Компьютерные инструменты в образовании, 2016
№ 1: 36–47
УДК: 004.4'422, 004.4'427
http://ipo.spb.ru/journal

ИНСТРУМЕНТАЛЬНОЕ СРЕДСТВО ОБУЧЕНИЯ ПРОГРАММИРОВАНИЮ И ТЕХНИКЕ ТРАНСЛЯЦИИ

Терехов Андрей Николаевич

Аннотация

Проект РуСи задуман в качестве инструмента обучения программированию школьников, студентов и взрослых людей, которые решили освоить эту замечательную специальность. Первоначальным толчком была необходимость создать простое, наглядное, но достаточно мощное средство программирования роботов, затем задача была расширена на обучение алгоритмической грамотности и информатике. Наконец, оказалось, что получившийся компактный компилятор с языка С (с некоторыми ограничениями) в коды оригинальной виртуальной машины может быть с успехом использован в курсе «Трансляция» специальности Программная инженерия. Архитектура виртуальной машины проекта РуСи будет описана в отдельной статье.

Ключевые слова: *язык С, компилятор, виртуальная машина, обучение программированию, роботы, школьная информатика.*

1. ВВЕДЕНИЕ

Проект возник из потребностей преподавателей кружков робототехники. Уже довольно давно они применяют нашу графическую технологию ТРИК-студия, особенно удачно она подходит для школьников младших и средних классов, но при этом хотелось бы, чтобы школьники могли прочитать те программы, которые генерируются из графических диаграмм, то есть образовать своеобразный «мостик» к урокам информатики. Сейчас во многих странах популярен подход к программированию на основе графических моделей с автоматической генерацией кода на целевом языке. Традиционно для этого применяется язык С, но нельзя забывать, что школьники младших и средних классов практически не знают английского языка. И так-то учиться программировать тяжело, а тут — незнакомые слова, незнакомые сообщения и т. п. Поэтому возникла идея разработать компилятор языка С [1] в коды виртуальной машины с русскими сообщениями, ключевыми словами и идентификаторами и интерпретатор этой машины (но никто не запрещает использовать и английские ключевые слова и идентификаторы). Как компилятор (я назвал его РуСи), так и интерпретатор реализованы на стандартном С, поэтому легко переносятся на все платформы. Интерпретатор перенесен на конструктор роботов ТРИК [2, 3], который разрабатывается сотрудниками и студентами кафедр системного программирования и теоретической кибернетики математико-механического факультета СПбГУ.

Когда я уже начал продумывать этот проект, то сообразил, что даже без графических моделей он может быть с успехом использован не только в кружках по робототехнике, но и на уроках информатики. Насколько мы знаем, хороших средств для обучения программированию младших и средних школьников в России нет или их очень мало. Рисование диаграмм (идти не от текстов, а от графических представлений) — довольно хороший способ обучения программированию, но если говорить о традиционном составлении программ в виде текстов, то здесь на первый план выходят лёгкость использования и качество сообщений об ошибках. Например, в С есть массивы, но никак не контролируются выходы индексов за границы массива — это одна из самых трудно обнаруживаемых ошибок. Несомненно, такой контроль нужно иметь. Да и обычные синтаксические ошибки в существующих трансляторах не всегда точно привязаны к фактическому месту ошибки, а сообщения о них не разъясняют суть ошибки. В РуСи предусмотрено порядка 100 сообщений о синтаксических ошибках, каждое сообщение — это длинная фраза на русском языке, объясняющая причину ошибки, причем перед сообщением выводится текст исходной программы, обрывающийся на месте, где компилятор обнаружил ошибку.

Поскольку в настоящее время С относится к классу старых языков, есть огромное количество версий, стандартов, нововведений, реализаций. Авторы новых версий С стараются сохранить преемственность, с тем чтобы ранее написанные программы работали и на новых трансляторах. Например, параметры функции можно описывать прямо в заголовке, можно описывать как обычные переменные после заголовка, а можно вообще не описывать! Можно задавать только типы параметров, а можно и их идентификаторы. Последнее особенно забавно: я передаю функцию параметром в другую функцию, могу указать идентификаторы параметров функции–формального параметра, но никто и никогда не проверит соответствие этих идентификаторов параметрам функции–фактического параметра. Тогда зачем их указывать?

Для целей обучения программированию преемственность с древними трансляторами не нужна, я во всех таких случаях выбирал один наиболее разумный, на мой взгляд, вариант записи, при котором достигается наибольшая возможность контроля ошибок пользователя. Начинающие программисты часто ошибаются, наша задача — дать им понятное объяснение ошибки, как можно более точно привязанное к месту ошибки. Не во всех вариантах С это возможно.

