9. Семантический анализ. Внутреннее представление

- Фаза контроля типов
- Идентификация. Работа с таблицами
- Идентификация. Работа с типами
- Причины использования промежуточных языков в компиляторах
- Различные формы промежуточных языков (ПЯ) в компиляторах:
  - Атрибутные деревья разбора
  - Прямая и обратная польские записи
  - Триады/тетрады
  - RTL

Лекция 9. Видозависимый анализ. Внутреннее представление

В данной лекции рассматриваятся следующие вопросы:

- Фаза контроля типов
- Идентификация. Работа с таблицами
- Идентификация. Работа с типами
- Причины использования промежуточных языков в компиляторах
- Различные формы представления промежуточных языков (ПЯ) в компиляторах:
  - Атрибутивные деревья разбора
  - Прямая и обратная польские записи
  - Триады/тетрады
  - RTL

Фаза контроля типов проверяет, удовлетворяет ли программа контекстным условиям. Главной составляющей контекстных условия является «правильное использование» программой типов данных, предоставляемых входным языком, т.е. корректность выражений, встречающихся в программе, с точки зрения использования типов. Данная задача включает, в частности, нахождение объявления в программе каждого используемого идентификатора, и проверку корректности его появления в использующем контексте.

Идентификация

- Одна из задач, решение которой необходимо для проверки правильности использования типов. Эта задача может быть полностью или частично решена на фазе синтаксического анализа.
- Определяющее вхождение идентификатора – это его вхождение в описание, например, int i
- Все остальные вхождения идентификатора являются использующими, например, i = 5 или i + 13

Идентификация

Идентификация идентификаторов – одна из задач, решение которой необходимо для проверки правильности использования типов.

Понятно, что мы не можем убедиться в правильности использования типов в какой-нибудь конструкции до тех пор, пока не определим типы всех ее составных частей. Например, для того, чтобы выяснить правильность оператора присваивания мы должны знать типы его получателя (левой части) и источника (правой части). Для того, чтобы выяснить, каков тип идентификатора, являющегося, например, получателем присваивания, мы должны понять, каким образом этот идентификатор был объявлен в программе.

Каждое вхождение идентификатора в программу является либо определяющим, либо использующим. Под определяющим вхождением идентификатора понимается его вхождение в описание, например, `int i`. Все остальные вхождения являются использующими, например, `i = 5` или `i+13`.

Цель идентификации идентификаторов, определить тип использующего вхождения идентификатора. Эта задача может быть полностью или частично решена на фазе синтаксического анализа. Все зависит от того, может ли использующее вхождение идентификатора встретиться в программе до определяющего вхождения или нет. Если все определяющие вхождения идентификаторов должны быть расположены текстуально перед использующими вхождениями, то мы можем выполнить идентификацию на фазе синтаксического анализа. Если же нет, то на фазе синтаксического анализа мы можем обработать определяющие вхождения идентификаторов и только на следующем просмотре текста программы выполнить собственно идентификацию.

Вне зависимости от того, на каком просмотре будет выполняться идентификация идентификаторов, при обработке определяющего вхождения идентификатора необходимо запомнить информацию о типе этого идентификатора. Это можно сделать несколькими путями:

- создавать узел в синтаксическом дереве для конструкции «описание идентификатора» и запоминать информацию о типе идентификатора в этом узле;

- создать *таблицу идентификаторов* (`IdTab`) и в ней запоминать информацию о типе идентификатора. Почему нам может потребоваться новая таблица? Понятно, что если транслируемая программа может иметь блочную структуру, и/или язык допускает создание и использование *перегруженных*[1] идентификаторов (*overloaded identifier*), то в *таблице представлений* (таблица представлений сопоставляет некоторому используемому в компиляторе обозначению идентификатора его представление в программе) информацию о типе идентификатора хранить нельзя, поскольку в этой таблице каждая лексема встречается только один раз. Таким образом, нам потребуется новая таблица для хранения информации об определяющих вхождениях идентификаторов.

На самом деле, между перечисленными способами много общего и мы остановимся на обсуждении второго способа.

[1] Под перегруженным идентификатором мы понимаем идентификатор, который является именем нескольких различных сущностей, например, в некоторых языках можно объявить две (или больше) процедур с одинаковым именем и различными параметрами. Проблема перегрузки идентификаторов аналогична проблеме перегрузки операторов.

