Санкт-Петербургский государственный университет

Программная инженерия
Кафедра Системного программирования

Митенев Алексей Васильевич

Реализация структур и указателей в трансляторе РуСи

Бакалаврская работа

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Рецензент:
д.т.н., профессор Новиков Ф. А.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering
Software Engineering department

Aleksey Mitenev

Implementation of structures and pointers for RuC translator

Bachelor's Thesis

Scientific supervisor:
professor Andrey Terekhov

Reviewer:
professor Fedor Novikov

Saint-Petersburg
2016

Оглавление

Введение

В настоящее время давно стало очевидным, что программировать в машинных кодах или в кодах ассемблера не слишком удобно, особенно для начинающих программистов. В современных языках программирования высокого уровня используются операторы языка – конструкции, записанные в более человеческом виде. Программировать на таких языках прозоляют трансляторы[5] – программы, которые преобразуют входную программу, написанную на языке высокого уровня, в машинный код, либо в коды языка более низкого уровня. В настоящее время большинство языков программирования поддерживают только английский язык, то есть в тексте программы, включая ключевые слова, идентификаторы переменных и сообщения об ошибках используются только английские буквы.

Транслятор РуСи разрабатывается на кафедре системного программирования математико-механического факультета. Транслятор РуСи может быть полезен для школьников, которые не знают английский язык, так как транялятор РуСи дает возможность программировать на всем известном языке C [1], используя русские буквы для ключевых слов языка и именования переменных, ошибки так же выводятся на русском языке.

РуСи имеет ряд внутренних особенностей и преимуществ перед обычным языком C, при этом является его подмножеством. РуСи написан на языке C[2], что дает возможность реализовать его более оптимально и быстро, и, кроме того, использовать на любых платформах. РуСи транслирует входную программу в коды собственной виртуальной машины, кроме того, есть возможность транслировать в коды LLVM [6]. В РуСи уже реализовано большинство конструкций языка C: работа с переменными, массивами, указателями и функциями. Но, ранее, в РуСи можно было использовать только базовые типы, такие как float, int, char. Чтобы приблизить язык транслятора к языку C и увеличить подмножество конструкций, было решено расширить транслятор и реализовать на нем работу со структурами[3].

В программировании иногда бывает удобно использовать одно имя для обозначения группы переменных, когда эти переменные имеют общее назначение. Структуры предоставляют возможность хранения некоторого набора различных значений, объединенных одним общим названием. Это увеличивает модульность программы и компактность программного кода. Структуры, как правило, используют тогда, когда в коде программы имеется большое количество родственных данных и их нужно сгруппировать вместе.

По этим причинам была поставлена задача: реализовать обработку структур в трансляторе РуСи в соответствии с тем, как они работают в языке С. Более подробно, необходимо реализовать работу с переменными структурного типа, работу с массивами структур и указателями на структуры, возможность создания функций, которые возвращают структуры или принимают структуру в качестве аргумента, то есть, все возможности базовых типов языка. Для достижения этой цели были поставлены следующие задачи:

- модифицировать лексический анализатор в соотвестствии с поставленной задачей;
- модифицировать синтаксический анализатор;
- внести изменения в видозависимый анализ;
- модифицировать генерацию кода виртуальной машины.

1. Обзор

1.1. Обзор реализации транслятора РуСи

Поскольку транслятор РуСи уже был реализован перед выполнением работы и имеет множество особенностей реализации, имеет смысл рассказать об этих особенностях и об устройстве транслятора в целом в целях улучшения понимания задачи. Общая схема транслятора отражена на Рис. 1. Трансляция осуществляется в два просмотра: на первом просмотре работает лексический, синтаксический и видозависимый анализ и проверяется корректрость введенной программы. После первого просмотра мы имеем промежуточное представление(дерево) программы, а также некоторые заполненные вспомогательные структуры. На втором просмотре происходит генерация кодов виртуальной машины из этого промежуточного представления. Ниже будут рассмотрены основные компоненты транслятора РуСи.

