PARSING AND SYNTACTIC ERROR RECOVERY FOR CONTEXT-FREE GRAMMARS BY MEANS OF COARSE STRUCTURES

Ernst-Wolfgang Dieterich
Institut für Informatik der TU München, 8 München 2, Arcisstr. 21

Introduction

In all high level programming languages there are some patterns of tokens often called delimiters by which a program is coarsely structured, i.e. which characterize certain substructures of a program. Such patterns determine a "coarse structure" (cs) of a grammar. For example, begin...end will be characteristic brackets for <block> and if...then...else...fi a characteristic pattern for <if statement> in an Algol-like language. In addition, the string occurring between if and then in a correct program is known to be reducible to the nonterminal <Boolean expression> using only a certain subgrammar. A programmer, too, tries to understand a possibly incorrect program by first looking for the syntactic structure of cs tokens.

In this paper we describe two applications of coarse structures: simplifying the parsing process and analysing and correcting syntax errors where global rather than local context is used. In section 1 we introduce a restricted type of coarse structures. For a general definition of coarse structure we refer to [3]. In section 2 we state the theoretical results for the simplification of the parsing process achieved by means of coarse structures which is the topic of section 3. The last section 4 deals with the application of coarse structures to error recovery.

1. Definitions and notation

Let $G = (V,T,P,Z)$ be a context-free grammar, where V is the vocabulary of G, $T \subset V$ the set of terminals, $N := V - T$ the nonempty set of nonterminals, P a finite set of productions and $Z \in N$ the axiom. V^* denotes the set of all words over V and $V^+ := V^* - \{\varepsilon\}$ where ε is the empty word. For $x \in V^*$ $lg(x)$ is the length of the word x, i.e. the number of symbols in x. $\underset{G}{\rightarrow}$ is the usual relation *directly derived* with respect to G and $\underset{G}{\Rightarrow}$ ($\underset{G}{\Rrightarrow}$) the transitive (transitive and reflexive) closure of $\underset{G}{\rightarrow}$. A rightmost derivation is represented by the relations $\overset{r}{\underset{G}{\rightarrow}}$, $\overset{r}{\underset{G}{\Rightarrow}}$, and $\overset{r}{\underset{G}{\Rrightarrow}}$. $S(G) := \{x \ / \ Z \underset{G}{\Rrightarrow} x\}$ is the *set of sentential forms* of G.

For any $K \subset V$ we define the homomorphism h_K as $h_K(X) := \begin{cases} X \text{ if } X \in K \\ \varepsilon \text{ otherwise.} \end{cases}$

A set $K \subset V$ is called *kernel alphabet* of G iff for all productions $N::=x \in P$ where $N \in K$ we have $h_K(x) \neq \varepsilon$.

The elements of a kernel alphabet are called *kernel tokens*. A kernel alphabet K determines the set of *kernel productions* $P_K := \{ X::=x \in P \ / \ h_K(x) \neq \varepsilon \}$.

Let $N_K := \{ X \ / \ \exists X::=x \in P_K \}$ and $N' := \{ X' \ / \ X \in N_K \}$ with $N' \cap V = \emptyset$. The set P' of productions is defined by the following algorithm:

(1) $P' := P - P_K$;
(2) $X::=x \in P'$ with $X \in N_K$ is replaced by $X'::=x$;
(3) if $X::=aYb \in P'$ with $Y \in N_K$ we add $X::=aY'b$ to P'.

The *kernel-free subgrammar* with axiom $u \in (V-K)^*$ is defined as $G_f(K,u) = (V \cup N' \cup \{Z_u\}, T_f, P_f, Z_u)$ where $T_f := T \cup N_K$ and $P_f := P' \cup \{ Z_u::=u \}$, $Z_u \notin V$.
Let $S_f(u)$ be the set of all sentential forms of $G_f(K,u)$, then we have $S_f(u) \subset (V-K)^*$.

The *K-coarse structure* CS(G,K) of G is a system of rules

$$R := \{ (x_0 A_1 x_1 \ldots A_n x_n \longleftarrow M \text{ and } x_\nu \in S_f(u_\nu),\ 0 \leq \nu \leq n) \ / \ M::=u_0 A_1 u_1 \ldots A_n u_n \in P_K,\ A_1 \ldots A_n \in K^+,\ u_0 \ldots u_n \in (V-K)^* \}$$
$$\cup \ \{ (\$x\$ \longleftarrow Z' \text{ and } x \in S_f(Z)) \} \text{ with } \$, Z' \notin V.$$

A rule of CS(G,K) is applicable only if $x_\nu \in S_f(u_\nu)$ holds for all ν. $A_1 \ldots A_n$ is called *kernel pattern*.

