МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное

образовательное учреждение высшего образования

«Тульский государственный университет»

Институт «Прикладной математики и компьютерных наук»

Кафедра «Вычислительной техники»

|  |
| --- |
| Утверждено на заседании кафедры  «Вычислительной техники»  «29» января 2019 г., протокол № 06 |
| Заведующий кафедрой  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.Н.Ивутин |

**Дисциплина**

**Системное программное обеспечение**

**Методические указания по выполнению курсовой работы**

Направление подготовки

**09.03.01 Информатика и вычислительная техника**

Профиль 1

**Электронно-вычислительные машины, комплексы, системы и сети**

Профиль 2

**Программное обеспечение средств вычислительной техники и автоматизированных систем**

Тула 2019 год

**Разработчик:**

Савин Н.И., доцент каф. ВТ, канд. техн. наук , доцент.

Методические указания к выполнению курсовой работы обсуждены на заседании кафедры ВТ ИПМиКН

протокол № 06 от 29.01.19 г.

***Зав. кафедрой А.Н.Ивутин***

# ЦЕЛЬ И ЗАДАЧИ ВЫПОЛНЕНИЯ КУРСОВОЙ РАБОТЫ

Курсовая работа предназначена для закрепления теоретических знаний в области проектирования программного обеспечения ЭВМ и получения практических навыков составления и отладки грамматик алгоритмических языков и реализации основных компонент трансляторов.

# ТРЕБОВАНИЯ К КУРСОВОЙ РАБОТЕ

## Тематика курсовой работы

Курсовая работа выполняется в полном соответствии с вариантом задания. Задание выдается в начале семестра, срок выполнения работы не превышает одного семестра.

В работе должны быть разработаны:

-грамматика языка программирования;

-лексический анализатор;

-синтаксический анализатор;

-программа генерации объектного кода.

***Объектный код одинаков для всех вариантов*** - Макроассемблер IBM PC.

Курсовая работа оформляется в соответствии с действующими стандартами на оформление программной документации и требованиями кафедры ЭВМ.

В пояснительной записке приводятся следующие разделы:

- техническое задание на проектирование;

- обзор литературных источников;

- постановка задачи на проектирование;

- разработка грамматики языка программирования;

- разработка лексического анализатора;

- разработка синтаксического анализатора;

- разработка программ генерации объектного кода;

- исходный текст программы транслятора;

- тестовые примеры с текстами промежуточного кода, лексической свертки, диагностикой ошибок.

Курсовая работа представляется на защиту в законченном виде, полностью работоспособной и правильно оформленной.

## Перечень вариантов заданий

Каждый вариант кодируется шестизначным числом.

Первая цифра ‑ язык программирования для которого проектируется транслятор:

1‑PASCAL; 2‑ С; 3‑FORTRAN; 4‑BASIC.

Вторая цифра ‑ подмножество языка:

1 - арифметические выражения целого типа, операторы присваивания, цикла с предусловием, бесформатного ввода-вывода;

2 ‑ арифметические выражения целого типа, операторы присваивания, условный, бесформатного ввода-вывода;

3 ‑ арифметические выражения вещественного типа, операторы присваивания, цикла с предусловием, бесформатного ввода-вывода;

4 ‑ арифметические выражения целого типа с индексированными и простыми переменными, операторы присваивания, цикла с постусловием, бесформатного ввода-вывода;

5 ‑ арифметические выражения целого типа, функции, операторы присваивания, условный, бесформатного ввода-вывода;

6 ‑ арифметические выражения целого типа, логические выражения операторы присваивания, цикла с предусловием, бесформатного ввода-вывода.

Третья цифра ‑ тип промежуточного кода программы:

1‑польская запись;

2‑дерево;

3‑тетрады;

4‑триады.

Четвертая цифра ‑ тип транслятора:

1‑компилятор;

2‑интерпретатор.

Пятая цифра ‑ метод разбора:

1‑нисходящий с возвратами;

2‑нисходящий без возвратов;

3‑простое предшествование;

4‑расширенное предшествование;

5‑алгоритм Эрли.

Шестая цифра ‑ язык программирования, на котором производится реализация транслятора:

1‑С++;

2‑С#;

3‑Python.

