САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Алиев Мирза Али оглы

Обзор методов совместного задания синтаксических анализаторов и принтеров для различных языков программирования

Курсовая работа

Научный руководитель:
асп. Подкопаев А.В.

Санкт-Петербург
2015

Оглавление

Введение

Во многие языковые процессоры входит синтаксический анализатор. Он получает на вход последовательность токенов, сравнивает её с грамматикой исходного языка и, если не появилось никаких ошибок, строит древовидную структуру программы, называемую синтаксическим деревом (см. рисунки 1, 2)

```
while(a<b){
    printf("a is less than b");
}
```

Рис. 1: Код программы.

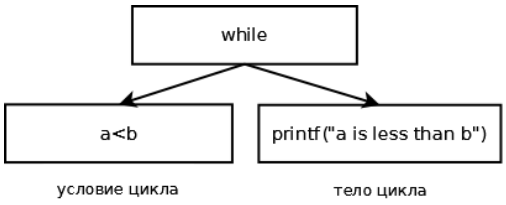

Рис. 2: Синтаксическое дерево.

Помимо реализации синтаксического анализатора языка, часто реализуется так называемый pretty-printer, который в дальнейшем будет называть принтером, а сам процесс его работы – печатью. Он позволяет получить текстовое представление программы на основе синтаксического дерева. Однако у программы может быть несколько представлений, и то, какой код получится, играет немаловажную роль. На рисунке 3 представлен код неотформатированной программы, на рисунке 4 тот же самый код, но уже с добавлением отступов, переносов строк, пробелов в определённых местах, что намного облегчает восприятие программы.

```
int foo(int a, int b){while(a<b)
{printf(''a is less than b'')
;a=a+1;}return a;}
```

Рис. 3: Неотформатированная программа

```
int foo(int a, int b)
{
    while(a<b)
    {
        printf("a is less than b");
        a = a+1;
    }
    return a;
}
```

Рис. 4: Форматированная программа

В большинстве случаев синтаксический анализатор и принтер никак не связаны между собой и являются двумя разными программами, хотя выполняют они функции,

обратные друг к другу, то есть для них можно ввести следующий инвариант: пусть имеется синтаксический анализатор

```
parser :: String -> AST
```

и принтер,

```
printer :: AST -> String
```

Все строки, которые равны с точки зрения синтаксиса языка, будут давать одно и тоже синтаксическое дерево, однако по одному и тому же синтаксическому дереву принтер может получить разные строки, отличающиеся, например, количеством пробелов как на рисунках 3, 4. Из этого следует, что композиция синтаксического анализатора и принтера будет тождественной функцией,

```
parser . printer = id
```

а обратная композиция не является тождетсвенной.

```
printer . parser /= id
```

Для разработки синтаксических анализаторов и принтеров на языке Haskell довольно часто используются библиотеки и встраиваемые предметно-специфичные языки (embedded domain-specific languages, EDSLs). Например, стандартные библиотеки ghc включают в себя библиотеку Parsec, встроенный синтаксический анализатор DSL (Leijen и Meijer 2001) для синтаксического анализа, а также EDSL для реализации принтеров (Hughes 1995). Эти библиотеки независимы между собой и сложны в реализации, так как от них требуется высокая скорость работы и корректность. Все это влечёт за собой несоблюдение инварианта принтера и синтаксического анализатора.

Возникает вопрос о совместном задании принтера и синтаксического анализатора. У этого подхода есть ряд положительных качеств. Во-первых, это позволит не реализовывать две схожие программы отдельно. Как следствие, уменьшается вероятность появления ошибок, связанных с их инвариантом. Кроме того, такой подход позволит не переносить изменения, сделанные в одном из компоненте, на другой. Во-вторых, все строки, которые генерируются принтером, корректно воспринимаются синтаксическим анализатором. В-третьих, инвариантность принтера и синтаксического анализатора получается автоматически за счёт задания их как единой структуры.

В данной работе были рассмотрены три статьи, предоставляющие возможности совместного задания принтера и синтаксического анализатора для формальных языков [7], [2], [3].

