Лекция 4. Низкоуровневый императивный язык программирования (НУИЯП)

Коновалов А.В.

20 марта 2024 г.

Синтаксис и семантика НУИЯП

Низкоуровневый императивный язык программирования (1)

НУИЯП (низкоуровневый императивный язык программирования) — это модельный язык программирования, трансляцию которого в модельный ассемблер мы будем рассматривать в нашем курсе.

Это промежуточный язык, который формируется на выходе стадии анализа и на входе стадии синтеза некоторого вымышленного компилятора.

Предполагаем, что в программах на исходном языке нет ошибок, т.к. ошибочные программы должны быть отвергнуты на стадии анализа исходного текста, генерация кода для них вызываться не должна.

По духу язык будет напоминать K&R Си (т.е. до стандартизации 1989 г.), а по синтаксису — Scheme.

Низкоуровневый императивный язык программирования (2)

Синтаксис НУИЯП мы будем описывать грамматикой типов на языке Рефал-5, программа будет считываться функцией `LoadExpr`.

Пример кода на НУИЯП:

```
(function gcd (x y)
  (var (rem 1))
  (while ((L y) "<>" 0)
    (rem "=" ((L x) "%" (L y)))
    (x "=" (L y))
    (y "=" (L rem))
  )
  (return (L y))
)
```

Формальный синтаксис языка (1)

```
e.Program ::= t.Definition*
t.Definition ::= t.Struct | t.Const | t.GlobalVar | t.Function
t.Struct ::= (struct s.Name (s.Name t.ConstExpr)*)
t.Const ::= (const s.Name "=" t.ConstExpr)
t.GlobalVar ::= (var s.Name t.ConstExpr e.Init?)
e.Init ::= "=" t.ConstExpr*
s.Name ::= s.WORD
```

Формальный синтаксис языка (2)

```
t.Function ::= (function s.Name (s.Name*) t.LocalVars? e.Code)
t.LocalVars ::= (var (s.Name t.ConstExpr)*)

e.Code ::= t.Statement*
t.Statement ::=
    (t.Expr "=" t.Expr)
  | (call t.Expr t.Expr*)
  | (return t.Expr)
  | (if t.BoolExpr e.Code)
  | (if t.BoolExpr e.Code else e.Code)
  | (while t.BoolExpr e.Code)
  | (asm s.ANY+)
```

Формальный синтаксис языка (3)

```
t.Expr ::=
    s.Name
  | s.NUMBER
  | (L t.Expr)
  | ("-" t.Expr)
  | (t.Expr s.BinOp t.Expr)
  | (call t.Expr t.Expr*)
  | (asm s.ANY+)
s.BinOp ::= "+" | "-" | "*" | "/" | "%" | "&" | "|" | "~"

t.BoolExpr ::=
    TRUE | FALSE
  | (t.Expr s.RelOp t.Expr)
  | (not t.BoolExpr)
  | (t.BoolExpr and t.BoolExpr)
  | (t.BoolExpr or t.BoolExpr)
s.RelOp ::= "<" | ">" | "=" | "<>" | "≥" | "≤"
```

Формальный синтаксис языка (4)

```
t.ConstExpr ::=
    s.Name
  | s.NUMBER
  | ("-" t.ConstExpr)
  | (t.ConstExpr s.BinOp t.ConstExpr)
s.BinOp ::= "+" | "-" | "*" | "/" | "%"
```

Значения константных выражений можно вычислить во время компиляции.

В качестве имён в константных выражениях могут использоваться только имена констант, структур и полей структур.

Имена в программах на НУИЯП (1)

Основным типом данных в НУИЯП является, как и в модельном ассемблере, слово.

Имена `s.Name` являются синонимами чисел.

Некоторые числа известны во время компиляции:

- Имя константы является синонимом числа, заданного в определении константы.
- Имя структуры является синонимом размера этой структуры.
- Имя поля структуры является синонимом смещения поля относительно начала структуры.

Имена в программах на НУИЯП (2)

Значения имён глобальных переменных и функций становятся известны только во время ассемблирования — в сгенерированном коде они представляются в виде меток.

Значения имён локальных переменных становятся известны во время выполнения — в сгенерированном коде их адрес вычисляется относительно регистра `FP`: `GETFP offset ADD` или `GETFP offset SUB`.

Использование в выражениях неопределённого имени ошибкой не является — подразумевается обращение к внешнему определению из другой единицы трансляции, т.е. имя компилируется как глобальная переменная или функция.

Конструкции верхнего уровня

```
e.Program ::= t.Definition*
t.Definition ::= t.Struct | t.Const | t.GlobalVar | t.Function
t.Struct ::= (struct s.Name (s.Name t.ConstExpr)*)
t.Const ::= (const s.Name "=" t.ConstExpr)
t.GlobalVar ::= (var s.Name t.ConstExpr e.Init?)
e.Init ::= "=" t.ConstExpr*
s.Name ::= s.WORD
```

Программа представляет собой последовательность определений структур, констант, глобальных переменных и функций.

Определения констант и структур код не порождают. Определения глобальных переменных и функций порождают код, начинающийся с метки

```
'_' <Explode s.Name>
```

Константы

```
t.Const ::= (const s.Name "=" t.ConstExpr)
```

Компилятор вычисляет константное выражение и связывает его с именем константы, код не порождается.

