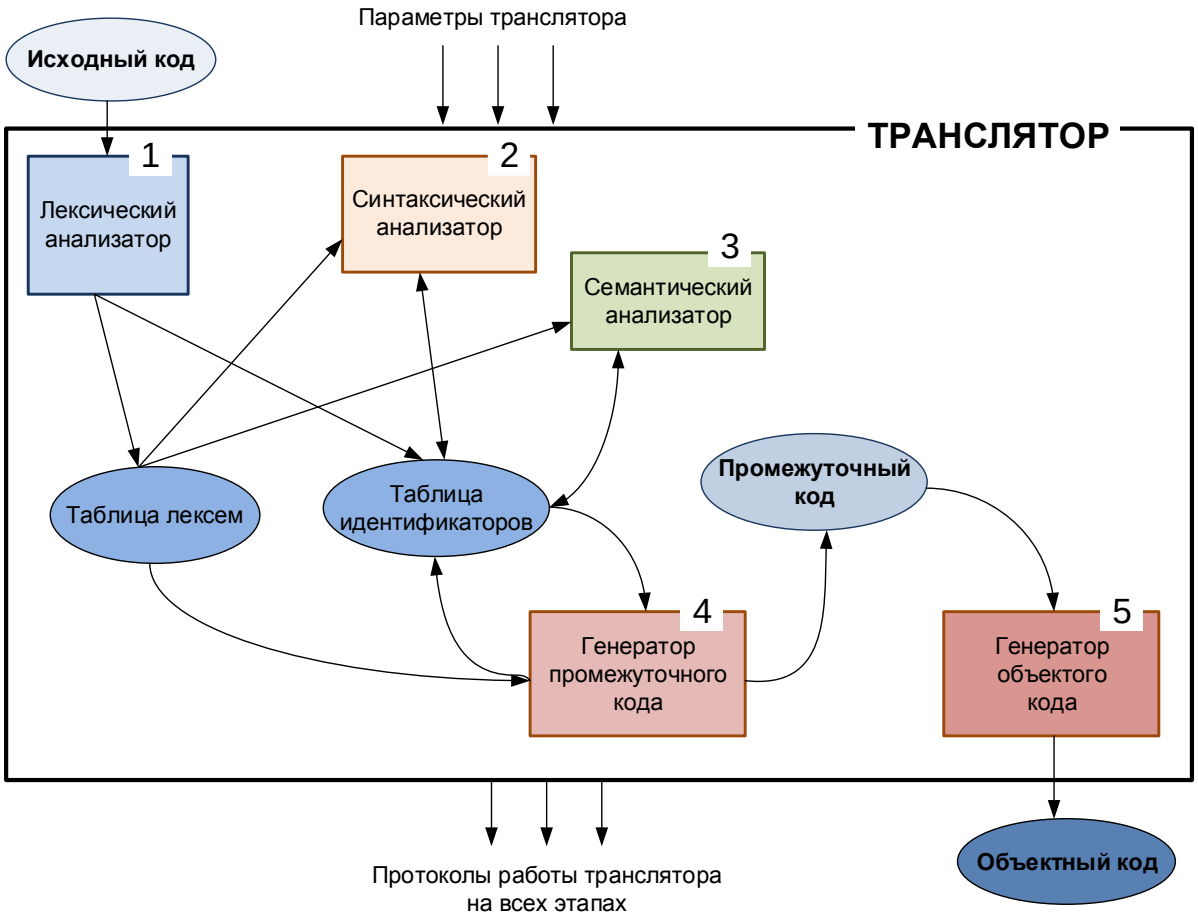
Лекция 22
БГТУ, ФИТ, ПОИТ, 3 семестр
Конструирование программного обеспечения

Генерация промежуточного кода

1. Структура транслятора

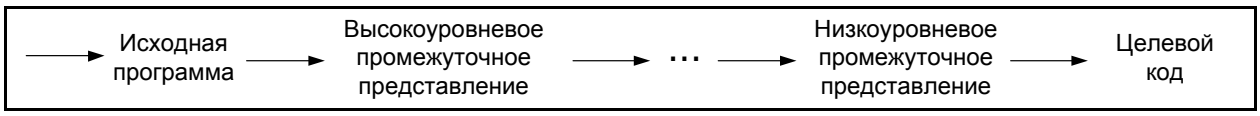


2. Генерация промежуточного кода. Методы генерации кода. Общие принципы генерации кода.

Промежуточный код: код удобный для генерации объектного кода.

Перед генерацией кода необходимо преобразовать выражения (сначала получить польскую запись, затем сгенерировать дополнительный код).

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка.



