A Practical Attribute Grammar Circularity Test *

Matthew Belmonte
Computer Science Department
Cornell University
Ithaca, New York 14853-7501

June 1988

Abstract

Efficient implementations for two optimisations to Knuth's attribute grammar circularity test are described. A new method for eliminating useless visits to productions is introduced. This improves upon a somewhat weaker mechanism introduced previously by Deransart *et al.* Data structures and algorithms for graph covering and elimination of redundant unions are discussed and proven correct.

1 Introduction

It has long been argued [6, 2] that circularity tests on attribute grammars arising in practice are tractable, even though circularity testing for attribute grammars in general requires an exponential number of steps [3]. The results described here support this claim. I begin with a short statement of the problem and summarise graph-covering and grammar partitioning optimisations. I then describe a method for using a flow graph defined by the grammar to ascertain the step at which a nonterminal dies, that is, the point in the flow graph after which the nonterminal will never again be referenced. A method for ensuring that no union of dependency graphs is generated more than once is introduced. A method for eliminating useless visits to productions is discussed. This is an improvement upon weak stability [2]. Finally, the complete algorithm is presented along with some practical results.

An attribute grammar is circular if and only if it can generate a derivation tree in which some attribute is circularly defined. Detection of circularities is important because a language processor generated from a circular attribute grammar definition will not work correctly for all inputs. To facilitate explanation

*This work was supported in part by the National Science Foundation under grant CCR-8514862.

of the circularity test, all terminal symbols are assumed to have no attributes. Therefore in the context of this analysis all terminal symbols can be replaced by the null string. Let $G = (N, \emptyset, P, S)$ be a context-free grammar, where N is the set of nonterminals, P is the set of productions, and $S \in N$ is the root symbol. A production $p \in P$ has the form

$$X_{p0} \rightarrow X_{p1} \dots X_{pn_p}$$

where the X_{pi} are (occurrences of) nonterminals. The set of attributes of a nonterminal X is denoted $A(X)$. The circularity test proceeds by computing for each nonterminal X a class $S(X)$ of dependency graphs [1]. Graphs are added until, for each nonterminal X, $S(X)$ contains dependency graphs representing the transitive dependencies of $A(X)$ through every possible derivation tree rooted at X. A general version of the algorithm follows. $D_p[G_1, \dots, G_{n_p}]$ denotes the union of the dependency graph D_p for production p with dependency graphs $G_1, \dots, G_{n_p}$. The transitive closures of these unions are projected onto the attributes of the left-hand-side nonterminal of the production.

for $X \in N$ **do** $S(X) := \emptyset$ **od**;
repeat
    choose production $p \in P$;
    choose dependency graphs $G_1, \dots, G_{n_p}$ from $S(X_{p1}), \dots, S(X_{pn_p})$, respectively;
    $S(X_{p0}) := S(X_{p0}) \cup (D_p[G_1, \dots, Gn_p]^+ \cap A(X_{p0})^2)$;
    if $D_p[G_1, \dots, Gn_p]$; contains a cycle, **then** the grammar is circular;
until no further change possible in any S

([5] contains a complete explanation of the algorithm.)

2 Graph Covering

The most immediate and beneficial optimisation to Knuth's original algorithm is attributed in [2] to Lorho and Pair. Since the fundamental operation on all dependency graphs is union, graphs in a dependency class that are covered by other graphs in the same class can be discarded without affecting the results of the algorithm. In tests using attribute grammars for programming languages (see section 7) about 70% to 80% of all graphs generated were covered. The penalty for this savings in the space used to store dependency graphs and the time used to generate combinations of them is the need to compare every new graph unioned into a dependency class $S(X)$ with $O(|S(X)|)$ old graphs. The comparison operation is accomplished in time proportional to the number of edges by imposing a linear ordering on the set of all possible edges and maintaining each graph as a list of edges in sorted order. This sorting costs nothing since it can be produced as a side effect of the process by which the graph is

[1] This set is denoted $\Sigma(X)$ in [1], $\mathcal{C}(X)$ in [6], $D(X)$ in [2], and $S(X)$ in [5].

generated. A graph is generated by projecting a transitively closed dependency matrix onto the attributes of the left-hand nonterminal of a production.