1.1.1. Вспомогательные таблицы транслятора

Прежде, чем рассказывать о работе основных частей транслятора, стоит описать вспомогательные таблицы транслятора. В них хранится информация, необходимая для трансляции. Эта информация последовательно накапливается в процессе трансляции.

Таблица представлений(reprtab) – таблица, в которой хранятся посимвольные представления идентификаторов. Таблица представлена массивом целых чисел. Каждый идентификатор описывается следующим образом:

1. Ссылка на предыдущий идентификатор с таким же значением хэш функции.

2. Отрицательное число, соответствующее номеру ключевого слова, если идентификатор – ключевое слово, либо положительное число – ссылка на таблицу идентификаторов identtab.

3. В следующих n ячейках – представление слова посимвольно, где n – количество букв в представлении.

Для упрощения поиска представления в таблице используется хэш функция, которая применяется к строке идентификатора. Представления с одинаковым хэшем связываются в цепной список.

Таблица идентификаторов(identtab) представлена массивом целых чисел и хранит в себе информацию об идентификаторах переменных и функций. Информация о каждом идентификаторе представлена в 4 словах:

1. ссылка на представление этого идентификатора в таблице reprtab;
2. ссылка на идентификатор с тем же представлением в объемлющем блоке(ссылка указывает на ячейку, соответствующую идентификатору в таблице identtab);
3. тип идентификатора (для функций – тип возвращаемого значения);
4. подсчитанное смещение в статике функций виртуальной машины для переменной либо ссылка на таблицу видов (modetab), если идентификатор - имя функции.

Таблица видов(modetab) представлена массивом целых чисел и содержит расширенные описания сигнатур функций. Каждая функция опиывается следующей информацией:

1. тип возвращаемого значения функции;
2. количество параметров функции – n;
3. далее представлены типы аргументов функции: для каждого из n параметров - его тип;

Таблица функций (functab) содержит информацию о функциях, например, ссылку на начало статики в памяти (высчитывается в процессе трансляции), ссылки на представления аргументов функций.

1.1.2. Лексический анализатор

Лексический анализатор – часть транслятора, которая работает напрямую с текстом входной программы и превращает его в набор лексем - последовательностей допустимых символов языка программирования, имеющая смысл для транслятора. Чтобы ускорить работу транслятора, в РуСи функции лексического анализатора вызываются из синтаксического анализа, чтобы узнать текущую лексему и следующую лексему, после чего лексический анализатор переключается на следующую лексему, делая ее текущей. Этого заглядывания вперед на одну лексему оказывается достаточно для синтаксического анализа такого сложного с виду языка, как язык Си. В случае, если следующая лексема является идентификатором, определяется, какой именно идентификатор был встречен. Для этого используется таблица представлений(reprtab). Если идентификатор встречается первый раз, то его представление добавляется в таблицу, иначе представление ищется в таблице.

В тот момент, когда лексический анализатор обрабатывает следующую лексему и эта лексема является идентификатором, ее символьное описание заносится в reprtab. Символьные описания в таблице представлений уникальны. Для ускорения поиска в таблице представлений используется хэш функция. Представления, имеющие одинаковый хэш связываются в цепной список. У представлений, соответствующих идентификаторам, ставится ссылка на identtab, чтобы по представлению можно было узнать, какому идентификатору оно принадлежит.

Ключевые слова в РуСи находятся в отдельном файле, и в начале трансляции лексический анализатор применяется для того, чтобы обработать этот файл и занести все ключевые слова в reprtab. Таким образом, когда во время обработки входной программы лексический анализ встретит идентификатор ключевого слова, он найдет его представление в reprtab и поймет, какую именно лексему нужно отдать.

