

Промежуточные представления (IR)

Вспомним, как выглядят фазы компиляции:

↓ *исходный текст*

фронтенд: **анализ** исходного текста. Если есть ошибки, то останавливаемся.

↓ *промежуточное представление*

... в сложных системах может быть несколько пром. представлений

↓ *промежуточное представление*

бэкенд: **синтез** - генерация программы, которая нам нужна вместе с какими-то оптимизациями.

↓ *целевой код*

После синтаксического и семантического анализа некоторые компиляторы генерируют явное *промежуточное представление* исходной программы, которое можно рассматривать как программу для абстрактной машины. Это представление должно легко создаваться и транслироваться в целевую программу.

При оптимизации кода производятся попытки улучшить промежуточный код, чтобы получить более эффективный машинный код.

Разберём, какие промежуточные представления бывают, одним из них позанимаемся более плотно.

Классификация

- по степени абстракции
- по структуре:
 - графические — представимы в виде графов
 - линейные
 - гибридные

Виды IR

Графические IR

Синтаксическое дерево и даг

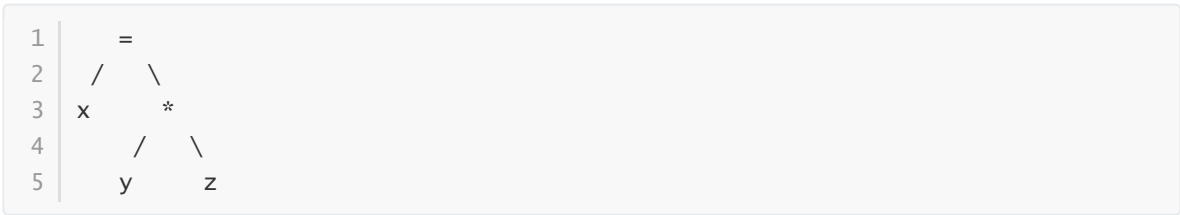
Всё прослушала, тут что-то рассказывали

Синтаксическое дерево получается из дерева вывода применением (до тех пор, пока это возможно) следующих преобразований:

- листья, соответствующие терминалам-операторам (и ключевым словам), удаляются, а данные терминалы связываются с родительским узлом.
- ветви дерева, соответствующие цепным правилам, сворачиваются в одну вершину.

В результате все узлы преобразованного дерева помечены терминалами (листья — операндами, а внутренние узлы — операторами и другими аналогичными по смыслу языковыми конструкциями). Атрибуты при семантическом анализе присоединяются к узлам дерева, как и в случае дерева вывода.

`x=y*z`



Поле дочерних узлов используем, чтобы показать...

Номер узла	Ссылка на узел	Левый операнд	Правый операнд
1	x	x.lexval	
2	y	y.lexval	
3	z	z.lexval	
4	*	2	3
5	=	1	3

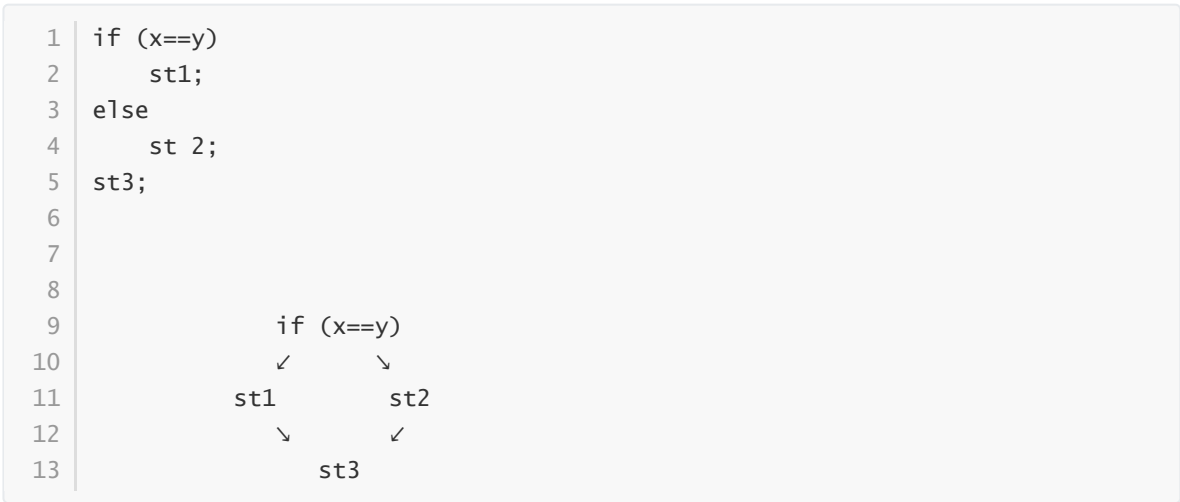
Если хотим создать даг, то нужно проверять, а нет ли уже такого узла. Как проверить? По **сигнатуре** — метке и ссылке на сыновей.

