторым параграфом, убирая лишний медленный переход между ними. Самая большая проблема в данном примере находится в параграфе `LBB0_5`: в нем находится всего 8 быстрых команд вычислений и работы с коммутатором и 33 команды чтения и записи во внутреннюю память процессора. Это является огромнейшей проблемой: 8 команд работы с коммутатором выполнятся быстрее чем даже **одна** операция записи во внутреннюю память. И это не является просто неудачным исходным кодом на Си: имея некоторый опыт написания программ прямо на ассемблере Мультиклета, можно написать тот же самый код с переходом к следующим вычислениям в том же параграфе, а не в следующем, и с передачей огромной кучи данных через коммутатор, а не внутреннюю память.

Иными словами, проблема является следующей: текущий компилятор не способен выстраивать достаточно длинные линейные участки кода, являющиеся основным преимуществом мультиклеточных процессоров.

Были также испытаны компиляторы GCC, TCC и LCC (наиболее активно использовался для предыдущего процессора, MultiClet P1), но LLVM, при всех недостатках, в итоге, показал наилучшие результаты.

LiME

Компиляторы – очень сложные программы, и, как выяснилось, вносить в них дополнительный уровень сложности, связанный с описанием параллельных процессов традиционными методами, нерационально. Отсюда и возникла необходимость создания нового компилятора, позволяющего:

- достичь такого представления программы, которое было бы удобно для последующей генерации кода процессоров с нетрадиционной архитектурой;
- применять необходимые алгоритмы оптимизации;
- расширять исходные языки конструкциями, позволяющими использовать особенности разрабатываемой архитектуры;
- упростить разработку за счет следующего:
    - Повторного использования реализаций языковых конструкций (напимер, код «*x* + *y*» во многих языках означает одно и то же).
    - Использования событийной модели трансляции кода. Можно ожидать некоторого упрощения работы за счёт этого, потому что событийные модели параллельных процессов (например, CSP) похожи на модели рекурсивных типов из теории языков программирования (μ-типы). Сами же событийные модели часто упрощают разработку сложных систем (например, каналы UNIX).
    - Поддержки процессоров с различными архитектурами для привлечения к работам сторонних разработчиков.
    - Освобождения разработчиков от необходимости заниматься управлением ресурсами на низком уровне за счёт предоставления им специального языка для описания семантики языковых конструкций.

С учетом вышеупомянутого был создан конструктор компиляторов LiME. Конструктор разбит на 3 независимых процесса:

1. `frontend` – группа программ, которая должна превратить текст на исходном языке в определённое описание синтаксического дерева программы.
2. `lime-kernel` – программа, которая по набору форм, описывающих правила вывода графа программы, и по синтаксическому дереву программы, выдаёт граф программы.
3. `codegen` – кодогенератор для соответствующей архитектуры.

Эти процессы специально сделаны независимыми, чтобы:

- Изолировать домены ошибок (failure domains) и утечек памяти
- Разделить работу через определение форматов входных данных
- Открыть разработчикам возможность описывать компоненты на наиболее удобных языках программирования

Frontend

Основная задача данной части – преобразовать исходный текст на некотором языке программирования в описание синтаксического дерева в некотором формате. Например, для уже существующего компилятора Си, в системе UNIX frontend запускается так:

```
mcpp < source-code.c | mc-cfe
```

В этой структуре **mcpp** – стандартный препроцессор для обработки директив препроцессора Си, таких как `#include` для подстановки в текущий исходный файл других файлов или `#define` для определения макросов. Кроме этого, задача препроцессора – вычищать комментарии, обрабатывать три-графы, сращивать строки.

mc-cfe – специальный транслятор с «чистого» Си в язык синтаксических деревьев, состоящий из набора конструкций, описывающих атомы:

```
TYPE hint.size."bytes"
```

где:

- **TYPE** – тип атома, который может быть:
  - A – лист дерева. Например, `A 02.1."a"` – идентификатор Си.
  - L – начало «бинарного» узла в дереве, левым поддеревом которого является предыдущее выражение.

    Для Си, например, `A 01.1."a"; L 14.1."+"` – описывает «частичное» выражение **`a +`**.
  - U – начало «унарного» узла в дереве.

    Например, `U 0c.3."p++"` – оператор пре-инкремента в Си.
  - E – конец описания «унарного» или «бинарного» узла в дереве. Описывает конец поддерева (правого, если узел «бинарный»). Например,

    ```
    A 02.1."a"; L 14.1."+"; A.02.1."b"; E 14.1."+"
    ```

    описывает выражение `a + b`.
- **hint** – определенная подсказка, которая является начальной типизацией, при помощи которой мы строим более сложные типы и отличаем строку "3" от числа 3. Атомами в системе описываются все токены исходной программы.
- **size** – длина строки, представляющей токен
- **bytes** – сама строка

Так, например, программу

```
fn () { return a + b; }
```

frontend превращает в дерево:

```
A 0.0 01.3."int"
L 0.4 06.11."@-func-decl"
A 0.0 02.2."fn"
L 0.3 06.6."@-argv"
A 0.0 ff.7."NOTHING"
E 0.0 06.6."@-argv" // 2
L 0.3 06.11."@-func-body"
U 0.0 0a.1."{"
A 0.2 01.6."return"
L 0.7 06.9."@retvalue"
A 0.0 02.1."a"
L 0.2 14.1."+"
A 0.2 02.1."b"
E 0.0 14.1."+" // 2
E 0.0 06.9."@retvalue" // 5
E 0.0 0a.1."{" // 8
E 0.0 06.11."@-func-body" // 10
E 0.0 06.11."@-func-decl" // 16
```

Вторая колонка здесь – это позиции токенов в исходном тексте программы, заданные своими смещениями относительно предыдущих в формате `line.offset`.

