Санкт-Петербургский государственный университет

Программная инженерия

Горохов Артем Владимирович

Поддержка расширенных контекстно-свободных грамматик в алгоритме синтаксического анализа Generalized LL

Выпуская квалификационная работа

Научный руководитель:
к. ф. -м. н., доц. Григорьев С. В.

Рецензент:
СУИ НИУ ИТМО, программист Авдюхин Д.А.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Software engineering

Artem Gorokhov

Support of extended context-free grammars in Generalized LL parsing algorithm

Graduation Thesis

Scientific supervisor:
associate professor Semyon Grigorev

Reviewer:
ITMO University, programmer Dmitry Avduhin

Saint-Petersburg
2017

Оглавление

Введение

Общеупотребимый способ описания синтаксиса языков программирования — расширенные контекстно-свободные грамматики. Например спецификации языков C, $C++$, $Java$ и т.д. С одной стороны, эта форма проста для понимания людей, а с другой, достаточно формальна и допускает автоматизированное создание синтаксических анализаторов.

Существуют различные инструменты для создания синтаксических анализаторов, которые по входной грамматике позволяют создать инструментальное средство для синтаксической обработки текстов, созданных на этом языке. Проблема заключается в том, что эти инструменты сначала преобразуют грамматику к контекстно-свободной форме, и только по ней строят синтаксический анализатор.

Есть работы, которые описывают синтаксический анализ с помощью расширенных контекстно-свободных грамматик(extended context-free grammar (ECFG)). Но нет инструментов, основанных на данных работах. Кроме того, подходы, описанные в данных исследованиях, поддерживают лишь подклассы контекстно-свободных языков.

Алгоритмы обобщённого синтаксического анализа, например Generalized LL [**?**], способны использовать контекстно-свободные грамматики описывающие произвольные контекстно-свободные языки. Но они так же не работают с грамматиками в форме EBNF без предварительного преобразования к контекстно-свободной форме.

В биоинформатике стоит задача поиска генов и иных последовательностей в геномах. Эти последовательности имеют некоторые общие свойства, которые можно описать контекстно-свободной грамматикой. Есть инструменты типа infernal [**?**], которые используют синтаксический анализ для поиска структур в геномах. Но не всегда генетический материал образует обычную последовательность. Метагеномные сборки - результат считывания генов нескольких организмов, представленный в виде графа, пути в котором задают гены этих организмов. И нет инструментов способных работать с метагеномными сборками. Было бы здорово применить полученный GLL алгоритм для синтаксического анализа метагеномных сборок.

На нашей кафедре, в рамках исследовательского проекта YaccConstructor [**?**], разрабатывается подход поиска структур заданных с помощью контекстно-свободной грамматики в метагеномных сборках, основанный на алгоритме Generalized LL. Предполагается, что синтаксический анализ по ECFG без преобразований даст ощутимый прирост производительности существующего подхода.

1. Постановка задачи

Целью данной работы является разработка модификации алгоритма GLL работающей с грамматиками в расширенной форме Бэкуса-Наура и проверка того, как полученный алгоритм влияет на производительность поиска структур заданных с помощью контекстно-свободной грамматики в метагеномных сборках. Для её достижения были поставлены следующие задачи.

- Выбрать или разработать подходящее представление ECFG.
- Спроектировать структуру данных для представления леса разбора по ECFG.
- Разработать алгоритм на основе Generalized LL, строящий лес разбора по ECFG.
- Реализовать алгоритм в рамках проекта YaccConstructor.
- Провести эксперименты и сравнение.

2. Обзор

2.1. Расширенные контекстно-свободные грамматики

Статический анализ программ обычно выполняется над структурным представлением кода. И парсинг - это классический способ получить такое представление. Генераторы парсеров часто используются для автоматизации создания парсера: эти инструменты получают парсер по грамматике.