Example 1: Consider the following extract of an Algol-like grammar G with axiom <block>:

1. <block> ::= begin <decl> . <stmt list> end | begin <stmt list> end
2. <decl> ::= <decl part> | <decl> ; <decl part>
3. <decl part>::= type id | type id = <sexpr>
4. <stmt list>::= <stmt> |<stmt> ; <stmt list>
5. <stmt> ::= id := <expr> | <block> |
   <if clause> <stmt list> fi |
   <if clause> <stmt list> else <stmt list> fi
6. <if clause>::= if <expr> then
7. <expr> ::= <sexpr> | <sexpr> = <sexpr>
8. <sexpr> ::= id | <sexpr> + id

If we choose the kernel alphabet K = { begin, . , end, if, then, <if clause>, else, fi } the rules of the K-coarse structure CS(G,K) are

(begin x . y end <-- <block> and x ∈ S_f(<decl>), y ∈ S_f(<stmt list>)),
(begin x end <-- <block> and x ∈ S_f(<stmt list>)),
(<if clause> x fi <-- <stmt> and x ∈ S_f(<stmt list>)),

(<if clause> x else y fi <-- <stmt> and x,y ∈ S_f(<stmt list>)),
(if x then <-- <if clause> and x ∈ S_f(<expr>)),
($\$$ x $\$$ <-- Z' and x ∈ S_f(<block>) = {<block>}).

The kernel token of the third alternative of production 5. is <if clause> fi. The kernel-free subgrammar G_f(K,<stmt list>) consists of the following productions:

```
Z<stmt list> ::= <stmt list>
<stmt list>  ::= <stmt> | <stmt> ; <stmt list> | <stmt'> | <stmt'> ; <stmt list>
<stmt'>      ::= id := <expr> | <block>
<expr>       ::= <sexpr> | <sexpr> = <sexpr>
<sexpr>      ::= id | <sexpr> + id                    □
```

It can be shown that the set of all strings reducible to Z' by CS(G,K) is exactly the set of sentential forms of G enclosed by $\$$.

According to the intuitive idea of coarse structures the syntactic structure of the kernel tokens should be recognized in a very simple way. This can be achieved by the following restrictions:

A K-coarse structure is called

- *simple* iff it does not contain two rules with kernel patterns a and b such that
  (i) $a = lbr$ with $l,r \in K^+$ or
  (ii) $a = lc$, $b = cr$ with $c,l,r \in K^*$ and $c \neq \varepsilon$, $lr \neq \varepsilon$;
- *deterministic* iff
  (i) for each rule ($x_0 A_1 x_1 \ldots A_n x_n$ <-- M and $x_\nu \in S_f(u_\nu)$, $0 \le \nu \le n$) we have $u_0 = u_n = \varepsilon$,
  (ii) there are not two different rules in R with the same kernel pattern.

The K-coarse structure of example 1 is simple and deterministic.

Replacing the conditions $x_\nu \in S_f(u_\nu)$ contained in the rules of a simple deterministic K-coarse structure by the weaker conditions $x_\nu \in (V-K)^*$ we get the *K-structure* St(G,K) of G. There is a one-to-one mapping from the rules of CS(G,K) to those of St(G,K).

The cs phrases are recognized by means of the rules of a K-structure, and the corresponding rules of the K-coarse structure contain all the information about how to parse the remaining "fine structure".
Note that kernel-free subgrammars may possess further K'-coarse structures.

2. Properties of kernel-free subgrammars

The substrings of a program occurring between two tokens of a kernel pattern of a K-structure have to be parsed by uniquely determined kernel-free subgrammars. In this section we study how difficult it is to parse these strings in comparison to the total grammar. From our definition of kernel-free subgrammars we could expect that there are less local ambiguities than in the total grammar. In that case the parsing algorithm for each subgrammar becomes simpler (or at least not more difficult) than that for the total grammar. Note that the axioms of kernel-free subgrammars are always substrings of right hand sides of productions.