Codegen

Генератор кода получает на вход выданный ядром граф программы. Он представлен в текстовом виде и синтаксис описания графа изначально описывался следующей грамматикой:

$G \rightarrow$ *"( )"* | *"("* A *")"*	Граф
$A \rightarrow a$ | a *";"* A	Список атрибутов
$a \rightarrow G$ | *n* | *num* | *atom* | *id*	Атрибут
n $\rightarrow$ *"."* *id* (G | *id* G)	Узел, напр.: `.S n5 ('02.1."b"; n1)`
num	Натуральное число в десятичной записи
atom	Атом
id	Идентификатор

Пример части графа, взятый из реально использующегося исходного кода, показан на листинге 1.

```
(
      .FEnv (("L"; '44.1."="); .F
      (
            .Nth op (.FIn (); (1));
            .FPut (0;
                  (("lvalue"; "left"); ("rvalue"; "right")); .F
                  (
                        .Nth l (.FIn (); (1; 0));
                        .Nth r (.FIn (); (1; 1));
                        .wr (.T ("I"; 4); l; r);
                        .FOut (0; (("address"; l)))
                  ));
                  ...
      )
)
```

Листинг 1

Синтаксис этого представления является предметом данной работы и будет более подробно рассмотрен позже.

Ядро

Задача ядра – по синтаксическому дереву построить граф программы. Этот граф выводится в соответствии с математической моделью RiDE. Причины такого решения:

- RiDE – это система динамического построения графа вычисления, а графы – везде графы;
- Эксперименты с RiDE показывают, что объём кода, описывающий продолжения вычислений, можно существенно сократить (прямая аналогия: продолжение распределённого вычисления – продолжение графа программы, в обоих случаях неопределённость по необходимым для этого данным);
- Философские рассуждения позволяют говорить: активация продолжения графа вычисления соответствуют динамическому построению сопоставлений по шаблону – основы компиляторов, и мы знаем, что динамические графы удобнее статических;
- Системы управляемые событиями логически проще, чем функциональные и императивные системы (например, конвейер процессов $a \mid b \mid c$ проще тройного вложенного цикла);
- Модель событийной системы CSP (communicating sequential processes) совпадает с моделью рекурсивных μ-типов – одного из оснований теории языков программирования.

Модель RiDE разрабатывалась практически одновременно с самим компилятором LiME. Можно сказать, что компилятор был тестовой площадкой для теоретических идей. Иными словами, LiME является прямой реализацией этой модели и, соответственно, использует все её преимущества.

Основной цикл

Ядро считывает синтаксические команды и на их основе перестраивает стек областей вывода. Области вывода описывают текущие графы программ для тех или иных подвыражений в исходном коде. В каждой области есть:

1. Два «реактора» (или, проще, дочерние области вывода)
2. Одна таблица связности с другими областями
3. Накопленный для этой области вывода граф программы

Каждый реактор представляет собой структуру:

- Таблица из пар *имя:значение*, в которой накапливается информация об опубликованных под определёнными именами значениях: узлах графа, типах, символах и выражениях из атомов, номеров, узлов, типов, символов;
- Список ожидающих своей активации форм;

Таблица связности между областями вывода – это просто таблица из пар вида *имя:«ссылка на область»*. Граф программы – это граф программы. Чисто технически он приписывается к 1-му реактору области вывода. Именами в этих разных таблицах служат выражения из типов, атомов и чисел. Выражения структурно являются подмножеством выражений для описания атрибутов узлов. Например:

```
(0; 1; ('0.1."a"; .T (34)); .T (.T (34); 76; '3.4."5678"))
```

Теперь можно описать основной цикл вывода:

- Встречая

  ```
  s = {A|U} hint.size."bytes"
  ```

  ядро размещает на стеке новый контекст, «засевая» его 0-й реактор парой *s:s* и формой, найденной по одному из имён (порядок поиска имеет значение):
  - `({A|U}; hint.size."bytes")`
  - `({A|U}; .T (hint.0.""))`

- Встречая

  ```
  L hint.size."bytes"
  ```

  ядро выталкивает из стека контекст c_L, размещает на вершине новый контекст c_t, «засевает» его, и заносит в таблицу связанных с c_t окружений окружение c_l под именем `0.4."LEFT"`

- Встречая

  ```
  E hint.size."bytes"
  ```

  ядро выталкивает из стека контекст c_R, проверяет, что это {U|L}-контекст с подходящим атомом, «засевает» контекст на вершине стека c_t, и заносит в таблицу известных в c_t окружение c_R под именем `0.5."RIGHT"`.

