Структурные базисные типы - массивы

ГЛАВНЫЙ вопрос реализации массивов в (современных) ЯП:

Len: A-> integer -- время связывания этой операции

Статические ЯП: ситуация сложнее (и разнообразнее)

Простейший вариант: чисто статические массивы

- C/C++ (массив - "недообъект")
- Паскаль (длина ВСЕГДА свойство типа)
- Ада (длина массива для типа может быть не ограничена, но для переменных (подтипа) - всегда задана)
- Модула-2, Оберон (длина ВСЕГДА свойство типа, но есть "открытые" массивы - формальные параметры)

Чем хорошо - эффективность распределения памяти - все адреса - статические

В чем проблема? - отсутствие гибкости (как программировать универсальные операции над массивами - любой длины и диапазона?)

Структурные базисные типы - массивы

ГЛАВНЫЙ вопрос реализации массивов в (современных) ЯП:

Len: A-> integer -- время связывания этой операции

Статические ЯП: ситуация сложнее (и разнообразнее)

Второй вариант: массивы- референциальные типы (как в динамических языках)

- C#, Java

T [] arr = new T[N]; // N может быть ЛЮБЫМ целочисленным выражением >= 0

Длина - динамический атрибут. Есть контроль индекса (как в динамических языках).

Чем хорошо - единообразно с объектами.

В чем проблема? - менее эффективно по сравнению с чисто статическим вариантом, есть проблема realloc (p, size + 1).

Нельзя менять длину массива после создания (для эффективности).

"Настоящие" динамические массивы - контейнеры из стандартной библиотеки (List, IList, Array)

Структурные базисные типы - массивы

ГЛАВНЫЙ вопрос реализации массивов в (современных) ЯП:

Len: A-> integer -- время связывания этой операции

Статические ЯП: ситуация сложнее (и разнообразнее)

Третий вариант: компромисс (язык Go)

Массивы - типы-значения со статической длиной (всегда)

```
var a [N]int // N - константное статическое выражение
var arr [4] int;
```

Для гибкости - есть особое понятие - срез массива (array slice), которое позволяет создавать универсальные операции над массивами, не теряя в эффективности распределения памяти

Срез - "окно", хранящее ссылку на массив и границы среза (полуоткрытый диапазон)

```
i := [4]int {1,2,3,4}
var s [] int
s = i[0:2]
```

Можно сразу связать срез с (анонимным) массивом, автоматически создаваемым в ДП.

```
s =  make([]int, 0, 6) // создан массив из int длины 6 и срез из всего массива
```

Срезы можно объединять, расширять, сужать и т.п.

Структурные базисные типы - массивы

Срезы - важная операция в некоторых современных ЯП. В любых ЯП?

Срезы Go и Python: синтаксически одно и тоже, но семантически разное.

Go - срез - "окно" на непрерывную область элементов какого-то массива (вспомним С!) - отдельный ТД (массивы - для распределения памяти, срезы - для манипуляций с массивами)

Python - срез - это операция, создающая ***новый*** список (массив)

lst = [1,2,3,4,5]	lst := [5]int{1,2,3,4,5}
s = lst[1:4]	s := lst[1:4]
s[0] = -1	s[0] = -1
lst не изменился	lst изменился

Срезы JavaScript - a.slice (l,r) - как в Python - создание нового массива

Java:

```
import java.utils.Arrays;
....
int [] arr = new int[] {1,2,3,4,5,6};
int [] slice = Arrays.copyOfRange(arr, 1,4); // тоже полуоткрытый интервал!
```

C# - до 8.0 не было - другие средства (Array.CopyTo(src, startSrc, dst, startDst, length) - dst нужно было создавать отдельно

В ЯП с чисто статическими массивами - иногда есть (Ада), но особой роли не играют (также семантика копирования)

Структурные базисные типы - массивы

Квазистатические массивы (происхождение - нестандартная библиотечная функция alloca(size))

Примеры:

- C99 /C++ - VLA (variable-length arrays)
- Ада - "динамические" массивы

VLA в C:

```
void f(int n) {
        float arr[n]; // разместили в стеке
        .....
}
```

Ада:

```
procedure F(N:INTEGER) is
    arr: array[0..N-1] of FLOAT;  -- разместили в стеке
begin
 ....
end F;
```

Чем хорошо? Автоматическое освобождение памяти при выходе из блока.