Уже во введении хотелось бы указать основные отличия РуСи от базового языка С. В РуСи нет типов SHORT, LONG, SIGNED, UNSIGNED, DOUBLE. Я считаю, что для начала вполне достаточно INT, CHAR и FLOAT, в РуСи их обозначают как ЦЕЛ, ЛИТЕРА и ВЕЩ, то есть достаточно целых чисел, литер и вещественных чисел одинарной точности.

Нет статических и регистровых переменных. Регистровые переменные — это вообще анахронизм, современные оптимизирующие трансляторы распределяют регистры намного лучше, чем любой пользователь. Статические переменные — вещь неплохая: внутри функции описывается какая-то переменная, ей присваивается значение, при выходе из функции значение сохраняется, а при следующем входе в эту же функцию оно вновь используется. Но здесь вполне можно обойтись глобальной переменной, и пусть надежность несколько уменьшается, зато целой концепцией языка становится меньше.

Указатели — одна из наиболее трудных для понимания языковых черт С. Совсем без них не обойтись, так как любая функция, массив или структура является указателем. Но можно вполне обойтись без арифметики над указателями — одним из основных источников ошибок.

РуСи является точным подмножеством C, при замене ключевых слов и имен переменных на английские все должно работать на любом компиляторе, удовлетворяющем требованиям стандарта ANSI C. Поскольку РуСи — все-таки язык для целей обучения, а не промышленный, то я решил уменьшить разнообразие его черт, чтобы облегчить понимание программ на языке РуСи (и базовом языке C), например, я не планирую реализацию UNION.

2. КАК УЧИТЬ ПРОГРАММИРОВАНИЮ И АЛГОРИТМИЧЕСКОЙ ГРАМОТНОСТИ

У меня есть некоторый опыт преподавания в школах, правда, только в математических, и большой опыт преподавания в Университете. Всегда остро стоит вопрос, на каких примерах учить. Традиционно учат программированию на примерах решения уравнений и систем уравнений, численного интегрирования и других расчетов. Понятно, что для школ это мало пригодно. Я всегда начинал с таких задач:

- Поиск числа в неупорядоченном массиве.
- Поиск числа в упорядоченном массиве.
- Сортировка массива методом пузырька.
- Сортировка массива методом фон Неймана.

На этих примерах можно хорошо показать основы теории сложности алгоритмов, для детей важно понимать, что не все алгоритмы работают одинаково быстро, что, даже если программа выдает правильный ответ на маленьком примере, она может работать очень долго на большом примере. Сложность перечисленных выше примеров для массива длины n составляет:

– для неупорядоченного массива $n/2$ (в среднем);
– для упорядоченного массива $\log(n)$;
– сортировка пузырьком $n^2/2$;
– сортировка по Нейману $n \cdot \log(n)$.

Эти абстрактные формулы не производят впечатления на обучаемых, но стоит показать им конкретные значения для n, равного 1000 или 1000000, сразу все становится на свои места.

Дальше надо бы переходить на изучение более сложных структур данных типа деревьев, произвольных графов, множеств, списков и алгоритмов над ними. Школьников-олимпиадников готовят именно так, но обычным ученикам эти задачи без реальных практических примеров скучны. Я это понял еще много лет назад и в начале 70-х годов начал вести кружок по технике трансляции алгоритмических языков. Транслятор — хороший пример для обучения: видно, что на входе, что на выходе (особенно если выходом является не код какой-то конкретной ЭВМ, а высокоуровневый код виртуальной машины), используются самые разные алгоритмы и методы, например хэш-таблицы, рекурсивный спуск, деревья промежуточных представлений с различными вариантами их обхода, заглядывание вперед без возвратов и многие другие.

Обучение технике трансляции на примере больших промышленных компиляторов затруднено из-за огромного количества технических деталей. РуСи для этой цели подходит значительно лучше, я использовал его уже дважды как основу практики школьников Лицея 239, Академической гимназии (45 интерната) и студентов мат-меха. Более того, перед вторым годом я переработал первоначальный однопросмотровый вариант на

два просмотра — анализ и генерация кода с промежуточным представлением в виде линейной развертки дерева, именно чтобы облегчить восприятие и показать еще несколько техник.