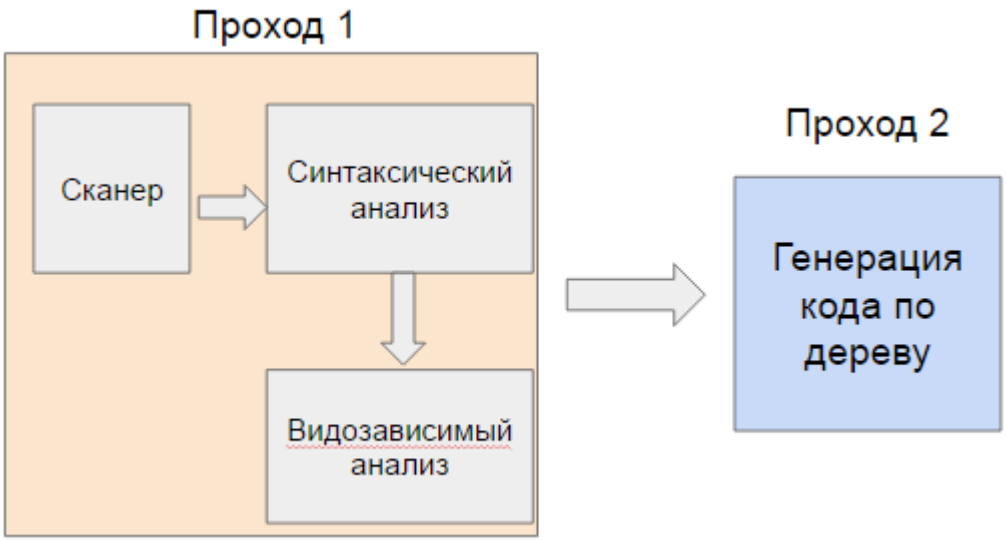

Рис. 1: Схема транслятора РуСи

1.1.3. Синтаксический анализ

Синтаксический анализ в РуСи осуществляется методом рекурсивного спуска[4]. Рекурсивный спуск – алгоритм нисходящего синтаксического анализа, реализуемый путём взаимного вызова процедур парсинга, где каждая процедура соответствует одному из правил контекстно-свободной грамматики или БНФ. Применения правил последовательно, слева-направо поглощают лексемы, полученные от лексического анализатора. Это один из самых простых алгоритмов парсинга, подходящий для полностью ручной реализации. Синтаксический анализ вызывает лексический анализатор для того, чтобы узнать текущую и следующую лексему, на выходе выдает промежуточное представление, представляющее программу. Во время синтаксического анализа происходит заполнение identtab. Кроме того, вместе с синтаксических анализом происходит и видозависимый анализ и заполнение таблицы modetab. После синтаксического анализа, получения дерева и заполнения вспомогательных таблиц начинается стадия генерации кода.

1.1.4. Генерация кода и исполнение

На стадии генерации кода происходит преобразование дерева программы в коды виртуальной машины. После этого все данные, необходимые для исполнения кода готовы, и можно передавать их для исполнения на виртуальной машине. Для исполнения в РуСи реализована стековая виртуальная машина. Вся работа происходит в массиве целых чисел mem. В начале в массиве располагаются сгенерированные коды виртуальной машины. Далее идет область глобальных данных. Каждая глобальная переменная получает здесь свою ячейку для хранения значения. На начало области глобальных данных указывает регистр g(global). Затем идет область статик функций. В статике функции выделяется место для аргументов функции и для локальных переменных. На начало текущей статики указывает регистр l(local). После каждой из статик идет динамический стек, на котором располагаются массивы, и на котором происходит работа виртуальной машины во время обработки соответствующей функции. Стек нужен для того, чтобы при вычислении выражения хранить промежуточные значения, и, кроме того, для хранения массивов. Массивы (локальные или глобальные) представляются ссылкой на адрес нулевого элемента массива на динамическом стеке. На текущую вершину стека указывает регистр x. Использование регистров l и g дает возможность указать, где переменная будет располагаться в памяти, используя всего одно число - смещение. Если смещение отрицательное, то смещение от регитра g, переменная глобальная. Если же смещение положительно, то смещение от регистра l, переменная локальная.

1.2. Структуры в языке Си

Для реализации структур в РуСи необходимо сначала разобраться в возможностях и конструкциях языка C, связанных со структурами. Информация взята из книги Керниган, Ричи “Язык Си”[1].