Чем плохо? Область действия - только блок, неэффективность при доступе к некоторым локальным переменным.

Структурные базисные типы - массивы

Многомерные массивы.

Паскаль (C, ...) - массивы массивов.

Если массивы - типы-значения, то нет (особых) проблем.

А если массивы - референциальные ТД (Java, C#, ...)?

В этом случае многомерные массивы - ступенчатые (jagged)

Аналог ступенчатых массивов в C: char * argv[] - параметр main()

Проблема - эффективность (как доступа, так и распределения памяти).

Java - смириться с проблемой.

Пример: https://ideone.com/W6CMPB

Структурные базисные типы - массивы

Многомерные массивы.
Паскаль (C, ...) - массивы массивов.
Если массивы - типы-значения, то нет (особых) проблем.
А если массивы - референциальные ТД (Java, C#, ...)?
В этом случае многомерные массивы - ступенчатые (jagged)
Аналог ступенчатых массивов в C: char * argv[] - параметр main()
Проблема - эффективность (как доступа, так и распределения памяти).
Java - смириться с проблемой.
C# - есть ступенчатые массивы (абсолютно как в Java), а есть прямоугольные
int [][] jagged;
int [,] rect;
Пример: https://ideone.com/Zm82zu

Структурные базисные типы - записи

Ключевой ТД (начиная с конца 60-х гг)

Как и массивы - последовательность (но разнотипных переменных)

Впервые - COBOL (1959) - как структура записи в файле

Как отдельный ТД - C.A.Hoare, N.Wirth - реализованы в ЯП Паскаль

Терминология - record (Паскаль, Ада....), struct (C.....)

Запись - последовательность именованных и (в общем случае разнотипных) переменных, которая рассматривается и ведет себя как единое целое (объявление, :=, передача параметров, возврат из функции).

Массив - тоже последовательность переменных, но имя элемента массива - вычисляемое - a[i].

Имя элемента записи (поля записи) - это имя скалярной переменной ЯП, но локализованное внутри записи.

Каждая переменная из записи доступна через операцию . (ТОЧКА)

R.NAME => ref T, где T - тип переменной - поля записи

Структурные базисные типы - записи

Имя элемента записи (поля записи) - это имя скалярной переменной ЯП, но локализованное внутри записи.

Каждая переменная из записи доступна через операцию . (ТОЧКА)

R.NAME => ref T, где T - тип переменной - поля записи

Общий синтаксис описания(для любого ЯП):

Начало_описания_записи

  последовательность_описаний_переменных

Конец_описания_записи

Структурные базисные типы - записи

Общий синтаксис описания(для любого ЯП):
Начало_описания_записи
  последовательность_описаний_переменных
Конец_описания_записи
Примеры:
Паскаль, Ада, Модула-2, Оберон:

```
RECORD
   X,Y : INTEGER;
   F: T; I : INTEGER;
END
```

(Ада - END RECORD)

Структурные базисные типы - записи

Общий синтаксис описания(для любого ЯП):

Начало_описания_записи

  последовательность_описаний_переменных

Конец_описания_записи

Примеры:

C,..... (угадайте язык)

```
struct Complex {
    float Re, Im;
}
```

Структурные базисные типы - записи

Имя элемента записи (поля записи) - это имя скалярной переменной ЯП, но локализованное внутри записи.

Каждая переменная из записи доступна через операцию . (ТОЧКА)

R.NAME => ref T, где T - тип переменной - поля записи

А вот точка едина для всех....