**Указание**. *Объектный код для всех вариантов – Ассемблерная программа.*

Таблица 1. Варианты курсовых работ для группы 1 (для группы 2 см. в прил.)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Варианты курсовой работы для первой группы потока | | | | | | | |
| № жур | № вар | Номера цифр кода варианта | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 |
| Значения цифр кода варианта | | | | | |
| 1 | 1 | 2 | 5 | 3 | 1 | 4 | 2 |
| 2 | 2 | 3 | 6 | 4 | 2 | 5 | 3 |
| 3 | 3 | 4 | 1 | 1 | 1 | 1 | 1 |
| 4 | 4 | 1 | 2 | 2 | 2 | 2 | 2 |
| 5 | 5 | 2 | 3 | 3 | 1 | 3 | 3 |
| 6 | 6 | 3 | 4 | 4 | 2 | 4 | 1 |
| 7 | 7 | 4 | 5 | 1 | 1 | 5 | 2 |
| 8 | 8 | 1 | 6 | 2 | 2 | 1 | 3 |
| 9 | 9 | 2 | 1 | 3 | 1 | 2 | 1 |
| 10 | 10 | 3 | 2 | 4 | 2 | 3 | 2 |
| 11 | 11 | 4 | 3 | 1 | 1 | 4 | 3 |
| 12 | 12 | 1 | 4 | 2 | 2 | 5 | 1 |
| 13 | 13 | 2 | 5 | 3 | 1 | 1 | 2 |
| 14 | 14 | 3 | 6 | 4 | 2 | 2 | 3 |
| 15 | 15 | 4 | 1 | 1 | 1 | 3 | 1 |
| 16 | 16 | 1 | 2 | 2 | 2 | 4 | 2 |
| 17 | 17 | 2 | 3 | 3 | 1 | 5 | 3 |
| 18 | 18 | 3 | 4 | 4 | 2 | 1 | 1 |
| 19 | 19 | 4 | 5 | 1 | 1 | 2 | 2 |
| 20 | 20 | 1 | 6 | 2 | 2 | 3 | 3 |
| 21 | 21 | 2 | 1 | 3 | 1 | 4 | 1 |
| 22 | 22 | 3 | 2 | 4 | 2 | 5 | 2 |
| 23 | 23 | 4 | 3 | 1 | 1 | 1 | 3 |
| 24 | 24 | 1 | 4 | 2 | 2 | 2 | 1 |
| 25 | 25 | 2 | 5 | 3 | 1 | 3 | 2 |

# ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

В задачу курсовой работы входит проектирование транслятора программ, написанных на алгоритмическом языке высокого уровня.

Под трансляцией понимается перевод исходного текста программы в объектный код. Схема процесса трансляции приведена на рис. 1.

Процесс перевода включает следующие этапы:

‑ лексический анализ;

‑ синтаксический анализ;

‑ получение промежуточного кода программы;

‑ оптимизация кода;

‑ генерация объектного кода;

Одним из основных требований, предъявляемых к трансляторам, является их способность обрабатывать любые входные цепочки символов, в том числе и ошибочные. В последнем случае транслятор должен выводить диагностическую информацию о предполагаемых ошибках.

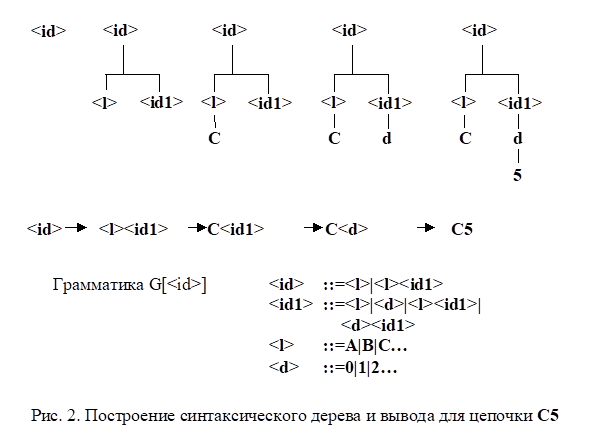
В целом, транслятор должен проводить анализ исходного текста программы, а затем синтез объектного кода.

# Синтаксический анализатор.

## Нисходящие методы разбора.

Нисходящий разбор с возвратами.

По алгоритму нисходящего разбора строится синтаксическое дерево, начиная с корня, постепенно спускаясь до уровня предложения, как это показано на рис.2.



На рисунке приведены конечные стадии разбора и не показаны деревья, которые соответствуют локальным неудачным попыткам. Алгоритм разбора должен обладать той особенностью, что при неудачной попытке разбора он должен перебирать варианты разбора сентенциальной формы вплоть до первой удачной, либо до конца всего множества вариантов. Эффективность алгоритма повышается, если он в полной мере использует уже разобранные фрагменты входной цепочки.

Сущность сокращенного алгоритма разбора можно описать следующим образом. Пусть требуется произвести разбор цепочки ***x***. Предложение должно удовлетворять грамматике **G[Z]**, имеющей вид:

Z::=U|V|W

U::=U1U2U3|U4U5U6...

V::=V1V2V3...

W::=W1W2W3...

...

Задачей разбора является поиск вывода **Z→+*x***, где **Z**‑ начальный символ. Следовательно, первым непосредственным выводом должен быть вывод **Z→U** в соответствии с правилом **Z::=U**. Если построить общий вывод, начинающийся с указанного непосредственного вывода нельзя, следует применить первым непосредственный вывод **Z→V** и т.д.. При выборе одного из возможных выводов, например **Z→U**, ставится задача непосредственного вывода **U→U1U2U3**, а затем **U1→+*x1***, где ***x1*** -некоторая подцепочка цепочки ***x***, такая, что ***x***= ***x1x2 …*** Если последний вывод произведен удачно, то выполняется вывод **U2→+*x2*** и т.д.. Успех вывода **U1→+*x1*** зависит от одновременного успеха множества конечных непосредственных выводов, порождаемых соответствующими правилами вывода и имеющих вид **Pi→ Tj**, где **Pi** -нетерминальный символ грамматики, а **Tj** -текущий терминальный символ входной строки. Успешность непосредственного вывода **Pi→Tj** зависит от того совпадает ли символ **Tj** с соответствующим текущим символом входной цепочки. Если непосредственный вывод удачен, то текущим символом становится следующий символ входной цепочки Рассмотрим для примера случай, когда в приведенной выше грамматике правило для **U** имеет вид