Пример.

```
(const SIZE "=" 100)
```

Структуры (1)

```
t.Struct ::= (struct s.Name t.Field*)
t.Field ::= (s.Name t.ConstExpr)
```

В определении структуры указывается имя структуры и её поля. Поле описывается как пара из имени поля и его размера, задаваемого константным выражением.

Имя структуры становится синонимом её размера.

Поля структуры являются глобальными константами, т.е. в разных структурах не может быть полей с одинаковым именем.

Значением имени поля константы является её смещение относительно начала структуры. Очевидно, что значением имени первого поля всегда будет `0`.

Структуры (пример)

```
(struct Point
  (Point_x 1)
  (Point_y 1)
)

(struct Line
  (Line_start Point)
  (Line_end Point)
)
```

Имя	Значение
Point	2
Point_x	0
Point_y	1
Line	4
Line_start	0
Line_end	2

Здесь размеры полей `Point_x` и `Point_y` заданы целыми числами, размеры полей `Line_start` и `Line_end` именем константы `Point`, равной `2`.

Глобальные переменные (1)

```
t.GlobalVar ::= (var s.Name t.ConstExpr e.Init?)
e.Init ::= "=" t.ConstExpr*
```

Константное выражение определяет размер этой переменной. Глобальная переменная компилируется в метку, за которой перечисляются начальные значения слов этой области памяти.

Если инициализатор не указан, память заполняется нулями. Если указан — соответствующими значениями. Если указаны не все начальные значения, остальные заполняются нулями.

Глобальные переменные (2) (пример)

Код

```
(var Counter 1)
(var Items 5 "=" 1 2 3 4 5)
(var rising Line "=" 0 0 10 10)
(var result_array 4)
```

скомпилируется в

```
:_Counter
0
:_Items
1 2 3 4 5
:_rising
0 0 10 10
:_result_array
0 0 0 0
```

Глобальные переменные (3)

Можно считать, что определение вида

```
(var X N)
(var Y M "=" C1 C2 C3)
```

эквивалентно на языке Си

```
int X[N];
int Y[M] = { C1, C2, C3 };
```

Глобальные переменные (4)

Массив структур можно определить следующим образом:

```
(const SIZE 10)
(struct Point (x 1) (y 1))
(var points (Point "*" SIZE))
```

Будет выделена память (доступная под меткой `_points` в ассемблере) размером 20 слов. Мы записали константное выражение `(Point "*" SIZE)`, где перемножили размер структуры на количество элементов.

Глобальные переменные (5)

Инициализаторы структур также могут включать имена глобальных переменных и функций — переменная будет проинициализирована указателем.

Код	скомпилируется в
`(var Counter 1)`	`:_Counter` `0`
`(var pointers 2 "=" Counter gcd)`	`:_pointers` `_Counter _gcd`
`(function gcd (x y) …)`	`:_gcd` `…`

Обращение к полям структур

```
(struct Point (Point_x 1) (Point_y 1))
(struct Line (Line_start Point) (Line_end Point))
(var l Line)
```

Переменная `l` ссылается на кусок памяти размером 4 слова. Для обращения `l.end.y` (в терминах Си) нужно будет к адресу переменной (а имя переменной — синоним адреса) прибавить сначала смещение поля `Line_end`, потом — `Point_y`:

```
((l "+" Line_end) "+" Point_y)
```

или

```
(l "+" (Line_end "+" Point_y))
```

Значением выражения в обоих случаях будет адрес поля `Point_y` поля `Line_end` структуры `l`.

И ещё про структуры (1)

```
(struct Point
  (Point_x 1)
  (Point_y 1)
)

(struct Line
  (Line_start Point)
  (Line_end Point)
)

(var count 1)
(var bar Line)
```

- Для переменных и полей структуры указывается их размер.
- Имя структуры является синонимом для её размера.
- Запись вида `(Line_start Point)` можно интерпретировать не только как указание размера поля `Line_start` со структуру `Point`, но и как указание типа (хотя типы НУИЯП не проверяет).

И ещё про структуры (2)

```
(const int 1)
(const ptr 1)

(struct Point
  (Point_x int)
  (Point_y int)
)

(struct ListNode
  (ListNode_value int)
  (ListNode_next ptr)
)

(var count int)
(var list_head ptr)
```

- Можно определить константы `int` и `ptr` со значением 1 и использовать их для обозначения слов, предназначенных для хранения чисел и указателей.
- НИУЯП нетипизированный язык программирования, ошибок в использовании этих полей проверить, конечно же, не сможет.
- Но такой приём позволит повысить выразительность программ.
- Далее будем предполагать, что единичные константы `int` и `ptr` определены.

И ещё про структуры (3)

Разрешим в качестве имён полей структур использовать имя `"-"`. Поле с данным именем будет считаться безымянным — соответствующей константы компилятор создавать не будет, однако размер резервироваться будет. Пример:

```
(struct Point
  (Point_x int)
  (Point_y int)
)

(struct ColorPoint
  ("-" Point)
  (ColorPoint_color int)
)
```

Здесь определена структура `ColorPoint` размером `3`, смещение поля `ColorPoint_color` равно `2`. Можно считать, что мы одну структуру унаследовали от другой.

Безымянные поля пригодятся при реализации ООП (для поля виртуальной таблицы).

