Theoretical Computer Science 12 (1980) 265–290

LOCAL CONSTRAINTS IN PROGRAMMING LANGUAGES PART I: SYNTAX*

Aravind K. JOSHI

Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, U.S.A.

Leon S. LEVY

Bell Telephone Laboratories, Whippany, NJ, U.S.A.

Kang YUEH

Bell Telephone Laboratories, Murray Hill, NJ, U.S.A.

Communicated by M. Nivat
Received May 1979
Revised October 1979

Abstract. The method of local constraints attempts to describe context-free languages in an apparently context-sensitive form which helps to retain the intuitive insights about the grammatical structure. This form of description, while apparently context-sensitive is, in fact, context-free and allows a program derivation structure to be represented as a tree with additional constraints, thus allowing for the possibility of a correctness proof in the form of Knuthian semantics. These semantic aspects will be discussed in a sequel to this paper (Part II: Semantics). Several detailed examples are given to motivate the use of local constraints grammars including some examples from the syntax of ALGOL 60. A parsing algorithm has been described; its purpose is to show that the computation of local constraints is quite reasonable. Transformation rules for transferring a context-free grammar into a local constraints grammar have been described and some heuristic approaches for the inverse transformation have been presented.

1. Introduction

The method of describing context-free languages in an apparently context-sensitive form, which helps to retain the intuitive insights about the grammatical structure, has linguistic origins and is referred to as local transformations [6]. We will use the term *local constraints* instead of local transformations in our present

* This part deals with syntactic concepts only. A sequel to this paper (Part II: Semantics) will deal with certain semantic aspects. This work was partially supported by NSF Grant MCS76-19466, MCS77-04834, MCS78-04801, MCS79-08401.

This paper is a substantially extended and revised version of an earlier working paper presented at the 5th Annual Symposium on Principles of Programming Languages (POPL), Tucson, AZ, January 1978.

context.[1] This form of description, while apparently context-sensitive is, in fact, context-free and allows a program derivation structure to be represented as a tree with additional constraints, thus allowing for the possibility of a correctness proof in the form of Knuthian semantics, as described in [4].[2]

One might wonder why the ubiquitous BNF style of grammar is not adequate for our purposes, since we are only considering sets which can be described by BNF grammars.[3] There are two aspects of BNF definition that are unsatisfactory – the first having to do with the construction of the grammar itself, and the second having to do with the parse trees produced by the grammar. In a BNF grammar, the only means of controlling the structure of the derivation trees are the syntactic categories themselves. Thus precedence and associativity rules must be encoded into the grammar through the use of additional nonterminals designating syntactic categories created solely for the purpose of assuring proper phrase structure. This is very clearly seen in the case of arithmetic expressions, where the usual BNF grammar (G_1 of Section 3.1, below) has the nonterminals E, T, and F to enforce left associativity and precedence of $*$ over $+$. While the precedence and associativity are well established mathematical conventions, the syntactic categories of expressions, terms, and factors are strictly ad hoc. Moreover, once the BNF grammar has been constructed, the additional syntactic categories lead to semantically vacuous chains in the derivation tree, yielding an inefficient representation of phrase structure with subsequent inefficiencies in passing semantic attributes up and down these non-terminal chains.[4]

In Section 2, some known definitions and results about local constraints are reviewed. In Section 3, a few examples are given to motivate the subsequent sections. A parsing algorithm is given in Section 4. The main purpose of this section is to show that the computation of the local constraints is quite reasonable, a fact which is not intuitively obvious. In Section 5 we first give some detailed examples to show that some portions of ALGOL 60 syntax can be restated as a local constraints grammar. A set of transformation rules for transforming a context-free grammar into a local constraints grammar is given in Section 5.1; a method for effecting a similar transformation based on the notion of skeletons is presented in Section 5.2. Although a tree automata theoretic approach for transforming a local constraints grammar into a context-free grammar exists in principle, this procedure is extremely unwieldy; therefore, it is useful to find some heuristic approaches which work for some special and useful cases. One such approach has been described in Section 5.3. The relevance of local constraints grammars to semantics, especially their relation to Knuthian semantics, will be discussed in a sequel to this paper.

[1] The term local transformations can also be justified (see Section 3 for further comments).

[2] These semantic aspects will be discussed in a sequel to this paper (Part II: Semantics).

[3] See, e.g., [9] for a discussion of non-context-free grammars.

[4] For further discussion of the chain problem see [5].

2. Background

2.1. Definition of local constraints

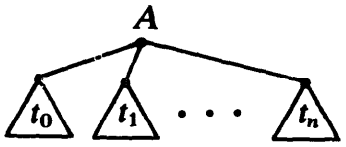
Context-sensitive grammars, in general, are more powerful (with respect to weak generative capacity) than context-free grammars. A fascinating result of Peters and Ritchie [10] is that if a context-sensitive grammar G is used for 'analysis', then the language 'analyzed' by G is context-free.

First, we describe what we mean by a context-sensitive grammar, G, used for 'analysis'. Given a tree t, we define a set of proper analyses of t. Roughly speaking, a proper analysis of a tree is a slice across the tree. More precisely, the following recursive definition applies:

Definition 2.1. The set of proper analyses of a tree t, denoted $P(t)$, is defined as follows:

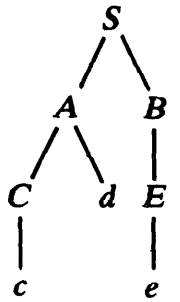
(i) if $t = \emptyset$ (the empty tree), then $P(t) = \emptyset$,

(ii) if $t =$



then $P(t) = \{A\} \cup P(t_0) \cdot P(t_1) \cdot \cdots \cdot P(t_n)$, where $t_0, t_1, \ldots, t_n$ are trees, and '$\cdot$' denotes concatenation (of sets).

Example 2.1.



$P(t) = \{S, AB, AE, Ae, CdB, CdE, Cde, cdB, cdE, cde\}$.

Let G be a context-sensitive grammar, i.e. its rules are of the form[5]

$$A \to \omega / \phi\_\Psi,$$

where $A \in V - \Sigma$ (V is the alphabet and Σ is the set of terminal symbols), $\omega \in V^+$ (set of non-null strings on V) and ϕ, $\Psi \in V^*$ (set of all strings on V). If ϕ and Ψ are both null, then the rule is a context-free rule. A tree t is said to be 'analyzable' with respect to G if for each node of t, some rule of G 'holds'. It is obvious how to check whether a

[5] $A \to \omega / \phi\_\Psi$ is just another way of writing $\phi A \Psi \to \phi \omega \Psi$.

context-free rule holds of a node or not. A context-sensitive rule $A \to \omega/\phi\_\Psi$ holds of a node labeled A, if the string corresponding to the descendants of that node is ω, and there is a proper analysis of t of the form $\rho_1\phi A\Psi\rho_2$ which 'passes through' the nodes ($\rho_1, \rho_2 \in V^*$). We call the contextual condition ϕ_Ψ, a proper analysis predicate.

Similar to these context-sensitive rules, which allows us to specify context on the 'right' and 'left', we often need rules to specify context on the 'top' or 'bottom'. Given a node labeled A in a tree t, we say that $\delta(A, \phi\_\Psi)$, $\phi, \Psi \in V^*$, holds of a node labeled A if there is a path from the root of the tree to the frontier, which passes through the node labeled A, and is of the form

$$\rho_1\phi A\Psi\rho_2, \quad \rho_1, \rho_2 \in V^*.$$

The contextual condition associated with such a 'vertical' proper analysis is called a *domination predicate.*

The general form of local constraint combines the proper analysis and domination predicates as follows:

Definition 2.2. A *local constraint* is a rule of the form

$$A \to \omega/C_A,$$

where C_A is a Boolean combination of proper analysis and domination predicates.

