Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

Кафедра математической
кибернетики и компьютерных наук

ГЕНЕРАТОР РАЗБОРА LR(1)-ГРАММАТИК

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 451 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Слуцкого Алексея Дмитриевича

Научный руководитель
доцент, к. ф.-м. н. _____ Г. Г. Наркайтис

Заведующий кафедрой
к. ф.-м. н. _____ С. В. Миронов

Саратов 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Лексический и синтаксический анализаторы являются важными этапами работы любого компилятора. На этапе лексического анализа входной поток символов превращается в структуры, необходимые для дальнейшей обработки синтаксическим анализатором.

Данный вид анализа используется для решения большого количества задач. От компиляторов, которые помогают программисту общаться с компьютером на языке высокого уровня до спеллчекеров, которые в наше время повсеместно — в браузерах, мобильных приложениях, текстовых редакторах.

На практике большую ценность имеет задача построения синтаксических анализаторов для, так называемых, контекстно-свободных открытых языков, то есть языков, задаваемых контекстно-свободной грамматикой, которая может быть расширена.

В ходе данной квалификационной работы описывается реализованное приложение, которое может генерировать лексический и синтаксический анализаторы в виде программ на языке C на основе задаваемых правил распознавания лексем и грамматики.

1 Основное содержание работы

1.1 Постановка проблемы

В настоящее время в любой операционной системе хоть на компьютере, хоть на телефоне есть приложение калькулятор. Как правило оно имеет максимально простой интерфейс. Пользователю предлагается ввести в форму некоторое арифметическое выражение, а калькулятор отобразит его значение.

Но каким образом это работает? Ведь выражение может быть очень сложным, может содержать одновременно левоассоциативные и правоассоциативные операции, алгебраические операции с разным приоритетом и символы меняющие порядок приоритета, например, круглые скобки. Такое выражение нельзя посчитать просто пройдя слева направо, накапливая значение в одной переменной. И конечно, существуют разные алгоритмы подсчета.

В данной работе рассматривается такой способ как восходящий синтаксический анализ. И в качестве модельной задачи был выбран калькулятор.

Задача построения генераторов анализаторов может быть решена методом декомпозиции. Разберем сначала существующие инструменты. Затем посмотрим, что из себя представляет анализатор как программа на языке C. И в итоге построим свой генератор. В основе построения как анализаторов, так и целых компиляторов лежит глубокий теоретический аппарат. Чтобы научиться строить свои анализаторы и, уж тем более, генераторы анализаторов, сначала следует ознакомиться с основными понятиями, которыми придется оперировать в дальнейшем.

Формальная грамматика в теории формальных языков — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита. Различают порождающие и распознающие (или аналитические) грамматики — первые задают правила, с помощью которых можно построить любое слово языка, а вторые позволяют по данному слову определить, входит оно в язык или нет.

Формальные грамматики лежат в основе синтаксического анализа. Но синтаксический анализ не может существовать без лексического, ведь сначала входную последовательность нужно преобразовать в пригодный для анализа формат.

Лексический анализ в информатике — это процесс разбиения входных данных на последовательности символов, которые именуются «токенами».

Группа символов, распознанная как токен, называется лексемой. Каждый токен однозначно характеризует лексему. А лексема, в свою очередь, однозначно определяет некоторую конструкцию языка. При лексическом анализе распознаются и выделяются лексемы из входной последовательности.

Лексический анализ проводится на основе заданного алфавита или набора языков, и грамматика языка задает список лексем, которые допустимо встретить во входных данных. Традиционно принято организовывать входную последовательность как поток. При таком подходе процесс сам управляет выборкой символов из входного потока.

Синтаксический анализ — это сопоставление последовательного набора лексем формального языка с его формальной грамматикой. В процессе анализа входные данные представляются в виде дерева, так как оно хорошо подходит для дальнейшей обработки. Это может быть дерево составляющих, дерево зависимостей, другие структуры или сочетания нескольких способов представления.

Во время синтаксического анализа производится проверка корректности данных, представленных в виде последовательности токенов и совокупности таблиц, и преобразование программы во внутреннюю форму, удобную для последующей обработки. Обычно применяется совместно с лексическим анализом

Этап синтаксического анализа является ключевым, так как он непосредственно взаимодействует с лексической фазой и обеспечивает создание основы для работы компилятора. [1]

1.2 Реализация с использованием утилит

`Flex` — это инструмент для лексического анализа текста, который генерирует лексические анализаторы. На базе пакетов `GNU` заменяет `UNIX`-генератор `Lex`, при этом имеет аналогичную функциональность и расшифровывается как «Fast LEX».

