This is a reprint of an explorative paper from the first ASMICS workshop on parsing theory, Milano, Italy, October 1994, published as technical report 126-94 by the University of Milano. This version has some minor errors corrected. The reference [STin] did not appear in the meantime and is therefore now marked as such; however [STrt] is to appear and is a shortened version of the cited [STin].

Towards a Unifying Theory of Context-Free Parsing

Thomas Schöbel-Theuer
Universität Stuttgart, Institut für Informatik
Breitwiesenstr. 20-22, 70565 Stuttgart
schoebel@informatik.uni-stuttgart.de

May 13, 1996

Abstract

We present an approach to context-free parsing which builds up a meta-theory from scratch on this subject. From this meta-theory one can derive any parsing algorithm known to the author (among many others, e.g. Earley's, Tomita's, LL, $LR(k)$ etc.) in a constructive way as well as new algorithms having promising properties. Conclusions from this theory are not only, that all algorithms stem from a single source regardless of the way they have been originally invented, but also open a way to derive specialized algorithms suited for particular demands in practice.

This paper is an informal introduction to a theory treated in much more detail in [STin]. Formalizations, proofs and detailed discussions can be found there.

Because of limited space we can present only some underlying ideas of a theory which unifies all context-free parsing techniques known to the author. We hope to convince the reader that practically all known parsing algorithms stem from a single source. This claim does apply both to general parsing methods like Earley's or Lang's (resp. Tomita's) as well as known deterministic methods like LL, LC or $LR(k)$.

We assume the reader to be familiar with the field of context-free parsing; our notation will be similar to [HU79]. Terminal symbols are denoted by small letters, nonterminals by capital letters, and strings by Greek letters or x, y, z. For detailed definitions and formalizations, see [STin].

1 "Simple" representations of sentential forms

It is intuitively clear that nearly any existing parsing method can be characterized by a *data structure* representing sentential forms of the grammar. Canonical parsing methods represent left-sentential forms; more general methods may represent arbitrary sentential forms. A core of forms is derived from the start symbol of the grammar, but may be enhanced by additional ones. In contrast to parsing schemata, we focus on the *meaning of a data structure*, not on the way of *how* something has to be computed or *in which order* or *how the computations* are interrelated.

1.1 Recognition graph

We start with some arbitrary objects $\dot{\mathsf{v}}$ forming a nonempty countable set $\dot{\mathsf{V}}$. The objects are treated as black boxes; all we know about them is their existence. To each object, a language (i.e. a set of words) over some alphabet V is assigned. To set up this assignment, we use a function $\mu : \dot{\mathsf{V}} \to 2^{V^*}$ called labelling function.

Next, we may have one or more binary relations on the set $\dot{\mathsf{V}}$. Assume that $\dot{\mathsf{E}} \subseteq \dot{\mathsf{V}} \times \dot{\mathsf{V}}$ is such a relation, then we call $\dot{\mathsf{V}}$ a set of nodes (vertices) and $\dot{\mathsf{E}}$ a set of edges, choose some node $\dot{\mathsf{s}} \in \dot{\mathsf{V}}$ called root, and put it all together to a structure $\dot{\mathsf{G}} := (\dot{\mathsf{V}}, \dot{\mathsf{E}}, \dot{\mathsf{s}}, \mu)$ shortly called graph. In conventional terminology, we have a labelled directed graph with a dedicated root node. Note that we write all parts belonging to a graph with a dot on it, to distinguish for example the nodes $\dot{\mathsf{V}}$ from an alphabet (vocabulary) V.

Of main interest for us is the language associated to such a graph. As the language $L(\dot{\mathsf{G}})$, we define

$$L(\dot{\mathsf{G}}) := \bigcup_{\dot{\mathsf{v}} \in \dot{\mathsf{V}}} \mu(\dot{\mathsf{v}})$$

This general definition is useful whenever the labelling function μ is somehow restricted, e.g. when the number of words $|\mu(\dot{\mathsf{v}})| \leq k$ is bounded by some constant k. Otherwise we would be able to denote the whole language by one single node.

We shall consider two graphs as equivalent if they represent the same language.

Since we are speaking about context-free recognition, we have to relate our labelled graphs to this problem area. We do this by defining:

$\dot{\mathsf{G}}$ is a recognition graph (for some grammar G) iff

$$L(\dot{\mathsf{G}}) = F_L(G)$$

where $F_L(G)$ is the set of left-sentential forms of some context-free grammar $G = (\Sigma, V, S, P)$ with terminal alphabet Σ, vocabulary V, start symbol S, and productions P as usual.