Структура таблиц

- Таблица внешних представлений
  значение hash-функции для лексемы
  лексический класс
  лексическая марка
  ссылка на определение идентификатора
- Таблица идентификаторов
  ссылка на внешнее представление
  ссылка на тип
  ссылка на предыдущее определение идентификатора

Структура таблиц

Наша задача – разработать такую структуру таблицы идентификаторов, чтобы минимизировать поиск в ней. При этом следует помнить о том, что каждый идентификатор мы заносим в таблицу представлений, поэтому желательно иметь прямой доступ из таблицы представлений в таблицу идентификаторов для каждого идентификатора. Добавим к элементу таблицы представлений поле, которое будет содержать ссылку в таблицу идентификаторов, если находящаяся в этом элементе сущность является идентификатором. Элемент таблицы идентификаторов организуем так:

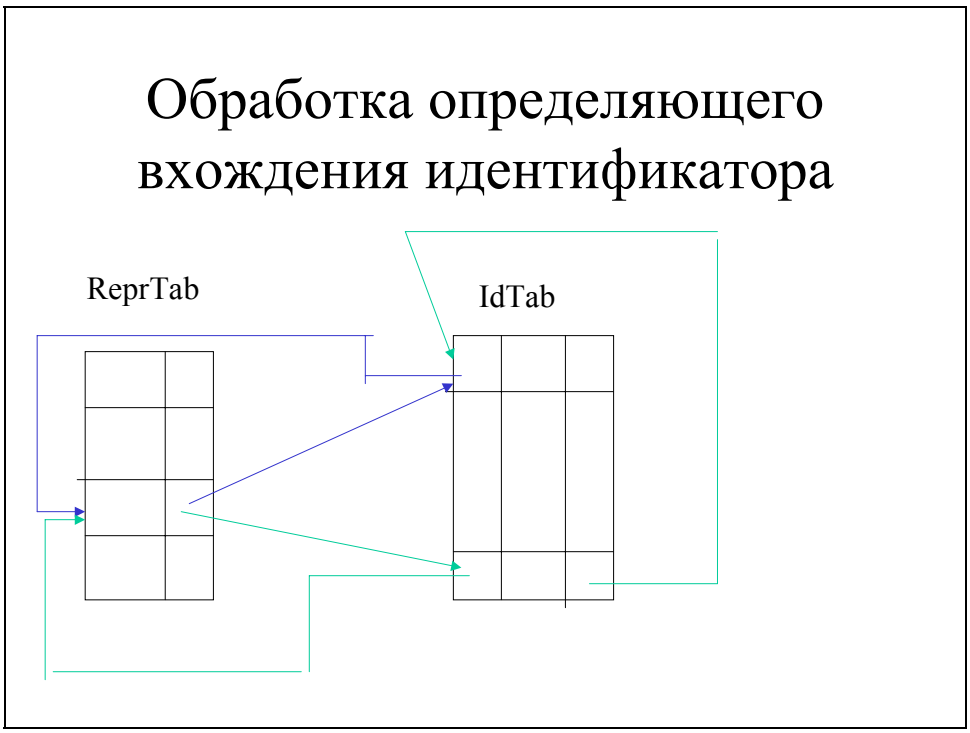
```
struct IdItem
{
      ReprInd toRepr;       /* ссылка в ReprTab на представление */
      TypeInd toMode;       /*  тип идентификатора */
      IdInd toId;           /* ссылка на элемент таблицы идентификаторов
                               с таким же значением поля toRepr   */
}
```

Расширим структуру таблицы внешних представлений следующим образом:

```
struct ReprItem
{
      HashInd toHash;            /* значение hash-функции для лексемы */
      unsigned  short LexClass;  /* лексический класс */
      unsigned  short LexMark;   /* лексическая марка */
      IdInd toId;                /*ссылка в IdTab на доступное определение
                                                  идентификатора */
}
```

Перейдем к формулировке алгоритма идентификации, который состоит из двух частей:

- первая часть алгоритма обрабатывает *определяющие* вхождения идентификаторов
- вторая часть алгоритма обрабатывает *использующие* вхождения идентификаторов



Обработа определяющего вхождения идентификатора

Обработка определяющего вхождения идентификатора происходит на фазе синтаксического анализа. Пусть лексический анализатор обработал очередную лексему, которая оказалась идентификатором. Лексический анализатор сформировал структуру типа `LEXEME`, которая содержит атрибуты выделенной лексемы, такие как ссылка в таблицу внешних представлений, лексический класс и лексическая марка. Далее вся эта информация передается синтаксическому анализатору. Предположим, что в данный момент синтаксический анализатор обрабатывает определяющее вхождение идентификатора. Таким образом, он знает ссылку в таблицу внешних представлений на обрабатываемый идентификатор и тип идентификатора. Основное семантическое действие, которое должен выполнить анализатор, заключается в занесении информации об идентификаторе в таблицу идентификаторов. Это происходит следующим образом:

- Создаем новый элемент таблицы идентификаторов
- В поле `toRepr` таблицы идентификаторов помещаем `rpr`.
- Поле `toId` элемента таблицы представлений помещаем в поле `toId` нового элемента таблицы идентификаторов. В поле `toId` элемента таблицы представлений помещаем ссылку на новый элемент таблицы идентификаторов

Отметим, что такая организация таблицы идентификаторов почти полностью исключает поиск в этой таблице, поскольку в нужный элемент таблицы мы попадаем по прямой ссылке из таблицы представлений.