3 Partitioning the Grammar

Chebotar [1] defines a dependency relation Γ_G on the set of nonterminals and uses it to compute the flow of dependencies through productions. In the graph of Γ_G, vertices can be uniquely labelled by nonterminals and edges can be labelled by productions. It is more convenient, for this implementation, to define a dependency relation Δ on the set of productions, since each iteration must choose a production through which to propagate dependency information. In the graph of Δ, vertices can be uniquely labelled by productions and edges can be labelled by nonterminals.

Definition 1 *$\Delta \subseteq P \times P$ is a dependency relation on productions such that $p\Delta q \iff \exists i_{\in[1,n_q]} X_{p0} = X_{qi}$.*

In other terms, p is related to q if and only if the nonterminal on the left-hand side of p occurs on the right-hand side of q. This definition follows the direction of the flow of dependency information during execution of the circularity test. The information flows right-to-left through productions. Also, define the inverse relation Δ^{-1} that gives a production's pre-image in Δ:

Definition 2 *$\Delta^{-1} \subseteq P \times P$ is an inverse dependency relation on productions such that $q\Delta^{-1}p \iff \exists i_{\in[1,n_q]} X_{p0} = X_{qi}$. Δ^{-1} can also be viewed as a mapping from P into its power set 2^P, so that $\Delta^{-1}(p)$ is understood as $\{q|p\Delta^{-1}q\}$.*

The computations using Δ parallel the computations on Γ_G. The graph of Δ can be partitioned into strongly connected components using a depth-first search [8]. Each of these components constitutes a sub-grammar. The circularity test is run on each of these sub-grammars in turn [1].

Definition 3 *$\Delta_0 \subseteq 2^P \times 2^P$ is a dependency relation on the strongly connected components of Δ such that $A\Delta_0 B$* **iff** *$\exists p_{\in A} \exists q_{\in B} p\Delta q$.*

The graph of Δ_0 has an edge between two components if and only if there is a dependency between those two components in Δ. This parallels the definition of the graph on the strongly connected components of Γ_G in [1]. Upon the completion of the circularity test on a sub-grammar, the dependency classes of all nonterminals that occur in the current strongly connected component and do not label any of its exit edges can be discarded. Because no production (vertex) in the current strongly connected component will ever again be selected, these nonterminals (non-exit edges) will never again be referenced. The nonterminals that do label exit edges must be saved until the completion of the strongly connected components upon which those edges are incident. The order in which the components are processed is established by a topological sort of Δ_0. The resulting ordering is placed in an array DELTA of size $|\Delta_0|$.

Definition 4 *A nonterminal X is* dead *in a component* **iff** *$S(X)$ will never be accessed during or after the processing of that component.*
A nonterminal dies *at a component* **iff** *it is live in that component and dead in the next component in DELTA.*

A mapping DEAD : $\mathcal{Z}^+ \rightarrow 2^N$ is constructed that gives the set of nonterminals that die at each component. This construction is accomplished by recording for each nonterminal X the largest index i into DELTA such that DELTA[i] contains a production in which X appears on the right-hand side. This can be done efficiently during the construction of DELTA. The nonterminals are then sorted into buckets so that DEAD[i] contains the nonterminals whose maximum DELTA index is i. After the completion of component DELTA[i], the dependency classes attached to the nonterminals in DEAD[i] are discarded.

4 Eliminating Old Graph Combinations

One wishes never to construct any union $D_p[g_1, \ldots, g_{n_p}]$ more than once [6], because redundant constructions provide no new dependency information.

Definition 5 *At a given time, a graph $g \in S(X)$ is* fresh *for occurrence X_{pj} of nonterminal X* **iff** *it has been created after the most recent selection of p. g is* stale *for occurrence X_{pj}* **iff** *it existed prior to the most recent selection of production p. A union $D_p[g_1, \ldots, g_{n_p}]$ is* fresh **iff** *it contains a fresh graph. Otherwise, the union is* stale.

