Федеральное агентство по образованию РФ

ГОУ ВПО Пермский государственный технический университет

кафедра ИТАС

КУЗНЕЦОВ Д.Б.

КОНСПЕКТ ЛЕКЦИЙ ПО ДИСЦИПЛИНЕ Теории языков программирования и методы трансляции

Для студентов направления

«Информатика и вычислительная техника» (230100)

Специальности АСОИУ, ЭВТ, ПОВТ

дневной, заочной и дистанционной форм обучения

2008

Составители: Д.Б. Кузнецов

Конспект лекций по дисциплине «Теории языков программирования и методы трансляции»: для студентов направления «Информатика и вычислительная техника» (230100), специальностей АСОИУ, ЭВТ, ПОВТ дневной, заочной и дистанционной форм обучения / ПГТУ. – Пермь: 2008.

утверждено на заседании кафедры ИТАС 6 февраля 2008г. протокол №20/ ПГТУ. – Пермь: 2008.

Оглавление

Формальные грамматики

$G = <V_N, V_T, P , S >$
V_N -множество нетерминальных символов;
V_T - множество терминальных символов ;
P - множество правил вывода;
$S \in V_N$ - начальный нетерминальный символ.

Пример грамматик :
Метаязык
::= есть по определению
| или (исключающее)
[] необязательные символы
, перечисление
<нетерминальный символ>

<SELECT>::= select <IDS> from <IDS>
<IDS>::= <WRD> | <WRD> , <IDS> ; рекурсивное определение индефикаторов
<WRD>::=<LET>|<LET><WRD>
<LET>::= a|b|c...|z

Разберем строку:
for (i=0; i<10; i++)
< FOR >::=for(< инициализация >;< условие >;< изменение >)
< инициализация >::=< переменная >=< число >
< условие >::=< переменная >< знак сравнения >< число >
< изменение >::=< переменная >++ | < переменная >--
< переменная >::=< буква > | < буква >< переменная >
< буква >::=a|b|...|z
< число >::=< цифра > | < цифра >< число >
< цифра >::=1|2|...|9|0
< знак сравнения >::=< | >

Формальные языки

Порождение непосредственное
Порождение со звездой

Язык – множество $\{X \in V^*_T\}$, цепочек терминальных символов, таких, что они получаются из начальных нетерминальных символов.
$L(G) = \{ X \in V^*_T \mid S =>^* x\}$

Классификация языков по Хомскому

Язык – множество $\{X \in V^*_T\}$, цепочек терминальных символов, таких, что они получаются из начальных нетерминальных символов.
$L(G) = \{ X \in V^*_T \mid S =>^* x\}$
Классификация основывается по типу правил. Если в одном языке правила можно отнести к разным типам, то выбирается худший тип.

Типы:

0-тип: Не накладывает ограничения на правила, поэтому и не рассматривается.
1-тип: Контекстно-зависимые грамматики.
Правила имеют следующую форму:
$v\alpha w ::= v\beta w$
$v, w \in V^*$ – цепочки любых символов(контекст)
$\alpha \in V_N$
$\beta \in V^*$
Пример:

S->A	|	
[A]-> [C]	|	
(A)->(B)	| =>	S=>(A)=>(B) =>(x)
B->x	|	
C->y	|	

2-тип: Контекстно-свободные грамматики.
Правила имеют следующую форму:
$\alpha ::= \beta$
$\alpha \in V_N$
$\beta \in V^*$

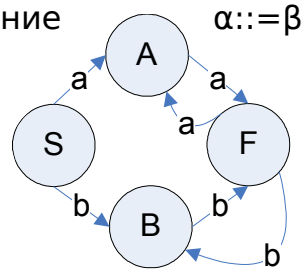
A->BcD
D->cA

3-тип: Автоматические грамматики.

- Левосторонние $\alpha ::= w\beta$ $\alpha ::= w$ $\alpha, \beta \in V_N$ $w \in V_T$
- Правосторонние $\alpha ::= \beta w$ $\alpha ::= w$

aabbbbaaaabb┘

1. S->bB



2. S->aA
3. A->aF
4. B->bF
5. F->aA
6. F->bB
7. F->┘

Распознающие автоматы – автоматы Мура с множеством выделенных состояний - конечных. Этот автомат недетерминированный и частичный.

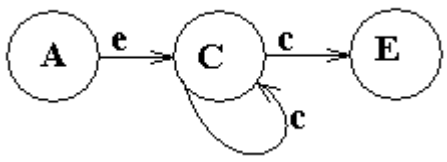
	S	A	B	F
q	A	F		A
b	B		F	B

====> переход к полному автомату

	S	A	B	F	Err
a	A	F	(E)	A	(E)
b	B	(E)	F	B	(E)

< нетерм. символ > → [< нетерм. символ >]< терм. символ >
A→Bc
B→Cc
C→d
< нетерм. символ > → < терм. символ >[< нетерм. символ >]
A→eC
C→cC
C→c
Зададим последнюю с помощью автомата

	A	C	E
a	-	C,E	-
e	C	-	-



Приведем к детерминированному

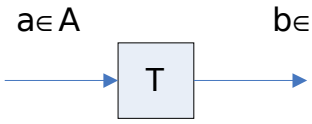
	A	C	E	{-}	{C,E}
c	{-}	{C,E}	{-}	{-}	{C,E}
e	C	{-}	{-}	{-}	{-}

Переход от праволинейной грамматики к автоматам

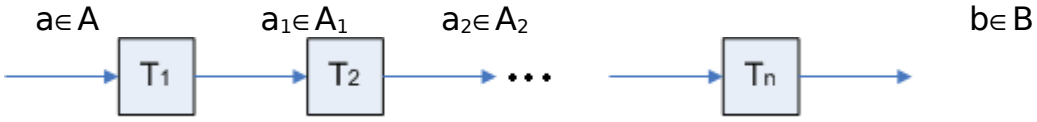
S->select ==> S->s<elect>
<elect> -> e<lect>
<lect> -> l<ect>
<ect> -> e<ct>
<ct> -> c<t>
<t> -> t

Трансляторы

Трансляторы - программа или устройство, переводящее входную строку из одного языка в другой без потери списка.



Для упрощения процесс транслирования разбивают на шаги

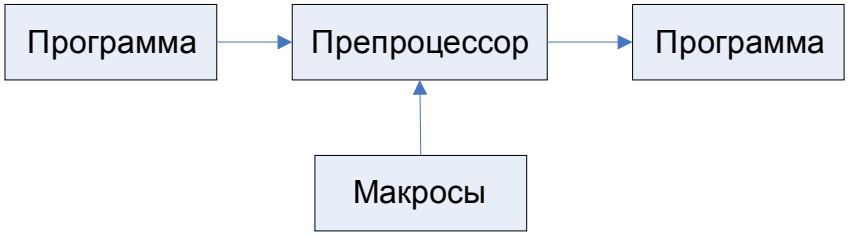


Виды трансляторов:

- Интерпретаторы – перевод из одного языка в другой по шагам.
- Компиляторы - переводит целиком.