Операция ТОЧКА и ссылки

a.name

p->name (*p).name

Операция ТОЧКА - время связывания?

Структурные базисные типы - записи

Имя элемента записи (поля записи) - это имя скалярной переменной ЯП, но локализованное внутри записи.

Каждая переменная из записи доступна через операцию . (ТОЧКА)

R.NAME => ref T, где T - тип переменной - поля записи

А вот точка едина для всех....

Операция ТОЧКА и ссылки

a.name

p->name

Операция ТОЧКА - время связывания?

В статических ЯП - только статическая (транслируется в отступ относительно адреса переменной-записи - вспомним STRUCT в MASM).

А в динамических ЯП?

Структурные базисные типы - записи

Операция ТОЧКА - время связывания?

В статических ЯП - только статическая (транслируется в отступ относительно адреса переменной-записи - вспомним STRUCT в MASM).

А в динамических ЯП?

А в каких динамических есть понятие записи???

Структурные базисные типы - записи

Операция ТОЧКА - время связывания?

В статических ЯП - только статическая (транслируется в отступ относительно адреса переменной-записи - вспомним STRUCT в MASM).

А в динамических ЯП?

А в каких динамических есть понятие записи???

Операция ТОЧКА есть везде, а вот понятие записи в динамических ЯП замещается понятием объекта.

JavaScript, Python ....

А вот в этих ЯП - время связывания - *динамическое*

Подразумевается поиск в некоторой таблице имен...

Структурные базисные типы - записи

Операция ТОЧКА есть везде, а вот понятие записи в динамических ЯП замещается понятием объекта

Также и во всех ООЯП - вместо записи - понятие класса, ТОЧКА - операция доступа, а понятие записи - record, struct - либо для совместимости, либо для эффективности (но с ограничением функциональности)

Java - нет вообще

C#, Swift, Delphi - struct - типы-значения (с ограничением функциональности)

C++ - struct === класс (отличаются умолчания доступа)

НО: современные статические ЯП, НЕ ЯВЛЯЮЩИЕСЯ ООЯП, содержат это понятие

Go, Rust - типы-значения (C-style)

Структурные базисные типы - объединения

Объединение типов. Зачем?

"Янус-проблема" (В.Ш.Кауфман)

Проблема проявляется ТОЛЬКО в статических ЯП

"Роль" <=> ТД, разные "роли" => разные ТД

Операции привязаны к ТД, что делать, если типы разные?

Event <=> void EventHandler (Event * pEvent)

Как описать тип Event?

Другой пример - штрих-коды.

Старый - линейный штрих код (кодирует 4 целых значения)

Новый - QR-код (кодирует строку длиной <= 2953 символа ISO-8859-1)

Структурные базисные типы - объединения

Эволюция

С - неразмеченные объединения (union) - самый простой вариант

Паскаль - размеченные и неразмеченные объединения (записи с вариантами)

Ада - размеченные объединения (параметризованные записи)

ООЯП - не нужно (наследование) => C#, Java, ... просто нет...

2010-ые - "иногда они возвращаются"...

Swift, Rust (только размеченные объединения типов)

Структурные базисные типы - объединения

С - неразмеченные объединения (union) - самый простой вариант

И самый ненадежный! Как struct, но все поля располагаются с нулевым смещением

```
union tag {
        T1 t1;
        T2 t2;
        T3 t3;
        T4 t4; .......
};
```

Зачем???

Структурные базисные типы - объединения

С - неразмеченные объединения (union) - самый простой вариант

И самый ненадежный! Как struct, но все поля располагаются с нулевым смещением

```
union tag {
        T1 t1;
        T2 t2;
        T3 t3;
        T4 t4; .......
};
```

Зачем???

Первая причина - проблема повторного использования выделенной памяти (50-60-е гг - актуальная проблема)

Структурные базисные типы - объединения

C - неразмеченные объединения (union) - самый простой вариант

Зачем???

