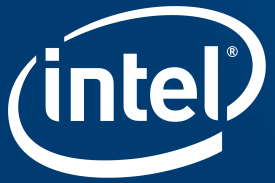


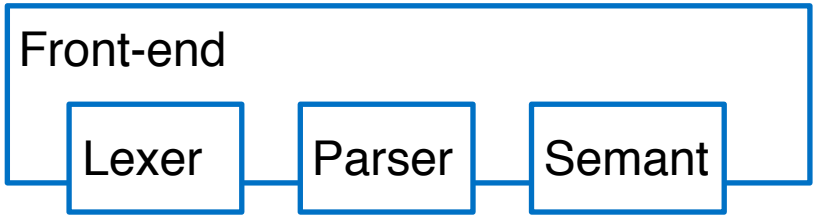
Runtime organization.

Горбань Игорь

~~13~~ 20 марта 2018

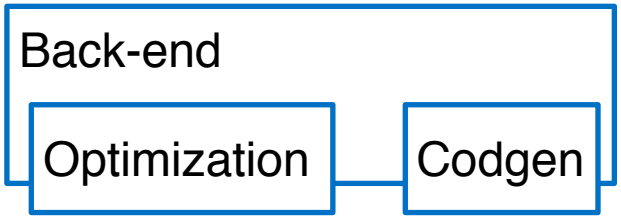
g.i.b@list.ru

Runtime



Отвечает за языковые определения

После этих фаз - гарантия того, что программа корректна



Создание исполняемого кода

Гарантия корректности сохраняется

Чтобы понять, как подходить к кодогенерации - рассмотрим несколько техник для структурирования исполняемого кода.

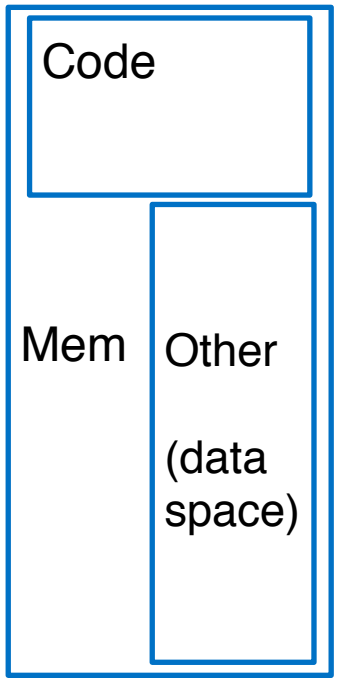
Management of runtime resources.

Необходимо найти соответствие между статическими (compile-time) и динамическими (runtime) структурами.

Исполнение программы начинается под контролем ОС. Когда программа запускается:

- ОС *аллоцирует* место под программу в оперативной памяти
- Исполняемый код загружается в часть этого места
- ОС начинает исполнять (совершает jump) программу с entry point (обычно подразумевается - main)

Memory



Младшие адреса

0x0...0

Старшие адреса

0xF...F

Такое распределение памяти - используется повсеместно.

Компилятор отвечает за

- Генерацию кода
- Использование Data Space памяти

Activations

Активация P - вызов процедуры P, включая все подвызовы процедур из P.

Предположения:

1. Исполнение программы - последовательное, переходы от одной точки программы к другой хорошо определены. (Может быть нарушено многопоточностью)
2. Когда вызываемая процедура заканчивает работу, контроль возвращается на инструкцию, следующую за вызовом. (Может быть нарушено исключениями)

Время жизни активации P:

- весь код исполнения P
- все вызовы, содержащиеся в P

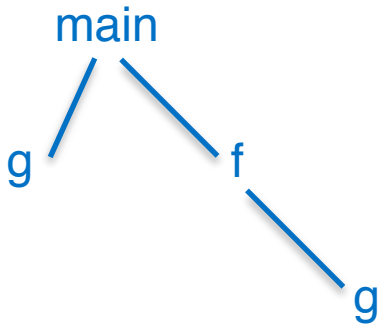
Scope & activation tree

Время жизни переменной x (scope) - участок кода, где x определен.

