The Primordial Soup Algorithm

A Systematic Approach to the Specification of Parallel Parsers

Wil Janssen, Mannes Poel, Klaas Sikkel, Job Zwiers

University of Twente, Dept. of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail:{janssenw,mpoel,sikkel,zwiers}@cs.utwente.nl

Abstract

A general framework for parallel parsing is presented, which allows for a unified, systematic approach to parallel parsing. The Primordial Soup Algorithm creates trees by allowing partial parse trees to combine arbitrarily. By adding constraints to the general algorithm, a large class of parallel parsing strategies can be defined. This is exemplified by CYK, (bottom-up) Earley and de Vreught & Honig parsers. From such a parsing strategy algorithms for various machine architectures can be derived in a systematic way.

1 Introduction

In this paper we present a general framework for parallel parsing algorithms. Parsing can be seen as a process in which a set of partial parse trees is recognized. One starts with the productions as elementary trees. Small trees can be combined into larger trees, yielding ever larger and larger structures, until completed parses for a particular target sentence are produced. We envisage the set of recognized trees as a kind of primordial soup. Small trees float around and if they fit together they can be combined into a larger tree. This is, in a nutshell, the Primordial Soup paradigm.

In the most general approach, trees can combine in arbitrary ways. That is, a new tree can be created from two existing trees if there is a partial overlap between the trees. The overlapping part is unified. Tree creation is non-destructive, in the sense that a tree can be used more than once for the production of a larger tree. Or, to put it in a different way, the initial soup contains an abundant amount of raw material. Thus all relevant trees can actually be created.

The Primordial Soup Algorithm can be refined into a variety of *parsing strategies* by adding constraints, either on the allowed type of trees or on the way in which existing trees can be combined. A parsing strategy specifies *which* trees will be recognized, without bothering to specify control and data structures that must be added in order to arrive at practical implementations.

For the development of parallel implementations, the (initial) absence of control structure is an asset, as decisions about system architecture can be deferred to a later stage when the strategy has been fleshed out in more detail. Our specification of the Primordial Soup Algorithm allows for a systematic derivation of implementations of parsing strategies as is shown in a more detailed technical report [JPSZ]. These derivations are carried out within a partial order framework as introduced in [JPZ]. In the restricted space available here we concentrate on the Primordial Soup Algorithm as a framework for the *specification* of parsing strategies.

In section 2 the Primordial Soup Algorithm is introduced, exemplified by a CYK-like approach. In section 3, the formalism is slightly extended so as to allow for the description of almost any parallel or sequential parser.

2 The Primordial Soup

The Primordial Soup Algorithm will be introduced after some remarks about notation and parsers. We show that the algorithm is a generalization of well-known parsing strategies.

2.1 Preliminaries

We use the following notational conventions. Nonterminals are denoted by $A, B, \cdots \in N$; terminals are denoted by $a, b, \cdots \in \Sigma$. We write V for $N \cup \Sigma$, with typical elements $X, Y, \cdots$. Terminal strings are denoted by $s, t, u, v, w, \cdots \in \Sigma^*$,

arbitrary strings by $\alpha, \beta, \cdots \in V^*$.

Let $G = (N, \Sigma, P, S)$ be a context free grammar. Let $w = a_1 \cdots a_n \in \Sigma^*$ be the sentence. While executing an arbitrary parsing algorithm, we maintain a set of trees that might be subtrees of a parse for w. Let $\mathcal{F}bt$ be the class of finitely branching trees, in which all nodes have a label from some universal class of symbols. Let $\mathcal{T}(G) \subset \mathcal{F}bt$ be the class of trees that can be constructed from P; i.e., if some node is labelled A and its children $X_1, \cdots, X_n$, then $A \rightarrow X_1 \cdots X_n \in P$. We will usually write $\mathcal{T}$ for $\mathcal{T}(G)$; individual trees are denoted $\rho, \sigma, \tau, \cdots \in \mathcal{T}$.

We write *root*(τ) for the label of the root of a tree τ. The *yield* of a tree τ, denoted by *yield*(τ), is defined as the concatenation of the labels of the leaves. Clearly, *yield*$(\tau) \in V^*$. Note that leaves labelled ε (generated by empty productions) are not visible in the yield as ε disappears in concatenation. A tree τ is a *parse tree* for w if *root*$(\tau) = S$ and *yield*$(\tau) = w$. For arbitrary $w \in \Sigma^*$ a subclass $\mathcal{T}_w \subset \mathcal{T}$ is defined that contains trees τ with *yield*$(\tau) = a_i \cdots a_j$ for some substring $a_i \cdots a_j$ of w. $\mathcal{T}_w$ is called the set of *subparses* of w. The root of a subparse need not be S, it can be any nonterminal $A \in N$.