A dependency class can be represented as a linked list of dependency graphs. Associated with each occurrence X_{pj} of each nonterminal X is a pointer STALE$_{pj}$ into the dependency class $S(X)$. All fresh graphs precede the position in the list $S(X)$ that is pointed to by STALE$_{pj}$, and all stale graphs are at or beyond this position. Since graphs are always added at the head of the list, all new graphs added to a dependency class $S(X)$ are automatically fresh for all occurrences of X. Initially, all STALE pointers are nil. Immediately after a production has been selected and all fresh unions have been generated, all graphs in the dependency classes of the right-hand-side nonterminals become stale for these nonterminal occurrences.

If a graph in the dependency class of some nonterminal is discarded because it is covered by a new graph that is unioned into the class, then the STALE pointers for all productions in which the nonterminal occurs on the right-hand side must be updated. Each STALE pointer that pointed to the discarded graph must be changed to point just past the discarded graph. (In the algorithm that appears in this paper, this updating operation is considered part of the union

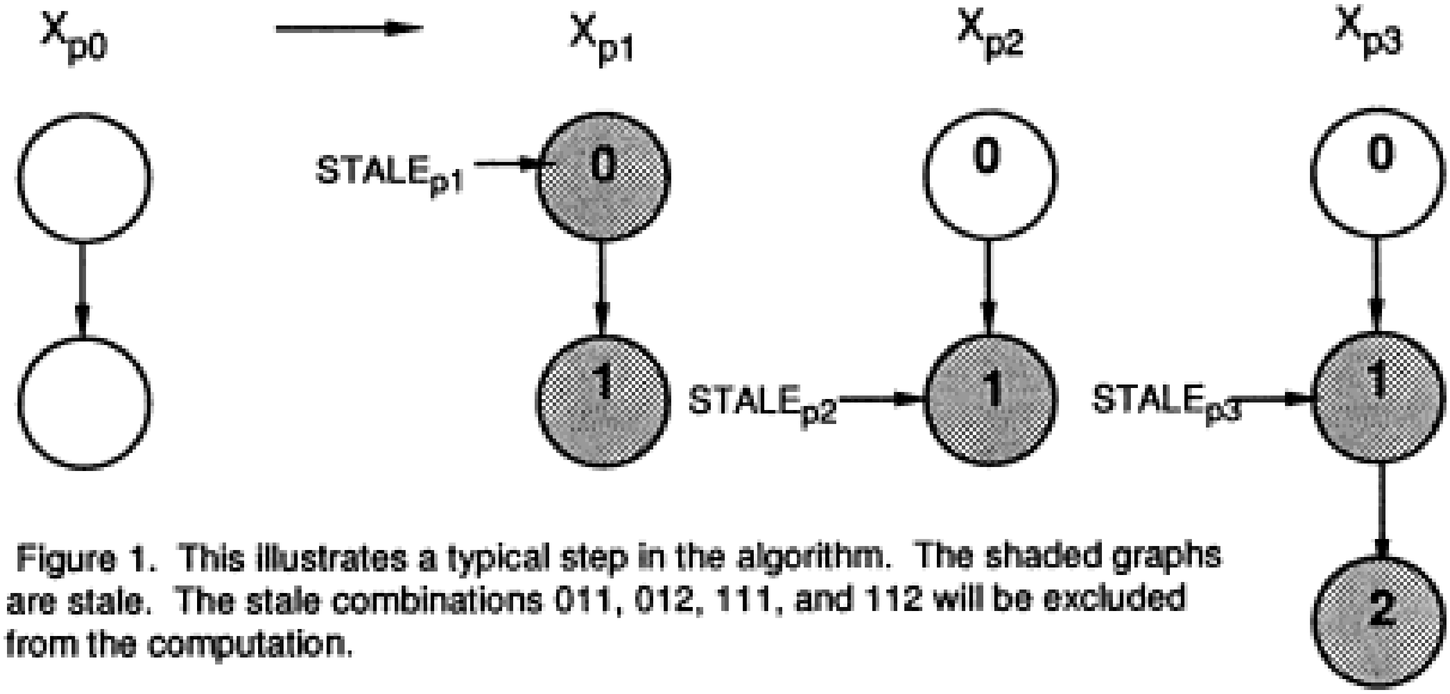

Figure 1. This illustrates a typical step in the algorithm. The shaded graphs are stale. The stale combinations 011, 012, 111, and 112 will be excluded from the computation.