Все узлы дага разбиваются на группы и используется хэш.

Зачем: чтобы не занимать лишнюю память и переиспользовать блоки кода

Граф потока управления (Control Flow Graph)

В качестве вершин этого графа используются блоки исполняемого кода или отдельные команды, в качестве рёбер — возможная передача управления между этими блоками.



Зачем: чтобы находить мёртвый код и циклы

Граф зависимостей данных

Очень похож на граф зависимостей для атрибутивных грамматик. Отражает связи и потоки данных между объявлением переменных и их использованием.

$w = w * 2 * x * y$

```
1 | load r1, @w
2 | mult r1, 2
3 | load r2, @x
4 | mult r1, r2
5 | load r3, @y
6 | mult r1, r3
```

```
1 | 1
2 | ↓
3 | 2   3
4 | ↓ ✓
5 | 4   5
6 | ↓ ✓
7 | 6
```

Пример гибридного, так как если в качестве узлов используем целые блоки, то внутри одного узла этот блок может быть представлен в виде линейного кода или синтаксического дерева.

Зачем: выкидывать ненужные переменные; эффективно параллелить программу.

Линейные IR

Типы отличаются друг от друга количеством адресов, используемых в своих командах.

Адрес — не относится к адресу в памяти. Это одно из трёх:

- имя переменной
- временное имя, сгенерированное компилятором
- константа

Одноадресный код

Команды могут использовать только один адрес. Нужен стек, так как унарных операций маловато. Есть возможность обменять вершину и элемент, лежащий под ней.

$x * y - 2$

```
1 | push x
2 | push y
3 | mult
4 | push 2
5 | subtr
```

Двухадресный код

Если к операндам применяется операция, то результат должен оказаться в одной из переменных

```
1 | mult x, y ;теперь результат лежит в x
2 | subtr x, 2
```

Деструктивный характер — всё время перезаписываем переменные.

Трёхадресный код

Громоздкое, но универсальное промежуточное представление. От того, как мы опишем множество команд и свяжем с целевой программой, зависит уровень абстракции. Может быть как аналогом синтаксического дерева, но может быть и приближен к математическому языку.

Набор команд, с которым будем работать:

- $x = y \text{ op } z$ — бинарная операция с присваиванием
- $x = \text{op } y$ — унарная операция с присваиванием
- $x = y$ — присваивание
- `goto L` — безусловный переход. Переменная тоже может быть меткой
- `if x goto L` и `if false x goto L` — условный переход
- `if x relop y goto L` — условный переход с оператором сравнения
- `y = x[i]` и `x[i] = y` — присваивание с индексацией

Чтобы вызвать функцию, нужно передать в неё параметр:

- $\text{param } x_1$ — передаём в функцию n аргументов
 $\text{param } x_n$
 $\text{call } f, n$
- `return x`
- ссылки и указатели:
 - $x = \&y$ — положить в x указатель на значение y
 - $x = *y$ — положить в x значение по адресу, лежащему в y
 - $*x = y$ — положить в объект, лежащий по адресу x , y

Вообще, компилятор проверяет L и R значения. Хотя слева и справа используются переменные, но то, что слева, используется как адрес, а справа — как значение. То есть слева нельзя записать R-значение, например, константу.

В качестве линейного промежуточного представления может быть какой-то язык. Например, промежуточным представлением для C++ когда то бы обычный Си.