In this section we present the solution for the class of LR, LL, BRC, and precedence grammars. For definitions we refer to [1].

Let $G = (V,T,P,Z)$ be a context-free grammar, $CS(G,K)$ a K-coarse structure and u a nonempty substring of a right hand side of a production of G.

Theorem 1: If G is LR(k), then $G_f(K,u)$ is LR(k') for some $k' \leq k$.

Proof: In the following we use the index f instead of G_f. Suppose $G_f(K,u) := G_f = (V_f,T_f,P_f,Z_u)$ is not LR(k). Then there are two rightmost derivations in G_f

$$(1) \quad Z_u \xrightarrow[f]{} u \overset{r}{\underset{f}{\Longrightarrow}} aAw \xrightarrow[f]{r} abw, \quad w \in T_f^* \text{ and}$$

$$(2) \quad Z_u \xrightarrow[f]{} u \overset{r}{\underset{f}{\Longrightarrow}} cBx \xrightarrow[f]{r} aby, \quad x \in T_f^*$$

with $First_k(w) = First_k(y)$ and $aAy \neq cBx$, i.e. $a \neq c$ or $A \neq B$ or $y \neq x$.

Because u is a substring of a right hand side of a production, there is a rightmost derivation in G

$$(3) \quad Z \overset{r}{\underset{G}{\Longrightarrow}} dut \quad \text{with } t \in T^*.$$

The terminal alphabet of G_f consists of some terminals and some nonterminals of G. Therefore we can write $First_k(w) = First_k(y) = x_o N_1 x_1 \ldots N_m x_m$ with $m \geq o$, $x_\mu \in T^*$, $o \leq \mu \leq m$, and $N_\mu \in N \cap T_f$, $1 \leq \mu \leq m$. Replacing the first m nonterminals $N_1,\ldots,N_m$ of w and y by terminal strings $t_1,\ldots,t_m$ with $N_\mu \overset{r}{\underset{G}{\Longrightarrow}} t_\mu$ we get

$$(4) \quad w \overset{r}{\underset{G}{\Longrightarrow}} w' \in T^* \text{ and } y \overset{r}{\underset{G}{\Longrightarrow}} y' \in T^*.$$

It can be shown that each $t_\mu \neq \varepsilon$. Therefore we have

$$(5) \quad First_k(w') = First_k(y') = First_k(x_o t_1 x_1 \ldots t_m x_m).$$

Combining (1) up to (5) we have

$$(6)\quad Z \overset{r}{\underset{G}{\Longrightarrow}} dut \overset{r}{\underset{G}{\Longrightarrow}} daAw't \overset{r}{\underset{G}{\longrightarrow}} dabw't$$

$$(7)\quad Z \overset{r}{\underset{G}{\Longrightarrow}} dut \overset{r}{\underset{G}{\Longrightarrow}} dcBx't \overset{r}{\underset{G}{\longrightarrow}} daby't$$

$$(8)\quad First_k(w't) = First_k(y't).$$

If $y \neq x$, then we have $y't \neq x't$. Otherwise we have $da \neq dc$ or $A \neq B$, i.e. $daAy't \neq dcBx't$, a contradiction to the LR property of G. □

Because of the symmetry between the definition of LR and LL grammars as well as between rightmost and leftmost derivations an analogous statement holds for LL grammars.

Theorem 2: If G is (m,n)-BRC, then $G_f(K,u)$ is (m',n')-BRC for some $m' \leq m$, $n' \leq n$.

Proof: Suppose $G_f(K,u)$ is not (m,n)-BRC. Then there are two rightmost derivations in G_f

$$(1)\quad \&^m Z_u \&^n \longrightarrow \&^m u \&^n \overset{r}{\underset{f}{\Longrightarrow}} \&^m aAw\&^n \overset{r}{\underset{f}{\longrightarrow}} \&^m abw\&^n \quad , \quad w \in T_f^*$$

$$(2)\quad \&^m Z_u \&^n \longrightarrow \&^m u \&^n \overset{r}{\underset{f}{\Longrightarrow}} \&^m cBx\&^n \overset{r}{\underset{f}{\longrightarrow}} \&^m cdx\&^m = \&^m a'by\&^n \quad , \quad y \in T_f^*$$

with (3) $lg(x) \leq lg(y)$

(4) $Last_m(a') = Last_m(a)$ and $First_n(w) = First_n(y)$

such that $a'Ay \neq cBx$, i.e. $a' \neq c$ or $A \neq B$ or $x \neq y$. Furthermore there is a rightmost derivation in G

$$(5)\quad \&^m Z \&^n \overset{r}{\underset{G}{\Longrightarrow}} euf \quad \text{with } f \in T^*\&^n.$$