U ::=id+id|id-id,

а входная цепочка "***a-b***" после обработки сканером имеет будет представлена как **id-id**. Тогда вывод **U1→+X1** условно выглядит как **id→id** и этот непосредственный вывод успешен, т.к. символ **U1** является терминальным и совпадает с текущим символом входной цепочки. Cообщение об успехе передается символу **U**, который рассматривает следующий текущий символ в правой части правила и порождает процесс вывода **U2→+X2**. Одновременно следующим текущим символом входной цепочки становится "‑". Однако в этом случае сравнение порожденного терминального символа "**+**" и текущего символа входной цепочки "**‑**"неуспешен. Следовательно символу **U** передается сообщение о неуспехе. Символ **U** возвращается на шаг назад и порождает новый вывод **U1→+ X1**. Если породить новый вывод невозможно, то осуществляется переход к следующей альтернативе, т.е. порождается непосредственный вывод **U→U4U5U6**. В данном примере это будет вывод **U**→**id‑id**, который полностью успешен, следовательно символу **U** необходимо передать сообщение об успехе. Символ **U** продвигается на один символ вперед по своей текущей альтернативе и, в общем случае, порождает новый непосредственный вывод. В данном примере достигается конец альтернативы, что означает успех всего разбора в целом. Из изложенного следует сделать такие выводы:

- при возврате продвижение по текущей альтернативе производится относительно места откуда был произведен соответствующий вывод, поэтому при порождении вывода это место необходимо запоминать;

– если при возврате получено сообщение об успехе, то продвижение по текущей альтернативе производится на один символ в прямом направлении;

– если при возврате получено сообщение о неуспехе, то продвижение по текущей альтернативе производится на один символ в обратном направлении;

– порождение выводов производится в соответствии с правилами заданной грамматики.

Положительной чертой этого метода можно считать то, что каждый символ должен помнить лишь о своей цели, о своем родителе, о своих непосредственных потомках, а также о своем месте в грамматике и во входной цепочке. Происходит абстрагирование от точных сведениях о том, что выполняется в других местах, т.е. в каждом сегменте программы или в подпрограмме необходимо заботится лишь о собственной входной и выходной информации.

Особенности реализация программы нисходящего разбора.

Задание грамматики.

Один из способов задания состоит в представлении грамматики списком в одномерном массиве GRAMMAR таким образом, что каждое множество правил **U::=x|y|...|z** представлено как **Ux|y|...|z|$**. То есть каждый символ занимает один элемент массива, за каждой правой частью **U** следует символ конца альтернативы – вертикальная черта **|**, а за последней правой частью **U** следует пара символов **|$** (конец альтернативы и конец правила). Таким образом, грамматика

|  |
| --- |
| Z::=E#  E::=T+E|T  T::=F\*T|F  F::=(E)|id |

будет выглядеть как GRAMMAR=(**ZE#|$ET+E|T|$TF\*T|F|$F(E)|id|$**).

Для порождения и уничтожения выводов в программе используется стек типа LIFO (Last–In–First–Out). Как правило для реализации стека используются массив **S** и счетчик **v**. При **v=0** стек пуст. При **v=n**, где **n>0**, в стеке находятся элементы **S(1)**, **S(2)**,..., **S(n)**, где **S(n)**–верхний элемент стека.

Каждый элемент стека соответствует одному порожденному непосредственному выводу и состоит из пяти компонент

**(GOAL, i, FAT, SON, BRO)**, которые означают следующее.

**GOAL** – цель, т.е. символ, который подлежит разбору. Таким образом, в незакрытой в данный момент части входной цепочки предстоит найти такую голову, которая приводится к **GOAL**, и закрыть эту часть цепочки символом **GOAL**. **GOAL** передается элементу стека родителем.

**i** – индекс в массиве GRAMMAR, указывающий на тот символ в правой части правила для **GOAL**, с которым производится работа в данный момент.

**FAT** – имя родителя (номер элемента стека, соответствующего родителю).

**SON** – имя самого последнего из непосредственных выводов для текущей альтернативы.

**BRO** – предыдущего непосредственного вывода в текущей альтернативе.

Нуль в любом из полей означает, что данный элемент отсутствует.

Массив GRAMMAR имеет 29 элементов:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Z | E | # | | | $ | E | T | + | E | | | T | | | $ | T | F | \* | T | | | F | | | $ | F | ( | E | ) | | | i | | | $ |
| 1. 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

В качестве иллюстрации работы распознавателя на рис.3 приведено синтаксическое дерево и стек для предложения **id\*id+id**.

Из содержимого стека видно, что имеется список непосредственных потомков каждого родителя и элементы этого списка "связаны" между собой. Стек в его окончательном виде есть ни что иное, как внутренняя форма синтаксического дерева. Следует отметить, что в конце каждого предложения в соответствии с грамматикой используется разделитель. По нему можно судить о том, что окончен разбор всего предложения, а не только какой–то из его подцепочек.