В процессе трансляции программы с некоторого исходного языка в код для заданной целевой машины компилятор может построить последовательность промежуточных представлений.
Высокоуровневые представления близки к исходному языку, а низкоуровневые – к целевому коду.

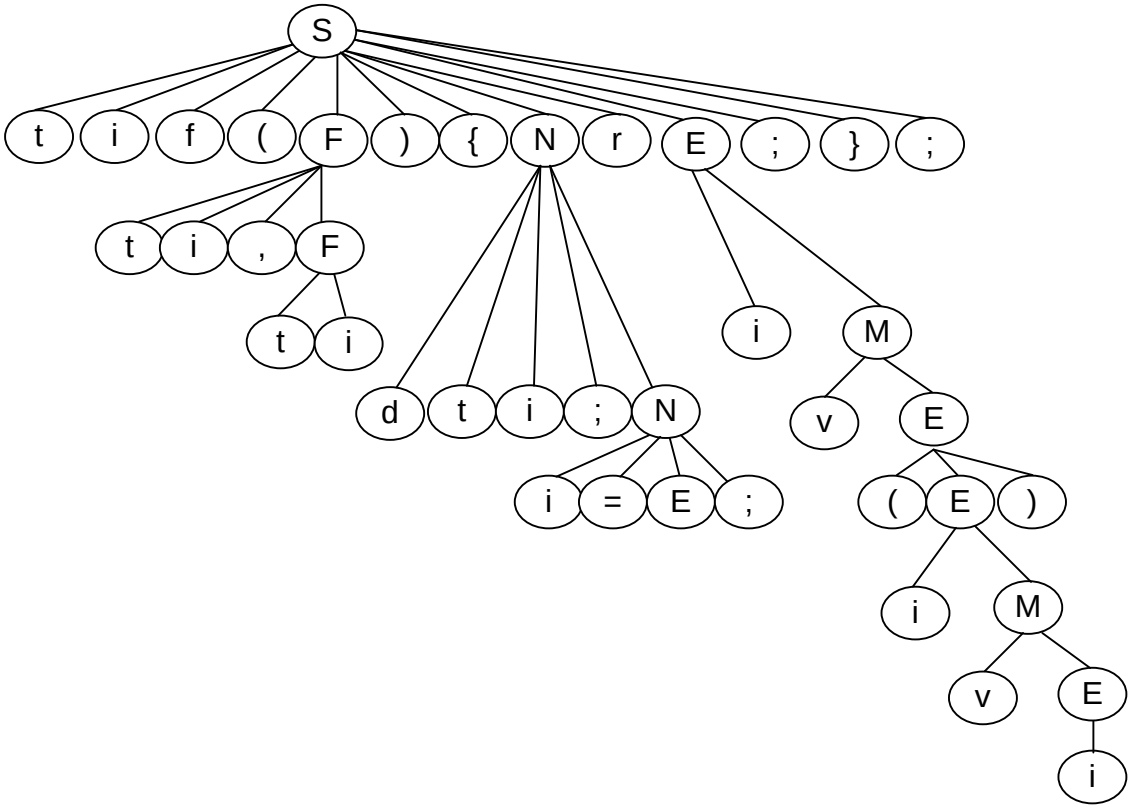
3. Способы внутреннего представления программ:

Формы внутреннего представления программ:

- списочные структуры, представляющие ***синтаксическое дерево***;
- многоадресный код с явно именуемым результатом (***тетрады***);
- многоадресный код с неявно именуемым результатом (***триады***);
- обратная (постфиксная) ***польская запись операций***;
- ***ассемблерный код*** или ***машинные команды***.

3.1 Синтаксические деревья

Это структура, представляющая собой результат работы синтаксического анализатора и отражающая синтаксис конструкций входного языка.



3.2 Многоадресный код с явно именуемым результатом (тетрады)

Тетрады – форма записи операций из ***четырех*** составляющих:

- операция;
- два операнда;
- результат операции.

<операция>(<операнд1>,<операнд2>,<результат>).

Пример. Запись выражения A:=B*C+D–B*10 в виде тетрад:

1) * (B, C, T1)
2) + (Tl, D, T2)
3) * (B, 10, T3)
4) – (T2, T3, T4)
5) := (T4, 0, A)

где идентификаторы T1, T2, T3, T4 обозначают временные переменные.

3.3 Многоадресный код с неявно именуемым результатом (триады)

Триады – форма записи операций из трех составляющих: операция и два операнда.