Расширение синтаксиса — автоматический размер переменной

Расширения синтаксиса НУИЯПа в дальнейшем (например, в заданиях на лабораторные работы) мы будем обозначать так:

```
‹конструкция› ::= …
  | ‹новый вариант›
```

В качестве примера рассмотрим переменные без явного указания размера:

```
t.GlobalVar ::= …
  | (var s.Name "=" t.ConstExpr+)
```

Фактически, это переменная с явно заданным инициализатором, её размер определяется количеством константных выражений после знака "=":

```
(var primes "=" 2 3 5 7 11 13 17 19 23 29 31)
```

Работа со строками

Заметим, что в синтаксисе НУИЯПа нигде не используются символы-литеры (`s.CHAR`), поэтому если в переменной `e.Program` находится корректная программа, то можно безопасно выполнить `<Ord e.Program>`.

Можно разрешить использовать в исходном тексте символы-литеры и применять функцию `Ord` к загруженному исходному тексту. Литеры в такой интерпретации будут эквивалентны макроцифрам с соответствующим кодом ASCII или Юникода в зависимости от реализации Рефала.

Запись

```
(var hello "=" 'Hello, World!' 0)
```

эквивалентна

```
(var hello "=" 72 101 108 108 111 44 32 87 111 114 108 100 33 0)
```

Компиляция НУИЯПа

Арифметические выражения (1)

```
t.Expr ::=
    s.Name
  | s.NUMBER
  | (L t.Expr)
  | ("-" t.Expr)
  | (t.Expr s.BinOp t.Expr)
  | (call t.Expr t.Expr*)
  | (asm s.ANY+)
s.BinOp ::= "+" | "-" | "*" | "/" | "%" | "&" | "|" | "~"
```

Арифметическое выражение — конструкция языка, которая вычисляет значение. Код, в который компилируется арифметическое выражение, после себя на стеке оставляет его значение.

Арифметические выражения (2)

Имена `s.Name` компилируются в соответствующие значения.

Если это имя локальной переменной — она компилируется код, вычисляющий адрес этой переменной: `GETFP смещение ADD` (для параметров функции) или `GETFP смещение SUB` (для локальных переменных).

Если это константа (`const`, имя структуры или имя поля структуры) — компилируется просто целое число, значение этой константы.

Иначе имя считается именем глобальной переменной/функции и компилируется в метку (начиная на знак _).

Арифметические выражения (3)

Число `s.NUMBER` компилируется в соответствующее число.

Операция **разыменования** (`L t.Expr`) компилируется в код, вычисляющий значение `t.Expr` (оно должно вычислять некоторый адрес в памяти) и инструкцию `LOAD`, которая снимает со стека этот адрес и кладёт на стек значение из памяти:

```
‹код для выражения t.Expr› LOAD
```

Аналогично компилируется операция смены знака (`"-" t.Expr`) — в команду `NEG`:

```
‹код для выражения t.Expr› NEG
```

Арифметические выражения (4)

Двуместная операция

```
(t.LeftExpr s.BinOp t.RightExpr)
```

компилируется следующим образом:

- сначала генерируется код для `t.LeftExpr`
- затем для `t.RightExpr`
- затем код соответствующей двуместной инструкции.

```
‹код для t.LeftExpr›
‹код для t.RightExpr›
‹операция для s.BinOp›
```

Арифметические выражения (5)

Операция вызова функции имеет вид:

```
(call t.Expr t.Arg1 ... t.ArgN)
```

Первым аргументом является выражение, вычисляющее адрес функции (как правило, просто имя функции), последующие аргументы — значения фактических аргументов функции.

Компилятор сначала должен скомпилировать аргументы в обратном порядке (т.е. будем пользоваться соглашением языка Си), затем — выражение `t.Expr`, затем инструкцию `CALL`.

Предполагается, что вызванная функция сама удалит свои аргументы и возвращаемое значение поместит в регистр `RV`.

Возвращаемое значение нужно поместить на стек.

Арифметические выражения (6)

Ассемблерная вставка имеет вид:

```
(asm s.ANY+)
```

Содержимое ассемблерной вставки как есть записывается в целевой код, занимая отдельную строчку текста. Компилятор её никак не проверяет её содержимое.

При написании ассемблерных вставок внутри выражений программист должен гарантировать, что результатом её работы будет одно новое слово на стеке, иначе нарушатся инварианты, которые гарантирует компилятор.

Арифметические выражения (7)

Пусть значения констант (включая имена и поля структур)
и смещения локальных переменных хранятся где-то в копилке.

И пусть у нас есть функции

```
<GetConst s.Name> == Found s.NUMBER | NotFound
<GetLocal s.Name> == Found { '+' | '-' } s.NUMBER | NotFound
```

Реализацию их рассматривать не будем.

Арифметические выражения (8)

Тогда генерацию кода для выражений можно описать так:

```
/*
  <GenExpr t.Expr> == e.AsmCode
  e.AsmCode ::= { s.WORD | s.NUMBER | s.CHAR }*
*/
GenExpr {
  s.Number, <Type s.Number> : 'N' e.1 = s.Number;

  ...
}
```

Арифметические выражения (9)

```
GenExpr {
   ...

  s.Name, <GetConst s.Name> : Found s.Value = s.Value;

  s.Name
    , <GetLocal s.Name> : Found s.Sign s.Offset
    , ('+' ADD) ('-' SUB) : e.1 (s.Sign s.Operation) e.2
    = GETFP s.Offset s.Operation;

  s.Name = '_' s.Name;

   ...
}
```