This definition of a recognition graph which represents all derivable left-sentential forms of the grammar is the backbone of our theory because such a graph is used (usually implicitly) in nearly every parsing algorithm. However, in many cases it is not obvious that an algorithm like $LR(k)$ uses such a graph.

Before proceeding, we look at some special cases of graphs: A tree is a fully connected graph where each node except the root has exactly one predecessor, and the root has no predecessor. A forest is a graph where each node has at most one predecessor. A labelled graph is in normal form, if for any node all sons are labelled differently.

1.2 Total labelling

First some basic terminology known from graph theory: The set of paths $\Psi(\dot{\mathsf{v}}, \dot{\mathsf{w}})$ is the set of all finite sequences of nodes leading from $\dot{\mathsf{v}}$ to $\dot{\mathsf{w}}$. If the first parameter of Ψ is the root node $\dot{\mathsf{s}}$, we shortly write $\Psi(\dot{\mathsf{v}}) := \Psi(\dot{\mathsf{s}}, \dot{\mathsf{v}})$. Note that in a cyclic graph, this set may contain infinitely many paths each having finite length.

By the total labelling $\vec{\mu}\ (\dot{\mathsf{v}})$, we denote the concatenation of the label languages of all paths to the node $\dot{\mathsf{v}}$, more formally

$$\vec{\mu}\ (\dot{\mathsf{v}}) := \bigcup_{(\dot{\mathsf{s}},\ldots,\dot{\mathsf{v}}) \in \Psi(\dot{\mathsf{v}})} \mu(\dot{\mathsf{s}}) \ldots \mu(\dot{\mathsf{v}})$$

This also works in the other direction by taking the reverse path:

$$\overleftarrow{\mu}\,(\dot{\mathsf{v}}) := \bigcup_{(\dot{\mathsf{s}},\ldots,\dot{\mathsf{v}})\in\Psi(\dot{\mathsf{v}})} \mu(\dot{\mathsf{v}})\ldots\mu(\dot{\mathsf{s}})$$

In the general form, the languages of the labels are concatenated. Often the label languages consist of one single word; in this case we write one-element sets as the element itself (as far as no confusion can arise).

As one can see easily, finite graphs having single-element labels correspond to finite automata where the input symbols are assigned to the nodes$\hat{=}$states (like Moore automata). Mealy automata can be obtained analogously by using edge labels and by building the total labelling over the edges of a path.

Important differences to finite automata are:

- the graph may be infinite
- by the total labelling, a language is assigned to *each* node, not only to dedicated final states.

1.3 Some further definitions

String items I_V of some vocabulary V are denoted $\alpha\cdot\beta$ where $\alpha,\beta\in V^*$ and $\cdot$ is a new meta symbol not appearing in V. Production items I_P of some grammar $G=(\Sigma,V,P,S)$ are widely known in $LR(k)$ and Earley parsing and are written $A\to\alpha\cdot\beta$ where $A\to\alpha\beta\in P$. In this paper we will mostly use production items and therefore call them shortly items. The prefix π of an item is the part before the dot, written

$$\pi(A\to\alpha\cdot\beta) := \alpha$$

The infix ι of an item is the symbol following the dot, written

$$\iota(A\to\alpha\cdot X\beta) := X$$

The suffix σ of an item is the remainder after the infix, written

$$\sigma(A\to\alpha\cdot X\beta) := \beta$$

Left-sentential forms can also be divided uniquely into prefix, infix and suffix, where the prefix yields all terminals until the first nonterminal appears in the sentential form, the infix yields this first nonterminal (commonly used to do the next left-derivation), and the suffix yields the rest behind the infix. More formally, let $xA\alpha\in F_L(G)$. Then

$$\pi(xA\alpha) := x,\ x\in\Sigma^*$$

$$\iota(xA\alpha) := A,\ A\in(V-\Sigma)\cup\{\varepsilon\}$$

$$\sigma(xA\alpha) := \alpha,\ \alpha\in V^*$$

such that $A=\varepsilon\Longrightarrow\alpha=\varepsilon$.

The last condition guarantees the uniqueness of the partition, since when the left-sentential form contains only terminal symbols, both the infix and the suffix have to be empty.

1.4 The tree of possible left derivations (TPLD)

Now we look at an example graph representing all the left-sentential forms of a grammar. In this example, the graph is a tree. Some similarity with Earley's algorithm is expressed by naming the rules predictor, scanner and completer.

