Чистый рекурсивный спуск на Рефале-5

Александр Коновалов
МГТУ имени Н.Э. Баумана, Москва

V совместное рабочее совещание
ИПС имени А.К. Айламазяна РАН и МГТУ имени Н.Э. Баумана
по функциональному языку программирования Рефал,

16 июня 2022

Рекурсивный спуск

- Рекурсивный спуск — методика написания парсеров вручную на основе LL(1)-грамматики.
- Программист описывает LL(1)-грамматику языка в БНФ или в РБНФ.
- Пользуясь несложными правилами преобразования, он переписывает правила грамматики в заготовку парсера на языке программирования.
- Полученная заготовка является работоспособным кодом, который либо молча принимает на входе текст, соответствующий грамматике, либо останавливается на первой ошибке с выдачей сообщения.
- Дальнейшая работа программиста заключается в наполнении полученной заготовки семантическими действиями: построение дерева, генерация кода, проверка контекстных зависимостей.
- При желании программист также может реализовывать восстановление после ошибок.

Рекурсивный спуск

- Следует отметить, что методика написания заготовки из грамматики полностью регулярная, механизированная.
- Творческими этапами программиста являются только первый — составление LL(1)-грамматики и последний — реализация семантики и восстановления при ошибках.
- Промежуточный этап творчества не требует.
- (В принципе, можно написать даже генератор кода, принимающий грамматику, и строящий заготовку парсера. Но мы это делать не будем.)

Рекурсивный спуск

- Классическая методика рекурсивного спуска предполагает использования *императивного* языка программирования с поддержкой рекурсии (FORTRAN 77 не подойдёт 😉).
- Каждому нетерминалу грамматики ставится в соответствие процедура ЯП, осуществляющая разбор этой сущности.
- Имеется глобальный итератор, представляющий цепочку токенов, представленный глобальной переменной `Sym` и процедурой `NextToken()`, считывающей из входного потока следующий токен.

Рекурсивный спуск

- Предусловием для процедуры нетерминала, является нахождение в переменной `Sym` первого токена цепочки, выводимой из нетерминала.
- Постусловием процедуры для нетерминала является нахождение в переменной `Sym` токена, который располагается во входной цепочке сразу после строки, выводимой из нетерминала.
- Следствие: после вызова процедуры для аксиомы в переменной `Sym` должен находиться токен конца файла.

Рекурсивный спуск

Правила грамматики в БНФ имеют вид:

$$N ::= u_1 \mid \ldots \mid u_n,$$

где N — нетерминал, а u_1, …, u_n — цепочки символов (терминалов или нетерминалов).

Так как грамматика должна быть LL(1), только одна из альтернатив может раскрываться в пустоту:

$$u_i \Rightarrow^* \varepsilon \text{ или } \varepsilon \in \mathrm{FIRST}(u_i)$$

Потребуем, чтобы это была последняя альтернатива:

$$\varepsilon \in \mathrm{FIRST}(u_n).$$

Рекурсивный спуск

Процедура для разбора правила

$$N ::= u_1 \mid \ldots \mid u_n$$

будет иметь вид ($\varepsilon \notin FIRST(u_n)$):

```
procedure N;
begin
  case Sym of
    ‹FIRST(u1)›: begin ‹PARSE(u1)› end;
    …
    ‹FIRST(un)›: begin ‹PARSE(un)› end;
  else
    ‹сообщить о синтаксической ошибке›
  end
end;
```

Рекурсивный спуск

Процедура для разбора правила

$$N ::= u_1 \mid \ldots \mid u_n$$

будет иметь вид ($\varepsilon \in FIRST(u_n)$):

```
procedure N;
begin
  case Sym of
    ‹FIRST(u1)›: begin ‹PARSE(u1)› end;
    …
    ‹FIRST(un-1)›: begin ‹PARSE(un-1)› end;
  else
    begin ‹PARSE(un)› end;
  end
end;
```

Рекурсивный спуск

Очевидная оптимизация: процедура для разбора правила

```
N ::= u
```

будет иметь вид:

```
procedure N;
begin
   ‹ PARSE(u) ›
end;
```

Рекурсивный спуск

- Функция ‹ `FIRST(u)` › на предыдущих слайдах, очевидно, раскрывается в список тегов токенов, с которых может начинаться нетерминал (при этом `ε` игнорируется).
- Функция ‹ `PARSE(u)` › порождает код.

