МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки Фундаментальная информатика
и информационные технологии

МОДИФИКАЦИЯ МЕХАНИЗМА ОБРАБОТКИ ОТРИЦАТЕЛЬНЫХ ЛИТЕРАЛОВ В КОМПИЛЯТОРЕ GHC

Курсовая работа

Студента 3 курса
Д. Ш. Мирзоева

Научный руководитель:
Старший преподаватель кафедры информатики и вычислительного
эксперимента Мехмата ЮФУ В. Н. Брагилевский

_____ _____

оценка(рейтинг) подпись руководителя

Ростов-на-Дону
2017

Содержание

Введение

Во многих языках программирования присутствует понятие литерала. Литерал — это предопределённая компилятором константа.

В данной курсовой работе нами рассмотрен язык Хаскель. В его стандарте [1] отсутствуют отрицательные литералы в качестве отдельной лексемы. Для записи отрицательных чисел используется унарная операция минус. Такой подход порождает некоторые особенности, что побудило сообщество разработчиков создать расширение для языка, добавляющее настоящие отрицательные литералы.

Как выяснилось, способ, которым было реализовано данное расширение, не позволял корректно обрабатывать значение “отрицательный ноль”. `-0` компилятор неверно распознавал как обыкновенный, положительный ноль. Такое поведение было воспринято как ошибка. Нами были рассмотрены несколько путей решения данной проблемы. Один из этих способов оказался приемлем для разботчиков компилятора. Изменения подготовленные нами были приняты в основной репозиторий.

1 Литералы в Haskell

1.1 Классификация

В Хаскель можно выделить четыре вида литералов: целочисленные, вещественные, символьные и строковые.

В отличие от многих других языков, логические значения `True` и `False` являются идентификаторами. Они входят в состав стандартного модуля `Prelude` и определены как показано в листинге 1.1.

Листинг 1.1 Определение значений True и False

```
data Bool = True | False
```

Целочисленные литералы представляют значения натуральных чисел, включая ноль. Существует синтаксис для записи в различных системах счисления: десятичной, восьмеричной, шестнадцатеричной. Примеры: `15`, `0o17`, `0O17`, `0xF`, `0XF`.

$$
\begin{array}{lcl}
\mathit{decimal} & \rightarrow & \mathit{digit}\{\mathit{digit}\} \\
\mathit{octal} & \rightarrow & \mathit{octit}\{\mathit{octit}\} \\
\mathit{hexadecimal} & \rightarrow & \mathit{hexit}\{\mathit{hexit}\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{integer} & \rightarrow & \mathit{decimal} \\
 & | & \texttt{0o}\ \mathit{octal} \mid \texttt{0O}\ \mathit{octal} \\
 & | & \texttt{0x}\ \mathit{hexadecimal} \mid \texttt{0X}\ \mathit{hexadecimal}
\end{array}
$$

Рис. 1: Лексическая структура целочисленных литералов[1]

Вещественные литералы представляют рациональные числа. Можно использовать как обычные десятичные дроби, так и научную форму записи с указанием мантиссы и порядка. Примеры: `123.456`, `0.123456e3`.

$$
\begin{array}{lcl}
\mathit{float} & \rightarrow & \mathit{decimal}\ \texttt{.}\ \mathit{decimal}\ [\mathit{exponent}] \\
 & | & \mathit{decimal}\ \mathit{exponent} \\
 & & \\
\mathit{exponent} & \rightarrow & (\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \mathit{decimal}
\end{array}
$$

Рис. 2: Лексическая структура вещественных литералов[1]

Символьные литералы записываются в одинарных кавычках, а строковые в двойных. Примеры: `'a'`, `"Hello, World!"`.