As a convenient notation for trees we write $\tau = \langle A \rightsquigarrow \alpha \rangle$ for an arbitrary tree with $A =$ *root*(τ) and $\alpha =$ *yield*(τ). In general $\langle A \rightsquigarrow \alpha \rangle$ is not uniquely determined, as every derivation $A \Rightarrow^+ \alpha$ defines a tree $\langle A \rightsquigarrow \alpha \rangle$. If we want to stress that a derivation $A \Rightarrow^+ \alpha\beta\gamma$ can be obtained as $A \Rightarrow^+ \alpha B \gamma \Rightarrow^+ \alpha\beta\gamma$ we write $\langle A \rightsquigarrow \alpha \langle B \rightsquigarrow \beta \rangle \gamma \rangle$ for the tree $\langle A \rightsquigarrow \alpha\beta\gamma \rangle$. Thus the tree notation is generalized into $\langle A \rightsquigarrow \xi_1 \cdots \xi_n \rangle$, where ξ_i is either a leaf or a subtree. This simple tree notation is extended with the following conventions:

- A tree $\langle A \rightsquigarrow \alpha \rangle$ corresponding to a single-step derivation $A \Rightarrow \alpha$ is also denoted as $\langle A \rightarrow \alpha \rangle$. This corresponds to a production $A \rightarrow \alpha \in P$.
- As a convenient shorthand, a tree

  $$\langle A \rightsquigarrow \alpha \langle B_1 \rightsquigarrow \beta_1 \rangle \cdots \langle B_n \rightsquigarrow \beta_n \rangle \gamma \rangle$$

  will be abbreviated to

  $$\langle A \rightsquigarrow \alpha \langle B_1 \cdots B_n \rightsquigarrow \beta_1 \cdots \beta_n \rangle \gamma \rangle.$$

2.2 Various bottom-up parsers

Our basic approach results from a generalization of various bottom-up parsing algorithms. The oldest and perhaps best known of these is the Cocke-Younger-Kasami (CYK) algorithm [You]. It requires the grammar to be in Chomsky Normal Form, i.e., productions have the form $A \rightarrow BC$ or $A \rightarrow a$. If we have trees $\tau_1 = \langle B \rightsquigarrow a_{i+1} \cdots a_k \rangle$ and $\tau_2 = \langle C \rightsquigarrow a_{k+1} \cdots a_j \rangle$ and if there is a production $A \rightarrow BC \in P$, we can construct a larger tree $\langle A \rightsquigarrow a_{i+1} \cdots a_j \rangle$ from τ_1 and τ_2. This can be continued until $\langle S \rightsquigarrow a_1 \cdots a_n \rangle$ has been derived, or no new trees can be constructed.

The CYK algorithm is usually described as a *recognizer*, rather than a parser. A recognition algorithm collects a set of items that denote the *existence* of trees, rather than trees themselves. If it is deduced that $A \Rightarrow^* a_{i+1} \cdots a_j$ (without having constructed a corresponding tree), this will be denoted by an item $[A \rightsquigarrow a_{i+1} \cdots a_j]$. In general, an item $[A \rightsquigarrow \alpha]$ denotes the existence of one or more trees $\langle A \rightsquigarrow \alpha \rangle$. The string w is grammatically correct if and only if an item $[S \rightsquigarrow w]$ can be recognized.

The CYK algorithm recognizes items of the form $[A \rightsquigarrow a_{i+1} \cdots a_j]$. For notational convenience, such an item is usually written as $[i, A, j]$. Thus we get the conventional description of CYK recognition: An item $[i, A, j]$ can be recognized iff $[i, B, k]$ and $[k, C, j]$ have been recognized previously for some $i < k < j$ and $A \rightarrow BC \in P$.

Several recognition and parsing algorithms deal with arbitrary context-free grammars along the same line as CYK, involving some more technicalities for handling productions of arbitrary length, including ε-productions. For example, a bottom-up variant of Earley's recognition algorithm [Ear, GHR] recognizes items of the form

$$[i, A \rightarrow \alpha \bullet \beta, j]$$

denoting the fact that $\alpha \Rightarrow^* a_{i+1} \cdots a_j$. That is, the *first part* of a production has been recognized. If $\beta = \varepsilon$, i.e. the item is of the form $[i, A \rightarrow \alpha \bullet, j]$, the entire production has been recognized; such an item denotes the existence of a tree $\langle A \rightsquigarrow a_{i+1} \cdots a_j \rangle$. We call this algorithm *Bottom-Up Earley* (BUE) in the sequel; the top-down filter of Earley's algorithm has been deleted so as to allow parallel bottom-up, rather than left-to-right processing of the string.

Still, BUE recognizes each individual nonterminal in left-to-right manner, for which there is no a priori reason. De Vreught and Honig [dVH] describe a similar, more general algorithm (which we abbreviate VH), using *double dotted items* $[i, A \rightarrow \alpha \bullet \beta \bullet \gamma, j]$ where $\beta \Rightarrow^* a_{i+1} \cdots a_j$. In this case β corresponds to a part of the string that has been recognized, whereas α and γ still need to be recognized.