operation.) All stale pointers associated with a production p are discarded upon completion of the strongly connected grammar component that contains p.

During the generation of unions of dependency graphs, the algorithm uses a vector S_p^0 to record the initial elements of the dependency classes of the right-hand-side nonterminals. This vector contains a pointer to the head of each right-hand-side nonterminal's linked list. A working vector S_p takes its initial value from S_p^0 and uses the initial elements recorded in S_p^0 to count through all fresh combinations of graphs. These vectors can become inconsistent with the dependency classes into which they point if production p is directly recursive. In this case, a covered graph in $S(X_{p0})$ which is discarded might be pointed to by S_p^0 or by S_p. Therefore the graph covering algorithm must bump any elements of these vectors which point to a graph that is being discarded. Such an alteration of an element of S_p can cause that element to become stale, so the graph covering algorithm should also update the *number_stale* counter described in the algorithm presented below.

5 An Improvement on Weak Stability

Deransart, Jourdan, and Lorho [2] introduced the notion of weak stability to prevent recomputation of dependency relations for terminal trees. Their version of the circularity test uses an outer loop (loop (1) in the algorithm in the next section) such that within the current strongly connected component, each production is selected once during each iteration of the outer loop. After the i-th iteration, the circularity test has computed dependency relations through all possible attributed tree segments of height bounded by i. The idea behind this optimisation is that tree segments whose leaves are all the null string[2] can generate no new dependency information and can therefore be excluded from the test. Initially, the set $\mathcal{W}$ of weakly stable productions is $\emptyset$. Assume that there are no useless productions. Then the updating of $\mathcal{W}$ after each iteration can be described in terms of the relation Δ by the operation $\mathcal{W} := \mathcal{W} \cup (\{p|\Delta^{-1}(p) \subseteq \mathcal{W}\})$. Note that $\{p|\Delta^{-1}(p) = \emptyset\}$ is exactly the set of null productions. These productions become weakly stable immediately after they are examined.

I introduce the notion of *availability* of a production. This is more powerful than weak stability and includes all the cases in which weak stability applies.

Definition 6 *A production p is* available **iff** *the dependency class of some nonterminal on its right-hand side contains a fresh graph, and the dependency class of each of its right-hand-side nonterminals is nonempty.*

This means that a production becomes unavailable immediately after it is selected, and a production can become available only when a new dependency graph is generated by a production in its pre-image in Δ. This idea of availability was mentioned (but not by any name) in [6], with the difference that all productions were available initially and dependency classes were initialised to $\{\emptyset\}$, the set of the empty graph. These loose initial conditions decrease efficiency, because they allow the algorithm to waste many iterations generating unions of empty graphs. In this version, the set of initially available productions is produced using the following algorithm:

for each null production p **do**
    compute D_p^+ and test it for circularity;
    $S(X_{p0}) := S(X_{p0}) \cup \{D_p^+\}$
od

Following the notation in [6], I denote the set of available productions by R. Denote by $\mathcal{U}_k$ and $\mathcal{W}_k$ the set of permanently unavailable productions and the set of weakly stable productions, respectively, at the end of iteration k. Note that $\forall p_{\in P}(p \in \bar{R} \wedge \Delta^{-1}(p) \subseteq \mathcal{U} \Longrightarrow p \in \mathcal{U})$, since if there is no path of available vertices (productions) in Δ to an unavailable vertex p, then that vertex cannot become available.