Связь линейного кода с графическим.

```

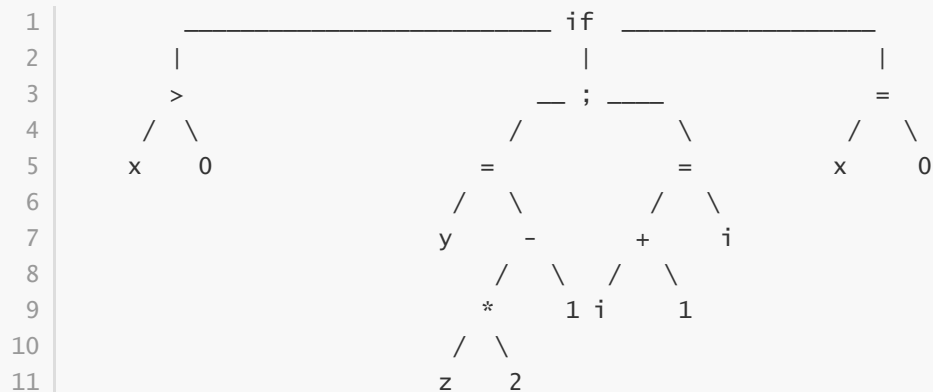
1  if (x > 0)
2  {
3      y = z*2 - 1
4      i++;
5  }
6  x = 0;

```

```

1  if False x > 0 goto 5
2  t1 = z*2
3  y = t1 - 1 ; t2 = t1 - 1, y = t2
4  i = i + 1
5  L5: x = 0

```



Представление внутри компилятора

- **Четвёрки**, из которых получается нумерованный список:
 2. Оператор
 3. Левый операнд
 4. Правый операнд
 5. Куда складываем результат

Номер (не используется)	Оператор	Левый операнд	Правый операнд	Результат
0	<i>if left > right goto</i>	<i>x</i>	0	<i>L5</i>
1	*	<i>z</i>	2	<i>t₁</i>
2	—	<i>t₁</i>	1	<i>y</i>
3	+	<i>i</i>	1	<i>i</i>
4	=	0		<i>x</i>

За счёт того, что строки не адресуют друг друга непосредственно, их можно спокойно менять местами, чем и пользуются некоторые оптимизирующие компиляторы. Но нужно больше памяти на лишний столбик.

- **Тройки** — линейное представление синтаксического дерева:

1. Оператор
2. Левый операнд
3. Правый операнд

Левый и правый операнд могут быть не только переменными и константами, но и номерами команд.

Номер тройки	Оператор	Левый операнд	Правый операнд
(0)	*	z	2
(1)	—	(0)	1
(2)	=	y	(1)

Перемешивать строчки просто так — сложно, поэтому используется косвенная адресация — *косвенные тройки*.

- **Косвенные тройки** — оставляем таблицу, как в обычных тройках, а переупорядочивать будем список указателей на эти тройки:

Индекс	...	33	34	35	...
Instruction	...	(0)	(1)	(2)	...

Трансляция выражений

$S \rightarrow id = E$	$S.code = E.code gen(\square 0 \square)$
$E \rightarrow E_1 + E_2$	$E.addr = new Temp(),$ $E.code = E_1.code E_2.code gen(\square 0 \square)$
$E \rightarrow (E_1)$	$E.addr = E_1.addr,$ $E.code = E_1.code$
$E \rightarrow id$	$E.addr = get(id.lexval),$ $E.code = ""$

Атрибуты (синтезируемые):

- `addr` — по адресам хранятся вычисленные значения
- `code` —

Обозначения:

`||` — конкатенация двух кодов

`{var}+{var}`` — строковая интерполяция.

Инструкции изменения потока управления

Булевы выражения

Нам их нужно либо вычислять, либо использовать. Это два разных контекста. Их можно различать с помощью, например, наследуемого атрибута. Нам интересно, как транслируются условия и циклы

$$B \rightarrow B || B \&\& B | !B | E \text{ rel } E | \text{true} | \text{false}$$

$$\text{rel} \in \{\leq, <, >, \geq, \neq, ==\}$$

Спецификация языка обязательно упоминает о сокращённых вычислениях. Например, если первый операнд в конъюнкции ложен, то остальные вычисляться не будут. Мы ими будем пользоваться.

```
1  if (x>200 || x<100 && x!=y)
2      x = 0;
```

```
1  if x > 200 goto L1
2  if False x < 100 goto L2
3  if False x != y goto L2
4  L1: x = 0;
5  L2: ...
```

$$S \rightarrow \text{if}(B)S_1$$

$$S \rightarrow \text{while}(B)S_1$$

$$S \rightarrow \text{if}(B)S_1 \text{ else } S_2$$

Атрибуты:

- next — метка следующей команды, которая будет выполняться после S
- true/false — метка первой команды, которая выполнится если B истинно\ложно

Примеры блоков:

if:

```
1      B.code
2  B.true  -> S1.code
3  B.false -> .. <- S.next
```

while

```
1      begin  -> B.code
2      B.true -> S1.code
3                      goto begin
4  B.false -> ... <- S.next
```

if-else

```

1 |           B.code
2 |   B.true ->  S1.code
3 |           goto [S.next]
4 |   B.false -> S2.code
5 |   S.next ->  ..