`Flex` генерирует программы, распознающие шаблоны в тексте. Он создает программу, которая, используя заданные правила поиска, использует их в шаблонах. Преимущество `Flex` в простоте задания правил по сравнению с написанием собственных инструментов для поиска по шаблону.

Генератор получает на вход цепочки символов и правила выделения лексем, которые необходимо выполнить в случае успеха распознавания. После

выполнения в качестве выходных данных выдает код анализатора в виде функции на языке C.

Полученную функцию часто используют совместно с генераторами синтаксических анализаторов. Они используют для поиска следующей лексемы функцию `yylex`, которую сгенерировал `Flex`. Обычно `Flex` используют с `YACC` или `GNU Bison`.

`Flex` работает по следующему принципу. Сначала подготавливается спецификация лексического анализатора на языке `Lex` — получается файл `lex.l`. Потом полученный файл компилируется `Lex`-ом для получения программы на языке C. Результат — `lex.yy.c`. Эта программа содержит диаграмму переходов в виде таблицы, построенной по регулярным выражениям файла `lex.l` и программу, использующую эту таблицу для распознавания лексем.

`Bison` — это инструмент для грамматического разбора текста, который генерирует синтаксические анализаторы на основе грамматики, описанной в нотации `BNF` (форма Бэкуса-Наура) или контекстно-свободной грамматики.

`GNU Bison` может использоваться для преобразования входной последовательности токенов в структурированный формат для дальнейшей обработки. На выходе выдает программу-парсер на языке C.

`Bison` работает только с грамматиками класса LALR(1), то есть необходимо, чтобы была возможность разобрать любую последовательность, заглядывая вперед не более, чем на одну лексему.

Чтобы `Bison` смог разобрать программу на некотором языке, нужно чтобы данный язык был описан контекстно-свободной грамматикой. Таким образом, любая синтаксическая группа должна составляться из составных частей. При этом допустимо рекурсивное определение, однако в таком случае необходимо присутствие хотя бы одного правила, выводящего из рекурсии.

Код анализатора — это код, определяющий функцию `yyparse`, которая реализует грамматику. Также для корректной работы необходимы дополнительные функции. Одна из них — лексический анализатор.

Главная задача `Bison` — это объединение лексем в группы в с соответствии с правилами грамматики. Часто `Bison` используют совместно с генератором лексических анализаторов `Flex`. И `Bison` вызывает лексический анализатор, когда ему необходима очередная лексема.

Мой проект состоит из нескольких отдельных программ. При добавле-

нии новой функции калькулятору, внесении правок в исходный код, отладке программы приходится перекомпилировать каждый файл и собирать его в единое целое по несколько раз. Чтобы не делать это вручную, можно воспользоваться стандартной утилитой GNU make. Она самостоятельно принимает решение о том, какие файлы нужно перекомпилировать и выполняет необходимые действия.

Для того, чтобы использовать make, нужно создать make-файл, описывающий зависимости между частями проекта и содержащий команды для их обновления. Обычно исполняемый файл зависит от объектных файлов, которые, в свою очередь, зависят от файлов с исходным кодом.

После создания make-файла достаточно просто ввести команду «make» и все необходимые файлы будут перекомпилированны. Утилита make на основе данных о последних модификациях частей проекта обновляет устаревшие файлы. Для всех них выполняются указанные в makefile команды. [2]

В общем случае makefile выглядит как набор из правил следующего вида:

```
target:   dependencies
          command
```

Target представляет собой имя файла, которое генерируется в процессе сборки. Dependencies — файл, используемый для порождения цели. Command — действие, которое необходимо выполнить, если dependencies устарели.

Правило описывает, каким образом необходимо обновлять файлы или как должно выполняться некоторое действие. Файл должен быть перекомпилирован всякий раз, когда изменилась одна из его зависимостей. При этом все зависимости, которые генерируются автоматически, должный быть первыми обновлены.

По умолчанию, make начинает работать с первой встреченной целью. Эта цель выбирается целью по умолчанию. Главная цель — это цель, которую стремится достичь make, как результат своей работы. Другие правила обрабатываются только потому, что их цели являются прямыми или косвенными зависимостями для главной цели. [3]

1.3 Реализация генератора

LR-анализаторы управляются таблицами. Грамматика, для которой можно построить таблицу восходящего синтаксического анализа, называется LR-грамматикой. Для того, чтобы грамматика была LR-грамматикой, достаточно, чтобы синтаксический анализатор, работающий слева направо методом переноса/свертки, был способен распознавать основы правосентенциальных форм при их появлении на вершине стека. LR-анализ полезен и удобен по множеству причин:

- LR-анализаторы могут быть созданы для распознавания практически всех конструкций языков программирования, для которых может быть написана контекстно-свободная грамматика. Контекстно-свободные грамматики, не являющиеся LR-грамматиками, существуют, однако для типичных конструкций языков программирования их вполне можно избежать.
- Метод LR-анализа — наиболее общий метод ПС-анализа без возврата, который, кроме того, не уступает в эффективности другим, более примитивным ПС-методам.
- LR-анализатор может обнаруживать синтаксические ошибки сразу же, как только это становится возможным при сканировании входного потока.
- Класс грамматик, которые могут быть проанализированы с использованием LR-методов, представляет собой надмножество класса грамматик, которые могут быть проанализированы с использованием предиктивных или LL-методов синтаксического анализа. В случае грамматик, принадлежащих классу LR(k), возникает необходимость распознать правую часть продукции в порожденной ею правосентенциальной форме с дополнительным предпросмотром k входных символов. Это требование существенно мягче требования для LL(k)-грамматик, в которых необходимо распознать продукцию по первым k символам порождения ее тела. Таким образом становится ясно, что LR-грамматики могут описать существенно больше языков, чем LL-грамматики.

Основная идея, лежащая в основе канонического LR-анализа заключается в построении LR(0)-автомата для заданной грамматики. Состояниями этого автомата являются множества пунктов из канонического набора LR(0), а пе-

реходы определяются функцией GOTO.

Очередной шаг синтаксического анализатора из приведенной выше конфигурации определяется считанным текущим входным символом a_i и состоянием на вершине стека s_m путем обращения к записи ACTION[s_m, a_i] в таблице действий синтаксического анализа. В результате выполнения указанного в записи действия (одного из четырех возможных типов) получаются следующие конфигурации:

1. Если ACTION[s_m, a_i] = перенос s, синтаксический анализатор выполняет перенос в стек очередного состояния s и его конфигурацией становится

$$(s_0 \ldots s_m s, a_{i+1} \ldots a_n \$)$$

   Символ a_i хранить в стеке не нужно, поскольку при необходимости он может быть восстановлен из s. Текущим входным символом становится a_{i+1}.
2. Если ACTION[s_m, a_i] = свертка $A \to \beta$, синтаксический анализатор выполняет свертку и его конфигурацией становится

$$(s_0 \ldots s_{m-r} s, a_i a_{i+1} \ldots a_n \$)$$

   Здесь r — длина β, а s = GOTO[s_{m-r}, A]. Синтаксический анализатор вначале снимает r символов состояний с вершины стека, что переносит на вершину стека состояние s_{m-r}, после чего на вершину стека помещается s, запись из GOTO[s_{m-r}, A]. При свертке текущий входной символ не изменяется.
3. Если ACTION[s_m, a_i] = принятие, синтаксический анализ завершается.
4. Если ACTION[s_m, a_i] = ошибка, синтаксический анализатор обнаруживает ошибку и вызывает подпрограмму восстановления после ошибки. [4]

Для построения генератора активно используется функция CLOSURE, которая заключается в следующем:

Пока не закончились пункты для добавления в I, будем просматривать пункты $[A \to \alpha \cdot B\beta, a]$ из I. Для каждого такого пункта будем искать продукцию $B \to \gamma$ и добавлять $[B \to \cdot\gamma, b]$ в множество I для всех терминалов $b \in$ FIRST($\beta\alpha$).

Теперь приведем правила построения LR(1)-функций `ACTION` и `GOTO` из множеств LR(1)-пунктов. Эти функции, как и ранее, представлены таблицей. Единственное отличие — в значениях записей таблицы.

1. Построить $C' = \{I_0 ... I_n\}$ — набор множеств LR(1)-пунктов для G'.
2. Состояние синтаксического анализатора строится из I_i. Действие синтаксического анализа для состояния i определяется следующим образом.
   - Если $[A \to \alpha \cdot a\beta, b] \in I_i$ и `GOTO`(I_i, a) = I_j, то устанавливается `ACTION`[i, a] равным «перенос j». Здесь a должно быть терминалом.
   - Если $[A \to \alpha\cdot, a] \in I_i$ и $A \neq S'$, то устанавливается `ACTION`[i, a] равным «свертка $A \to \alpha$».
   - Если $[S' \to S\cdot, \$] \in I_i$, то устанавливается `ACTION`[i, \$] равным «принятие».

   Если при применении указанных правил обнаруживаются конфликтующие действия, делается вывод о том, что грамматика не принадлежит классу LR(1).
3. Переходы для состояния i строятся для всех нетерминалов A с использованием следующего правила: если `GOTO`(I_i, A) = I_j, то `GOTO`(i, A) = j.
4. Все записи, не определенные правилами (2) и (3), получают значение «ошибка».
5. Начальное состояние синтаксического анализатора строим из множества пунктов, содержащего $[S' \to \cdot S, \$]$.