Both BUE and VH can easily be extended

to *parsing* algorithms, producing partial parse trees of the form $\langle A \rightarrow \langle \alpha \rightsquigarrow a_{i+1} \cdots a_j \rangle \beta \rangle$ and $\langle A \rightarrow \alpha \langle \beta \rightsquigarrow a_{i+1} \cdots a_j \rangle \gamma \rangle$, respectively.

2.3 The Primordial Soup Algorithm

VH is by no means the most general algorithm. As the ultimate generalization we can allow *any tree in* $\mathcal{T}$. The top is a nonterminal and the leaves can be any symbol in V; a tree may or may not be part of a parse for w.

Initially we start with elementary trees that correspond to the productions in our grammar. New trees can be added by merging (copies of) existing trees which agree on their common parts. This can be seen as some kind of *unification process* on parse trees. The string is parsed when a tree $\tau = \langle S \rightsquigarrow a_1 \cdots a_n \rangle$ is produced; the algorithm terminates when no new trees can be added. Metaphorically speaking, one can think of the initial set of trees as a primordial soup in which small structures react with each other, creating ever larger and more complicated structures. We therefore call it the *Primordial Soup Algorithm*. Superficially, it may resemble the *unification space* of Vosse and Kempen [VK], who think of molecules floating in a test-tube and entering into chemical bonds with other molecules. The paradigms are different however, as in the primordial soup, unlike the test-tube, raw material abounds and and multiple copies of any structure can be created.

The most general version of the Primordial Soup Algorithm—allowing to combine trees by unification of arbitrary overlapping parts—is a formalism in which a wide variety of parsing algorithms can be specified with great ease. Before that, we first formalize a slightly limited, but somewhat easier version of the Primordial Soup Algorithm.

The algorithm starts of with an initial set of recognized trees $\mathcal{S}$ consisting of trees corresponding to the productions in our grammar. New trees can be added to $\mathcal{S}$ by taking combinations of existing trees. The simplest way to combine trees is the following.

Let $\sigma = \langle A \rightsquigarrow \alpha B \gamma \rangle \in \mathcal{S}$ and $\tau = \langle B \rightsquigarrow \beta \rangle \in \mathcal{S}$. We can unify the leaf B in σ with the root B in τ, yielding a new tree $\langle A \rightsquigarrow \alpha \langle B \rightsquigarrow \beta \rangle \gamma \rangle$. This tree is denoted by $\sigma \lhd \tau$. The (partial) function $\lhd : \mathcal{F}bt \times \mathcal{F}bt \rightarrow \mathcal{F}bt$ is called *composition*. Note that there can be multiple occurrences of B in $yield(\sigma)$, which means that $\sigma \lhd \tau$ need not be determined uniquely. Also, we will use the operator $\lhd$ in a liberal way, allowing more than one extension to be made at the same time. Let $\sigma = \langle A \rightsquigarrow \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \rangle$ and $\tau_i = \langle B_i \rightsquigarrow \beta_i \rangle$. We write

$$\sigma \lhd \tau_1, \tau_2$$

for the tree $\langle A \rightsquigarrow \alpha_0 \langle B_1 \rightsquigarrow \beta_1 \rangle \alpha_1 \langle B_2 \rightsquigarrow \beta_2 \rangle \alpha_2 \rangle$, using $\lhd$ as a polyadic operator with one left-hand argument and an arbitrary number of right-hand arguments.

As initial contents of the primordial soup, we take the trees $\langle A \rightarrow \alpha \rangle$ corresponding to productions $A \rightarrow \alpha \in P$. Such a tree $\langle A \rightarrow \alpha \rangle$ is called a *production tree* or a *production* for short. We define an operator $\mathcal{A} : 2^{\mathcal{T}} \rightarrow 2^{\mathcal{T}}$ that yields *all* new trees that can be composed from the contents of the soup by

$$\mathcal{A}(\mathcal{S}) \stackrel{\text{def}}{=} \{\sigma \lhd \tau_1, \cdots, \tau_k \in \mathcal{T} \mid \{\sigma, \tau_1, \cdots, \tau_k\} \subset \mathcal{S}\}.$$