Рекурсивный спуск

Обозначим X — терминальный символ, N — нетерминальный. Тогда

```
‹PARSE(X u)› ⇒
  if Sym = X then
    NextToken()    { обновляет Sym }
  else
    ‹сообщить о синтаксической ошибке›;
  ‹PARSE(u)›

‹PARSE(N u)› ⇒ N(); ‹PARSE(u)›

‹PARSE(ε)› ⇒       { ничего не делаем }
```

Рекурсивный спуск

Пара замечаний:

- При использовании РБНФ, содержащей вложенные альтернативы, опцию ($u^?$) и итерацию (u^*, u^+), функция ‹ `PARSE(u)` › усложнится, будет порождать условные конструкции и циклы.
- При добавлении семантических действий процедуры нетерминалов получают дополнительные параметры и возвращаемые значения.

Рекурсивный спуск императивен

- Как видно, методика подразумевает наличие *изменяемого состояния*: переменной `Sym`, которая видна всем процедурам нетерминалов.
- При использовании функциональных языков и стиля программирования без побочных эффектов писать парсеры рекурсивным спуском становится сложнее.

Рекурсивный спуск императивен

- У чистых функций нет доступа к изменяемому глобальному состоянию.
- Исходные данные они должны получать через аргументы, все результаты вычислений должны становиться возвращаемым значением.
- Поэтому вместо переменной `Sym` и процедуры `NextToken` приходится заранее готовить список токенов, передавать его в каждую функцию как параметр и остаток этого списка возвращать как результат.

Рекурсивный спуск императивен

- Если функция (после добавления семантических действий) возвращает полезное значение (например, дерево), то придётся возвращать кортеж.
- На Haskell глобальное мутабельное состояние можно «спрятать» в монаду `State`.
- Но на Рефале, языке первого порядка, это не получится.

Рекурсивный спуск на Рефале

- Конечно, в имеющихся реализациях Рефала есть императивные средства, обеспечивающие изменяемое состояние.
- В Рефале-5 есть функции закапывания-выкапывания (Br, Dg, Cp, Rp).
- В других реализациях Рефала есть «ящики»: статические (глобальные переменные) и динамические (изменяемые объекты в куче).
- Но нам интересно рассмотреть методику написания именно чистых парсеров:
  - чистый код можно преобразовывать имеющимися суперкомпиляторами (SCP4, MSCP-A),
  - были попытки написать параллельную реализацию Рефала-05, в которой параллелиться могут только чистые функции.

Рекурсивный спуск на Рефале

- Очевидно, от списка токенов, передаваемого в функции и возвращаемого из них, нам никуда не деться. Нужно лишь придумать подход, в котором это будет относительно удобно.
- Также нам хочется, чтобы методика была регулярной — перевод грамматики в код был механизирован и не требовал творческой деятельности.
- Методика должна допускать добавление семантики и восстановления при ошибках.

Рекурсивный спуск на Рефале

- Будем обозначать правило, прочитанное неполностью как

```
N ::= u . v
```

- где точка отделяет прочитанную часть от непрочитанной.
- Также будем считать, что множества имён терминалов и нетерминалов не пересекаются. В примере имена терминалов будут литерами (characters), имена нетерминалов — идентификаторами (compound symbol).
- Будем обозначать символами `N` и `X`, соответственно, терминальный и нетерминальный символы.

Рекурсивный спуск на Рефале

Функции, записываемые для нетерминалов, будут иметь следующий формат:

```
<N (e.Symbols) s.NTerm? e.Tokens>
  == s.NTerm e.Tokens | ERR e.Message
e.Symbol ::= { s.Term | s.NTerm }*
e.Tokens = s.Term*
s.Term ::= CHAR
s.NTerm ::= WORD
```

Здесь `e.Symbols` — символы грамматики, `e.Tokens` — входные символы, `s.Term` и `s.NTerm` — терминальный и нетерминальный символы соответственно, `e.Message` — сообщение о синтаксической ошибке.

В «кармане» хранится распознанная часть правила.

Рекурсивный спуск на Рефале

Для правила вида

$$N ::= u_1 \mid \ldots \mid u_n,\ \varepsilon \notin FIRST(u_n)$$

строится функция вида

```
N {
   (e.Scanned) ERR e.Msg = ERR e.Msg;

   ‹PARSE(N ::= . u1)›
   …
   ‹PARSE(N ::= . un)›

   (e.Scanned) s.Unexpected e.Tokens
     = ERR 'Unexpected "' s.Unexpected '" in N';
}
```

Рекурсивный спуск на Рефале

Для правила вида

$$N ::= u_1 \mid \ldots \mid u_n,\ \varepsilon \in FIRST(u_n)$$

строится функция вида

```
N {
   (e.Scanned) ERR e.Msg = ERR e.Msg;

   ‹PARSE(N ::= . u1)›
   …
   ‹PARSE(N ::= . un)›

   () e.Tokens = N e.Tokens;
}
```

