ВНУТРЕННИЕ ФОРМЫ ПРЕДСТАВЛЕНИЯ ПРОГРАММ

В большинстве случаев при трансляции целевой код не создается транслятором непосредственно по исходному коду. Вместо этого исходная программа вначале переводится в некую *внутреннюю форму*, называемую также *промежуточным кодом* (англ. intermediate code), а соответствующий язык называется *промежуточным языком* (англ. Intermediate language). Использование внутренней формы представления программы при трансляции позволяет разбить транслятор на два крупных блока (см. рис. 1), что дает ряд преимуществ перед непосредственной трансляцией с исходного языка на целевой.

Рис. 1 - Разделение транслятора при использовании промежуточного языка

Такое разделение транслятора на части и использование промежуточного языка позволяет упростить построение трансляторов с различных исходных языков и для разных целевых платформ. Так, в случае, когда требуется построить транслятор с нового языка, достаточно разработать ту часть транслятора, которая отвечает за генерацию промежуточного кода. В том случае, когда необходимо построить компилятор для новой целевой платформы или транслятор для нового целевого языка, требуется лишь разработать часть, отвечающую за генерацию целевого кода по промежуточному коду.

Более того, замена части, отвечающей за генерацию целевого (машинного) кода интерпретатором промежуточного кода, делает возможным построение интерпретаторов для тех входных языков, для которых уже существуют трансляторы во внутреннюю

форму представления. При этом часть транслятора, отвечающая за оптимизацию промежуточного кода, во всех приведенных случаях остается прежней.