Definition: The tree of possible left derivations (TPLD) of some grammar G is the smallest tree with root $\dot{\mathsf{s}}$ labelled $S' \to \cdot S \dashv$ produced by the following rules:

Predictor: Let $\dot{\mathsf{v}}$ be a node with label $A \to \alpha \cdot B\beta$. Then there is also a son $\dot{\mathsf{w}}$ of $\dot{\mathsf{v}}$ labelled $B \to \cdot\gamma$ for each production $B \to \gamma \in P$.

Scanner: Let $\dot{\mathsf{v}}$ be a node with label $A \to \alpha \cdot a\beta$. Then there is also a brother $\dot{\mathsf{v}}'$ of $\dot{\mathsf{v}}$ labelled $A \to \alpha a \cdot \beta$.

Completor: Let $\dot{\mathsf{v}}$ be a node with label $B \to \gamma\cdot$ and $\dot{\mathsf{u}}$ the father of this node labelled $A \to \alpha \cdot B\beta$. Then there is also a brother $\dot{\mathsf{u}}'$ of $\dot{\mathsf{u}}$ labelled $A \to \alpha B \cdot \beta$.

The result of this definition is a usually infinite tree representing all left-sentential forms of the grammar by using the total labelling generated by the suffixes of the labels.

Theorem: in a TPLD, for any node $\dot{\mathsf{v}}$ the following holds:

$$\exists_{x \in \Sigma^*} : \; x \; \iota \circ \mu(\dot{\mathsf{v}}) \; \overleftarrow{\sigma \circ \mu}\,(\dot{\mathsf{v}}) \in F_L(G)$$

Explanation: for any node $\dot{\mathsf{v}}$, there exists a terminal string x such that x followed by the infix ι of the label $\mu(\dot{\mathsf{v}})$ followed by the total suffixes of the path from $\dot{\mathsf{v}}$ to the root is a left-sentential form. The proof is by easy induction over the predictor, scanner and completor rules. Moreover, this string x associated to a new node generated by some rule is either the same or is a prolongation of the string associated to the old node it was generated from.

Definition: A matching TPLD is a TPLD where all branches not matching some input word $a_1 \ldots a_n \dashv$ are cut off. Matching means that the above string x must be a prefix of the input word $a_1 \ldots a_n \dashv$.

Since in a matching TPLD the terminal string x is always a prefix of the input word, we can code this prefix also into the tree by using another labelling function $\chi : \dot{\mathsf{V}} \to \mathbb{N} \cup \{\bot\}$. Then the whole left-sentential form is represented by $a_1 \ldots a_{\chi(\dot{\mathsf{v}})} \; \iota \circ \mu(\dot{\mathsf{v}}) \; \overleftarrow{\sigma \circ \mu}\,(\dot{\mathsf{v}})$ if the input position χ is incremented accordingly in the scanner rules whenever the shifted-over terminal matches the input. In case $\chi(\dot{\mathsf{v}}) = \bot$ the node $\dot{\mathsf{v}}$ by definition does not contribute to the language $L(\dot{\mathsf{G}})$; this can be used to build some noise into the graph which is required for the simulation of some sophisticated parsing algorithms.

To overcome the problems of an infinite structure, we will demonstrate later how to compress such a tree to a graph with polynomial behaviour.

As suggested by the names of the rules, the matching TPLD has a close relation to Earley's algorithm: Apart from the fact that it is an infinite tree, the main difference is that the edge relation $\dot{\mathsf{E}}$ is directly between items, not indirectly via a pointer i and an Earley-stateset S_i.

1.5 Tree generators

The matching TPLD ist just one example of a recognition graph. In order to build up other tree structures for recognition purposes, we separate the rules for building up the structure of the tree from the rules which show how the tree is labelled.

Definition: A tree generator is a set , $\subseteq Z^* \times (\Sigma \cup \{\varepsilon\}) \times \bar{Z}^*$, where Z is some set of states (also called labels) and $\bar{Z} := Z \times \{true, false\}$ is an association of boolean values to

the states. , is containing rules of the form $((y_1,\ldots,y_k),X,(\tilde{z}_1,\ldots,\tilde{z}_l))$ with the following meaning:

If there exists a chain $\dot{v}_1,\ldots,\dot{v}_k$ of nodes labelled appropriately $\mu(\dot{v}_i)=y_i$, then there also exists another chain $\dot{v}'_1,\ldots,\dot{v}'_l$ with common father $\dot{v}_0$ (a so-called brother chain) with labels $\mu(\dot{v}'_i):=z_i$ where $\tilde{z}_i=(z_i,b_i)$. The boolean component b_i is used to influence the χ-labelling in the following way:

$$\chi(\dot{v}'_i):=\begin{cases}\chi(\dot{v}_k)+|X| & \text{if } \tilde{z}_i=(z_i,true)\\ \bot & \text{if } \tilde{z}_i=(z_i,false)\end{cases}$$

In the following, we denote the generation of $\bot$-nodes (i.e. $b_i=false$) by brackets.