1.2.1. Объявление структур

В языке C для того, чтобы объявить переменную типа структуры, сначала нужно объявить тип этой структуры:

```
struct point {
  int x;
  int y;
};
```

После мы можем создать переменную с типом объявленной структуры:

```
struct point p;
```

Кроме того, имеется возможность для одновременного объявления типа структуры и объявляния переменной. В следующем примере сначала произойдет объявляние типа структуры point, а после создаются переменные этого типа: p1 – просто переменная, parray – массив структур, ppointer – указатель на структуру:

```
struct point {
  int x;
  int y;
} p1,  parray[10],  *ppointer;
```

Имеется возможность объявлять типы структур, которые не имеют имени. В этом случае все переменные этого типа должны быть определены сразу после определения типа:

```
struct {
  int x;
  int y;
} p1,  parray[10],  *ppointer;
```

При определении типа структуры мы можем использовать поля структурного типа, например:

```
struct line {
  struct point{
    Int x;
    Int y;
  } p1, p2;
} l1, larray[3];
```

В качестве полей структуры можно использовать массивы и указатели, в том числе и рекурсивно ссылаться на объявляемый тип:

```
struct stack
{
  struct stack *next;
  int value;
} st;
```

1.2.2. Работа со структурами

После создания переменной структурного типа нужно определиться с тем, как можно с ней работать. Работа с переменными и массивами структурного типа внешне мало отличается от работы с переменными базовых типов – int, float, char.

Переменную структурного типа можно инициализировать. В следующем примере в поле x переменной p будет присвоено значение 1, а в поле y - 2:

```
struct point {int x; int y;}  p = {1, 2};
```

Основное отличие в работе с переменными структурного типа заключается в возможности получения значения поля этой структурной переменной. Для этого используется операция “.”:

```
struct point {
  int x;
  int y;
```

```
} p;
p.x = 4;
p.y = 5;
```

При работе с указателем на структуру возможно присваивать или получать значение полей из исходной переменной с помощью операции “->”:

```
struct point {
  int x;
  int y;
} p, *ppointer;
ppointer  = p;
int a = ppointer -> x;
ppointer -> x = 5;
```

Кроме того, можно написать функцию, которая принимает параметр структурного типа или возвращает значение структурного типа:

```
struct line {
  struct point{
    int x;
    int y;
  } p1, p2;
};

struct line makeLine(struct point p1, struct point p2)
{
    struct line res = {p1, p2};
    return res;
}
```

2. Описание решения

Для реализации структур в трансляторе РуСи появилась необходимость внести изменения во все части транслятора: в лексический анализ, в синтаксический анализ, в видозависимый анализ, в генерацию кода.

2.1. Модификация лексического анализатора в трансляторе РуСи

В рамках работы над реализацией структур в РуСи лексический анализатор претерпел небольшие изменения: В файл с ключевыми словами были добавлены два новых ключевых слова: ”СТРУКТУРА”, ”ОПРТИПА”. ”СТРУКТУРА” соответствует ключевому слову ”struct”, а ”ОПРТИПА” – ”typedef”. Как и все ключевые слова в РуСи, новые слова можно писать либо полностью заглавными, либо полностью прописными буквами. Были введены две новые лексемы: лексема DOT, соответствующая операции “.” для взятия поля структуры и лексема ARROW, соответствующая операции “->” для взятия поля структуры через указатель.

2.2. Модификация синтаксического анализатора в трансляторе РуСи

В синтаксическом анализе была проделана работа по добавлению обработки конструкций грамматики языка C, которые связны с обработкой структур, а именно:

1. struct-or-union-specifier ::= struct-or-union identifier struct-declaration+ | struct-or-union struct-declaration+ | struct-or-union identifier;
2. struct-declaration ::= specifier-qualifier* struct-declarator-list;
3. struct-declarator-list ::= struct-declarator | struct-declarator-list , struct-declarator.