Вторая причина - "дыра" в контроле типов (для C - неактуально, но может быть эффективно)

Пример - быстрый доступ к байтам в целом значении

Как получить значение старшего и младшего байтов в целом?

```
union integer {
        char    bytes[NUM_BYTES];
        int     i;
} ui;
ui.i = 0x12345678;
ui.bytes[0] - ?  // абсолютно машинно-зависимо (порядок байтов, размер)
ui.bytes[3] - ?
```

Эффективней, чем со сдвигами и выделением по маске....

Структурные базисные типы - объединения

C - неразмеченные объединения (union) - самый простой вариант

Зачем???

Третья причина - объединение типов

Проблема: Event <=> void EventHandler (Event * pEvent)

```
union event {
        struct KBEvent kbEvent;
         struct MouseEvent mouseEvent;
         struct TimerEvent timerEvent; ........
};
```

А как EventHandler будет различать варианты событий?

Структурные базисные типы - объединения

С - неразмеченные объединения (union) - самый простой вариант

Зачем???

Третья причина - объединение типов

Проблема: Event <=> void EventHandler (Event * pEvent)

```
union event {
        struct KBEvent kbEvent;
         struct MouseEvent mouseEvent;
         struct TimerEvent timerEvent; ........
};
```

А как EventHandler будет различать варианты событий?

Решение - "разметить" каждый вариант, то есть снабдить все варианты меткой типа ("поле типа")

Структурные базисные типы - объединения

Размеченные ("дискриминированные") объединения - есть специальное поле (дискриминант), определяющее вариант.

С не содержит *специальной* конструкции => размеченные объединения моделируются по соглашению - в КАЖДОМ типе-варианте по ОДИНАКОВОМУ смещению (проще всего - 0) выделяется поле типа (а какой тип подойдет лучше всего?)

```
enum EEventType {
        EVT_KEYBOARD,
        EVT_MOUSE,
        EVT_TIMER, .....
};
struct Event {enum EEventType eventType;}; // везде можно заменить enum на int
struct KBEvent {enum EEventType kbEventType; ......}; // kbEventType === EVT_KEYBOARD (должно быть!!!)
struct MouseEvent {enum EEventType moseEventType; ......}; // в этих структурах название поля НИКАКОГО значения не имеет ....
struct TimerEvent {enum EEventType timeEventType; ......}; // kbEve timeEventType ntType === EVT_TIMER (должно быть!!!)
union event {
        struct Event event;
        struct KBEvent kbEvent;
         struct MouseEvent mouseEvent;
         struct TimerEvent timerEvent; ........
};
typedef union event Event;
```

Структурные базисные типы - объединения

Размеченные ("дискриминированные") объединения - есть специальное поле (дискриминант), определяющее вариант.

выделяется поле типа (а какой тип подойдет лучше всего?)

```
void EventHandler (Event * pEvent)
{
        switch (pEvent->event.eventType) {
        case EVT_KEYBOARD :
                ProcessKBEvent(&pEvent->kbEvent);
                break;
        case EVT_MOUSE :
                ProcessMouseEvent(&pEvent->mouseEvent);
                break;
        case EVT_TIMER :
                ProcessTimerEvent(&pEvent->timerEvent);
                break;
        .....
        default:
                ProcessUnknownEvent(pEvent);
        }
}
```

Структурные базисные типы - объединения

Размеченные ("дискриминированные") объединения - есть специальное поле (дискриминант), определяющее вариант.

С не содержит *специальной* конструкции => размеченные объединения моделируются по соглашению - в КАЖДОМ типе-варианте по ОДИНАКОВОМУ смещению (проще всего - 0) выделяется поле типа (а какой тип подойдет лучше всего?)