Обработав очередной синтаксический элемент ядро переходит к циклу вывода в различных областях. Модель вычислений RiDE позволяет этот вывод осуществлять в произвольном порядке, даже сохраняя контексты во внешнюю память. Вывод продолжается, пока ядро не встретит в одной из активированных на вершине стека узел `.Go`. Таким образом ядро создаёт дерево взаимодействующих процессов вывода, отражающее структуру синтаксического дерева.

Цикл вывода и формы

После манипуляции областями на стеке и «засева» их данными ядро переходит к циклу вывода графов программы, соответствующих различным узлам деревьев. Основной сущностью в процессе вывода графов является форма. Формы – это параметризованный кусочек графа. Параметризован этот кусочек с «двух сторон». С одной стороны, со стороны входов, к некоторым узлам, определённым внутри этого кусочка можно протянуть дуги от некоторых уже существующих в графе программы узлов и значений отмеченных в области вывода некоторыми именами. С другой стороны, стороны выходов, этот кусочек

после интерпретации ядром сам может стать источником именованных узлов и значений в области вывода. Кроме того, интерпретация формы может привести к публикации в некоторых областях вывода новых форм, ожидающей своей активации.

- Форма может быть *опубликована* в контексте вывода. При публикации формы задаются имена, появление которых в соответствующей области вывода к активации формы и к «протягиванию» в её граф дуг от этих узлов.
- Форма может быть *активирована*. В этом процессе ядро на основании анализа тела формы:
    - достраивает граф программы, связанный с текущим контекстом вывода;
    - публикует формы, по описаниям в теле активированной формы;
    - именует некоторые узлы и значения (не обязательно новые) по описаниям в теле активированной формы.

  С появлением новых имён у прежних или новых узлов графа и новых опубликованных форм процесс вывода продолжается.

  Простой пример:

```
.FEnv (("binop"; '14.1."+"); .F
(
     .Nth l (.FIn (); (1; 0));
     .Nth r (.FIn (); (1; 1));
     .add result (.T ("I"; 4); l; r);
     .FOut (0; (("result"; result)))
));
```

Эта простая форма описывает кусочек графа с единственным узлом `.add`, задающим операцию целочисленного (в терминах Си-99, `int`) сложения. То, что именно такой узел описывает именно такое сложение «знает» только генератор кода.

Телом формы служит граф, описанный в списке атрибутов узла `.F`. Форма описывает наращивание графа узлом `.add`, соответствующим машинной операции, этот узел «привязывается» к двум другим узлам, которые задаются

узлами `.Nth`. Несмотря на не совсем очевидный синтаксис конструкции относительно простые:

- `FIn` – ссылка на входы формы;
- `Nth` – позволяет назвать необходимые узлы из входов формы;
- `FOut` – опубликовывает определённое имя для определённого узла;
- `FEnv` – регистрация формы с определённой сигнатурой в окружении (в частности, ядро ищет по этим сигнатурам формы для «посева»).

Функциональность ядра реализована программой `mc-knl`, которая на стандартный вход принимает исходный файл, описывающий синтаксическое дерево программы, и считывает поэлементно входные A, L, U, E инструкции. По этим инструкциям ядро создаёт дерево контекстов вывода по описанному выше алгоритму.

Контекст R, который сначала размещается на стеке, а потом вписывается в дерево, является специальной структурой, в которой происходит вывод кусочка графа выражения, синтаксическому корню которого соответствует R. Технически контексты реализованы при помощи структуры данных `Array`, которая воплощает ассоциативные массивы, и некоторого множества функций, имена которых начинаются со слова `area`. Удобнее называть контексты *областями вывода*. Чтобы в некоторой области начался вывод, туда нужно записать определённую информацию. Эта информация размещается там в виде форм.

Функционал реализован с помощью следующих конструкций:

- *Области вывода*. Область вывода – это некая структура из ассоциативных массивов. В области вывода есть таблица именованных «выходов», то есть некоторых именованных ключами значений.
- *Ключи*. Ключами в этой таблице могут служить выражения, которые состоят из типов, атомов и разнообразных списков из этих объектов. Ключи позволяют отыскать значения в этой таблице. Значениями могут

быть различные значения, в частности, те же самые типы, числа, атомы, ссылки на узлы уже построенных участков графа, символы, различные списки из них. Выходами они называются, потому что, в основном, это выходы из графа, то есть, когда в этой таблице есть ссылка на узел графа, то из этого узла можно провести к другому узлу из дописываемый в граф формы дугу, которая ведёт от выхода ко входу, если говорить в традиционных терминах графов. Например, в этой таблице выходов может быть такая пара ключ → значение:

```
('0.1."A"; 10; (.T ('0.1."I"; 4); '0.2."xy")) → ((10;
                                    '0.1."B"); N)
```