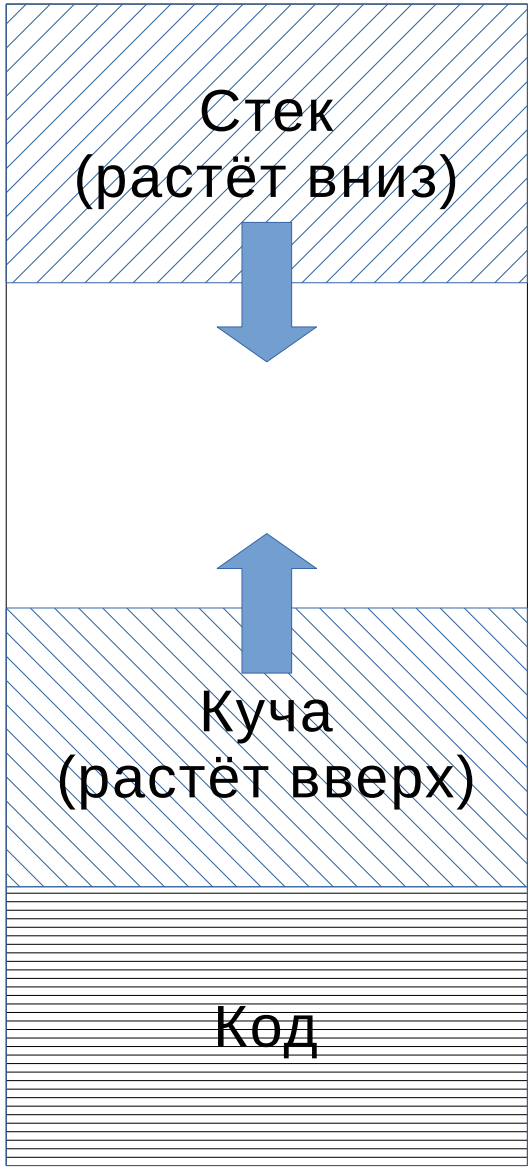
Арифметические выражения (10)

```
GenExpr {
   ...

  (L t.Expr) = <GenExpr t.Expr> LOAD;
  ("-" t.Expr) = <GenExpr t.Expr> NEG;

  (t.Left s.BinOp t.Right)

    , ("+" ADD) ("-" SUB) ("*" MUL) ("/" DIV) ("%" MOD)
      ("|" BITOR) ("&" BITAND) ("~" BITNOT)
    : e.1 (s.BinOp s.Command) e.2

    = <GenExpr t.Left> <GenExpr t.Right> s.Command;

   ...
}
```

Арифметические выражения (11)

```
GenExpr {
   ...

  (call t.Func e.Args)
    = <CompileArgs e.Args> <GenExpr t.Func> CALL GETRV;

  (asm e.Code) = '\n' e.Code '\n';
}

CompileArgs {
  e.Args t.Arg = <GenExpr t.Arg> <CompileArgs e.Args>;

  /* пусто */ = /* пусто */;
}
```

Адресное пространство программ



(По-хорошему, этот слайд должен быть в первой лекции.)

Стек находится в старших адресах и растёт вниз.

Код находится в младших адресах.

Над кодом располагается куча, которая растёт вверх.

(Реализацию кучи будем рассматривать позже.

Фрейм стека (1)

НИУЯП поддерживает рекурсию. Для хранения параметров и локальных переменных каждого (а также другой, служебной информации) используются, как и на многих других архитектурах, **фреймы стека.**

Фреймы стека будут иметь следующую структуру (от старших адресов ко младшим):

- параметры функции (от последнего к первому),
- адрес возврата,
- предыдущее значение регистра фрейма FP,
- локальные переменные,
- значения подвыражений.

Фрейм стека (2)

Рассмотрим вызов `(call f 10 20 30)` следующей функции:

```
(function f (x y z)
  (return
    (
      (x "*" x)
      "+"
      (call g
        (y "+" 1)
        (z "+" 1))
      )
  )
)

(function g (a b) …)
```

Фрейм стека в момент вызова функции `g`:

Слово на стеке	Пояснение
…	
30	параметр z
20	параметр y
10	параметр x
‹ret›	
‹FP›	
100	значение `(x "*" x)`
31	параметр b
21	параметр a
‹ret›	

Роль регистра фрейма FP (1)

Адреса локальных переменных становятся известны только в момент вызова функции: функция может в программе вызываться неоднократно из различных функций и её локальные переменные будут располагаться на разной глубине. Более того, функция может быть рекурсивной, а значи, одновременно на стеке будет присутствовать несколько экземпляров её локальных переменных.

К локальным переменным нужно каким-то образом обращаться — знать их адреса.

Роль регистра фрейма FP (2)

Можно ссылаться на них, вычитая их смещение из регистра SP — адреса верхушки стека, однако, это будет плохой идеей. Действительно, предположим, что у нас регистра фрейма нет и мы компилируем следующую функцию:

```
(function f (x) (return (x "*" x)))
```

Выражение `(x "*" x)` содержит два вхождения переменной `x`. При компиляции обоих вхождений содержимое стека будет различным:

Стек	Смещение
x	+1
‹ret›	0

Стек	Смещение
x	+2
‹ret›	+1
x	0

Роль регистра фрейма FP (3)

На стек добавился первый операнд для *, смещения локальных переменных увеличились на 1.

Учёт глубины стека при адресации относительно SP возможен, но будет усложнять компиляцию — придётся отслеживать текущую глубину стека, чтобы правильно вычислять адреса локальных переменных.

Поэтому удобно иметь некоторую глобальную переменную, которая хранит адрес фрейма текущего стека и её значение не меняется во время выполнения функции. В роли такой переменной используется регистр FP.