Поскольку транслятор получился довольно стройный и относительно короткий по тексту, то его оказалось удобно использовать для целей обучения по курсу «Трансляция». Дело в том, что в международных программах по Программной инженерии и Информатике, а следовательно, и в программах учебных дисциплин нашего Университета на третьем курсе отводится 2 часа в неделю на лекции по теории трансляции (именно на устройство трансляторов, а автоматы, грамматики и тому подобные вещи изучаются в отдельных курсах) и два часа на практику. Причем в международных рекомендациях по курсу трансляции написано, что это никак нельзя все сводить к работе у доски, рассказывать просто об устройстве составных частей транслятора. Обязательно надо, чтобы студенты получали какие-то практические задания и разбирали какие-то конкретные трансляторы.

3. ОПИСАНИЕ ТРАНСЛЯТОРА РуСи

Транслятор РуСи имеет традиционную двух просмотровую структуру — на первом просмотре работает сканер, то есть лексический анализатор, видонезависимый анализатор (как теперь говорят — парсер) и видозависимый анализатор. Результатом первого прохода является дерево синтаксического разбора. Это дерево подается на вход второму просмотру кодогенерации, который выдает результат в кодах виртуальной машины. Почему я остановился на генерации виртуальной машины, а не кодов какой-то одной конкретной аппаратной архитектуры? Дело в том, что в разных школах и разных вузах используют разную вычислительную технику с разными операционными системами, поэтому трудно угодить всем. А виртуальная машина хороша тем, что если у вас есть интерпретатор виртуальной машины, написанный на языке С, то его очень легко перенести на MacOS, Windows, Linux и другие операционные системы. Кроме того, виртуальная машина обычно существенно проще и не содержит многих технических деталей, которые затемняют суть дела и существенно усложняют разъяснение работы генерирующей части транслятора. Поэтому для целей обучения, несомненно, виртуальная машина лучше.

Разберем составные части более детально.

3.1. Сканер

На вход транслятора РуСи поступает набор литер. Надо сказать, что даже здесь я столкнулся с проблемами — в Linux и MacOS наиболее популярной на данный момент кодировкой литер является UTF-8, а в Windows используется своя кодировка 1251. Английские литеры по кодировке совпадают и там, и там, но русские выглядят совершенно по-разному, поэтому пришлось немного повозиться и создать отдельный вариант РуСи для Windows, отличающийся только кодировкой русских букв в сообщениях об ошибках и в разборе входных литер.

Задачей лексического анализатора — сканера является сборка лексем из отдельных литер. Лексема — это минимальная единица, которая поступает на вход синтаксическому анализатору. Некоторые литеры сами по себе являются лексемами, например круглые, фигурные и квадратные скобки. В других случаях лексема можем состоять из двух

или более литер, например знаки сравнения «==» или «<=». Ключевые слова также являются независимыми лексемами, они могут состоять из многих литер. Еще одним важным примером нетривиальных лексем являются числа. «13» — это одна лексема, состоящая из двух литер. А может быть, к примеру, число «3.14E-5» — это всё одна лексема. И, наконец, обычные идентификаторы. Скажем, «abc» — это три литеры, но одна лексема. В принципе, для построения лексических анализаторов давно уже разработаны всякие автоматические средства, но мне показалось, что это все равно, что стрелять из пушки по воробьям.

Еще много лет назад при реализации транслятора с Алгола 68 мы придумали технику заглядывания вперед на одну литеру. Есть переменные `curchar` (от слова current) и `nextchar`. Еще раз повторю — сами эти идентификаторы появились в нашем коллективе еще лет 40 назад и с тех пор не меняются. В переменной `curchar` находится литера, которая сейчас обрабатывается сканером, а в `nextchar` — литера, которая прочитана из файла, но еще не обработана. Сканер представляет из себя большой переключатель по коду литеры: если литера такова, что она не входит ни в какие комбинации из многих литер, то по литере просто выдается код лексемы (в трансляторе все лексемы — это константы, определенные через `#define`), если же литера такова, что за ней может быть продолжение (а его может и не быть), то в этом случае надо смотреть на значение переменной `nextchar`. Если там есть, например, еще один знак «=» (в случае, когда лексема `curchar` имеет значение «=»), то мы понимаем, что должны получить лексему сравнения на равно, пара `curchar nextchar` сдвигается вправо на одну литеру: `nextchar` копируется в `curchar`, а в `nextchar` считывается еще одна литера из исходного файла.

Такой механизм оправдал себя, все работает линейно, быстро и без возвратов.