Теперь обсудим обработку использующего вхождения идентификатора, которая происходит на фазе идентификации идентификаторов. Предположим, что уже построена (полностью или частично) таблица идентификаторов. Пусть лексический анализатор обработал очередную лексему, которая оказалась идентификатором. Лексический анализатор сформировал структуру типа `LEXEME`, которая содержит атрибуты выделенной лексемы, такие как ссылка в таблицу внешних представлений, лексический класс и лексическая марка. Далее вся эта информация передается фазе идентификации идентификаторов. Эта фаза знает, что она обрабатывает использующее вхождение идентификатора, причем ей известна помимо прочего ссылка в таблицу внешних представлений на обрабатываемый идентификатор. Теперь для того, чтобы получить информацию о типе идентификатора нам достаточно просто взять ее из того элемента

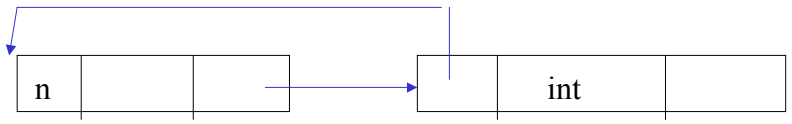
таблицы идентификаторов, на который указывает поле *toId* текущего элемента таблицы представлений.

Если исходный язык допускает блочную структуру программы, то мы должны при выходе из блока восстановить в таблице представлений ссылки на идентификаторы, объявленные в объемлющем блоке. Прежде всего при каждом входе в блок мы будем создавать в таблице идентификаторов специальный (информационный) элемент, который в поле `toRepr` будет содержать специальную марку BLOCK_BEGIN, а поле `toId` – ссылку на информационный элемент объемлющего блока. Таким образом, при выходе из блока мы должны просканировать все элементы таблицы идентификаторов вплоть до информационного элемента текущего блока для того, чтобы восстановить ссылки на идентификаторы объемлющего блока.

Пример

Рассмотрим программу следующей структуры:

{int n; …; n++; .. {float n; … n = 3.14; … } …n--; … }

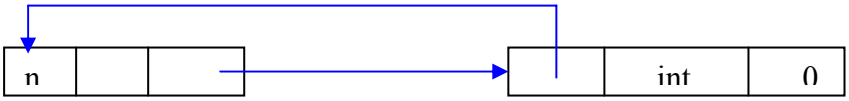


Пример

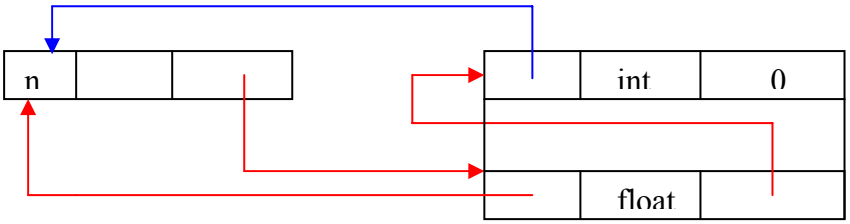
Рассмотрим программу следующей структуры:

```
{int n; …; n++; .. {float n; … n = 3.14; … } …n--; … }
```

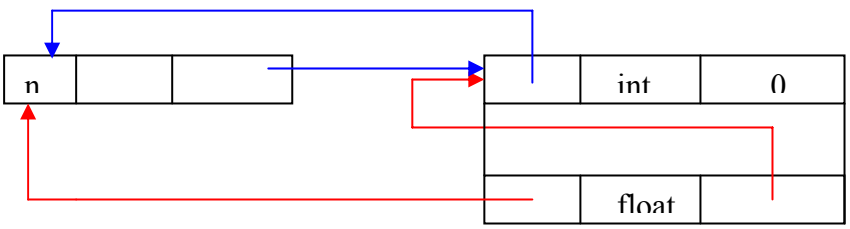
При входе во внешний блок идентификатор *n* будет занесен в таблицу представлений, кроме того, будет создан элемент таблицы идентификаторов.