Можно моделировать объединения типов и без **union**

Пример: адреса сокетов (Berkeley sockets)

```
struct sockaddr {
        unsigned short sa_family; // sa_family_t sa_family
        char           sa_data[14];    /* 14 bytes of protocol address */
};
struct sockaddr_in {
        unsigned short sin_family;  // AF_INET
        unsigned short int sin_port;    // Port number
        struct in_addr     sin_addr;    // Internet address
        unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
struct sockaddr _un{
        unsigned short sun_family;  // AF_UNIX
        char sun_path[UNIX_PATH_MAX];
};
```

Системные вызовы:

```
int bind (int sd, struct sockaddr  * sa, int len);
int connect (int sd, struct sockaddr  * sa, int len);
int accept (int sd, struct sockaddr  * sa, int * pLen );
```

Все работают по switch (sa->sa_family) { .... }

Структурные базисные типы - объединения

С - неразмеченные объединения (union) - самый простой вариант

Паскаль - размеченные и неразмеченные объединения (записи с вариантами) - более логично и организованно

синтаксис:

запись ::= **record** пост_часть [вариантная_часть] **end**

пост_часть ::= {объявление_переменных}

вариантная_часть ::= **case** [имя :] тип **of** {вариант}

вариант ::= конст: (пост_часть [вариантная_часть]);

Примеры :

type Complex = **record** Re, Im : real **end**; /* только пост. часть */

Структурные базисные типы - объединения

Паскаль - размеченные и неразмеченные объединения (записи с вариантами) - более логично и организованно

Примеры неразмеченного объединения:

вариантная_часть ::= **case** [имя :] тип **of** {вариант}

вариант ::= конст: (пост_часть [вариантная_часть]);

```
type Word = record
            case Boolean of
            true: (w: integer);
            false: (bits: packed array [0..MAX_BITS-1] of Boolean);
 end; /* только вариантная часть */
X : Word;
X.w := -123456;
X.bits[0] - ?  //все как в C - но синтаксис другой....
X.bits[MAX_BITS-1] - ?
```

Структурные базисные типы - объединения

Паскаль - размеченные и неразмеченные объединения (записи с вариантами) - более логично и организованно

Примеры размеченного объединения:

```
type EventType = (KBEvent, MouseEvent, TimerEvent);
Event = record
        time: DateTime;  /* это постоянная часть */
        case eType: EventType of /*!!!! eType - дискриминант (метка) записи */
        KBEvent : (scancode: byte; pressed: Boolean);
        MouseEvent: (button: MouseButton; pressed: Boolean);
        TimerEvent : ();
end;
```

На что похоже? - конечно на оператор выбора

Как обрабатывать? - конечно оператором выбора.

Структурные базисные типы - объединения

```
procedure EventHandler (var e: Event);
begin
        case e.eType of
        KBEvent : ProcessKBEvent(e.time, e.scancode,
e.pressed);
        MouseEvent: ProcessMouseEvent(e.time, e.button,
                                                e.pressed);
        TimerEvent : ProcessTimerEvent(e.time);
        end
end
```

Структурные базисные типы - объединения

Выделение памяти под объединения:

- по максимуму
- по значению дискриминанта

По максимуму - размер самого длинного варианта:

C - malloc (sizeof (union event))

Паскаль - new (P)

По значению дискриминанта:

C - а там есть дискриминант? - malloc(proger_knows_real_size)

Паскаль - new (P, t_1, t_2, t_3.....) - t_i - значения дискриминантов

var PW: ^Word; new (PW, true);

var pevent: ^Event; new (pevent, TimerEvent);

pevent^.eType := TimerEvent;

Замечание: new в стандарте Паскаля - это не инициализатор!

Структурные базисные типы - объединения

В чем главная проблема? - ненадежность.

Неразмеченные объединения - сознательная "дыра" в системе контроля типов (нет проблем....)

void * <=> T * - аналогично....

Размеченные объединения - нет контроля правильности операций

- изменение дискриминанта
- внутри варианта оператора выбора (обращение к варианту записи, не соответствующему дискриминанту)

Как решать?

Структурные базисные типы - объединения

В чем главная проблема? - ненадежность.