Lemma 1
$\forall k \mathcal{W}_k \subseteq \mathcal{U}_k$
Proof (by induction on iterations of loop (1)):
Base case:
All null productions are permanently unavailable, since there are no edges

[2] [2] uses a terminal string instead of the null string. In this context of abstract syntax, the two notions are equivalent.

incident on them in Δ and thus no paths through which dependency information can flow into them. Therefore,
$\mathcal{W}_1 \subseteq \mathcal{U}_1$.
Inductive step:
Assume $\mathcal{W}_i \subseteq \mathcal{U}_i$.

Show $\forall_{p \in P} p \in \mathcal{W}_{i+1} - \mathcal{W}_i \Longrightarrow p \in \mathcal{U}_{i+1}$
Assume $p \in \mathcal{W}_{i+1} - \mathcal{W}_i$

$\Delta^{-1}(p) \subseteq \mathcal{W}_i$, *by definition of weak stability.*
$\therefore \Delta^{-1}(p) \subseteq \mathcal{W}_i$
$\Delta^{-1}(p) \subseteq \mathcal{U}_i$, *by inductive hypothesis and transitivity of* $\subseteq$.
$p \in \bar{R}$ *at some point during iteration* $i+1$, *because selecting* p *will make it unavailable.*
$\therefore p \in \mathcal{U}_{i+1}$

□

This shows that the availability mechanism is at least as powerful as weak stability. The following case illustrates that it is strictly more powerful. Consider some available production $p : A \to X_{p1} \ldots X_{pn_p}$ where some X_{pi} on the right-hand side is not weakly stable. It is possible that all the graphs generated from fresh unions are covered by graphs already present in $S(A)$. (If most of the right-hand-side nonterminals are the left-hand sides of productions that have been unavailable since the previous visit to this production, there is a good chance that this situation will arise.) In this case, $S(A)$ is unchanged, so the productions in $\Delta(p)$ that are unavailable remain unavailable. This saves useless iterations. This interaction with the graph covering optimisation is the reason for availability's superiority to weak stability.

6 The Algorithm

Within a strongly connected grammar component, productions are chosen in a depth-first order to increase information flow. This depth-first ordering is a byproduct of finding the strongly connected components of Δ. The full algorithm is presented below, and a discussion follows.

construct Δ;
construct Δ_0;
let DELTA[*] be a topological sort of Δ_0 and construct DEAD[*];
initialise all $S(X)$ to $\emptyset$;
initialise all STALE_{pj} to nil;
$R := \emptyset$;
$n := 0$;
do $n \neq |\Delta_0| \longrightarrow$
  for each null production p in grammar component DELTA[n] **do**
    compute D_p^+ and test it for circularity;
    $S(X_{p0}) := S(X_{p0}) \cup \{D_p^+\}$;

```
        $R := R \cup \Delta(p)$
    **od**
1:  **do** $R \cap \mathrm{DELTA}[n] \neq \emptyset \longrightarrow$
        choose $p \in R \cap \mathrm{DELTA}[n]$;
        $R := R - \{p\}$;
        **for** $j \in [1, n_p]$ **do**
            $S_p^0[j] := S(X_{pj})$
        **od**;
        $S_p := S_p^0$;
        number_stale := 0;
2:      **do** $S_p[\text{number\_stale} + 1] = \mathrm{STALE}_{p,\text{number\_stale}+1} \longrightarrow$
            number_stale := number_stale + 1
        **od**;
        {invariant: All unions involving fresh graphs from $S(X_{p1})..S(X_{p,\text{number\_stale}})$ have been generated.}
3:      **do** number_stale $\neq n_p \longrightarrow$
            $D := D_p$;
            $j := 1$;
4:          **do** $j \leq n_p \longrightarrow$
                $D := D \cup S_p[j]\uparrow$.graph;
                $j := j + 1$
            **od**;
            compute $D^+$ and test it for circularity;
            let $g$ be the projection of $D^+$ into $A(X_{p0})$;
            $S(X_{p0}) := \mathrm{COVER}(S(X_{p0}) \cup \{g\})$, maintaining consistency of STALE, $S_p^0$, and $S_p$;
            $j := n_p$;
5:          **do** $j > 1 \wedge S_p[j]\uparrow$.link = nil $\longrightarrow$
                {$S_p[j]$ has reached the end of its list, so reset it.}
                $S_p[j] := S_p^0[j]$;
                $j := j - 1$
            **od**;
            $S_p[j] := S_p[j]\uparrow$.link;
6:          **do** number_stale $< n_p$ **cand** $S_p[\text{number\_stale} + 1] = \mathrm{STALE}_{p,\text{number\_stale}+1} \longrightarrow$
                number_stale := number_stale + 1
            **od**;
            {If all pointers have become stale, then force an exit from the loop:}
7:          **if** $j \leq$ number_stale $\longrightarrow$ number_stale := $n_p$
            $[\!]$ $j >$ number_stale $\longrightarrow$ skip
            **fi**
        **od**;
        if $S(X_{p0})$ has changed, then $R := R \cup \{q | p\Delta q \wedge \forall i_{1 \leq i \leq n_q} S(X_{qi}) \neq \emptyset\}$
    **od**;
    discard all $\{S(X) | X \in \mathrm{DEAD}[n]\}$;
    $n := n + 1$
```