<операция>(<операнд1>,<операнд2>)

Пример. Запись выражения A:=B*C+D–B*10 в виде триад:

1) * (B, C)
2) + (^1, D)
3) * (B, 10)
4) – (^2, ^3)
5) := (A, ^4)

Знак ^ означает ссылку операнда одной триады на результат другой.

3.4 Обратная польская запись операций

Обратная (постфиксная) польская запись – удобная форма записи операций и операндов для вычисления выражений. Эта форма предусматривает, что знаки операций записываются после операндов.

Пример. Польская запись не содержит скобок.

a **+** ***b******** ***c*** **–** **a/(** ***a*** **+** ***b*****)** **=>** ***a b c*** ***** **+** ***a a b*** **+ / –**

3.5 Ассемблерный код и машинные команды

Команды ассемблера представляют собой форму записи машинных команд.

Внутреннее представление программы зависимо от архитектуры вычислительной системы, на которую ориентирован результирующий код.

Примеры синтаксических конструкций для различных языков программирования.

1). Операторы цикла:

do while <логическое_выражение> <операторы> **loop**	Повторяет <операторы> пока <логическое_выражение> истинно
do until <логическое_выражение> <операторы> **loop**	**?**
while <логическое_выражение> **{** <операторы> **}**	**?**
while <логическое_выражение> **:** <оператор> <оператор>	**?**
for [инициализация_счетчика]**;**[условие]**;**[изменение_счетчика] **{** <операторы> **}**	**?**
for ([инициализация_счетчика]**;**[условие]**;**[изменение_счетчика]**) {** <операторы> **}**	**?**
repeat { <операторы> **} while** <логическое_выражение>	**?**
for объект_последовательности **in** последовательность **{** <операторы> **}**	**?**
…	

2). Функции

Пример функции возведения в квадрат в некоторых языках программирования:

Императивный **C**:	Функциональный **Scheme**:

int square(int x)
{
  return x * x;
}
	(define square
	(lambda (x)
	(* x x)))

Конкатенативный **Joy**:	Конкатенативный **Factor**:
DEFINE square == dup * .	: square (x -- y) dup *;

Конкатенативный язык **Cat** (функциональный стековый язык программирования, обеспечивает статическую типизацию с выводом типов):

```
define square {
    dup *
}
```

здесь **dup** создает копию аргумента, находящегося на вершине стека, и заносит эту копию в стек;

* – функция умножения с сигнатурой `*:: (int) → (int) → (int);`

Вызов функции:

```
3 square
```

где `3` – тоже функция с сигнатурой `3::() → (int)`, т.е. возвращающая сама себя.

Пример. Hello world, выглядящий в этой нотации несколько непривычно:

```
"Hello world" print
```

В стек заносится строка «Hello World», а затем функция `print` извлекает элемент с вершины стека и выводит его в консоль.

Пример:

```
10 [ "Hello, " "Factor" append print ] times
```

Используется операция над ***цитатами***:

в стек кладется 10;

цитата – это код, заключенный в [];

times выполняет код из цитаты указанное количество раз;

код внутри цитаты последовательно кладет в стек две строки, затем соединяет их и выводит в консоль.

4. ПОЛИЗ: внутренняя форма представления программы.

Форма представления в обратной польской нотации достаточна проста для последующей интерпретации с использованием ***стека***.

Постфиксную запись выражений можно определить следующим образом:

1) если выражение E является единственным операндом a, то ПОЛИЗ выражения E – этот операнд a;

2) ПОЛИЗ выражения E_1 *op* E_2,
где *op* – знак бинарной операции,
E_1 и E_2 операнды для этой бинарной операции,
является запись:
E_1' E_2' *op* ,
где E_1' и E_2' – ПОЛИЗ выражений E_1 и E_2 соответственно;

3) ПОЛИЗ выражения (E) является ПОЛИЗ выражения E.

Замечание: для интерпретации, кроме ПОЛИЗ выражения, необходима дополнительная информация об операндах, хранящаяся в таблицах.

Замечание: может оказаться, что знак бинарной операции по написанию совпадает со знаком унарной операции; например, знак «–» в большинстве языков программирования означает и бинарную операцию вычитания, и унарную операцию изменения знака. В этом случае во время интерпретации операции «–» возникнет неоднозначность: сколько операндов надо извлекать из стека и какую операцию выполнять. Устранить неоднозначность можно двумя способами:

- заменить унарную операцию бинарной, т.е. считать, что –a означает 0–a;
- либо ввести специальный знак для обозначения унарной операции; например, –a **заменить** на a.

Важно: это изменение касается только внутреннего представления программы. Входной язык не требует изменения.

ПОЛИЗ для некоторых операторов некоторого входного языка.

