parse_transform и все-все-все

Никита Калашников

15 февраля, 2014

Нелирическое вступление №1: процесс компиляции

Процесс компиляции

(не только в Erlang и в общих чертах)

1. Исходный код.
2. Обработанный код.
3. Синтаксическое дерево.
4. Байт-код.

Процесс компиляции

(не только в Erlang и в общих чертах)

1. Исходный код.
2. Обработанный код.
3. **Синтаксическое дерево.**
4. Байт-код.

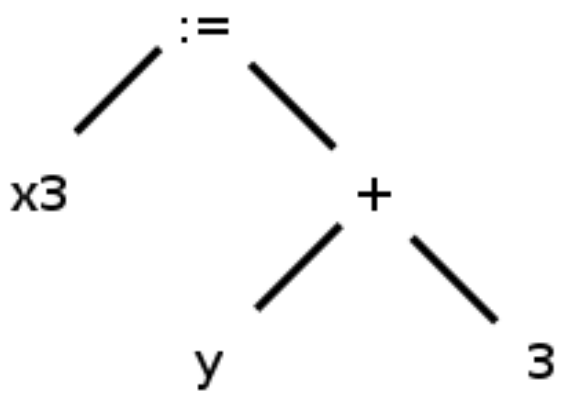
Синтаксическое дерево

Оно же дерево разбора, дерево вывода

Результат группировки токенов исходного кода программы в грамматические фразы синтаксическим анализатором.

Абстрактное синтаксическое дерево — подвид дерева разбора, отличающийся тем, что в нём отсутствуют элементы для токенов, которые не влияют на семантику программы.

```
x3 := y + 3;
```



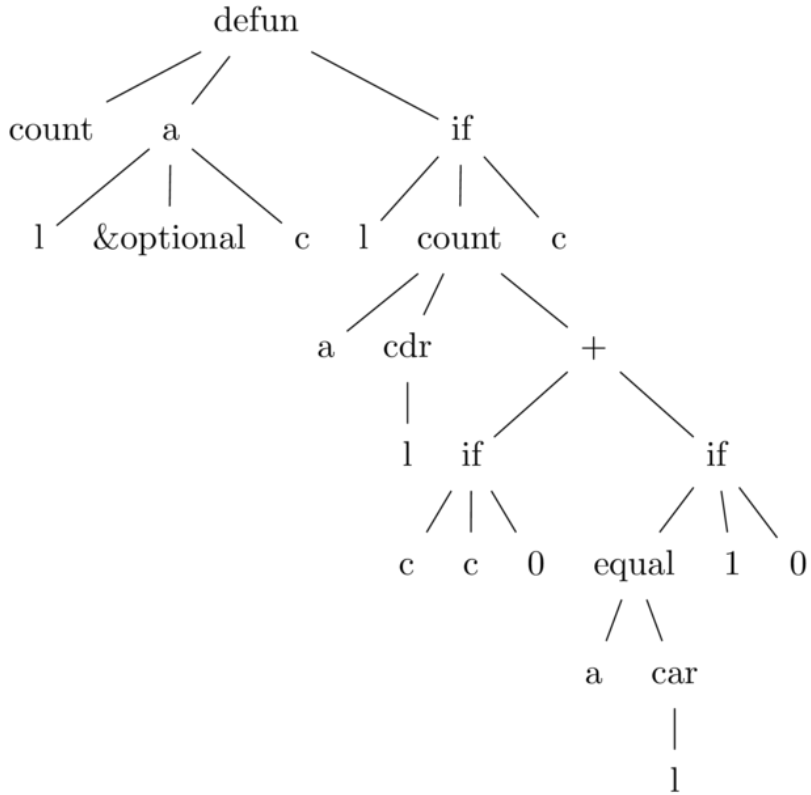
Нелирическое вступление №2: самый известный пример деревьев

Lisp и его синтаксис

What the non-Lisper sees

Lisp и его синтаксис

На самом деле синтаксиса нет.
Код на Lisp сам по себе является синтаксическим деревом.




```
(defun count (a l &optional c)
  (if l
    (count a
      (cdr l)
      (+ (if c c 0)
        (if (equal a (car l)) 1 0)))
    c))
```

К чему всё это?