Адресация относительно FP (1)

Адрес, хранимый в FP, для каждого вызова функции будет своим, поэтому при вызове нужно сохранять старое значение FP, а при возврате из функции его восстанавливать.

Удобно сохранять старое значение FP в самом начале выполнения функции, после чего устанавливать значение FP равным текущему значению SP:

```
GETFP GETSP SETFP
```

Адресация относительно FP (2)

Стек вызова функции примет вид:

Стек	Смещение
параметр N	$+(N+1)$
...	...
параметр 1	$+2$
адрес возврата	$+1$
старое значение FP	0

На вершине стека будет находиться старое значение регистра FP, значения регистров FP и SP будут совпадать.

Адресация относительно FP (3)

Пусть функция содержит локальные переменные

`(var (v1 s1) … (vn sn))`

где `v1`, …, `vn` — их имена, а `s1`, …, `sn` — их размеры.

Тогда память для них можно выделить командой

`‹размер› PUSHN`

где $\langle \text{размер} \rangle = \sum_{i=1}^{n} s_i$.

Адресация относительно FP (4)

Стек примет вид:

Стек	Смещение
параметр N	$+(N+1)$
…	…
параметр 1	$+2$
адрес возврата	$+1$
старое значение FP	0
переменная v1	$-s_1$
переменная v2	$-s_1 - s_2$
переменная v3	$-s_1 - s_2 - s_3$
…	…
переменная vj	$-\sum_{i=1}^{j} s_i$
…	…
переменная vn	$-\sum_{i=1}^{n} s_i$

Адресация относительно FP (5) — выводы

Параметры функций имеют положительные смещения, локальные переменные — отрицательные. Такой подход позволяет компилировать как функции с переменным числом параметров (адрес первого параметра известен, смещения остальных растут последовательно), так и функции, выделяющие на стеке место произвольного размера (компиляция `int xs[N];`, где `N` — переменная).

Заметим, что регистр `FP` содержит адрес слова, содержащего старое значение регистра `FP`, а то содержит адрес слова с пред-предыдущим значением `FP` и т.д. Таким образом, фреймы стека образуют однонаправленный список, который оканчивается значением `0` (при запуске программы `FP` равен нулю и он будет сохранён в самом первом фрейме).

То, что фреймы образуют однонаправленный список, нам ещё пригодится.

Компиляция логических выражений

```
t.BoolExpr ::=
    TRUE | FALSE
  | (t.Expr s.RelOp t.Expr)
  | (not t.BoolExpr)
  | (t.BoolExpr and t.BoolExpr)
  | (t.BoolExpr or t.BoolExpr)
s.RelOp ::= "<" | ">" | "=" | "<>" | "⩾" | "⩽"
```

Арифметическое выражение вычисляет значение, которое оно оставляет на вершине стека.

Компиляция логических выражений отличается принципиально — сгенерированный код совершает условный переход в зависимости от истинности выражения.

Наивная компиляция логических выражений (1)

В псевдокоде далее мы будем использовать следующее обозначение:

`[[код для t.BoolExpr → (метка1 / метка2)]]`

означающее переход на `метку1`, если выражение истинно, и на `метку2`, если оно ложно.

Логические константы `TRUE` и `FALSE` компилируются в безусловные переходы:

- `[[код для TRUE → (метка1 / метка2)]]` преобразуется в `метка1 JMP`,
- `[[код для FALSE → (метка1 / метка2)]]` преобразуется в `метка2 JMP`,

Наивная компиляция логических выражений (2)

Операция отношения компилируется в условный переход:

`[[код для (t.LeftExpr s.RelOp t.RightExpr) → (метка1 / метка2)]]`

преобразуется в

```
[[код для t.LeftExpr]]
[[код для t.RightExpr]]
CMP ‹метка1› [[инструкция для s.RelOp]]
‹метка2› JMP
```

Здесь `[[инструкция для s.RelOp]]` это, соответственно `JLT` для "<", `JGT` для ">", `JEQ` для "=", `JLE` для "⩽", `JGE` для "⩾" и `JNE` для "◇".

Очевидно, что сравнение с нулём `(t.Expr s.RelOp 0)` можно оптимизировать — не генерировать код для второго выражения и команду `CMP`.

Наивная компиляция логических выражений (3)

Логическое отрицание просто меняет цели переходов:

`[[код для (not t.BoolExpr) → (метка1 / метка2)]]`

компилируется как код

`[[код для (t.BoolExpr) → (метка2 / метка1)]]`

Наивная компиляция логических выражений (4)

Компиляция конъюнкции и дизъюнкции работают по короткозамкнутой схеме — если истинность/ложность понятна из первого аргумента, то второй не вычисляется.

```
[[код для (t.Left and t.Right) → (метка1 / метка2)]]
```

преобразуется в

```
[[код для t.Left → (‹новая метка› / метка2)]]
:‹новая метка›
[[код для t.Right → (метка1 / метка2)]]
```

Здесь `‹новая метка›` — идентификатор, который в программе ранее не встречался.