Рассказывая о сканере, можно показать студентам несколько интересных техник, которые широко применяются в трансляторах. Прежде всего, это хэш-таблица и работа с таблицей представлений. Понятно, что одной из главных задач сканера является единственное представление лексемы: если в тексте программы десять раз встретится идентификатор `abc`, транслятор должен понимать, что это один и тот же идентификатор, а не 10 разных. Для этого строится таблица `reprtab` (Representation Table), куда заносятся идентификаторы. Таблица строится в стековом режиме, есть указатель на последнюю занятую строчку `rp` таблицы. Скажем, в тексте появился идентификатор, он по одной литере записывается в конец таблицы `reprtab`, а потом надо посмотреть, не было ли такого же идентификатора ранее. Если не было, идентификатор остается в таблице, если же уже встречался, последний идентификатор из таблицы выкидывается (просто указатель `rp` возвращается на ту позицию, где он был перед чтением последнего идентификатора). В любом случае сканер выдает лексему `IDENT`, а в глобальной переменной `repr` ссылку на `reprtab`.

Сравнение литер — это вообще одна из самых медленных операций. Современные ЭВМ рассчитаны на работу с числами — работа с литерами, особенно если нет специальной аппаратной поддержки, довольно медленная, поэтому хорошо было бы ее сэкономить. Тут и применяется идея хэш-таблиц. Сначала по идентификатору строится функция свертки (какая-нибудь однозначная в одну сторону функция). Мы традиционно применяем просто сумму всех кодов литер по модулю 256, то есть складываем все литеры, а потом берем младший байт суммы и называем результат значением хэш-функции. Есть таблица `hashtab` с индексами от 0 до 255, все элементы которой сначала имеют значение 0. Если по идентификатору получилось значение хэш-функции `i`, обращаемся в таблицу `hashtab` с индексом `i`. Если там ноль, значит, идентификатора с такой хэш-функцией

еще не было. Мы оставляем его в таблице `reprtab`, а в `hashtab` записываем ссылку на его позицию в `reprtab`. Если же в `hashtab` не ноль, то это начало цепного списка в `reprtab` всех идентификаторов, у которых одинаковая хэш-функция. На рис. 1 представлена ситуация, когда идентификатор `bac` только что занесен на верхушку таблицы, но поиск по хэш-списку еще не начат.