- *Список форм*. Ещё одним важным компонентом области вывода является список форм. Каждая форма попадает в этот список вместе со списком имён «входов». Входы – это аргументы для формы, которые могут использоваться в её теле для настройки кусочка графа, который связан с этой формой. Форма находится в этом списке ожидающих форм, пока в таблице выходов не появятся все выходы, имена которых совпадают с входами для формы. Если необходимые выходы были образованы, то форма «подключается» своими входами к этим выходам и обрабатывается.
- *Окружения*. Окружения сконструированы как деревья из ассоциативных массивов (всё те же `Array`). Зная ссылку на какое-нибудь окружение и ключ можно отыскать какое-нибудь значение, находящееся по дереву выше (корень у дерева вверху). Такие окружения хранят различные именованные объекты, необходимые для вывода графов программы, среди которых следующие (в каждом пункте сначала пример):
  - `'02.1."c" → S:4` – имена символов со ссылками в таблицу символов, в которой все символы пронумерованы последовательно;
  - `('0.6."struct"; '02.1."X") → T:5` – имена типов со ссылками в таблицу типов, в которой все типы пронумерованы последовательно; в Си определения вида `struct X` привязаны к областям видимости (`scope`), и здесь нужен механизм для

отслеживания разных структур (то есть, разных типов) с одинаковым именем `struct X`;

- `('0.1."A"; .T ('02.0.""))` → `(some form body)` – имена тел различных форм; в окружении хранят только тела форм без указания входов, то есть, без указания именованных объектов, которые должны возникнуть в области вывода, чтобы форма активировалась; то есть, хранится только параметризуемый граф в виде списка узлов.

Объекты разного вида (типы, символы, формы и прочие) могут иметь одинаковые имена (ключи).

- *Засеивание (инициализация) области вывода*. Когда ядро загружает очередную синтаксическую инструкцию из описания дерева и создаёт новую пустую область вывода для неё на стеке или трансформирует уже существующую (это в случае инструкций вида E), то оно размещает в этой области новый выход и новую форму, тело которой разыскивается в текущем окружении по определённому ключу.

При обработке форм ядро умеет выполнять некоторые системные конструкции, которые представлены некоторыми узлами в графе, который описан в теле формы. Среди этих инструкций следующие:

- `FOut` – размещение определённого вывода (выхода) в указанной области. Например,

```
.FOut(0, (
     (('0.1."X"; 1; '0.3."key"); v1);
     (('0.1."Y"; 2; ('0.3."key"; '0.4."plus")); v2))
```

разместит в текущей рабочей области (что описывает 0) два выхода. Здесь `v1`, `v2` – ссылки на какие-то другие узлы графа.

- `FPut` – размещение формы с некоторым телом и определённой сигнатурой (набором входных имён, ожидаемых формой). Например,

```
.FEnv (0; (('0.3."key"; 1); '0.4."key2"); .F
(
     .Nth l (.FIn (); (1; 0));
     .Nth r (.FIn (); (1; 1));
     .add result (.T ("I"; 4); l; r);
     .FOut (0; (("result"; result)))
));
```

разместит в текущей области вывода (0 это задаёт) форму с описанным в подграфе F телом, ожидающую двух входов с указанными именами.

- `FEnv` – размещение или поиск тел форм в окружениях. Например:

```
.FEnv (("some"; "new"; "name"); .FEnv(("binop"; '14.1."+")))
```

В этой конструкции вложенный FEnv заставит ядро отыскать тело формы с соответствующим именем-ключом в иерархии текущего окружения, а внешний (отличается от внутреннего наличием двух параметров) разместит это тело в текущем окружении под новым именем. Здесь конструкции вида `"bytes"` – это атомы с `hint = 0` и соответствующей длиной, вычисляемой автоматически.

- `Ex` – проверяет наличие ключа для объекта заданного вида в родительских окружениях текущего окружения (от слова exists). Например:

```
.Ex("FEnv"; ("binop", '14.1."+"))
```

Ядро в процессе обработки форм оценивает эту конструкцию в один из атомов:

  - "FREE" – значения с соответствующим именем (ключом) отсутствует;
  - "BOUND" – значение с соответствующим именем существует в текущем окружении;
  - "SHADOW" – значение с соответствующим именем существует где-то в одном из родительских для текущего окружений. Нам нужно различать ситуации BOUND и SHADOW, так как различные имена в Си могут вести себя по-разному. Кроме того, нужно помнить о универсальности системы.

- `Nth` – конструкция позволяет разобрать дерево выражения. Например, так можно разбирать входы, которые получает форма. S-выражения представляют собой деревья с различными степенями узлов, поэтому, чтобы указать какое-то поддерево, мы используем список номеров, задающих путь по узлам таком дереве. Например:

```
.Nth(.FIn(); (1; 2; 1; 3; 4))
```

- `FIn` оценивается в дерево, представляющее входы для формы. `Nth` в примере будет оценено в 4 поддерево, 3-го поддерева, 1-гоподдерева, 2-го поддерева, 1-го поддерева входного S-выражения. 1-ое поддерево этого выражения описывает выражения, которые соответствуют ключам, записанным в 0-ое поддерево. Если форма ожидала на входе, допустим, ключи `("key"; 1)` и `("key"; 2)`, а во множестве выводов были пары

```
("key"; 1) → ("value"; 1)
("key"; 1) → ("value"; 2)
```

  то дерево, к которому можно обратиться через конструкцию `FIn` будет выглядеть так:

```
(
     (("key"; 1); ("key"; 2));
     (("value"; 1); ("value"; 2))
)
```

  и в этом случае

```
.Nth(.FIn(); (1; 0; 1))
```

  будет вычислено в значение 1.