In transformational linguistics the context-sensitive and domination predicates are used to describe conditions on transformations, hence we have referred to these local constraints elsewhere as local transformations [6].

2.2. Results on local constraints

Theorem 2.1 [6]. *Let G be a finite set of local constraints and $\tau(G)$ the set of trees analyzable by G. Then the string language $L(\tau(G)) = \{x \mid x \text{ is the yield of } t \text{ and } t \in \tau(G)\}$ is context-free.*

Example 2.2. Let $V = \{S, T, a, b, c, e\}$ and $\Sigma = \{a, b, c, e\}$, and G be a finite set of local constraints:

1. $S \to e$
2. $S \to aT$
3. $T \to aS$
4. $S \to bTc/(a\_) \wedge \delta(S, T\_)$
5. $T \to bSc/(a\_) \wedge \delta(T, S\_)$.

In rules 1, 2, and 3 the context is null, and these rules are context-free. In rule 4 (and in rule 5) the constraint requires an 'a' on the left, and the node dominated (immediately) by a T (and by an S in rule 5).

The language generated by G can be derived by G_1:

$$S \to e \qquad S \to aT$$
$$S \to aT \qquad T \to aS_1$$
$$T \to aS \qquad T_1 \to bSc.$$
$$S_1 \to bTc$$

In G_1 there are additional variables S_1 and T_1 which enable the context checking of the local constraints grammar, G, in the generation process.

It is easy to see that under the homomorphism which removes subscripts on the variables T_1 and S_1, each tree generable in G_1 is analyzable in G. Also, each tree analyzable in G has a homomorphic pre-image in G_1.

The methods used in the proof of the theorem use tree automata to check the local constraints predicates, since tree automata used as recognizers accept only tree sets whose yield languages are context-free.

We now give an informal introduction to the ideas of (bottom-up) tree automata.[6] Tree automata process labeled trees, where there is a left-to-right ordering on the successors of a node in the tree. When all the successors of a node ν have been assigned states, then a state is assigned to ν by a rule which depends on the label of ν and the states of the successors of ν considering their left-to-right ordering. Note that the automaton may immediately assign states to the nodes on the frontier of the tree since these nodes have no successors. If the set of states is partitioned into final and non-final states, then a tree is accepted by the automaton if the state assigned to the root is a final state. A set of trees accepted by a tree automaton is called a recognizable set. Note that the automaton may operate non-deterministically, in which case, as usual, a tree is accepted if there is some set of state assignments leading to its acceptance.

The importance of tree automata is that they are related to the sets of derivation trees of context-free grammars. Specifically, if T is the set of derivation trees of a context-free grammar, G, then there is a tree automaton which recognizes T. Conversely if T is the set of trees recognized by a tree automaton, A, then T may be systematically relabeled as the set of derivation trees of a context-free grammar.

The basic idea presented in detail in [6] is that because tree automata have nice closure properties, they can do the computations required to check the local constraints. Consequently, the local constraints could be encoded in BNF grammars.

The reader is referred to [6], where significant generalizations of Theorem 2.1 are presented.

A further important idea is the skeletal automaton developed in [8]. A skeleton is a tree in which only the frontier nodes are labeled, and a skeletal automaton is an automaton which processes skeletons, assigning a state to a node ν based on the

[6] For a more complete discussion see [14] or [7].

states of the successors of ν. Thus, the skeletal automaton can process structured strings in which phase structuring has been developed, but where no syntactic categories have been assigned. These skeletal automata characterize the set of context-free derivation trees, and, because they start with phrase structuring only, can be used to discover context-free and local constraints grammars.

3. Local constraints in syntax

We will give here a few simple examples. Additional examples appear in Section 5.

3.1. Examples

Example 3.1. Local constraints can be used to simplify the syntax of context-free grammars, for example, the most commonly used grammar for the class of simple arithmetic expressions is

G_1: $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to a \mid (E).$

G_1' below is an equivalent grammar with local constraints. Note that G_1' gives unambiguous structural descriptions, in accordance with the precedence relationships, for the arithmetic expressions.

G_1': $E \to E+E/(\text{not } +\_) \wedge (\text{not } *\_) \wedge (\text{not}\_*)$

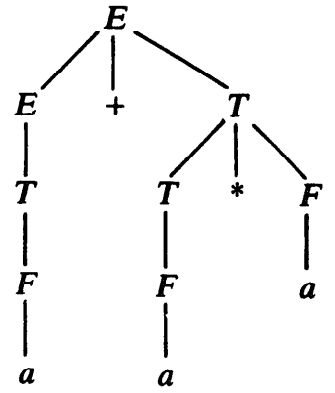
$E \to E*E/(\text{not } *\_)$

$E \to (E)$

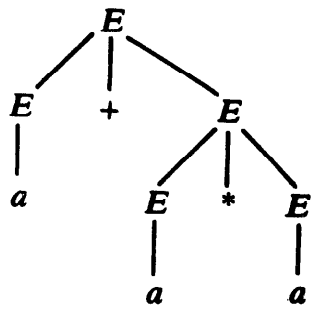
$E \to a.$

Note that in the local constraints grammar G_1' the additional non-terminals T, F of G_1 do not occur, and, further, the derivation trees in G_1' do not exhibit the chains of productions that derivation trees in G_1 exhibit We show below the respective derivation trees in G_1 and G_1' of $a+a*a$:

in G_1:



in G_1':



Example 3.2. Another example is a grammar for expressions in a typed grammar. The usual way of representing this is a set of productions of the form

$$S \to E_t;$$

$$E_t \to E_t \theta E_t \quad \text{for each type}$$

$$E_t \to a_t.$$

An equivalent local constraints grammar is as follows:

$$S \to E$$

$$E \to E \theta E$$

$$E \to a_t \mid -\theta a_t \vee -;$$

Local constraints can be used to specify certain types of syntactic concord in linguistics such as the agreement in number of subject and verb, or lexical constraints like the restriction to animate or human subject. For example, consider the following local constraint grammar [13]:

$\langle S \rangle \to \langle NP \rangle \langle VP \rangle$

$\langle NP \rangle \to \langle DET \rangle \langle NP - SNG \rangle / - \langle VP - SNG \rangle$

$\langle NP \rangle \to \langle DET \rangle \langle NP - PL \rangle / - \langle VP - PL \rangle$

$\langle VP \rangle \to \langle VP - SNG \rangle$

$\langle VP \rangle \to \langle VP - PL \rangle$

$\langle DET \rangle \to THE$

$\langle NP - SNG \rangle \to DOG$

$\langle NP - PL \rangle \to DOGS$

$\langle VP - SNG \rangle \to BARKS$

$\langle VP - PL \rangle \to BARK.$

This grammar generates sentences like

THE DOG BARKS, THE DOGS BARK

with the subject and verb agreement in number.

3.2. Efficiency of representation

It is clear that since grammars with local constraints can only describe recognizable sets, that claims about the descriptive power of such grammars will, in general, appeal to their naturalness or understandability. While such considerations are quite important, they are also partly subjective. However, the following objectives claim

can be made: Local constraints allow more efficient representations of the grammars of context-free languages.