This definition of $\mathcal{A}$ has one shortcoming, however. Rather than *all* parses for *all* sentences we only want the parses for one particular sentence $w \in \Sigma^*$. In general, this problem is tackled by redefining $\mathcal{A}$ as

$$\mathcal{A}(\mathcal{S}) \stackrel{\text{def}}{=} \{\sigma \lhd \tau_1, \cdots, \tau_k \in \mathcal{T} \mid \{\sigma, \tau_1, \cdots, \tau_k\} \subset \mathcal{S} \wedge \mathit{allowed}(\sigma \lhd \tau_1, \cdots, \tau_k)\}$$

in which a predicate *allowed* specifies which trees are allowed to be added. Which trees can be discarded right away, and which ones should be added to the soup? As we are only interested in trees that can be extended to parses for some specific sentence w, the terminal part of the yield should be *extendable* to w. That is, w can be produced from $yield(\tau)$ by replacing every nonterminal in τ with some string of terminals. Formally, for terminal strings $s \in \Sigma^*$ we define

$$\mathit{extends}(s, t) \stackrel{\text{def}}{=} \exists u, v \in \Sigma^* (t = usv),$$

i.e. s is a substring of t. For strings in V^* containing at least one nonterminal, we define *extends* recursively as

$$\mathit{extends}(\alpha B \beta, t) \stackrel{\text{def}}{=} \exists s \in \Sigma^* (\mathit{extends}(\alpha s \beta, t)).$$

Finally we define

$$\mathit{allowed}(\tau) \stackrel{\text{def}}{=} \mathit{extends}(yield(\tau), w),$$

in accordance with the informal definition given above. Note, however, that we still may create an infinite number of useless trees, simply by *not* adding terminals to the yield! If $yield(\tau) \in N^*$ then $\mathit{allowed}(\tau)$ holds: each leaf can be extended to ε, and the empty string is indeed a substring of w. In 3.2 we will see how this problem can be tackled in general; here we will only regard a subclass of $\mathcal{T}$ that does not contain trees with arbitrarily large nonterminal yields.

This finally allows us to define the Primordial Soup Algorithm.

Program *primordial_soup*
declare
  $\mathcal{S}$: set of $\mathcal{T}$
begin
  $\mathcal{S} := \{\tau \in \mathcal{T} \mid production(\tau)\}$;
  while $(\mathcal{A}(\mathcal{S}) - \mathcal{S}) \neq \emptyset$ **do** $\mathcal{S} := \mathcal{S} \cup \mathcal{A}(\mathcal{S})$
end {*primordial_soup*}

2.4 Specifying parse strategies

More specific and more useful instances of the algorithm can be defined by imposing restrictions on the trees to be added. A *strategy* is a characterization of trees that are to be added to the primordial soup $\mathcal{S}$ under some additional constraints. Different constraints specify different strategies. We call it strategy, rather than algorithm, as no control structure is specified explicitly. For the sake of simplicity we assume that $\mathcal{A}(\mathcal{S})$ is added all at once, but it should be understood that, if so desired, only a subsets of $\mathcal{A}(\mathcal{S})$ need be added at each step. A strategy can be refined into a (parallel or sequential) algorithm by adding control structure and data structures so as to keep track of intermediate results in an efficient manner. For examples of the design of parsing algorithms from such strategies, see [JPSZ].

Parsing strategies can be characterized by two types of restrictions: on the *types of trees* allowed in the soup and on the *operators* that create new trees from existing ones. Both kinds of restrictions are interchangeable most of the time; if trees are allowed to combine only in some specific way, the set of generated trees will be restricted, and vice versa.

As a simple example, we will specify a strategy for the CYK parser. To that end, we define an additional predicate

$$complete(\tau) \stackrel{\text{def}}{=} yield(\tau) \in \Sigma^*$$

i.e., a tree is *complete* if its yield does not contain any nonterminal. Such a tree can only be used as a right-hand side argument of a composition. Recalling that the CYK algorithm is defined only for grammars in Chomsky Normal Form (i.e., productions are of the type $A \rightarrow BC$ and $A \rightarrow a$), we can define the CYK strategy by

$$\mathcal{A}_{CYK}(\mathcal{S}) \stackrel{\text{def}}{=} \{\sigma \lhd \tau_1, \tau_2 \mid complete(\sigma \lhd \tau_1, \tau_2) \wedge allowed(\sigma \lhd \tau_1, \tau_2)\}.$$

Apart from the initial production trees, $\mathcal{S}$ will only contain trees of the form $\langle A \rightsquigarrow a_{i+1} \cdots a_j \rangle$. The *complete* predicate specifies that newly created trees have a terminal yield; this must be a subtring of w due to the *allowed* predicate. It is trivial to verify that *all* such trees are added to $\mathcal{S}$ in due course. Hence the specification of CYK is sound and complete.

3 Other parse strategies

We redefine the Primordial Soup Algorithm from section 2 in a more general manner, and show its power and elegance by specifying the parsing strategies of Bottom-Up Earley, De Vreught & Honig and some variants of CYK.

3.1 Unification and superposition

In section 2 we used only the composition operator $\lhd$ to create new trees from existing ones. Composition can be seen as a specific case of *superposition*, in which arbitrary overlapping parts of trees can be unified.

