Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 19Б.13-мм

СУСЛОВ Алексей Витальевич

Синтаксический анализатор Reason с восстановлением от ошибок

Отчёт по учебной практике

Научный руководитель:
к.ф.-м.н., доцент кафедры информатики Григорьев С.В.

Консультант:
разработчик ООО «ИнтеллиДжей Лабс» Косарев Д.С.

Санкт-Петербург
2020

Оглавление

1. Введение

Reason — это расширение синтаксиса OCaml и новый набор инструментов, созданные в Facebook в 2016 году. Для языка был придуман C-подобный синтаксис, так же похожий и на JavaScript, что бы программистам, пишущим на этих языках, было легче адаптироваться [?]. Одной из целей создания нового языка было упрощение взаимодействия с JavaScript, добавление поддержки JSX и возможности компиляции в JavaScript [?]. В наследство от OCaml Reason-у досталась возможность компилироваться в нативный и байт-коды, а так же система типов со всеми преимуществами, такими как уменьшение количества ошибок в программах и повышение удобства поддержки кода.

Однако существует проблема у людей, уже пишущих на OCaml. Так как Reason остался достаточно похожим на него, в языках имеются одинаковые синтаксические конструкции, отличающиеся только оформлением. Из-за этого довольно сложно переключиться на новый синтаксис, и, как следствие, программист по ошибке может написать вполне корректный для OCaml участок кода, но не подходящий для Reason. В таком случае стоит указать человеку на его ошибку предупреждением и продолжить синтаксический анализ программы, корректно прочитав конструкцию из чужого языка.

Таким образом, актуальной задачей является добавление новых правил в синтаксический анализатор Reason с восстановлением от типичных ошибок, вызванных смешением языков.

2. Постановка задачи

Целью данной работы является дополнение существующего парсера Reason новыми правилами восстановления от ошибок, вызванных смешением синтаксиса Reason и OCaml.

Для её выполнения были поставлены следующие задачи:

1. Исследовать предметную область:
   (a) Рассмотреть принципы работы LR(1) анализаторов.
   (b) Рассмотреть принцип работы генераторов синтаксических анализаторов.
   (c) Изучить существующую реализацию синтаксического анализатора на OCaml (Menhir).
2. Реализовать дополнительные правила восстановления от ошибок.
3. Оценить качество восстановления от ошибок.

3. Обзор

3.1. Синтаксические анализаторы

Синтаксический анализ или парсинг (parsing) — процесс сопоставления последовательности токенов (лексем) дерева разбора, называемого абстрактным синтаксическим деревом (AST).

Синтаксический анализатор или парсер (parser) — это программа, выполняющая синтаксический анализ.

Так же от синтаксического анализатора ожидаются сообщения обо всех выявленных ошибках, причем достаточно внятных и полных, а кроме того, умение обрабатывать обычные, часто встречающиеся ошибки и продолжать работу с оставшейся частью программы. Имеется три основных типа синтаксических анализаторов: универсальные, восходящие и нисходящие [**?**]. Мы остановимся на восходящем анализе. Восходящие синтаксические анализаторы строят дерево разбора начиная с листьев (снизу) и идя к корню (вверх). Поток символов сканируется последовательно — слева направо.

Здесь будет рассмотрен общий вид восходящего анализа типа «перенос/свёртка» (shift-reduce). При анализе типа перенос-свёртка для хранения символов грамматики используется стек, а для хранения остающейся непроанализированной части входной строки — входной буфер. На каждом шаге свёртки (reduction) определенная подстрока, соответствующая правой части продукции, заменяется на нетерминал, являющийся левой частью продукции. Основа — это подстрока, которая соответствует телу продукции и свертка которой представляет собой один шаг правого порождения в обратном порядке. Использование стека в анализаторе объясняется тем важным фактом, что *основа* всегда находится на вершине стека и никогда — внутри него.

LR(k)-грамматики — это грамматики, для которых может быть построен синтаксический анализатор, работающий по принципу переноса-свёртки.