После входа во внутренний блок будет добавлен еще один элемент в таблицу идентификаторов, соответствующий идентификатору *n*.



После выхода из внутреннего блока таблицы будут выглядеть следующим образом.



Конструирование типов

- Примитивные типы: *int*, *char*, *boolean, invalid*
- Основные конструкторы типов:
  array, *, *struct*, *pointer*, *proc*

Конструирование типов

Типы языков программирования конструируются из примитивных типов, таких как *boolean, char, integer, real* и *void*, с помощью конструкторов типов. К примитивным типам естественно отнести и тип, который не используется при программировании, но весьма полезен для сигнализации о возникшей ошибке в типах; это тип – *invalid.* Для построения более сложных типов из примитивных обычно используются следующие конструкторы:

a) Массивы. Если T – тип, то *array (I, T)* – тип, обозначающий тип массива с элементами типа T и индексным множеством I. Например, описание языка Pascal:

```
var A: array [1..10] of integer;
```

связывает выражение над типами *array (1..10, T)* с A .

b) Произведение. Если T_1 и T_2 - типы, то их декартово произведение $T_1 * T_2$ также является типом.

c) Структуры. Конструктор *struct* применяется к кортежу пар (*имя поля, тип поля).* Например, фрагмент программы на языке Pascal:

```
type item =
          record
                index: integer;
                word: array [1..15] of char
          end;

 var table: array [1..13] of item;
```

Тип *item* строится из примитивных типов следующим образом:
*struct ((index*integer) ×(word*array (1..15, char)))*.

d) Указатели. Если T – тип, то и *pointer (T)* – тип, определяющий “указатель на объект типа T”. Например, описание языка Pascal:

```
var p: ^item;
```

определяет переменную *p*, имеющую тип *pointer (type)*.

e) Функции. Если T_1, T_2 - типы, то *proc* (T_1, T_2) – тип, определяющий процедуру, типы формальных параметров которой есть T_1, а тип результата - T_2. Например, функция *mod,* вычисляющая остаток, имеет тип *proc (int *int, int)*,
а функция, определенная как

```
function f (a: integer, b: char) : ^integer;
```

имеет тип *proc (integer×char, pointer (integer))*.

Представление типов

- Деревья
- DAG
- Линейная запись деревьев или dag’ов.
- Битовые шкалы
- Пример.

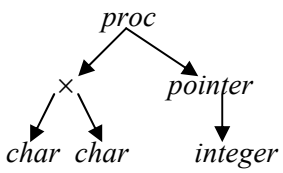
  proc (char, char) ^ integer

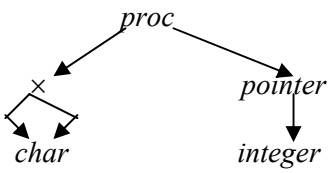
  Линейная запись деревьев

  proc, 2, char, char, pointer, integer

Представление типов

Удобным путем представления выражений над типами являются графы. Мы можем конструировать деревья или DAG'и (ориентированные ациклические графы), листьями которых будут примитивные типы. Например:





Использование dag’ов более предпочтительно, поскольку в этом случае происходит иногда весьма значительная экономия памяти (вместо самих типов в этом случае хранится ссылка на них).

Мы можем использовать и линейное представление деревьев или dag’ов, например,

proc, 2, m1, m2, m2,
где *m1* – указатель в таблицу на тип *pointer*, *integer*,
а *m2* – указатель в таблицу на тип *char*.

Рассмотрим еще один способ кодирования типов, который был использован в компиляторе C, разработанном Ричи (D.M.Ritchie). Ограничимся тремя конструкторами типов: указателями, функциями и массивами: *pointer (t)* обозначает указатель на тип *t*, *freturns (t)* обозначает функцию от некоторых аргументов, которая возвращает значение типа *t* и, наконец, *array (t)* обозначает массив некоторой неопределенной длины элементов типа *t.* Приведем примеры типов:

char
freturns (char)
pointer (freturns (char))
array (pointer (freturns (char))).

Каждый из этих типов может быть представлен последовательностью битов. Поскольку у нас есть только три конструктора типа, мы можем использовать для кодирования два бита:

pointer	01
array	10
freturns	11

Примитивные типы кодируются четырьмя битами:

boolean	0000
char	0001
integer	0010
real	0011

Используя такой способ кодирования, риведенные выше типы мы можем закодировать следующим образом:

char	000000 0001
freturns (**char**)	000011 0001
pointer (**freturns** (**char**))	000111 0001
array (**pointer** (**freturns** (**char**)))	100111 0001

Теперь вернемся к обсуждению структуры таблицы идентификаторов. Одно из полей (`toMode`) мы собирались использовать для определения типа идентификатора. Определив представление типов, которые могут появиться в программе, мы можем теперь зафиксировать, что поле `toMode` в – это либо указатель на дерево или dag, либо ссылка в таблицу типов `ModeTab`, хранящую линейное представление типов.