Scope - статическое понятие

Время жизни активации - может быть представлено в виде дерева:

```
Class Main {
    g() : Int {1};
    f() : Int {g()};
    main() : Int {{g(); f();}};
}
```



Recursive example

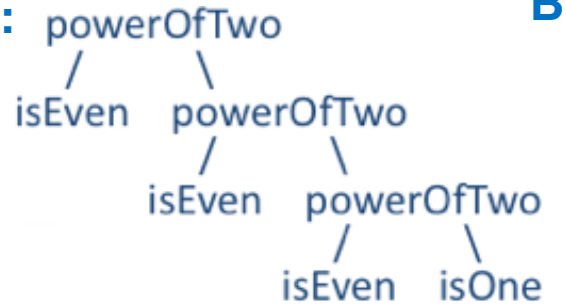
```
Class Main {
    g() : Int {1};
    f(x : Int) : Int {if x == 0 then g() else f(x - 1) fi };
    main() : Int {{f(3);}};
}
```

Task

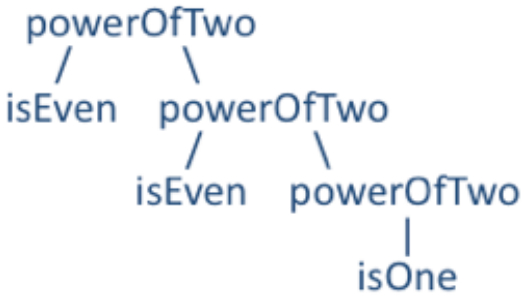
Функция powerOfTwo (), возвращает true, если ее аргумент равен двум, в противном случае - false. Какое дерево активации получится для powerOfTwo (4)?

```
isEven(x:Int) : Bool {x % 2 == 0};
isOne(x:Int) : Bool {x == 1};
powerOfTwo(x:Int) : Bool {
   if isEven(x) then powerOfTwo(x / 2)
   else isOne(x)
};
```

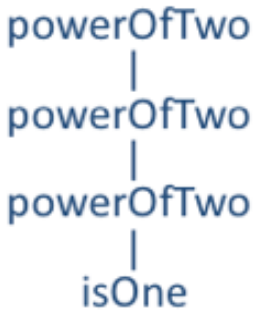
A:



B:



C:



D:

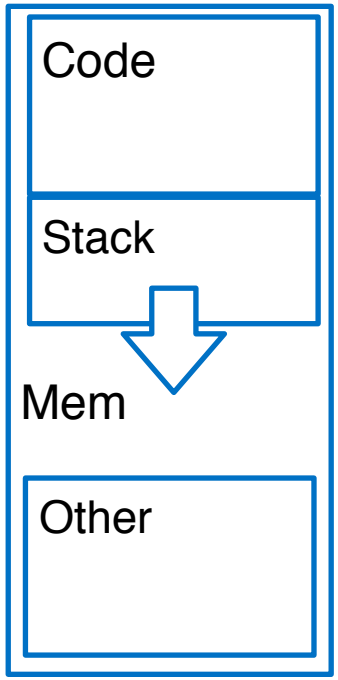
Stack in activation tree

Дерево вызовов (активаций) :

- зависит от runtime поведения
- может различаться для различных входных данных
- вложено(один вызов порождает один или несколько других).

Так что возможно использовать стек для отслеживания вызовов и возвращения значений

Memory map ++



Младшие адреса

0x0...0

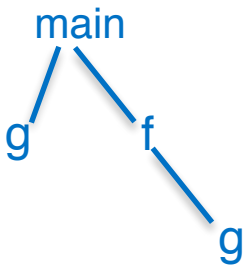
Старшие адреса

0xF...F

Пример использования стека:

```
Class Main {
    g() : Int {1};
    f() : Int {g()};
    main() : Int {{g(); f();}};}
```

Tree:



Stack:

-> main

-> g

<- g

-> f

-> g

<- g

<- f

<- main

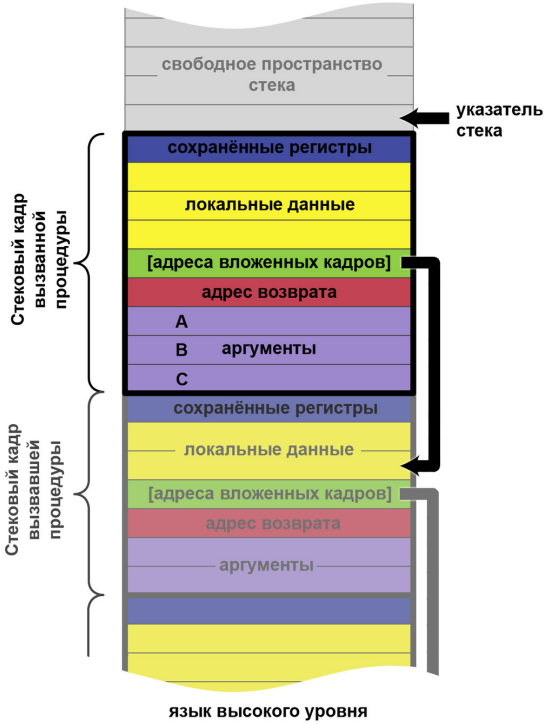
Activation record (frame).