Целью данной работы является обзор существующих статей в данной предметной области, выявление их положительных и отрицательных сторон, а именно: как сильно можно варьировать вывод принтера, можно ли использовать параметры вывода, например, ширина. Для этих целей на основе каждой статьи реализуется синтакси-

ческий анализатор и принтер для учебного языка L. На рисунках 5, 6 показаны два различных представления кода в зависимости от того, какая ширина вывода.

```
printf("a is less than b"); a = b;
```

Рис. 5: Представление программы при ширине вывода больше 34

```
printf("a is less than b");
a = b;
```

Рис. 6: Представление программы при ширине вывода меньше 34

1. Обзор статьи T. Rendel, K. Ostermann

Авторами данной статьи был разработан EDSL язык синтаксических дескрипторов (syntax descriptions) для объединения синтаксического анализа и печати, основанный на частичных изоморфизмах (partial isomorphisms), с помощью которых можно осуществлять обратимые вычисления. Авторы предоставляют интерфейс синтаксических дескрипторов как множество классов типов (type classes). Также они разработали единые комбинаторы для синтаксического анализа и печати.

Рассмотрим пример. На рисунке 7 представлен результат обработки строки "((SK)K)" на языке **data** SK = S | K | App SK SK синтаксическим анализатором.

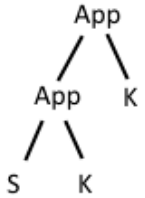

Рис. 7: Синтаксическое дерево.

Программа, реализующий синтаксический анализатор и принтер выглядит следующим образом.

```
parserSK = exp1 where
    exp0 = s <$ tok "S"
       <|> k <$ tok "K"
       <|> tok "(" *> exp1 <* tok ")"
    exp1 = foldl app <$> exp0 <*> many exp0
    tok k = string k <* spaces
```

Рис. 8: Синтаксический анализатор и принтер

Рассмотрим комбинатор (<$>), как если бы он был реализован в библиотеке, которая осуществляет лишь синтаксический анализ. Этот комбинатор имел бы тип $(\alpha \to \beta) \to (Parser\ \alpha \to Parser\ \beta)$ и позволял бы получить синтаксический анализатор некоторого типа β, имея синтаксический анализатор типа α и функцию из α в β. Теперь рассмотрим этот же комбинатор с точки зрения принтера. Он имел бы тип $(\beta \to \alpha) \to (Printer\ \alpha \to Printer\ \beta)$

Как видно, два типа очень похожи, за исключением того, что в одном случае первый аргумент имеет тип $\alpha \to \beta$, а во втором $\beta \to \alpha$.

Для обобщения комбинаторов для синтаксического анализа и печати авторами вводится понятие частичного изоморфизма. Заводится конструктор типа данных Iso, такой, что Iso α β это тип частичного изоморфизма между α и β

```
data Iso α β
    = Iso (α → Maybe β) (β → Maybe α)
```

Слово частичный здесь не случайно, как видно из определения конструктора Iso f g, функции f и g по входному параметру могут получить Nothing.

Теперь функции могут быть использованы как вперед, так и в обратную сторону. Например, модифицированный комбинатор <$>, выглядящий следующим образом:

$$(<\$>) :: Syntax\ \delta \Longrightarrow Iso\ \alpha\ \beta \to (\delta\ \alpha \to \delta\ \beta)$$

выполняющий следующие функции: f<$>$p$ синтаксический анализатор будет использовать функцию f вперед для того, чтобы конвертировать значения после синтаксического анализа, а f<$>$p$ принтер будет использовать f в обратную сторону перед печатью. Однако, не все функции могут быть использованы при частичном изоморфизме, а только те, которые могут быть обратимы. Например, авторы реализуют обратимую функцию свёртки

```
foldl :: (α → β → α) → α → [β] → α

foldl ::  Iso (α, β) α → Iso (α, [β]) α
```

При обращении функции **foldl** конструкторам типа Iso получится функция **unfoldl**