	Оператор языка	Операция	Вид	ПОЛИЗ
1	Присваивание	:=	I := E	I E :=
2	Безусловный переход	!	goto L	p !
3	Условный оператор	!F	if B then S_1 else S_2	B p_1! !F S_1 p_2! S_2
4	Цикл с предусловием		while B do S	B p_1! !F S p_0!
5	Ввод	R	read (I)	I R
6	Вывод	W	write (E)	E W

1. Оператор присваивания `I E :=`

:=	– бинарная операция;
I и E	– операнды;
I	– адрес переменной I.

Семантика: …. (определяется спецификацией языка программирования)

2. Оператор перехода `p !` – унарная операция.

! – переход к элементу ПОЛИЗ с номером p, помеченному меткой L;
p – номер лексемы в таблице лексем (нумерация начинается с 1).

3. Условный оператор вида:

```
if <условие> then <выражение S1> else <выражение S2>
```

Введем вспомогательную операцию: условный переход ***по лжи*** с семантикой

```
if (not B) then goto L
```

Тогда условный оператор может быть описан:

```
        if (not B) then goto L2;
        S1;
        goto L3;
L2:  S2;
L3:  …
```

ПОЛИЗ условного оператора будет таким:

```
 B  p2 !  !F  S1  p3 !  S2
```

где

!F – условный оператор,
! – оператор перехода,
B – условие,
p_2 – номер элемента, с которого начинается ПОЛИЗ выражения S_2, помеченного меткой L_2,
p_3 – номер следующего за условным оператором.

Пример. `if x>0 then x:=x+8 else x:=x–3`

x	0	>	14	!	!F	x	x	8	+	:=	19	!	x	x	3	–	:=	…
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

4. Оператор цикла

Семантика оператора цикла **`while B do S`** может быть описана как:

```
L0: if (not B) then goto L1;
      S;
      goto L0;
L1: …
```

Тогда ПОЛИЗ оператора цикла **`while`** будет таким:

```
B p1!  !F S p0! …
```

,

где p_i — номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, $i = 0,1$.

Пример.

Простая инструкция `while` порождается продукцией:

```
S –> while (C) S1
```

где S – нетерминал;

C – нетерминал, представляет условное выражение, которое после вычисления принимает значение `true` или `false`.

Выполнение `while`.

- вычисление условного выражения;
- если оно истинно, то управление передается ***коду*** для выражения S_1;
- если ложно, то управление передается ***коду***, следующему за `while` (*S.next*).

Необходимо генерировать метки:

- *L1* – начало ***кода*** инструкции `while` для вычисления условного выражения C; по окончании ***кода*** S_1 будет осуществлен переход к ней;
- *L2* – определяет начало ***кода*** S_1, к которому передается управление, если `C.true`.

while	Встроенные действия для инструкции while (помечены красным)
S -> **while** (C) S_1	*L1* = new(); *L2* = new(); *C.false* = *S.next*; *C.true* = *L2*; S_1*.next* = *L1*; *S.code* = ***label*** || *L1* || *C.code* || ***label*** || *L2* || S_1*.code*

Здесь переменные *L1* и *L2* хранят метки, которые генерируются как первое действие продукции.

|| – конкатенация фрагментов кода.

5. Оператор ввода – унарная операция

`R` — операция ввода.

Тогда оператор ввода

```
read (I)
```

в ПОЛИЗ будет записан как

```
I R
```

6. Оператор вывода – унарная операция

`W` — операция вывода.

Тогда оператор вывода

```
write (E)
```

будет записан как

```
E W
```

> ***Замечание:***
>
> Цикл
>
> **for (<выражение1>; < выражение2>; <выражение3>)<оператор>**
>
> можно преобразовать к эквивалентному циклу с предусловием:
>
> **<выражение1>;**
> **while (<выражение2>){**
> **<оператор>;**
>
> **<выражение3>;**
>
> **}**

> ***Дополнение:***
>
> Одномерный массив. Введем операцию для одномерного массива в ПОЛИЗ, например `[`:
>
> `<имя_массива><индекс>[`

Пример. Схема компиляции для перевода выражений в обратную польскую запись.

Рассмотрим грамматику языка арифметических выражений с операциями: +, –, * и /:

```
G({+,–, /, *, a, b}, {S, T, E}, P, S):
P:
S –> S+T | S–T | T
T –> T*E | T/E | E
E –> (S) | a | b
```

Схему компиляции будем строить по следующему принципу:
имеется ***выходная*** цепочка символов ***R*** и известно текущее положение указателя ***p*** в этой цепочке. Распознаватель, выполняя подбор альтернативы по правилу грамматики, может записывать символы в выходную цепочку и менять текущее положение указателя в ней.

