Коваников Дмитрий, группа M4138

Специализация конкретного синтаксиса для своего языка программирования

Язык реализации: Haskell

Платформа: LLVM

Ниже приводится список взятых дополнений для языка программирования *Rogue* — императивного языка с элементами синтаксиса функциональных языков, — и возможные улучшения (если найдётся для них время/желание), после чего следует описание конкретных синтаксических конструкций.

Усложения:

1. Синтаксические улучшения (красивости): необязательность «;» в конце строки, guards.
2. Pattern matching для констант простых типов, короткое тело функции.
3. Разделение переменных на мутабельные и иммутабельные.
4. Функции высшего порядка.

Возможные дополнения:

1. Вызов функций с именованными аргументами (в том числе поддержка значений аргументов по умолчанию).
2. Анонимные функции.
3. Каррирование и частичное применение.

Описание синтаксиса

Листинг 1: Объявление переменных

```
1 mut x: Int = 0, y = 3
2 let z = true                // `z` has type Bool
3 mut t: Int = 3, k = false // variables of different types in one line
```

Ключевое слово `mut` свидетельствует о создании мутабельной переменной, `let` — иммутабельной; наличие типа необязательно, а инициализация переменных должна присутствовать.

Листинг 2: Объявление функции

```
1 f : (b: Bool) -> (mut x: Int) -> (y: Int) -> Int {
2     let z = x + 10
3     return z
4 }
```

Аргументы разделяются последовательностью символов `->`, последний тип не имеет имени переменной — это возвращаемый тип функции (хотя в теории он мог бы иметь, например в PascalABC он имеет неявное имя переменной result, или же сделать что-то аналогичное этой фишки в Go). Сразу после типа идёт блок, в котором находится тело функции. Блок обязательно должен содержать ключевое слово `return`, если возвращаемым типом функции не является `Unit` (аналог `void`). Также возможно разделить объявление и реализацию функции в случае pattern matching'а согласно следующему синтаксису (заодно показывающему некоторые особенности языка):

Листинг 3: Pattern matching

```
f : (b: Bool) -> (mut x: Int) -> (y: Int) -> Int
f false ... = x + 10
f (y = 0) {
    x = x - 3
    let a = x / 2
    return x + a
}
f (y = 1)
    | x > 1 = x - 1
    | else    { let t = x % 2; return t }
f ... = x + 1
```

Многоточие указывает на неявное присутствие всех аргументов слева от =, чтобы в будущем не возникло проблем, если захочется поддержать η-редукцию. Реализация в виде {}-блока может быть только последней. Многоточие обязательно, если не указывается какой-то аргумент и затем следует =, справа от которого может быть только некоторое выражение.

Ниже приводится пример функции быстрого возведения в степень, демонстрирующий остальные синтаксические особенности языка (вызов функции, условный оператор, циклы, чтение и запись в консоль):

Листинг 4: Бинарное возведение в степень

```
binPow : Unit {
    mut k: Int = 0, n: Int = 0
    read k // space-separated arguments list
    read n
    mut res = 1
    while k > 0 {
        if k % 2 == 1 {
            res = res * n
        } else {
            skip // `skip` is empty operator
        }
        n = n * n
        k = k / 2
    }
    print res
}
```

При передаче в качестве аргумента функции другой функции указывать имена переменных аргументов передаевой функции необязательно. В данный момент в этом вообще не будет смысла, так как не поддерживается вызов с именованными аргументами, но теоретически это возможно.

Листинг 5: Передача функции в качестве аргумента

```
decrementAndCheck : (mut x: Int) -> (p: Int -> Bool) -> Bool {
    x = x - 1
    return (p x)
}
```