Since every context-free grammar is, by definition, also a local constraints grammar – with vacuous constraints, it is clear that the local constraints grammars are at least as efficient in representing context-free languages. To show that local constraints are more efficient, it suffices to show that for any specified constraint, there is a context-free language, L_q, such that the context-free grammar for L_q requires at least $\log q$ times the storage of the local constraints grammar for L_q. We define $L_q = \Sigma \cdot \{w\}$, where $w = a_1 a_2 \cdots a_q$ is a word of length a. Then L_q is a q-definite event. The local constraints grammar for L_q is

$$G_1: \quad (1)\ X \to \sigma X, \quad \sigma \in \Sigma$$

$$(2)\ X \to A_q / a_1 a_2 \cdots a_{q-1}.$$

There are $|\Sigma|$ productions of the first kind, and one production of the second kind. Hence, the storage required for G_1 is $k_1|\Sigma| + K_2 q$. Assuming that $|\Sigma| \ll q$, G_1 requires $O(q)$ storage.

Since L_q is q definite, it requires a q-state minimal machine with $|\Sigma|$ productions of the form $X_i \to X_j$ for each state, plus one production of the form $X_{q-1} \to a_q$. Since there are q-states, the representation of each state will require $O(\log q)$ and the regular grammar, G_2 for L_q requires $O(\log q)$ and the regular grammar, G_2 for L_q requires $O(q \log q)$ storage. Hence we have shown

Theorem 3.1. *For any constant k, there are languages whose local constraint grammar requires less than* $1/k$*th the storage of a context-free grammar generating the same structural descriptions.*

4. Parsing of grammars with local constraints

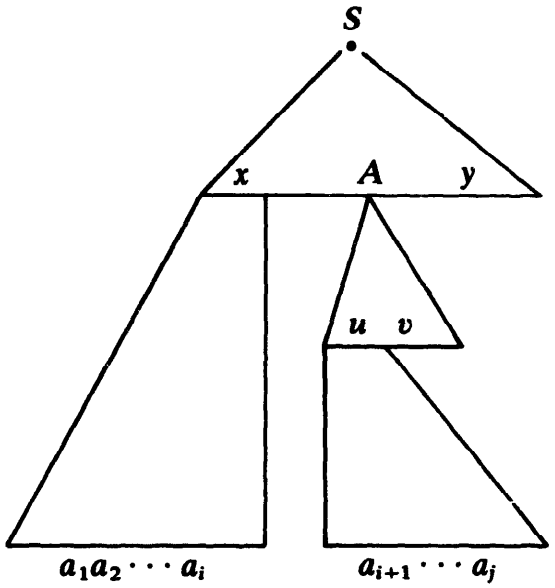
In this section we will present an algorithm for parsing local constraints grammars. The parser will check the constraints when the derivation trees are built and discard those trees which do not satisfy the constraints, so only the correct derivation trees which are built so far need to be kept. In this manner, both the time and space requirements will be cut down significantly. In order to check the constraints, the parsing algorithm should reflect the derivation trees explicitly, and Earley's algorithm [3] seems to fit that purpose.[7]

[7] We realize that Earley's algorithm is not the most practical one used. Our purpose in showing that Earley's algorithm can be extended to local constraints grammars is to show that the computation of these constraints is quite reasonable, a fact which is not intuitively obvious.

4.1. Parsing algorithm

In Earley's algorithm for an input string $w = a_1 a_2 \cdots a_n$, the dotted rule $[A \to u \cdot v, i]$ is in list I_j for $0 \leq i \leq j$ if and only if $A \to uv$ is a production in $G = (V, T, P, S)$ and there exist x and y in V^* such that $S \Rightarrow^* xAy$, $x \Rightarrow^* a_1 a_2 \cdots a_i$ and $u \Rightarrow^* a_{i+1} a_{i+2} \cdots a_j$.

The above statement can be better understood by the following pictorial representation, where the derivation is represented in the form of the corresponding syntax tree:



Intuitively, if $[A \to u \cdot v, i]$ is in I_j, then we are working on a potentially valid parse in that there is a sentential form xAy, where $x \Rightarrow^* a_1 a_2 \cdots a_i$ and the dotted rule $[A \to u \cdot v, i]$ indicates that $u \Rightarrow^* a_{i+1} \cdots a_j$. We know nothing yet about v or y.

To extend Earley's algorithm to check for local constraints, it can be seen from the above diagram that the left portion of constraints can be checked with the parsed portion x, but the right portion of constraints has to be carried along with the dotted rule in order to be checked later with the unknown portion of y. For the domination predicates, both the top and bottom portions can be checked with the parse tree when $[A \to uv \cdot, i]$ is generated.

In the first step, we have to separate the proper analysis predicates into left and right portions by the following rules:

(1) $A \to t/\text{not } (u\_v)$ is changed to $A \to t/(\text{not } u\_) \vee (\_\text{not } v)$,

(2) $A \to t/(u\_v) \wedge (x\_y)$ is changed to $A \to t/(u \wedge x)\_(v \wedge y)$,

(3) $A \to t/(u\_v) \vee (x\_y)$ is changed to $A \to t/(u\_v)$ and $A \to t/(x\_y)$.

The domination predicates will be left unchanged. After this separation, the proper analysis predicates are of the form

$$(u_1 \wedge u_2 \wedge \cdots \wedge u_k)\_(v_1 \wedge v_2 \wedge \cdots \wedge v_m),$$

where u and v are strings over V^* or negation of them.

For a given grammar $G = (V, T, P, S)$ with local constraints and an input string $w = a_1 a_2 \cdots a_n$ in T^*, the parser scans the input string from left to right. As each symbol a_j is scanned, a list of item I_j is constructed which represents the condition of the recognition process at that point in the scan. Each item in the list represents:

(1) a production such that we are currently scanning an instance of its right side,

(2) a dot in the production which indicates how much of the production we have scanned,

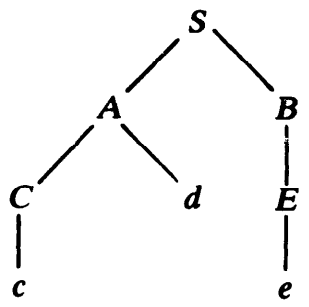
(3) a pointer back to the position in the input string at which we began to look for that instance of the production,

(4) a forest representation which shows how the scanned portion in the production derives the terminal string,

(5) a set of strings which shows the constraints remaining to be checked with the unscanned portion of the production.

The item $[A \to u(s) \cdot v(t), i]$ is in list I_j for $0 \leq i \leq j$ if and only if $A \to uv/C_A$ is a production in P, and for some x and y we have $S \Rightarrow^* xAy$, $x \Rightarrow^* a_1 a_2 \cdots a_i$, $u \Rightarrow^* a_{i+1} \cdots a_j$, s is a forest representation of u, and t is a set of strings remaining to be checked with the set of proper analyses of v. The forest representation is for convenience of exposition. The implementation does not have to proceed in this way. (See later remarks on checking for the membership of a proper analysis.)

Definition 4.1. A tree is represented by its *bracket expression*, for example,



is represented as $S(A(C(c), d), B(E(e)))$.

Definition 4.2. A *forest* is a sequence of trees and is represented as the concatenation of their tree representations.