Erlang Abstract Format

Местные синтаксические деревья. Представление исходного кода программы в термах Erlang.

```
-module (factorial).
-export ([fact/1]).

fact(N) when N < 0 -> error;
fact(0)            -> 1;
fact(N)            -> N * fact(N - 1).
```

```
epp:parse_file("factorial.erl", [], []).
```

Erlang Abstract Format

```
[{attribute,1,file,{"factorial.erl",1}},
 {attribute,1,module,factorial},
 {attribute,2,export,[{fact,1}]},
 {function,4,fact,1,
           [{clause,4,
                    [{var,4,'N'}],
                    [[{op,4,'<',{var,4,'N'},{integer,4,0}}]],
                    [{atom,4,error}]},
            {clause,5,[{integer,5,0}],[],[{integer,5,1}]},
            {clause,6,
                    [{var,6,'N'}],
                    [],
                    [{op,6,'*',
                         {var,6,'N'},
                         {call,6,
                               {atom,6,fact},
                               [{op,6,'-',{var,6,'N'},{integer,6,1}}]}}]}]},
 {eof,7}]
```

parse_transform

Изменение дерева Erlang Abstract Format в процессе компиляции в пределах возможностей синтаксиса языка с помощью другого Erlang-кода. То есть метапрограммирование.

parse_transform

Глазами разрабочика

Файл трансформаций — обычный модуль Erlang с единственной экспортируемой функцией `parse_transform/2`, которая принимает на вход абстрактное дерево оригинального исходного кода и возвращает изменённое.

Модуль, который нужно трансформировать, компилируется с опцией `{parse_transform, your_transform_module}`.

Примеры из жизни?

lager

github.com/basho/lager

Фреймворк для логгирования. Использует parse_transform для преобразования вызовов вроде `lager:info()` в структуру, также включающую в себя метаданные вызова (например, дату, имя модуля, номер строки и т. д.)

```
lager:info("Logged in as ~s", [Username]),
```

```
2014-01-11 13:12:44.109 [info] <0.118.0>@app_core:login:203 Logged
in as someuser
```

erlando

github.com/rabbitmq/erlando

Набор синтаксических расширений (карринг, do-нотация, импорт функций из других модулей).

Например, использование реализации монады Error:

```
write_file(Path, Data, Modes) ->
    Modes1 = [binary, write | (Modes -- [binary, write])],
    do([error_m ||
        Bin <- make_binary(Data),
        Hdl <- file:open(Path, Modes1),
        Result <- return(do([error_m ||
                             file:write(Hdl, Bin),
                             file:sync(Hdl)])),
        file:close(Hdl),
        Result]).
```

вместо дерева проверяющих case'ов на каждый вызов.

shen

github.com/5HT/shen

Трансляция Erlang в JavaScript :). Полностью строит JS-код из Erlang Abstract Format.

```
-module(fac).
-compile({parse_transform, shen}).
-compile(export_all).

-js([start/0,fac/1]).

start() ->
    N = fac(5),
    console:log("factorial ~p", [J, N]).

fac(0) -> 1;
fac(N) -> N * fac(N-1).

var pattern = require("matches").pattern;
```

```
var start = pattern({
    '': function() {
        j = 5;
        n = fac(j);
        return console.log('factorial ~p',[j,[n,[]]]);
}});
var fac = pattern({
    '0': function(x1) {
        return 1;
    },
    'n': function(n) {
        return n * fac(n - 1);
}});
start();
```

Бочка дёгтя

при использовании parse_transform

- Из-за появления неявных преобразований код становится менее очевиден.
- Дополнительные сложности в отладке (необходимость учитывать трансформации, потеря возможности диалайзинга исходного кода до трансформаций).
- Придумать свой диалект Erlang всё равно не получится.

Писать собственные трансформации стоит с предельной осторожностью! Чуть реже, чем всегда, можно обойтись без подобных глубинных вмешательств.

Спасибо!
Вопросы?

Никита Калашников

 github.com/Mendor

 twitter.com/archydragon

This slide is intentionally left blank.