Претрансляция

-текстовая замена макроопределений.



define include не библиотеки

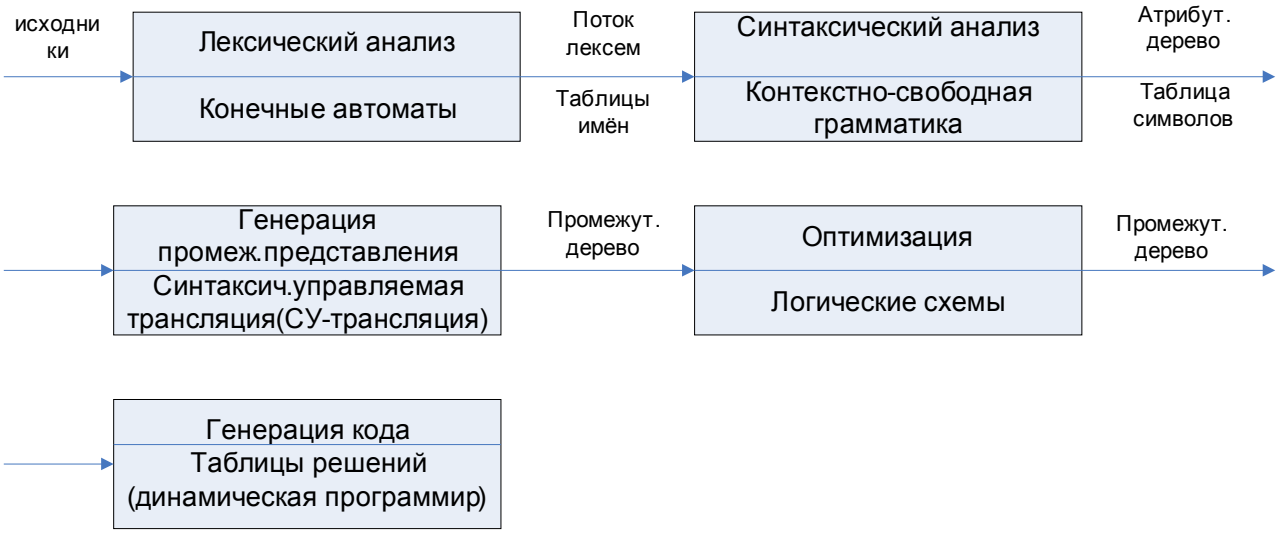
Структура компилятора

Функции компиляторов.

1. Лексический анализ – выделяет лексемы в строчке и проверяет на правильность.
2. Синтаксический анализ – проверяет порядок лексем.
3. Семантический анализ- проверка на правильность присваивания.
4. Генерация промежуточного представления
5. Оптимизация

6. Генерация выходного текста

Схема работы компилятора



Лексический анализ

Функции лексического анализа:
1. Выделения численных констант
2. Выделение индификаторов
3. Выделение сложных символов: /* */ //
4. Определение ошибок ввода

строковые константы...?

Для лексического анализа используются автоматные (регулярные) грамматики. В средствах лексического анализа используются регулярные выражения.

Регулярные выражения

- это язык, описывающий шаблоны строк (дополнительную информацию искать в man 7 regex).

Существуют 2 типа регулярных выражений:
1) стандарта POSIX
2) принятый в языке Perl

- Одиночный символ, кроме / [\ { + * - $? ^
  $ grep “a” f.txt ищет строки, содержащие “a”

Для использования метасимволов их экранируют.
$ grep “ a\[i\] ” f.txt

- Мнимые символы
  ^ начало строки
  $ конец строки
  $ grep “ ^ups$ ” f.txt ищет строку, состоящую из слова “ups”
- Любой символ кроме конца и начала строки обозначается точкой (.).
  $ grep ‘ab.d’ f1.txt
- Классы символов []
  [abc] любой из символов 'a', 'b' и 'c'
  [a-zA-z] любая английская буква
- Альтернатива
  $ grep “ p|P” f.txt ищем строку, где встречается p или P
- Группировка ()
  ‘(hi)|(hello)’
- Квантификатор- показывает сколько раз символ должен повторяться
  + один и более раз
  * ноль и более раз
  ? 1 или 0 раз
  {n} n раз
  {n,} n раз и более
  {n,m} от n до m раз

Пример.
‘[a-zA-z]+@([a-z]+\ .)+[a-z]+’ шаблон на e-mail

Регулярные выражения используются в конфигурационных файлах, базах данных, командах grep, sed, awk, lex и т.д.

lex

Lex - команда Unix. Предназначена для ганерации программы на Си, которая будет проводить лексический анализ входного текста в соответствии с правилами.

Формат файла на lex:
раздел деклараций : имя значение.
%%
раздел правил : шаблон(регулярное выражение) действие
%%
Код на Си

Раздел деклараций: декларация регулярное_выражение
Раздел правил: регулярное_выражение { код на Си }
получается фция yylex – считывает со стонадартного входа текст, сопоставляет с ним шаблоны, при совпадении выполняет соответствующие действия. Если ни один шаблон не совпадает – вызывает обработчик ошибок - yyerror.

Для обработки ошибок описываем функцию
yyerror()

{ printf (“Ошибка”); }

Подсчёт идентификаторов во входном потоке

```
################ p.l ################
digit [0-9]
letter [a-z]
%%
{letter}({letter}|{digit})*  { i++; }
%%
int i;
main()
{
   yylex();
   printf( “ %d", i );
}
yyerror (){}
#########################################
```

yylex() - из раздела правил преобразует в Си

В командной строке пишем :

```
$ flex -oprogname.c progname.l
$ cc -o progname progname.c -lfl
$ ./progname [ < filename ]
```

Синтаксический анализ

Выделяют 2 типа синтаксического анализа: сверху вниз и снизу вверх. Используются контекстно-свободные грамматики.

< нетерм. символ > → < цепочка символов >

A→bC

C→eA

A→dCCA

A→b

c→E

Слева должен быть нетерм. символ – без контекста

Пример:

