

Выполняется обход грамматики, согласованный со входной строкой — есть два указателя: в грамматику и позицию во входе. Указатель в грамматику — слот:  $X \rightarrow x_0..x_k \cdot x_{k+1}..x_n$

Возможны несколько ситуаций. Предположим, что слот выглядит одним из следующих способов:  $X \rightarrow \alpha \cdot x\beta$  или  $X \rightarrow \alpha \cdot$ . Указатель во входе находится на позиции  $i$ .

1.  $x = \omega[i + 1]$ . Указатель в грамматике перемещается в позицию  $X \rightarrow \alpha x \cdot \beta$  и указатель во входе сдвигается в позицию  $i + 1$ .
2.  $x$  — это нетерминал  $A$ . Запоминаем точку возврата: кладём на стек слот  $X \rightarrow \alpha x \cdot \beta$ . Сдвигаем указатель в грамматике в позицию  $A \rightarrow \cdot \gamma$ . Здесь можно воспользоваться FIRST и FOLLOW для детерминизации выбора. Позиция во входе неизменна.
3. Указатель в грамматике в позиции  $X \rightarrow \alpha \cdot$  и стек непуст. На вершине стека должен быть "адрес возврата" вида  $Y \rightarrow \delta X \cdot \mu$ . Он извлекается из стека и становится текущим указателем в грамматике.
4. Указатель в грамматике в позиции  $S \rightarrow \alpha \cdot$  ( $S$  — стартовый нетерминал) и стек пуст. Успешное завершение разбора. Иначе провал.

На шаге 2 можно не продолжать сразу все возможные варианты, а создать дескриптор — сущность, позволяющую возобновить обход с места, которое он описывает. Для каждого полученного слота вида  $A \rightarrow \cdot \gamma$  создаём дескриптор, содержащий его, указатель на вход в позицию  $i$  и указатель на стек, на адрес возврата, который только что был добавлен ( $X \rightarrow \alpha A \cdot \beta$ ). Для организации стека можно использовать GSS, а дескриптор будет указывать на вершину в нём.

Возможны проблемы с бесконечным количеством дескрипторов и "потерей" точек возврата.

- $R$  — мн-во дескрипторов для обработки
- $U$  — мн-во созданных дескрипторов
- $P$  — мн-во тех, для кого надо не забыть сделать *pop*

```

let _add L v i =
  if (L, v, i) not in U
  then
    add (L, v, i) to U
    add (L, v, i) to R

let _pop v i =
  if v <> v_0
  then
    add (v,i) to P
    for each u in v.child do _add v.L u i

let _create L v i =
  if (L,i) not in GSS then add (L,i) to GSS
  let u = GSS.get (L,i)
  if there is not an edge from u to v
  then
    add edge from u to v
    for all (u,k) in P do _add L v k
  u

```

Нам нужен табличный вариант.

```

let rec dispatcher () =
  if R.Count <> 0
  then
    (L,v,i) := R.Get()
    dispatch := false
  else
    stop := true

and processing () =
  dispatch := true
  match L with
  | (X -> a . x b where x = input[i + 1]) ->
    i := i + 1
    L := (X -> a x . b)
    dispatch := false

```

```

| (X -> a . x b where x is nonterminal) ->
  v := _create (X -> a x . b) v i
  let slots = pTable[x][input[i]]
  for L in slots do
    _add L v i
| (X -> a .) -> _pop v i
| _ -> final result processing and error notification

let control () =
  while not !stop do
    if !dispatch then dispatcher() else processing()

control()

```

Если граф детерминированный, то кроме использования в качестве позиции вершины не требуется иных модификаций, так как в первом случае ровно один вариант совпадения входного символа с ожидаемым в правиле.

Лес разбора ( $L$  — слот,  $i, j, k$  — координаты).

- Терминальные узлы  $(T, i, j)$
- Нетерминальные узлы  $(N, i, j)$ . Сыновья — запакованные узлы вида  $N \rightarrow \gamma \cdot, k$
- Промежуточные узлы  $(L, i, j)$ . Сыновья — запакованные узлы с меткой  $L, k$ , где  $i \leq k \leq j$
- Запакованные узлы  $(L, k)$ . Один или два сына. Правый — терминал или нетерминал с меткой  $(S, k, j)$ . Левый (если есть) — терминал, нетерминал или промежуточный с меткой  $(S, j, k)$

Теперь в дескрипторе надо запоминать ссылку на узел дерева:  $(L, v, i, a)$  Дескрипторы с непустой ссылкой создаются в момент вызова *пор*. Правый сын создаваемого узла — только что построенный узел. Левый сын хранится на ребре, исходящем из вершины, которую только что достали со стека. Метка нового узла — адрес возврата, полученный из достанного узла.

Теперь функции должны работать ещё и с узлами леса.

```
let _add L v i a =
  if (L, v, i, a) not in U
  then
    add (L, v, i, a) to U
    add (L, v, i, a) to R

let _pop v i z =
  if v <> v_0
  then
    add (v,z) to P
    for each edge (a, u) in v.outEdges do
      let y = getNodeP v.L a z
      _add v.L u i y

let _create L v i a =
  if (L,i) not in GSS then add (L,i) to GSS
  let u = GSS.get (L,i)
  if there is not an edge from u to v labelled a
  then
    add edge from u to v labelled a
    for all (u,z) in P do
      let y = getNodeP L a z
      let (_,_,k) = z
      _add L v k y
  u

let getNodeT x i =
  let h = if x = \eps then i else i + 1
  if there is no SPPF node with label (x,i,h)
  then create node with label (x,i,h)
  return node with label (x,i,h)

let getNodeP (X -> w1 . w2) a z =
  if w1 is terminal or non-nullable nonterminal and w2 <> \eps
  then return z
  else
```

```

    let t = if w2 = \eps then X else (X -> w1 . w2)
    let (q,k,i) = z.lbl
    if a <> dummy
    then
        let (s,j,k) = a.lbl
        let y = find_or_create SPPF.nodes (n.lbl = (t,i,j))
        if y does not have a child with label (X -> w1 . w2)
        then
            let y' = new_packed_node(a,z)
            y.chld.add y'
        return y
    else
        let y = find_or_create SPPF.nodes (n.lbl = (t,k,i))
        if y does not have a child with label (X -> w1 . w2)
        then
            let y' = new_packed_node(z)
            y.chld.add y'
        return y

let rec dispatcher () =
    if R.Count <> 0
    then
        (L,v,i,cN) := R.Get()
        cR := dummy
        dispatch := false
    else
        stop := true

and processing () =
    dispatch := true
    match L with
    | (X -> a . x b where x = input[i + 1]) ->
        if cN = dummyAST
        then cN := getNodeT i
        else cR := getNodeT i
        i := i + 1
        L := (X -> a x . b)

```

```

        if !cR <> dummy
        then cN := getNodeP L cN cR
        dispatch := false
    | (X -> a . x b where x is nonterminal) ->
        v := _create (X -> a x . b) v i cN
        let slots = pTable[x][input[i]]
        for L in slots do
            _add L v i dummy
    | (X -> a .) ->
        _pop v i cN
    | _ -> final result processing and error notification

let control () =
    while not !stop do
        if !dispatch then dispatcher() else processing()

control()

```