Контроль типов

- Статический контроль типов означает, что проверка правильности использования типов осуществляется во время трансляции исходной программы.
- Динамический контроль типов означает, что проверка правильности использования типов осуществляется во время выполнения объектной программы.

Контроль типов

Если контроль типов осуществляется во время трансляции программы, то мы говорим о *статическом контроле типов (static type checking)*, в противном случае, то есть если контроль типов производится во время исполнения объектной программы, мы говорим о *динамическом контроле типов (dynamic type checking)*. В принципе, контроль типов всегда может выполняться динамически, если в объектном коде вместе со значением будет размещаться и тип этого значения. Понятно, что динамический контроль типов приводит к увеличению размера и времени исполнения объектной программы и уменьшению ее надежности. Язык программирования называется *языком со статическим контролем типов или строго типизированным языком* (strongly typed language), если тип любого выражения может быть определен во время трансляции, то есть если можно гарантировать, что объектная программа выполняется без типовых ошибок. К числу строго типизированных языков относится, например, Pascal. Однако даже для такого языка как Pascal некоторые проверки могут быть выполнены только динамически. Например,

```
table: array [0..255] of char;
i: integer;
```

Компилятор не может гарантировать, что при исполнении конструкции `table[i]` значение `i` действительно будет не меньше нуля и не больше 255. В некоторых ситуациях осуществить такую проверку может помочь техника, подобная data flow analysis, но далеко не всегда. Понятно, что на самом деле этот пример демонстрирует ситуацию общую для большинства языков программирования, то есть здесь речь идет о контроле индексов вырезки. Конечно, почти всегда такая проверка выполняется динамически.

Эквивалентность типов

- *Структурная эквивалентность типов (Structural equivalence of types)*
- *Именная эквивалентность типов (Named equivalence of types)*

Эквивалентность типов

Необходимой частью контроля типов является проверка *эквивалентности типов (equivalence of types)*. Крайне необходимо, чтобы компилятор выполнял проверку эквивалентности типов быстро.

Структурная эквивалентность типов (Structural equivalence of types) . Два типа называются эквивалентными, если они являются одинаковыми примитивными типами, либо они были сконструированы применением одного и того же конструктора к структурно эквивалентным типам. Иными словами, два типа структурно эквивалентны тогда и только тогда, когда они идентичны. Проверить, являются ли два типа структурно эквивалентными, можно следующей процедурой:

```
bool sequiv (s, t)
{
       if (s и t – два одинаковых примитивных типа)
       {
              return true;
       }
       else if (s == array (s1, s2) && t == array (t1, t2))
       {
              return sequiv (s1, t1) && sequiv (s2, t2);
       }
       else if (s == s1*s2 && t == t1*t2)
       {
              return sequiv (s1, t1) && sequiv (s2, t2);
       }
       else if (s==pointer (s1) && t == pointer (t1))
       {
              return sequiv (s1, t1);
       }
       else if (s==proc (s1, s2) && t == proc (t1, t2))
       {
              return sequiv (s1, t1) && sequiv (s2, t2);
       }
       else
       {
              return false;
       }
}
```

В некоторых языках типам можно давать имена, которые иногда называют индикантами типа. Рассмотрим пример программы на языке Pascal:

```
type link = ^cell;
var next: link;
       last: link;
       p: ^cell;
       q, r: ^cell;
```

Возникает вопрос, одинаковые ли типы имеют описанные переменные? К сожалению, ответ зависит от реализации, поскольку в определении языка Pascal не определено понятие “идентичные типы”. В принципе здесь возможны две ситуации. Одна из них связана со структурной эквивалентностью типов. С этой точки зрения все объявленные переменные имеют одинаковый тип.

Второй подход связан с понятием *эквивалентности имен* (name equivalence). В этом случае каждое имя типа рассматривается как уникальный тип, таким образом, два имени типов эквивалентны, если они идентичны. При таком подходе переменные `p`, `q`, `r` имеют одинаковый тип, а переменные `p` и `next` – нет. Обе эти концепции используются в различных языках программирования. Например, Algol 68 поддерживает структурную эквивалентность.