Схема компиляции:
с каждым правилом грамматики связаны некоторые ***действия***, которые записаны после «**;**» (точки с запятой) за правой частью каждого правила. Если никаких действий выполнять не нужно, в записи следует пустая цепочка ***(λ)***.

S –> S+T; R(p) = "+", p=p+1
S –> S–T; R(p) = "–", p=p+1
S –> T; λ
T –> T*E; R(p) = "*", p=p+1
T –> T/E; R(p) = "/", p=p+1
T –> E; λ
E –> (S); λ
E –> a; R(p) = "a", p=p+1
E –> b; R(p) = "b", p=p+l

Подобную схему компиляции можно использовать для любого распознавателя без возвратов, допускающего разбор входных цепочек на основе правил данной грамматики. Для каждого правила грамматики, распознаватель будет выполнять необходимые дополнительные действия, связанные с этим правилом. В результате будет построена цепочка *R*, содержащая представление исходного выражения.

Типы.

Проверка типов. Необходимо проверить, что типы операндов соответствуют типам, допустимым для данной операции.

Применение в трансляции. По типу переменной компилятор определяет количество памяти, требуемое для данного идентификатора. Например, для вычисления адреса при обращении к элементу массива; для добавления команд явного преобразования типов и т.п.

Фундаментальными типами многих языков программирования являются `boolean, char, integer`.

Правило определения эквивалентности типов:

> два типа эквивалентны тогда и только тогда, когда выполняется одно из условий:
> - они представляют собой собой один и тот же фундаментальный тип;
> - один тип представляет собой имя, обозначающее другой тип.

Объявление типов.

Правила упрощенной грамматики объявления типов по одному:

```
D –> T id; D | λ
T –> BC
B –> int | float | char
C –> λ | [num]
```

Грамматика работает с фундаментальными типами и массивами.
Нетерминал ***D*** порождает последовательность объявлений.
Нетерминал ***T*** порождает фундаментальные типы и массивы.
Нетерминал ***B*** порождает фудаментальные типы int или float, или char.
Нетерминал ***C*** порождает строку, состоящую из пустой строки или целого числа, размещенного в квадратных скобках. Тогда тип массива состоит из фундаментального типа, определяемого ***B***, за которым следует копонент, порождаемый нетерминалом ***C***, который указывает количество элементов массива.

Замечание.

Исходя из типа имени можно определить количество памяти, требуемое для размещения данного имени в процессе выполнения программы.
Во время компиляции эти значения могут использоватья для назначения каждому имени относительного адреса.

Тип и относительный адрес для данного имени может хранится в таблице идентификаторов.

Замечание.
Надо помнить о ***выравнивании*** адресов. Если массив состоит из 10 символов, то компилятор может выделить для него 12 байт – число кратное 4, оставляя неиспользованными 2 байта.

Размер типа равен количеству байт (единиц) памяти, необходимому для хранения объектов данного типа.
Память для массивов, классов выделяется одним непрерывным блоком байтов.

Для продукции `B -> int`, тип устанавливается как `integer` и ширина типа полагается равной 4 байтам, размеру целого числа.

Замечание.
Размерность целочисленного типа в битах в 32-битной архитектуре равна 32, а в 64-битной архитектуре – 64.

Размер массива вычисляется путем умножения размера элемента на количество элементов в массиве.

Компилятор должен выполнять проверки типов, т.е. определять удовлетворяют ли типы набору правил языка программирования. Реализация языка является ***строго типизированной***, если компилятор ***гарантирует***, что полученная в результате программа не будет иметь ошибок, связанных с типами данных.

В компиляторах могут использоваться разные модели кода, например, исходный текст программы, поток лексем, дерево разбора, трехадресный код, граф потока управления, байткод — стандартный или собственный, и т.д.

5. Рассмотрим схему преобразования промежуточного кода:

- предварительно преобразуем выражения и другие операторы языка в представление в виде обратной польской записи
- сгенерируем дополнительный код.

Способы внутреннего представления программ

Формы внутреннего представления программ:

- списочные структуры, представляющие синтаксические дерево;
- многоадресный код с явно именуемым результатом (тетрады);
- многоадресный код с неявно именуемым результатом (триады);
- обратная (постфиксная) польская запись операций;
- ассемблерный код или машинные команды.

Схема генерации промежуточного кода в трехадресный код

Внутреннее представление программы ближе к машинным командам.

Триады не зависят от архитектуры вычислительной системы, на которцю ориентирован результирующий код. Это машинно-независимая форма внутреннего представления программы.