Расширенная форма Бэкуса-Наура [?] является метасинтаксисом для представления контекстно-свободных грамматик. В дополнение к конструкциям используемым в форме Бэкуса-Наура, в ней используется следующие конструкции: альтернатива |, необязательные символы [...], повторение ... и группировка (...).

Эта форма широко используется для спецификации грамматики в технической документации ввиду того, что выразительная сила ECFG делает спецификацию синтаксиса более компактной и удобочитаемой. Поскольку документация является одним из основных источников информации о языке для разработчиков синтаксических анализаторов, было бы полезно иметь генератор анализаторов, который поддерживает грамматики в EBNF. Заметим, что EBNF является лишь стандартизированной формой для *расширенных контекстно-свободных грамматик* [?], которые могут быть определены следующим образом:

***Определение* 1.** *Расширенная контекстно-свободная грамматика (ECFG) [?] - это кортеж* (N, Σ, P, S)*, где* N *и* Σ *конечные множества нетерминалов и терминалов соответственно,* $S \in N$ *является стартовым символом, а* P *(продукция) является отображением из* N *в регулярное выражение над алфавитом* $N \cup \Sigma$*.*

ECFG широко используется в качестве входного формата для генераторов парсеров, но классические алгоритмы синтаксического анализа часто требуют CFG, и, как результат, генераторы анализаторов требуют преобразования в CFG. Возможно преобразование ECFG в CFG [?], но это преобразование приводит к увеличению размера грамматики и изменению её структуры: при трансформации добавляются новые нетерминалы. В результате синтаксический анализатор строит дерево вывода относительно преобразованной грамматики, и разработчику языка сложнее отлаживать грамматику и использовать результат синтаксического анализа.

Существует широкий спектр методов анализа и алгоритмов [?, ?, ?, ?, ?, ?, ?, ?], которые способны обрабатывать грамматику в ECFG. Детальный обзор результатов и задач в области обработки ECFG представлены в статье “Towards a Taxonomy for ECFG and RRPG Parsing” [?]. Заметим только, что большинство алгоритмов основаны на классических методах LL [?, ?, ?] и LR [?, ?, ?], но они работают только с ограниченными подклассами ECFG. Таким образом, нет решения для обработки произвольных (в том числе неоднозначных) ECFG.

Алгоритмы синтаксического анализа на основе LL более интуитивны, чем основанные на LR, и могут обеспечить лучшую диагностику ошибок. В настоящее время LL(1) представляется наиболее практичным алгоритмом. К сожалению, некоторые языки не являются LL(k) (для любого k), и леворекурсивные грамматики являются проблемой для инструментов на основе LL. Другим ограничением для LL анализаторов являются неоднозначности в грамматике, которые, вместе с предыдущими недостатками, усложняют создание синтаксических анализаторов. Алгоритм Generalized LL, предложенный в [**?**], решает все эти проблемы: он обрабатывает произвольные CFG, в том числе неоднозначные и леворекурсивные. В худшем случае временная и пространственная сложность GLL зависит кубически от размера входа. А для LL(1) грамматик, он демонстрирует линейную временную и пространственную сложность.

2.2. Структурированный в виде графа стек

todo

2.3. Сжатое представление леса разбора

todo

2.4. Алгоритм Generalized LL

Цель обобщенных алгоритмов синтаксического анализа - обеспечить создание синтаксических анализаторов по произвольным контекстно-свободным грамматикам. Алгоритм Generalized LL (GLL) [**?**] включает в себя свойства классических LL алгоритмов: он более интуитивен и обеспечивает более хорошую диагностику ошибок, чем обобщенные LR алгоритмы. Кроме того, опыт показывает, что решения на основе GLR более сложны, чем основанные на GLL, что согласуется с наблюдением в [11], что синтаксические анализаторы ECFG на основе LR очень сложны. Таким образом, в качестве основы для решения был выбран GLL алгоритм.