Проблемы, возникающие в Pascal’e, связаны с тем, что многие реализации связывают с каждым определяемым идентификатором неявное имя типа. Таким образом, приведенные объявления некоторыми реализациями могут трактоваться следующим образом:

```
type link = ^cell;
       np = ^cell;
       npg = ^cell;
var next: link;
       last: link;
       p: np;
       q, r: npq;
```

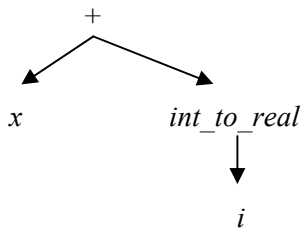
Преобразование типов

- Неявные преобразования типов или приведения
- Явные преобразования типов

Преобразование типов

Рассмотрим формулу *x+i,* где *x* – вещественная переменная, *i* – целая переменная. Так как представления целых и вещественных чисел в памяти компьютера различны, различные команды используются для целых и вещественных значений, и обычно нет команд, операндами которых являются значения смешанных типов, то компилятор должен преобразовать один из операндов к типу другого.

Описания языков определяют, какие преобразования возможны и необходимы. Если целое значение присваивается вещественной переменной или наоборот, выполняется преобразование к типу получателя присваивания. Хотя преобразование вещественного значения в целое, вообще говоря, некорректно. В формуле обычно выполняется преобразование целого операнда к вещественному типу. Фаза контроля типов вставляет эти операции преобразования в промежуточное представление исходной программы. Например, для формулы *x+i* после фазы контроля типов будет получено следующее дерево:



Преобразование типов называется *неявным* (*implicit conversion*), если они выполняются компилятором автоматически. Неявные преобразования или *приведения* (*coercions*), во многих языках ограничиваются такими ситуациями, когда никакая информация не теряется при преобразовании, например, целое может быть преобразовано в вещественное, обратное преобразование крайне не желательно.

Преобразование называется *явным* (*explicit conversion*), если программист должен написать что-нибудь для того, чтобы это преобразование было выполнено. Явные преобразования подобны вызовам функций, определенных над типами. В языке Pascal встроенная функция *ord* отображает литеру в целое, а функция *chr* выполняет обратное преобразование. Язык C приводит, т.е. преобразует неявно, ASCII литеры в целое в арифметических формулах.

Роль промежуточных языков в компиляторе

- Традиционная организация компилятора:
  Front-end — intermediate language — Back-end
- Промежуточный язык иногда также называют внутренним представлением программы
- Основная задача ПЯ – обеспечить интерфейс между анализом и синтезом (в однопроходных компиляторах можно вообще обойтись без ПЯ)
- ПЯ должен быть удобным для проведения на нем преобразований, оптимизаций и т.п.

Роль промежуточных языков в компиляторе

Большинство современных компиляторов – многопросмотровые; даже те языки программирования, которые теоретически могли бы быть скомпилированы за один проход (например, Pascal) чаще всего анализируются в несколько просмотров для того, чтобы улучшить качество генерируемого кода или упростить написание самого компилятора. Для связи между различными просмотрами компилятора используется некоторый *промежуточный язык (ПЯ)*, который иногда также называют внутренним представлением программы в компиляторе. По ходу компиляции внутреннее представление нагружается различной дополнительной информацией, полученной во время выполнения просмотров.

Как мы уже говорили, транслятор условно разделяют на две логические части: front-end (внешний интерфейс) и back-end (внутренний интерфейс), по степени приближенности к исходному или целевому языку компиляции. Таким образом, промежуточный язык можно воспринимать как интерфейс между анализом и синтезом программы. Поэтому ПЯ должен отражать функциональность исходного языка программирования и обеспечивать удобство выполнения основных задач синтеза, таких как оптимизация программы и генерация эффективного объектного кода.