The central part of the parsing algorithm is the generation of the sequence of lists $I_0, I_1, \ldots, I_n$. There are three operations – predictor operation, scanner operation, and completer operation – which work on the items in a list I_j. The left portion of proper analysis predicates is checked during the predictor and completer operations, the right portion of proper analysis predicates is checked during the scanner and completer operations, and the domination predicates are checked during the completer operation. The detail of the operations will be explained later.

First, we construct I_0 as follows:

(1) if $S \to r/u\_v$ is a production in P, then add $[S \to \cdot r, 0]$ to I_0 if $u = e$, where e is the empty string,

(2) perform predictor and completer operations on I_0 until no new items can be added to I_0.

Then construct I_j for $1 \leqslant j \leqslant n$ as follows:

(3) perform scanner operation on I_{j-1},

(4) perform predictor and completer operations on I_j until no new items can be added to I_j.

A string $w = a_1 a_2 \cdots a_n$ is in $L(G)$ if and only if after constructing I_j for $0 \leqslant j \leqslant n$, there exist some items of the form $[S \to t(x) \cdot (y), 0]$ in I_n, where y is either an empty string or some negative constraints. The parse for w is $S(x)$.

We will consider only the case where the constraints are of Peters–Ritchie's type, that is, they are of the form u_v, where $u, v \in V^*$. The case of general constraints is considered in [15].

The three operations which work on the items in a list I_j are:

(1) *Predictor operation* is applicable to an item when there is a non-terminal to the right of the dot. Suppose that $[A \to u(x) \cdot Bv(y), i]$ is an item in I_j, then for all productions in P of the form $B \to r/s\_t$, if the left constraint s is satisfied such that LEFT$(s, [A \to u(x) \cdot Bv(y), i])$ is TRUE, we then add $[B \to \cdot r, j]$ to I_j. The checking of left constraints is a recursive call as follows:

Procedure LEFT$(s, [A \to u(x) \cdot Bv(y), i])$
begin if $s \in$ Suffix(PA(x)), where PA(x) is the set of
proper analyses of x, **then** TRUE
else if $s = zq$ such that $q \in$ PA(x) and there exists
an item $[D \to u'(x') \cdot Av'(y'), k]$ in I_i, and
LEFT$(z, D\tau u'(x') \cdot Av'(y'), k])$ is TRUE **then** TRUE
else FALSE
end

(2) *Scanner operation* is applicable to an item when there is a terminal to the right of the dot. Suppose that $[A \to u(x) \cdot av(y), i]$ is an item in I_{j-1}. If $a = a_j$ and INIT(a, y) is TRUE, then we add $[A \to ua(x, a) \cdot v(z), i]$ to I_j, where $z =$ DEL(a, y) is the remainder of y after deleting prefix a.

The checking of a against the prefix of y is defined as:

Procedure INIT(a, y)
Comment: y may be of the form $y_1 \wedge y_2 \wedge \cdots$
begin if $y = y_1 \wedge y_2$ **then** INIT$(a, y_1) \wedge$ INIT(a, y_2)
else if $y = e$, where e is the empty string, or
$y = az$ for some $z \in V^*$ **then** TRUE
else FALSE
end

The deleting of a from the prefix of y is defined as:

Procedure DEL(a, y)
Comment: y may be of the form $y_1 \wedge y_2 \wedge \cdots$
begin if $y = y_1 \wedge y_2$ **then** DEL(a, y_1) $\wedge$ DEL(a, y_2)
else if $y = az$ for some $z \in V^*$ **then** z
else e
end

Note here that we mix the interpretation of '$\wedge$', sometimes we want to execute the Boolean '$\wedge$' operation to yield a TRUE/FALSE value, and sometimes we use '$\wedge$' to form the set of strings; but it should be clear which was '$\wedge$' is used in a given context.

(3) *Completer operation* is applicable to an item if the dot is at the end of its production. Suppose that $[B \rightarrow r(x) \cdot (y), i]$ is an item in I_j and $B \rightarrow r/s\_t$ is a production in P. Examine the list I_i for items of the form $[A \rightarrow u(g) \cdot Bv(h), k]$. For each one found, if LEFT(s, $[A \rightarrow u(g) \cdot Bv(h), k]$) and PAMATCH($x$, h) are TRUE, then we add $[A \rightarrow uB(g, B(x)) \cdot v(t \wedge y \wedge z), k]$ to I_j, where z = PADEL(x, h).

The checking of parsed portion x against unmatched constraints y is defined as:

Procedure PAMATCH(x, y)
Comment: y may be of the form $y_1 \wedge y_2 \wedge \cdots$
x is a forest representation
begin if $y = y_1 \wedge y_2$ **then** PAMATCH(x, y_1) $\wedge$ PAMATCH(x, y_2)
else if $y \in$ Prefix(PA(x)) **then** TRUE
else if $y = zt$ and $z \in$ PA(x) **then** TRUE
else FALSE
end

The deleting of strings in the set of proper analyses of x from the prefix of y is defined as:

Procedure PADEL(x, y)
Comment: y may be of the form $y_1 \wedge y_2 \wedge \cdots$
x is a forest representation
begin if $y = y_1 \wedge y_2$ **then** PADEL(x, y_1) $\wedge$ PADEL(x, y_2)
else if $y \in$ Prefix(PA(x)) **then** e
else if $y = zt$ and $z \in$ PA(x) **then** t
end

Example 4.1. Let the grammar be

G: $E \rightarrow E + E/(\text{not } +\_) \wedge (\text{not } *\_) \wedge (\text{not } \_*)$

$E \rightarrow E * E/(\text{not } *\_)$

$E \rightarrow a$.

The grammar after the constraints are separated is

$$G': \quad E \to E + E/(((\text{not } +) \wedge (\text{not } *))\_\text{not } *)$$
$$E \to E * E/(\text{not } *\_)$$
$$E \to a.$$

For an input string $w = a + a * a$, the parse lists are generated as:

$$I_0: \quad [E \to \cdot E + E, 0]$$
$$[E \to \cdot E * E, 0]$$
$$[E \to \cdot a, 0],$$

$$I_1: \quad [E \to a(a)\cdot, 0]$$
$$[E \to E(E(a))\cdot + E, 0]$$
$$[E \to E(E(a))\cdot * E, 0],$$

$$I_2: \quad [E \to E + (E(a), +)\cdot E, 0]$$
$$X[E \to \cdot E + E, 2]$$
$$[E \to \cdot E * E, 2]$$
$$[E \to \cdot a, 2].$$

The item $[E \to \cdot E + E, 2]$ is deleted because the left constraint is not satisfied in the predictor operation;

$$I_3: \quad [E \to a(a)\cdot, 2]$$
$$[E \to E + E(E(a), +, E(a))\cdot, 0]$$
$$[E \to E(E(a))\cdot * E, 2]$$
$$[E \to E(E(E(a), +, E(a)))\cdot + E(\text{not } *), 0]$$
$$X[E \to E(E(E(a), +, E(a)))\cdot * E(\text{not } *), 0],$$

$$I_4: \quad [E \to E * (E(a), *)\cdot E, 2]$$
$$X[E \to \cdot E + E, 4]$$
$$X[E \to \cdot E * E, 4]$$
$$[E \to \cdot a, 4].$$

The last item in I_3 does not satisfy the constraint in the scanner operation, so it is not carried on to I_4. Also the middle two items in I_4 are deleted because they do not satisfy the left constraint in the predictor operation;

I_5: $[E \to a(a)\cdot, 4]$

$[E \to E*E(E(a), *, E(a)), 2]$

$[E \to E+E(E(a), +, E(E(a), *, E(a))), 0]$

$[E \to E(E(E(a), *, E(a)))\cdot * E, 2]$

$[E \to E(E(E(a), +, E(E(a), *, E(a))))\cdot + E(\text{not } *), 0]$

$X[E \to E(E(E(a), +, E(E(a), *, E(a))))\cdot * E(\text{not } *), 0].$

The string $w = a+a*a$ is accepted because the item $[E \to E+E(E(a), +, E(E(a), *, E(a))), 0]$ is in I_5. The parse tree for $a+a*a$ is $E(E(a), +, E(E(a), *, E(a)))$.