Таким образом, в программном коде появилась возможность использовать следующие конструкции:

1. Обычное объявление типа структуры:

```
структура точка {цел а; цел б;};
```

2. Обычное объявление типа структуры и определение переменной этого типа:

```
структура точка {цел а; цел б;} т1, т2;
```

3. Объявление типа структуры без имени типа:

```
структура {цел а; цел б;} т1, т1;
```

4. Объявление переменной уже использованного типа:

```
структура точка т1;
```

2.3. Изменение таблицы видов и видозависимого анализа

В видозависимом анализе была проделана наибольшая работа. Ранее в РуСи были представлены только базовые типы (int, float, char), массивы из элементов базовых типов, и указатели на базовые типы. По размерности массивов так же были введены ограничения: допускались только одномерные и двумерные массивы. Таким образом, получалось конечное, определенное количество типов, каждому из которых можно было сопоставить число. На этом строился весь видозависимый анализ. Структуры совершенно не вписывались в эту концепцию: типов структур можно создавать любое, заранее не определенное число, и кроме того, для каждого из этих типов структур необходимо иметь возможность создавать массивы и указатели. Было решено в рамках данной

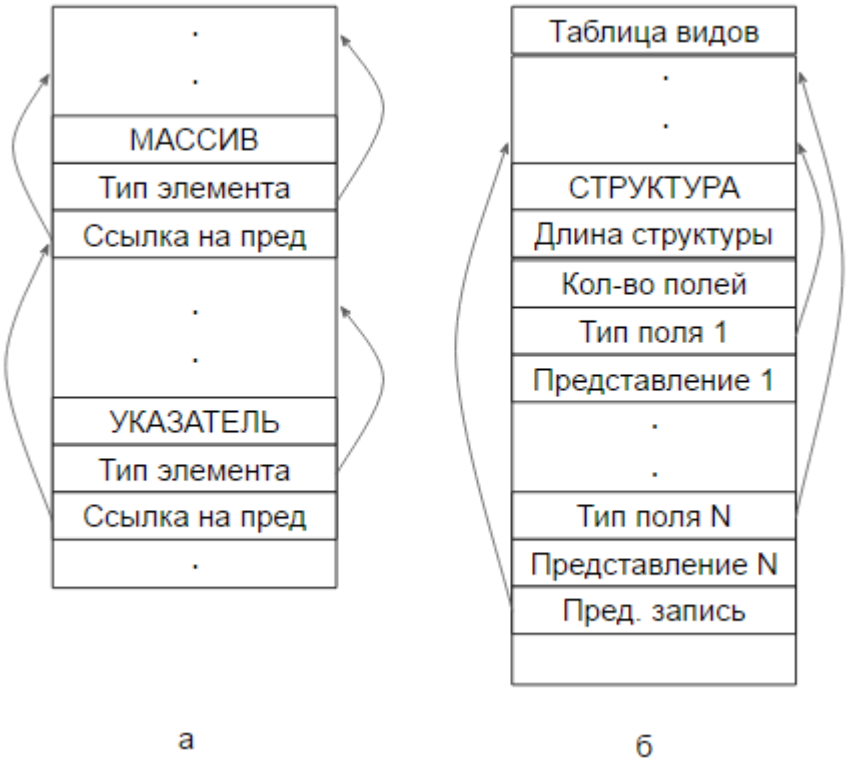

Рис. 2: Таблица видов

работы исправить так же и ограничение на размерность массивов. Сделано это было следующим образом: информация о типах структур, а так же об типах массивов и указателей была добавлена в таблицу видов modetab. Ранее в modetab хранились только записи для функции. На Рис. 2(а) схематически обозначены поля, которые описывают тип структуры в таблице modetab.

Сначала идет слово "Структура", которое определяет, что эта запись относится именно к типу структуры. После идет размер структуры – количество памяти, которое необходимо выделить для этой структуры во время исполнения программы. Размер структуры считается в функции как сумма размеров типов всех полей, которые содержатся в структуре. Соответственно, если поле имеет базовый тип, то его длина равна единице. Иначе, если поле структуры так же представлено структурным типом, его длина высчитывается рекурсивно так же с помощью этой функции.