Наивная компиляция логических выражений (5)

Дизъюнкция компилируется похожим способом:

```
[[код для (t.Left or t.Right) → (метка1 / метка2)]]
```

преобразуется в

```
[[код для t.Left → (метка1 / ‹новая метка›)]]
:‹новая метка›
[[код для t.Right → (метка1 / метка2)]]
```

Компиляция операторов (1)

Слово *statement* мы будем переводить как *оператор*.

```
e.Code ::= t.Statement*
t.Statement ::=
    (t.Expr "=" t.Expr)
  | (call t.Expr t.Expr*)
  | (return t.Expr)
  | (if t.BoolExpr e.Code)
  | (if t.BoolExpr e.Code else e.Code)
  | (while t.BoolExpr e.Code)
  | (asm s.ANY+)
```

В отличие от выражений, операторы не порождают значений.

После выполнения оператора стек остаётся неизменным.

Компиляция операторов (2) — присваивание и вызов

Оператор присваивания (`t.Target "=" t.Value`) компилируется тривиально:

```
[[код для t.Target]]
[[код для t.Value]]
SAVE
```

Оператор вызова функции (`call t.Expr t.Expr*`) компилируется аналогично операции вызова функции в выражении, но должен удалить со стека возвращаемое значение:

```
[[код для выражения (call t.Expr t.Expr*)]] DROP
```

Компиляция операторов (3) — ветвление

Условный оператор

```
(if t.BoolExpr e.Code-True else e.Code-False)
```

компилируется как

```
[[код для t.BoolExpr → (‹метка-true› / ‹метка-false›)]]

:‹метка-true›
[[код для e.Code-True]]
‹метка-выход› JMP

:‹метка-false›
[[код для e.Code-False]]

:‹метка-выход›
```

Компиляция операторов (4) — ветвление

Условный оператор без отрицательной ветки

```
(if t.BoolExpr e.Code-True)
```

компилируется как

```
[[код для t.BoolExpr → (‹метка-true› / ‹метка-выход›)]]

:‹метка-true›
[[код для e.Code-True]]

:‹метка-выход›
```

Компиляция операторов (5) — цикл

Цикл с предусловием

```
(while t.BoolExpr e.Body)
```

компилируется как

```
:‹метка-цикл›
[[код для t.BoolExpr → (‹метка-true› / ‹метка-выход›)]]

:‹метка-true›
[[код для e.Body]]
‹метка-цикл› JMP

:‹метка-выход›
```

Компиляция операторов (6) — возват

Оператор возврата (`return t.Expr`) вычисляет значение выражения, сохраняет его в регистре RV и переходит на эпилог (см. далее):

```
[[код для t.Expr]]
SETRV
‹метка для эпилога› JMP
```

Компиляция операторов (7) — ассемблерная вставка

Ассемблерная вставка (`asm s.ANY+`) как есть выписывается в целевой код, занимая одну строчку текста. Компилятор никак не проверяет её содержимое.

Код в ассемблерной вставке не должен нарушать инварианты фрейма стека.

Эффективная компиляция логических выражений (1)

При компиляции `if` и `while` переход по истине осуществляется непосредственно на код, следующий за кодом логического условия. Это наталкивает на мысль о генерации кода условий таким образом, чтобы когда условие истинное — переход никуда не совершался, а когда ложное — переход выполнялся на некоторую метку (которой помечена отрицательная ветка в `if` или выход из цикла `while`).

Будем обозначать псевдокод для такой генерации как

```
[[ код для t.BoolExpr →→ метка ]]
```

Код осуществляет переход на метку, если `t.BoolExpr` ложное, и никуда не переходит (т.е. выполняется код непосредственно после кода условия), когда условие истинно.

Эффективная компиляция логических выражений (2)

```
[[ код для TRUE →→ метка ]]
```

Тождественно истинное условие компилируется в отсутствие кода — действительно, код, следующий за условием, будет выполняться непосредственно.

```
; пусто
```

Бесконечный цикл компилируется красиво и естественно:

Переход на ‹метку-выход› никогда не генерируется (если, конечно, цикл не содержит оператор `break` — один из вариантов лабораторной).

Эффективная компиляция логических выражений (3)

```
[[ код для FALSE →→ метка ]]
```

Тождественое ложное условие компилируется в безусловный переход на метку:

```
метка JMP
```

Эффективная компиляция логических выражений (4)

`[[ код для (t.LeftExpr s.RelOp t.RightExpr) →→ метка ]]`

Отношение компилируется в переход на метку
по противоположному условию:

```
[[ код для t.LeftExpr ]]
[[ код для t.RightExpr ]]
CMP метка [[ обратный переход для s.RelOp ]]
```

Обратный переход — это `JGE` для "`<`", `JGT` для "$\leqslant$", `JNE` для `=`
и т.д.

Код для сравнения с нулём (`t.Expr s.RelOp 0`) можно
оптимизировать — не компилировать код для `0` и команду `CMP`.

Эффективная компиляция логических выражений (5)

```
[[ код для (t.LeftExpr and t.RightExpr) →→ метка ]]
```

Конъюнкция компилируется красиво. Если левое условие `t.LeftExpr` ложное — переход осуществляется на метку. Если истинно — на код проверки второго условия `t.RightExpr`. Соответственно, если правое условие ложно — переход на метку, если истинно — на код после условия:

```
[[ код для t.LeftExpr →→ метка ]]
[[ код для t.RightExpr →→ метка ]]
```

Эффективная компиляция логических выражений (6)

```
[[ код для (t.LeftExpr or t.RightExpr) →→ метка]]
```

Дизъюнкция компилируется сложнее, чем конъюнкция.

Левое выражение `t.LeftExpr` должно передавать управление на код, следующий за кодом условия, когда оно истинно, и на код проверки `t.RightExpr`, когда оно ложно. Это противоречит соглашению, что код условия делает переход на метку по лжи, но идёт дальше в случае истины.