Рекурсивный спуск на Рефале

Функция ‹ PARSE(N ::= u . v) › определяется следующим образом:

```
/* особый случай для первого нетерминала */
‹ PARSE(N ::= . N' v) › ⇒
  () ‹N'› e.Tokens = <N (‹N'›) e.Tokens>;

  () s.Sym e.Token
    , <OneOf s.Sym ‹FIRST(N')›> : True
    = <N () <N' () s.Sym e.Tokens>>;

  ‹ PARSE(N ::= N' . v) ›
```

Рекурсивный спуск на Рефале

Функция ‹ PARSE(N ::= u . v) › определяется следующим образом:

```
‹PARSE(N ::= u . X v) ⇒
   (‹u›) ‹X› e.Tokens = (‹u› ‹X›) e.Tokens;

  ‹PARSE(N ::= u X . v)›
```

```
‹PARSE(N ::= u . N' v) ⇒
   (‹u›) ‹N'› e.Tokens = <N (‹u› ‹N'›) e.Tokens>;

   (‹u›) e.Tokens = <N (‹u›) <N' () e.Tokens>>;

  ‹PARSE(N ::= u N' . v)›
```

Рекурсивный спуск на Рефале

Функция ‹ PARSE(N ::= u . v) › определяется следующим образом:

```
‹ PARSE(N ::= u .) ⇒
   (‹ u ›) e.Tokens = N e.Tokens;
```

Рекурсивный спуск на Рефале

Главная программа (здесь `S` — аксиома):

```
$ENTRY Go {
  /* пусто */
    , <S () <GetTokens>>
    : {
        S "eof" = <Prout 'Ok'>;

        S s.Unexpected e.TokensRest
          = <Prout
              'Unexpected "' s.Unexpected '", '
              'expected EOF'
            >;

        ERR e.Message = <Prout e.Message>;
      }
}
```

Рекурсивный спуск на Рефале

Если мы хотим организовать восстановление при ошибках, формат функции нетерминала меняется на следующий:

```
<N (e.Symbols) s.NTerm? e.Tokens t.ErrorList>
  == s.NTerm e.Tokens t.ErrorList
e.Symbol ::= { s.Term | s.NTerm }*
e.Tokens = s.Term*
s.Term ::= CHAR
s.NTerm ::= WORD
```

Здесь `t.ErrorList` — список ошибок.

Рекурсивный спуск на Рефале

Если мы хотим строить дерево и хранить дополнительную информацию в токенах (значение и позицию), формат усложняется до:

```
<N (e.Symbols) t.NTerm? e.Tokens t.ErrorList>
  == t.NTerm e.Tokens t.ErrorList
e.Symbol ::= { t.Token | t.NTerm }*
e.Tokens = t.Token*
t.Token ::= (s.Tag t.Pos e.Info)
t.NTerm ::= (s.Tag e.Info)
```

Значение `e.Info` зависит от типа токена или нетерминала.

Рекурсивный спуск на Рефале

Если мы хотим передавать семантическую информацию, то она, как и список ошибок, передаётся в конце:

```
<N
   (e.Symbols) t.NTerm? e.Tokens
   t.SymTable t.ErrorList
>
   == t.NTerm e.Tokens t.SymTable t.ErrorList

e.Symbol ::= { s.Token | s.NTerm }*
e.Tokens = t.Token*
t.Token ::= (s.Tag t.Pos e.Info)
t.NTerm ::= (s.Tag e.Info)
```

Рекурсивный спуск на Рефале (пример)

Рассмотрим грамматику арифметических выражений:

```
E   ::= T E1
E1  ::= + T E1 | - T E1 | ε
T   ::= F T1
T1  ::= * F T1 | / F T1 | ε
F   ::= n | ( E )
```

Полный текст примера доступен по ссылке:
https://github.com/Mazdaywik/rec-desc-refal

Неоптимальность и специализация

- Очевидно, что код вида

```
 /* F ::= ( E ) */
() '(' e.Tokens = <F ('(') e.Tokens>;

('(') E e.Tokens = <F ('(' E) e.Tokens>;
('(') e.Tokens = <F ('(') <E () e.Tokens>>;

('(' E) ')' e.Tokens = <F ('(' E ')') e.Tokens>;
('(' E ')') e.Tokens = F e.Tokens;
```

  оптимальным не назовёшь.

- Однако, очевидно и то, что функции можно специализировать по содержимому кармана.
- Проблемой здесь может быть то, что карман на каждом следующем вызове растёт и он может быть обобщён из-за срабатывания отношения Хигмана-Крускала.
- Необходимы методы специализации, которые здесь к проблемам не приводят.

Выводы

- Сформулирована методика, позволяющая писать парсеры для Рефала методом рекурсивного спуска.
- Код пишется регулярно, правила грамматики просто переводятся в код на Рефале.
- Парсер, полученный в итоге, оказывается функционально чистым.
- Его можно специализировать соответствующими инструментами.