Далее идет удвоенное количество полей в структуре. Для того, что-

бы объяснить, зачем нужно удвоение, для начала надо рассказать про проверку дублирования в modetab. После того, как в modetab заносится новый тип, происходит проверка на дублирование – если в modetab уже существует такой тип – новый тип занесен не будет, и вернется ссылка на старый тип. Удвоенное количество полей нужно, чтобы в функции проверки дублирования знать, сколько полей нужно будет проверять на равенство в функции проверки дублирования. Далее идет n пар: тип поля и ссылка на начало представления в таблице reprtab. После идет ссылка на предыдущую запись в modetab: записи в modetab связаны в цепной список как раз для проверки на дублирование: каждый раз происходит проход по этому списку и проверка на совпадение.

Теперь базовым типам сопоставляется отрицательное число (int: -1; float:-2; char: -3), остальные типы представлены положительным числом – ссылкой на modetab. Типы массивов и указателей так же хранятся в modetab. Устройство типа массива предствалено на Рис. 2(б). Сначала идет слово "МАССИВ", которое показывает, что эта запись относится к типу массива. Далее идет тип элемента массива. Тип элемента может быть ссылкой надругой тип массива в таблице modetab, этим и обеспечивается неограниченное увеличение размерности массива.

Стоит рассказать про то, как происходит определение типа. Прежде всего, была добавлена процедура gettype(), которая предназначена для получения типа при объявлении переменной или функции. Если текущая лексема соответствует базовому типу – int, float или char – возвращается соответствующий тип. Если же текущая лексема соответствует ключевому слову "СТРУКТУРА" – начинается обработка 3 возможных случаев:

1. Обычное объявление типа структуры. Пример:

   ```
   структура точка {цел а, б;}
   ```

   В данном случае следует разбор полей структуры и занесение информации в modetab. Представление имени структуры во время лексического анализа помещается в таблицу представлений

reprtab, кроме того, добавляется запись в identtab для того, чтобы при встрече того же имени типа структуры в следующий раз можно было понять, что такой тип уже объявлен.

2. Объявление типа структуры без имени типа. Пример:

```
структура {цел а, б;} с1, с2;
```

   В этом случае данные тоже заносятся в modetab, но никакого добавления данных в reprtab и identtab не происходит. Переменные, которые объявлены после определения типа структуры, получают тип - ссылку на modetab.

3. Объявление переменной или функции уже объявленного типа. Пример:

```
структура точка т1;
```

   В этом случае проверяется, добавлено ли представление "точка" в reprtab, если да - из представления в reprtab происходит переход по ссылке в identtab, в котором уже указан тип - ссылка на modetab. Если такого представления в reprtab нет - это ошибка. Подробная схема изображена на Рис. 3.

Таким образом, видозависимый анализ был значительно изменен, были добавлены типы структур и было убрано ограничение на размерность массива.

2.4. Изменения генерации кода виртуальной машины

Генерация кода проходит во время второго просмотра транслятора. На вход генератор кода виртуальной машины принимает промежуточное представление программы, по которому будет генерироваться код