Противоречие можно разрешить, если генерировать код не непосредственно для левого условия, а для его отрицания:

```
[[ код для (not t.LeftExpr) →→ ‹новая метка› ]]
[[ код для t.RightExpr →→ метка ]]
:‹новая метка›
```

Эффективная компиляция логических выражений (7)

```
[[ код для (not t.BoolExpr) →→ метка ]]
```

А вот отрицание компилируется сложнее всего.

Эффективная компиляция логических выражений (7)

```
[[ код для (not t.BoolExpr) →→ метка ]]
```

А вот отрицание компилируется сложнее всего.

Можно, конечно, накостылить и кучу лишних переходов:

```
[[ код для t.BoolExpr →→ ‹новая метка› ]]
метка JMP
:‹новая метка›
```

но это некрасиво.

Эффективная компиляция логических выражений (8)

Нормальная форма отрицания — форма записи логического выражения, в которой операция НЕ не применяется к логическим операциям (И, ИЛИ, НЕ).

Головная нормальная форма отрицания — форма записи логического выражения, где операция НЕ не может быть на самом верхнем уровне.

Построение головной нормальной формы отрицания:

- Отрицания И и ИЛИ преобразуются по законам де Моргана.
- Двойное отрицание просто снимается $\neg\neg E \Rightarrow E$.
- Отрицание отношения преобразуется в противоположное отношение ("<" $\leftrightarrow$ " $\geqslant$ ", ">" $\leftrightarrow$ " $\leqslant$ ", "$=$" $\leftrightarrow$ "$\diamond$").
- Отрицания логических констант TRUE и FALSE — константы с противоположным значением.

Эффективная компиляция логических выражений (7)

`[[ код для (not t.BoolExpr) ↠ метка ]]`

Так что эффективная компиляция логического отрицания сводится к компиляции кода для его головной нормальной формы отрицания.

Строить нормальную форму отрицания нецелесообразно, т.к. для выражения

$$\neg(((\ldots(r_1 \wedge r_2) \ldots \wedge r_{n-1}) \wedge r_n)$$

компиляция потребует квадратичного времени работы,
в то время как для головной нормальной формы отрицания — линейного (здесь r_i — отношения).

Компиляция функций

Функция

```
(function s.Name (e.Params)
  (var e.Vars)
  e.Body
)
```

компилируется в

```
:_ s.Name
[[ пролог для e.Params, e.Vars ]]
[[ код для e.Body ]]
:‹метка для эпилога›
[[ эпилог для e.Params, e.Vars ]]
```

Здесь «пролог» и «эпилог» — шаблонный код, который генерируется для захода и выхода в функцию. На `‹метку для эпилога›` осуществляет переход `return`.

Пролог

Пролог мы уже рассматривали

```
[[ пролог для e.Params, e.Vars ]]
```

компилируется в

```
GETFP GETSP SETFP
<размер e.Vars> PUSHN
```

Если локальных переменных нет, то команда `PUSHN` не генерируется.

Задача пролога — инициализировать фрейм стека: сохранить старое значение регистра фрейма и установить его новое значение на «точку отсчёта», от которой вычисляются смещения параметров (положительные) и смещения локальных переменных (отрицательные).

Эпилог (1)

`[[ эпилог для e.Params, e.Vars ]]`

Задача эпилога — разобрать фрейм функции. Вызов функции является выражением, а значит, он должен после своего завершения на стеке оставлять возвращённое значение. Как мы помним, при компиляции вызова функции на стек кладутся аргументы функции, затем адрес входа в функцию, затем вызывается инструкция `CALL`.

Эпилог должен

- удалить локальные переменные,
- восстановить старое значение регистра фрейма,
- удалить параметры функции,
- совершить возврат в точку вызова.

Эпилог (2)

```
[[ эпилог для e.Params, e.Vars ]]
```

Самое простое — удалить локальные переменные. Нам нужно восстановить указатель стека на место, которое предшествует локальным переменным. Мы помним, что регистр фрейма указывает на ячейку памяти со старым значением регистра фрейма, которая располагается на стеке под локальными переменными. Т.е. для удаления локальных переменных нам надо присвоить регистру стека SP значение регистра фрейма FP:

```
GETFP SETSP
```

Теперь значение старого адреса фрейма лежит на вершине стека. Чтобы его восстановить, достаточно его снять:

```
SETFP
```

Эпилог (3)

```
[[ эпилог для e.Params, e.Vars ]]
```

Теперь на стеке лежат значения параметров (возможно, изменённые, т.к. они тоже — локальные переменные) и адрес возврата.

Теперь нам нужно уничтожить параметры и совершить возврат.

У нас есть замечательная инструкция `RETN`, которая со стека снимает число, адрес возврата и `n` слов (т.е. смещает указатель стека). Удаляемое количество слов должно быть равно числу параметров у функции.

```
RETN (-26) : ... x1…xN a N --> ...     (CP := a)
```

Для функции без параметров можно использовать `JMP`.

Эпилог (4) — компиляция (`return t.Expr`)

`[[ эпилог для e.Params, e.Vars ]]`

Обозначим ‹Npar› = |e.Params| — число параметров функции.
Эпилог для ‹Npar› > 0 компилируется так:

```
:‹метка для эпилога›
GETFP SETSP SETFP
‹Npar› RETN
```

Эпилог для ‹Npar› == 0 компилируется в

```
:‹метка для эпилога›
GETFP SETSP SETFP
JMP
```

А если нет `return`'а?