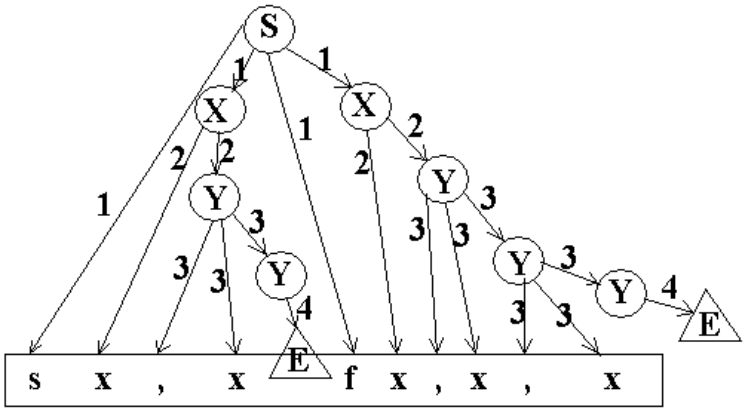
select name1, name2 from tab1, tab2

S x,x,…x f x,x,…,x

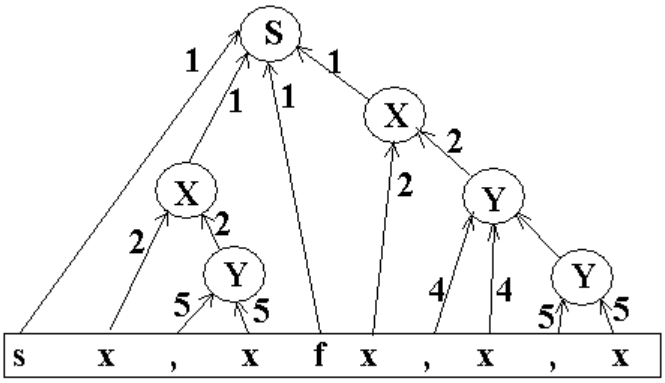
Анализ сверху вниз

1) S→sXfX
2) X→xY
3) Y→,xY
4) Y→E

s x , x f x , x , x



Анализ снизу вверх
1) S→sXfX
2) X→xY
3) X→x
4) Y→,xY
5) Y→,x



S и q грамматики

-Контекстно-свободные грамматики , на которые наложены ряд ограничений :

правые части правил начинаются с терминальных символов, причем для одного и того же левого символа правые части начинаются с разных символов.

S -> aT
S -> TbS
T -> bT
=
T -> bS
=

q-грамматика отличается от s-грамматики наличием аннулирующего правила (в правой части есть пустой символ)
=> ε
Из-за аннулирующих правил для q-грамматики вводится понятие следующего символа. N(A) - множество терминальных следующих (Next) за A символов.
В данном случае за A могут следовать a или b - {a,b}.

LL(1)- грамматика

Правая часть может начинаться с нетерминального символа, но при этом должна сохраняться детерминальность в выборе правил, т.е. множества выборов из правил при одном и том же нетерминальном символе не должны пересекаться.

LL(1)=Left-left most- самый самый левый.

F(α) - множество терминальных символов, стоящих первыми (First) в цепочках, выводимых из строки α.

N(A) - множество терминальных символов, следующих (Next) в цепочках за данным
нетерминальным символом A.

Множество выбора для каждого правила формируется с учетом множества первых и
множества следующих символов.

1. S -> AbB	E(1) = F(AbB) = {a, b, c, e}
2. S -> d	E(2) = {d}
3. A -> CAb	E(3) = F(CAb) = {a, e}
4. A -> B	E(4) = F(B) N(A) = {c}U{b} = {c}U{b}
5. B -> cSd	E(5) = F(cSd) = {c}
6. B -> ε	E(6) = F(ε) U N(B) = { ε }U {b, d, ┤}
7. C -> a	E(7) = {a}
8. C -> ed	E(8) = {e}

Автоматы с магазинной памятью

Понятие автомата
Понятие стека
Пример работы со стеком
МП-автомат скобки матрешка

МП-автомат скобки

Правила МПА для LL

что по осям

клетки

aBS

BC

Э

a

МП-автоматы (автоматы с магазинной памятью) – в стек помещается начальный нетерминальный символ

	x	Δ
(	↓x	↓x
)	↑x	-
┤	-	+

Преобразуем выражение (()())

(	(	)	(	)	)	┤
Δ	x	x	x	x	x	x
	Δ	x	Δ	x	Δ	
		Δ		Δ		

Метод рекурсивного спуска

LL(1) грамматики распознаются с помощью метода рекурсивного спуска.

Пусть дана грамматика:

1. I -> LP {ab)
2. P -> LP {ab}
3. P -> DP {1,2}
4. P -> E N(P)ↄ
5. L -> a|b
6. D -> 1|2

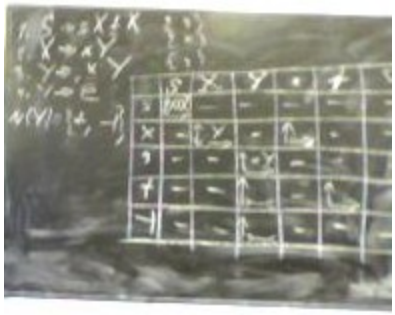
Метод рекурсивного спуска позволяет писать программы синтаксического анализа на языке, допускающем рекурсию, прямо по грамматическим правилам.

Метод рекурсивного спуска:

```
###########
int c;
main()
{I();printf(“+”);}
I()
{c=getchar();
    Switch(c)    {
            case 'a':
            case 'b': L();P(); break;
            default printf (“-”); exit(0);
                    }
}
L() {}
D() {}
P()
{ c = getchar();
    Switch(c)    {
            case 'a':
            case 'b': L();P(); break;
            case '1':
            case '2': D();P(); break;
            case '\n' : break;
            default printf (“-”); exit(1);
                    }
}
```

Метод рекурсивного спуска по МП-автомату

ф-ции – магазинные символы



```
#include <stdio.h>

char str[256];
int i = 0;

main()
{
 scanf( "%s", str );
 S(); stack_end();
}

stack_end()
{
 switch( str[i] )
 {
  case '\0': exit(0);
  default  : printf("error near %c", str[i]); exit(1);
 }
}

S()
{
 switch( str[i] )
 {
  case 's': i++; X(); f(); X(); break;
  default  : printf("error near %c", str[i]); exit(1);
 }
}

X()
{
 switch( str[i] )
 {
  case 'x': i++; Y(); break;
  default  : printf("error near %c", str[i]); exit(1);
 }
}

Y()
{
 switch( str[i] )
 {
  case ',': i++; x(); Y(); break;
  case '\0': break;
  case 'f': break;
  default  : printf("error near %c", str[i]); exit(1);
 }
}
```

```
x()
{
 switch( str[i] )
 {
  case 'x': i++; break;
  default  : printf("error near %c", str[i]); exit(1);
 }
}

f()
{
 switch( str[i] )
 {
  case 'f': i++; break;
  default  : printf("error near %c", str[i]); exit(1);
 }
}
```

LR-грамматика

Left-Right most- самый правые части для самых левых нетерминальных символов.

Грамматика с предшествованием

Между рядом симв
свертка – цепочка символов порождаемых одним правилом
Правила расстановки отношений между символами грамматики :

1. Если S_i и S_j входят в одну свёртку, то они равны $S_i = S_j$
   $\underline{\ldots S_i\ S_j \ldots}$
   Свёртка –правая часть правила.
2. Если S_j в правой части свёртки, то $S_i < S_j$
   $S_i\ \underline{S_j \ldots}$
3. $\underline{\ldots S_i}\ S_j \Rightarrow S_i > S_j$
4. $\underline{\ldots S_i}\ \underline{S_j \ldots} \Rightarrow S_i > S_j$

Все отношения не являются симметричными.