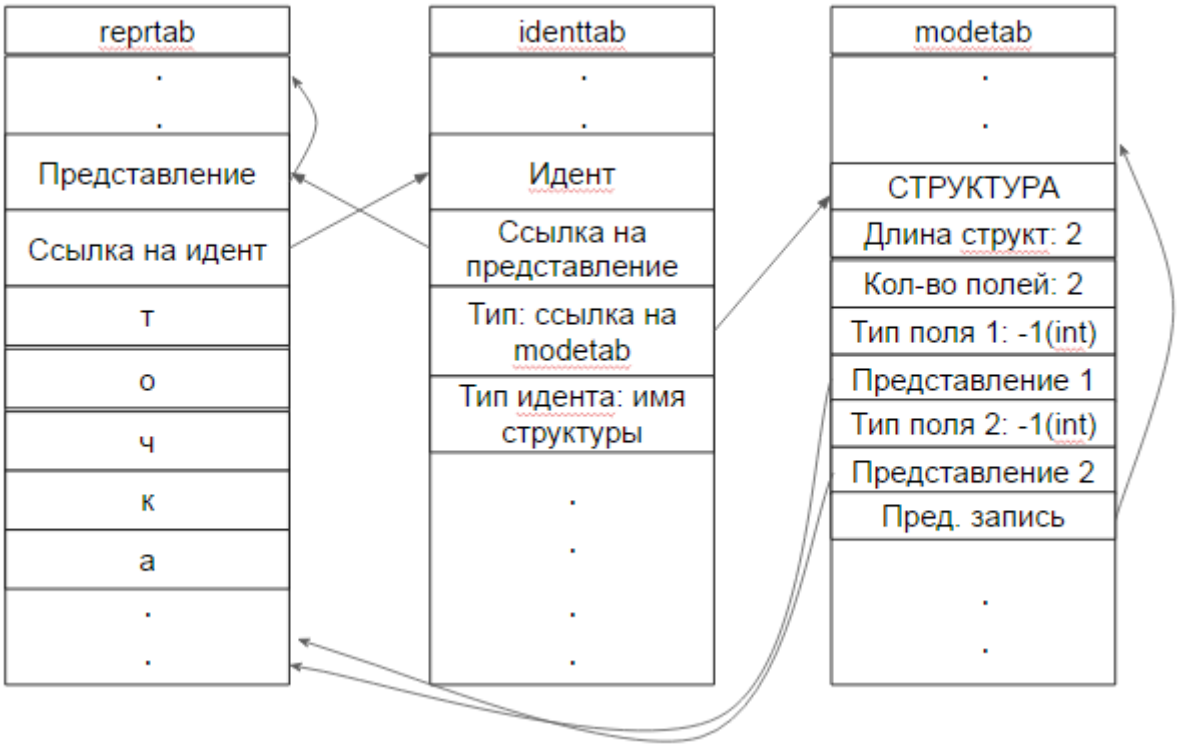

Рис. 3: Связь вспомогательных таблиц

виртуальной машины, а так же заполненные вспомогательные структуры. На данной стадии трансляции уже точно известно, что код корректен сточки зрения лексического, синтаксического и видозависимого анализа. Сначала нужно рассказать про то, как хранятся структуры в памяти виртуальной машины. Переменные структур хранятся в памяти так же, как и обычные переменные (локальные или глобальные), но они требуют больше памяти для хранения (количество ячеек памяти, требуемое для хранения переменной структурного типа подсчитывается во время трансляции и хранится в таблице modetab). Для обработки новых конструкций были добавлены новые команды виртуальной машины и изменены некоторые старые команды. Далее каждая команда будет разобрана подробнее.

2.4.1. Команда SELECT

Команда SELECT предназначена для получения адреса поля структуры. Команда SELECT ожидает, что на стеке будет лежать адрес структуры. Команда select принимает в качестве аргумента смещение

от начала структур до нужного поля и кладет на стек адрес нужного поля структуры. Кроме того, была сделана оптимизациядля команды SELECT: в случае, если подряд применяется более 1 взятия поля структуры, например:

```
СТРУКТУРА линия {
    СТРУКТУРА точка{
      цел коорд1, коорд2;
    } точка1, точка2;
} линия1;
линия1.точка2.коорд1 = 15;
```

то смещение поля коорд1 высчитывается сразу во время трансляции, и используется только одна команда: select 2. Смещение в данном случае 2, потому, что поле точка2 имеет смещение 2, а поле коорд1 имеет смещение 0.

2.4.2. Команды COPY и COPYAT

Ранее транслятор РуСи работал только с базовыми типами, и все переменные базового типа занимали одну ячейку в памяти(включая массивы и указатели, которые были просто ссылками). Переменные структурного типа могут занимать любое количество ячеек памяти, в зависимости от размера структуры. Структуры не являются ссылками на стек, поэтому для того, чтобы присвоить структуру в другую переменную, необходимо скопировать часть памяти, в которой лежит структура. Для этого нужно знать размер структуры и адрес начала. Кроме того, нужно учесть, что адрес памяти, куда нужно скопировать структуру, не всегда можно вычислить в процессе трансляции. Например, в случае присваивания структуры в переменную

```
СТРУКТУРА точка {
    ЦЕЛ a, б;
} т1, т2;
т1.a = 2;
```

```
т1.б = 3;
т2 = т1;
```

мы можем вычислить смещение переменной т2 в процессе трансляции, а в случае

```
СТРУКТУРА точка {
    ЦЕЛ а, б;
} т1, т2[2];
т1.а = 2;
т1.б = 3;
т2[1] = т1;
```

адрес памяти, куда нужно копировать, может быть вычислен только в процессе исполнения.

