Руководство по выполнению   
цикла лабораторных работ   
по курсу   
Проектирование   
трансляторов и интерпретаторов

# Введение

Предлагаемый цикл лабораторных работ нацелен на выработку у студентов умений и навыков в области проектирования предметно-ориентированных языков, а также разработки трансляторов для этих языков с использованием генератора компиляторов.

Coco/R (Coco/R *compiler compiler* generating *recursive descent* parsers - **ко**мпилятор **ко**мпилятора, генерирующего **р**екурсивные парсеры (анализаторы) ) - это генератор компилятора, который принимает атрибутную грамматику исходного языка и генерирует сканер и парсер для данного языка. Сканер работает как детерминированный конечный автомат. Парсер использует метод рекурсивного спуска. LL(1) конфликты могут быть разрешены просмотром нескольких символов вперед или смысловыми проверками. Таким образом, класс используемой грамматики является LL(*к*) для произвольного *к*.

Существуют версии Coco/R для Java, C#, C++, Delphi, Модула-2, Оберон и др. языков. Официальный сайт приложения:

http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/.

Эта программа является свободным программным обеспечением в соответствии с условиями GNU General Public License.

Для каждого языка в пакет Coco/R входит 3 файла:

Coco.exe – программа-генератор компилятора

Scanner.frame – заготовочный файл для лексического анализатора

Parser.frame - заготовочный файл для синтаксического анализатора.

# Входные данные для Coco/R

Входным для генератора компиляторов является файл, содержащий описание атрибутной грамматики

\*. ATG

# Требования к грамматике

Генератор формирует код по грамматике, заданной в виде правил РБНФ. Т.к. парсер стоится рекурсивным спуском, а это разбор сверху-вниз, то грамматика должна принадлежать LL(k). В идеале LL(1), но в Coco/R реализовано разрешение конфликтов.

LL(1) конфликты решаются за счет семантических действий и просмотра впереди стоящих лексем.

Грамматика принадлежит классу LL(1), если для двух различных продукций   
S->A|B выполняются условия:

1. Не существует такого терминала *а*, для которого А и В порождают строку, начинающуюся с а.

2. Пустую строку может порождать не более чем одна из продукций А или В.

# Язык Сoco/R

Грамматика записывается в файл.

Все правила должны заканчиваться точкой.

Продукционное правило вида S->AB будет записана как S=A B.

При записи правил могут использоваться метасимволы

| — или

( ) — группа

[ ] — необязательное однократное вхождение

{ } — 0 или более вхождений

## Подключаемые модули

### (для С++).

#include <string>

#include <sstream>

#include <iostream>

#include <fstream>

#include <vector>

### (для С#).

using System;

public class CLN

{

public static void Main(string[] arg)

{

Scanner scanner = new Scanner(arg[0]); // создать сканер

Parser parser = new Parser(scanner); // создать парсер

parser.Parse(); // запустить парсер

Console.WriteLine("Обнаружено ошибок: "+ parser.errors.count);

}

}

## Идентификатор компилятора

Представляет собой ключевое слово COMPILER, а за ним нетерминал, с которого начинается ваша грамматика.

COMPILER expr

## Глобальные переменные и функции.

После ключевого слова COMPILER можно вставить секцию с переменными и функциями. Если функций много, то лучше их вынести в отдельный файл, т.к. грамматика и без них получится очень страшной и нечитабельной.

int toInt(const std::wstring& strbuf)

{

std::wstringstream converter;

int value = 0;

converter << strbuf;

converter >> value;

return value;

}

std::wstring toString ( int Number )

{

std::wostringstream ss;

ss << Number;

return ss.str();

}

# Пример работы с генератором Coco/R

Описать синтаксис простого языка на основе языка CLN. С помощью COCO/R получить программы лексического и синтаксического анализов на языке C# (или другом). Добавив программу с основным классом CLN, получить исполнимый файл синтаксического анализатора. Проверить работу анализатора на нескольких примерах.