Substituting each nonterminal in $First_n(w) = First_n(y)$ by an appropriate terminal string and the other nonterminals in w and y by some others we get w' and y' and analogously to the previous proof $First_n(w') = First_n(y')$. Combining (1) and (2) with (5) we get two rightmost derivations in G with the same n symbols to the right and the same m symbols to the left of a possible right hand side but not with the same parsing action, a contradiction to the (m,n)-BRC property of G. □

Theorem 3: If G is an operator precedence or a simple precedence grammar, then $G_f(K,u)$ is of the same type.

Proof: Let $u = A_1 \ldots A_n$ and $\$$ be an endmarker not in V, we have $\$ \lessdot_f X$ and $Y \gtrdot_f \$$ for all $X \in First^{Gf}(A_1)$ and all $Y \in Last^{Gf}(A_n)$ where $Gf := G_f(K,u)$. If we have $A \rho_f B$ then $A \rho B$ holds where ρ_f and ρ is one of the precedence relations of $G_f(K,u)$ and G, respectively. If $A' \rho_f B$, $A \rho_f B'$ or $A' \rho_f B$ holds then we also have $A \rho_f B$. Since u is a substring of a right hand side of a production of G, we have $A_\nu \doteq A_{\nu+1}$ as well as $A_\nu \doteq_f A_{\nu+1}$ for $1 \le \nu \le n-1$. Therefore if $G_f(K,u)$ has two or more different relations between two symbols then the same is valid in G, which contradicts the assumption. □

One could think u to be any substring of a sentential form occurring in a rightmost or a leftmost derivation, respectively. The following examples show that in such a case we can get kernel-free subgrammars which are more complicated than the total grammar.

Example 2: We consider the following grammar G:

Z::=$ E $ E::=E + T ! T T::=T * F | F F::=(E) | i

G is known to be LR(1). There are sentential forms of rightmost derivations a substring of which is $E\ \{+\ i\}^n$, $n \geq o$. It can easily be shown that for each $n \ge o$ $G_f(\{\$\},\ E\ \{+\ i\}^n)$ is LR(2n+1) but not LR(2n). □

Example 3: The grammar with the productions

Z::=$ L $ L::=N A C N::=M B M::=K A K::=B

is an operator and a simple precedence grammar. The string MBA is a substring of the rightmost sentential form $MBAC$. Because of the axiom production Z_{MBA}::=MBA of $G_f := G_f(\{\$\},MBA)$ we get the relation $B \doteq_f A$ in addition to the relation $B \gtrdot_f A$ such that G_f is neither a simple nor an operator precedence grammar. □

3. Simplification of parsing

The results of section 2 suggest that the use of coarse structures simplifies parsing of context-free grammars. In order to demonstrate the power of this strategy we consider the following grammar:

Example 4:

o. Z::=A	3. L::=L+I	8. B::=B+C
1. A::=[L then B]	4. L::=I	9. B::=C
2. A::=[B if L]	5. L::=A	1o. B::=A
	6. I::=i	11. C::=i
	7. I::=(L)	12. C::=(B)

Since the sublanguages for L and B are identical the grammar is neither LR nor LL.

We choose the kernel alphabet K = {[,], if, then} which determines a simple deterministic K-coarse structure. The only two kernel-free subgrammars are $G_f(K,L)$ and $G_f(K,B)$ both of which are (1,o)-BRC. □

Simplifying the syntactic analysis in this way we arrive at the following problem: Given a unique context-free grammar G we wish to parse it by means of a simple deterministic K-coarse structure such that all kernel-free subgrammars are of a given "simple" grammar class X. Because parsing of coarse structures is oriented bottom-up on principal we will choose only bottom-up classes, e.g. $X \in \{LR(k), (m,n)\text{-BRC} / m,n,k \geq o\}$.