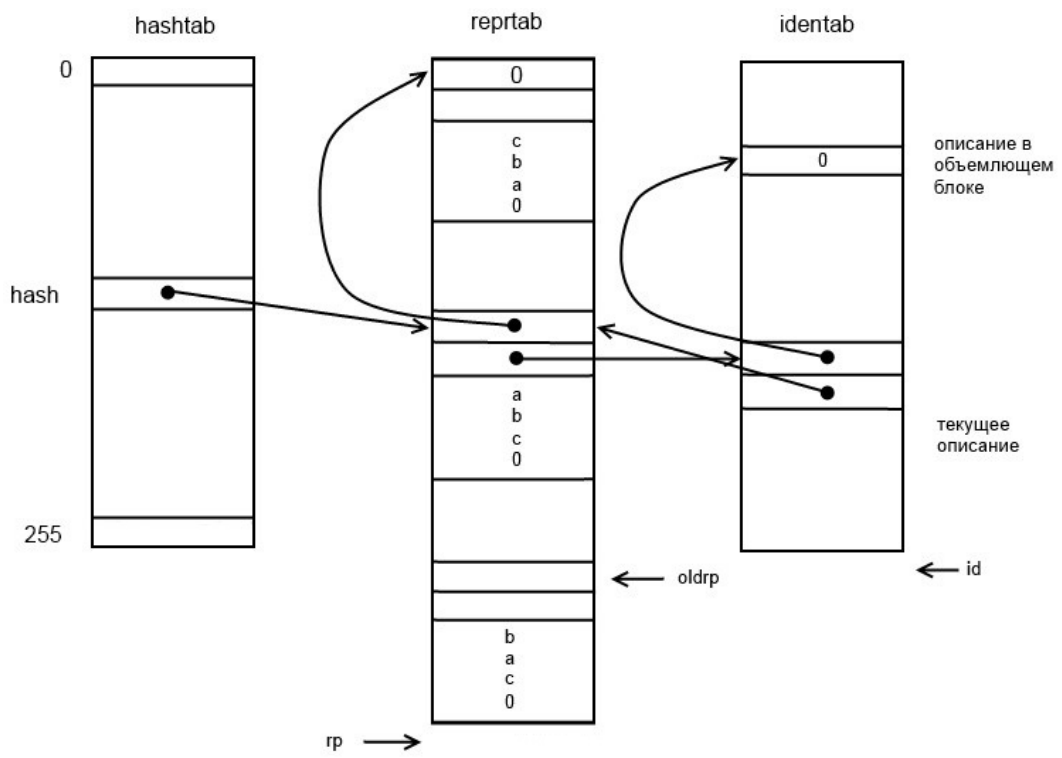

Рис. 1

Сканер бежит по этому цепному списку и там уже сравнивает все политерно. В чем здесь выигрыш? На самом деле, хэш-функции асимптотически никакого выигрыша не дают, но на практике выигрыш есть, причем очень большой. Например, вряд ли можно представить себе программу, в которой больше тысячи идентификаторов. Пусть мы работаем именно с такой программой с тысячью идентификаторов. Даже если мы будем как-то упорядочивать таблицу reprtab (что стоит дорого — с каждым новым идентификатором таблицу придется переупорядочивать), то теория говорит, что мы можем найти такой же идентификатор или показать, что его нет, не менее чем за 10 сравнений (двоичный логарифм от 1000). В случае хэш-таблицы, если хэш-функция подобрана хорошо и распределяет все более-менее равномерно (а упомянутая выше наша традиционная хэш-функция именно такова), то с самого начала вся таблица reprtab разделяется на 256 подсписков (в среднем получается, что в каждом из этих подсписков будет не более 4 вхождений, то есть поиск будет производиться за 4 сравнения).

В этом месте я обязательно останавливаюсь и рассказываю студентам всякие истории, как развивалась информатика. Еще когда я сам был студентом, меня учили, что есть существенная разница между математикой и информатикой. В математике самое главное — первое решение. На мат-мехе ходит такая легенда, что когда Юрий Матиясевич в 1970 году решил десятую проблему Гильберта, а через несколько месяцев другой ленин-

градский математик, зная решение Матиясевича, представил доказательство, которое было короче и понятнее, чем первое, были какие-то поползновения считать решение десятой проблемы — решением Матиясевича и того другого математика. Но математики всего Советского Союза возмутились, и этот фокус не прошел — в математике важно только первое решение.

В информатике и программировании чаще всего первое решение вообще никому не нужно. Скажем, алгоритм игры в шахматы. Легко написать программу, как ходит пешка, слон, конь и так далее, а потом написать такой алгоритм: перебрать все ходы белых, в ответ на них перебрать все ходы черных — скажем, на глубину 8 (я где-то читал, что гроссмейстеры могут продумывать позиции на 8 ходов вперед). Вроде бы все — вы можете выиграть у любого гроссмейстера, но число вариантов в этом алгоритме будет настолько большое, что этот алгоритм никогда работать не будет. К сожалению, это очень частая практическая ситуация.

Я не против теории, я сам профессор математики, доктор физ-мат наук, но все-таки мое практическое «я» часто побеждает мое теоретическое «я». И хэш таблицы — отличный тому пример. Да, они не дают асимптотического выигрыша, но в практически полезных случаях мы получаем реальное ускорение. Как я уже сказал, на программе из 1000 идентификаторов ускорение поиска в таблице `reprtab` — более, чем в 2 раза. Я таких примеров знаю огромное количество и стараюсь их по мере сил к месту, а иногда и не к месту, рассказывать студентам, чтобы привить им понимание, что «суха теория, мой друг, но древо жизни вечно зеленеет» (Гёте «Фауст»).

Расскажем теперь про работу с числами. С ними все просто — встречаем цифру, начинаем по схеме Горнера вычислять все число, выбирая из исходного файла по одной цифре. Для целых это будут только цифры, для вещественных — пока не встретится точка и/или E, а потом плюс/минус порядок. Все это никаких сложностей не несет. Забавно, что когда я дал свой транслятор на пробу своим студентам, в первом же тесте кто-то написал число, превосходящее 2^{31}, а у меня транслятор это не поймал. Понятно, что такие вещи надо ловить и ругаться, если пользователь написал слишком длинное число, не представимое в 32-разрядной сетке, принятой в РуСи, но мне как-то и в голову не пришло, что студенты будут играться с такими числами.

3.2. Синтаксический (видонезависимый) анализ

Транслятор вначале обращается к сканеру, а потом лексема, которую выдал сканер, тут же попадает в синтаксический анализатор. Кстати, в синтаксическом разборе применяется ровно та же техника, как в сканере — заглядывание вперед. Но сейчас мы заглядываем уже не на одну литеру, а на одну лексему: `cur` — текущая лексема, а `next` — следующая лексема. Оказалось, что этого вполне хватает, чтобы разобрать такой богатый язык, как C.