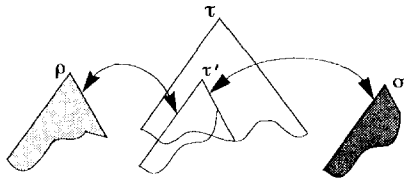
We will first define *unification*, which is a special case of superposition in which the roots of two trees are mapped onto each other. for the definition of unification, we use the derivation operator $\Rightarrow$ for trees. If $\tau = \langle A \rightsquigarrow \alpha B \gamma \rangle$ and $\sigma = \langle A \rightsquigarrow \alpha \langle B \rightarrow \beta \rangle \gamma \rangle$, we write $\tau \Rightarrow \sigma$. A tree σ is called an *extension* of τ if $\tau \Rightarrow^* \sigma$, where $\Rightarrow^*$ means applying the derivation $\Rightarrow$ zero or more times. Now two trees τ and σ *unify* if a tree ρ exists that is an extension of both σ and τ. I.e.,

$$unify(\sigma, \tau) \stackrel{\text{def}}{=} \exists \rho \in \mathcal{T}(\tau \Rightarrow^* \rho \wedge \sigma \Rightarrow^* \rho) .$$

ρ is called an *upper bound* of τ and σ. Furthermore, if σ and τ unify, there is a unique *least upper bound*, denoted by $\tau \sqcup \sigma$, satisfying

if $\tau \Rightarrow^* \rho$ and $\sigma \Rightarrow^* \rho$ then $\tau \sqcup \sigma \Rightarrow^* \rho$.

$\tau \sqcup \sigma$ is called the *unification* of τ and σ. Note that the roots of τ and σ coincide in $\tau \sqcup \sigma$. Unification can be generalized to *superposition* by allowing the root of one tree to be unified with an arbitrary node of the other tree, under the constraint that the overlapping parts of both trees are be identical; see Figure 1. This superposition operator is denoted by $\sim$. Note that, in general, superposition is not uniquely determined. Hence it is defined as a function $\sim : \mathcal{F}bt \times \mathcal{F}bt \rightarrow 2^{\mathcal{F}bt}$, whereas unification is defined as a partial function $\sqcup : \mathcal{F}bt \times \mathcal{F}bt \rightarrow \mathcal{F}bt$. For a more formal definition, see [JPSZ].



if $\sigma \sqcup \tau' = \rho$ for a subtree τ' of τ, then τ' is replaced by ρ.

Figure 1: superposition of trees

3.2 Some general restrictions

As discussed in 2.3, we do not want to recognize *all* trees leading to parses of *arbitrary* strings. We introduced the general idea that a tree is allowed only if the terminal part of the yield extends to the sentence. For the CYK algorithm, this simple criterion is fine. In general, however, it is too restrictive, in the sense that some familiar parsing algorithms cannot handle it. Suppose, for example, that a tree $\langle A \rightsquigarrow aB \rangle$ is extended with a production $\langle B \rightarrow bCd \rangle$ into $\langle A \rightsquigarrow abCd \rangle$. In principle, this should only be allowed if ab and d occur in w in this order. A parser which uses only local information, e.g. an LR(1) parser, cannot determine whether a terminal d occurs *somewhere* in the string, perhaps after a large substring produced by C.

We will use a rather more subtle scheme to match the yield of a tree against the sentence, so as to allow for refinement into arbitrary parsing algorithms. Having a tree $\langle A \rightsquigarrow aB \rangle$ we can check that a occurs in w and *mark* the leaf a accordingly. Marking a leaf is denoted by underlining the terminal symbol. The tree $\langle A \rightsquigarrow \underline{a}B \rangle$ can be extended to $\langle A \rightsquigarrow \underline{a}\langle B \rightarrow bCd \rangle \rangle = \langle A \rightsquigarrow \underline{a}bCd \rangle$ and then to $\langle A \rightsquigarrow \underline{ab}Cd \rangle$, irrespective of whether d occurs in the string at all.

The notion of marking terminals with occurrences in the string fits quite well to parsing natural languages, rather than un-interpreted context-free grammars. In practical NL parsing, the *word categories* rather than the individual words are used as terminals, although they are in fact pre-terminals. Using the word categories as terminals, a marked terminal is a word category applied to a word from the sentence.

As an example, consider the sentence the bird flies. The initial soup might contain:

$\langle S \rightarrow NP\ VP \rangle$	$\langle det \rightarrow$ the$\rangle$
$\langle NP \rightarrow det\ noun \rangle$	$\langle noun \rightarrow$ bird$\rangle$
$\langle VP \rightarrow verb \rangle$	$\langle noun \rightarrow$ flies$\rangle$
	$\langle verb \rightarrow$ flies$\rangle$

Word categories need not be uniquely defined. In this case the word flies fits into two categories. A tree $\langle NP \rightsquigarrow$ the *noun*$\rangle$ could be combined with $\langle noun \rightarrow$ flies$\rangle$, yielding a noun phrase the flies. This tree is ruled out by *extends*, however, as the flies does not extend to the bird flies.