F

id

Z

#

E

E

T

+

\*

T

F

id

T

F

id

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТЕК | GOAL | i | FAT | SON | BRO |
| 1 | Z | 4 | 0 | 15 | 0 |
| 2 | E | 10 | 1 | 11 | 0 |
| 3 | T | 18 | 2 | 7 | 0 |
| 4 | F | 28 | 3 | 5 | 0 |
| 5 | id | 0 | 4 | 0 | 0 |
| 6 | \* | 0 | 3 | 0 | 3 |
| 7 | T | 20 | 3 | 8 | 6 |
| 8 | F | 28 | 7 | 9 | 0 |
| 9 | id | 0 | 8 | 0 | 0 |
| 10 | + | 0 | 2 | 0 | 3 |
| 11 | E | 12 | 2 | 12 | 10 |
| 12 | T | 20 | 11 | 13 | 0 |
| 13 | F | 28 | 12 | 14 | 0 |
| 14 | id | 0 | 13 | 0 | 0 |
| 15 | # | 0 | 1 | 0 | 2 |

Рис. 3. Синтаксическое дерево и стек для выражения **id\*id+id**

Требования к грамматикам для нисходящего разбора.

В алгоритме, описанном выше, есть недостаток, который проявляется, когда цель определена с использованием левосторонней рекурсии. Если **X**– цель, а первое же правило для **X** имеет вид **X::=X...**, то происходит бесконечное порождение непосредственных выводов **X→X...** Поэтому грамматика написана с применением правосторонней рекурсии вместо привычной левосторонней. Более предпочтительный способ избавится от прямой левосторонней рекурсии – записывать правила, используя итеративные и факультативные цепочки. При этом правила преобразуются к следующему виду.

Вместо E::=E+T|T будет E::=T{+T}.

Вместо T::=T\*F|T/F|F будет T::=F{\*F|/F}

На практике используются два принципа, на основании которых правила языка, включающие прямую левостороннюю рекурсию, преобразуются в эквивалентные правила, использующие итерацию.

Факторизация.

Если существуют правила вида

**U::=xy | xw |...| xz**, то их надо заменить на

**U::=x(y|w|...|z)**, где скобки **(**и **)** являются метасимволами.

Допустима факторизация и в более общей форме, такая как в арифметических выражениях. Например, если **y=fv, w=fm** то **U::=x(f(v|m)|....** Следует отметить, что исходные правила U::=x|xy преобразуются к виду **U::=x(y|λ)**, где **λ**–пустая цепочка. Когда бы не использовалось подобное преобразование, **λ**всегда помещается как альтернатива, так как мы принимается условие, что если цель –**λ**, то эта цель всегда сопоставляется. Помимо того что факторизация позволяет исключать прямую рекурсию, использование этого приема сокращает размеры грамматики и позволяет проводить разбор более эффективно.

Итерация.

После факторизации в грамматике останется не более одной правой части с прямой левосторонней рекурсией для каждого из нетерминалов. Если такая правая часть есть, необходимо выполнить следующее. Пусть **U::=x|y|...|z|Uv** – правила, у которых осталась леворекурсивная правая часть. Эти правила означают, что членом синтаксического класса **U** является **x**, **y**или z, за которыми ничего не следует, либо следует сколько–то v. Тогда правила преобразуются к виду **U::=(x|y|...|z) {v}**.

После таких изменений должен измениться и алгоритм нисходящего разбора. Теперь алгоритм должен уметь обрабатывать альтернативы не только во всей правой части, но и в подцепочках, должен учитывать в своей работе существование пустой цепочки **λ**, должен уметь обрабатывать итерацию. Устранение прямой левосторонней рекурсии не позволяет устранить общую левостороннюю рекурсию. Таким образом, правила **U::=Vx** и **V::=Uy|v** дают вывод **U→+Uyx**. Избавится от этого не так просто, но обнаружить такую ситуацию возможно, если использовать следующий под– ход. Из исходной грамматики исключаются все правила с левосторонней рекурсией. Символ **U**, получившейся в результате преобразования грамматики, может быть леворекурсивным тогда и только тогда, когда **U FIRST+ U**. Существуют алгоритмы проверки этого отношения.

## Восходящие методы разбора

Простое предшествование.

Метод простого предшествования основан на том, что между двумя соседними символами **Si Si+1** приводимой строки могут существовать только три отношения:

**Si<•Si+1** - отношение верно, если символ **Si+1** - самый левый символ некоторой основы;

**Si•>Si+1** - отношение верно, когда **Si** - символ, являющийся самым правым в некоторой основе;

**Si•=Si+1** - верно, если **Si** и **Si+1** принадлежат одной основе.

Эти отношения можно пояснить следующим образом. Для текущей цепочки

...Si<•Si+1•=…•=Sj•>Sj+1...Sk•>Sk+1...

символы **Si+1…Sj** составляют основу.

Отношения **<•, •=, •>** называются отношениями предшествования.

Отношения предшествования являются отношениями порядка, т.е. они зависят от порядка следования символов **Si** и **Si+1**.Если имеется отношение **Si•=Si+1** - это не означает, что **Si+1•=Si** .

Если для каждой пары символов грамматики существует не более, чем одно отношение предшествования, то на каждом шаге синтаксического анализа можно выделить основу. Основой называется самая левая

группа символов, связанных отношениями предшествования вида:

<•Si•= Si+1… Sj-1•=Sj•>.