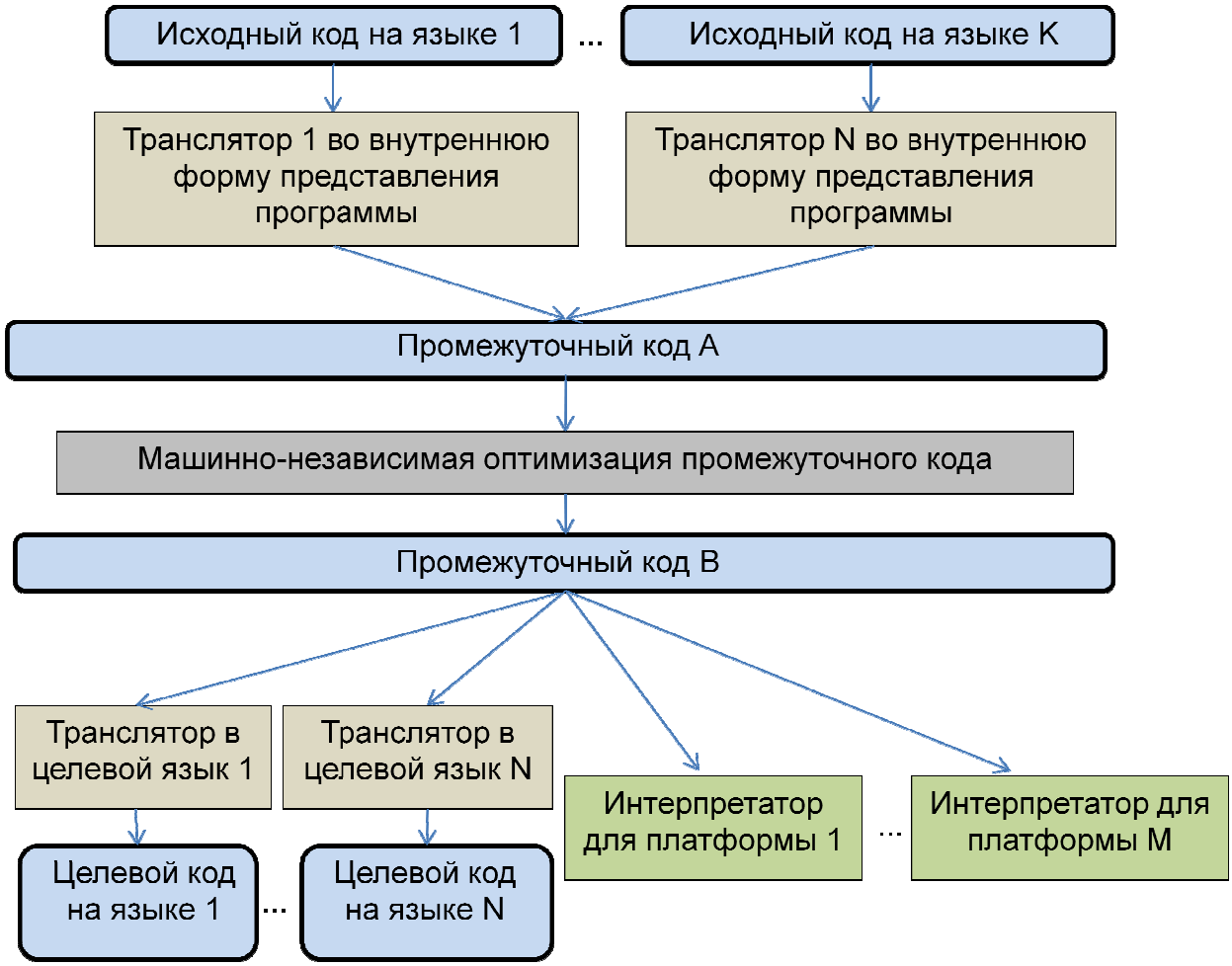

Рис. 2 - Построение трансляторов для различных языков и платформ

1. Требования к внутренним формам представления программ

Основные требованиями к внутренним формам представления программ заключаются в следующем [2]:

- трансляция с предполагаемых исходных языков во внутреннюю форму представления должна выполняться достаточно просто;

- трансляция внутренней формы представления программы в целевые языки (машинный код целевых платформ) должна выполняться достаточно просто;

- внутренняя форма представления должна быть удобна для выполнения машинно-независимой оптимизации;

- в том случае, если требуется создание интерпретатора, внутренняя форма представления должна допускать эффективную интерпретацию.

Естественно, что какая-либо одна форма внутреннего представления не может в равной степени удовлетворять всем предъявляемым требованием, что привело к созданию целого ряда форм внутреннего представления.

2. Классификация внутренних форм представления программ

Все внутренние формы представления программ разделяют на 3 класса [1]:

- высокоуровневые формы представления,
- формы представления среднего уровня и
- низкоуровневые формы представления.

2.1 Высокоуровневые формы представления программ

Высокоуровневые формы представления в большинстве случаев используются на ранних этапах процесса трансляции. Основное требование к высокоуровневой форме представления заключается в том, она должна адекватно отражать конструкции языков высокого уровня. Так, например, внутренняя форма представления высокого уровня может включать:

- условный оператор

```
if <выражение сравнения> then <операторы> [else <операторы>]
```

- оператор цикла со счетчиком, имеющий вид

```
for <переменная> <операнд> [by <операнд>] to <операнд>
      <операторы>
end
```

- операцию обращения к многомерному массиву

```
<имя массива>[<индекс>, …, <индекс>]
```

и т.д.

К высокоуровневым формам представления относят деревья грамматического разбора, а также *абстрактные синтаксические деревья* (англ. Abstract syntax tree - AST). Последняя форма представления весьма похожа на дерево грамматического разбора цепочки, которое может быть получено на этапе синтаксического анализа. Однако в абстрактных синтаксических деревьях опускаются многие несущественные детали синтаксиса языка, например, разделители (скобки, запятые и т.п.). Фактически каждая вершина абстрактного синтаксического дерева отвечает за некоторую конструкцию исходного языка.

Пример абстрактного синтаксического дерева для функции

```
int f(int a, int b)
{
```

```
    int c;
    c = a + 2;
    print(b, c);
}
```

представлен на рисунке 3 (пример взят из [1]).