Таким образом, авторами был разработан интерфейс, который позволяет совместно задать синтаксический анализатор и принтер (см. рисунок refКлассСинтДеск).

```
class (IsoFunctor δ, ProductFunctor δ, Alternative δ)
   ⇒ Syntax δ where
  -- (<$>) :: Iso α β → δ α → δ β
  -- (<*>) :: δ α → δ β → δ (α, β)
  -- (<|>) :: δ α → δ α → δ α
  -- empty  :: δ α
  pure   :: Eq α ⇒ α → δ α
  token  :: δ Char
```

Рис. 9: Класс синтаксических дескрипторов

Дескриптор token связывает символ с самим собой, а pure x в случае синтаксического анализа возвращает x, не требуя какого-либо ввода, а pure x, в случае принтера, отбрасывает значения, равные x. Используя этот интерфейс, можно реализовать функцию, которая является и синтаксическим анализатором, и принтером.

В данном подходе никоим образом не задается вариативность принтера в зависимостиот ширины вывода. Однако при реализации данного подхода для учебного языка L было выяснено, что можно задать функцию от булевого флага, с помощью которой можно варьировать количество пробелов, переносов строк, отступов. Ниже представлена часть программы, реализующий флаг, и пример работы этого флага.

```
optSpace' :: Syntax delta => Bool -> delta ()
optSpace' True  =  ignore [()]  <$>  (many (text ''\n   '') <|> many (text '' ''))
optSpace' False =  ignore [()]  <$> (many (text '' '') <|> many (text ''\n''))

statement = stm 1 where
    stm 0 = readStm <$> readstm
        <|> write <$> write'
        <|> while <$> while'
        <|> ifExp <$> ifexp

    stm 1 = (seqStm <$> (stm 0 <*> text '';'' *> stm 1))
        <|> stm 0
    readstm = keyword ''read''
            *> parens (identifier)
    write'  = keyword ''write''
            *> parens (expression)
    while' = keyword ''while''
            *> parens (expression) <*> optSpace' False *> (statement)
    ifexp  = keyword ''if''
            *> optSpace  *> parens (expression)
                                <*> optSpace  *> (statement)
                                <*> optSpace  *> keyword ''else''
                                 *> optSpace  *> (statement)
```

Рис. 10: Реализация булевого флага

При выставлении у функции optSpace' параметра True, происходит форматирование, как на рисунке 18, а при параметре False, как на рисунке 19

```
while(1)
    read(x)
```

Рис. 11: Флаг равен True

```
while(1)  read(x)
```

Рис. 12: Флаг равен false

Однако мы стремимся получить вариатиность не от булевого флага, а в зависимости от ширины вывода. Для того, чтобы появилась такая возможность, нужно либо придумывать более сложные структуры, нежели флаг, либо обернуть получение параметров вывода в монадные вычисления. Однако функции, осуществляющие данные вычисления, довольно трудоёмко реализовать обратимыми.

Одним из ограничений данного метода является использование расширений TemplateHaske NoMonomorphismRestriction, RelaxedPolyRec языка Haskell.

2. Обзор статьи K. Matsuda M. Wang

В данной статье авторы разработали систему FliPpr для получения синтаксического анализатора из принтера. Одним из важных элементов системы FliPpr является язык Surface Language, оснащённый комбинаторами библиотеки Вадлера для печати [5], с помощью которого реализуется принтер. Далее программа на языке Surface Language преобразуется в язык Core Language с помощью системы FliPpr. Следующим этапом работы системы является генерация контексно-свободной грамматики по программе на Core Language. В итоге полученная грамматика передается парсер-генератору и получается синтаксический анализатор (рис. 13)