With this model, it is rather easy to specify rules for building up trees. As can be seen, this model is very similar to (nondeterministic) pushdown automata: Just view the (unique) path from the root to a node $\dot{v}$ as contents of the stack. Then a rule $((y_1,\ldots,y_k),X,(\tilde{z}_1,\ldots,\tilde{z}_l))$ of the tree generator can be interpreted as popping off the left-hand side $y_k,\ldots,y_1$, scanning the input symbol X, and then pushing the right-hand side $z_1,\ldots,z_l$. However, there are some important differences to pushdown automata, from which are the most important:

- tree generators operate in parallel, with no specified order
- "pops" do not remove old calculations

The rules for generating the TPLD can be easily expressed as a tree generator: Let $G=(\Sigma,V,S,P)$ be a grammar. The TPLD tree generator , $\subseteq I_P^*\times(\Sigma\cup\{\varepsilon\})\times\tilde{I_P}^*$ is defined by

$$A\to\alpha\cdot B\beta\in I_P\wedge B\to\cdot\gamma\in I_P \textit{ iff. } ((A\to\alpha\cdot B\beta),\varepsilon,([A\to\alpha\cdot B\beta],B\to\cdot\gamma))\in ,$$

$$A\to\alpha\cdot a\beta\in I_P \textit{ iff. } ((A\to\alpha\cdot a\beta),a,(A\to\alpha a\cdot\beta))\in ,$$

$$A\to\alpha\cdot\in I_P\wedge B\to\gamma\cdot A\delta\in I_P \textit{ iff. } ((B\to\gamma\cdot A\delta,A\to\alpha\cdot),\varepsilon,(B\to\gamma A\cdot\delta))\in ,$$

Note: the first line corresponds to the predictor, the second to the scanner, and the last to the completor.

1.6 The general compression theorem in graphs

Definition: A subgraph $\dot{S}(\dot{G},\dot{v})$ of a graph $\dot{G}$ for some node $\dot{v}$ consists of all nodes and edges reachable from the new root node $\dot{v}$.

Theorem: Let $\dot{v}$ and $\dot{v}'$ be two different nodes of a labelled recognition graph $\dot{G}=(\dot{V},\dot{E},\dot{s},\mu,\chi)$. Whenever the conditions $\mu(\dot{v})=\mu(\dot{v}')$, $\chi(\dot{v})=\chi(\dot{v}')$ and $L(\dot{S}(\dot{G},\dot{v}))=L(\dot{S}(\dot{G},\dot{v}'))$ hold for a certain representation of the graph language by total labellings, one can melt the nodes $\dot{v}$ and $\dot{v}'$ by uniting their predecessors in the graph, and by choosing one of the sets of successors of $\dot{v}$ or $\dot{v}'$ as set of successors for the melted node. The resulting graph is equivalent to the old one, i.e. representing the same language.

There is an interesting special case of this theorem: $\dot{v}'$ may be a member of the subgraph $\dot{S}(\dot{G},\dot{v})$ (or vice versa), but the infinite sublanguages of both $\dot{v}$ and $\dot{v}'$ may be even the same (in the case of the TPLD, this may for example occur in a grammar with left recursion).

Now when we melt these two nodes, we get a *cycle* in the graph. This cycle is correct, since it now represents the infinite nesting of sublanguages occurring in the tree due to the lemma of Bar-Hillel.

Note that the compression theorem can also be applied in reverse order to split nodes.

Theorem: In a graph $\dot{\mathsf{G}}$ generated by a tree generator , , all chains with same labels μ und χ produced by the right-hand sides of some rule can be melted.

This theorem is a simplified version of the more general one found in [STin], where some technical details have to be handled in order to melt nodes which may have labels $\chi(\dot{\mathsf{v}}) = \perp$.

When this theorem is fully applied to the matching TPLD, one gets nearly the same parsing algorithm as Earley's. The main difference is that back pointers are not indirect by input positions i pointing to a set of states where the completor has to find out which items are to be shifted over, but rather directly addressing the appropriate items backwards via the edges of the graph.