**Порядок работы**

1. В новый рабочий каталог размещаем файлы Coco.exe, Scanner.frame и Parser.frame.
2. Создаем файл CLN.ATG с описанием лексики и грамматики языка. Например:

COMPILER CLN

CHARACTERS

letter = 'A'..'Z' + 'a'..'z'.

digit = "0123456789".

cr = '\r'. lf = '\n'. tab = '\t'.

TOKENS

Ident = letter {letter | digit}.

Number = digit {digit}.

COMMENTS FROM "//" TO lf

IGNORE cr + lf + tab

PRODUCTIONS

CLN = { StatL }.

Op = '+' | '-' | '\*'| '/'.

RelOp = '=' | '<' | '>'.

StatL = [Number ':'] Stat.

Stat = Assign | Goto.

Assign = Ident '=' RValue [Op RValue].

RValue = Ident | Number.

Goto = ["if" RValue RelOp RValue] "goto" Number.

END CLN.

1. Запускаем в командной строке Coco.exe CLN.ATG. В случае отсутствия ошибок в каталоге создадутся файлы Scanner.cs и Parser.cs. При нормальном завершении выдаются сообщения:

Coco/R (Sep 19, 2006)

checking

CLN deletable

parser + scanner generated

0 errors detected

Ошибки выглядят так:

Coco/R (Sep 19, 2006)

checking

CLN deletable

No production for Stat1

1 errors detected

В данном случае в грамматике не найдено правило для нетерминала Stat1.

При запуске COCO.exe можно использовать различные режимы работы, которые задаются так:

COCO.exe CLN.atg -trace ASF.

В последовательности символов буквы означают следующее:

A - print the states of the scanner automaton

F - print the first sets and follow sets of all nonterminals

G - print the syntax graph of all productions

I - trace the computation of first sets

J - list the ANY and SYNC sets used in error recovery

P - print statistics about the run of Coco/R

S - print the symbol table and the list of declared literals

X - print a cross reference list of all terminals and nonterminals

1. Создаем файл CLN.cs с главной программой:

using System;

public class CLN

{

public static void Main(string[] arg)

{

Scanner scanner = new Scanner(arg[0]); // создать сканер

Parser parser = new Parser(scanner); // создать парсер

parser.Parse(); // запустить парсер

Console.WriteLine("Обнаружено ошибок: "+ parser.errors.count);

}

}

1. Оттранслируем главную программу, сканер и парсер. В результате получится исполнимый файл синтаксического анализатора CLN.exe!
2. Напишем тестовую программу на языке CLN test.cl. Например:

s = 0 // сумма квадратов от 1 до N

N = 101

k = 1

1: n = k \* k

s = s + k

k = k + 1

if k < N goto 1

n = 0

1. Выполним ее: CLN.exe test.cl При отсутствии ошибок выведется сообщение "Обнаружено ошибок: 0".

# Лабораторная работа № 1. Разработка синтаксиса предметно-ориентированного языка. Генерация лексического и синтаксического анализаторов с использованием Coco/R.

## Правила для генерации сканера.

### IGNORECASE

ключевое слово для игнорирования регистра символов.

### Символы языка

Секция CHARACTERS содержит описание множества допустимых символов. Например, что такое для нас буква и что такое цифра.

CHARACTERS

letter = «ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz».

digit = «0123456789».

Так же здесь можно описать символы-разделители.

cr = '\r'.

lf = '\n'.

tab = '\t'.

Запись вида char 1.. char2 обозначает последовательность символов от char1 до char2.

digit = '0'..'9'.

ANY — любая последовательность символов.

+ включая множество

— не включая

Например, можно описать множество символов, в которое не входят двойные кавычки:

verbatimStringChar = ANY — '"'.

Множество символов для шестнадцатеричной системы счисления:

hexDigit = digit + «ABCDEFabcdef».

### Лексемы.

Ключевое слово раздела — TOKENS. Здесь описываются правила для построения лексем «идентификатор» и «число».

TOKENS

ident = letter {letter | digit | "\_"}.

number = digit {digit}.

Лексема «строка», как любая последовательность символов, заключенная в двойные кавычки:

string = "\"" {verbatimStringChar} "\"".