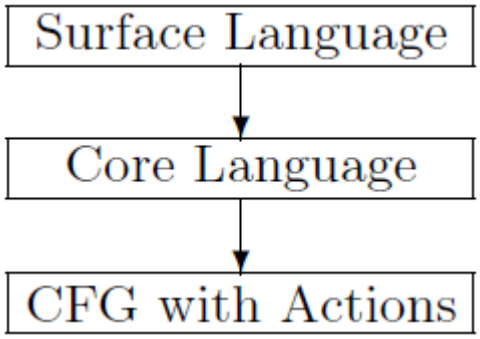

Рис. 13: Схема работы FliPpr

Рассмотрим работу систему FliPpr на примере. На рисунке 14 представлен принтер для небольшого языка **data** E = Sub E E | One.

```
ppr x = ppr x <+ text "(" <> nil <> ppr x <> nil <> text ")"
ppr One = text "1"
ppr (Sub e1 e2) = group (ppr e1 <>
                    nest 2 (line' <> text "-" <> space' <> pprP e2))
pprP x = pprP x <+ text "(" <> nil <> pprP x <> nil <> text ")"
pprP One = text "1"
pprP (Sub e1 e2) = text "(" <> nil <> group (ppr e1 <> nest 2 (line'
                  <> text "-" <> space' <> pprP e2)) <> nil <> text ")"
space' = space <+ text ""
line' = line <+ text ""
```

Рис. 14: Пример принтера

Помимо вадлеровских комбинаторов для печати group, nest, авторами вводится комбинатор выбора (<+) (Biased Choice) Он нужен для того, чтобы синтаксический анализатор мог воспринимать не только строчки, которые производит принтер. Принцип действия этого комбинатора следующий: когда программа воспринимается как принтер, то выполняется действие лишь левого операнда (<+). Однако с точки зрения синтаксического анализатора этот комбинатор интерпретируется как недетерме-

нированный выбор, который принимает как левый, так и правый операнд. Например, можно определить количество пробелов и переносов строк следующим образом:

$$nil = text\ ""\ <+\ space$$
$$space = (text\ "\ "\ <+\ text\ "\backslash n")\ <>\ nil$$

где *nil* позволял бы воспринимать ноль или больше пробелов с переносом строки во время синтаксического анализа и получать ноль пробелов во время печати, а *space* позволял бы воспринимать один или больше пробелов с переносом строки во время синтаксического анализа и получать один пробел во время печати. Например, строки вида $1\quad -1$, $(1)-((1))$, $(1-(1))$ анализируются корректно.

Данный подход поулчения синтаксического анализатора из принтера основывается на предыдущей работе авторов об grammar-based inversion [1]. Используя идеи из статьи [1], программа на языке Core Language может быть инвертирована для получения контекстно-свободной грамматики особого вида (CFG with actions)(см. рис. 15) исходного языка. Далее в программе FliPpr авторами используется парсер-генератор, основанный на статье автора Frost и др. [6] для получения синтаксического анализатора. Однако для того, чтобы инверсия была возможна, на программу на языке Core Language накладываются ограничения, а именно программа должна быть linear и treeless [4]. Вследствие этих ограничений, нельзя релизовать хороший принтер с гибкими настройками форматирования, поэтому был разработан Surface Language, но уже без выше указанных ограничений, дающий возможности хорошей печати, а далее с помощью техник программной трансформации deforestation [4] и др. программа на языке Surface Language переводится в программу на Core Language.

```
PPr → Ppr_                          {$ 1}
    | "(" Nil Ppr Nil ")"           {$ 3}
Ppr_→ 1                                    {$ One}
    | Ppr Line' "−" Space' PprP             {Sub $1 $5 }
...
```

Рис. 15: Пример CFG with actions

На рисунке 16 представлена часть реализации принтера на яыке Surface Language для учебного языка L, которая обрабатывает цикл while.