Идея алгоритма GLL основана на обработке так называемых дескрипторов, которые могут однозначно определить состояние процесса синтаксического анализа. Дескриптор представляет собой четырехэлементный кортеж (L, i, T, S), где:

- L указатель на позицию в грамматике вида $(S \to \alpha \cdot \beta)$;
- i — позиция во входе;
- T — корень построенного леса разбора;
- S — текущий узел стека (GSS) [**?**].

GLL двигается одновременно по входу и грамматике, создавая множество дескрипторов в случае неоднозначности и использует очередь для управления обработкой дескрипторов. В начальном состоянии есть только один дескриптор, который состоит из начальной позиции в грамматике ($S \to \cdot\beta$), во входе (i = 0), фиктивного узла дерева (\$) и дня стека. На каждом шаге алгоритм извлекает дескриптор из очереди и действует в зависимости от грамматики и входа. Если есть неоднозначность, то алгоритм помещает в очередь дескрипторы для всех возможных случаев, чтобы обработать их позже. Для достижения кубической временной сложности важно помещать в очередь только дескрипторы, которые не создавались ранее. Для того чтобы решить добавлять дескриптор или нет используется глобальное хранилище всех созданных дескрипторов. Существует подход на основе таблиц [**?**] для реализации GLL, который генерирует только таблицы для данной грамматики вместо полного кода синтаксического анализатора. Эта идея похожа на алгоритм в оригинальной статье и использует те же техники построения леса разбора и обработки стека. Псевдокод, иллюстрирующий этот подход, можно найти в приложении 7. Обратите внимание, что в эту работу не включена проверка для множеств first/follow.

2.5. Проект YaccConstructor

2.6. Анализ метагеномных сборок

Метагеномная сборка это граф Анастасия Рагозина в своей магистерской предложила подход к анализу метагеномных сборок с помощью алгоритма GLL. Позже этот подход был модернизирован в рамках проекта.

3. Представление ECFG

Чтобы облегчить задание грамматики в форме ECFG для синтаксического анализатора будем использовать рекурсивный автомат (Recursive Automaton (RA) [**?**] для представления ECFG. Будем использовать следующее определение RA.

***Определение* 2.** *Рекурсивный автомат R это кортеж $(\Sigma, Q, S, F, \delta)$, где Σ — конечное множество терминалов, Q - конечное множество состояний, $S \in Q$ — начальное состояние, $F \subseteq Q$ — множество конечных состояний, $\delta : Q \times (\Sigma \cup Q) \to Q$ — функция перехода.*

В рамках этой работы единственное различие между рекурсивным автоматом и общеизвестным конечным автоматом (FSA) состоит в том, что переходы в RA обозначаются либо терминалом (Σ), либо состоянием автомата (Q). Далее в этой работе будем называть переходы по элементам из Q *нетерминальными-переходами*, а по терминалам — *терминальными переходами*.

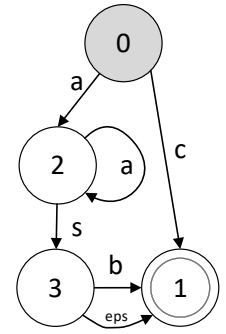
Заметим, что позиции грамматики эквивалентны состояниям автомата, которые строятся из правых частей продукций. Правые части продукций ECFG являются регулярными выражениями над объединенным алфавитом терминалов и нетерминалов. Итак, наша цель — построить RA с минимальным числом состояний для заданной ECFG, что можно сделать следующими шагами.

- Построить конечный автомат, используя метод Томпсона для каждой правой части продукций. [**?**].
- Создать карту из каждого нетерминала в соответствующее начальное состояние автомата. Эта карта должна оставаться консистентной на протяжение всех следующих шагов.
- Преобразовать автоматы из предыдущего шага в детерминированные без ε-переходов используя алгоритм, описанный в [**?**].
- Минимизировать детерминированные автомат, используя, например, алгоритм Джона Хопкрофта [**?**].
- Заменить нетерминальные переходы переходами по, стартовым состояниям автоматов, соответствующим данным нетерминалам, используя карту M. Результат этого шага — искомый рекурсивный автомат. Также используем карту M для определения функции $\Delta : Q \to N$ где N — имя нетерминала.