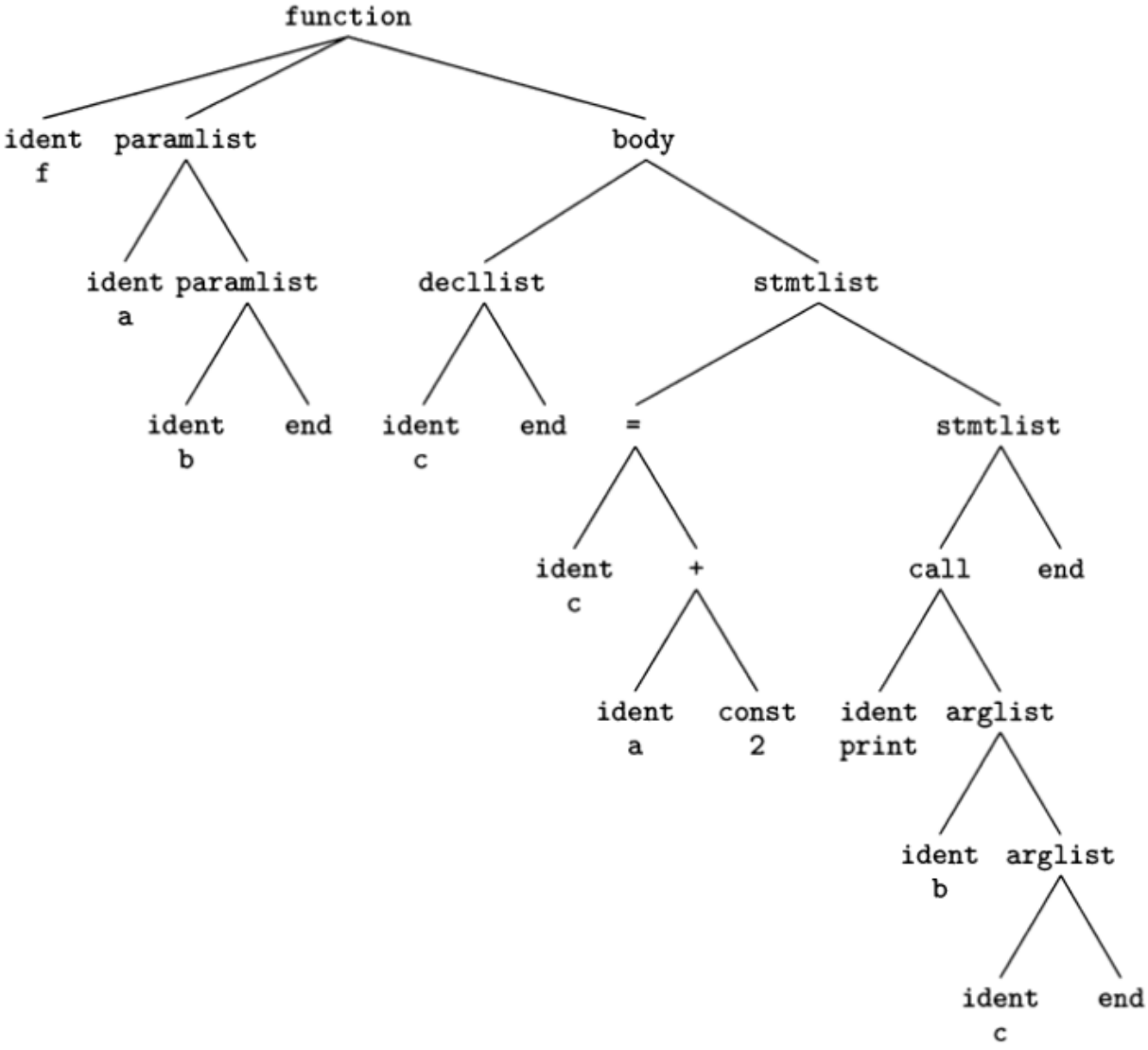

Рис.3 - Абстрактное синтаксическое дерево

Другим примером высокоуровневой внутренней формы представления программ является *направленный ациклический граф* (англ. directed acyclic graph - DAG). Такой граф не содержит циклов и удобен для анализа зависимостей в коде и устранения общих подвыражений.

Конечно, список высокоуровневых форм представления программ не ограничивается приведенными выше примерами. Важно понимать, что в любом случае высокоуровневые формы представления достаточно близки ко входным языкам и не ориентированы на какие-либо модели аппаратных платформ.