Для синтаксических анализаторов разработано огромное количество автоматических программ построения анализаторов — самая популярная из них YACC. Я много лет назад по просьбе академика Ершова прочитал статью Стивена Джонсона «YACC: Yet Another Compiler Compiler» [4] и доложил ее содержание на заседании Рабочей группы по Алголу 68. Тогда это было новинкой, все это было интересно. Сейчас это уже commodity — общеизвестный факт, но я не люблю пользоваться этими средствами, хотя у нас на кафедре системного программирования есть люди, которые занимаются разработкой новых синтаксических анализаторов для новых классов языков (например динамически генерируемых текстов, таких как встроенный SQL).

В этих автоматических анализаторах есть один, но очень существенный недостаток — очень слабая сигнализация об ошибках. Например, задана грамматика языка, по этой грамматике автоматически построен анализатор с огромными таблицами. На вход поступает программа, работает магазинный автомат, выбирает из исходного текста одну за другой лексемы. Представим, что получили лексему, которая не подходит ни к какому грамматическому правилу. Всё, что может автоматический анализатор, — это написать: «Символ не подходит». К сожалению, это самая частая ситуация во многих существующих трансляторах. Нет бы сказать, в какой конструкции встретилась ошибка, что ожидалось, что именно не подходит, но из автоматически сгенерированных таблиц эту информацию выудить трудно.

Поэтому испокон века в нашем коллективе принята реализация методом рекурсивного спуска, у которого есть следующие преимущества:

- Он очень прост в реализации. Увидел лексему ЕСЛИ — запустил процедуру анализа условного оператора, в которой последовательно надо проверить наличие выражения в скобках, оператора, а потом заглянуть на одну лексему вперед, и если там окажется ИНАЧЕ, то проверить наличие еще одного оператора.Если внутри снова встретится лексема ЕСЛИ — ничего страшного, вновь рекурсивно вызовется процедура разбора условного оператора.
- Рекурсивный спуск дает хорошую возможность для сигнализации об ошибках, потому что всегда можно сказать: «В конструкции такой-то встретилось то-то» или, наоборот, «В конструкции такой-то не встретилось то, что нужно в этом операторе». Поэтому всегда можно сказать в какой конструкции что именно произошло, что не понравилось транслятору, а если еще аккуратно распечатать фрагмент исходной программы до места ошибки, то можно дать пользователю хорошее понимание того, что именно не нравится транслятору.

В РуСи свыше ста типов ошибок, причем каждая ошибка сигнализируется длинной фразой на русском языке с вкраплениями идентификаторов или чего-то еще. Скажем, неописанный идентификатор обязательно будет сопровождаться текстом, какой именно идентификатор, в какой строке. Если будет несоответствие типов, то тоже будет указано, в какой строке и что не подходит. Сто разных типов ошибок — это не самоцель, это стремление как можно аккуратнее представить ошибку пользователю, чтобы ему было легче разобраться, в чем же он ошибся.

В РуСи применен такой технический прием: конструкции-операторы разбираются рекурсивным спуском, прямо начиная от внешних описаний, через функции и дальше через операторы, но если мы перебрали все лексемы, с которых могут начинаться операторы, то с текущей лексемы может начинаться только выражение (в том числе и присваивание), и транслятор переходит в другой механизм — к анализу снизу-вверх.

В грамматике С те конструкции, которые могут служить операндами выражений, называются унарными выражениями (`unarexpr`). Унарное выражение — это возможная унарная операция (, !, +, -, &, *, ++, - -), за которой следует `postexpr`, представляющее из себя первичное (`primary`, то есть число, строка, идентификатор или выражение в круглых скобках), за которым, возможно, следует левая квадратная скобка (вырезка элемента массива), левая круглая скобка (вызов функции) или постфиксные варианты операций ++ и - -. Транслятор вычитывает из текста унарное выражение (соответственно, обрабатывает все унарные операции) и заглядывает вперед на одну лексему. Если там знак операции (теперь уже только бинарной), значит, продолжается работа по вычитыванию

выражения. Если нет — значит, мы закончили разбор выражения и можем возвращаться в рекурсивный спуск.