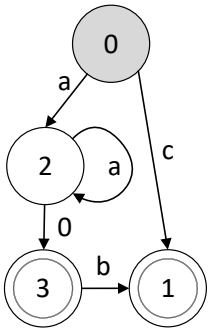
Пример преобразования ECFG в RA представлен на рис. 1, где состояние 0 — начальное состояние результирующего RA.

$S ::= a^+ S\ b? \mid c$

(a) Грамматика G_1



(b) Конечный автомат для G_1



(c) Рекурсивный автомат R_1 для G_1

Рис. 1: Преобразование грамматики в рекурсивный автомат

4. Лес разбора по ECFG

Результатом процесса синтаксического анализа является структурное представление ввода — дерево вывода или лес разбора в случае нескольких выводов. Для начала, определим дерево вывода для рекурсивного автомата: это дерево, корень которого помечен начальным состоянием, листовые узлы помечены терминалом или ε, а внутренние узлы помечены нетерминалами N и их дети образуют последовательность меток в пути в автомате, который начинается в состоянии q_i, где $\Delta(q_i) = N$.

***Определение* 3.** *Дерево вывода последовательности α для рекурсивного автомата $R = (\Sigma, Q, S, F, \delta)$ это дерево со следующими свойствами:*

- *Корень помечен $\Delta(S)$;*
- *Листья —теминалы $a \in (\Sigma \cup \varepsilon)$;*
- *Остальные узлы — нетерминалы $A \in \Delta(Q)$;*
- *У узла с меткой $N_i = \Delta(q_i)$ есть:*
  - *дети $l_0 \dots l_n (l_i \in \Sigma \cup \Delta(Q))$ тогда и только тогда, когда существует путь p в R, $p = q_i \xrightarrow{l_0} q_{i+1} \xrightarrow{l_1} \dots \xrightarrow{l_n} q_m$, где $q_m \in F$, $l_i = \begin{cases} k_i, & \text{if } k_i \in \Sigma, \\ \Delta(k_i), & \text{if } k_i \in Q, \end{cases}$*
  - *только один ребенок помеченный ε тогда и только тогда, когда $q_i \in F$*

Для произвольных грамматик RA может быть неоднозначным с точки зрения допустимых путей, и, как результат, можно получить несколько деревьев разбора для одной входной строки. Shared Packed Parse Forest (SPPF) [**?**] может использоваться как компактное представление всех возможных деревьев разбора. Будем использовать бинаризованную версию SPPF, предложенную в [**?**], для уменьшения потребления памяти и достижения кубической наихудшей временной и пространственной сложности.

Бинаризованный SPPF может использоваться в GLL [**?**] и содержит следующие типы узлов (здесь i и j назовём правым и левым extent, если i и j - начало и конец выведенной подстроки во входной строке):

- Упакованные узлы вида (S, k), где S состояние автомата, k — начало выведенной подстроки правого ребёнка. У упакованных узлов обязательно есть правый ребёнок — символьный узел, и опциональный левый — символьный или промежуточный узел.
- Символьный узел помечен (X, i, j) где $X \in \Sigma \cup \Delta(Q) \cup \{\varepsilon\}$. Терминальные символьные узлы ($X \in \Sigma \cup \{\varepsilon\}$) — листья. Нетерминвльные символьные узлы ($X \in \Delta(Q)$) могут иметь несколько упаковынных детей.
- Промежуточные узлы помечены (S, i, j), где S состояние в автомате, могут иметь несколько упаковынных детей.