1.	S -> sXfX	s ≐ X X ≐ f f ≐ X
2.	X -> x	s < x x > f f < x
3.	X -> xY	x ≐ Y Y > f
4.	Y -> ,x	x < , , ≐ x
5.	Y -> ,xY	

В грамматике с предшествованием между двумя одинаковыми символами не стоят разные отношения.
├ s x , x f x ┤ расставим знаки в цепочке символов.

$$=> \vdash < s < x < , \doteq x > f < x > \dashv =>$$
$$\underline{Y} \qquad \underline{X}$$
$$=> \vdash < s < x < (, \doteq x) > f (< x >) \dashv =>$$
$$\underline{X}$$
$$=> \vdash < s < (x \doteq Y) > f \doteq X > \dashv => \vdash < s \doteq X \doteq f \doteq X > \dashv = > \vdash < S > \dashv$$

Функции предшествования

Этот интересный метод придумал Р.Флойд – автор многих остроумных решений в программировании. Вместо матрицы строятся две специальные функции f и g , такие что:

1. Если $S_i \cdot{*}> S_j \Rightarrow f(S_i) > g(S_j)$.
2. Если $S_i <{*} S_j \Rightarrow f(S_i) < g(S_j)$.
3. Если $S_i ={*} S_j \Rightarrow f(S_i) = g(S_j)$.

Тогда, вместо поиска с помощью матрицы отношения предшествования между символами, просто происходит сравнение числовых значений соответствующих функций на больше меньше равно.

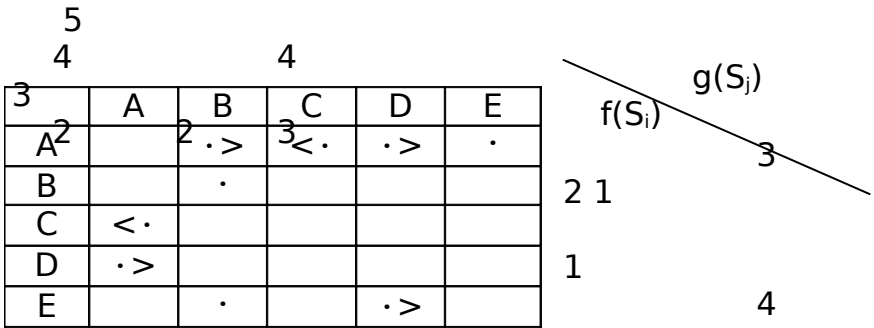
Построение функций предшествования:

0. Строится матрица предшествования и начальные значения функций принимаются равными единице: $f(S_i) = g(S_j) = 1$.
1. Матрица просматривается по строкам в поисках отношений $\cdot{*}>$ и, если $f(S_i) > g(S_j)$, то идем дальше, если же $S_i {*}> S_j$, а $f(S_i) \le g(S_j)$, то увеличиваем значение $f(S_i)$ - $f(S_i) = g(S_j) + 1$.
2. Матрица просматривается по столбцам в поисках отношений $<{*}\cdot$ и, если $f(S_i) < g(S_j)$, то идем дальше, если же $S_i <{*} S_j$, а $f(S_i) \ge g(S_j)$, то увеличиваем значение $g(S_j)$ - $g(S_j) = f(S_i) + 1$.
3. Матрица просматривается в поисках отношений $={*}$ и, если $f(S_i) = g(S_j)$, то идем дальше, если $S_i ={*} S_j$, а $f(S_i) \ne g(S_j)$, то выравниваем значения функций путем увеличения меньшего из значений до большего - $f(S_i) = g(S_j) = \max[f(S_i), g(S_j)]$.
4. Возвращение к первому пункту.
   Повторять до тех пор, пока рост функций не прекратится (или когда значение одной из функций не превысит 2*n, где n - размерность матрицы - в этом случае алгоритм не сходится).

Пример.

На основе матрицы предшествования в соответствии с описанным алгоритмом построим функции предшествования.

Уточняемые значения функций будем располагать левее строк и выше столбцов с соответствующими символами.



	A	B	C	D	E
A		·>	<·	·>	·
B		·			
C	<·				
D	·>				
E		·		·>	

В результате получим числовые значения (табличных) функций для всех символов.

	A	B	C	D	E
f	3	2	4	6	2
g	5	2	4	1	3

Однако, этот метод не свободен от недостатков:

1. Алгоритм не всегда сходится (не всегда приводит.к построению функций).
2. При переходе к функциям происходит «незаконное доопределение» матрицы. То есть как бы появляются отношения предшествования между парами символов, для которых в исходной матрице отношение отсутствовало.

YACC

Yet **A**nother **C**ompiler of **C**ompilers

Предназначен для генерации программы на Си, которая бы проводила синтаксический разбор входной информации. Грамматика может быть неоднозначна, поэтому для преодоления этого нужно использовать правила предшествования.

Имеет следующую структуру:

Раздел деклараций
%%
Раздел правил
%%
Пользовательский код

Раздел деклараций
%{
Пользовательский код
}%
%token лексем

Раздел правил
Интерпретирует правила типа:
Нетерм. символ : цепочка символов {код на Си}
|цепочка символов{код на Си};

Пример. Рассмотрим предыдущий пример.

s x, x f x

Программа будет состоять из двух частей: синтаксический разбор (на yacc) и лексический на lex).

В лексическом разборе мы будем определять буквы x, y, z и запятую. И передавать в yacc программу найденное.

1. На YACC

```
##########prim1.y################
%token S X F Z
%%
es:        S iks F iks    {printf(“OK!”);}          //первое правило
    ;
iks:       X
    | X igrek
    ;
igrek:     Z X
    | Z X igrek
    ;
%%
yyerror() { printf(" Ошибка! "); }
main()  {
yyparse(); }                          //функция, которая получается из раздела правил
#####################################
```

2. На Lex

```
########prim1.l#################
%{
#include "prim1_y.h"        //там лексемы будут определены как макросы
%}
%%
s           { return( S ); }
x           { return( X ); }
f           { return( F ); }
[,]         { return( Z ); }
.           { return( yytext[0] ); }
%%
############################
```

Синтаксический разбор будет выглядеть так:

```
$ yacc -d  -o     prim1_y.c    prim1.y
 # по -d создается prim1_y.h, в котором описываются макросы X Y Z P
$ lex –o      prim1_l.c     prim1.l
$ cc -o      prim1     prim1_y.c     prim1_l.c
```

Семантический анализ

Используется для проверки типов данных при операциях и области действия переменных.

Задачей контекстного анализа является установление свойств объектов и их использования. Наиболее часто решаемой задачей является определение существования объекта и соответствия его использования контексту, что осуществляется с помощью анализа типа объекта. Под контекстом здесь понимается вся совокупность свойств текущей точки программы, например множество доступных объектов, тип выражения и т.д.

Таким образом, необходимо хранить объекты и их типы, уметь находить эти объекты и определять их типы, определять характеристики контекста. Совокупность доступных в данной точке объектов будем называть средой. Обычно среда программы состоит из частично упорядоченного набора компонент

Компоненты образуют дерево, соответствующее этому частичному порядку. Частичный порядок между компонентами обычно определяется статической вложенностью компонент в программе. Эта вложенность может соответствовать блокам, процедурам или классам программы. Компоненты среды могут быть именованы. Поиск в среде обычно ведется с учетом упорядоченности компонент. Среда может включать в себя как компоненты, полученные при трансляции «текущего» текста программы, так и «внешние» (например, раздельно компилированные) компоненты.