Число из шестнадцатеричной системы счисления:

hex = «0x» {hexDigit hexDigit}.

### Секция специальных опций.

### Комментарии.

COMMENTS FROM "/\*" TO "\*/" NESTED

COMMENTS FROM "//" TO cr lf

### Разделители.

Здесь указываются разделители, которые мы игнорируем.

IGNORE cr + lf + tab

## Правила для генерации парсера

Начинаются с ключевого слова PRODUCTIONS.

Пример грамматики выражений:

Expr= ident ':=' NumExpr ';'

NumExpr = Term {('+'| '-' ) NumExpr}

Term = Multiplier {('\*'| '/' )Term}

Multiplier = ident | number | '(' NumExpr ')'

Все атрибуты пишутся в скобках < >. Если вы что-то используете из stl, например, list, то атрибуты должны быть помещены в скобки с точками, т.е. <. .>.

Атрибуты пишутся у нетерминалов и транслируются как параметры функций.

У терминала нет атрибутов, но если так сильно хочется, то его можно обернуть в нетерминал:

LogOp<std::wstring &op>=("&&"|"||") (.op=t->val;.).

Семантические действия пишутся в скобках с точками (. .). Этот тот код, который генератор вставит в парсер. Код пишется на том языке, на котором генерируются лексер и парсер.

ANY - это ключевое слово в секции парсера обозначает любой токен. Так что мы можем описать «текст» как {ANY}.

## Ключевое слово END,

за которым следует имя, которое было указано после COMPILER, завершает описание языка.

## Разрешение конфликтов.

Очень важно понимать, что грамматика должна соответствовать типу анализатора. Если это условие нарушено, то генератор начинает ругаться страшными словами.

### Факторизация.

Правила начинаются с одного и того же токена. Генератор так и пишет several alternatives

Пример:

S->a '=' B | a '('C')'

Как видите, 2 правила начинаются с токена «а», что нарушает первое правило LL(1). Переписываем его как S-> a ('='B |'(' C ') ')

Некоторые конфликты, такие как if-else решить невозможно.

Statement = if '(' Eepr ')' Statement [else Statement].

if (a>b) if(a>c) max=a; else max=b;

Не ясно, что тут выбрать, поэтому было принято соглашение: выбирать первую альтернативу. В данном примере выбор правильный, но в своей грамматике лучше избегать таких неоднозначностей.

Неудачная запись.

S= a[id b] A.

A= id{.id}.

Неясно, какое правило выбрать, т.к. [id b] и A начинаются с одинаковых токенов. В таком случае лучше всего переписать грамматику:

S= a id ( b A| {. id}).

### Левая рекурсия.

Левая рекурсия для LL грамматик никаким образом не допустима. Ее нужно удалить путем преобразования. К счастью любую левую рекурсию можно преобразовать в правую.

A->Aa1|...|Aan|b1...|bm

заменить на

A->b1B|..|bmB

B->a1B|..|anB|e

Запись в РБНФ:

A = A b| c.

заменить на

A = c {b}.

### Семантическая значимость.

В некоторых случаях выбор правил производится исходя из их семантики. Например, выражение с типами:

Expr=Factor { '+' Factor }.

Factor = '('ident')' Factor | '(' Expr ')' | ident| number.

Т.е. такая грамматика допускает следующий цепочки:

a+b

(a)+b

(int)a+b

В последней цепочке выбор правила '('ident')' Factor обусловлен семантикой идентификатора. Т.е. Это правило выбирается, если у нас в качестве ident выступает тип.

! Крайне неудачный пример с точки зрения построения языка. Обычно в грамматике «ключевые слова» описываются отдельным правилом. Тогда нет необходимости в проверках идентификатора.

Другой пример.

A= ident (. x=1; .) {', ' ident (.x++;.) } ':' | ident (.Foo();.) {',' ident (.bar();.) } ';'.

В этом случае нельзя отредактировать грамматику, т. к. у одинаковых частей правила разные семантические действия. Чтобы определить правило, необходимо просмотреть все токены до двоеточия или точки с запятой. Только тогда станет ясно, какое правило выбрать.