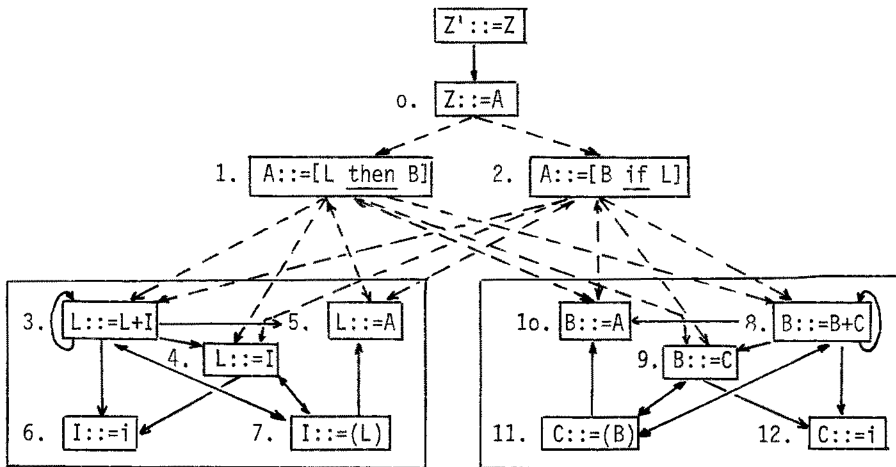
In order to find an appropriate kernel alphabet K we proceed as follows:
(1) We construct an X-parser for G.
(2) If G is an X-grammar we are done. Otherwise several ambiguous entries will occur in the parsing table. These entries correspond to a set of *critical pairs* of productions. Now we may try to construct a simple deterministic K-coarse structure such that none of its kernel-free subgrammars contains a critical pair of productions, i.e. the K-coarse structure *separates* the critical pairs of productions.

To that end we consider the augmented grammar G' derived from G [1]as a directed graph Gr(G') as follows [6]:
Each production is a vertex in Gr(G'). Let p = N::=n, p' = M::=m ∈ P, then there is an arc from p to p' iff n = aMb for some $a,b \in V^*$. Note that for a reduced augmented grammar G' the graph Gr(G') is connected where a vertex corresponding to the axiom production Z'::=Z is an ancestor of each other vertex. A K-coarse structure divides the grammar G' into several kernel-free subgrammars. This is represented in Gr(G') as follows: (i) Each vertex corresponding to a kernel production is deleted from Gr(G') together with each arc starting or ending in this vertex; Gr(G') is divided into a set of connected subgraphs. (ii) For each kernel-free subgrammar with axiom u determined by the K-coarse structure we have to insert a vertex corresponding to the axiom production Z_u::=u and all arcs according to the above definition. Thus in general some connected subgraphs are recombined into one connected subgraph. Changing some nonterminals N into N' and introducing some new productions according to the definition of kernel-free subgrammars does not change the subgraphs essentially. Now a kernel-free subgrammar with axiom u consists of all productions corresponding to vertices which are descendants of the vertex corresponding to Z_u::=u.

In constructing an appropriate K-coarse structure we have to determine for each critical pair (p,p') of productions sets of vertices such that in the subgraph of Gr(G') without these vertices there is no common ancester of p and p'. Such sets of vertices are called *separating sets*. Because we are interested only in simple

deterministic K-coarse structures some of the separating sets for a critical pair of productions are rejected from the beginning. Now we have to construct a kernel alphabet K such that CS(G',K) is simple and deterministic and the set of the kernel productions contains the productions corresponding to a separating set for each critical pair of productions. For details cf. [3]. In the last step we have to guarantee that no axiom production will recombine two productions of a critical pair.

Example 5: From example 4 we derive the following graph Gr(G'):



For each X ∈ {(m,k)-BRC,LR(k) / m≥1,k≥o} we have the two critical pairs of productions: (5.,1o.) and (6.,11.). A separating set for both is {1.,2.}. Looking for a simple deterministic K-coarse structure with 1. and 2. as kernel productions we get the one given in example 4. The vertices corresponding to the two axiom productions Z_L::=L and Z_B::=B are ancestors of only the left and right framed part of the graph, respectively. □

4. Syntactic Error Recovery

In the philosophy of top-down programming a programmer chooses a certain construct such as a block or a loop which has to do something, and in a second step he develops this construct. In general such constructs are characterized by special delimiters, i.e. in our terminology by special kernel patterns. This way of program development in mind we suppose that kernel tokens are syntactically more important than other symbols. Using a simple deterministic K-coarse structure for syntactic error reco-

very in connection with bottom-up parsing yields the following advantages:

(i) syntactic errors are separated on two levels: errors in the coarse structure where we have to repair the h_K-image of the program, and errors in certain subgrammars where error recovery is often much simpler than in the total grammar;
(ii) if efficiency doesn't matter, e.g. in a separate syntax check, we can first build up and eventually repair the syntactic structure of the kernel tokens. If a reasonable repair is impossible we can stop the parse of that particular part of the program asking the programmer for correction of this error. Thus in the case of serious errors like missing ends we avoid the usual listing of senseless error messages like "identifier not declared" etc.;
(iii) looking for a kernel pattern enclosing the error position we use, in contrast to most of the other error recovery methods, global rather than local context for repairing the error.

In order to find a kernel pattern as close to the error position as possible we favour large kernel alphabets.

If a bottom-up parser detects some syntax error it has to search in the stack for the longest string $A_n \ldots A_1$ of kernel tokens which is a prefix of a kernel pattern. Three cases are possible:
(a) it finds kernel tokens $B_1, \ldots, B_m$ in the input such that $A_\nu \ldots A_1 B_1 \ldots B_m$ ($\nu \leq n$) is a kernel pattern. The error position is between A_1 and B_1;
(b) some of the kernel tokens of the input form a kernel pattern which has to be reduced before the pattern prefix in the stack can be completed;
(c) no kernel pattern enclosing the error position can be found because there is no prefix of a kernel pattern or this prefix cannot be completed, i.e. an error in the coarse structure is detected.

In the first case the kernel-free subgrammar in which the error occurs is uniquely determined and an appropriate repairing algorithm can be used.

Example 6: We consider the following erroneous sentences of the grammar of example 1.

(1) begin id = id + id; type id = id. id := id + id end
↑

(2) begin id = id + id; id := id + id end
↑

In both cases the error position is between the second and the third symbol of the program. We use the K-coarse structure given in example 1. Then in program (1) we find the enclosing kernel pattern begin . end indicating that the error has to be corrected in the kernel-free subgrammar G_f(K,<decl>). Correcting this error a type will be inserted after the begin symbol. In program (2) the enclosing kernel pattern is begin end and the correction has to take place in the kernel-free

subgrammar G_f(K,<stmt list>) where the minimum distance correction will be changing the "=" to a ":=". Thus we can distinguish the two locally identical error situations by means of kernel context. A third type of such an error is given in example 7. □

After having reduced some kernel patterns case(b) will lead to case(a) or case(c).

If there is an error in the coarse structure (case(c)) we have to repair the syntactic structure of the kernel tokens. This can be done using a local correction method for kernel tokens similar to that introduced in [2] or a modification of the algorithm of [5] generalized from pairs of brackets to patterns. As another possibility we present the application of a generalized precedence method in parsing the kernel tokens together with an appropriate error recovery method.

Let $K \subset V$ be a kernel alphabet of the context-free grammar $G = (V,T,P,Z)$ and $A,B \in V$. Then we define the following relations on V [4]:

$$A \ \alpha[K] \ B \quad :\Leftrightarrow \quad \exists u \in (V-K)^* \ \exists v \in V^* \text{ such that } B::=uAv \in P$$

$$A \ \omega[K] \ B \quad :\Leftrightarrow \quad \exists u \in V^* \ \exists v \in (V-K)^* \text{ such that } B::=uAv \in P$$

$$A \ \sim[K] \ B \quad :\Leftrightarrow \quad \exists u \in (V-K)^* \ \exists v,w \in V^* \ \exists C \in N \text{ such that } C::=vAuBw \in P$$

We denote the transitive (transitive and reflexive) closure of $\alpha[K]$ and $\omega[K]$ by $\overline{\alpha}[K]$ and $\overline{\omega}[K]$ ($\overline{\overline{\alpha}}[K]$ and $\overline{\overline{\omega}}[K]$). Using products of relations and the inverse $(\ )^{-1}$ of a relation we define the generalized precedence relations for kernel tokens as follows:

$$\doteq[K] := \ \sim[K] \cap (K \times K)$$

$$\lessdot[K] := \ \sim[K](\overline{\alpha}[K])^{-1} \cap (K \times K)$$

$$\gtrdot[K] := \ \overline{\omega}[K] \sim[K] \ (\overline{\overline{\alpha}}[K])^{-1} \cap (K \times K)$$

Now we can parse the h_K-image of a program according to a usual precedence method with the only modification that a reduction will cause pushing the left hand side L onto the stack only if L is a kernel token. In addition an error recovery method for precedence parsers (e.g. [7]) may be applied.