$$
\begin{array}{lcl}
\mathit{char} & \rightarrow & \texttt{'}\ (\mathit{graphic}_{\langle \texttt{'} \mid \backslash \rangle} \mid \mathit{space} \mid \mathit{escape}_{\langle \backslash\, \& \rangle})\ \texttt{'} \\
\mathit{string} & \rightarrow & \texttt{"}\ \{\mathit{graphic}_{\langle \texttt{"} \mid \backslash \rangle} \mid \mathit{space} \mid \mathit{escape} \mid \mathit{gap}\}\ \texttt{"} \\
\mathit{escape} & \rightarrow & \backslash\ (\ \mathit{charesc} \mid \mathit{ascii} \mid \mathit{decimal} \mid \texttt{o}\ \mathit{octal} \mid \texttt{x}\ \mathit{hexadecimal}\ ) \\
\mathit{charesc} & \rightarrow & \texttt{a} \mid \texttt{b} \mid \texttt{f} \mid \texttt{n} \mid \texttt{r} \mid \texttt{t} \mid \texttt{v} \mid \backslash \mid \texttt{"} \mid \texttt{'} \mid \texttt{\&} \\
\mathit{ascii} & \rightarrow & \texttt{\^{}}\ \mathit{cntrl} \mid \texttt{NUL} \mid \texttt{SOH} \mid \texttt{STX} \mid \texttt{ETX} \mid \texttt{EOT} \mid \texttt{ENQ} \mid \texttt{ACK} \\
 & | & \texttt{BEL} \mid \texttt{BS} \mid \texttt{HT} \mid \texttt{LF} \mid \texttt{VT} \mid \texttt{FF} \mid \texttt{CR} \mid \texttt{SO} \mid \texttt{SI} \mid \texttt{DLE} \\
 & | & \texttt{DC1} \mid \texttt{DC2} \mid \texttt{DC3} \mid \texttt{DC4} \mid \texttt{NAK} \mid \texttt{SYN} \mid \texttt{ETB} \mid \texttt{CAN} \\
 & | & \texttt{EM} \mid \texttt{SUB} \mid \texttt{ESC} \mid \texttt{FS} \mid \texttt{GS} \mid \texttt{RS} \mid \texttt{US} \mid \texttt{SP} \mid \texttt{DEL} \\
\mathit{cntrl} & \rightarrow & \mathit{ascLarge} \mid \texttt{@} \mid \texttt{[} \mid \backslash \mid \texttt{]} \mid \texttt{\^{}} \mid \texttt{\_} \\
\mathit{gap} & \rightarrow & \backslash\ \mathit{whitechar}\ \{\mathit{whitechar}\}\ \backslash
\end{array}
$$

Рис. 3: Лексическая структура символьных и строковых литералов[1]

1.2 Отрицательные числа

Как можно заметить из приведённых фрагментов стандарта, знак не является частью лексемы обозначающей число. Тем не менее запись `-10` является синтаксически правильной. Минус в данном случае является отдельной лексемой, обозначающей операцию унарный минус. Во время компиляции он заменяестя вызовом функции `negate`, определённой в классе `Num` стандартного модуля `Prelude`.

Вместе с другими особенностями синтаксиса и системы типов это рождает некоторые трудности.

1.3 Особенность 1: вызов функции записывается без скобок

Функции играют в языке Хаскель важную роль. Для такого частого действия, как вызов функции, разработчики решили сделать максимально простой синтаксис. Что бы вызвать фукнцию `f` с аргументом `x`, достаточно просто написть `f x`.

Листинг 1.2 демонстрирует пробему, связанную с отсутствием необходимости писать скобки вокруг аргументов. Выражение `print -1` воспринимается компилятором, как вызов функции `print` с двумя аргументами: `-` и `1`. Такое действие невозможно, о чём нам любезно сообщает компилятор. Вероятно, программист хотел напечатать число минус один. Для того что бы сделать это, ему необоходимо обернуть `-1` в скобки, то есть написать `print (-1)`.