Идея массового применения ПЯ

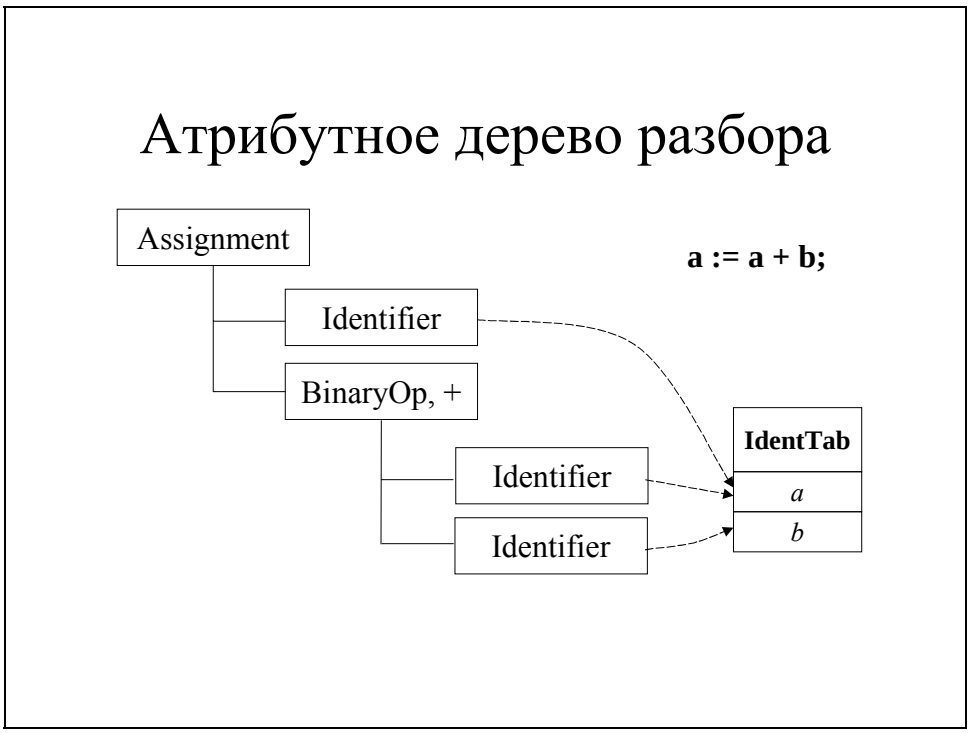
- Один и тот же ПЯ может быть использован сразу в нескольких компиляторах:
  - При массовом написании компиляторов (m входных языков для n целевых платформ) унификация ПЯ может привести к уменьшению трудоемкости с m*n компиляторов до m+n
  - Java bytecode реализует частный случай с одним входным языком и множеством целевых платформ
  - MSIL представляет собой случай с множеством языков и множеством платформ
  - При едином ПЯ дорогостоящие в реализации машинно-независимые оптимизации могут быть использованы многократно (например, GNU C)

Идея массового применения промежуточного языка

Задача проектирования ПЯ особенно сложна при проектировании многоязыковых систем трансляции, позволяющих генерировать код сразу для нескольких целевых платформ. В этом случае становится выгодно спроектировать единый промежуточный язык для всего семейства трансляторов. Таким образом, можно свести задачу написания *m*n* компиляторов к реализации разбора *m* входных языков с построением в процессе анализа единого внутреннего представления и последующему написанию *n* просмотров, синтезирующих объектный код для *n* целевых платформ.

Эта идея была известна уже более 30 лет назад (UNCOL, Warren Abstract Machine, Виртовский p-code, АЛЬФА и т.д.), но лишь недавно такой подход стал широко применяться на практике, причем в основном совместно с идеей динамической компиляции:

- Java bytecode представляет собой подход, в котором один входной язык проецируется сразу на множество целевых платформ с помощью единого промежуточного языка. Собственно синтез объектного кода возлагается на реализаторов Java Virtual Machine для данной платформы. Интересно, что несмотря на тот факт, что Java bytecode создавался только для одного входного языка, впоследствии появилось множество компиляторов, генерирующих Java bytecode для других исходных языков (например, Кобол).
- MSIL представляет собой более общий случай той же идеи, с множеством входных языков и множеством целевых платформ. При этом синтез машинного кода выполняется .NET runtime во время выполнения программы.
- Возможно, наиболее "чистой" реализацией идеи единого промежуточного языка является семейство компиляторов GNU, в котором единое внутреннее представление GNU RTL (RTL расшифровывается как Register Transfer Language) может порождаться из целого ряда языков (C/C++, Fortran, Ада, CHILL) и впоследствии может быть использовано для генерации объектного кода сразу для целого ряда платформ. GNU RTL также позволяет писать независимые просмотры оптимизации, причем наличие информации об относительной стоимости выполнения различных операций позволяет задавать как машинно-независимые, так и машинно-зависимые оптимизации.



Атрибутное дерево разбора

Атрибутное дерево разбора является, наверное, самой распространенной формой организации внутреннего представления программы. При таком подходе каждая исходная конструкция языка представляется в виде узла дерева, содержащего ссылки на все возможные элементы этой конструкции (естественно, каждый отдельный элемент тоже может иметь сложную структуру и, таким образом, также может быть поддеревом). Кроме того, каждый узел дерева может нагружаться дополнительными атрибутами, такими, как ссылки в таблицы представлений или таблицы идентификаторов. В итоге, вся программа представляется в виде единого дерева разбора.