Построение лексического анализатора заключается в обработке регулярных выражений и построении детерминированного конечного автомат (ДКА). Но перед тем как приступить к рассмотрению непосредственного перехода от регулярных выражений к ДКА, сначала следует исследовать построение НКА и рассмотреть роли, которые играют различные состояния автоматов. Будем называть состояние НКА важным, если оно имеет исходящий не-ϵ-переход. Заметим, что в процессе построения подмножеств используются только важные состояния в множестве U при вычислении `scope(go(U, a))`. Таким образом, множество состояний `go(s, a)` непустое, только если состояние s — важное. В процессе построения подмножеств два множества состояний НКА могут отождествляться, т.е. рассматриваться как единое множество, если они либо имеют одни и те же важные состояния, либо оба содержат принимающие

состояния, либо оба их не содержат.

При построении НКА из регулярных выражений единственными важными состояниями являются те, которые созданы базисом алгоритма в качестве начальных для конкретных символов в регулярных выражениях; т.е. каждое важное состояние соответствует некоторому операнду в регулярном выражении. Построенный НКА имеет только одно принимающее состояние, у которого нет исходящих переходов, и важным оно не является. Добавляя к регулярному выражению r справа уникальный ограничитель $\#$, мы даем принимающему состоянию для r переход по символу $\#$, делая это состояние важным в НКА для $(r)\#$. Другими словами, используя расширенное регулярное выражение $(r)\#$, можно забыть о принимающих состояниях при построении подмножеств; когда построение будет завершено, любое состояние с переходом по символу $\#$ должно быть принимающим.

Важные состояния НКА соответствуют позициям в регулярном выражении, в которых хранятся символы алфавита. Это удобно для представления регулярного выражения его синтаксическим деревом, в котором листья соответствуют операндам, а внутренние узлы — операторам. Внутренний узел называется cat-узлом, or-узлом или star-узлом, если он помечен соответственно оператором конкатенации $(.)$, объединения $(|)$ или звездочкой $(^*)$. Синтаксическое дерево для регулярного выражения может быть построено так же, как для арифметического. [5]

Для построения ДКА непосредственно из регулярного выражения мы строим его синтаксическое дерево, а затем вычисляем четыре функции `nullable`, `firstpos`, `lastpos` и `followpos`, определяемые следующим образом (каждое определение использует синтаксическое дерево для расширенного регулярного выражения $(r)\#$).

1. Значение `nullable(n)` для узла n синтаксического дерева равно `true` тогда и только тогда, когда подвыражение, представленное n, содержит в своем языке ϵ.
2. `firstpos(n)` представляет собой множество позиций в поддереве с корнем n, которые соответствуют первому символу как минимум одной строки в языке подвыражения с корнем n.
3. `lastpos(n)` представляет собой множество позиций в поддереве с корнем n, которые соответствуют последнему символу как минимум одной

строки в языке подвыражения с корнем n.

4. `followpos(p)` для позиции p представляет собой множество позиций q в синтаксическом дереве в целом, для которых существует строка $x = a_0...a_n$ языка $L((r)\#)$, обладающая тем свойством, что для некоторого i a_i соответствует позиции p, а a_{i+1} — позиции q.

ЗАКЛЮЧЕНИЕ

Задачей данной квалификационной работы было разобрать теоретическую основу использования анализаторов, на практическом примере отработать реализацию таких анализаторов на языке C и реализовать свой собственный генератор. В качестве модельной задачи был реализован ряд приложений, позволяющих осуществлять анализ арифметических выражений и получать правый разбор в соответствии с заданной грамматикой.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Ахо, А.* Компиляторы. Принципы, технологии, инструменты / А. Ахо. — М.: Вильямс, 2008.

2 Как устроен Makefile и что это такое [Электронный ресурс]. — URL: http://microsin.net/programming/avr/makefile.html (Дата обращения 15.04.2018). Загл. с экр. Яз. рус.

3 GNU Make. Программа управления компиляцией [Электронный ресурс]. — URL: http://linux.yaroslavl.ru/docs/prog/gnu_make_3-79_russian_manual.html (Дата обращения 15.04.2018). Загл. с экр. Яз. рус.

4 *Серебряков, В.* Лекции по конструированию компиляторов / В. Серебряков. — М.: Вильямс, 2003.

5 *Muchnick, S.* Advanced Compiler Design and Implementation / S. Muchnick. — CA.: Morgan Kaufmann, 1997.