Грамматика предшествования.

Пусть **U,C,D** - нетерминальные символы; ***x,y,ω*** - любые цепочки.

Грамматикой предшествования называется такая грамматика класса 2, которая отвечает следующим двум требованиям:

‑ для каждой упорядоченной пары терминальных и нетерминальных символов должно выполняется не более, чем одно из трех отношений предшествования, определяемых следующим образом.

**Si=•Sj**, если существует правило **U::=*x* SiSi+1*y***;

**Si<•Sj**,если существует правило **U::=*x*SiD*y*** и существует вывод из **D** цепочки **Si*ω***;

**Si•>Sj**,если существует правило **U::=*x* CSi+1*y*** и существует вывод C→\****ω*Si**.

‑ разные порождающие правила должны иметь разные правые части. Например, правило для списка идентификаторов ‑

id\_list::=id|id\_list,id

и правило для элемента выражения

F::=id|(E)…

имеют одинаковые правые, но разные левые части. Устранить отрицательные последствия таких правил можно лишь преобразовав надлежащим образом грамматику. Например, если ввести несколько иной смысл в общепринятое понятие списка элементов, то правило грамматики примет вид.

id\_list::=id,id|id\_list,id

Распознаватель метода простого предшествования.

Алгоритм разбора, основанный на отношениях предшествования, заключается в следующем. Символы входной строки поочередно переписываются в стек до тех пор, пока между символом на вершине стека **Sl** и очередным символом входной строки **Si** не появится отношение **•>**. Это признак того, что в стеке находится основа. Стек в этом случае просматривается в направлении от вершины к началу до тех пор, пока между двумя очередными символами стека не появится отношение **<•, т.е. Sк-1<•Sk**. В этом случае часть стека от вершины **Sl** до символа **Sk** является основой. После этого среди порождающих правил отыскивается правило  
 **U::= Sk…Sl**,   
и последовательность символов в стеке **Sk…Sl**заменяется на найденный символ **U.** Далее процесс повторяется аналогично.

Если на некотором шаге процесса разбора обнаружится, что между символами **Sp** и **Sq** не существует отношения предшествования, то это означает, что во входной цепочке допущена синтаксическая ошибка.

Матрица предшествования.

Матрица предшествования формируется на предварительном этапе

проектирования транслятора и должна содержать отношения предшествования между всеми символами грамматики. Обозначим **L(U)** - множество самых левых символов относительно нетерминала**U**. Формально это множество определяется следующим образом.

L(U)={S| ∃ (U→\*Sz)},

где S-любая строка, возможно пустая, ∃-квантор существования.

Это множество можно определить рекурсивно:

L(U)={S| ∃(U::=S...)∪ [∃(U::=U'...) ∩ S∈L(U')]}.

Аналогично определяется множество **R(U)**.

С использованием множеств **R(U)** и **L(U)** отношения предшествования принимают вид пригодный для практического использования.

Si=•Sj⇔ ∃(U::=...SiSj...)

Si<•Sj⇔ ∃(U::=...SiD…) ∩ Sj∈L(D)

Si•>Sj⇔ ∃(U::=...DSj…) ∩ Si∈R(D)

где **D**-нетерминальный символ.

Построение **L(U)** и **R(U)** происходит следующим образом.

Для каждого нетерминального символа грамматики отыскивается

правило, в левой части которого находится символ **U**. В множество **L(U)** включаются все самые левые символы правых частей правил для **U**. В множество**(U)**включаются все правые символы из правил для **U**.

Рассматривается полученное множество **L(U)**.Если оно содержит нетерминальные символы **U',U'',...**, то левые символы для **U',U'',...**заносятся в **L(U)**.

Аналогично, если **R(U)** содержит нетерминальный символ **U1,U2,...**,то в **R(U)** включаются все правые символы для **U1,U2,...** Процесс продолжается до тех пор, пока множества **L(U)** и **R(U)** не перестают изменяться.

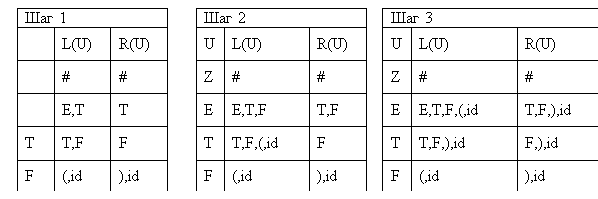
Пример. Для приведенной ниже грамматики показаны шаги формирования множеств левых и правых символов:

Z::=#E#

E::=E+E|T

T::=T\*F|F

F::=(E)|id



Построение матрицы предшествования.

Матрица предшествования имеет размерность n\*n, где n - количество символов грамматики. Символы грамматики записываются в первый столбец и первую строку, которые называются входными для отыскания отношений предшествования. Первый символ отношения выбирается из входного столбца, второй символ отношения ‑ из входной строки.

На первом этапе отыскиваются символы, между которыми существует отношение **=• .**Это отношение полностью определено правилами грамматики. Для найденных символов в соответствующую ячейку матрицы предшествования записывается символ отношения.

На втором этапе отыскиваются символы, между которыми существует отношение <**• .**Эти символы определяются путем просмотра правых частей правил грамматики и затем использования определяющего соотношения. Символ отношения записывается в соответствующую ячейку матрицы.