Опишем модификации исходных функций построения SPPF. Функция **getNodeT**(x, i), которая создает терминальные узлы, повторно используется без каких-либо модификаций из базового алгоритма. Чтобы обрабатывать недетерминизм в состояниях, определим функцию **getNodes**, которая проверяет, является ли следующее состояние RA финальным и в этом случае строит нетерминальный узел в дополнение к промежуточному. Она использует изменённую функцию **getNodeP**: вместо позиции в грамматики он принимает в качестве входных данных отдельно состояние RA и символ для нового узла SPPF: текущий нетерминал или следующее состояние RA.

function GETNODES(S, A, w, z)
  if (S is final state) **then**
    $x \leftarrow$ **getNodeP**(S, A, w, z)
  else $x \leftarrow \$$
  if ($w = \$$)& not (z is nonterminal node and its extents are equal) **then**
    $y \leftarrow z$
  else $y \leftarrow$ **getNodeP**(S, S, w, z)
  return (y, x)

function GETNODEP(S, L, w, z)
  $(\_, k, i) \leftarrow z$
  if ($w \neq \$$) **then**
    $(\_, j, k) \leftarrow w$
    $y \leftarrow$ find or create SPPF node labelled (L, j, i)
    if ($\nexists$ child of y labelled (S, k)) **then**
      $y\prime \leftarrow$ **new** $packedNode(S, k)$
      $y\prime.addLeftChild(w)$

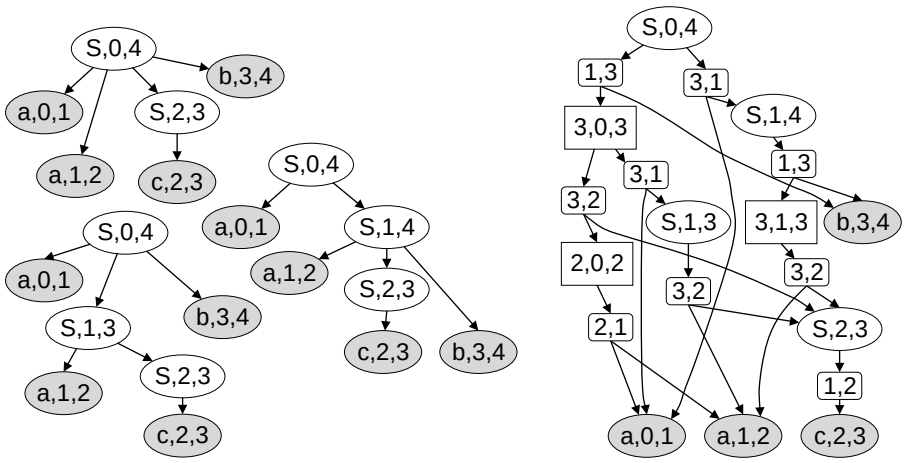
$y\prime.addRightChild(z)$
$y.addChild(y\prime)$

else

$y \leftarrow$ find or create SPPF node labelled (L, k, i)
if ($\nexists$ child of y labelled (S, k)) **then**
$y\prime \leftarrow$ **new** $packedNode(S, k)$
$y\prime.addRightChild(z)$
$y.addChild(y\prime)$

return y

Рассмотрим пример SPPF для ECFG G_1 (рис. 1a). Эта грамматика содержит конструкции (условное вхождение(?) и повторение(+)), которые должны быть преобразованы с использованием дополнительных нетерминалов для создания обычного GLL-парсера. Предложенный генератор строит рекурсивный автомат R_1 (рис. 1c) и парсер для него. Возможные деревья ввода последовательности $aacb$ показаны на рис. 2a. SPPF, созданный парсером (рис. 2b), содержит в себе все три дерева.



(a) Возможные деревья вывода (b) SPPF

Рис. 2: Пример для входа $aacb$ и автомата R_1

5. Алгоритм построения леса разбора по ECFG

В этом разделе мы описываем изменения в управляющих функциях базового алгоритма Generalised LL, необходимые для обработки ECFG. Основной цикл аналогичен базовому GLL: на каждом шаге основная функция **parse** извлекает из очереди дескриптор R, подлежащий обработке. Пусть текущий дескриптор – кортеж (C_S, C_U, i, C_N), где C_S — состояние RA, C_U — узел GSS, i — позицию во входной строке ω, C_N — узел SPPF. В ходе обработки дескриптора могут возникнуть следующие не исключающие друг друга ситуации.