In summary, we distinguish two types of initial trees:

$$\mathit{initial}(\tau) \stackrel{\text{def}}{=} \mathit{production}(\tau) \vee \mathit{marker}(\tau),$$

$$\mathit{production}(\langle A \rightarrow \alpha \rangle) \stackrel{\text{def}}{=} A \rightarrow \alpha \in P\ ,$$

$$\mathit{marker}(\langle a \rightarrow \underline{a} \rangle) \stackrel{\text{def}}{=} a \in T, \underline{a} \text{ in the sentence.}$$

The *extends* predicate can be defined so as to apply to strings of markings (i.e. words) rather than terminals. Furthermore, if we do not want to construct arbitrarily large trees with a non-marked yield, we can define

$$\mathit{allowed}(\tau) \stackrel{\text{def}}{=}$$
$$\mathit{extends}(\mathit{yield}(\tau), w) \wedge |\mathit{yield}(\tau)| < |w|.$$

Finally, allowing arbitrary tree construction with superposition ($\sim$) rather than composition ($\lhd$), a general version of the operator $\mathcal{A}$ is given by

$$\mathcal{A}(\mathcal{S}) \stackrel{\text{def}}{=} \{\rho \in \tau \sim \sigma \mid \tau \in \mathcal{S} \wedge \sigma \in \mathcal{S} \wedge \mathit{allowed}(\rho)\}.$$

The algorithm is now given by

Program *primordial_soup*
declare
  $\mathcal{S}$: set of $\mathcal{T}$
begin
  $\mathcal{S} := \{\tau \in \mathcal{T} \mid \mathit{initial}(\tau)\}$;
  while $(\mathcal{A}(\mathcal{S}) - \mathcal{S}) \neq \emptyset$ **do** $\mathcal{S} := \mathcal{S} \cup \mathcal{A}(\mathcal{S})$
end {*primordial_soup*}

For *acyclic* grammars (i.e., grammars that do not allow a derivation $A \Rightarrow^+ A$), only a finite number of trees can be constructed, hence the algorithm is guaranteed to halt. When a grammar is cyclic, an infinite number of parses exist. Every finite (subtree of a) parse will be found within a finite number of steps.

From the point of efficiency, the above algorithm isn't sensible at all. Its strength, however, derives from the fact that a very large class of parallel parsing algorithms can be defined as specializations, by constraining the general algorithm in various ways. Some examples will be given shortly.

We have concentrated on context-free grammars for the sake of simplicity. It should be clear, though, that extension to various types of unification grammars is straightforward.

3.3 Different breeds of trees

As we have seen in the CYK example, *complete trees are an important class of trees*. But, having introduced markers, it is obvious that we consider a tree to be complete only if the entire yield has been marked. Therefore we redefine

$$\mathit{complete}(\tau) \stackrel{\text{def}}{=} \mathit{yield}(\tau) \in \underline{\Sigma}^*.$$

Note that all marker trees are complete, and that production trees are complete iff they correspond to an ε–production.

Palm trees consist of a roof (corresponding to a single production) and a trunk (consisting of a number of adjacent complete trees). They are the result of composing production trees and complete trees. We can define them as

$$\mathit{palm}(\tau) \stackrel{\text{def}}{=} \tau = \langle A{\rightarrow}\alpha\, \langle \beta {\rightsquigarrow} \underline{v} \rangle\, \gamma \rangle \wedge \alpha\gamma \neq \varepsilon \wedge \beta \neq \varepsilon.$$

By notational convention, $A{\rightarrow}\alpha\beta\gamma$ is a production and $\underline{v} \in \underline{\Sigma}^*$. Note that in general β is a sequence of symbols $X_1 \cdots X_n$; each X_i is the root of a complete tree $X_i {\rightsquigarrow} \underline{v}_i$. Degenerate cases, with only a trunk ($\alpha\gamma = \varepsilon$) or only a roof ($\beta = \varepsilon = \underline{v}$) are excluded explicitly.

As a generalization of palm trees, we may consider trees with more than one trunk. This type of tree is denoted by *baobab*.[1]

$$\mathit{baobab}(\tau) \stackrel{\text{def}}{=} \tau = \langle A{\rightarrow}\alpha_0\, \langle \beta_1 {\rightsquigarrow} \underline{v}_1 \rangle\, \alpha_1 \cdots \langle \beta_n {\rightsquigarrow} \underline{v}_n \rangle\, \alpha_n \rangle \wedge \alpha_1 \cdots \alpha_n \neq \varepsilon \wedge \beta_1 \cdots \beta_n \neq \varepsilon.$$

For baobabs, like palms, we exclude degenerate cases. Note, however, that any palm is also a baobab. Palms and baobabs are illustrated in Figure 2.

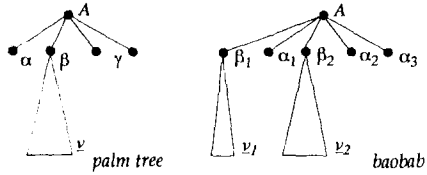


Figure 2: palm trees and baobabs

[1]The baobab is an African tree that has branches from which roots originate, supporting the roof. Such roots grow out to additional trunks.