Синтаксический разбор выражений в РуСи совмещен с переводом в обратную польскую запись. В результирующем дереве синтаксического разбора выражение представляется в виде последовательности идентификаторов, констант, вырезок, вызовов и операций над ними, например, выражение

```
a*b - c/2 + 3.14
```

будет представлено в дереве последовательностью

```
a, b, *, c, 2, /, -, 3.14, +
```

Для этапа генерации кода удобно, чтобы выражение заканчивалось специальным признаком конца выражения `TExprend`, этот признак ставится в процедуре `exprassn`, занимающейся разбором выражений с присваиваниями, которые имеют наименьший приоритет. Поскольку такие операции бывают вложенными, например,

```
a = b += c -= d,
```

процедура `exprassn` имеет параметр `level`, самая левая операция присваивания получает параметр `level`, равный 0, а все остальные — свой уровень вложенности, который всегда больше 0, именно, чтобы в правильном месте поставить `TExprend`.

Для перевода в обратную польскую запись используются два стека: `stack`, куда пишется приоритет операции, и `stackop`, куда пишется сама операция.

Алгоритм перевода очень прост: встречается операнд — подаем его на выход (в дерево), встретилась операция — выталкиваем на выход все операции из стека, приоритет которых больше или равен приоритету новой операции, а новую операцию кладем в стек.

3.3. Видозависимый анализ

Одновременно с построением дерева разбора выполняется видозависимый анализ. Для каждого выражения «вычисляется» его тип, а во всех местах, где требуется значение определенного типа (в индексах массивов — целый, в присваиваниях — такой же, как у левой части присваивания, в вызовах функций — тип фактического параметра должен совпадать с видом формального параметра и т. д.), выполняются нужные проверки. Это «вычисление» типов осуществляется с помощью стека `stackopnd`. Листьями дерева разбора являются идентификаторы и константы. Константы имеют тип, который определяется прямо по их записи, а идентификаторы получают тип в процессе идентификации, то есть поиска описания по применению. Каждый узел дерева, получив типы своих операндов из стека `stackopnd`, определяет тип своего результата и кладет его в тот же стек.

Традиционно для языков с блочной структурой принято строить таблицу идентификаторов `identab`, отражающую блочную структуру программы. Сначала описание идентификатора ищется в минимальном объемлющем блоке, потом в более внешних, но без параллельных подблоков. Это довольно медленный процесс, связанный с большим перебором.

Много лет назад мы придумали более оптимальный алгоритм. Как мы уже видели, по идентификатору можно довольно быстро с помощью хэш-списков найти его единственное вхождение в `reprtab`. В этой таблице мы поддерживаем ссылку на текущее описание этого идентификатора в `identab`. Когда встречается описание идентификатора, мы заносим информацию об этом описании в `identab`, а в `reprtab` помещаем ссылку

на это новое описание, при этом в identab хранится ссылка на описание такого же идентификатора во внешнем блоке (если оно было) — см рис. 1. Таким образом, по каждому применению идентификатора мы без всякого поиска получаем ссылку на его текущее описание, единственный накладной расход — это восстановить старую ссылку в момент выхода из блока. Мы исходим из того, что описаний идентификаторов мало, а их применений много, поэтому наше решение существенно эффективнее.

Простые типы, такие как ЦЕЛ, ЛИТЕРА, ВЕЩ, массивы из них и указатели на них, кодируются отрицательными целыми числами. Для функций и структур используется специальная таблица modetab, ссылки на которую положительные. В этой таблице тип функции описывается следующей последовательностью: тип значения функции (отрицательное число), N — количество параметров (может быть нулем), N типов формальных параметров, причем если тип параметра положителен, то это означает, что параметром является функция, а тип — это ссылка в modetab. Для структур: N — число полей структуры (положительное), затем N пар — тип поля, ссылка в reprtab на идентификатор поля. Заметим, что по первому элементу в modetab можно отличить функцию от структуры.

Только функция может быть дважды описана в блоке — предописание и описание. В предописании указывается тип результата, идентификатор функции и типы параметров — идентификаторы параметров не указываются. Чтобы отличить предописание от описания в identab, второе поле (ссылка на reprtab) у предописания отрицательное. Обычно третье поле identab (тип) отрицательное (ЦЕЛ, ЛИТЕРА, ВЕЩ, массивы, указатели), а у функций и структур — положительное, там ссылка на modetab.

Четвертое поле identab — номер функции (просто очередной номер для каждого предописания и описания, номера предописаний нигде не используются, но так оказалось проще в реализации).