Решение:

В грамматику можно вставить булеву функцию, по которой будет выбрана альтернатива.

S= a[id b] A.

A= id{.id}.

S= a [ IF(isAlias()) id b] A.

IsAlias() функция, которая просматривает 2 впередистоящих токена. Если это b, то возвращает true.

Token t -только что распознанный токен

Token la — следующий токен

t.val — значение токена

t.kind — тип токена, определяется лексером

A= IF (FollowedByColon())

ident (. x=1; .) {', ' ident (.x++;.) } ':'

| ident (.Foo();.) {', ' ident (. Bar();.) } ';'.

bool FollowedByColon(){

//берем следующий токен

Token x = la;

//пока текущий токен запятая или переменная

while(x.kind==\_comma || x.kind== \_ident)

//двигаем сканер на следующий токен

x=scanner.Peek();

//возвращаем true, если мы встретили двоеточие

return x.kind==\_colon;

}

Примечания:

**IF**ключевое слово языка генератора.

Функция FollowedByColon() относится к первому правилу. Если она выдала true, то именно его она и рассматривает.

Имена типам токенов присваивает сканер. Но если в секции TOKENS сделать такие объявления

ident = letter {letter | digit | "\_"}.

comma = ','.

semicolon = ';'.

colon = ':'.

То сканер сгенерирует константы с хорошими именами:

const int \_EOF=0;

const int \_ident=1;

const int \_comma=2;

const int \_semicolon=3;

const int \_colon=4;

С точки зрения построения языка, отдельное описание каждого специального символа как токена не имеет смысла. Но если возникла потребность в написании условий, в которых требуется проверка впередистоящих токенов, такое описание может быть полезно.

Если в первом правиле у вас была функция, в которой вы проверяли токены, а во втором правиле у вас тоже есть функция, то сканер должен вернуться в первоначальное положение. Сбросить позицию можно функцией scanner.ResetPeek().

## Порядок выполнения работы

* 1. Разработать синтаксис и семантику языка для заданной предметной области.
  2. Представить грамматику языка в форме РНБФ.
  3. Привести грамматику к классу LL, удалив левую рекурсию и правила, имеющие общую левую часть.
  4. Подготовить файл грамматики \*.ATG.
  5. Сгенерировать сканер и парсер для заданного языка.
  6. Откомпилировать транслятор языка.
  7. Протестировать работу транслятора на тестовых программах на предметно-ориентированном языке.

# Лабораторная работа № 2. Трансляция программы на предметно-ориентированном языке в исполняемый файл.

## Порядок выполнения работы

1. Разработать алгоритмы реализации конструкций заданного предметно-ориентированного языка.
2. Разработать программные модули реализации конструкций заданного языка.
3. Сгенерировать сканер и парсер для заданного языка.
4. Откомпилировать транслятор языка.
5. Протестировать работу транслятора на тестовых программах на предметно-ориентированном языке.

# Варианты заданий

Каждый предметно-ориентированный язык должен включать:

* + стандартные арифметические операции и типы данных;
  + не менее 2 типов данных для объектов предметной области (описание и использование);
  + операции над введенными типами данных (не менее 4);
  + операторы ввода и вывода для организации диалога с пользователем в текстовом или графическом режиме;
  + операторы условного, безусловного перехода и цикла.

|  |  |
| --- | --- |
| Вариант | Предметная область |
|  | Шахматная партия |
|  | Музыкальное произведение |
|  | Матричные и векторные операции |
|  | Размещение геометрических объектов на плоскости |
|  | Описание и композиция машин Тьюринга |
|  | Детерминированные конечные автоматы |
|  | Автоматы с магазинной памятью |
|  | Графы и операции над ними |
|  | Таблицы и работа с ними |
|  | Составление компьютерных тестов |
|  | Системы линейных уравнений и их решения |
|  | Множества и операции над ними |
|  | Предметная область по выбору студента |

# Источники информации

1. Официальный сайт Coco/R <http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/>

2. Руководство пользователя Coco/R