Для обозначения участков программы, в которых доступны те или иные описания, используются понятия области действия и области видимости. Областью действия описания является процедура (блок), содержащая описание, со всеми входящими в нее (подчиненными по дереву) процедурами (блоками). Областью видимости описания называется часть области действия, из которой исключены те подобласти, в которых по тем или иным причинам описание недоступно, например, оно перекрыто другим описанием. В разных языках понятия области действия и области видимости уточняются по-разному.

Обычными операциями при работе со средой являются:

- включить объект в компоненту среды;

- найти объект в среде и получить доступ к его описанию;

- образовать в среде новую компоненту, определенным образом связанную с остальными;

- удалить компоненту из среды.

Среда состоит из отдельных объектов, реализуемых как записи (в дальнейшем описании мы будем использовать имя TElement для имени типа этой записи). Состав полей записи, вообще говоря, зависит от описываемого объекта (тип, переменная и т.д.), но есть поля, входящие в запись для любого объекта:

TObject Object - категория объекта (тип, переменная, процедура и т.д.);

TMode Mode - вид объекта: целый, массив, запись и т.д.;

TName Name - имя объекта;

TType Type - указатель на описание типа.

Атрибутная грамматика

Атрибутная грамматика – это четверка $G = <V_N, V_T, P, S>$, в которой

V_N - ***нетерминальный словарь*** (множество нетерминальных символов);

V_T - ***терминальный словарь*** (множество терминальных символов) ;

P - ***множество грамматических правил***;

$S \in V_N$ - ***начальный нетерминальный символ***.

A(x) множество атрибутных символов

Атрибут определяется для каждого правила, входящего в грамматику .

$a_0<i_0>= fpa_0<l_0> fpa_1<l_1>... a_j<i_j>$

$a_k<i_k>$ атрибут x_i

Атрибуты делятся на 2 вида :

1.Синтезируемые атрибуты

a<0>= fpa<0>(...)

2.Наследуемые атрибуты

$x_0 \rightarrow x_1 ... x_i ... x_{np}$

a<i>= fpa<i>(...)

Таким образом, атрибутная грамматика :

$АГ = <Г, A_s, A_i, R>$

Г- контекстная свободная грамматика

A_s – множество синтезируемых атрибутов

A_i – множество наследуемых атрибутов

R- семантические правила

Пример : Число из 2сс перевести в 10сс(систему исчисления) 1011

Грамматика

$P \rightarrow S$

$S \rightarrow B$

$S^1 \rightarrow BS^2$

$B \rightarrow 1|0$

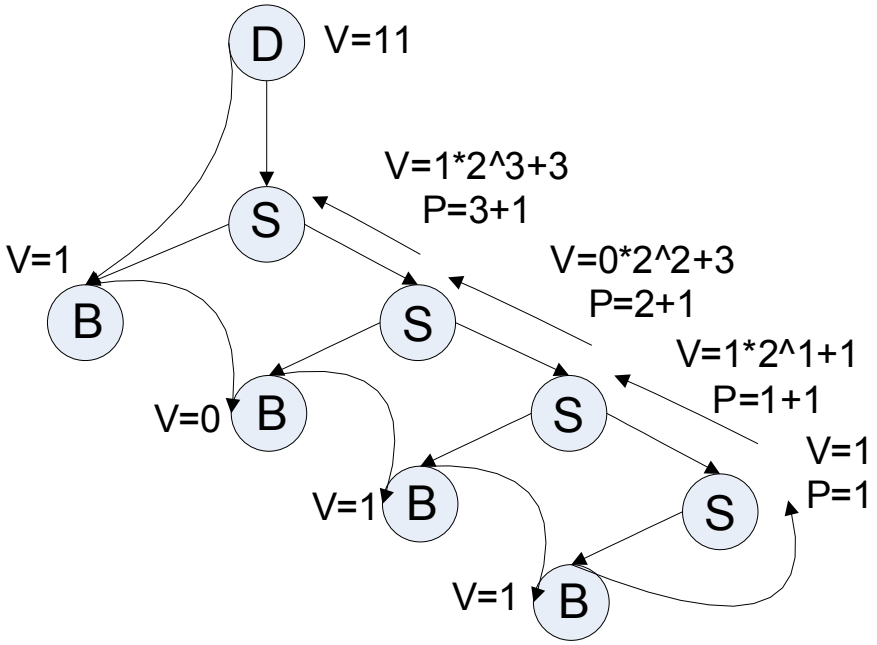
$V(D)=V(S)$

$V(S)=V(B)*2^0$; $p(S)=1$

$V(S^1)=V(B)*2^{p(S\hat{}2)}+V(S^2)$; $p(S^1)=P(S^2)+1$

Значения атрибутов терминальных символов – это константы, т.е. они определены ,но нет семантичных правил.

Дерево разбора обходится сверху вниз слева направо.



Система семантического анализа атрибутных грамматик с использованием грамматик.

ELEGANT, FGC-2,OX,RIE или можно использовать Lex иYacc.

B2d.l

```
%{
#include <stdlib.h>
# include <b2d_y.h>
%}
bit [0-9]
%%
{bit}      {yylval.val=atoi(yytext);  /*формирование числового значения,
yylval- передаваемый в  yacc значение */
    Return(BIT);}
.   {return(yytext[0]);} /* . любой символ*/
```

##############

B2d.y

```
%{
#include <math.h>
long double p=0;
%} //  для передачи используется union
%union {
int val;
}
%token <val> BIT // лексемы, которые будут обмениваться Lex и Yacc
% type <val> str
%%
dec:       . str {printf(“%d”,$1);}
    ;
```

```
str:        BIT {$$=$<val>1;}
    |BIT str {$$=$ <val>1*pow(2L,p+$1); p++;}
    ;
%%
Yyerror() {}
Main()    {yyparse()}

###############

$ yacc –d –o bed_y.c b2d.y
$ lex –o b2d_l.c b2d..l
$ cc –o b2d  b2d_l.c  b2d_y.c –lfl –lm
-----
$ b2d <enter>
 1011 <enter>
```

Генерация промежуточного представления

Преобразователи с магазинной памятью

Рассмотрим важный класс абстрактных устройств, называемых преобразователями с магазинной памятью. Эти преобразователи получаются из автоматов с магазинной памятью, если к ним добавить выход и позволить на каждом шаге выдавать выходную цепочку.

Синтаксически управляемый перевод

Другим формализмом, используемым для определения переводов, является схема синтаксически управляемого перевода. Фактически, такая схема представляет собой КС-грамматику, в которой к каждому правилу добавлен элемент перевода. Всякий раз, когда правило участвует в выводе входной цепочки, с помощью элемента перевода вычисляется часть выходной цепочки, соответствующая части входной цепочки, порожденной этим правилом.