Достоинством данного метода является то, что для принтера используются комбинаторы из библиотеки Вадлера. Это предоставляет широкие возможности для форматирования, в том числе с использование ширины вывода. Также стоит отметить вариативность в плане выбора парсер-генератора. Однако ограничения, которые накладываются на Core Language, могут в перспективе вызвать ограничения при реализации синтаксических анализаторов и принтеров для сложных структур.

```
ppr' i d (ExWhileZ _ x y ) = parensIf (d == L) $ group $
    text "while" <> nil <> space' <> ppr 0 R x <> line <>
    text "do" <> space <> nest 3 (ppr 0 R y);

parensIf b d = if b then parens d else d ;
manyParens d = d <+ parens (manyParens d);
parens d = text "(" <> nest 1 (nil <> d <> nil) <> text ")";
space = (text " " <+ charOf (spaceChars `butnot` char ' ')) <> nil;
nil = text "" <+ (charOf spaceChars <> nil);
```

Рис. 16: Обработка цикла while для языка L

3. Обзор статьи M. Boespflug

В данной статье авторы представляют EDSL с семейством обратимых комбинаторов для совместного задания синтаксического анализа и печати.

Для реализации данного подохода авторы разработали тип под названием кассета (Cassette). Кассета состоит из двух треков, которые представлены функциями.

```
data K7 a b c d = K7 { sideA :: a -> b, sideB :: d -> c }
```

Кассеты можно склеивать, для получения новых.

```
(<>) :: K7 b c b' c' -> K7 a b a' b' -> K7 a c a' c'
~(K7 f f') <> ~(K7 g g') = K7 (f . g) (g' . f')
```

Кассета, которая на стороне A содержит синтаксический анализатор, продуцирующий данные типа a, и котора на стороне B имеет принтер данных типа a, называется P/P парой:

```
type PP a = forall r r'. K7 (C (a -> r)) (C r) (C (a -> r')) (C r')
```

где конструктор C имеет тип **type** C r = (**String** -> r) -> **String** -> r

С кассетами можно делать следующие действия. Их можно проиграть:

```
play :: K7 a b c d -> a -> b
play csst = sideA csst
```

Можно поменять стороны A и B местами:

```
flip :: K7 a b c d -> K7 d c b a
flip (K7 f g) = K7 g f
```

Простейший синтаксический анализатор из кассеты получается следующим образом:

```
parse :: PP a -> String -> Maybe a
parse csst = play csst (\_ _ x -> Just x) (const Nothing)
```

Если перевернуть кассету, то получится принтер:

```
pretty :: PP a -> a -> Maybe String
pretty csst = play (flip csst) (const Just) (\_ _ -> Nothing) ""
```

Комбинатор (<|>) позволяет выбирать последовательно между несколькими кассетами. Если одна из них, например, не может проанализировать входную строку, то выбирается следующая кассета.

```
(<|>) :: PP a -> PP a -> PP a
K7 f f' <|> K7 g g' =
  K7 (\k k' s -> f k (\s' -> g k k' s) s)
     (\k k' s x -> f' k (\s' -> g' k k' s) s x)
```

Основной особенностью данного подхода является то, что симметричность синтаксического анализатора и принтера реализуется с помощью задания их индуктивно (Continuation-passing style, CPS). По мнению авторов, такой способ задания комбинатора выбора (<|>) показывают более лучшие результаты в плане производительности, нежели с помощью методов, описанных в первой статье [7], где были использованный вложенные кортежи.

Данный метод никак не позволят варьировать работу принтера в зависимости от ширины вывода. Как и в случае с первой статьей, был реализован синтаксический анализатор и принтер с булевым флагом, в зависимости от которого продуцировался разный вывод. Ниже представлена часть программы, реализующая флаг и результаты работы программы в зависимости от параметра.

При выставлении у функции optSpace' параметра True, происходит форматирование, как на рисунке 18, а при параметре False, как на рисунке 19.

Ограничением данного подхода является то, что он использует расширение RankNTypes языка Haskell.

```
optSpace' ::  Bool -> PP0
optSpace' True = unshift ''\n   '' $ many (satisfy isSpace)
optSpace' False = unshift '' '' $ many (satisfy isSpace)

stmnt :: PP Stmt
stmnt =   seqL    -->
          parens(stmnt<> optSpace <> string '';'' <> optSpace <> stmnt)
      <|> readL   -->
          string ''read'' <> parens(ident)
      <|> writeL  -->
          string ''write'' <> parens(optSpace <> mainExpr <> optSpace)
      <|> whileL  -->
     string ''while'' <> parens(optSpace <> mainExpr <> optSpace) <> optSpace' True <>
       optSpace <> stmnt <> optSpace
      <|> assignL -->
          ident <> optSpace <> string ''='' <> optSpace <> mainExpr
```