```

Теперь опишем семантическую грамматику:

Грамматика для трансляции блоков кода

Вспомогательные функции и процедуры:

- `newlabel()` — новая метка;
- `label(...)` — пометить команду меткой;
- `gen(...)` — генерирует код. В кавычках — обычная строка, без кавычек — подставляем значение.

$P \rightarrow S$	$S.next = newlabel(),$ $P.code = S.code label(S.next)$
$S \rightarrow assign$	$S.code = assign.code$
$S \rightarrow if(B)S_1$	$S_1.next = S.next,$ $B.true = newlabel(),$ $B.false = S.next,$ $S.code = B.code label(B.true) S_1.code$
$S \rightarrow while(B)S_1$	$begin = newlabel(),$ $S_1.next = begin,$ $B.true = newlabel() \text{ — тело цикла,}$ $B.false = S_1.next$ $S.code = label(begin) B.code label(B.true) S_1.code gen('goto' begin')$
$S \rightarrow if(B)S_1 else S_2$	$S_1.next = S.next,$ $S_2.next = S.next$ $B.true = newlabel(),$ $B.false = newlabel()$ $S.code = B.code$ $ label(B.true)$ $ S_1.code$ $ gen('goto' S.next)$ $ label(B.false)$ $ S_2.code$
$S \rightarrow S_1; S_2$	$S_1.next = newlabel(),$ $S_2.next = S.next,$ $S.code = S_1.code label(S_1.next) S_2.code$

Грамматика для трансляции булевых операторов


```

20
21
22  if x > 200 goto L2
23  goto L3
24  L3: if x < 100 goto L4
25      goto L1
26  L4: if x!=y goto L2
27      goto L1
28  L2: x = 0
29  L1: ...

```

Сокращённые вычисления

Введём новую команду *fall* — “не генерируй команды перехода” или “провались к следующей команде”. Зачем она нужна? Рассмотрим трансляцию выражения $x < 100$ в примере выше:

```

1  if x < 100 goto L4
2      goto L1
3  L4: ...

```

Если бы мы использовали конструкцию *if False* запись стала бы короче, и управление бы “проваливалось” к следующей команде автоматом:

```

1  if False x < 100 goto L1
2  L4: if x!=y goto L2

```

Для этого в правиле $S \rightarrow if(B)S_1$ мы заменим действие $B.true = newlabel()$ на $B.true = fall$, а из генерации кода выкинем шаг с генерацией метки для $B.true$. Таким же образом можно изменить правила для конструкций *if – else* и *while*:

$S \rightarrow if(B)S_1$	$S_1.next = S.next,$ $B.true = fall,$ $B.false = S.next,$ $S.code = B.code S_1.code$
$S \rightarrow while(B)S_1$	$begin = newlabel(),$ $S_1.next = begin,$ $B.true = fall,$ $B.false = S_1.next,$ $S.code = label(begin) B.code label(B.true) S_1.code gen('goto' begin')$
$S \rightarrow if(B)S_1 else S_2$	$S_1.next = S.next,$ $S_2.next = S.next$ $B.true = fall,$ $B.false = newlabel(),$ $S.code = B.code$ $ S_1.code$ $ gen('goto' S.next)$ $ label(B.false)$ $ S_2.code$

Булевы выражения тоже можно упростить. Рассмотрим правила:

- $B \rightarrow B_1 || B_2$

- $B \rightarrow E_1 \text{ rel } E_2$

Если B_1 — истинен, то можно не считать B_2 , а сразу переходить к тому коду, который выполняется в случае истинности всего выражения. Если B_1 всё таки ложно, то надо попробовать посчитать B_2 . Для этого положим в B .*false* нашу новую команду *fall*.