4.2. Correctness and running time of the parsing algorithm

The correctness of the parsing algorithm for grammars with local constraints can be proved in two steps. First the correctness of Earley's algorithm has been proved in [1, 3], we only need to show that the checking of constraints is correct and here we will give an informal proof of that.

In the parsing algorithm presented, the domination predicates and the left portion of proper analysis predicates are checked during the completer operation, and the right portion of proper analysis predicates is checked during the scanner and completer operations. From the pictorial representation of Earley's algorithm, we can see that when $[A \to uv\cdot, i]$ is an item in list I_j, all the top and bottom portions of the parse tree with respect to the production $A \to uv$ are already parsed, and the left portion is parsed when $[A \to \cdot uv, i]$ is added to list I_i. So we are able to check both the domination predicates and the left portion of constraints during the completer operation. We can even check the left portion of constraints during the predictor operation to eliminate some items that do not satisfy the left constraints before adding $[A \to \cdot uv, i]$ to I_i. However, we still have to check the left portion of constraints again during the completer operation to prevent cases in which an item is added based on some right constraints that remain to be checked, and later the item that matches the right constraints is added based on some different left constraints. Such case may arise because the checking of left and right portions of constraints are separated in the parsing algorithm.

The right portion of constraints is carried along with each item and is checked off during both scanner and completer operations. An item $[A \to u(x)\cdot av(y), i]$ in list I_{j-1} will be added to list I_j only if the terminal a_j in the input string matches with both a and the first character in y. The constraint z carried in $[A \to ua(x, a)\cdot v(z), i]$ is y with first character a_j deleted. Now z stands for the constraints remaining to be checked with the parse of v.

For an item $[B \to r(x) \cdot (y), i]$ considered in the completer operation, we look back in list I_i for items of the form $[A \to u(g) \cdot Bv(h), k]$, here h is the set of constraints remaining to be checked with the parse of Bv. Now $B \to r$ is successfully parsed with parse tree x. Before adding the item $[A \to uB \cdot v, k]$, we have to check that the proper analyses of parse tree x match with h. The constraints carried in the new item will be h with the matched portion deleted, in addition, the constraints will include y, because y in $[B \to r(x) \cdot (y), i]$ represents the constraints remaining to be checked with the portion to the right of $B \to r$, and include the right constraint t from the production $B \to r/s\_t$.

Although the number of proper analyses of trees grows exponentially, Buneman and Levy [2] have shown that the time for checking the membership of proper analyses is proportional to e^3 where e is the number of edges in the given tree.

The process of checking that a string is a proper analysis of a tree consists first of constructing a non-deterministic finite state automaton whose transitions correspond to the edges of the tree and which accepts just those strings that are proper analyses. The labels can be associated with edges rather than with nodes in a tree by moving the label for every node other than the root up to the edge which is directed at that node. For the root, we add a new node with one edge directed to the root and transfer the label from the root to that edge. It is then just a simple matter to check that a string is accepted by the NDFA (non-deterministic finite automaton). The running time for that is proportional to e^3, where e is the number of transitions in the NDFA (or the number of edges in the tree).

In the parsing algorith presented, the checking of local constraints predicates involves mainly the checking of membership of proper analyses. Suppose that m is the maximal length of constraints in the grammar, then in checking for the membership of proper analyses, we can construct a NDFA from the tree with no path longer than m. Now the number of transitions in the NDFA is proportional to m, and the time for checking the membership of proper analyses is proportional to m^3, which is independent of the length of the input string.

Since the parsing time of Earley's algorithm with no constraints checking is proportional to n^3 and the checking of constraints is independent of n, the running time of parsing algorithm for grammars with local constraints is proportional to n^3.

It should be noted that the above discussion shows that the parsing time is $O(n^3)$ when the local constraints grammar is unambiguous. If the grammar is ambiguous then the algorithm presented does not guarantee that the parsing time will be $O(n^3)$. It will be, however, polynomial in the length of the input string due to the fact that the growth of the I lists is at most $O(n^2)$ and the time for checking for membership of a proper analysis is $O(n^3)$. Hence, the parsing time will be $O(n^6)$. The fact that the only paths in the NDFA that need to be checked are no longer than m^3, where m is the maximum constraint length, suggests that it may be possible to show that the time is $O(n^3)$ even in the ambiguous case, but it is an open question; the proposed algorithm as it is does not show this.

5. Transformations between context-free grammars and local constraints grammars

The languages described by local constraints grammars are context-free. Hence, for every local constraints grammar there exists an equivalent context-free grammar. For every context-free grammar there is, of course, a trivially equivalent local constraints grammar, namely, the original grammar itself. We will consider now some non-trivially equivalent local constraints grammars. The most direct way to transform a given context-free grammar to local constraints grammar is by 'massaging' the context-free grammar and using some transformation rules to eliminate some non-terminals and introducing local constraints to ensure that the language generated is still the same.

5.1. Direct transformations

5.1.1. Example

First we will show in more detail how a context-free grammar can be transformed to a local constraints grammar by direct transformations. We take an example from ALGOL 60 syntax.

The definition of numbers in ALGOL 60 is:

⟨unsigned integer⟩ ::= ⟨digit⟩ |
  ⟨unsigned integer⟩⟨digit⟩
⟨integer⟩ ::= ⟨unsigned integer⟩ |
  +⟨unsigned integer⟩ |
  −⟨unsigned integer⟩
⟨decimal fraction⟩ ::= ·⟨unsigned integer⟩
⟨exponent part⟩ ::= 10⟨integer⟩
⟨decimal number⟩ ::= ⟨unsigned integer⟩ |
  ⟨decimal fraction⟩ |
  ⟨unsigned integer⟩⟨decimal fraction⟩
⟨unsigned number⟩ ::= ⟨decimal number⟩ |
  ⟨exponent part⟩ |
  ⟨decimal number⟩⟨exponent part⟩
⟨number⟩ ::= ⟨unsigned number⟩ |
  +⟨unsigned number⟩ |
  −⟨unsigned number⟩

In the first step, we can eliminate ⟨decimal fraction⟩ and ⟨exponent part⟩. This is just a substitution. The grammar obtained is equivalent to the original one.