На слайде в качестве примера приведено атрибутное дерево разбора, порожденное по оператору *a := a + b;* исходного языка. Отметим, что форма представления дерева, использованная на слайде, является типичной при компиляции, так как позволяет изобразить практически сколь угодно сложные деревья на экране компьютера (попробуйте представить себе традиционное изображение дерева разбора для сколько-нибудь сложной программы!).

Деревья разбора привлекательны прежде всего своей гибкостью и возможностью использования в самых разных этапах компиляции – их можно спроектировать таким образом, чтобы они мало зависели от исходного языка и целевой платформы. Деревья разбора легко строить во время анализа исходной программы, а все последующие просмотры компилятора могут быть реализованы в виде самостоятельных обходов этого дерева. Кроме того, некоторые просмотры, такие, как оптимизация программы, удобнее всего выполнять именно над деревьями разбора.

Польская запись

- Прямая (префиксная) польская запись:
  `(a+b)*(c-d) -> *+ab-cd`
- Обратная (постфиксная) польская запись:
  `(a+b)*(c-d) -> ab+cd-*`
- Польская запись может быть распространена и на другие конструции исходного языка

Польская запись

Польская запись была предложена польским логиком Лукасевичем. В этой форме записи все операторы непосредственно предшествуют операндам. Так, обычное выражение *(a+b)*(c–d)* в польской записи может быть представлено как **+ab–cd*.

Такую форму записи называют также префиксной. Аналогичным образом вводится *обратная* или *постфиксная польская запись*, в которой все операторы выписываются после операндов. Скажем, пример, приведенный выше, в обратной польской записи будет записан следующим образом: *ab+cd–**. Для представления унарных операций в польской записи можно воспользоваться эквивалентными выражениями, использующими бинарные операции, как в следующем примере: $-b \rightarrow 0 - b$, а можно ввести новый знак операции, скажем, *@b*. Польская запись может быть распространена не только на арифметические выражения, но и на прочие конструкции языка. Например, оператор *a := a + b;* может быть записан в польской записи как *:=a+ab*, а условный оператор **if** <expr> **then** <$instr_1$> **else** <$instr_2$> может быть записан как следующая последовательность операторов:
<expr> <c_1> BZ <$instr_1$> <c_2> BR <$instr_2$>,
где c_1 указывает на первую инструкцию <$instr_2$>, а c_2 – на первую инструкцию, следующую за <$instr_2$>, BR – безусловный переход на адрес <c_2>, а BZ – переход на <c_1> при условии равенства нулю выражения <$expr_1$>.

Пользуясь такой терминологией, мы можем называть традиционную форму записи выражений *инфиксной*, так как в ней знаки операций расположены между операндами. Понятно, что любое выражение может быть переведено из инфиксной формы в польскую запись и наоборот. Польская запись замечательна тем, что при ее использовании исчезает потребность в приоритетах операций – каждая операция выполняется в порядке появления в исходной цепочке (хотя очевидно, что приоритет операций необходимо учитывать при преобразованиях из инфиксной формы).

Польская запись (особенно, обратная) очень хорошо накладывается на стековую модель: каждый встреченный операнд загружается в стек, а операции производятся только на вершине стека: каждая операция снимает необходимое количество операндов с вершины стека и кладет на стек свой результат. Именно такая модель используется в MSIL для реализации большинства операций.

Триады и тетрады

Триады и *тетрады* представляют собой низкоуровневые формализмы записи промежуточного представления программы, приближающие программу к объектному коду. В этих формализмах все операции записываются в виде последовательности действий, выдающих результаты.

Тетрады (также называемые "четверками" или трехадресным кодом) состоят из двух операндов, разделенных операцией, и результата операции, записываемого с помощью равенства и обозначаемого целым числом (см. пример на слайде). Это целое число является номером временной переменной, в которую записывается результат операции. Таким образом, тетрады содержат явную ссылку на результат операции. В каком-то смысле, это может считаться недостатком тетрад, так как при прямолинейной генерации кода приходится порождать по одной временной переменной на каждую операцию в программе.

Триады (также называемые "тройками" или двухадресным кодом) построены аналогичным образом, но не содержат явного указания на результат операции, хотя на эти результаты по-прежнему можно ссылаться в последующих командах. Подразумевается, что задачу отслеживания и нумерации всех триад выполняет сам компилятор. Понятно, что триады компактнее тетрад, но с другой стороны, отсутствие явного указания на результат операции может затруднить фазу оптимизации. Эту проблему можно решить путем использования *косвенных триад*, в которых вместо ссылки на ранее использовавшуюся триаду используется ссылка на элемент специальной таблицы указателей на триады.

Естественно, триады и тетрады также могут быть расширены для записи всех операций, поддерживаемых на данной целевой платформе.