На третьем этапе аналогичным образом отыскивается отношение **•>**.

Пример матрицы предшествования для приведенной выше грамматики показан в табл.1.

Из матрицы предшествования видно, что данная грамматика не является грамматикой предшествования, так как существует неоднозначность отношений предшествования. Неоднозначность отношений предшествования для данной грамматики является следствием того, что в методе простого предшествования используется минимальный контекст.

Для устранения неоднозначности отношений предшествования следует использовать более глубокий контекст.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 2. Матрица предшествования | | | | | | | | | |
| Si\Sj | E | T | F | + | \* | ( | ) | id | # |
| E |  |  |  | =• |  |  | =• |  | =• |
| T |  |  |  | •> | =• |  | •> |  | •> |
| F |  |  |  | •> | •> |  | •> |  | •> |
| + |  | =• | <• |  |  | <• |  | <• |  |
| <• |
| \* |  |  | =• |  |  | <• |  | <• |  |
| ( | =• | <• | <• |  |  | <• |  | <• |  |
| <• |
| ) |  |  |  | •> | •> |  | •> |  | •> |
| id |  |  |  | •> | •> |  | •> |  | •> |
| # | =• | <• | <• |  |  | <• |  | <• |  |
| <• |

Так, в контексте **...(Е)...** справедливо отношение **(=•Е** и возможна прямая редукция**:(Е)→E**.

В контексте **...(Е+Т)...** или **...+T\*F...** между ( и Е имеется отношение **<•** и необходима прямая редукция **Е+Т=>E**, а между **+** и **Т** справедливо отношение **<•**и ,следовательно, необходима редукция**Т\*F=>T**.

В данном случае формально неоднозначность отношений предшествования вытекает из того, что, например, **(** и **Е** встречаются рядом в правой части, а с другой стороны **Е** принадлежит **L(E)**,то есть **L(E)** рекурсивно для символа **Е**. Избежать неоднозначность отношений предшествования можно, используя метод стратификации(отделения), который устраняет рекурсивность множества **L(U)** или **R(U)** для тех символов U, для которых отношения неоднозначны.

Например, после введения новых правил

E::=E'

E'::=E+T|T

множество левых символов **L(E)={E',T,F,(,id}** уже не содержит символа **Е**.

Аналогично вводя стратификацию для символа T можно записать:

|  |  |
| --- | --- |
| Z::=#E  E::=E'  E'::=E'+T|T | T::=T'  T'::=T'\*F|F  F::=(E)|id. |

Расширенное предшествование.

Практическое применение метода простого предшествования ограничено вследствие появления неоднозначности отношений предшествования, которые часто встречаются при проектировании грамматик реальных языков программирования. Общего алгоритма приведения исходной грамматики к грамматике, отвечающей требованиям метода простого предшествования не существует. Поэтому разработаны методы, позволяющие преодолеть отрицательные эффекты неоднозначных отношений. Одним из таких методов является метод расширенного предшествования. Ниже рассмотрены основные положения, связанные с использованием расширенного предшествования **(1,2)(2,1)**.

Основная идея метода расширенного предшествования состоит в следующем. Пусть между символами **SJ**и **SJ+1** существует неоднозначное отношение, например, **SJ**<•**SJ+1** и **SJ** =•**SJ+1.** Требуется определить, какое отношение нужно взять в контексте **...SJ SJ+1 SJ+2.…** Если справедливо отношение <• и грамматика однозначна, то должно существовать правило, которое начинается символами **SJ+1SJ+2.…** ,то есть правило имеет вид: **U::=SJ+1SJ+2….** Если такого правила не существует, то справедливым будет отношение =•. Аналогично, если между символами **SJ**и **SJ+2** существуют отношения **SJ+1** =•**SJ+2** и **SJ+1** •>**SJ+2**и требуется определить, какое из них необходимо взять в контексте ...**SJSJ+1SJ+2…**то есть является ли **SJ+1** правым концом основы, то отношение •> будет верным, если существует правило вида **U::=...SJSJ+1.** В противном случае надо брать другое конфликтное отношение.

С целью сокращения просмотров грамматики при определении верного отношения предшествования предварительно заготавливается вспомогательная информация в виде множеств левых и правых троек. Множество левых троек используется для выделения левого конца основы, множество правых троек - для выделения правого конца основы.

Множество левых троек **LT** содержит такие тройки символов **S1S2S3**, что символы **S2S3** начинают правую часть какого-либо правила, а между символами **S1** и**S2** имеется неоднозначность отношений (=•,<•).

Множество правых троек **RT** содержит такие тройки символв**S1S2S3**, что символы **S2S3** заканчивают правую часть какого-либо правила, а между символами **S1** и**S2** имеется неоднозначность отношений (=•,•>).

Составление таблицы правых троек.

Просмотреть матрицу предшествования и найти пары символов **S2S3**,для которых существуют конфликтные отношения (=•,•>).

Для каждой выявленной пары символов **S2S3** просмотреть столбец **S2** матрицы предшествования и найти все символы **S1**, связанные с **S2** отношением **S1**=•**S2.**

Просмотреть порождающие правила. Если существует правило, которое заканчивается парой **S1S2**т.е. **U::=... S1S2**,то тройка **S1S2S3** записывается в таблицу правых троек.