3.4 CYK revisited

The only addition to our previous specification of CYK is that it should produce trees with marked yields. To that end, we can define an initial step

$$\mathcal{A}_0(\mathcal{S}) \stackrel{\text{def}}{=} \{\sigma \lhd \tau \in \mathcal{T} | \sigma, \tau \in \mathcal{S} \wedge \mathit{production}(\sigma) \wedge \mathit{marker}(\tau)\}.$$

For the remainder of the algorithm, production trees $\sigma = \langle A{\rightarrow}BC \rangle$ are composed with two complete trees τ_1 and τ_2 as usual, denoting ternary composition by $\sigma \lhd \tau_1, \tau_2$.

To keep in line with other algorithms to follow, we could alternatively define CYK with a binary composition operator. As a consequence, a new tree is created in two steps. First a production tree is combined with a complete tree, giving a palm. In the second step the palm is combined with a second complete tree, giving a new complete tree. We define two functions $\mathcal{A}$:

$$\mathcal{A}_1(\mathcal{S}) \stackrel{\text{def}}{=} \{\sigma \lhd \tau \in \mathcal{T} | \sigma, \tau \in \mathcal{S} \wedge \mathit{production}(\sigma) \wedge \mathit{complete}(\tau)\},$$

$$\mathcal{A}_2(\mathcal{S}) \stackrel{\text{def}}{=} \{\sigma \lhd \tau \in \mathcal{T} | \sigma, \tau \in \mathcal{S} \wedge \mathit{palm}(\sigma) \wedge \mathit{complete}(\tau) \wedge \mathit{allowed}(\sigma \lhd \tau)\}.$$

But as intermediate palm trees do not occur in the CYK algorithm as such, we define the function $\mathcal{A}'_{CYK}$ (for other than initial steps) as

$$\mathcal{A}'_{CYK}(\mathcal{S}) \stackrel{\text{def}}{=} \mathcal{A}_2(\mathcal{A}_1(\mathcal{S}) \cup \mathcal{S}).$$

A more liberal approach would be to allow the intermediate results to be in the soup:

$$\mathcal{A}''_{CYK}(\mathcal{S}) \stackrel{\text{def}}{=} \mathcal{A}_1(\mathcal{S}) \cup \mathcal{A}_2(\mathcal{S}).$$

For grammars in Chomsky Normal Form this hardly seems sensible. But when CYK is extended to *arbitrary CFGs*, a complete tree can be created from a production tree through an intermediate series of palm trees. If symbols in the right-hand side of a production can be recognized in arbitrary order, the condition $\mathit{palm}(\sigma)$ in the definition of $\mathcal{A}_2$ should be replaced by $\mathit{baobab}(\sigma)$.

3.5 Bottom-Up Earley

The BUE algorithm is defined for arbitrary context-free grammars. It is usually described as a recognition algorithm. An item $[i, A{\rightarrow}\alpha\bullet\beta, j]$ denotes the fact that $\alpha \Rightarrow a_{i+1} \cdots a_j$ has been recognized. From $[i, A{\rightarrow}\alpha\bullet B\gamma, j]$ and $[j, B{\rightarrow}\beta\bullet, k]$ a new item $[i, A{\rightarrow}\alpha B\bullet\gamma, k]$ can be derived. We will define the algorithm on trees, rather than items. Trees of the form $\langle A{\rightarrow}\langle \alpha {\rightsquigarrow} \underline{v} \rangle\, \beta \rangle$ are recognized for $v = a_{i+1} \cdots a_j$ a substring of w.

We define the set of *Earley trees* $\mathcal{E} \subset \mathcal{T}$ as

$$\mathcal{E} \stackrel{\text{def}}{=} \{\langle A \rightarrow \langle \alpha \rightsquigarrow \underline{v} \rangle \beta \rangle \in \mathcal{T} \mid A \rightarrow \alpha\beta \in P \wedge \underline{v} \in \underline{\Sigma}^*\}.$$

Note that productions ($\alpha = \varepsilon$) and complete trees ($\beta = \varepsilon$) are also included in $\mathcal{E}$. The operation of the algorithm is described by

$$\mathcal{A}_{BUE}(\mathcal{S}) \stackrel{\text{def}}{=} \{\, \sigma \lhd \tau \in \mathcal{E} \mid \sigma, \tau \in \mathcal{S} \wedge \mathit{allowed}(\sigma \lhd \tau)\}.$$

From the definition of $\mathcal{E}$ it follows that $\sigma \lhd \tau \in \mathcal{E}$ iff $\mathit{complete}(\tau)$ and the leftmost unmarked symbol of $\mathit{yield}(\sigma)$ is $\mathit{root}(\tau)$. The soundness follows from the definitions and completeness is trivially proven with induction on the size of the tree, hence the algorithm is correct.