In [STin], a general parsing algorithm based on the above theorems is presented which can handle arbitrary tree generators. If it is fed with the tree generator for the TPLD, it runs with asymptotically the same time and space complexities as Earley's algorithm, even with unambiguous grammars or bounded state grammars (see [Ear70]).

1.7 Some other equivalence relations

Only for purpose of completeness, we mention some other equivalence relations besides the general compression theorem which can be established on recognition graphs. With these equivalence relations, one can build up variants and improvements of generator-based parsing which cannot be presented here due to lack of space.

Removal of ε-nodes: If the relevant part of a label contributing to the total labelling (in case of the TPLD, this is the suffix of the μ-labelling) is empty (i.e. ε), and if this node has exactly one predecessor and exactly one successor, then this node can be removed from the graph by using a shortcut edge directly connecting the predecessor to the successor. This theorem can also be applied in reverse order, by introducing an ε-node into any edge of the graph.

However, this works only in $1:1$ correspondences of nodes. To achieve the same in $n:m$ correspondences where an ε-node has n predecessors and m successors, we have to connect all n predecessor nodes to all m successors when removing the ε-node. Whenever some n nodes called sources are each connected with some m nodes called destinations, we call them together a clique. For the reversal of this theorem, we have to ensure that the n sources and m destinations form a clique with nm connections, before we can introduce a new ε-node in the middle which now reduces the number of edges to $n+m$.

It is also possible to move on parts of string labels from one node to another, but only for one fixed concatenation direction. Here we show the construction for the concatenation direction $\rightarrow$. To ensure correctness, we must have a hard clique, where each of the n sources has no other successors than the m destinations and vice versa. Then, if the sources have labels $\alpha_i\beta$ and the destinations have labels γ_j, then we can move on the β from the sources to the destinations. As a result, the sources are now labelled $\beta\gamma_j$, while the destinations are labelled α_i.

1.8 Shift-reduce parsing

As another example of a recognition graph, we present a shift-reduce parsing which is both similar to the TPLD and to the classical stack notation. The difference to the TPLD is the following: Whenever a *brother* is generated in the TPLD (this can occur in the scanner and in the completor), we now generate a *son* instead. However, to ensure the correctness of the total labelling, we have to change the χ-label of the father to $\perp$, such that this node does not count any more for the thus modified total labelling. As before, we denote the generation of $\perp$-nodes (i.e. $b_i = false$) by brackets.

Let G be a grammar. The shift-reduce tree generator , is defined by

$$A \to \alpha \cdot B\beta \in I_P \wedge B \to \cdot\gamma \in I_P \textit{ iff. } ((A \to \alpha \cdot B\beta), \varepsilon, ([A \to \alpha \cdot B\beta], B \to \cdot\gamma)) \in ,$$

$$A \to \alpha \cdot a\beta \in I_P \textit{ iff. } ((A \to \alpha \cdot a\beta), a, ([A \to \alpha \cdot a\beta], A \to \alpha a \cdot \beta)) \in ,$$

$$A \to \alpha\cdot \in I_P \wedge B \to \gamma \cdot A\delta \in I_P \textit{ iff. } \alpha = X_1 \ldots X_k \text{ and}$$

$$((B \to \gamma \cdot A\delta,\ A \to \cdot\alpha,\ A \to X_1 \cdot X_1 \ldots X_k,\ \ldots,\ A \to X_1 \ldots X_{k-1} \cdot X_k,\ A \to \alpha\cdot),$$

$$\varepsilon,\ ([B \to \gamma \cdot A\delta],\ B \to \gamma A \cdot \delta)) \in ,$$

The difference to the TPLD is that some old calculations are not removed from the stack, but accumulate until a sequence $A \to \cdot X_1 \ldots X_p, \ldots, A \to X_1 \ldots X_p\cdot$ (the so-called handle) is on the stack. Then the completor has to remove this whole sequence.

To see the connection to classical *nondeterministic* shift-reduce parsing, we introduce another function on items, the pre-infix ι', which is defined by $\iota'(A \to \alpha X \cdot \beta) := X$ and $\iota'(A \to \cdot\alpha) := \varepsilon$, i.e. the pre-infix is the one symbol *before* the dot. It is now easy to see that we just have to build up the total labelling over this pre-infix ι' and to consider *all* nodes even with $\chi = \perp$: we then have the stack contents of a classical shift-reduce stack.