Составление таблицы левых троек.

Просмотреть матрицу предшествования и найти пары символов **S1S3**,для которых существуют конфликтные отношения (=•,<•).

Для каждой выявленной пары символов **S1S2** просмотреть строку **S2** матрицы предшествования и найти все символы **S3**, связанные с **S2** отношением **S2**=•**S3.**

Просмотреть порождающие правила. Если существует правило, которое начинается парой **S2S3**т.е. **U::=S2S3...**,то тройка **S1S2S3** записывается в таблицу правых троек.

Для примера, приведенного выше видно, что неоднозначных отношений (=•,•>) не существует, поэтому множество правых троек будет пустым.

Множество левых троек LT={ # E + , ( E + , + T \* }

## Семантические программы

В результате синтаксического и лексического анализов производится некоторое в общем случае неэквивалентное преобразование исходного текста в промежуточный код. Семантические же программы осуществляют перевод программы в форму, эквивалентную исходному тексту. Семантическая обработка, как правило, связывается с синтаксической и предназначена для формирования одного из промежуточных кодов программы. На практике используются следующие промежуточные коды: польская запись; тетрады и триады. Этап семантической обработки необходим для упрощения задачи перевода программы в объектный код.

Формирование польской записи для исходной программы.Рассмотрим восходящий метод разбора. Синтаксический распознаватель всякий раз, когда найдена основа, и её можно привести к нетерминальному символу **U**, вызывает программу, связанную с правилом **U::=x**, где **х** - найденная основа. Эта программа осуществляет семантическую обработку всех символов цепочки х и формирует часть символов польской записи, которая имеет непосредственное отношение к **х**. При этом, с учетом того, что основа редуцируется при каждом её нахождении, если в основе встречается некоторый нетерминал**V**, то часть польской записи, включающая подцепочку, которая приводится к **V**, уже сгенерирована.

Предположим, что генерируемая польская запись находится в одномерном массиве **Р**. Целая переменная **J** (начальное значение равно единице) содержит индекс, который указывает на первый свободный элемент массива **Р**. Каждый элемент массива может содержать один символ: идентификатор или оператор. Пусть вызванная семантическая программа имеет доступ к символам основы **S[1],S[2],...,S[i]**. Рассмотрим программу, связанную с правилом **Е1::=Е2+Т**. Если данная семантическая программа вызвана, то польская запись для **Е2** и **Т** уже получена, т. е. она содержит **<код для Е2><код для Т>** . Кроме того символ, который находится правее**Т** в исходной программе, является терминальным. Следовательно от данной семантической программы требуется только занести символ **'+'** в польскую запись **...<код для Е2><код для Т>+**. Т.е. инфиксная запись **Е2+Т** переводится в польскую запись **Е2Т+**.

Итак, семантическая программа имеет следующий вид:

P[J]:='+'; J:=J+1.

Семантическая программа, связанная с правилом **F::=id**, имеет вид

P[J]:=S[i]; J:=J+1,

где **S[i]** - это вершина синтаксического стека.

Семантическая программа, связанная с правилом **F::=(E)**, никаких действий не производит, т. к. цепочка для**Е** уже сформирована, а приоритет операции в польской записи задается порядком следования операндов и операторов, т. е. без применения скобок.

Все типы семантических программ для грамматики алгебраических выражений приведенной выше можно свести в следующую таблицу.

Таблица 3. Семантические программы

|  |  |  |
| --- | --- | --- |
| N | Правило | Семантическая программа |
| 1 | Z::=E | нет |
| 2 | E::=T | нет |
| 3 | E::=E+T | P[J]:='+'; J:=J+1; |
| 4 | E::=E-T | P[J]:='-'; J:=J+1; |
| 5 | E::=-T | P[J]:='@'; J:=J+1; |
| 6 | T::=F | нет |
| 7 | T::=T\*F | P[J]:='\*'; J:=J+1; |
| 8 | T::=T/F | P[J]:='/'; J:=J+1; |
| 9 | F::=id | P[J]:=S[i]; J:=J+1; |
| 10 | F::=(E) | нет |

Формирование промежуточного кода в виде тетрад и триад имеет некоторые особенности, связанные с сохранением семантической информации при редукции основы, остальные принципы аналогичны тем, по которым формируется польская запись.

## Генерация объектного кода.

Для генерации объектного кода необходимо заготовить тексты программ на объектном языке, выполняющих действия полностью соответствующие по смыслу вызываемым семантическим программам. В указанных заготовках необходимо предусмотреть поля, в которые при генерации заносится конкретизирующая информация, такая, например, как имена переменных.

В качестве объектного кода используется язык Макроассемблера.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Системное программное обеспечение /А.В. Гордеев, А.Ю. Молчанов. – СПб.: Питер, 2001. – 736 с.
2. Бек Л. Введение в системное программирование. - М.:Мир,1988. - 448 с.
3. Лебедев В.М. Введение в системы программирования. М.: Статистика, 1975. - 312 с.
4. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. - М.: Мир, 1978, в 2-х т.
5. Грис Д. Конструирование компиляторов для цифровых вычислительных машин. - М.: Мир, 1975. - 544 с.
6. Хопгуд Ф. Методы компиляции. - М.: Мир, 1972.
7. Вайнгаартен Ф. Трансляция языков программирования. - М.: Мир, 1977. - 190 с.
8. .Кнут Д. Искусство программирования для ЭВМ.т.1 Основные алгоритмы. Перевод с английского. - Москва: Мир, 1976. - 735 с.
9. Мартин Дж. Организация баз данных в вычислительных системах: Перевод с английского. - Москва: Мир, 1980. - 664 с.
10. Зиглер К. Методы проектирования программных систем: Перевод с английского. - Москва: Мир, 1985. - 32 с. ил.