Схемы синтаксически управляемого перевода

Определение. Схемой синтаксически управляемого перевода (или трансляции, сокращенно: СУ-схемой) называется пятерка $Tr = <V_N, V_T, П, R, S>$, где
конечное множество нетерминальных символов;
конечный входной алфавит;
конечный выходной алфавит;
конечное множество правил перевода вида
A -> u,v
где u - V*, v - $(V_N, П)$*
всем нетерминалам из v должны соответствовать терминалы из u.
A -> bcDeF, FxyDz
начальный символ, выделенный нетерминал из V_N

порождение:

<xAy, x'Ay'> => <xuy, x'vy'>

in

```
x=y+y+y+y;

out
z=0;
z+=y;
z+=y;
z+=y;
z+=y;
x=z;
```

```
S-> «x=Y» , «z=0; Y»
Y-> «yP» , «z+=y; P»
P-> «+yP» , «z+=y; P»
P-> «;» , «x=z;»
```

<S,S> -> <x=Y , z=0; Y>

вхождения нетерминалов в цепочку v образуют перестановку вхождений нетерминалов в цепочку u, так что каждому вхождению нетерминала B в цепочку u соответствует некоторое вхождение этого же нетерминала в цепочку v; если нетерминал B встречается более одного раза, для указания соответствия используются верхние целочисленные индексы;

Определим выводимую пару в схеме Tr следующим образом:
(S, S) - выводимая пара, в которой символы S соответствуют друг другу;
если (xAy, x'Ay') - выводимая пара, в цепочках которой вхождения A соответствуют друг другу, и A u, v - правило из R, то (xuy, x'vy') - выводимая пара. Для обозначения такого вывода одной пары из другой будем пользоваться обозначением : (xAy, x'Ay') (xuy, x'vy'). Рефлексивно-транзитивное замыкание отношение обозначим *.
Переводом (Tr), определяемым СУ-схемой Tr, назовем множество пар
Если через P обозначить множество входных правил вывода всех правил перевода, то G = (N, T, P, S) будет входной грамматикой для Tr.
СУ-схема Tr = (N, T, , R, S) называется простой, если для каждого правила A u, v из R соответствующие друг другу вхождения нетерминалов встречаются в u и v в одном и том же порядке.

Перевод, определяемый простой СУ-схемой, называется простым синтаксически управляемым переводом (простым СУ-переводом).

Пример 5.2. Перевод арифметических выражений в ПОЛИЗ (польскую инверсную запись) можно осуществить простой СУ-схемой с правилами

E E + T,
ET+
E T,
T
T T * F,
TF+
T F,
F
F id,
id
F (E),
E

Найдем выход схемы для входа id*(id+id). Нетрудно видеть, что существует последовательность шагов вывода
(E, E) (T, T) (T *F, TF*) (F *F, FF*) (id*F, id F*) (id* (E), id E*) (id * (E + T), id E T + *) (id * (T + T), id T T + *) (id * (F + T), id F T + *) (id * (id + T), id id T + *) (id * (id + F), id id F + *) (id * (id + id), id id id + *),
переводящая эту цепочку в цепочку id id id + *.

Обобщенные схемы синтаксически управляемого перевода

```
in
x=y+y+y+y+y+y;

out1
z=0;
z+=y;
z+=y;
z+=y;
z+=y;
x=z;

out2
z=y+y;
x=z;
```

```
z=y+y;
x=z;
z=x+y;
x=z;
```

S->Y

В процессе трансляции компилятор часто используют промежуточное представление (ПП) исходной программы, предназначенное прежде всего для удобства генерации кода и/или проведения различных оптимизаций. Сама форма ПП зависит от целей его использования.

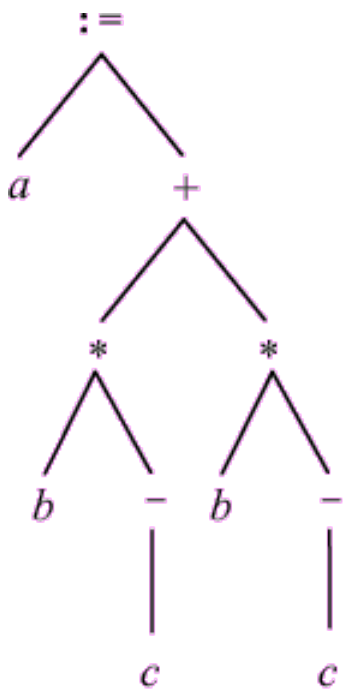
Наиболее часто используемыми формами ПП является ориентированный граф (в частности, абстрактное синтаксическое дерево, в том числе атрибутированное), трехадресный код (в виде троек или четверок), префиксная и постфиксная запись.

Представление в виде ориентированного графа

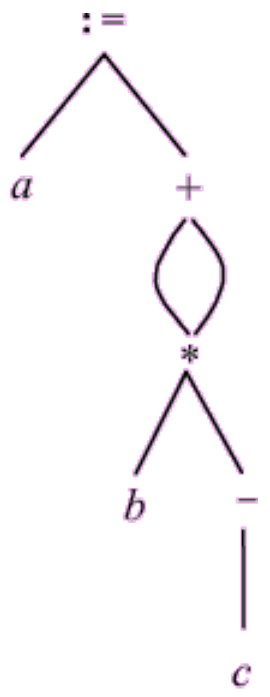
Простейшей формой промежуточного представления является синтаксическое дерево программы. Ту же самую информацию о входной программе, но в более компактной форме дает ориентированный ациклический граф (ОАГ), в котором в одну вершину объединены вершины синтаксического дерева, представляющие общие подвыражения. Синтаксическое дерево и ОАГ для оператора присваивания

a := b *-c + b *-c

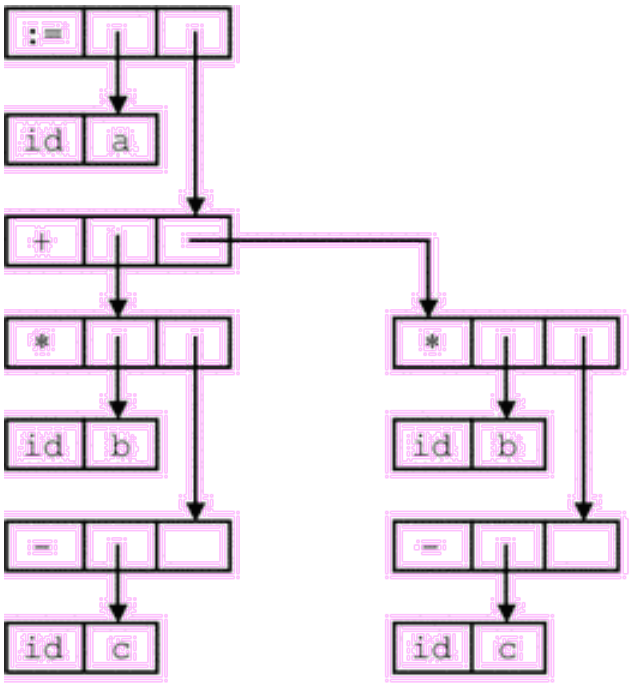
приведены на рис. 8.1.