Механизм посева реализован прямо в ядре. Алгоритм можно описать следующим образом:

1. Ядро считывает очередную синтаксическую команду

```
X l.c hint.size."bytes"
```

   Здесь X – одна из строк A, L, U, E.

2. Ядро соответствующим образом модифицирует стек областей вывода (как уже было описано ранее).
3. Ядро размещает в области вывода на вершине стека (текущей области вывода) вывод:

```
.FOut (0; (
     ("lexem"; (
          ("X"; 'hint.size."bytes");
          ("X"; .T ('hint.0.""))))
))
```

Смысл конструкции следующий: нам необходимо отыскать в окружении форму, которая реализует начало вывода графа для соответствующего синтаксического элемента. Элементы могут быть как конкретные (типа ключевого слова `extern`), так и принадлежать определённому виду (типа идентификаторов). Должно быть два варианта представления этих элементов.

- Конкретный: с полной точной спецификацией для атома
- Обобщённый: с указанием только типа атома

Средствами ядра тип атома вычислить пока нельзя, поэтому ядро делает это явно.

4. Ядро размещает в текущей области выводы и форму, соответствующую считанной синтаксической инструкции.

Промежуточное представление LiME

Это представление, описанное на страницах 14-15, имеет ряд проблем, которые мне и предстояло исправить:

1. Семантика этих конструкций не особо выдающаяся и очевидная. Так, выражение

   ```
   .add n10 (n1; n7; n9)
   ```

   описывает узел графа, в который ведут дуги из узлов `n1`, `n7`, `n9` (порядок этих узлов важен только для генератора кода, ядро системы оперирует всеми подобными узлами общим способом).

   Имя узла, **если присутствует**, указывается идентификатором после его типа (выражения вида «.id», в данном случае – `n10`), откуда возникает проблема неясности: узел `.add` имеет имя `n10` и 3 атрибута, или он имеет 2 атрибута: `n10` и список из 3-х элементов, как единый второй атрибут?

   Человек, которому привычен синтаксис S-выражений может увидеть второй вариант, хотя он неправильный.
2. Излишнее количество символов «;», никак не влияющих на семантику языка и лишь отвлекающих внимание.
3. Так как синтаксис получился уникальным, нельзя воспользоваться текстовыми редакторами с подсветкой синтаксиса для облегчения работы с кодом без создания дополнительного плагина для этого конкретного языка. У каждого человека свои предпочтения, в том числе в текстовых редакторах, и создавать для каждого из них плагин – нерационально.

   Например, в листинге 2 включена стандартная подсветка для языка Lisp (S-выражений), и символы «;» здесь воспринимаются как начало комментариев.

```
(
      .FEnv (("L"; '44.1."="); .F
      (
            .Nth op (.FIn (); (1));
            .FPut (0;
                  (("lvalue"; "left"); ("rvalue"; "right")); .F
                  (
                        .Nth l (.FIn (); (1; 0));
                        .Wr (.T ("I"; 4); l; r);
                        .FOut (0; (("address"; l)))
                  ));
                  ...
      )
)
```

Листинг 2

Основной причиной возникновения необходимости исправления синтаксиса послужили отзывы разработчиков: многие указали на излишнее количество символов «;» как на основное неудобство синтаксиса.

Так как одной из целей создания компилятора LiME было облегчение разработки, а немалая доля ядра состоит из описанных на внутреннем представлении форм, соответствующих синтаксическим элементам, то синтаксис необходимо максимально упростить и сделать наиболее читаемым, чтобы не отпугнуть потенциальных разработчиков. Придумывать новый нерационально: удобство синтаксиса является очень субъективным, поэтому нужно что-то, привычное наибольшему числу людей. Промежуточное представление является текстовым описанием дерева: уже существующим и общепринятым здесь является синтаксис S-выражений, поэтому также было принято решение сделать синтаксис внутреннего представления наиболее близким к общепринятому.

Итак, задачей являлось улучшить грамматику промежуточного представления до такого вида:

$G \rightarrow$ “( )” | “(” A “)”	Граф
$A \rightarrow a \mid a\ A$	Список атрибутов
$a \rightarrow G \mid n \mid num \mid atom \mid id$	Атрибут
$n \rightarrow$ “.” $id\ G$ | “.” id “:” $id\ G$	Узел, напр.: `(.S:n5  '02.1."b" n1)`
num	Натуральное число в десятичной записи
atom	Атом
id	Идентификатор

Промежуточное представление должно в итоге выглядеть так:

```
(
     (.FEnv ("L" '44.1."=")
          (.F
               (.Nth:op (.FIn) (1))
               (.FPut 0 (("lvalue" "left") ("rvalue" "right"))
                    (.F
                         (.Nth:l (.FIn ) (1 0))
                         (.Nth:r (.FIn ) (1 1))
                         (.wr (.T "I" 4) l r)
                         (.FOut 0 ("address" l))
                    )
               )
          )
          ...
     )
)
```

Листинг 3

Как видно из примера, изменения позволяют:

1. Гораздо проще определять имя узла
2. Отделять атрибуты от имени узла (в 7-й строчке отчетливо видно, что атрибутов у узла 2: новый узел и список.
3. Пользоваться встроенной подсветкой языка S-выражений

Детали реализации

Часть, отвечающая за работу с промежуточным представлением, разделена на 2 части, обе написаны на языке Си: загрузка графа из текстового формата во внутренние структуры на Си (`loaddag.c`) и выгрузка графа из внутренних структур в текстовый формат (`dump.c`, функция `dumpdag`).