It should be noted that we have only to require disjointness between $\gtrdot[K]$ and $\doteq[K] \cup \lessdot[K]$ (weak precedence), since in a simple K-coarse structure no kernel pattern can be a postfix of another kernel pattern.

All blank entries in our precedence matrix correspond to the empty relation ⊙ of [7]. As correcting actions we admit deletion or insertion of a kernel token. If in one situation both corrections are possible the decision about the right correction can sometimes be made by looking for special symbols in the kernel-free environment.

The following example will illustrate the proposed method for error recovery of

kernel tokens:

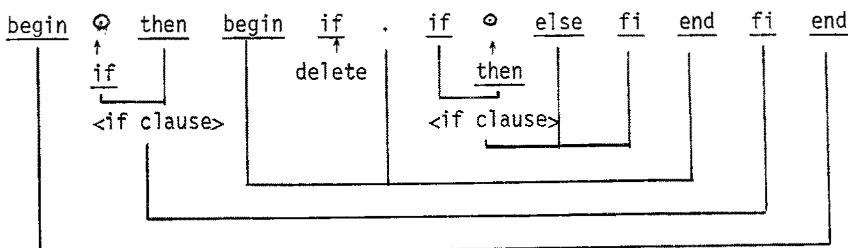
Example 7: First we give the table of the generalized precedence relations for the grammar and the kernel alphabet given in example 1. For brevity we omit the [K].

	begin	.	end	<if clause>	if	then	else	fi
begin	$\lessdot$	$\doteq$	$\doteq$	$\lessdot$	$\lessdot$			
.	$\lessdot$		$\doteq$	$\lessdot$	$\lessdot$			
end	$\gtrdot$		$\gtrdot$	$\gtrdot$	$\gtrdot$		$\gtrdot$	$\gtrdot$
<if clause>	$\lessdot$			$\lessdot$	$\lessdot$		$\doteq$	$\doteq$
if						$\doteq$		
then	$\gtrdot$			$\gtrdot$	$\gtrdot$		$\gtrdot$	$\gtrdot$
else	$\lessdot$			$\lessdot$	$\lessdot$			$\doteq$
fi	$\gtrdot$		$\gtrdot$	$\gtrdot$	$\gtrdot$		$\gtrdot$	$\gtrdot$

Consider the following erroneous program:

```
begin id = id + id then begin if id = id + id; type id .
        ↑
if id = id  id := id + id else id := id fi; id := id end fi end
```

The first error again occurs between the second and third symbol of the program. Looking for an enclosing kernel pattern we detect an error in the coarse structure. According to our precedence algorithm the first begin is pushed onto the stack. The empty relation holds between begin and then. There are two equally expensive corrections: deleting then or inserting if in front of then. In the first case the kernel-free string preceeding then must yield a statement in which a ":=" has to occur. Because this is not the case we choose the second alternative.If in some situation no appropriate correction can be done we stack the element and try to make a minimum distance correction when an erroneous right hand side is to be reduced. (Cf. the correction of the first if in our example.) We get the following structure for our example:



The completely corrected program will be given in example 8. □

The only information we get from an algorithm correcting the kernel tokens is the correction point within the h_K-image of the program. No problem arises when the algorithm decides to delete a kernel token. But if we have to insert a token A_μ in order to yield the pattern $A_1 \ldots A_m$ we do not know its exact position in the program. Let $N ::= A_1 u_1 A_2 \ldots u_{m-1} A_m$ be the corresponding kernel production. If the density of further errors in the fine structure is not too large we have some criteria to determine the exact position where A_μ is to be inserted.

For $1 < \mu < m$ we parse a prefix of the substring s occuring between $A_{\mu-1}$ and $A_{\mu+1}$ using the kernel-free subgrammar $G1 := G_f(K, u_{\mu-1})$. If we reach the string $u_{\mu-1}$ we have to test the next k input symbols $b = B_1 \ldots B_k$ ($k \geq 1$). There are three cases:

(1) $b \in Follow_k^{G1}(u_{\mu-1})$ and $b \notin First_k^{G2}(u_\mu)$ where
$Follow_k^{G1}(u) := \{z \,/\, Z_u \overset{*}{\underset{G1}{\Longrightarrow}} aux \text{ and } z \in First_k^{G1}(u)\}$ and $G2 := G_f(K, u_\mu)$.
Then parsing in G1 is continued.