Информация, необходимая для управления одним вызовом называется activation record или stack frame.

Предположим, что процедура F вызывает внутри своего тела процедуру G. Тогда при вызове G, необходимо сохранить информацию:

- Аргументы для G
- Адрес возврата в F
- Иметь возможность передать возвращаемое из G значение так, чтобы F знал откуда его забрать

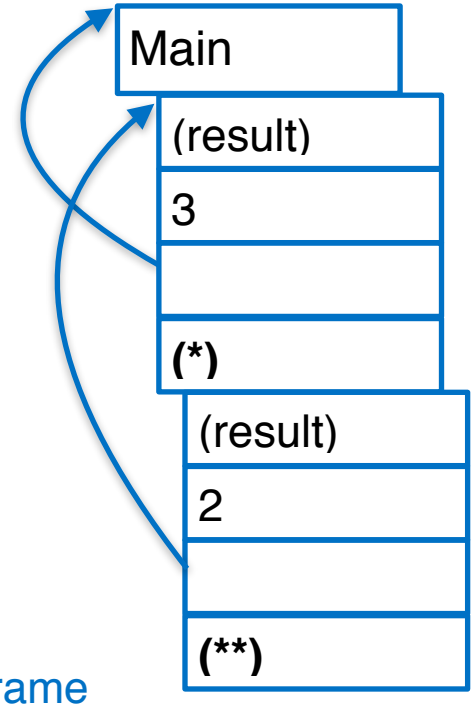
Эти данные и сохраняются при вызове процедуры на стек (всегда ваш, капитан О)



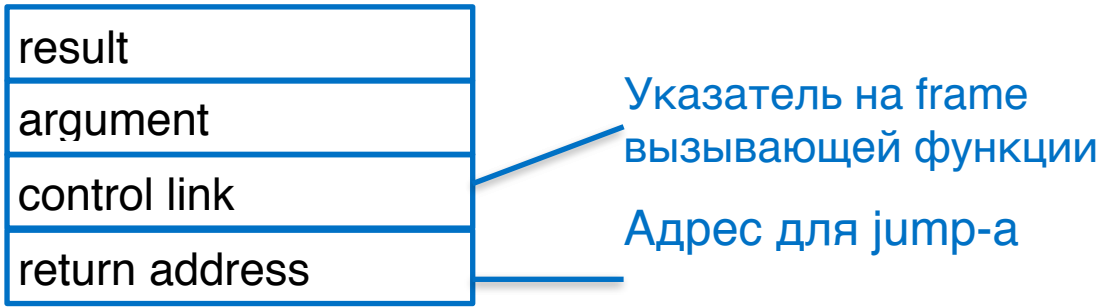
Картинка из википедии.

Frame example (1)

```
Class Main {
    g() : Int {1};
    f(x : Int) : Int {if x == 0 then g() else f(x - 1) (**) fi };
    main() : Int {{f(3); (*)}};
}
```



Frame для f:



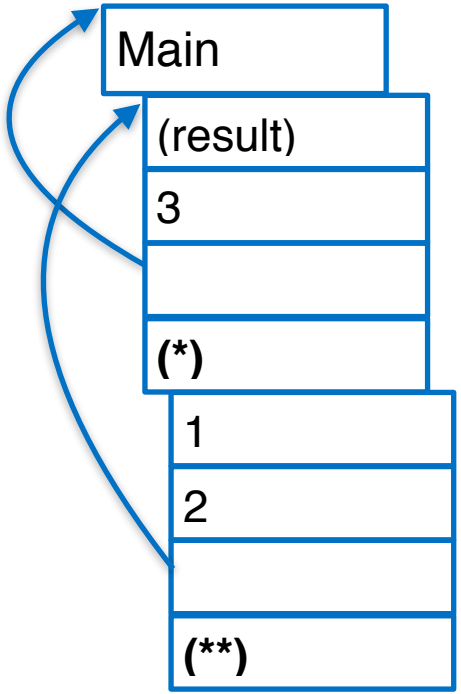
Frame example (2)

Main - не имеет аргументов или локальных переменных и ее результат не используется, т.е. ее frame не представляет интереса.