Листинг 1.2 Ошибка при отсутствии скобок вокруг унарного минуса

```
Prelude> print (-1) -- OK
Prelude> print -1   -- ERROR

<interactive>:2:1:
    Non type-variable argument in the constraint:
                                  Num (a -> IO ())
    (Use FlexibleContexts to permit this)
    When checking that ‘it’ has the inferred type
      it :: forall a. (Num (a -> IO ()), Show a)
            => a -> IO ()
```

Стоит сказать, что многие программисты на Хаскель стараются не писать лишних скобок. Для них необходимость оборачивать отрицательные числа в скобки может может быть раздражающей и даже мешающей, ухудшающей читаемость кода.

1.4 Особенность 2: тип значения представляемого литералом определяется контекстом

Система типов языка Хаскель позволяет литералам иметь разный тип в разных контекстах. Сами по себе целочисленные и вещественные ли-

тералы не имеют конкретного предопределённого типа. Он определяется во время компиляции в каждом месте использования таким образом, чтобы удовлетворять наложенным ограничениям. К примеру, если функция имеет параметр типа `Int8`, и мы передадим ей в качестве аргумента число `42`, то компилятор определит его тип как `Int8`.

Листинг 1.3 Полиморфные литералы

```
Prelude> :t 42
42 :: Num a => a
Prelude> :t 3.14
3.14 :: Fractional a => a
```

Вместе с отсутствием отрицательных литералов такая особенность может порождать ошибки или как минимум замедлять работу кода. Такую ситуацию демонстрирует листинг 1.4. Рассмотрим более подробно, что происходит в процессе компиляции этого фрагмента. Для выражения `-128` указан тип `Int8`. Таким должен быть результат функции `negate`, применённой к литералу `128`. Её тип — `Num a => a -> a`, то есть её параметр и результат должны быть одного типа, поэтому `128` тоже будет иметь тип `Int8`. Теперь обратим внимание на значения, представимые в этом типе: `-128..127`. Число `128` не входит в этот отрезок, поэтому произойдёт переполнение в значение `-128`. Во время выполнения к нему будет применена функция `negate`, результатом которой было бы число `128`, если бы оно было представимо в этом типе. На самом деле произойдёт ещё одно переполнение в значение `-128`.

Листинг 1.4 Переполнения на граничных значениях

```
Prelude Data.Int> print (-128::Int8)
<interactive>:3:9: Warning:
    Literal 128 is out of the Int8 range -128..127
    If you are trying to write a large negative
    literal, use NegativeLiterals
-128
```

Такое количество переполнений безусловно нежелательно и может привести к потере в производительности. Конечно, в данном случае мы по-

лучили желаемый результат: напечатано число -128, но в случае с другими типами, для которых реализован класс Num, результат может быть совершенно неожиданным.

1.5 Особенность 3: частичное применение операторов

Ещё одной особенностью языка Хаскель является синтаксис частичного применения инфиксных операторов. Проще всего её продемонстрировать на примере. + является бинарным инфиксным оператором. Результат выражения 2+2, как ни парадоксально, равен 4. Но можно опустить один из аргументов. В таком случае результатом будет не число, а функция. 2+ эквивалентно $\lambda x \to 2+x$. Аналогично +2 эквивалентно $\lambda x \to x+2$.

Стоит сказать, что - является единственным унарным оператором. Вероятно, это связано с тем, что он создаёт коллизии с синтаксисом частичного применения операторов. Без дополнительных правил невозможно определить, что означает выражение -2: функцию вычитающую двойку из своего аргумента или число минус два. В этом случае действует правило считать такое выражение числом.

Листинг 1.5 Унарный минус — особый случай

```
Prelude> (+2) 2
4
Prelude> (*2) 2
4
Prelude> (/2) 2
1.0
Prelude> (2-) 2
0
Prelude> (-2) 2

<interactive>:10:1:
    Non type-variable argument in the constraint:
                                      Num (a -> t)
    (Use FlexibleContexts to permit this)
    When checking that ‘it’ has the inferred type
      it :: forall a t. (Num a, Num (a -> t)) => t
```

2 Расширение NegativeLiterals

3 Проблема обработки отрицательного нуля в расширении NegativeLiterals

Список литературы

[1] Simon Marlow(editor), *Haskell 2010 Language Report*