2.2 Формы представления программ среднего уровня

Формы представления среднего уровня разрабатываются таким образом, чтобы, с одной стороны, адекватно отображать особенности входного языка, а с другой стороны, допускать эффективную генерацию выходного кода для множества целевых языков (аппаратных платформ). Формы представления программ среднего уровня хорошо подходят для решения большей части задач по оптимизации программы.

В промежуточном языке среднего уровня, конечно, отсутствуют такие конструкции, как операторы цикла и операции обращения к многомерным массивам. Однако промежуточный язык среднего уровня может вместо условного оператора if-then-else включать оператор условного перехода в виде:

```
if <выражение сравнения> goto <метка>
```

поэтому цикл, который в случае высокоуровневой формы внутреннего представления мог быть записан в виде

```
for x a to b <операторы> end
```

в форме представления среднего уровня будет записан в виде

```
     x := a
     t1 := b
L1:  if (x>t1) goto L2
     <операторы>
     x:=x+1
     goto L1
L2:
```

Обращение к элементу массива

```
y:=x[i,j]
```

также будет выглядеть иначе:

```
t1:=i*10
t2:=t1+j
t3=addr(x)
t4:=t3+t2
y:=*t4
```

В качестве примеров промежуточного языка среднего уровня можно привести *тетрады и триады*, о которых пойдет речь чуть позднее.

2.3 Низкоуровневые формы представления программ

Низкоуровневые формы представления имеют набор инструкций, который, как правило, почти полностью соответствует инструкциям целевой платформы. Такие формы

внутреннего представления позволяют произвести максимально возможную оптимизацию кода. Генерация выходного кода по такой внутренней форме представления выполняется достаточно просто и может быть связана с генерацией набора команд, соответствующих одной инструкции внутренней формы представления, или, наоборот, объединением нескольких команд внутреннего представления в одну операцию целевой платформы.

С использованием промежуточного языка низкого уровня приведенный выше пример с обращением к элементу массива может выглядеть так:

```
r1:=[fp-1]
r2:=[fp-2]
r3:=r1*10
r4:=r3+r2
r5:=fp-102
r6:=r5+r4
r7:=[r6]
```

В качестве примера промежуточного языка низкого уровня можно привести язык RTL (register-transfer language), используемый в компиляторе gcc.

Следует отметить, что при построении транслятора использовать только строго один из уровней представления довольно сложно. Использование только высокоуровневой формы внутреннего представления с одной стороны накладывает ограничения на входные языки, ограничивая их класс, а с другой стороны требует больших усилий на создание транслятора с внутренней формы представления в целевой код. Использование только низкоуровневой формы представления накладывает ограничения на выходные языки, ограничивая их класс, а с другой стороны, требует больших усилий на создание транслятора с исходного кода во внутреннюю форму представления программы.

Одним из способов решения этой проблемы является использование внутренних форм представления, совмещающих в себе несколько уровней (многоуровневых). В этом случае в форме представления присутствуют как конструкции, характерные для высокоуровневых форм представления (например, циклы, индексация многомерных массивов), так и операции, характерные для форм представления более низкого уровня.

Другим способом является использование в трансляторе не одной, а нескольких форм внутреннего представления. В этом случае во время трансляции программа преобразуется из одной формы в другую, в зависимости от решаемой на данном этапе задачи.

3. Внутренние формы представления для регистровых и стековых вычислительных машин

При разработке внутренней формы представления программы за основу может быть взята некоторая модель вычислительной машины, и здесь можно выделить два основных направления. В первом из них за основу берется модель *регистровой ЭВМ* (или *ЭВМ с произвольным доступом к памяти*), во втором - *стековая машина*. Если в первом случае предполагается, что операции производятся с регистрами ЭВМ или непосредственно с памятью с произвольным доступом, то во втором – все операции производятся с содержимым стека, то есть при выполнении операции операнды извлекаются из вершины стека, и на их место помещается результат.

