Итоги летней практики по компиляторным технологиям

Баранов Александр, гр. БПИ183

Реализованный проект: транслятор подмножества языка Haskell в Python, язык транслятора – Java

Поддерживаемые возможности языка:

- поддерживаемый тип данных – Int (целочисленный)
- поддерживаемые операции – сложение, вычитание, умножение, целочисленное деление, унарный минус
- поддерживаются переменные (объявление с помощью конструкции `let имя = значение`)
- поддерживаемый тип функций – Int→Int
- объявление функций возможно с помощью guards, конструкции if..then..else, pattern matching
- в guards обязательно должна присутствовать ветка otherwise
- при pattern matching неспецифированный шаблон должен стоять в коде после всех специфированных
- нельзя объявлять одну и ту же функцию с помощью pattern matching и какого-либо другого способа
- функции вычисляются НЕ лениво, но мемоизированно (т.е. результат кэшируется в памяти)
- функции рекомендуется аннотировать перед объявлением
- функции высших порядков, функции от многих аргументов, частичное применение функций, каррирование, и т.д. НЕ поддерживаются
- функции с побочными эффектами (не функционально чистые) НЕ поддерживаются
- имеются функции-заглушки для ввода-вывода (`...<-getInt` и `print(...)`)
- однострочные комментарии (начинающиеся с `--`) игнорируются
- транслятор предупреждает, если в коде используются переменная и функция с одним и тем же именем

По неизвестным причинам проверка на вызов только объявленной функции работает некорректно (почему-то неправильно считывается значение из HashMap functions, соответствующее функции), диагностика использования только объявленных переменных также не работает (потому что она срабатывала и на параметры функций, а при попытке починить это наследуемым атрибутом ANTLR выдавал, что ‘*rule ... is left recursive but doesn't conform to a pattern ANTLR can handle*‘).

Для компиляции транслятора рекомендуется GNU Make.

Команда компиляции транслятора: `make`

Команда запуска транслятора: `java Haskellv6Parser файл_Haskell.hs`

Транслятор можно перекомпилировать, заменив в исходном коде значение переменной Strict на true – это сделает трансляцию более строгой (так, аннотации функций будут строго обязательны; нельзя будет дважды объявлять одну и ту же переменную со словом `let`)

Ниже приведены примеры программ, транслирующихся реализованным транслятором

```
let f <- getInt
let c = f - 32
c = c*5
c = c/9
print(c)
```

```
fibbonacci :: Int -> Int

fibbonacci 0 = 1
fibbonacci 1 = 1
fibbonacci n = fibbonacci(n-1) + fibbonacci(n-2)

let x <- getInt
print(fibbonacci(x))
```

```
-- Type annotation (optional, same for each implementation)
factorial :: (Integral a) => a -> a

-- Using recursion (with the ifthenelse expression)
factorial n = if n < 2
then 1
else n * factorial (n - 1)

-- Using recursion (with pattern matching)
factorial 0 = 1
factorial n = n * factorial (n - 1)

-- Using recursion (with guards)
factorial n
| n < 2     = 1
| otherwise = n * factorial (n - 1)
```