а



б



а

0	id	b	
1	id	c	
2	–	1	
3	*	0	2
4	id	b	
5	id	c	
6	–	5	
7	*	4	6
8	+	3	8
9	id	a	
10	:=	9	8

б

На рис. 8.2 приведены два представления в памяти синтаксического дерева на рис. 8.1, а. Каждая вершина кодируется записью с полем для операции и полями для указателей на потомков. На рис. 8.2, б, вершины размещены в массиве записей и индекс (или вход) вершины служит указателем на нее.

Трехадресный код

Трехадресный код - это последовательность операторов вида x := y op z, где x, y и z - имена, константы или сгенерированные компилятором временные объекты. Здесь op - двуместная операция, например операция плавающей или фиксированной арифметики, логическая или побитовая. В правую часть может входить только один знак операции.

Составные выражения должны быть разбиты на подвыражения, при этом могут появиться временные имена (переменные). Смысл термина «трехадресный код» в том, что каждый оператор обычно имеет три адреса: два для операндов и один для результата. Трехадресный код - это линеаризованное представление синтаксического дерева или ОАГ, в котором временные имена соответствуют внутренним вершинам дерева или графа. Например, выражение x+y*z может быть протранслировано в последовательность операторов

```
t1 := y * z
t2 := x + t1
```

где t1 и t2 - имена, сгенерированные компилятором.

В виде трехадресного кода представляются не только двуместные операции, входящие в выражения. В таком же виде представляются операторы управления программы и одноместные операции. В этом случае некоторые из компонент трехадресного кода могут не использоваться. Например, условный оператор

```
if A > B then S1 else S2
```

может быть представлен следующим кодом:

```
t := A - B
JGT t, S2
  ...
```

Здесь JGT - двуместная операция условного перехода, не вырабатывающая результата.

Разбиение арифметических выражений и операторов управления делает трехадресный код удобным при генерации машинного кода и оптимизации. Использование имен промежуточных значений, вычисляемых в программе, позволяет легко переупорядочивать трехадресный код.

```
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
 a := t5
```

а

```
t1 := -c
t2 := b * t1
t5 := t2 + t2
 a := t5
```

б

Представления синтаксического дерева и графа в виде трехадресного кода дано на а и б, соответственно.

Трехадресный код - это абстрактная форма промежуточного кода. В

реализации трехадресный код может быть представлен записями с полями для операции и операндов. Рассмотрим три способа реализации трехадресного кода: четверки, тройки и косвенные тройки.

Четверка - это запись с четырьмя полями, которые будем называть op, arg1, arg2 и result. Поле op содержит код операции. В операторах с унарными операциями типа x := -y или x := y поле arg2 не используется. В некоторых операциях (типа «передать параметр») могут не использоваться ни arg2, ни result. Условные и безусловные переходы помещают в result метку перехода. На рис. 8.4, а, приведены четверки для оператора присваивания a := b *-c + b *-c. Они получены из трехадресного кода на рис. 8.3, а.

	op	*arg1*	*arg2*	*result*
(0)	-	c		t1
(1)	*	b	t1	t2
(2)	-	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

а) четверки

	op	*arg1*	*arg2*
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

б) тройки

Обычно содержимое полей arg1, arg2 и result - это указатели на входы таблицы символов для имен, представляемых этими полями. Временные имена вносятся в таблицу символов по мере их генерации.

Чтобы избежать внесения новых имен в таблицу символов, на временное значение можно ссылаться, используя позицию вычисляющего его оператора. В этом случае трехадресные операторы могут быть представлены записями только с тремя полями: op, arg1 и arg2, как это показано на рис. 8.3, б. Поля arg1 и arg2 - это либо указатели на таблицу символов (для имен, определенных программистом, или констант), либо указатели на тройки (для временных значений). Такой способ представления трехадресного кода называют тройками. Тройки соответствуют представлению синтаксического дерева или ОАГ с помощью массива вершин.

	op	*arg1*	*arg2*
(0)	[]=	x	i
(1)	:=	(0)	y

а) $x[i] := y$

	op	*arg1*	*arg2*
(0)	=[]	y	i
(1)	:=	x	(0)

б) $x := y[i]$

	оператор
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	-	c	
(15)	*	b	(14)
(16)	-	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Числа в скобках - это указатели на тройки, а имена - это указатели на таблицу символов. На практике информация, необходимая для интерпретации различного типа входов в поля arg1 и arg2, кодируется в поле op или дополнительных полях. Тройки рис. 8.4, б, соответствуют четверкам рис. 8.4, а.

Для представления тройками трехместной операции типа x[i] := y требуется два входа, как это показано на рис. 8.5, а, представление x := y[i] двумя операциями показано на рис. 8.5, б.

Трехадресный код может быть представлен не списком троек, а списком указателей на них. Такая реализация обычно называется косвенными тройками. Например, тройки рис. 8.4, б, могут быть реализованы так, как это изображено на рис. 8.6.

При генерации объектного кода каждой переменной, как временной, так и определенной в исходной программе, назначается память периода исполнения, адрес которой обычно хранится в таблице генератора кода. При использовании четверок этот адрес легко получить через эту таблицу.

Более существенно преимущество четверок проявляется в оптимизирующих компиляторах, когда может возникнуть необходимость перемещать операторы. Если перемещается оператор, вычисляющий x, не требуется изменений в операторе, использующем x. В записи же тройками перемещение оператора, определяющего временное значение, требует изменения всех ссылок на этот оператор в массивах arg1 и arg2. Из-за этого тройки трудно использовать в оптимизирующих компиляторах.

В случае применения косвенных троек оператор может быть перемещен переупорядочиванием списка операторов. При этом не надо менять указатели на op, arg1 и arg2. Этим косвенные тройки похожи на четверки. Кроме того, эти два способа требуют примерно одинаковой памяти. Как и в случае простых троек, при использовании косвенных троек выделение памяти для временных значений может быть отложено на этап генерации кода. По сравнению с четверками при использование косвенных троек можно сэкономить память, если одно и то же временное значение используется более одного раза. Например, можно объединить строки (14) и (16), после чего можно объединить строки (15) и (17).

Линеаризованные представления

В качестве промежуточных представлений весьма распространены линеаризованные представления деревьев. Линеаризованное

представление позволяет относительно легко хранить промежуточное представление во внешней памяти и обрабатывать его. Наиболее распространенной формой линеаризованного представления является польская запись - префиксная (прямая) или постфиксная (обратная).

Постфиксная запись - это список вершин дерева, в котором каждая вершина следует (при обходе снизу-вверх слева-направо) непосредственно за своими потомками. Дерево в постфиксной записи может быть представлено следующим образом:

```
a b c - * b c - * + :=
```

В постфиксной записи вершины синтаксического дерева явно не присутствуют. Они могут быть восстановлены из порядка, в котором следуют вершины и из числа операндов соответствующих операций. Восстановление вершин аналогично вычислению выражения в постфиксной записи с использованием стека.