Было решено сначала изменить выгрузку графа, так как это гораздо проще, чем загрузка. Здесь отсутствует разбор синтаксиса: при обходе графа в файл выдаются лишь фиксированные конструкции, поэтому необходимо поменять лишь их формат. Например, чтобы убрать точку с запятой из вывода, нужно лишь удалить следующие строчки, проверяющие, что элемент из списка атрибутов завершен и пора ее выводить:

```
if(l != st->L)
{
     assert(fputc(';', f) == ';');
}
```

Также нужно было изменить формат вывода узла. Здесь необходимо было поменять только строку самого формата с

```
assert(0 < fprintf(f, "\n%s\t%p\t.%s\tn%u",
    st->tabstr,
    (void *)n.u.list,
     atombytes(atomat(U, nodeverb(n, NULL))), i.u.number));
```

на

```
assert(0 < fprintf(f, "\n%s\t%p\t(.%s:n%u",
     st->tabstr,
     (void *)n.u.list,
     atombytes(atomat(U, nodeverb(n, NULL))),
     i.u.number));
```

, одновременно перенося скобку перед типом узла и ставя метку после него, через двоеточие.

С загрузкой графа из файла все оказалось далеко не так просто. Здесь входной файл считывается последовательно, вызывая цепочки из функций в

зависимости от встречаемых символов. Данные накапливаются в структурах и передаются между функциями через аргументы:

- Текущий контекст: `const LoadContext *const ctx`. Структура `LoadContext`

```
typedef struct
{
	FILE *const file;
	const unsigned fileatom;
	Array *const universe;
	const Array *const dagmap;
} LoadContext;
```

  содержит в себе такие параметры, как дескриптор файла, постоянно используемый для чтения следующих символов из файла, идентификатор этого файла, список узлов, которые являются подграфами.
- Текущее окружение: `List *const env`. Содержит текущее окружение меток.
- Список накопленных узлов: `List *const nodes`
- Список ссылок: `List *const refs`

Основной проблемой оказалось переставление открывающейся скобки внутри синтаксиса узла. Ранее она означала только одно: открытие списка, будь до просто список, или список атрибутов узла. Теперь же она может значить 2 вещи: создание списка или создание узла. Это является проблемой по следующим причинам:

- Открытие списка означает создание нового окружения и сброс списка ссылок. При создании нового узла его метка помещается в окружение того уровня, где создается этот узел, а все его атрибуты попадают уже в новое окружение. Пока скобка стояла после создания узла, проблемы не было: узел создается, попадает в текущее окружение, затем создается список и последующие атрибуты попадают в новое окружение. Теперь, если не

менять логику, то сначала создается новое окружение, а затем в него попадает как сам узел, так и его атрибуты, что совершенно неверно.

- Функция загрузки графа (`loaddag`) вызывается в самом начале и проверяет наличие открывающейся скобки: граф может начаться только с нее. Однако она может быть вызвана, если встретился подграф, то есть частный случай узла. Раньше это не было проблемой, так как список атрибутов узла начинался со скобки в любом случае.

Чтобы реализовать нужные изменение в синтаксисе, пришлось несколько поменять логику разбора. Функционал создания списка и создания узла находился в разных функциях: `list` и `core` соответственно (приложение В). Так как открывающаяся скобка теперь может значить создание и того, и другого, было принято решение объединить эти 2 части в одну функцию и проводить соответствующие проверки там, такие как:

- Наличие открывающейся скобки
- Проверка на пустой список
- Если далее за скобкой идет символ точка, значит это узел: нужно добавить его к текущему окружению, создать новое, и передать его дальше.
- В остальных случаях создается список так же, как раньше.

Результат объединения этих частей показан в приложении Г. Часть, отвечающая за создание узла в функции `core`, соответственно, удалена. Также удалена проверка на открывающуюся скобку из функции `loaddag`.

Далее, нужно было избавиться от символов «;», что тоже оказалось не очень тривиально. Ранее этот символ означал конец атрибута, узла или списка, однако после последнего атрибута в списке он не ставился. Чтобы не переносить функционал на пробельные символы, которые, обычно, просто пропускаются, принято решение проверять следующий символ, идущий за предполагаемым концом атрибута: если далее идет закрывающаяся скобка, то вызывается

функционал закрытия списка, иначе – управление передается функции `core`, которая попытается обработать следующий атрибут. Листинг оригинальной функции и измененной приведен в приложении Д.

Последнее, что нужно было сделать, это изменить представление метки узла, чтобы она указывалось через двоеточие после типа узла. Данная часть находится в функции `node`. Здесь нужно было после прочитывания типа узла заменить пропускание пробелов и проверку следующего токена на проверку на наличие двоеточия. Результат замены показан в приложении Е.