⟨unsigned integer⟩ ::= ⟨digit⟩ |
  ⟨unsigned integer⟩⟨digit⟩
⟨integer⟩ ::= ⟨unsigned integer⟩ |
  +⟨unsigned integer⟩ |
  −⟨unsigned integer⟩

⟨decimal number⟩::=⟨unsigned integer⟩|
·⟨unsigned integer⟩|
⟨unsigned integer⟩·⟨unsigned integer⟩
⟨unsigned number⟩::=⟨decimal number⟩|
10⟨integer⟩|
⟨decimal number⟩10⟨integer⟩
⟨number⟩::=⟨unsigned number⟩|
+⟨unsigned number⟩|
−⟨unsigned number⟩

Next, we eliminate ⟨decimal number⟩. The grammar becomes:

⟨unsigned integer⟩::=⟨digit⟩|
⟨unsigned integer⟩⟨digit⟩
⟨integer⟩::=⟨unsigned integer⟩|
+⟨unsigned integer⟩|
−⟨unsigned integer⟩
⟨unsigned number⟩::=⟨unsigned integer⟩|
·⟨unsigned integer⟩|
⟨unsigned integer⟩·⟨unsigned integer⟩|
10⟨integer⟩/_not 10|
⟨unsigned number⟩10⟨integer⟩/_not 10
⟨number⟩::=⟨unsigned number⟩|
+⟨unsigned number⟩|
−⟨unsigned number⟩

Next, we eliminate ⟨unsigned number⟩. The grammar becomes:

⟨unsigned integer⟩::=⟨digit⟩|
⟨unsigned integer⟩⟨digit⟩
⟨integer⟩::=⟨unsigned integer⟩|
+⟨unsigned integer⟩|
−⟨unsigned integer⟩
⟨number⟩::=⟨unsigned integer⟩|
·⟨unsigned integer⟩|
⟨unsigned integer⟩·⟨unsigned integer⟩|
10⟨integer⟩/_not 10|
⟨number⟩10⟨integer⟩/_not 10|
+⟨number⟩/not (+, −)_not 10|
−⟨number⟩/not (+, −)_not 10

(the constraint 'not (+, −)_' means that neither '+' nor '−' can appear on the left).

Finally, we eliminate ⟨integer⟩. The grammar becomes:

⟨unsigned integer⟩::=⟨digit⟩|
⟨unsigned integer⟩⟨digit⟩

⟨number⟩::=⟨unsigned integer⟩ |
·⟨unsigned integer⟩/not 10_ |
⟨unsigned integer⟩·⟨unsigned integer⟩/not 10_ |
+⟨unsigned integer⟩/10_ |
−⟨unsigned integer⟩/10_ |
10⟨number⟩/not 10_not 10 |
⟨number⟩10⟨number⟩/not 10_not 10 |
+⟨number⟩/not (10, +, −)_not 10 |
−⟨number⟩/not (10, +, −)_not 10

By eliminating the rules +⟨unsigned integer⟩ and −⟨unsigned integer⟩, we get:

⟨unsigned integer⟩::−⟨digit⟩ |
⟨unsigned integer⟩⟨digit⟩
⟨number⟩::=⟨unsigned integer⟩ |
·⟨unsigned integer⟩/not (10, 10+, 10−)_ |
⟨unsigned integer⟩·⟨unsigned integer⟩/not (10, 10+, 10−)_ |
10⟨number⟩/not (10, 10+, 10−)_not 10 |
⟨number⟩10⟨number⟩/not (10, 10+, 10−)_not 10 |
+⟨number⟩/not (+, −)_not 10 |
−⟨number⟩/not (+, −)_not 10

We can also eliminate ⟨unsigned integer⟩ or even ⟨digit⟩ to leave only one syntactic category ⟨number⟩ for numbers. The grammar after eliminating ⟨unsigned integer⟩ is:

⟨number⟩::=⟨digit⟩ |
⟨number⟩⟨digit⟩ |
·⟨number⟩/not (10, 10+, 10−, ·)_not (·, ⟨digit⟩) |
⟨number⟩·⟨number⟩/not (10, 10+, 10−, ·)_not (·, ⟨digit⟩) |
10⟨number⟩/not (10, 10+, 10−, ·)_not (·, ⟨digit⟩, 10) |
⟨number⟩10⟨number⟩/not (10, 10+, 10−, ·)_not (·, ⟨digit⟩, 10) |
+⟨number⟩/not (+, −, ·)_not (·, ⟨digit⟩, 10) |
−⟨number⟩/not (+, −, ·)_not (·, ⟨digit⟩, 10)

5.1.2. Transformation rules

We will present a set of transformation rules which can be used to transform a given context-free grammar to an equivalent grammar with local constraints. These transformation rules not only keep the language of the transformed grammar equivalent to the original grammar, but also keep the derivation trees similar:

(1) *Forward substitution*

$A \to w_1 | \cdots | w_m, \quad B \notin w_i$

$A \to u_1 B v_1 | \cdots | u_p B v_p$

$B \to z_1 | \cdots | z_n, \quad B \notin z_i.$

They are transformed to

$$A \to w_1 | \cdots | w_m$$

$$A \to u_1 z_i v_1 | \cdots | u_p z_i v_p, \quad 1 \leqslant i \leqslant n.$$

This transformation is used when $p \times n$ is small, otherwise, rule (3) is used instead to minimize the total number of symbols used in the grammar.

This transformation just replaces non-terminal B by the right side strings it derives. It is obvious that the language generated will still be the same.

(2) *Backward substitution*

$$A \to w_1 | \cdots | w_m$$

$$A \to u_1 z v_1 | \cdots | u_p z v_p$$

$$A \to z.$$

They are transformed to

$$A \to z$$

$$A \to w_1 | \cdots | w_m \Big/ \bigwedge_{i=1}^{p} (\text{not } (u_i\text{–}v_i))$$

$$A \to u_1 A v_1 | \cdots | u_p A v_p \Big/ \bigwedge_{i=1}^{p} (\text{not } (u_i\text{–}v_i)).$$

This transformation is used when z is a long string of symbols, which is repeated in several productions. This transformation does not change the number of productions, nor the number of non-terminal symbols. It aims to minimize the total number of symbols used in representing the grammar. This transformation cannot be applied if there is a sentential form which includes $u_i A v_i$.

It is obvious that the replacement of the production $A \to uzv$ by the pair $A \to uBv$ and $B \to z$, where B is a new non-terminal, will not alter the language generated. Combining with transformation rule (3) to replace non-terminal B by A, we can get the productions as stated. It will be shown later that transformation rule (3) does not alter the language generated either.

(3) *Elimination*

$$A \to w_1 | \cdots | w_m, \quad B \notin w_i$$

$$A \to u_1 B v_1 | \cdots | u_p B v_p$$

$$B \to z_1 | \cdots | z_n, \quad B \notin z_i$$

$$B \to x_1 B y_1 | \cdots | x_q B y_q$$

$$D_k \to s_k B t_k, \quad k = 1, r.$$

They are transformed to

$$\left.\begin{array}{l} A \to w_1 | \cdots | w_m \\ A \to u_1 A v_1 | \cdots | u_p A v_p \end{array}\right\} \Big/ \left(\bigwedge_{i=1}^{p} (\text{not } (u_i\_v_i)) \right) \wedge \left(\bigwedge_{j=1}^{q} (\text{not } (x_j\_y_j)) \right) \wedge \left(\left(\bigwedge_{k=1}^{r} (\text{not } (s_k\_t_k)) \right) \vee \text{not } \delta(D\_) \right)$$

$$\left.\begin{array}{l} A \to z_1 | \cdots | z_n \\ A \to x_1 A y_1 | \cdots | x_q A y_q \end{array}\right\} \Big/ \left(\bigvee_{i=1}^{p} (u_i\_v_i) \right) \vee \left(\bigvee_{j=1}^{q} (x_j\_y_j) \right) \vee \left(\bigvee_{k=1}^{r} ((s_k\_t_k) \wedge \delta(D\_)) \right)$$

$$D_k \to s_k A t_k, \quad k = 1, r.$$

This transformation is the most commonly used one to eliminate non-terminal symbol B.