1.9 Leftmost versus rightmost derivations

When we consider the connection between the TPLD (which could be characterized as a variant of nondeterministic $LL(0)$ parsing) and shift-reduce parsing, which is commonly called a "bottom-up" method, we easily arrive at the following:

Theorem: For all nodes $\dot{\mathsf{v}} \in \dot{\mathsf{V}}$ in the TPLD the following holds, provided that $a_1 \ldots a_n \in L(G)$:

$$a_1 \ldots a_{\chi(\dot{\mathsf{v}})}\ \iota\circ\mu(\dot{\mathsf{v}})\ \overleftarrow{\sigma\circ\mu}\ (\dot{\mathsf{v}}) \in F_L(G)$$

$$\overrightarrow{\pi\circ\mu}\ (\dot{\mathsf{v}})\, a_{\chi(\dot{\mathsf{v}})+1} \ldots a_n \in F_R(G)$$

In words, if we concatenate the prefixes of the items from the root to $\dot{\mathsf{v}}$ and append the rest of the input, then we get a **right sentential form**. This can be seen easily because it is nearly mirror symmetric to the left-sentential form. Note that in classical LR theory, roughly the total prefix $\overrightarrow{\pi\circ\mu}\ (\dot{\mathsf{v}})$ is called *viable prefix*.

The consequence from this theorem is relevant to the classification of parsing techniques: Since it is simply a matter of interpretation, whether one and the same TPLD represents left- or right-sentential forms, the classical distinction between LL and LR, where the second letter should denote Left resp. Right-derivation, becomes therefore questionable. In our opinion, the traditional and well-established notions LL and LR should no more be

associated with the meaning of left- or right-derivation; not only these meanings found in many text books, but also the traditional "top-down" and "bottom-up" characterizations are not totally appropriate. For a more detailed discussion, see [STin].

1.10 Some variants of generator-based parsing

Only some variants can be touched here shortly; for deeper investigations see [STin].

True bottom-up parsing: Just relax the so-called *top-down restriction*, and predict always all items $A \to \cdot\gamma$ regardless of the context. As a result, the graph will represent some "wrong" sentential forms, but as is argued in [STin], these wrong forms do not contribute to the result of the parsing process, namely representing all parse trees. Methods with a top-down restriction should therefore be considered "natural", since relaxing it produces some trial-and-error calculations which can be avoided.

Nondirectional, island-driven parsing etc. can also be done by using forests instead of trees for intermediate representations. However, we get nearly the same graph in the end as would result from the corresponding directional method.

FIRST- and FOLLOW-restrictions are essentially restrictions on the relation $\dot{\in}$.

Introducing lookahead can be described either as remembering input symbols by additional labelling functions and then delaying parsing decisions, or as remembering more context about expected terminalisations of the total suffix and matching this against the input.

2 "More complex" representations

In this main section we briefly sketch a general method for precomputing information out of the grammar, such that at run time less operations are necessary to build up a recognition graph.

2.1 Macro graphs

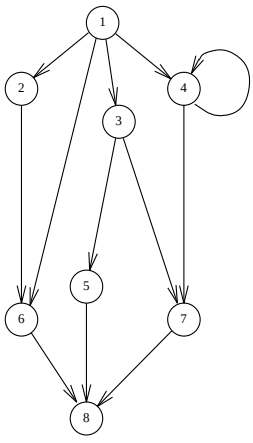

Figure 1: Micro graph

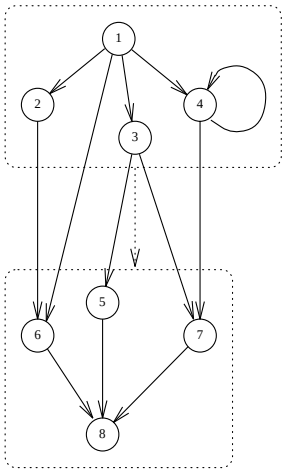

Figure 2: Micro graph with macros

The basic idea is relatively simple: We divide the the set of nodes of a graph into chunks, called partitions or macros. For each partition, we create a macro node, and assign to it the partition of the underlying micro graph. Then, we build up another (usually less complex) graph structure at the macro level by adding macro edges which contain a representation of the underlying micro edges of the micro graph.

In figures 1 and 2, the labels of the micro graph are left out; instead, the micro nodes are numbered. In figure 2, the dotted boxes indicate which micro nodes and edges are packed into one macro. As can be seen in this example, one cannot conclude from the dotted edge (macro edge) which micro edges connect the two macros.

A formalization of this rather simple idea is much more complicated and can be found in [STin]. To show a few of the problems, here are some characterizations.
Elements of a macro node are:

- a partition of the micro graph
- dedicated input/output nodes
- all micro nodes are numbered

Elements of a macro edge are:

- a set of pairs (a, b) of numbers representing connections between the numbered input/output nodes of the connected macro nodes.