- C_S — **финальное состояние.** Это возможно только если C_S — стартовое состоение текущего нетерминала. Следует построить нетерминальный узел с ребёнком (ε, i, i) и вызвать функцию **pop**, так как разбор нетерминала окончен.
- **Существует терминальный переход** $C_S \xrightarrow{\omega.[i]} q$**.** Во-первых, построить терминальный узел $t = (\omega.[i], i, i+1)$, далее вызвать функцию **getNodes** чтобы построить родителя для C_N и t. Функция **getNodes** возвращает кортеж (y, N), где N — опциональный нетерминальный узел. Создать дескриптор $(q, C_U, i+1, y)$ и поместить в очередь вне зависимости от того был ли он создан до этого. Если $N \neq \$$, вызвать функцию **pop** для этого узла, состояния q и позиции во входе $i+1$.
- **Есть нетерминальные переходы из** C_S**.** Это значит что следует начать разбор нового нетерминала, поэтому должен быть создан новый узел GSS, если такового ещё нет. Для этого нужно вызвать функцию **create** для каждого такого перехода. Она осуществляет необходимые операции с GSS и проверяет наличие узла GSS для текущих нетерминала и позиции во входе.

Псевдокод для необходимых функций представлен ниже:

Функция **add** помещает в очередь дескриптор, если он не был создан до этого; эта функция не изменилась.

```
function CREATE(S_call, S_next, u, i, w)
    A ← Δ(S_call)
    if (∃ GSS node labeled (A, i)) then
        v ← GSS node labeled (A, i)
        if (there is no GSS edge from v to u labeled (S_next, w)) then
            add GSS edge from v to u labeled (S_next, w)
            for ((v, z) ∈ P) do
                (y, N) ← getNodes(S_next, u.nonterm, w, z)
                (_, _, h) ← y
                add(S_next, u, h, y)
```

```
            if N ≠ $ then
                (_, _, h) ← N; pop(u, h, N)
    else
        v ← new GSS node labeled (A, i)
        create GSS edge from v to u labeled (S_next, w)
        add(S_call, v, i, $)
    return v

function POP(u, i, z)
    if ((u, z) ∉ P) then
        P.add(u, z)
        for all GSS edges (u, S, w, v) do
            (y, N) ← getNodes(S, v.nonterm, w, z)
            add(S, v, i, y)
            if N ≠ $ then pop(v, i, N)

function PARSE
    R.enqueue(StartState, newGSSnode(StartNonterminal, 0), 0, $)
    while R ≠ ∅ do
        (C_S, C_U, i, C_N) ← R.dequeue()
        if (C_N = $) and (C_S is final state) then
            eps ← getNodeT(ε, i)
            (_, N) ← getNodes(C_S, C_U.nonterm, $, eps)
            pop(C_U, i, N)
        for each transition(C_S, label, S_next) do
            switch label do
                case Terminal(x) where (x = input[i])
                    T ← getNodeT(x, i)
                    (y, N) ← getNodes(S_next, C_U.nonterm, C_N, T)
                    if N ≠ $ then pop(C_U, i + 1, N)
                    R.enqueue(S_next, C_U, i + 1, y)
                case Nonterminal(S_call)
                    create(S_call, S_next, C_U, i, C_N)
    if SPPF node (StartNonterminal, 0, input.length) exists then
        return this node
    else report failure
```

6. Реализация

7. Эксперименты

Заключение

В рамках данной работы разработана и реализована модификация алгоритма GLL, работающая с расширенными контекстно-свободными грамматиками и показано, что полученный алгоритм повышает производительность поиска структур заданных с помощью контекстно-свободной грамматики в метагеномных сборках. Более детально, были получены следующие результаты:

- В качестве подходящего представления ECFG выбраны рекурсивные конечные автоматы.
- Спроектирована структура данных для представления леса разбора по ECFG на основе сжатого леса разбора(SPPF).
- Разработан алгоритм на основе Generalized LL, строящий лес разбора по ECFG.
- Алгоритм реализован в рамках проекта YaccConstructor.
- Проведены эксперименты, показавшие двухкратный прирост производительности на имеющихся метагеномных сборках по сравнению с существуюшим решением.
- Результаты работы успешно представлены на международной конференции “Tools and Methods of Program Analysis”(Москва, 2017г.)