Размеченные объединения - нет контроля правильности операций

- изменение дискриминанта
- внутри варианта оператора выбора (обращение к варианту записи, не соответствующему дискриминанту)

Как решать?

Язык Ада

- только размеченные объединения
- дискриминант - обязательная инициализация и невозможность изменения

Структурные базисные типы - объединения

В чем главная проблема? - ненадежность.

Размеченные объединения - нет контроля правильности операций

- изменение дискриминанта
- внутри варианта оператора выбора (обращение к варианту записи, не соответствующему дискриминанту)

Как решать?

Язык Ада

- только размеченные объединения
- дискриминант - обязательная инициализация и невозможность изменения

Но вторая проблема - остается.

Структурные базисные типы - объединения

ООЯП - не нужно (наследование) => C#, Java, ... просто нет...

Пример - C#:

```
class Event {
		public virtual void EventHandler() { ..... }
}
class KBEvent : Event {
		byte ScanCode;
		bool pressed;
		public override void EventHandler() {
				Event. EventHandler(); // вызов обработчика из базы
				... // обработка, специфичная для клавиатурных событий
		}
}
```

и т.д.

```
// где-то далеко-далеко в другом классе....
void ProcessEvent(Event e) {
		e.EventHandler();
		// ну и что-то интересное для другого класса
}
```

В C# есть и альтернативная возможность - делегаты (фактически - функциональные ТД)

В других ООЯП - аналогично....

Структурные базисные типы - объединения

ООЯП - не нужно (наследование) => C#, Java, ... просто нет...

Технологическая потребность (ТП) - объединение типов (=== размеченное объединение).

Является ли оно критичной ТП?

???

"Критерий истины - практика"

2010-ые - "иногда они возвращаются"...

Swift, Rust (только размеченные объединения типов)

Размеченные объединения <=> перечислимые ТД

Пример из Rust - уже рассматривали

Нет особых операций над перечислимым ТД, кроме match...

Структурные базисные типы - объединения

Swift, Rust (только размеченные объединения типов)

Размеченные объединения <=> перечислимые ТД

Пример: Swift - перечислимые типы (из Swiftbook https://swiftbook.ru/content/languageguide/enumerations/

)

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}
var directionToHead = CompassPoint.west
.... directionToHead = .east
// пока никаких значений (0,1,2 ... и т.п.)
```

Структурные базисные типы - объединения

Пример: Swift - перечислимые типы

Связь с оператором выбора (как и во всех ЯП с перечислимым ТД) - но здесь она практически неразрывна (как и в Rust)

```
directionToHead = .south
switch directionToHead {
case .north:
    print("Lots of planets have a north")
case .south:
    print("Watch out for penguins")
case .east:
    print("Where the sun rises")
case .west:
    print("Where the skies are blue")
}
// Выводит "Watch out for penguins"
```

Структурные базисные типы - объединения

Пример: Swift - перечислимые типы
Как связать значения ПТД с конкретными типами?
Если только для итерации по всем значениям, то вот ответ:

```
enum Beverage: CaseIterable {
    case coffee, tea, juice
}
let numberOfChoices = Beverage.allCases.count
print("\(numberOfChoices) beverages available")
// Выведет "3 beverages available"
// кстати: интерполяция строк - нынче модно
```

Ну а вот и итерация:

```
for beverage in Beverage.allCases {
    print(beverage)
}
// coffee
// tea
// juice
```

Структурные базисные типы - объединения

Пример: Swift - перечислимые типы
Как связать значения ПТД с конкретными типами?