(2) $b \notin Follow_k^{G1}(u_{\mu-1})$ and $b \in First_k^{G2}(u_\mu)$. A_μ is to be inserted and the remainder of s is to be parsed according to G2.

(3) $b \in Follow_k^{G1}(u_{\mu-1}) \cap First_k^{G2}(u_\mu)$. There is no exact criterium where A_μ should be inserted.

For $\mu = m$ $First_k^{G2}(u_\mu)$ must be replaced by $Follow_k^{G3}(N)$ with $G3 := G_f(K, Z)$.

Because we have not a parser for $G_f(K, u_1)$ working from right to left we use for $\mu = 1$ a stronger condition, for example: the terminal alphabet of $G_f(K, u_1)$ does not contain a symbol preceeding the left hand side N in any sentential form of G. Note that this set as well as Follow and First can easily be constructed form the given (sub-)grammars [1].

If we consider the production <block>::= <u>begin</u> <stmt list> <u>end</u> of the example grammar we get a formal criterium why it is impossible to determine the exact place where to insert a missing <u>end</u> :

For all $k \geq 1$ we have $Follow_k^{G1}(\langle stmt\ list \rangle) = Follow_k^{G2}(\langle block \rangle)$

with $G1 := G_f(K, \langle stmt\ list \rangle)$ and $G2 := G_f(K, \langle block \rangle)$.

<u>Example 8</u>: During the parse of the fine structure we get the following corrections for the program of example 7:

1. The missing <u>if</u> is to be inserted after the <u>begin</u>.
2. In $G_f(K, \langle decl \rangle)$ a <u>type</u> will be inserted at the place of the deleted <u>if</u>.
3. The missing <u>then</u> is to be inserted after "id = id", since

$id := \notin Follow_2^{G1}(\text{<expr>})$ and $id := \in Follow_2^{G2}(\text{<if clause>})$ with

$G1 := G_f(K,\text{<expr>})$ and $G2 := G_f(K,\text{<block>})$.

Thus we get the following corrected program:

<u>begin</u> <u>if</u> id = id + id <u>then</u> <u>begin</u> <u>type</u> id = id + id; <u>type</u> id .
<u>if</u> id = id <u>then</u> id := id + id <u>else</u> id := id <u>fi</u>; id := id <u>end</u> <u>fi</u> <u>end</u> □

Conclusion

Using the term "coarse structure of a context-free grammar" in accordance with a natural understanding and parsing of programs we have treated two of its main applications: we showed that two-level parsing by means of coarse structures can simplify the parsing process and that the concept of coarse structures can advantageously be used for syntax error recovery using global rather than local context of the error position.

Acknowledgement

The author is grateful to J. Ciesinger and W. Lahner for helpful discussions.

References

[1] AHO,A.V., ULLMAN,J.D.: The Theory of Parsing, Translation, and Compiling, Vol. I, Prentice Hall, Inc., Englewood Cliffs, N.J., 1972

[2] CIESINGER,J.: Generating error recovery in a compiler generating system, Informatik Fachberichte 1, 4. GI-Fachtagung über Programmiersprachen, 1976, 185-193

[3] DIETERICH,E.-W.: Grobstrukturen kontextfreier Grammatiken, Fachbereich Mathematik der TU München, Dissertation, 1976

[4] EICKEL,J.: Methoden der syntaktischen Analyse bei formalen Sprachen, Lecture Notes in Economics and Mathematical Systems, Vol. 78, 1972, 37-53

[5] MEERTENS,L.G.TH., VAN VLIET,J.C.: Repairing the paranthesis skeleton of Algol 68 programs, Stichting mathematisch centrum, Amsterdam, IW 2/73, 1973

[6] VOLLMERHAUS,W.: Die Zerlegung von kontextfreien Semi-Thue-Systemen mit Anwendung auf das Analyseproblem kontextfreier Sprachen, Beiträge zur Linguistik und Informationsverarbeitung, 12, 1967, 23-35

[7] WIRTH,N.: PL36o, A Programming Language for the 36o Computers, JACM 15.1, 1968, 37-74