А если в функции не будет оператора `return`?

Оператор `return` присваивает возвращаемое значение регистру `RV` и совершает переход на эпилог. Если оператор `return` не выполнялся, эпилог будет выполнен в силу того, что он располагается после самого последнего оператора. Значение регистра `RV` окажется неопределённым.

Сгенерированный код использует регистр `RV` только для возврата значений из функции, поэтому если функция не завершилась `return`'ом, она вернёт значение какой-то другой, ранее вызванной функции.

Т.е. без оператора `return` функция завершится корректно, но возвращаемое значение будет неопределённым.

Рантайм (1)

Код на НУИЯПе организован в виде функций, функция, с которой начинаются вычисления, носит название `main`, как и в языке Си. Функция `main` не имеет параметров, её возвращаемое значение становится кодом возврата для инструкции `HALT`.

Т.е. программа на ассемблере должна начинаться с кода вызова `main` и завершения виртуальной машины:

```
_main CALL
GETRV HALT
```

Рантайм (2)

Поскольку размер доступной памяти выполняемой программе неизвестен (неизвестно, во сколько слов она скомпилируется и какой объём памяти будет выдан интерпретатору), имеет смысл эти значения сохранить в переменные при инициализации:

```
GETSP _MEMORY_SIZE SWAP SAVE
_main CALL
GETRV HALT
:_MEMORY_SIZE 0
:_PROGRAM_SIZE PROGRAM_SIZE
```

Таким образом, сгенерированному коду будут доступны переменные `MEMORY_SIZE` и `PROGRAM_SIZE`.

Вспомогательные функции

Ассемблерные вставки позволяют написать обёртки над инструкциями ввода-вывода и инструкциями GETFP и HALT:

```
(function in ()
  (return (asm IN))
)

(function out (char)
  (asm GETFP 2 ADD LOAD OUT)
)

(function getFP ()
  (return (asm GETFP))
)

(function halt (code)
  (asm GETFP 2 ADD LOAD HALT)
)
```

Здесь `GETFP 2 ADD LOAD` — обращение к первому аргументу функции.

Первые две функции нужны для осуществления ввода-вывода, третья — потребуется в алгоритмах сборки мусора, четвёртая — просто полезна.

Представление НУИЯПа на других языках

Представление НУИЯПа на других языках (1)

НУИЯП — язык, не имеющий конкретного синтаксиса и описываемый своим синтаксическим деревом.

В лекции описано представление НУИЯПа на Рефале — по мнению преподавателя реализация алгоритмов на нём оказывается наиболее лаконичной.

Однако, в лабораторной работе не обязательно использовать именно Рефал-5. Синтаксическое дерево можно описывать на любом языке программирования.

Представление НУИЯПа на других языках (2)

Для примера рассмотрим представление логических выражений на разных языках. На Рефале-5 оно описано так:

```
t.BoolExpr ::=
    TRUE | FALSE
  | (t.Expr s.RelOp t.Expr)
  | (not t.BoolExpr)
  | (t.BoolExpr and t.BoolExpr)
  | (t.BoolExpr or t.BoolExpr)
s.RelOp ::= "<" | ">" | "=" | "<>" | "≥" | "≤"
```

В примерах дальше метод `generate` не имеет параметров и возвращаемого значения. В реальной программе он может (и должен) иметь иную сигнатуру.

Представление НУИЯПа на Java (1)

```
interface BoolExpr {
  public void generate();

  // Синглтоны для констант
  static final TRUE = new BoolExpr() {
    public void generate() { … }
  }

  static final FALSE = new BoolExpr() {
    public void generate() { … }
  }
}
```

Представление НУИЯПа на Java (2)

```
enum RelOp { LT, GT, EQ, NE, GE, LE }

record RelOpExpr(Expr left, RelOp op, Expr right)
    implements BoolExpr {
  public void generate() { … }
}

record NotExpr(BoolExpr expr) implements BoolExpr {
  public void generate { … }
}

record AndExpr(BoolExpr left, BoolExpr right)
    implements BoolExpr {
  public void generate { … }
}
```

Представление НУИЯПа на Python’е (1)

```
class BoolExpr(abc.ABC):
    pass

@dataclass
class ConstBoolExpr(BoolExpr):
    value : Bool

    def generate(self):
        …

class RelOp(enum.Enum):
    LT = '<'
    GT = '>'
    EQ = '='
    NE = '≠'
    GE = '≥'
    LE = '≤'
```

```
@dataclass
class RelOpExpr(BoolExpr):
    left : Expr
    op : RelOp
    right : Expr

    def generate(self):
        …

@dataclass
class NotExpr(BoolExpr):
    expr : Expr

    def generate(self):
        …
```

Представление НУИЯПа на Python’е (2)

```
@dataclass
class AndExpr(BoolExpr):
    left : Expr
    right : Expr

    def generate(self):
        …

@dataclass
class OrExpr(BoolExpr):
    left : Expr
    right : Expr

    def generate(self):
        …
```

Представление НУИЯПа на других языках (3)

Однако, программу на НУИЯПе хардкодить в компилятор нельзя — она должна загружаться из внешнего файла. Поэтому десериализатор из человеко-читаемой формы (XML, JSON, …) нужно будет написать самостоятельно.

Допустимое читерство: можно использовать `eval`, если она доступна в языке реализации.