```
enum ASCIIControlCharacter: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}
enum Planet: Int {
    case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune
}
enum CompassPoint: String {
    case north, south, east, west
}
```

Структурные базисные типы - объединения

Пример: Swift - перечислимые типы
Можно связать ПТД с базовым ТД, но они не эквивалентны
Базовый ТД <= ПТД
var i = 0
i = Planet.earth // ОШИБКА - несоответствие типов
i = Planet.earth.rawValue // i == 3
let sunsetDirection = CompassPoint.west.rawValue
//sunsetDirection == "west"
Базовый ТД => ПТД
let Terra = Planet(rawValue:3)
// кстати - тип Terra - Planet? - и это не вопрос!

Структурные базисные типы - объединения

Пример: Swift - перечислимые типы

Внимание: вопрос! А где здесь объединения?

Ассоциативные значения ПТД (до этого - исходные и неявные исходные значения)

Пример - штрих-коды (1D - 4 целых, 2D- строка)

Структурные базисные типы - объединения

Пример: Swift - перечислимые типы

Пример - штрих-коды (1D - 4 целых, 2D- строка)

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
} // здесь значения еще не связаны!!!!
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
let productBarcode2 = Barcode.qrCode("ABCDEFGHIJKLMNOP")
```

Структурные базисные типы - объединения

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
} // здесь значения еще не связаны!!!!
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
let productBarcode2 = Barcode.qrCode("ABCDEFGHIJKLMNOP")
switch productBarcode2 {
case .upc(let numberSystem, let manufacturer, let product, let check):
    print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
case .qrCode(let productCode):
    print("QR code: \(productCode).")
}
// Выводит "QR code: ABCDEFGHIJKLMNOP."
```

Для краткости можно вынести let

Структурные базисные типы - объединения

```
let productBarcode2 = Barcode.qrCode("ABCDEFGHIJKLMNOP")
// Для краткости можно вынести let
switch productBarcode2 {
case let .upc(numberSystem, manufacturer, product, check):
    print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
case let .qrCode(productCode):
    print("QR code: \(productCode).")
}
// Выводит "QR code: ABCDEFGHIJKLMNOP."
// Кстати - еще одна "модная" операция - интерполяция строк
//(выражения внутри строк)
```

Структурные базисные типы - объединения

Сравним Rust, Swift и старые ЯП с точки зрения надежности объединений

Даже Ада - проигрывает

Ну и напоследок подарок от Криса Латтнера:

рекурсивные перечислимые типы (рекурсивные объединения)

Вспомним простейшую грамматику выражений:

E := i | E+E | E * E (очень плоха для анализа, но для представления готовых выражений - ОК)

Структурные базисные типы - объединения

Сравним Rust, Swift и старые ЯП с точки зрения надежности объединений
Даже Ада - проигрывает
Ну и напоследок подарок от Криса Латтнера:
рекурсивные перечислимые типы (рекурсивные объединения)
Вспомним простейшую грамматику выражений:
E := i | E+E | E * E (очень плоха для анализа, но для представления готовых выражений - ОК)

```
enum ArithmeticExpression {
    case number(Int)
    indirect case addition(ArithmeticExpression, ArithmeticExpression)
    indirect case multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

Структурные базисные типы - объединения

Swift: рекурсивные перечислимые типы (рекурсивные объединения)

```
enum ArithmeticExpression {
    case number(Int)
    indirect case addition(ArithmeticExpression, ArithmeticExpression)
    indirect case multiplication(ArithmeticExpression, ArithmeticExpression)
}
// (5 + 4) * 2
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)
let sum = ArithmeticExpression.addition(five, four)
let product = ArithmeticExpression.multiplication(sum,
ArithmeticExpression.number(2))
```

Структурные базисные типы - объединения

Swift: рекурсивные перечислимые типы (рекурсивные объединения)

```
enum ArithmeticExpression {
    case number(Int)
    indirect case addition(ArithmeticExpression, ArithmeticExpression)
    indirect case multiplication(ArithmeticExpression, ArithmeticExpression)
}
func evaluate(_ expression: ArithmeticExpression) -> Int {
    switch expression {
    case let .number(value):
        return value
    case let .addition(left, right):
        return evaluate(left) + evaluate(right)
    case let .multiplication(left, right):
        return evaluate(left) * evaluate(right)
    }
}

print(evaluate(product))
// Выведет "18"
```

Структурные базисные типы - объединения

That's all, folks!