Приложение

Рекомендуемые варианты курсовых работ

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Рекомендуемые варианты курсовой работы для первой группы потока | | | | | | | |
| № жур | № вар | Номера цифр кода варианта | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 |
| Значения цифр кода варианта | | | | | |
| 1 | 1 | 2 | 5 | 3 | 1 | 4 | 2 |
| 2 | 2 | 3 | 6 | 4 | 2 | 5 | 3 |
| 3 | 3 | 4 | 1 | 1 | 1 | 1 | 1 |
| 4 | 4 | 1 | 2 | 2 | 2 | 2 | 2 |
| 5 | 5 | 2 | 3 | 3 | 1 | 3 | 3 |
| 6 | 6 | 3 | 4 | 4 | 2 | 4 | 1 |
| 7 | 7 | 4 | 5 | 1 | 1 | 5 | 2 |
| 8 | 8 | 1 | 6 | 2 | 2 | 1 | 3 |
| 9 | 9 | 2 | 1 | 3 | 1 | 2 | 1 |
| 10 | 10 | 3 | 2 | 4 | 2 | 3 | 2 |
| 11 | 11 | 4 | 3 | 1 | 1 | 4 | 3 |
| 12 | 12 | 1 | 4 | 2 | 2 | 5 | 1 |
| 13 | 13 | 2 | 5 | 3 | 1 | 1 | 2 |
| 14 | 14 | 3 | 6 | 4 | 2 | 2 | 3 |
| 15 | 15 | 4 | 1 | 1 | 1 | 3 | 1 |
| 16 | 16 | 1 | 2 | 2 | 2 | 4 | 2 |
| 17 | 17 | 2 | 3 | 3 | 1 | 5 | 3 |
| 18 | 18 | 3 | 4 | 4 | 2 | 1 | 1 |
| 19 | 19 | 4 | 5 | 1 | 1 | 2 | 2 |
| 20 | 20 | 1 | 6 | 2 | 2 | 3 | 3 |
| 21 | 21 | 2 | 1 | 3 | 1 | 4 | 1 |
| 22 | 22 | 3 | 2 | 4 | 2 | 5 | 2 |
| 23 | 23 | 4 | 3 | 1 | 1 | 1 | 3 |
| 24 | 24 | 1 | 4 | 2 | 2 | 2 | 1 |
| 25 | 25 | 2 | 5 | 3 | 1 | 3 | 2 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Рекомендуемые варианты курсовой работы для второй группы потока | | | | | | | |
| № жур | № вар | Номера цифр кода варианта | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 |
| Значения цифр кода варианта | | | | | |
| 1 | 26 | 2 | 3 | 3 | 1 | 2 | 3 |
| 2 | 27 | 3 | 4 | 4 | 2 | 3 | 1 |
| 3 | 28 | 4 | 5 | 1 | 1 | 4 | 2 |
| 4 | 29 | 1 | 6 | 2 | 2 | 5 | 3 |
| 5 | 30 | 2 | 1 | 3 | 1 | 1 | 1 |
| 6 | 31 | 3 | 2 | 4 | 2 | 2 | 2 |
| 7 | 32 | 4 | 3 | 1 | 1 | 3 | 3 |
| 8 | 33 | 1 | 4 | 2 | 2 | 4 | 1 |
| 9 | 34 | 2 | 5 | 3 | 1 | 5 | 2 |
| 10 | 35 | 3 | 6 | 4 | 2 | 1 | 3 |
| 11 | 36 | 4 | 1 | 1 | 1 | 2 | 1 |
| 12 | 37 | 1 | 2 | 2 | 2 | 3 | 2 |
| 13 | 38 | 2 | 3 | 3 | 1 | 4 | 3 |
| 14 | 39 | 3 | 4 | 4 | 2 | 5 | 1 |
| 15 | 40 | 4 | 5 | 1 | 1 | 1 | 2 |
| 16 | 41 | 1 | 6 | 2 | 2 | 2 | 3 |
| 17 | 42 | 2 | 1 | 3 | 1 | 3 | 1 |
| 18 | 43 | 3 | 2 | 4 | 2 | 4 | 2 |
| 19 | 44 | 4 | 3 | 1 | 1 | 5 | 3 |
| 20 | 45 | 1 | 4 | 2 | 2 | 1 | 1 |
| 21 | 46 | 2 | 5 | 3 | 1 | 2 | 2 |
| 22 | 47 | 3 | 6 | 4 | 2 | 3 | 3 |
| 23 | 48 | 4 | 1 | 1 | 1 | 4 | 1 |
| 24 | 49 | 1 | 2 | 2 | 2 | 1 | 2 |
| 25 | 50 | 2 | 3 | 3 | 1 | 2 | 3 |