Пример. Запись выражения A:=B*C+D–B*10 в виде триад:

1) * (B, C)	генерируется временная переменная для результата
2) + (^1, D)	2-я тетрада ссылается на результат 1-ой тетрады
3) * (B, 10)	
4) – (^2, ^3)	
5) := (A, ^4)	

Знак ^ означает ссылку операнда одной триады на результат другой.

1. В правой части команды **не более** одной операции.
2. Адресом может быть:
   - имя переменной;
   - константа;
   - временная переменная, генерируемая компилятором.

3. Распространенные трехадресные команды:
   a) Команды присваивания вида `x = y op z`,
      где *op* – бинарная арифметическая или логическая операция,
      x, y, z – адреса.
   b) Присваивание вида *x = op y*,
      где *op* – унарная операция (арифметические, логические, операторы преобразования переменных из одного типа в другой).
   c) Команды копирования вида *x = y*,
      в которых переменной x присваивается значение *y*.
   d) Безусловный переход *goto L*.
      После этой команды будет выполнена команда с адресом L.
   e) Условный переход вида `if x goto L` или `ifFalse x goto L`.
      Если значение x истинно (или ложно), то следующей выполняется команда L. В противном случае выполняется следующая команда за условным переходом.
   f) Условные переходы вида `if x relop y goto L`.
      Если оператор отношения relop (<, >, = и т. п.) верен, то следующей выполняется команда L. В противном случае выполняется следующая команда за условным переходом.
   g) Вызов процедуры вида `p(x1,x2,…,xn),` представляется в виде последовательность:

```
param x1
param x2
…
param xn
call p, n
```

      где n – количество фактических параметров. Вызовы могут быть вложенными.

   h) `return y` для возврата результата, где `y` обозначает необязательное возвращаемое значение
   i) Массивы (индексированные переменные) вида `x = y[i]` или `x[i] = y` (присвоение переменной `x` значения `i`-го элемента относительно `y` и занесение в `i`-ю ячейку памяти по отношению к `x` значения `y`).
   j) Присваивание адресов и указателей вида `x = &y, x = *y` и `*x = y`.

Рассмотрим инструкцию:

```
do   i = i + 1;   while  (a[i] <  v);
```

Возможные трансляции:

Символьные метки	Номера команд
L: t1 = i + 1	100 t1 = i + 1
i = t1	101 i = t1
t2 = i * 4	102 t2 = i * 4
t3 = a [t2]	103 t3 = a [t2]
if t3 < v goto L	104 if t3 < v goto 100

Вычисление выражений с помощью обратной польской записи

Вычисление выражений в обратной польской записи выполняется с использованием стека. Выражение просматривается в порядке слева направо, и встречающиеся в нем элементы обрабатываются по следующим правилам:

1. Если встречается операнд, то он помещается в вершину стека.
2. Если встречается знак унарной операции, то операнд выбирается с вершины стека, операция выполняется и результат помещается в вершину стека.
3. Если встречается знак бинарной операции, то два операнда выбираются с вершины стека, операция выполняется и результат помещается в вершину стека.

Вычисление выражения заканчивается, когда достигается конец записи выражения.

6. Пример:польская запись.

```
tfi(ti,ti)
{
 dti;
 i=iv(ivi);   z=x*(x+y)
 ri;
};
tfi(ti,ti)
{
 dti;
 dtfi(ti,ti,ti);
 i=i(i,l,l)vi;   substr(a,1,3)+b;
 ri;
};
m
{
 dti;
 dti;
 dti;
 dti;
 dti;
 dti;
 dtfi(ti);
 i=i;
 i=l;
 i=l;
 i=l;
 i=i(i,i);
 i=i(i,i);
 pl;
 pi;
 pi;
 pi(i);
 rl;
};
```

```
tfi(ti,ti)
{
 dti;
 i= iiivv;   z=xxy+*
 ri;
};
tfi(ti,ti)
{
 dti;
 dtfi(ti,ti,ti);
 i=ill@3iv;
 ri;
};
m
{
 dti;
 dti;
 dti;
 dti;
 dti;
 dti;
 dtfi(ti);
 i=i;
 i=l;
 i=l;
 i=l;
 i=ii@2;
 i=ii@2;
 pl;
 pi;
 pi;
 pi@1;
 rl;
};
```

7. Пример: генерация дополнительного кода.