3.6 De Vreught and Honig's algorithm

The VH algorithm also uses complete trees and palm trees, with the difference that the trunk of a palm tree does not necessarily cover the leftmost part of the roof. We define a set $\mathcal{V}$ of trees, analogously to the set of Earley trees by

$$\mathcal{V} \stackrel{\text{def}}{=} \{\langle A \rightarrow \alpha \langle \beta \rightsquigarrow \underline{v} \rangle \gamma \rangle \in \mathcal{V} \mid A \rightarrow \alpha\beta\gamma \in P \wedge \underline{v} \in \underline{\Sigma}^*\}.$$

The functions to combine trees are defined differently, however:

$$\mathcal{A}_1(\mathcal{S}) \stackrel{\text{def}}{=} \{\sigma \lhd \tau \in \mathcal{V} \mid \sigma, \tau \in \mathcal{S} \wedge \mathit{production}(\sigma) \wedge \mathit{complete}(\tau)\},$$

$$\mathcal{A}_2(\mathcal{S}) \stackrel{\text{def}}{=} \sigma \sqcup \tau \in \mathcal{V} \mid \sigma, \tau \in \mathcal{S} \wedge \mathit{palm}(\sigma) \wedge \mathit{palm}(\tau) \wedge \mathit{allowed}(\sigma \sqcup \tau)\},$$

$$\mathcal{A}_{VH}(\mathcal{S}) \stackrel{\text{def}}{=} \mathcal{A}_1(\mathcal{S}) \cup \mathcal{A}_2(\mathcal{S}).$$

The first operation was originally called *inclusion,* the second *concatenation.* The former combines a nonterminal tree and a complete tree to a palm tree, whereas the latter combines two palm trees into a palm tree with a wider trunk, using unification. It cannot result in a proper baobab because of the definition of $\mathcal{V}$. A subtle difference to the original algorithm is that we allow trunks of σ and τ to overlap, which is prohibited in their approach. It is not difficult to add this condition, if required.

A similar result is obtained by replacing the functions $\mathcal{A}_1$ and $\mathcal{A}_2$ by a function similar to the one used for Earley's algorithm (but now for trees in $\mathcal{V}$ instead of in $\mathcal{E}$). Thus a generalized bottom-up Earley parser, for which left-to-right parsing of a constituent is not necessary, is defined by

$$\mathcal{A}(\mathcal{S}) \stackrel{\text{def}}{=} \{\sigma \lhd \tau \in \mathcal{V} \mid \sigma, \tau \in \mathcal{S} \wedge \mathit{complete}(\tau) \wedge \mathit{allowed}(\sigma \lhd \tau)\}.$$

4 Conclusions

The Primordial Soup paradigm facilitates the specification of *parsing strategies*, i.e., high-level specifications or parsing algorithms, without explicit control flow and data structures.

A specification without control flow is a good basis for the design of a parallel implementation, as it allows a further refinement of the design before any decision on architecture is taken. For more details, see [JPSZ], where this has been exemplified with a design for a parallel CYK parser, using the Primordial Soup paradigm and the formalism introduced in [JPZ].

The Primordial Soup framework can be used to design new parsing algorithms by mixing features of existing algorithms. For example, the Earley operator for tree composition in combination with the De Vreught & Honig set of allowed trees yields a generalized Earley parser that has been rigorously defined in only two lines.

The specification of parsing strategies is given in a formalism closely resembling predicate logic. This makes it almost trivial to derive prototype implementations in (parallel) logic programming languages like Prolog or Parlog [JPSZ].

References

[Ear] J. Earley. An efficient Context-Free Parsing Algorithm. *Comm. ACM*, **13** (1970) 90–102.

[GHR] S.L. Graham, M.A. Harrison, W.L. Ruzzo. An Improved Context-Free Recognizer. *Trans. on Prog. Lang. and Syst.* **2** (1980) 415–462.

[JPSZ] W. Janssen, M. Poel, K. Sikkel, J. Zwiers. The Primordial Soup Algorithm. Memoranda Informatica 91-77, University of Twente (1991).

[JPZ] W. Janssen, M. Poel, J. Zwiers. Action Systems and Action Refinement in the Development of Parallel Systems, *CONCUR '91*, Springer Lectures Notes in Computer Science 527 (1991) 298–316.

[VK] T. Vosse, G. Kempen. A Hybrid Model of Human Sentence Processing: Parsing Right-Branching, Center-Embedded and Cross-Serial Dependencies. *Proc. 2nd Int. Workshop on Parsing Technologies*, Cancun, (1991) 73–78.

[dVH] J.P.M. de Vreught, H.J. Honig. A Tabular Bottom-Up recognizer. Report 89-78, Faculty of Technical Mathematics and Informatics, Delft University of Technology (1989).

[You] D.H. Younger. Recognition of context-free languages in time n^3. *Information and Control* **10** (1967) 189–208.