(*) и **(**)** - адреса возврата для вызовов f.

Это только один из возможных вариантов frame, подобный используется в C, Pascal, Fortran и т.д.

Пусть мы возвращаемся из вызова f(2). Текущий frame меняется, однако frame для f(2) все еще хранится в памяти. Для того, чтобы получить результат - достаточно считать значение относительно начала frame-а по заранее известному отступу.



Conclusions

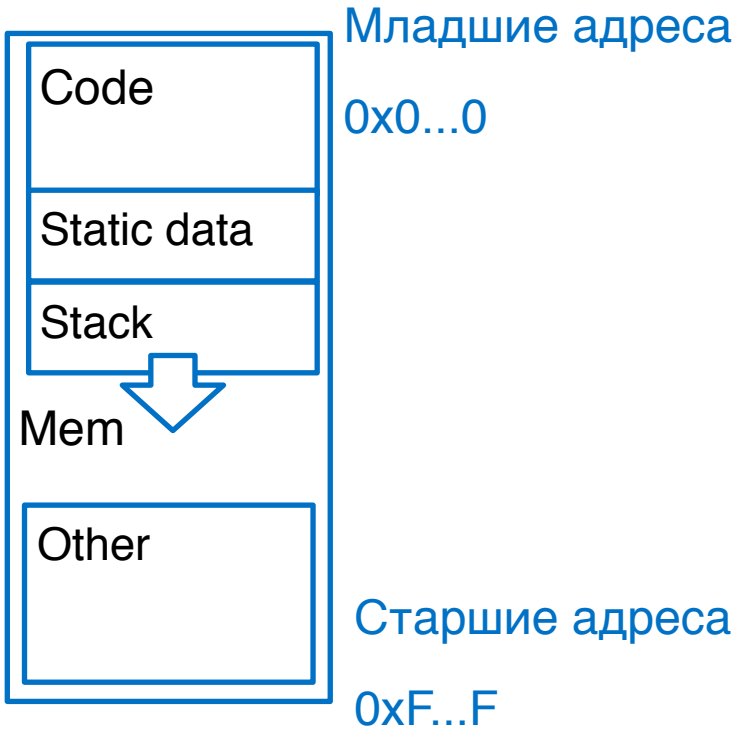
Если рассмотреть наиболее часто используемые архитектуры, встает вопрос, **почему бы не сохранять значения** не в таблице, а **в регистрах**. Однако в таком случае возникает проблема при вызовах одной процедуры из разных мест.

Компилятор должен определять runtime расположение фреймовых записей и codegen должен ссылаться на конкретные офсеты(отступы от указанного адреса), таким образом **codegen и frame-структура должны проектироваться вместе**.

Global variables

Для глобальных переменных - все ссылки должны указывать на один и тот же объект. Очевидно, что сохранять глобальную переменную во frame - плохая идея.

Глобальные переменные получают фиксированный адрес однажды (переменные с фиксированным адресом называются "статически аллоцированными")



Heap (1)

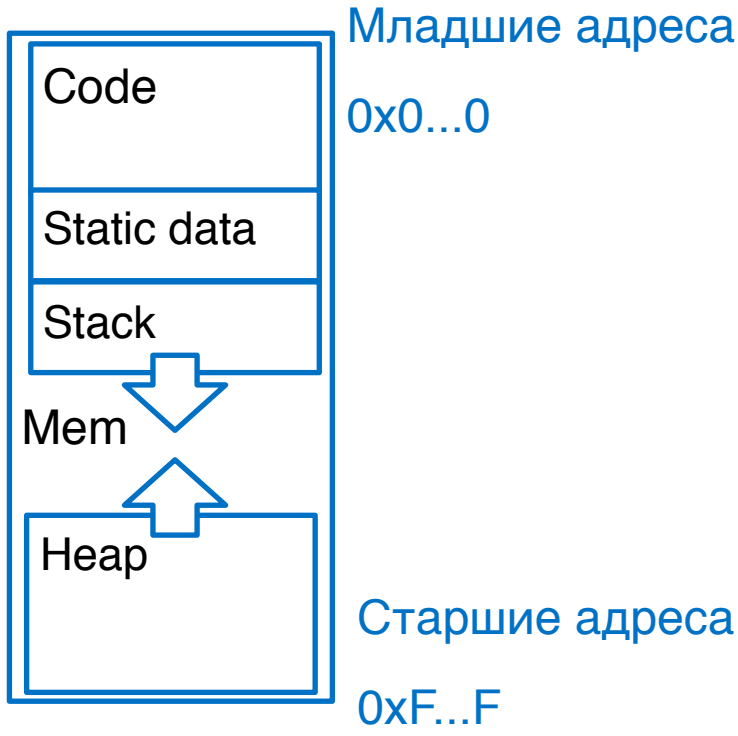
Объект, который живет за пределами процедуры, создавшей его не может храниться во frame-е. Ex : method foo() {new Bar; }

Языки с динамически-выделяемой памятью используют кучу для хранения памяти динамических объектов.