Целый пласт проблем создают функции, переданные параметром. Собственно, значением, представляющим функцию, является её номер. В более сложных языках, чем C, нужен еще адрес базы — адрес статики, где описаны глобальные переменные, используемые в данной функции, но в C все функции описаны на одном уровне, что, естественно, ограничивает язык, но упрощает реализацию. Под функцию-параметр отводится 1 ячейка в памяти вызывающей функции наряду с ячейками для других параметров (на верхушке стека). Таким образом, функция-параметр представляется смещением от *l*. В момент вызова в соответствующем месте генерируется команда LI N функции (загрузка в стек непосредственного операнда, в данном случае — номера функции), если фактическим параметром является функция, описанная в программе, или LOAD смещение, если функция-параметр сама когда-то была передана параметром, а теперь передается параметром дальше.

При формировании значения идентификатора в процедуре identtoval сначала проверяется знак поля тип (третье поле в identab). Если там отрицательное значение, то идентификатор не является функцией и обрабатывается обычным порядком, если же там положительное значение, то это ссылка на modetab, а идентификатор является функцией или структурой. Если четвертое поле identab положительное, то это номер функции, а если отрицательное, то это минус смещение от *l*.

4. ГЕНЕРАЦИЯ КОДА

На вход генератору кода поступает синтаксически правильное дерево разбора (если в процессе синтаксического разбора были найдены ошибки, генерация кода не произ-

водится). Генерация кода без глобальной оптимизации — довольно простая задача, все конструкции явно названы, выражения записаны в обратной польской записи, операции даны в своих окончательных кодах (целые, вещественные, с сохранением результата на стеке или без, операнды являются простыми переменными или вычисляемыми на стеке значениями). Как и в синтаксическом анализе, начинаем с генерации функций и глобальных переменных, внутри функций генерируем код операторов, а когда в дереве встречается что-то отличное от оператора, генерируем код выражения.

Стоит упомянуть, разве что генерацию циклов, например команды для цикла

```
ДЛЯ (i = 0; i < n; i++)
    оператор;
```

лучше бы генерировать в таком порядке:

```
 i = 0; i < n; переход по невыполнению условия на выход из цикла; оператор;
i++; переход на проверку условия;
```

то есть `i++` (а это может быть сколь угодно сложное выражение) лучше сгенерировать не там, где это выражение написано. За счет того, что входом является именно дерево (и в узлах всех сложных конструкций проставлены адреса в дереве всех прямых потомков), а не последовательный файл, это можно сделать: когда подойдем к генерации инкрементальной части, запомним этот адрес и перепрыгнем на генерацию оператора, а потом вернемся к запомненному адресу инкремента.

5. ЗАКЛЮЧЕНИЕ

После серии практических занятий по технике трансляции мне удалось привлечь к работе по развитию проекта РуСи нескольких студентов мат-меха по темам:

- реализация автоматического регрессионного анализа;
- реализация структур и операций над ними;
- `run time` окружение для работы с параллельными нитями.

Один довольно хорошо программирующий ученик 11 класса Лицея 239 реализовал удобную `IDE` (Integrated Development Environment), позволяющую редактировать, компилировать и запускать на счет примеры на языке С. Таким образом, можно утверждать, что проект РуСи «пошел в жизнь».

Список литературы

1. *Брайан Керниган, Деннис Ритчи.* Язык программирования С. М.: Вильямс, 2006.
2. *Terekhov A., Luchin R., Filippov S.* Educational Cybernetical Construction Set for Schools and Universities, Advances in Control Education. Vol. 9, Part 1.
3. http://blog.trikset.com/ (дата обращения: 27.02.2016).
4. Yacc: Yet Another Compiler-Compiler, Stephen C. Johnson, http://dinosaur.compilertools.net/yacc/ (дата обращения: 27.02.2016).

PROGRAMMING AND COMPILER TECHNIQUES EDUCATIONAL TOOL

Terekhov A. N.

Abstract

The project RuC was designed as a tool for teaching programming among pupils, students and adults who have decided to learn this wonderful profession. Initially there was the need to create a simple, intuitive, yet powerful tool for robots programming, then the task was extended to the training of algorithmic and computer science. Finally, it was found that the resulting compact compiler with C language (with some restrictions) to the codes of the original virtual machine can be successfully used in the course «Translation» for software engineering specialty. The architecture of the RuC virtual machine project will be discussed in a separate article.

Keywords: *C programming language, compiler, virtual machine, programming education, robotics, school informatics.*

Терехов Андрей Николаевич,
доктор физико-математических наук,
профессор, заведующий кафедрой
системного программирования СПбГУ,
A.Terekhov@spbu.ru

© Наши авторы, 2016.
Our authors, 2016.