3.1 Трехадресный код. Тетрады и триады

В том случае, когда при разработке внутреннего представления программы за основу берется регистровая машина или модель ЭВМ с произвольным доступом к памяти, для формирования внутреннего представления часто используется так называемый *трехадресный код (тетрады)*. В нем каждая инструкция описывается как четверка:

(<оператор>,<операнд1>,<операнд2>,<результат>)

Операндами и результатом здесь могут являться переменные и константы, описанные в исходной программе или временные переменные, автоматически генерируемые компилятором. Сам оператор во внутренней форме представления должен входить в набор команд, поддерживаемый выбранной моделью ЭВМ. Главное здесь, что каждая команда выполняет ровно одну базовую операцию над аргументами, находящимися в регистрах (памяти) и помещает результат также в регистр (память).

С использованием тетрад, например, следующий фрагмент кода на C++, вычисляющий сумму элементов массива:

```
s = 0;
for (i = 0; i < 10; ++i) {
    s += ar[i];
}
```

может быть представлен следующим образом:

```
      (:=, 0, , s)       // s = 0
      (:=, 0, , i)       // i = 0
L1:   (<, i, 10, t0)     // t0 = i<10
      (JZ, t0, L2,)      // if (t0 == 0) goto L2
      (+, ar, i, t1)     // t1 = ar+i
```

```
      (**, t1, , t2)     // t2=*t1, разыменование
      (+, s, t2, s)      // s=s+t2
      (+, i, 1, i)       // i=i+1
      (JMP, L1, ,)       // goto L1
L2:
```

Основным недостатком тетрад является большое количество временных переменных, порождаемых при генерации внутренней формы представления программы. Частично разрешить эту проблему помогают так называемые *триады*, имеющие следующий вид:

(<оператор>,<операнд1>,<операнд2>)

В триадах отсутствует поле результата и если какая-либо операция должна будет воспользоваться результатом сгенерированной ранее тетрады, в качестве аргумента будет использоваться ссылка на сгенерированную ранее триаду. В качестве примера приведем пример для выражения a*b+c*d:

```
1:    (*, a, b)
2:    (*, c, d)
3:    (+,(1),(2))
```

Таким образом, тетрады и триады представляют собой довольно близкую по своей сути форму к объектному коду программы. Достоинствами тетрад и триад является относительная простота оптимизации внутренней формы представления программы, так как их можно легко переставлять и удалять лишние [5].

3.2 Польская инверсная запись

Внутреннее представление программы, при разработке которого за основу взята абстрактная стековая машина, в большинстве случаев основано на выполнении операций над аргументами, располагающимися в двух верхних позициях стека, а не в регистрах или памяти с произвольным доступом. Отметим, что код виртуальной стековой машины используется в качестве внутренней формы представления достаточно часто. Например, стековыми являются байт-коды виртуальной Java машины (JVM) и байт-код, используемый платформой .NET Framework (Common Intermediate Language - CIL).

Одним из наиболее известных способов внутреннего представления для выполнения стековой машиной является польская инверсная запись (ПОЛИЗ), известная также как постфиксная нотация. В отличие от обычной инфиксной формы записи в постфиксной нотации операции записываются после аргументов, а не между ними. Например, операции a + b и a := b, представленные в традиционной инфиксной форме, в постфиксной нотации примут вид: a b + и a b :=.

Операции в выражениях, представленных в постфиксной форме записи, выполняются в порядке записи слева-направо, приоритеты и ассоциативность операций не принимаются во внимание. Не используются в постфиксной форме записи и скобки.

Для вычисления вручную выражения, представленного в постфиксной форме записи, оно читается слева-направо. Как только в выражении встречается операция, она выполняется над двумя стоящими непосредственно перед ней операндами, после чего операнды и операция вычеркиваются из выражения и на их место вписывается результат выполнения операции. После этого выражение вычисляется дальше по тому же правилу, пока не будет получено единственное значение - результат вычисления выражения.

В качестве примера рассмотрим вычисление выражения (2+3)*7+4/2. Соответствующая ему польская инверсная запись имеет вид: 2 3 + 7 * 4 2 / +.

2 3 + 7 * 4 2 / +

5 7 * 4 2 / +

35 **4 2 /** +

35 2 +

37

Составить вручную ПОЛИЗ по инфиксной записи для небольших выражений не составляет большого труда. Для этого необходимо помнить, что операции в ПОЛИЗе всегда следуют в том порядке, в котором они должны выполняться, а операнды – в том же порядке, в каком они следовали в инфиксной записи. При этом операторы должны располагаться в ПОЛИЗе сразу же за своими операндами.