In the macro graph, a labelling function assigns partitions of the micro graph to the macro nodes, and another labelling function assigns sets of pairs of numbers to the macro edges.

In order to get a finite set of such labels for the macro graph, we have to renumber all micro nodes in a partition beginning from 1. Also, the macro edges must be labelled *uniquely*, that means from the labels of the involved macro nodes the set of pairs must follow in a unique way.

General problem: Graph isomorphism.

Solution: Coding algorithm.

Due to the complexity of the graph isomorphism problem, the precomputing is rather expensive. However, since the partitions are relatively small in practice and also the micro nodes carry rich label information, there should be less room for permutations and the problem should therefore be tractable.

A general algorithm for precomputing macros is given in [STin, Chapter 4]. As parameters of this algorithm, one can supply an extremely wide variety of heuristics about what should be packed into the macros. The generality of this algorithm lies in the fact that it produces a new tree generator , $'$ from an old one , , whose macros contain subgraphs of what would be in the parsing graph at run time when parsing with , (modulo some splits).

2.2 $LR(0)$ simulation by macro graphs

We now show in an intuitive manner that there is a special packing heuristic for producing macros, which can simulate classical $LR(0)$ parsing.

The basic idea is to use the shift-reduce tree generator of section 1.8 and to pack exactly the same items into a macro that would be packed into an item set by the classical subset algorithm of $LR(k)$. In this way, we get a true simulation of $LR(0)$.

(The reader may take an example from [ASU86, page 235] where the lookahead information has to be wiped out. Imagine that the circles show the (static) macro borders. Add (static) micro edges inside the circles and between the circles along the depicted arcs which can be interpreted as (static) macro arcs. The whole has to be *instantiated* at run time.)

Since the structure of the underlying shift-reduce micrograph is reconstructible from the precomputed macros, it follows that (nondeterministic) $LR(0)$ parsing is nearly the same as a variant of Earley parsing. The main difference is that operations at run time are avoided by precomputing them into macros, such that instead of instantiating many micro nodes, we can instantiate a whole macro in one step. In the deterministic special case, when the relevant portion of a recognition is limited to a linear chain, the recognition power in terms of language classes is much bigger than with LL since the step relation on the instantiated macro nodes is more dense, because different paths or even cycles of the micro graph are packed into a macro.

As one can also see, the macroization technique is much more powerful than the classical construction with *sets* of items, since a macro can contain more complicated structures denoting complex representations.

2.3 Conclusion: More powerful macros

Precomputing of macros can be done by computing a step relation on macros in an incremental way: apply all effects of the micro tree generator to a macro, thereby enlarging it by the newly generated chains. Then split the macro somewhere into as many pieces as desired (or don't split it at all). By applying all rules in combination with any possible input symbols, one can get a macroization of the tree generator which is itself again a tree generator operating on larger chunks of graphs. By various heuristics, one can decide which pieces are to be packed into which macros.

As a general idea, splitting of macros may be delayed, yielding complex macros. For example, when applying a macroization to the TPLD, one can transform a regular sublanguage of the grammar to a tree generator which does no push and pop operations during the parsing of this regular sublanguage, by always keeping the boundedly growing stack (which must be achieved by elimination of ε-nodes from right recursion) in one macro, only changing the macro contents accordingly to the input progress.

It has shown up that nearly all known optimizations of LR techniques (e.g. elimination of unit productions, "stack controlling" LR) can be simulated by macros.

When deriving new algorithms via this macro technique by applying heuristics, we have a general problem: the number of "different" (in the sense of graph isomorphism) macros may become unbounded if care is not taken. The solution is to force a split whenever a macro exceeds a maximal size.

It is therefrom clear that there exist macroizations that are better than previously known algorithms at least with respect to the number of operations needed at run time. However, this may be at the expense of the space needed, since the number of macros may grow exponentially. On the other hand, the macroization may also be used to derive algorithms which are somewhere within the bandwith between LL and LR, with respect to macro size, "deterministicness", runtime behaviour, or other aspects. Other macroizations may extend far beyond the degree of determinism found in LR parsing, or may be incomparable to previously known algorithms.

Current research topics are to find good heuristics in the dilemma between space and time. Some of the questions which have to be solved are: where and when to split nodes for best ("optimal") results? Does there exist a globaly optimal macroization of arbitrary grammars for arbitrary input words (the author believes "no")? What are the deterministic grammar classes for the deterministic special case of various macroizations?