Наиболее эффективные восходящие методы работают только с под-

классами грамматик, однако некоторые из этих классов, такие как LR(k) грамматики, достаточно выразительны для описания большинства синтаксических конструкций языков программирования. L — здесь означает сканирование входного потока слева направо, R — построение правого порождения в обратном порядке, а k — количество предпросматриваемых символов входного потока, необходимых для принятия решения.

Одними из самых эффективных синтаксических анализаторов являются LR(1) [**?**]. LR(1)-анализатор состоит из входного буфера, выхода, стека, программы-драйвера и таблицы синтаксического анализа. Программа-драйвер одинакова для всех LR-анализаторов, от одного анализатора к другому меняются таблицы синтаксического анализа. Программа ситаксического анализатора по одному считывает символы из входного буфера. После прочтения очередного символа она обращается к управляющей таблице и совершает соответствующее действие. Процесс чтения продолжается, пока входная цепочка не закончится.

3.2. Menhir

Построение анализаторов LR(1) вручную достаточно трудоемкий процесс, поэтому чаще всего пользуются генераторами синтаксических анализаторов.

Генератор синтаксических анализаторов — это программа, которая по спецификации грамматики строит синтаксический анализатор. Одним из таких генераторов и является Menhir [**?**]. Он был выбран командой Reason, как лучшая, по сравнению с ocamlyacc, версия генераторов синтаксических анализаторов.

3.3. Примеры ошибок

Здесь будут приведены примеры синтаксических конструкций, в которых часто встречаются ошибки, в формате сравнения кода на OCaml и Reason соответственно.

```
let greeting person =
  match person with
  | Teacher   →  "Hey␣Professor!"
  | Director   →  "Hello␣Director."
  | Student ("Andrew")  →  "Still␣here␣Ilya?"
  | Student (anyOtherName)  →  "Hey,␣" ^ anyOtherName ^ "."
```

1.a

```
let greeting = person ⇒
  switch (person) {
  | Teacher ⇒ "Hey␣Professor!"
  | Director ⇒ "Hello␣Director."
  | Student("Andrew") ⇒ "Still␣here␣Ilya?"
  | Student(anyOtherName) ⇒ "Hey,␣" ++ anyOtherName ++ "."
  };
```

1.b

Листинг 1

На листинге **??** представлена конструкция сопоставления с образцом (pattern matching). Программист может перепутать оформление стрелок (-> и =>), выбрать ключевые слова из другого языка (match - with и switch(-)) или забыть поставить фигурные скобки и точку с запятой.

```
for i = 1 to 10 do
  print_int i;
  print_string ("␣")
done
```

2.a

```
for (i in 1 to 10) {
  print_int(i);
  print_string("␣");
};
```

2.b

Листинг 2

На листинге **??** представлен стандартный цикл. Человек может вме-

сто фигурных скобок поставить do и done, неправильно инициализировать переменную (i = 1 и i in 1) или забыть скобки после for.

```
type person = {
  name: string;
  age: int;
}
```

3.a

```
type person = {
  name: string,
  age: int,
};
```

3.b

Листинг 3

На листинге **??** приведено описание типа, называемого записью. В первом случае поля отделяются точкой с запятой, во втором случае просто запятой. В коде на OCaml завершать эту конструкцию точкой с запятой нет необходимости.

3.4. Реализация дополнительных правил

4. Реализация дополнительных правил восстановления от ошибок

Дополнительные правила реализуются стандартными средствами Menhir. В Menhir существует 2 способаИнкрементальный API Menhir (§9.2) позволяет взять на себя управление при обнаружении ошибки. Ведь как только инвалидтокен обнаружен, синтаксический анализатор выдает контрольную точку вида HandlingError . На этом этапе, если кто-то решитчтобы позволить синтаксическому анализатору продолжить работу, просто вызвав возобновление, Menhir войдет в свой традиционный режим обработки ошибок (§10).Однако вместо этого можно решить взять на себя управление и выполнять обработку ошибок или исправление ошибок любым способом.радует. Например, можно создать и отобразить диагностическое сообщение на основе текущего стека автомата.и / или состояние. Или можно изменить входной поток, вставив или удалив токены, чтобы подавить ошибку,и возобновите нормальный разбор. В принципе, возможности безграничны.