Следует отметить, что польская инверсная запись короче инфиксной формы записи. Это дает возможность уменьшить объём программ по сравнению с традиционной формой записи, что немаловажно для тех вычислительных устройств, где имеются жёсткие требования по использованию памяти.

Следует отметить, что с использованием постфиксной формы записи возможно описание не только арифметических операций, но и любых других операций [5], в частности операций управления потоком вычислений в программе. Так, можно ввести унарную команду безусловного перехода JMP и бинарную команду условного перехода по лжи (если значение первого аргумента ложно - имеет нулевое значение) JZ. Рассмотрим, как в этом случае в ПОЛИЗе будут выглядеть традиционные управляющие конструкции языков высокого уровня.

Условный оператор, имеющий вид

`if (<условие>) <оператор1> else <оператор2>;`

в ПОЛИЗе может быть представлен выражением:

```
<условие> <адр7> JZ   <оператор1> <адр2> JMP  <оператор2> …
адр1      адр2   адр3 адр4        адр5   адр6 адр7        адр8
```

Оператор цикла с предусловием, имеющий вид

```
while (<условие>) <оператор1>;
```

в ПОЛИЗе выглядит следующим образом:

```
<условие> <адр7> JZ   <оператор1> <адр1> JMP  …
адр1      адр2   адр3 адр4        адр5   адр6 адр7
```

Оператор цикла с постусловием, имеющий вид

```
do <оператор1> while (<условие>);
```

в ПОЛИЗе выглядит следующим образом:

```
<оператор1> <условие> NOT  <адр1> JZ   …
адр1        адр2      адр3 адр4   адр5 адр6
```

Для представления цикла

```
for (<выражение1>; <выражение2>; <выражение3>) <оператор>
```

достаточно вспомнить, что этот цикл эквивалентен циклу с предусловием:

```
<выражение1>;
while (<выражение2>) {
    <оператор>;
    <выражение3>;
}
```

Особое внимание следует уделить операции обращения по индексу. В том случае, когда допускается использование только одномерных массивов, такая операция в ПОЛИЗе может иметь вид

```
<имя_массива> <индекс> [
```

Если допустимо использование многомерных массивов операция принимает вид:

```
<имя_массива> <индекс1><индекс2>…<индексN> N [
```

В качестве примера приведем следующий фрагмент программы на языке С:

```
for (i=0; i<10; i++) {
    if (i<5)&&(j>0)  a[i] = j;
    else  a[i]=0;
}
```

Соответствующая приведенному выше фрагменту запись в ПОЛИЗе имеет вид:

```
i 0 := i 10 < 36 JZ i 5 <  j  0  >  AND 24 JZ a  i  [  j  :=
0 1 2  3 4  5 6  7  8 9 10 11 12 13 14  15 16 17 18 19 20 21
```

```
29 JMP a  i  [  0  := i  i  1  +  := 3  JMP
22 23  24 25 26 27 28 29 30 31 32 33 34 35  36
```

В заключение отметим, что в постфиксной записи недопустимо использование одних и тех же обозначений операций для унарных и бинарных операций, так как из контекста записи невозможно определить унарность или бинарность операции. По этой причине для унарных операций следует вводить новые обозначения, либо обеспечивать эмуляцию таких операций через соответствующие бинарные операции (например, унарную операция «-a» трактовать, как бинарную «0-a»).

Литература

1. Steven Muchnick Advanced Compiler Design and Implementation / Morgan Kaufmann, 1997, 856p.

2. Torben Mogensen Basics of Compiler Design. Anniversary edition / Department of computer science. University of Copenhagen, 2010, 319p. ISBN 978-87-993154-0-6

3. Компиляторы: принципы, технология / А. Ахо, Р. Сети, Дж. Ульман. - М.: Вильямс, 2003. - 767 с.

4. В.А.Серебряков, М.П.Галочкин Основы конструирования компиляторов http://citforum.ru/programming/theory/serebryakov/

5. Основные структуры данных и алгоритмы компиляции / М. А. Шамашов. - Самара : СМС, 1999. - 115 с.