The overall lesson from this work is that any existing context-free parsing technique known to the author can be simulated by the sketched theory (and some minor enhancements not presented here), mostly relying on only two main principles: the compression theorem as a theoretical foundation of *dynamic programming*, and the macroization as a model for *precomputing*.

3 Further remarks

3.1 Example: deterministic parsing of some ambiguous grammars

This example sheds some light on the general usefulness of the macro construction:

Let G be an $LR(0)$ grammar. Construct from it a new grammar G' having the same productions, plus new ones: For each nonterminal A, add a new nontermimal $\bar{A}$ and productions $A \to \bar{A}$ and $\bar{A} \to A$. For each production $A \to X_1 X_2 \ldots X_l$, add a new production $\bar{A} \to \bar{X}_1 \bar{X}_2 \ldots \bar{X}_l$ where all nonterminals are in the barred form.

As is intuitively clear, this modified grammar is no longer an $LR(0)$ grammar. Since it is cyclic, the number of syntax trees is unbounded. Although the generated languages are the same for both grammars, we get lots of reduce-reduce conflicts in G': Any time an item $A \to X_1 \ldots X_l \cdot$ is complete, there exists also an item $\bar{A} \to \bar{X}_1 \ldots \bar{X}_l \cdot$ in the same $LR(0)$ state. It is therefore clear that deterministic parsing with classical $LR(0)$ is impossible.

However, our macro construction can do it deterministically: Just pack not only the LR sets of G, but also their corresponding barred forms into a macro, i.e. we simply keep the doubled items together with the $A \to \bar{A}$ and $\bar{A} \to A$ cycles in a macro.

This rather construed example can be applied analogously in practice. It shows that some "local" ambiguity can be handled by the macro construction. As long as ambiguous branches of the recognition tree are always at nearly the same depth, they can be packed into a macro and parsed deterministically. Also, noncanonical "bottom-up" parsing, i.e. making reductions inside a parse stack in a limited depth, can be simultated by macros just by keeping the limited part of the stack in one macro. This is further evidence for the generality of the macro method.

3.2 The problems in Tomita's algorithm

The above $LR(0)$ simulation by macros is described in [STin] as a special heuristics which allows the transformation of a shift-reduce tree generator , into an $LR(0)$ tree generator , $'$. When running , $'$ with the general algorithm described in [STin], all the problems of Tomita's algorithm [Tom86] (cyclic grammars, hidden left recursion, complexity worse than $O(n^3)$) are solved.

The interesting point is that the same algorithm that can simulate Earley parsing is also able to simulate nondeterministic LR (Tomita-like) parsing. One has only to feed a different tree generator into this algorithm.

3.3 Representation of parse trees

Another consequence from the macro construction is about the underlying micro graph: When simulating for example nondeterministic LR, the underlying micrograph is (modulo some splits) the same as in the above nondeterministic shift-reduce parsing which in turn is very similar to Earley parsing. Since the edges of the BMPL micro graph roughly correspond to Earley items (remember that an edge can be viewed as a direct replacement of the indirect Earley backpointer which goes indirectly through the input position), there is no need to represent parse forests as described by Tomita. This is because it is already known (see e.g. [AU72, Algorithm 4.6, page 328], some general issues also in [Ruz79]) that reconstructions of parse trees from Earley's structure are possible. As can be seen from a variant of Earley's structure presented in [STin], $O(n^2)$ space suffices to represent all parse trees in such a way that the reconstruction of one single tree requires only a constantly bounded number of instructions for one node of the tree, whereby the previously known algorithms may take more time.

3.4 Concluding remarks

A general theoretical framework for deriving new algorithms is presented in [STin]. For a number of questions, solutions are sketched there, while some others remain open.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison Wesley, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1. Prentice-Hall, 1972.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, Feb 1970.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[Ruz79] W. L. Ruzzo. On the complexity of general context-free language parsing and recognition. *Automata, Languages and Programming: Lecture Notes in Computer Science*, 71:489–497, 1979.

[STrt] Thomas Schöbel-Theuer. *Ein Ansatz für eine allgemeine Theorie kontextfreier Spracherkennung.* submitted dissertation in German, submitted 19. April 1996 to the Fakultät Informatik, Universität Stuttgart.

[STin] Thomas Schöbel-Theuer. *Eine allgemeine Theorie kontextfreier Spracherkennung.* Enlarged version of [STrt], unpublished manuscript in German, appearance is uncertain.

[Tom86] Masaru Tomita. *Efficient parsing for natural language.* Academic Publishers, Boston, 1986.