Учитывая вышесказанное, для копирования структур были реализованы 2 новые команды:

1. команда COPY, которая принимает в качестве аргумента размер структуры, а адрес мест в памяти, откуда и куда нужно копировать, берет на стеке.

2. команда COPYAT, которая принимает в качестве аргумента размер структуры и смещение переменной, в которую происходит копирование, относительно регистра l, если переменная локальная, или от регистра g, если переменная глобальная. Адрес памяти, откуда начинается копирование, находится на стеке.

2.4.3. Изменение в хранении массива в памяти

Ранее в РуСи было ограничение: поддерживались только одномерные и двумерные массивы. Для объявление массива и выделения под него места в памяти ранее использовались команды DEFARR и DEFARR2 для объявление одномерных и двумерных массивов соответственно. В

связи с изменениями в видозависимом анализе, а именно с изменениями, касающимися хранения типов массивов в таблице modetab, появилась возможность убрать это ограничение. Было решено оставить только одну команду DEFARR, которая служит для объявления и выделения памяти для массивов любой размерности. Команда DEFARR теперь принимает два параметра: размер элемента массива D и размерность массива N. На стеке ожидается N чисел: m1, m2, ... , mN, каждое из которых означает соответственно длину. Для следующего примера кода будет сгенерирована команда DEFARR N=2, D=2, а на стеке будут лежать числа 9, 7.

```
СТРУКТУРА точка {
    ЦЕЛ а, б;
} массив Т[7][9];
```

Кроме того, произошли изменения в хранении массива в памяти: ранее, перед нулевым элементом массива в памяти лежало число N - количество элементов массива. Теперь перед количеством элементов лежит размер элемента D. То есть, теперь массив хранится в памяти таким образом: D, N, A1, ... , AN, где последовательность A - элементы массива.

Заключение

Цель работы - реализация структур в трансляторе РуСи была достигнута, при этом следующие задачи были выполнены:

- в лексический анализ добавлены новые лексемы и ключевые слова;
- в синтаксический анализ добавлена обработка новых конструкций;
- были внесены значительные изменения в видозависимый анализ, переработана таблица видов;
- были внесены изменения в генерацию кода и добавлены новые команды виртуальной машины.

Выполнение данной работы увеличило подмножество языка Си, которое может обрабатывать транслятор РуСи. Таким образом, теперь транслятор может обрабатывать почти весь язык C, кроме арифметики над указателями и объединений (union), которые реализовывать не планируется. Кроме того, были убраны ограничения на размерность массивов и сделаны другие изменения в системе типов, что позволило сделать систему типов более гибкой и расширить видозависимый анализ.

Список литературы

[1] Kernighan Brain W., Ritchie Dennis M. The C programming language / Ed. by ATT Bell Laboratories. — Murray Hill, New Jersey : PRENTICE HALL, Englewood Cliff, 1978.

[2] Lee Jeff. C grammar. — 1985. — URL: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html.

[3] Mike Banahan Declan Brady Mark Doran. The C Book. — 1991. — URL: http://publications.gbdirect.co.uk/c_book/chapter6/structures.html.

[4] Wikipedia. Recursive descent parser. — 2012. — URL: https://en.wikipedia.org/wiki/Recursive_descent_parser.

[5] Wikipedia. Translator. — 2012. — URL: https://en.wikipedia.org/wiki/Translator_(computing).

[6] Болотов Сергей. Создание транслятора из РуСи в LLVM IR. — 2015. — URL: http://se.math.spbu.ru/SE/YearlyProjects/spring-2015/YearlyProjects/2015/371/371-Bolotov-report.pdf (дата обращения: 30.06.2015).