- Область **кода** - содержит бинарный код для большенства языков размер фиксирован и эта область является - read-only
- Область **Static-data** - не содержит кода. У данных отсюда фиксированный адрес (доступ может быть как read-only так и read-write)
- **Стек** содержит фреймы для каждой вызванной процедуры. Обычно фиксированного размера, хранит локальные переменные
- **Куча** хранит все остальные объекты - в C работа с кучей происходит посредством malloc/free.

Heap (2)

В большинстве языков куча и стек - растут и необходимо удостоверится, что они не налезут друг на друга, поэтому стек растет навстречу куче.



Alignment

Выравнивание. Большенство машин 32 или 64-битные и имеют:

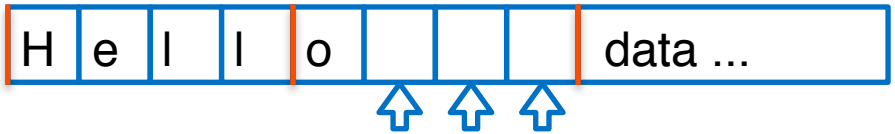
- 8 битов в байте
- 4 или 8 битов в слове
- аддресация или по байтам или по словам

Данные называются выровненными по слову (word-aligned), если их начало расположено у границы слова. Большенство машин имеет ограничения на выравнивание (или ухудшение производительности в случае отсутствия выравнивания).

Alignment (2)

Ex : строка "Hello" - 5 символов (без \0)

Для выравнивания следующего слова(за строкой) добавляется 3 "пробельных" (padding) символа.



"Пробелы" - не являются частью строки, просто неиспользуемая память, которая теряется из-за выравнивания.

Stack machine (1)

Stack machine - простейшая модель работы с frame-ами для кодогенератора. Для работы использует только стек.

Инструкция r = F(a1, ... , an)

- Читает n операндов со стека
- Вычисляет F, используя операнды
- Кладет результат r на стек

Для реализации необходимы инструкции push i - положить значение на стек, операции (например add - сложить 2 числа из стека)

Stack machine (2)

Расположение операндов/результата - не задано явно, они всегда на вершине стека.

Стоит отметить, что в инструкции сильно упрощаются с виду, однако для вычисления каждой необходимо лезть в память.

Промежуточные машины, между чисто-стек и чисто-регистрыми называются n-регистровыми машинами (с использованием n регистров)

Далее мы будем рассматривать 1-регистровую стек-машину.

Ex: (7 + 5)

push 7

push 5

add

Pure-stack vs 1-register stack

Сравнение чистой stack-machine с 1-регистровой stack-machine:

pure-stack machine :	1-register stack machine :
add - использует 3 обращения к памяти, 2 чтения и 1 запись	add : acc <- acc + top_of_stack Имеет только один memory операнд.

acc - accumulator - регистр для хранения результата выполнения последней операции.

То есть считаем, что, по-умолчанию результат последней операции хранится в регистре acc.

Execute cool operation.

Ex: For Cool - описание выражения - op(e1, ... , en), где e1, ... , en - так же выражения, вычисления можно представить так:

Вычисление операндов

```
For each ei (0 < i < n)
    посчитать ei (результат сохранится в acc)
    положить результат в стек (push acc)
en -> посчитать (результат в acc)
```

Вычисление op

```
Получить n-1 значения из стека и посчитать op
Положить результат в acc
```

Add example with 1-register stack machine

Example: 3 + (7 + 5)

Обратите внимание - вычисление выражений не меняет stack.

Code	Acc	Stack
acc <- 3	3	***<init>***
push acc	3	3, <init>
acc <- 7	7	3, <init>
push acc	7	7, 3, <init>
acc <- 5	5	7, 3, <init>
acc <- acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc <- acc + top_of_stack	15	3, <init>
pop	15	***<init>***


DA WAY