This transformation cannot be applied if there is a sentential form which includes $u_i A v_i$, $x_j A y_j$ or $s_k A t_k$, unless u_i, v_i are null strings or s_k, t_k are null strings. If $A \to B$ is a production in the grammar, then the self production rule $A \to A$ can be eliminated after transformation. Any constraints associated with the production $A \to B$ will be added to productions of the form $B \to w$ if appropriate.

The local constraints introduced by this transformation rule are used to identify the place, where non-terminal B appears although B is now written as A. So the derivation structures of the transformed grammar are the same as the original grammar except B is replaced by A, and the chain $A \to B$ in the original derivation trees is deleted. Hence, the language generated after transformation is not altered.

(4) *Merging*

$A \to B$

$A \to s_i A t_i / x_i\_y_i, \quad 1 \leq i \leq n$

$A \to s_i B t_i / u_i\_v_i, \quad 1 \leq i \leq n$

$A \to w_1 | \cdots | w_m$

$B \to z_1 | \cdots | z_n.$

They are transformed to

$A \to B$

$$A \to s_i A t_i \Big/ ((x_i\_y_i) \vee (u_i\_v_i)) \wedge \left(\bigwedge_{j=1}^{n} (\text{not } (u_j s_j\_t_j v_j)) \right), \quad 1 \leq i \leq n$$

$$A \to w_1 | \cdots | w_m \Big/ \bigwedge_{j=1}^{n} (\text{not } (u_j s_j \_ t_j v_j))$$

$$B \to z_1 | \cdots | z_n.$$

This transformation rule replaces the derivation $A \to s_i B t_i$ by the pair $A \to s_i A t_i$ and $A \to B$. The local constraints introduced will help to identify the place where non-terminal B appears. Hence the language generated is not altered.

Note that s, t, u, v, w, x, y and z in the above transformation rules are strings over V^*.

Here we do not show the constraints that may be associated with the productions when using the above transformation rules. Any such constraints should be carried along to the transformed grammar. However, the transformation cannot be performed if the associated constraints contain a non-terminal that is going to be eliminated by the transformation rule applied.

In the above transformation rules, the local constraints introduced involve the whole string in the production rule. However, in most cases when we massage the grammar, one or two symbols in the constraints are good enough to ensure the correct productions be used. The set of transformation rules here is just a guideline about how local constraints should be introduced.

The contextual restrictions can also be simplified and merged by the following rules:

(1) $A \to w/(x\_y) \vee (ux\_yv)$ is equivalent to $A \to w/(x\_y)$,
(2) $A \to w/(x\_y) \wedge (ux\_yv)$ is equivalent to $A \to w/(ux\_yv)$,
(3) $A \to w/(\text{not } (x\_y)) \vee (\text{not } (ux\_yv))$ is equivalent to $A \to w/\text{not } (ux\_yv)$,
(4) $A \to w/(\text{not } (x\_y)) \wedge (\text{not } (ux\_yv))$ is equivalent to $A \to w/\text{not } (x\_y)$,
(5) $A \to w/(x\_y) \vee (\text{not } (x\_y))$ is equivalent to $A \to w$,
(6) $A \to w/(x\_y) \wedge (\text{not } (x\_y))$; this production can be deleted.

5.2. From skeletons

One notion of the phrase structure of a sentence is that it captures and formalizes our intuitive grasp of the organization of the elements in a sentence; this is developed in greater detail in [8], where it is formalized as a skeletal rewriting system.

Definition 5.1. A *skeletal rewriting system*, $G = (I, P)$, consists of a finite set of skeletons and a finite set of replacement rules, $S_1 \to S_2$, where S_1 and S_2 are skeletons.

The following theorems are given in [8]:

Theorem 5.1. *Let G be a skeletal rewriting system. The set of skeletons generated by G is the set of skeletons of a local set. (A local set is the set of derivation trees of a context-free grammar.)*

Theorem 5.2. *For every local set, L, there is a skeletal rewriting system which generates the set of skeletons of L.*

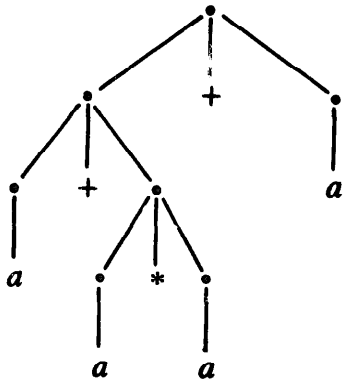
Also shown in [8] is the fact that a finite set of skeletons of a language uniquely determines a minimal context-free grammar for the language with the same phrase structure. Further, one can define skeletal automata analogous to bottom-up tree automata, and the class of skeletal automata characterizes the skeletal set of local sets.

Definition 5.2. A (frontier-to-root) *skeletal automaton*, M_S, is an automaton $M_S = (S, T, M, F)$, where S is a finite set of states, T a finite set of node labels, $F \subseteq S$ a set of final states, and $M \subseteq (S)^* \times S$.

M is the *transition relation* which assigns a state to a node labeled m given the states of all its direct successors. If the state of m is uniquely determined, then M_S is said to be *deterministic*; otherwise, M_S is *non-deterministic*. If (w, x) is in M, where w is in (S^*) and x in S, we write $w \to x$. The nodes on the frontier of a skeleton have no direct successors, they are assigned 'initial' states.

Definition 5.3. Let $G = (V, T, P, S)$ be a wide sense context-free grammar, [12], M_G, the recognizer of G is a skeletal automaton defined as the 'inverse of G'. $M_G = (V, T, M, S)$, where $w \to x$ is in M iff $x \to w$ is in P. (Note that M_G is, in general, non-deterministic.) The accepting states of M_G are the starting symbols in G, and each transition rule in M_G is the inverse of a generative rule in G.

Example 5.1. In the phrase structuring of arithmetic expressions, if we agree that the operators associate to the left, that parentheses dominate, and use the accepted precedence of operators, then the phrase structuring of each arithmetic expression is uniquely determined – although the syntactic category names are arbitrary. Thus, the phrase structure of the expression $a + a * a + a$ is given by the skeleton:



The unique minimal (wide-sense) grammar of arithmetic expressions with parentheses is summarized by the schemes:

$$S = \{E_0, E_+, E_*\}$$

1. $E_0 \to a$
2. $E_0 \to (V_1)$
3. $E_+ \to V_1 + V_2$
4. $E_* \to V_2 * E_0$,

where $V_1 \in \{E_0, E_+, E_*\}$ and $V_2 \in \{E_0, E_*\}$.

Rule scheme 2 stands for 3 rules, rule scheme 3 stands for 6 rules, and rule scheme 4 stands for 2 rules.

The minimal grammar G is derived, in general, by finding the unique minimal (deterministic) skeletal automaton, assigning a different variable to each distinct state, and inverting the rules of the automaton, A. Thus,

$$w \to x_1 \cdots x_n \in G \Leftrightarrow \quad \bullet \quad \to w \in A.$$
$$\qquad\qquad\qquad\qquad\qquad\quad x_1 \cdots x_n$$

Having developed a minimal context-free grammar, G, yielding the skeletal set, one can systematically develop all of the contexts in which each variable can occur, and generate a local constraints grammar. Suppose that $A \to uBv$ for some $A, B \in N$ and $u, v \in V^*$ in G, then u_v is an allowable context for B in G. In other words, there is a proper analysis of some derivation tree in G of the form $xuBvy$, where $x, y \in V^*$. Since the set of proper analyses is just the set of sentential forms, it is a context-free language and one can effectively compute the set of allowable contexts up to any fixed length k. The possible left and right contexts of length 1 are:

	left contexts	right contexts
E_*	+, (	+, *,)
E_+	(	+,)
E_0	+, *, (	+, *,)

Note that these contexts are exactly those specified by the local constraints grammar of arithmetic expressions given in Example 3.1.