od

I wish to show that loop (3) computes all fresh unions. Loop (4) computes the union of the graphs pointed to by the vector S_p, so it is sufficient to show that (4) is executed for every fresh combination of S_p.

Assign a number k to each element of $S(X_{pj}), 1 \leq j \leq n_p$, such that each list $S(X_{pj})$ is arranged sequentially in order of increasing k-value. Then combinations of graphs can be mapped onto sequences of k-values with one k-value drawn from each list. The k-values are like digits on an odometer or clock display, and the process of generating combinations of them is like rolling those digits in sequence, with the digits arranged from right to left in order of increasing significance. (The analogy does not hold perfectly, because the sizes of the classes are not all the same.) So loop (5) generates a combination of each $S_p[i]\uparrow$.graph with all $S_p[j]\uparrow$.graph such that $i < j \leq n_p$.

Loop (2,6) ensures that number_stale is the index of the rightmost element of a contiguous block S_p[1..number_stale] of elements that have become stale. Therefore, at each iteration of (3), all unions involving fresh graphs from S_p[1..number_stale] have been generated. The conditional statement at (7) is executed in the case that some entire dependency class S_p[number_stale + 1] is fresh. It resets number_stale after the leftmost such class is fully processed. Loop (3) terminates only when all elements of S_p have become stale. Therefore, all fresh combinations are generated.

Suppose some stale union is generated. Either (2) or (6) has just been executed when the loop guard for (3) is evaluated. Therefore, number_stale is always current when the guard is evaluated. No stale union can be generated while some element of S_p is fresh. Therefore number_stale $= n_p$ when the first stale union is generated. But if number_stale $= n_p$, then (3) terminates without generating any more unions. Therefore, no stale union can be generated.

Lemma 2 *Loop (3) computes all fresh unions and no stale unions.*
Proof: by the above discussion. □

Lemma 3
All possible unions of dependency graphs are generated exactly once.
Proof (by induction on iterations of (3)):
Base case:
Initially, there are no dependency graphs and therefore no possible unions.
Inductive step:
Assume *all possible unions up through the i-th iteration have been generated.*
All fresh unions on iteration $i+1$ are generated, by lemma 2.
$\therefore$ *All possible unions up through the $i+1$-th iteration are generated, by the definition of freshness.*
□

7 Practical Results

The circularity test described here has been run on several attribute grammars distributed with the Cornell Synthesizer Generator [7]. These test cases included grammars for two large programming languages, C and Pascal, as well as Toy, a small programming language used in compiler construction courses at Cornell, EFS, an environment for defining and reasoning in formal systems, IDEAL, a geometric picture specification language, and PV, a program correctness verifier. The Pascal and Toy grammars include code generation, while the C grammar is only a semantic checker. The practical effect of the optimisations described here is that the circularity test on most attribute grammars runs in about twenty seconds of real time on a Sun-3/60.