tfi(ti,ti)
{
dti;
 	
i= iiivv; `z=xxy+*`
 	
ri;
};
tfi(ti,ti)
{
dti;
dtfi(ti,ti,ti);
i=ill@$_3$iv;
 	
ri;
};
m
{
dti;
dti;
dti;
dti;
dti;
dti;
dtfi(ti);
i=i;
i=l;
i=l;
i=l;
i=ii@$_2$;
i=ii@$_2$;
pl;
pi;
pi;
pi@$_1$;
rl;
};
	tfi(ti,ti)
	{
	dti;
	dti; `вводим временную переменную t1`
	i=iiv; `t1=x+y;`
	i=iiv; `z=x*t1;`
	ri;
	};
	tfi(ti,ti)
	{
	dti;
	dtfi(ti,ti,ti);
	dti;
	dti;
	i=ill@$_3$; `t4=substr(a,1,3)`
	i=iiv;
	ri;
	};
	m
	{
	dti;
	dti;
	dti;
	dti;
	dti;
	dti;
	dtfi(ti);
	i=i;
	i=l;
	i=l;
	i=l;
	i=ii@$_2$;
	i=ii@$_2$;
	pl;
	pi;
	pi;
	pi(i);
	rl;
	};

8. Пример: промежуточное представление в виде тетрад

Тетрады: `операция (операнд1, операнд2, результат3)`

start(p1,p2,p3)	Создать таблицу ссылок
	p1 = null //операнд1
	p2 = null //операнд2
	p3 = адрес таблицы ссылок //результат

entry(p1,p2,p3)	Поместить ссылку на локальную функцию в таблицу ссылок. Инициализировать счетчик стека. p1 = адрес таблицы ссылок p2 = имя функции p3 = адрес ссылки

mentry(p1,p2,p3)	Поместить ссылку на главную локальную функцию в таблицу ссылок. Инициализировать счетчик стека.
	p1 = адрес таблицы ссылок
	p2 = main //операнд1
	p3 = адрес ссылки //операнд2

ext(p1,p2,p3)	поместить ссылку на внешнюю функцию в таблицу ссылок p1 = адрес таблицы ссылок p2 = null p3 = адрес ссылки на вызываемую функцию
stackaddr(p1,p2,p3)	Вычислить адрес в стеке p1 = смещение p2 = null p3 = адрес
push(p1,p2,p3)	Записать в стек p1 = длина p2 = значение p3 = null
pop(p1,p2,p3)	Сдвинуть стек p1 = количество байт p2 = null p3 = null
+(p1,p2,p3)	Вычислить сумму двух integer-значений, результат поместить в стек p1 = значение 1 p2 = значение 2 p3 = адрес результата в стеке
*(p1,p2,p3)	Вычислить произведение двух integer-значений, результат поместить в стек p1 = значение 1 p2 = значение 2 p3 = адрес результата в стеке

cont(p1,p2,p3)	Конкатенация двух string-значений, результат поместить в стек p1 = значение 1 p2 = значение 2 p3 = адрес результата в стеке
str(p1,p2,p3)	Сформировать строку p1 = длина p2 = null p3 = адрес результата в стеке
store(p1,p2,p3)	Скопировать данные (*a = b*) p1 = адрес источника (*b*) p2 = адрес получателя (*a)* p3 = null
callstd(p1,p2,p3)	Вызов внешней функции p1 = адрес в таблице ссылок p2 = null p3 = null
callloc(p1,p2,p3)	Вызов локальной функции p1 = адрес в таблице ссылок p2 = null p3 = null
prints(p1,p2,p3)	Писать строку в стандартный вывод p1 = строка p2 = null p3 = null
printi(p1,p2,p3)	Писать string-значение в стандартный вывод p1 = string-значение p2 = null p3 = null
goto(p1,p2,p3)	Переход по адресу p1 = адрес перехода p2 = null p3 = null

Исходный код	***Таблица лексем***	***Тетрады***
		start(null,null,start0) //создание таблицы ссылок

integer function
fi(integer x, integer y)
{
	tfi(ti,ti)
	{
		entry(start0, //адрес таблицы
		fi, //имя функции
		entryfi) //адрес ссылки на fi
		//помешаем в стек (используется соглашение о вызовах)
		stackaddr(0, null, fi01) //адрес точки возврата (0)
		stackaddr(4, null, fi02) //адрес результата (смещение 4)
		stackaddr(8, null, fix) //адрес параметра x (8)
		stackaddr(12, null, fiy) //адрес параметра y (12)

declare integer z;	dti;	push(4,0,fiz) //поместить в стек z
z= x*(x+y);	**dti;**	push(4,0,fi03) //поместить в стек временную переменную fi03
	i=iiv;	+(fix,fiy,fi04) //x+y → в стек store(fi04,fi03,null) //записать в fi03
	i=iiv;	*(fix,f03,fi05) //x*y → в стек store(fi05,fiz,null) //записать в fiz