Example 5.2. Let G be a grammar for typed expressions as given in Example 3.2. The minimal wide sense grammar for G is:

$S \to E_1;$

$S \to E_2;$

$E_1 \to E_1 \theta E_1$

$E_2 \to E_2 \theta E_2$

$E_1 \rightarrow a_1$

$E_2 \rightarrow a_2$

The allowable contexts (on the right) for E_1, E_2 are, up to length 2,

		right contexts	
E_1	;,	θE_1,	θa_1
E_2	;,	θE_2,	θa_2

Incorporating these constraints yields the local constraints grammar

$S \rightarrow E$;

$E \rightarrow E\theta E$;

$E \rightarrow a_1 | -\theta a_1 \vee -$;

$E \rightarrow a_2 | -\theta a_2 \vee -$;

5.3. Transformation of local constraints grammars to context-free grammars

We will describe a procedure which can directly transform a given local constraints grammar to an equivalent context-free grammar.

In general, a formal approach should follow the proof in [6] which involves the following steps:

(1) construct frontier-to-root automata which are able to check each individual constraint,

(2) construct a composite tree automaton from the negation and combination of those tree automata,

(3) convert the composite tree automaton into an equivalent context-free grammar.

However, the formal approach is very difficult to state in a clear way, and the composite tree automaton constructed is so complicated that it is not practical to follow this formal construction. Here we introduce a heuristic approach for converting a local constraints grammar to an equivalent context-free grammar, which works directly on the grammar itself rather than going through tree automata constructions. However, this heuristic procedure may not work for all local constraints grammars: it works mostly for grammars whose constraints can be easily checked.

The basic idea is to introduce all necessary non-terminal symbols first in order to separate non-terminals because of their different constraints. It is analogous to assigning different names to different states in a tree automaton. We then write down all possible productions for those newly created non-terminals, and then check all production rules, and delete those which do not satisfy the constraints.

The procedure of transforming a given local constraints grammar to a context-free grammar consists of the following steps:

(1) if non-terminal A appears in the left side of more than one productions all of which have different constraints, then A is treated as different symbols for rules with

different constraints. For example, if

$$A \to x/r_1, \qquad A \to y/r_2,$$

where r_1 and r_2 are different constraints, and

$$B \to uAv/r_3$$

then they are transformed to

$$A_1 \to x/r_1 \qquad B \to uA_1v/r_3$$

$$A_2 \to y/r_2 \qquad B \to uA_2v/r_3$$

(2) if non-terminal A in (1) is a starting symbol, then the newly created symbols $A_1, A_2, \ldots$ are in the set of starting symbols;

(3) check all production rules and delete those which do not satisfy the constraints. For a production $A \to uBv/C_A$, first check if the constraints C_B in $B \to r/C_B$ satisfies the context u and v, or their descendants; if u or v partially matches with C_B, then check the constraints further with productions of the form $D \to sAt/C_D$;

(4) after all constraints are satisfied in the production rules, the constraints can then be dropped;

(5) remove all useless non-terminals and productions.

Note that the resulting grammar is a wide sense context-free grammar.

Example 5.3.

G_1: $S \to aSa/\text{not } a\_$
$S \to bSb/\text{not } b\_$
$S \to a/\text{not } a\_$
$S \to b/\text{not } b\_$
$S \to e/(\text{not } a\_) \wedge (\text{not } b\_).$

After steps (1) and (2), the grammar becomes:

$S = \{S_1, S_2, S_3\}$
$S_1 \to aS_1a/\text{not } a\_$
$S_1 \to aS_2a/\text{not } a\_$
$S_1 \to aS_3a/\text{not } a\_$
$S_1 \to a/\text{not } a\_$
$S_2 \to bS_1b/\text{not } b\_$
$S_2 \to bS_2b/\text{not } b\_$
$S_2 \to bS_3b/\text{not } b\_$
$S_2 \to b/\text{not } b\_$
$S_3 \to e/(\text{not } a\_) \wedge (\text{not } b\_).$

Since S_1 cannot have 'a' as left context, the first production $S_1 \to aS_1a$ apparently does not satisfy the constraints and thus can be deleted. After deleting all non-

satisfying productions in step (3), the grammar becomes:

$S = \{S_1, S_2, S_3\}$
$S_1 \to aS_2a/\text{not } a\_$
$S_1 \to a/\text{not } a\_$
$S_2 \to bS_1b/\text{not } b\_$
$S_2 \to b/\text{not } b\_$
$S_3 \to e/(\text{not } a\_) \wedge (\text{not } b\_).$

Now we can drop the constraints associated with the productions. The resulting grammar is:

$S = \{S_1, S_2, S_3\}$
$S_1 \to aS_2a$
$S_1 \to a$
$S_2 \to bS_1b$
$S_2 \to b$
$S_3 \to e.$

References

[1] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol. 1* (Prentice-Hall, Englewood Cliffs, NJ, 1972).

[2] P. Buneman and L.S. Levy, Proper analysis algorithms, *Proc. 1978 Conference on Information Sciences and Systems*, Johns Hopkins University, Baltimore, MD (1978).

[3] J. Earley, An efficient context-free parsing algorithm, *Comm. ACM* **13** (1970).

[4] S.L. Gerhart, Correctness preserving program transformations, *Proc. ACM-SIGACT POPL Symposium* (1976).

[5] J.N. Gray and M.A. Harrison, On the covering and reduction problems for context-free grammars, *J. ACM* **19** (1972) 675–695.

[6] A.K. Joshi and L.S. Levy, Constraints on structural descriptions: local transformations, *SIAM J. Comput.* (1977).

[7] L.S. Levy, Automata on trees: a tutorial survey, *Egypt. Comput. J.*, to appear.

[8] L.S. Levy and A.K. Joshi, Skeletal structural descriptions, *Information and Control* **39** (1978) 192–211.

[9] M. Marcotty, H.F. Ledgard and G.V. Bochmann, A sampler of formal definitions, *Comput. Surveys* **8** (2) (1976) 191–276.

[10] S. Peters and R.W. Ritchie, Context-sensitive immediate constituent analysis – context-free languages revisited, *Proc. ACM Symposium on Theory of Computing* (1969).

[11] T.A. Standish et al., The Irvine program transformation catalog, Technical Report, Department of Information and Computer Science, University of California, Irvine (1976).

[12] A. Salomaa, *Formal Languages* (Academic Press, New York, 1973).

[13] B.I. Soroka, LOCAL – A system for experimenting with local transformations, Masters Thesis, University of Pennsylvania, Philadelphia (1976).

[14] J.W. Thatcher, Tree automata: an informal survey, in: A.V. Aho, ed., *Currents in Theory of Computing* (Prentice-Hall, Englewood Cliffs, NJ, 1973).

[15] K. Yueh, Local constraints in the syntax and semantics of programming languages, Ph.D. Dissertation, University of Pennsylvania, Philadelphia (1978).