Рис. 17: Реализация булевого флага

```
while( 1 )
   read(x)
```

Рис. 18: Флаг равен True

```
while( 1 ) read(x)
```

Рис. 19: Флаг равен false

4. Сравнение предложенных подходов

Методы, предложенные в первой и третьей статье, являются похожими друг на друга, у обоих методов совпадают некоторые комбинаторы, в обоих случаях никак не используются параметры вывода. Однако проблема обратимости функций, описанная в первой статье, была решена в третьей за счёт задания синтаксического анализатора и принтера в CPS. Однако оба этих метода используют некоторые расширения языка Haskell. Во второй статье используется другая концепция с использованием вадлеровских комбинаторов для печати, которая предоставляет большие возможности для вариативности принтера, в том числе и параметры ввода и вывода.

	Syntax Descriptions	FliPpr	Cassette
Использование ширины вывода	Нет	Да	Нет
Что нужно реализовать	Синтаксический анализатор	Принтер	Синтаксический анализатор
Язык L (строк кода)	Около 250	Около 60	Около 230
Ограничения	Использование расширений языка Haskell, обратимость функций	Ограничения на Core Language	Использование расширений языка Haskell

Также авторами третьей статьи утверждалось, что их подход лучше подхода, описанного в первой статье, в плане производительности. Для проверки данного предположения был произведён замер скорости работы реализаций для языка L. Для каждой программы генерировалось одинаковое синтаксическое дерево с разным количеством элементов. Замер производился отдельно для синтаксического анализа и печати. Результаты представлены в таблице ниже.

	Syntax Descriptions	Cassette
100 элементов	3.1127e-2	1.092e-3
1000 элементов	1.228305	7.031e-3
2000 элементов	5.772979	1.3437e-2

Рис. 20: Производительность для синтаксического анализа

	Syntax Descriptions	Cassette
100 элементов	5.282e-3	9.91e-4
1000 элементов	3.2422e-2	1.0082e-2
10000 элементов	0.452966	8.1065e-2

Рис. 21: Производительность для печати

Как видно из таблиц, предположения авторов третьей статьи об улучшении производительности подтвердились.

Заключение

В данной работе представлен обзор статей на тему совместного задания синтаксического анализатора и принтера. Было выяснено, что только один из подходов использует параметры вывода, например ширину. Для каждой из статей был реализован синтаксический анализатор и принтер для учебного языка L и было выяснено, что некоторая вариативность для двух других методов может быть задана, однако она никак не использует параметры вывода. Также был произведён замер производительности двух подходов для подтверждения улучшения производительности.

Список литературы

[1] A Grammar-based Approach to Invertible Programs. / Matsuda K., Mu S., Hu Z., Takeichi M. // ESOP 2010: 19th European Symposium on Programming. — 2010. — P. 448—467.

[2] K. Matsuda, M. Wang. FliPpr: A Prettier Invertible Printing System. // ESOP 2013: 22nd European Symposium on Programming. — 2013. — P. 101—120.

[3] M. Boespflug. Rewinding the stack for parsing and pretty printing // Unpublished. — 2012.

[4] P. Wadler. Deforestation: transforming programs to eliminate trees. // 2'nd European Symposium on Programming. — 1990. — Vol. 73. — P. 231–248.

[5] P. Wadler. A prettier printer // Gibbons, J., de Moor, O., eds.: The Fun of Programming. — 2003. — P. 223–244.

[6] R.A. Frost, R. Hafiz, P Callaghan. Parser Combinators for Ambiguous Left-Recursive Grammars // PADL. LNCS. — 2008. — Vol. 4902. — P. 167–181.

[7] T. Rendel, K. Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. // Haskell 2010: Proceedings of the 2010 ACM SIGPLAN Haskell Symposium. — 2010. — P. 1–12.