Заключение

В результате проделанной работы был изменен синтаксис промежуточного представления компилятора LiME. Задачей являлось сделать его более простым и понятным для как можно большего числа людей. Так как удобство языка и, в частности, его синтаксиса является субъективным свойством, то было решено сделать синтаксис близким к общепринятому, то есть к синтаксису языка S-выражений, задачей которого также является описание деревьев.

В итоге, синтаксис промежуточного представления был упрощен за счет:

- Упразднения символов «;», что позволило сделать языковые конструкции менее загроможденными и несколько сократить объем кода.
- Переноса открывающейся скобки при описании узла, что позволило приблизить синтаксис к общепринятому и открыло возможность пользоваться удобствами различных текстовых редакторов, такими как подсветка синтаксиса.
- Указания метки узла через двоеточие, что повышает видимость этой метки и отделяет ее от остальных атрибутов узла.

Компилятор LiME является перспективным проектом, но требует продолжения разработки. Улучшения со стороны удобства и простоты разработки над проектом позволят сделать его более привлекательным для сторонних разработчиков.

Список литературы

1. Лацис А.О. «Параллельная обработка данных», 2010 г.
2. «Мультиклеточный процессор – это что?» https://habrahabr.ru/post/226773/
3. «Мультиклет R1 – первые тесты» https://geektimes.ru/post/264422/
4. «Компилятор C/C++ на базе LLVM для мультиклеточных процессоров: быть или не быть?» https://habrahabr.ru/post/302776/
5. «РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ. Программное обеспечение мультиклеточного процессора MultiClet R1» http://multiclet.com/docs/PO/Manual_Soft_R1.pdf
6. Бахтерев М.О. «Конструктор компиляторов LiME. Технический отчёт.»

Приложение А

Листинг функции `sha256_transform` на языке Си.

```
void sha256_transform(SHA256_CTX *ctx, const BYTE data[])
{
	WORD a, b, c, d, e, f, g, h, i, j, t1, t2, m[64];

	for (i = 0, j = 0; i < 16; ++i, j += 4)
		m[i] = (data[j] << 24) | (data[j + 1] << 16) | (data[j + 2] << 8)
		| (data[j + 3]);
	for ( ; i < 64; ++i)
		m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];

	a = ctx->state[0];
	b = ctx->state[1];
	c = ctx->state[2];
	d = ctx->state[3];
	e = ctx->state[4];
	f = ctx->state[5];
	g = ctx->state[6];
	h = ctx->state[7];

	for (i = 0; i < 64; ++i) {
		t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
		t2 = EP0(a) + MAJ(a,b,c);
		h = g;
		g = f;
		f = e;
		e = d + t1;
		d = c;
		c = b;
		b = a;
		a = t1 + t2;
	}

	ctx->state[0] += a;
	ctx->state[1] += b;
	ctx->state[2] += c;
	ctx->state[3] += d;
	ctx->state[4] += e;
	ctx->state[5] += f;
	ctx->state[6] += g;
	ctx->state[7] += h;
}
```

Приложение Б

Листинг части функции `sha256_transform` на ассемблере Мультиклета.

```
sha256_transform:
                    jmp LBB0_1
                    setl #SP, #SP, -608
complete

LBB0_1:
                    jmp LBB0_3
      SR4    :=     exal #SP, 352
                    wrq @0, #SP, 344
                    wrq @SR4, #SP, 248
complete

LBB0_5:
                          jmp LBB0_6
      SR4           :=    rdl #SP, 612
      SR5           :=    addl @SR4, 0x50
      SR6           :=    addl @SR4, 0x54
      SR7           :=    addl @SR4, 0x58
      SR8           :=    addl @SR4, 0x5c
      SR9           :=    addl @SR4, 0x60
      SR10   :=     addl @SR4, 0x64
      SR11   :=     addl @SR4, 0x68
      SR12   :=     addl @SR4, 0x6c
      SR4           :=    rdl @SR5
      SR13   :=     rdl @SR6
      SR14   :=     rdl @SR7
      SR15   :=     rdl @SR8
      SR16   :=     rdl @SR9
      SR17   :=     rdl @SR10
      SR18   :=     rdl @SR11
      SR19   :=     rdl @SR12
                    wrq @0, #SP, 320
                    wrq @SR4, #SP, 312
                    wrq @SR13, #SP, 304
                    wrq @SR19, #SP, 296
                    wrq @SR18, #SP, 288
                    wrq @SR17, #SP, 280
                    wrq @SR16, #SP, 272
                    wrq @SR15, #SP, 264
                    wrq @SR14, #SP, 256
                    wrq @SR4, #SP, 240
                    wrq @SR5, #SP, 224
                    wrq @SR13, #SP, 216
                    wrq @SR6, #SP, 192
                    wrq @SR14, #SP, 184
                    wrq @SR7, #SP, 160
                    wrq @SR15, #SP, 152
                    wrq @SR8, #SP, 128
                    wrq @SR16, #SP, 120
                    wrq @SR9, #SP, 96
                    wrq @SR17, #SP, 88
                    wrq @SR10, #SP, 64
                    wrq @SR18, #SP, 56
                    wrq @SR11, #SP, 32
                    wrq @SR19, #SP, 24
                    wrq @SR12, #SP
complete
```