Список литературы

Приложение

A. Псевдокод Generalized LL алгоритма

function ADD(L, u, i, w)
  if $(L, u, i, w) \notin U$ **then**
    $U.add(L, u, i, w)$
    $R.enqueue(L, u, i, w)$

function CREATE(L, u, i, w)
  $(X ::= \alpha A \cdot \beta) \leftarrow L$
  if ($\exists$ GSS node labeled (A, i)) **then**
    $v \leftarrow$ GSS node labeled (A, i)
    if (there is no GSS edge from v to u labeled (L, w)) **then**
      add GSS edge from v to u labeled (L, w)
      for $((v, z) \in \mathcal{P})$ **do**
        $y \leftarrow$ **getNodeP**(L, w, z)
        add(L, u, h, y) where h is the right extent of y
  else
    $v \leftarrow$ **new** GSS node labeled (A, i)
    create GSS edge from v to u labeled (L, w)
    for each alternative α_k **of** A **do**
      add$(\alpha_k, v, i, \$)$
  return v

function POP(u, i, z)
  if $((u, z) \notin \mathcal{P})$ **then**
    $\mathcal{P}.add(u, z)$
    for all GSS edges (u, L, w, v) **do**
      $y \leftarrow$ **getNodeP**(L, w, z)
      add(L, v, i, y)

function GETNODET(x, i)
  if $(x = \varepsilon)$ **then** $h \leftarrow i$
  else $h \leftarrow i + 1$
  $y \leftarrow$ find or create SPPF node labelled (x, i, h)
  return y

function GETNODEP$(X ::= \alpha \cdot \beta, w, z)$
  if (α is a terminal or a non-nullable nontermial) & $(\beta \neq \varepsilon)$ **then**
    return z
  else
    if $(\beta = \varepsilon)$ **then** $L \leftarrow X$
    else $L \leftarrow (X ::= \alpha \cdot \beta)$

```
        (_, k, i) ← z
        if (w ≠ $) then
            (_, j, k) ← w
            y ← find or create SPPF node labelled (L, j, i)
            if (∄ child of y labelled (X ::= α · β, k)) then
                y′ ← new packedNode(X ::= α · β, k)
                y′.addLeftChild(w)
                y′.addRightChild(z)
                y.addChild(y′)
        else
            y ← find or create SPPF node labelled (L, k, i)
            if (∄ child of y labelled (X ::= α · β, k)) then
                y′ ← new packedNode(X ::= α · β, k)
                y′.addRightChild(z)
                y.addChild(y′)
        return y
function DISPATCHER
    if R ≠ ∅ then
        (C_L, C_u, i, C_N) ← R.dequeue()
        C_R ← $
        dispatch ← false
    else stop ← true
function PROCESSING
    dispatch ← true
    switch C_L do
        case (X → α · xβ) where (x = input[i] | x = ε)
            C_R ← getNodeT(x, i)
            if x ≠ ε then  i ← i + 1
            C_L ← (X → αx · β)
            C_N ← getNodeP(C_L, C_N, C_R)
            dispatch ← false
        case (X → α · Aβ) where A is nonterminal
            create((X → αA · β), C_u, i, C_N)
        case (X → α·)
            pop(C_u, i, C_N)
function PARSE
    while not stop do
```

if *dispatch* **then** **dispatcher**()
else **processing**()
if SPPF node $(StartNonterminal, 0, input.length)$ exists **then**
return this node
else report failure