В префиксной записи сначала указывается операция, а затем ее операнды. Например, для приведенного выше выражения имеем

```
:= a + * b - c * b - c
```

Организация информации в генераторе кода

Синтаксическое дерево в чистом виде несет только информацию о структуре программы. На самом деле в процессе генерации кода требуется также информация о переменных (например, их адреса), процедурах (также адреса, уровни), метках и т.д. Для представления этой информации возможны различные решения. Наиболее распространены два:

- информация хранится в таблицах генератора кода;
- информация хранится в соответствующих вершинах дерева.

Рассмотрим, например, структуру таблиц, которые могут быть использованы в сочетании с Лидер-представлением. Поскольку Лидер-представление не содержит информации об адресах переменных, значит, эту информацию нужно формировать в процессе обработки объявлений и хранить в таблицах. Это касается и описаний массивов, записей и т.д. Кроме того, в таблицах также должна содержаться информация о процедурах (адреса, уровни, модули, в которых процедуры описаны, и т.д.).

При входе в процедуру в таблице уровней процедур заводится новый вход - указатель на таблицу описаний. При выходе указатель восстанавливается на старое значение. Если промежуточное представление - дерево, то информация может храниться в вершинах самого дерева.

Уровень промежуточного представления

Как видно из приведенных примеров, промежуточное представление программы может в различной степени быть близким либо к исходной программе, либо к машине. Например, промежуточное представление может содержать адреса переменных, и тогда оно уже не может быть

перенесено на другую машину. С другой стороны, промежуточное представление может содержать раздел описаний программы, и тогда информацию об адресах можно извлечь из обработки описаний. В то же время ясно, что первое более эффективно, чем второе. Операторы управления в промежуточном представлении могут быть представлены в исходном виде (в виде операторов языка if, for, while и т.д.), а могут содержаться в виде переходов. В первом случае некоторая информация может быть извлечена из самой структуры (например, для оператора for - информация о переменной цикла, которую, может быть, разумно хранить на регистре, для оператора case - информация о таблице меток и т.д.). Во втором случае представление проще и унифицированней.

Некоторые формы промежуточного представления удобны для различного рода оптимизаций, некоторые - нет (например, косвенные тройки, в отличие от префиксной записи, позволяют эффективное перемещение кода).

Оптимизация

1.Предварительные вычисления выражений.
x:=2 ; y:=3 ; z:=x+y+10; => z=15;

2. Исключение невыполнимых ветвей.

Switch
……..
Case : ……. Break; exit(0);

3. Выделение общих частей.

a:= (x+y)*z-35 |оптимизируем ==> w:= (x+y)*z; a:=w-35;
b:= (x+y)*z/a | b:=w/a;

4. Вынесение за цикл.

Выносятся значения, которые в цикле не вызываются.
… for (i=0; i<strlen(S); i++) printf(“%c”, S[i])…
… l=strlen(S); for (i=0; i<l; i++) …

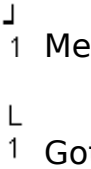
5. Вычисление логических выражений.

if (i<strlen(str)&&str[i] !=’x’)
cand если не выполняется 1-ое условие, то не выполнять дальнейшие условия.
Лучше делать вложенные циклы.

6. Изменение последовательности команд с целью оптимизации пересылки данных и обеспечения возможности реализации их параллельной обработки центральным процессором

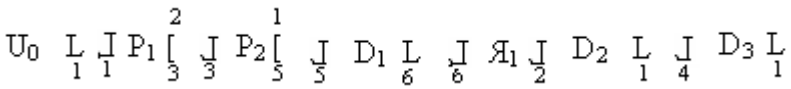
/без примера/

Пример1. Язык логических схем.



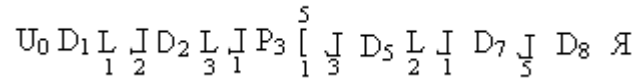
P – условие;
U_0 – начало
нет

]
да

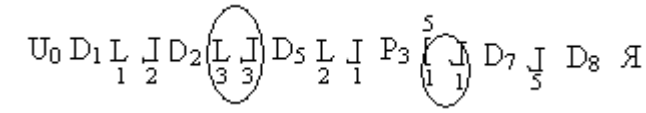


[P_1 : x > y; P_2 : y > x; D_1 : z:=x; D_2 : x:=x-y; D_3 : y:=y – x]

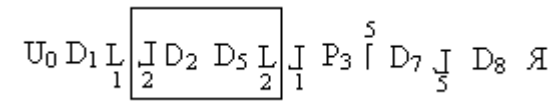
Пример2.



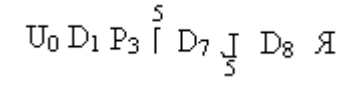
==>



==>



==>



Генерация выходного кода

При рассмотрении вопросов генерации выходного текста надо иметь в виду то, что реально выходной текст программы после трансляции - это, как правило, не выполняемый код, а некоторая промежуточная форма, поскольку программа в дальнейшем может быть

загружена для выполнения в разные места памяти и т.д. С другой стороны, выполняемая

программа (или программа в близком к такой форме виде) машинно-зависима, то есть

использует конкретную систему команд и другие конкретные архитектурные особенности. Основная идея генерации выходного кода заключается в подстановке входных конструкций в готовые шаблоны.

Язык:

```
create id 0
create iid 0
create sch 3
create ssh 2
inc sch
loop sch
    inc id
    loop ssh
            inc iid
    pool
pool
print id
```

```
print iid

#### forc.l    #####
%{ …
# include “forc_y.h”                //файл с лексемами
%}
letter [a-z]
digit [0-9]
%%
inc             {return (INC) ;}
create          {return (CREATE) ;}
print           {return (PRINT) ;}
loop            {return (LOOP) ;}
pool            {return (POOL) ;}
{letter}+       {strcpy(yylval.var, yytext); return(VAR);}
{digit}+        {sscanf(yytext,”%d”, &yylval.val); return(VAL);}
…
############

### forc.y #####
%union
{int val; char var[16];}
%token <var> VAR
%token <val>  VAL
%token  CREATE INC LOOP POOL PRINT
%%
prog:     str           {printf ( “main() {%s}”,$1);}
    ;
str:      oper          {sprintf($$,”%s”,$1);}
    | oper str    {sprintf($$,”%s \n %s”,$1,$2);}
    ;
oper:     CREATE VAR VAL          {sprintf($$,”int %s=%d”,$<var>2,
$<val>3);}
    |INC VAR                  {sprintf($$,”%s++”,$<var>2);}
    |LOOP VAR str POOL     {sprintf($$,”for(;%s>=0;%s--)
{%s}”,<var>2,<var>2, $3);}
    |PRINT VAR                      {sprintf($$,”printf(\”%s\”,%s)”, “%d”,
$<var>2) ;}
%%
…
############
```

Таблицы решений

На более низком уровне для подстановки используются таблицы решений генерации выходного кода.

ADD [регистр или память], регистр

A1+A2

A1	регистр	регистр	память	память
A2	регистр	память	регистр	память

код	ADD A1,A2	ADD A2,A1	ADD A1,A2	MOV A1,R ADD A2,R