Приложение В

Листинг функции `list` и части функции `core` до изменений.

```
static LoadCurrent list(
	const LoadContext *const ctx, List *const env, List *const nodes)
{
	assert(env == NULL || iskeymap(env->ref));
	assert(nodes == NULL || isnode(nodes->ref));

	FILE *const f = ctx->file;
	assert(f);

	const int c = skipspaces(f);
	switch(c)
	{
	case ')': // В случае, если список пустой
		return LC(nodes, NULL);
	}

	if(isfirstcore(c))
	{
		assert(ungetc(c, f) == c);
		List *lenv = pushenv(env);
		const LoadCurrent lc = core(ctx, lenv, nodes, NULL);
		assert(popenv(&lenv) == env);
		return lc;
	}
	errexpect(c, ES("(", ".", "'", "[0-9]+", "[A-Za-z][0-9A-Za-z]+", ")"));
	return LC(NULL, NULL);
}

static LoadCurrent core(
	const LoadContext *const ctx,
	List *const env, List *const nodes, List *const refs)
{
	...

	switch(c)
	{
		case '.':
		{
			const LoadCurrent lc = node(ctx, env, nodes);
			List *const l = lc.refs;
			assert(l && isnode(l->ref) && l->next == l);
			return ce(ctx, env, lc.nodes, append(refs, l));
		}
		...
	}
	...
}
```

Приложение Г

Листинг результата объединения – новой функции `listornode`.

```
static LoadCurrent listornode(
	const LoadContext *const ctx, List *const env,
	List *const nodes, List *const refs)
{
	assert(env == NULL || iskeymap(env->ref));
	assert(nodes == NULL || isnode(nodes->ref));

	FILE *const f = ctx->file;
	assert(f);
	// Наличие открывающейся скобки:
	int c;
	if((c = skipspaces(f)) != '(')
	{
		errexpect(c, ES("("));
	}
	c = skipspaces(f);
	char str[2] = {c, 0};
	fprintf(stderr, "listornode(%s)\n", str);
	switch(c)
	{
	case ')': // В случае, если список пустой
		return LC(nodes, NULL);

	case '.':
		const LoadCurrent lc = node(ctx, env, nodes);
		List *const l = lc.refs;
		assert(l && isnode(l->ref) && l->next == l);
		List *lenv = pushenv(env);
		return ce(ctx, lenv, lc.nodes, append(refs, l));
	}

	if (isfirstcore(c))
	{
		assert(ungetc(c, f) == c);
		List *lenv = pushenv(env);
		const LoadCurrent lc = core(ctx, lenv, nodes, NULL);
		assert(popenv(&lenv) == env);
		return lc;
	}
	errexpect(c, ES("(", ".", "'", "[0-9]+", "[A-Za-z][0-9A-Za-z]+", ")"));
	return LC(NULL, NULL);
}
```

Приложение Д

Листинг функции завершения атрибута до и после изменений.

```
// До изменений:
static LoadCurrent ce(
	const LoadContext *const ctx,
	List *const env, List *const nodes, List *const refs)
{
	assert(env && iskeymap(env->ref));
	assert(nodes == NULL || isnode(nodes->ref));

	const int c = skipspaces(ctx->file);
	switch(c)
	{
	case ')':
		return (LoadCurrent) { .nodes = nodes, .refs = refs };

	case ';':
		return core(ctx, env, nodes, refs);
	}
	errexpect(c, ES(")", ";"));
	return (LoadCurrent) { .nodes = NULL, .refs = NULL };
}


// После изменений:
static LoadCurrent ce(
	const LoadContext *const ctx,
	List *const env, List *const nodes, List *const refs)
{
	assert(env && iskeymap(env->ref));
	assert(nodes == NULL || isnode(nodes->ref));

	const int c = fgetc(ctx->file);
	int posc = ftell(ctx->file);
	assert(ungetc(c, ctx->file) == c);
	const int next = skipspaces(ctx->file);
	int posnext = ftell(ctx->file);
	assert(ungetc(next, ctx->file) == next);

	char str[2] = {c, 0};
	str[0] = next;

	switch(next)
	{
		case ')':
			fgetc(ctx->file);
			return (LoadCurrent) { .nodes = nodes, .refs = refs };
	}
	return core(ctx, env, nodes, refs);
}
```

Приложение Е

Листинг части функции загрузки узла `node`.

```
// До изменений:
c = skipspaces(f);
assert(ungetc(c, f) == c);
if(isfirstid(c))
{
	const Ref label = loadtoken(U, f, 0, "[0-9A-Za-z]");
	ref = keytoref(env, label);

	if(ref->code == FREE)
	{
		assert(ref->u.pointer == NULL);
	}
	else
	{
		ERR("node label is in scope: %s",
			atombytes(atomat(U, label.u.number)));
	}
}


// После изменений:
c = skipspaces(f);
if (c == ':')
{
	const Ref label = loadtoken(U, f, 0, "[0-9A-Za-z]");
	ref = keytoref(env, label);

	if(ref->code == FREE)
	{
		assert(ref->u.pointer == NULL);
	}
	else
	{
		ERR("node label is in scope: %s",
			atombytes(atomat(U, label.u.number)));
	}
}
else
{
	assert(ungetc(c, f) == c);
}
```