Räihä and Saarinen [6] define a grammar to be $G(k)$ if the grammar can be tested for circularity such that the size of every dependency class is bounded by k at all times during the execution of the circularity test. (The $G(1)$ grammars are a subset of the class of absolutely noncircular grammars.) During the circularity testing process on a $G(k)$ grammar, it is possible for dependency classes to grow larger than k because of the order in which productions are selected. Hence the algorithm does not necessarily find the tightest possible upper bound on k. The grammar for C is $G(2)$, and the grammar for Pascal is $G(5)$. These figures are in general agreement with those reported for the programming language examples in [6] and [2]. [2] gives a generalisation of the G notation, and a plethora of practical results for various combinations of optimisation strategies. The size distributions for dependency classes have large peaks at low sizes and decrease rapidly to almost zero. The figures given in the table illustrate this claim. d is a variable whose value is the maximum size attained by a dependency class. Thus if for some grammar, for some k, $d_{\max} = k$, then that grammar is $G(k)$. Because there is one dependency class for each nonterminal, $|N|$ is the number of data points for d. $\#(k)$ is the number of classes whose maximum size is k. A comparison of the $\#(k)$ values for very small k with $|N|$ reveals a strong tendency toward very small dependency classes; $\#(1)$ approaches $|N|$. These figures, which I take to be typical of those arising in practice, demonstrate the somewhat misleading nature of $G(k)$ size bounds with respect to the quantification of the practical complexity of circularity testing. All grammars of complexity greater than $G(3)$ tested had only one graph in the largest class. In order to capture the complexity of the process of circularity testing for a language, it is necessary to consider not the maximum dependency class size but instead the distribution of dependency class sizes.

Grammar	$\|N\|$	$d_{\max}$	$E(d)$	S.D.	#(1)	#(2)
C	51	3	1.12	0.427	47	2
Pascal	91	5	1.07	0.440	88	2
Toy	15	6	1.40	1.254	13	1
IDEAL	21	4	1.14	0.639	20	0
EFS	41	4	1.17	0.580	37	2
PV	18	3	1.22	0.533	15	2

The average strongly connected grammar component size for all grammars tested was less than 1.35. This reflects the largely hierarchical structure of most practical attribute grammars. Two larger components in most programming languages are the sub-grammar that generates expressions and the sub-grammar that generates statements. These are groups of mutually recursive productions.

In every grammar there were many cases in which a visit to a production that was unavailable but not weakly stable was avoided. The number of occurrences of this case ranged from 40% of the number of productions in EFS to 55% in PV, 57% in IDEAL, 64% in Pascal, 77% in C, and 82% in Toy. These results demonstrate the practical superiority of the availability mechanism over weak stability.

References

[1] K. S. Chebotar. "Some Modifications of Knuth's Algorithm for Verifying Cyclicity of Attribute Grammars." *Programming and Computer Software* 7:1, 58-61 (January 1981).

[2] P. Deransart, M. Jourdan, B. Lorho. "Speeding up Circularity Tests for Attribute Grammars." *Acta Informatica* 21, 375-391 (December 1984).

[3] M. Jazayeri, W. F. Ogden, W. C. Rounds. "The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars." *Communications of the ACM* 18:12, 697-706 (December 1975).

[4] D. E. Knuth. "Semantics of Context-Free Languages." *Mathematical Systems Theory* 2:2, 127-145 (June 1968).

[5] D. E. Knuth. "Semantics of Context-Free Languages: Correction." *Mathematical Systems Theory* 5, 95-96 (March 1971).

[6] K. Räihä and M. Saarinen. "Testing Attribute Grammars for Circularity." *Acta Informatica* 17:2, 185-192 (1982).

[7] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*, 2/e. Cornell University Department of Computer Science (1987).

[8] R. E. Tarjan. "Depth-First Search and Linear Graph Algorithms." *SIAM J. Computing* 1:2, 146-160 (June 1972).