Для операции $\&\&$ мы бы наоборот, положим *fall* в B .*true*, потому что, чтобы проверить истинность всего высказывания, нужно проверить истинность всех частей

Генерация кода для правила $B \rightarrow E_1 \text{ rel } E_2$:

```

1 test = E_1.addr rel.op E_2.addr #сравниваем два операнда
2
3 if B.true != fall && B.false != fall:
4     # обе метки реальные, нужны оба перехода
5     s = gen(`if {test} goto {B.true}`)
6     || gen(`if False {test} goto {B.false}`)
7 elif B.true != fall, B.false == fall:
8     # если B - ложно, то попробуем следующий блок
9     s = gen(`if {test} goto {B.true}`)
10 elif B.true == fall, B.false != fall:
11     # если B - ложно, то дальше считать ничего не будем
12     s = gen(`if False {test} goto {B.false}`)
13 else s = ''
14
15 B.code = E_1.code || E_2.code || s

```

$B \rightarrow B_1 B_2$	$B_1.true = \begin{cases} B.true, & B.true \neq fall \\ newlabel(), & otherwise \end{cases}$ $B_1.false = fall,$ $B_2.true = B.true,$ $B_2.false = B.false,$ $B.code = \begin{cases} B_1.code B_2.code, & B.true \neq fall \\ B_1.code B_2.code label(B_1.true), & otherwise \end{cases}$
$B \rightarrow B_1 \&\& B_2$	$B_1.false = \begin{cases} B.false, & B.false \neq fall \\ newlabel(), & otherwise \end{cases}$ $B_1.true = fall,$ $B_2.true = B.true,$ $B_2.false = B.false,$ $B.code = \begin{cases} B_1.code B_2.code, & B.false \neq fall \\ B_1.code B_2.code label(B_1.false), & otherwise \end{cases}$
$B \rightarrow !B_1$	
$B \rightarrow E_1 \text{ rel } E_2$	см. код над этими табличками

И с помощью этой грамматики выражение

```

1 if (x>200 || x<100 && x!=y)
2     x = 0;

```

превратиться в

```
1  if x > 200 goto L1
2  if False x < 100 goto L2
3  if False x != y goto L2
4  L1: x = 0;
5  L2: ...
```

Метод обратных поправок

Два прохода, чтобы спускать наследуемые атрибуты. А хотим — за один раз с помощью восходящего анализа. Маркеры нам помогут!

$S \rightarrow if(B)S_1$. Если B ложно, то мы должны выполнить код, который следует за S_1 . Но, когда мы разбираем B , про S_1 мы ещё ничего не знаем! Поэтому метки в GOTO мы оставляем пустыми, и запоминаем, что эту команду мы ещё не заполнили. Когда метка станет известным, можно будет эту команду дозаполнить. Поэтому **обратные поправки**

Нужны новые синтезируемые атрибуты, наследуемые будем прятать в маркеры.

Списки команд перехода, в которые нужно вставить метки команд, которые нужно выполнить в случае истинности или ложности B

- $B.truelist$
- $B.falselist$

У S появится атрибут из списка команд, к которым нужно перейти после выполнения S

- $S.nextlist$

Нужны вспомогательные функции:

- $makelist(i)$ — создаёт список из единственной команды с номером i , возвращает ссылку на этот список
- $merge(p_1, p_2)$ — слияние списков

Вспомогательная процедура:

- $backpatch(p, i)$ берёт все команды, по которым не проставлен переход, и заполняет их командой перехода к

15		E	<	E		B	&&	M		B		
16						/		\		/		\
17		x		100		E	>	E		E	!=	E
18												
19						x		200		x		y
20												
21												
22												
23												
24												
25												
26												
27												
28												
29												
30												
31												
32												
33												
34												
35												
36												
37												
38												
39												
40												
41												

Когда сворачиваем первый куст, получаем первые списки, t и f.
 В списке B.t будет лежать команда 1, в B.f - команда 2
 Когда сворачиваем второй куст, B.t = {3}, B.f = {4}
 Маркер получает номер команды номер 5
 Последний куст: B.t = {5}, B.f = {6}
 начинаем сворачиваться. B.t = {5}, B.f = {4,6}.
 А ещё нужно выполнить обратную поправку. В левом ребёнке в t лежит 3,
 значит в третью команду нужно вставить команду из маркера.
 Сворачиваем верхнее B. Берём false у B_2, обратная поправка тоже работает с f.
 Из 4 и 6 незаполненные команды должны вести в 7. Но пока семёрки нет в
 каком-то списке, мы их заполнить не можем
 S.n = {4,5}
 M = 7

Код:

```

1  if x < 100 goto ____ (7)
2  goto ____ (3) line 34
3  if x > 200 goto ____ (5) line 32
4  goto ____
5  if x != y goto ____ (7)
6  goto ____
7  x = 0

```

КОНЕЦ!

Экзамен:

- задача — получить за день до экзамена
- простой вопрос
- вопрос посложнее