return z;};	ri};	store(fiz,fi02,null) //копируем в fi02 возвращ. значение
		pop(16, null, null) //очистка стека на 16 байтов
		goto(fi01,null,null) //переход по адресу fi01

string function
fs(string a, string b) {
	tfi(ti,ti)
	{
		entry(start0, //адрес таблицы
		‘fs’, //имя функции
		entryfs) //адрес ссылки на fs
		
		stackaddr(0, null, fs01) //ret
		stackaddr(4, null, fs02) //rc
		stackaddr(8, null, fsa) //адрес параметра fsa

		stackaddr(12, null, fsb) //адрес параметра fsb

declare string c;	dti;	str(0, null,fs05) //временная переменная (строка 1+255)
		push(4,fs05,fsc) //адрес

declare string function substr(string a, integer p, integer n);	dtfi(ti,ti,ti);	ext(start0 ,'substr@s@i@i', fs04) //push

c = substr(a, 1,3)+ b;	**dti;**	str(0, null,fs06) // new fs06 1+255
		push(4,fs06,fs07) // адрес

	dti;	str(0, null,fs08) // new 1+255
		push(4,fs06,fs09) // адрес

	i=ill@3;	push(4,fs10, null) //адрес
		push(4,3,null) //параметр: литерал 3
		push(4,1,null) //параметр: литерал 1
		push(4,fsa,null) //параметр: fsa
		callstd(fs04,null,null) //вызов внешней функции
		fs10:store(fs10,fs07,null) //скопировать: fs07= fs10
		pop(16,null, null) //очистка стека на 16 байтов

	i=iiv;	cont(fs07,fsb,fs11) //адрес результата конкатенации в стеке
		store(fs11,fsc,null) //скопировать: fsc= fs11

return c;
};
	ri;};	store(fsc,fc02,null) //скопировать: fs02= fsc
		pop(16, null, null) //очистка стека на 16 байтов
		goto(fs01,null,null) //выход по адресу fs01

main
{
	m
	{
		mentry(start0, null, null)
		stackaddr(0, null, main01) // ret
		stackaddr(4, null, main02) // rc

declare integer x;	dti;	push(4,0,mainx)
declare integer y;	dti;	push(4,0,mainy)
declare integer z;	dti;	push(4,0,mainz)

declare string sa;	dti;	str(0, null,main03) // new 1+255
		push(4, main03, mainsa) // адрес

declare string sb;	dti;	str(0, null,main04) // new 1+255
		push(4, main04, mainsb) // адрес

declare string sc;	dti;	str(0, null,main05) // new 1+255
		push(4, main05, mainsc) // адрес

declare integer function strlen(string p);	dtfi(ti);	ext(start0,'strlen@s', main04) //push
x = 1;	i=l;	store(1,mainx,null) //копировать: mainx=1
y = 5;	i=l;	store(5,mainy,null) //копировать: mainy=5
sa = '1234567890';	i=l;	store('1234567890',mainsa,null) //копировать: mainsa=литерал
sb = '1234567890';	i=l;	store('1234567890',mainsb,null) //копировать в mainsb

z = fi(x,y);	**i=ii@2;**	push(4,main07,null) //ret
		push(4,0,main08) //rc
		push(4, mainx,null) //параметр
		push(4, mainy,null) //параметр
		callloc(entryfi,null,null) //вызов локальной функции
		main05:store(main08,mainz,null) //копировать: z= rc
		pop(16,null, null) //очистка стека на 16 байтов

sc = fs(sa,sb);	**i=ii@2;**	push(4,main09,null) //ret
		push(4,0,main10) //rc
		push(4,mainsb,null) //параметр
		push(4,mainsa,null) //параметр

		callloc(entryfs,null,null) //вызов локальной функции
		main07:store(main10,mainsc,null) //копировать: sc= rc
		pop(16,null, null) //очистка стека на 16 байтов

print ‘контрольный пример‘;	pl;	prints(‘контрольный пример‘,null,null) //вывод строки-литерала
print z;	pi;	printi(mainz,null,null) //вывод строки-значения
print sc;	pi;	prints(mainsc,null,null) //вывод строки

print strlen(sc);	**pi@$_1$**;	push(4, main11,null) //ret
		push(4,0,main12) //rc
		push(4,mainsc,null) //параметр
		callstd(main04,null,null) //вызов внешней функции
		main11: pop(12,null, null) //очистка стека на 12 байтов
		printi(main12, null, null)

return 0;
};
	rl;
	};
		store(0,main02,null) //копировать: rc=0
		goto(main01,null,null) // переход по main01 (ret)