An Introduction to Refinement Metrics: Assessing a Programming Language's support of the Stepwise Refinement Process

Robert G. Reynolds, Jonathan I. Maletic
Wayne State University
Computer Science Department
Detriot, Mi 48202

1. Introduction.

Much of the software life cycle is realized through programming languages. The structure of the particular programming language used to implement a system can influence the effort expended at some of the different phases of the life cycle. This impact is particularly evident in phases such as maintenance that deal with completed code. For example, the presence of certain code structuring units in a language is generally thought to improve the readability of programs that employ them [1].

However, the impact that language structure has on the design process is less clear cut. The stepwise refinement of pseudocode is a common mechanism for program design in a variety of languages. Yet, the effects of a language's structure on the refinement process has not been quantitatively documented. The need for such assessment is particularly acute in light of the fact that many new program design languages (PDL) support the syntactic constructs of a particular target language. For example, the Ada programming language is the basis for at least six available or planned program design languages [2].

The general question addressed in the paper is: how does the syntactic structure of a program design language affect the decision making effort associated with the stepwise refinement process. A model of the stepwise refinement process is presented that explicitly ties the grammatical structure of the pseudocode design language into the stepwise refinement activity. This allows the development of refinement metrics that assess the decision making complexity of a refinement step in terms of the underlying syntactic structure of the grammar for the program design language.

2. Metrics to Measure Language Support for the Stepwise Refinement Process.

Historically, programming languages have been compared from several different perspectives. These comparisons were based upon language properties such as syntactic structure [3], semantic structure [3], psychological complexity [4,5,6], and relevance to particular application domains [7]. Here, the comparison between languages will be made using syntactic complexity, as measured in terms of the context free grammar for the languages. The motivation for using context free grammars is that the stepwise refinement process takes place in the context of a language's syntactic structure. That is, the replacement of a stub in a pseudocode program is constrained by the grammar in the target language. The nature of these constraints will be described in terms of the following model of the stepwise refinement process.

In this model it is assumed that the program design language employed in the stepwise refinement process is based upon a context free grammar for the target language. This is consistent with our current objective of measuring target language support for the stepwise refinement process. However, in general the model does not require that the program design language correspond precisely with the grammar for the target language. A context free language is defined as G =<T,N,P,S> where T is the set of terminals, N is the set of nonterminals, P is the set of productions of the form A::=w where A is an element of N and w is in V* where the vocabulary V = (T $\cup$ N), and S is the start symbol. A pseudocode program can therefore be considered as a collection of terminal symbols in the target language along with a collection of stub names. Each stub in the pseudocode corresponds to a high level implementation goal that still needs to be achieved by the designer. It is also assumed that the designer associates each high level implementation goal with a low level goal of implementing a corresponding language structure taken from the set of nonterminals in the grammar for the language. As a result the stepwise refinement process can be viewed as taking place at both a semantic level and a syntactic one.

A refinement step in the model is the replacement of one of the labeled nonterminals with a segment of code brought about by a finite, non-empty sequence of production applications in the grammar. In other words, it is presumed in the model that the designer has internalized at least a subset of productions in the grammar, and uses them as a basis for the introduction of new terminal symbols and stub names into the pseudocode. Such an internalization has also been suggested

by MacLennan [3].

The problem solving paradigm that underlies the stepwise refinement process is that of divide and conquer or problem reduction. This paradigm involves the decomposition of a problem into constituent subproblems, each of which is less difficult to solve than the original. Here, each stub in the pseudocode can be associated concurrently with a goal to be solved at both the semantic and syntactic level. As a result, the problem reduction activity can take place at one level or the other or both. Since the focus of this paper is on the syntactic level, it will be assumed that each new stub or stubs produced by a refinement step will be associated semantically with a simpler goal.

At the syntactic level, support of the problem reduction paradigm means that each new stub produced by a refinement activity will be associated with an implementation task that is smaller than that of the parent. The size of the task is a function of the syntactic class of the stub. If a refinement step is considered in the model to be a composition of productions in the grammar then what combination of productions will be sufficient to generate refinement steps that satisfy the problem reduction constraint here?

In order to answer this question, three basic categories of productions sequences are defined: directly recursive, indirectly recursive, and non-recursive. A production sequence is said to be recursive if there exists one or more production applications in the sequence such that a nonterminal, A, on the left hand side of a production in the sequence is generated on the right hand side (i.e. A =>+ w1Aw2, where w1,w2 are elements of V*). If a production sequence is recursive and the number of production applications needed to make it recursive exceeds one then the sequence is said to be indirectly recursive. For example, if A::=P and P::=A are productions in a grammar, then the sequence A=>P=>A is indirectly recursive. If the production sequence is recursive and the number of production application needed to make it recursive is no more than one, then this sequence is said to be directly recursive. A production sequence is non-recursive if it is neither directly or indirectly recursive.

If a refinement step is modeled by a production sequence from the grammar for the target language, it can be classified in terms of that sequence. In general, there are often many ways of describing the syntactic structure of a refinement step, i.e. there can be more than one way to derive a particular set of terminal symbols. Here, a refinement step is non-recursive if there exists a non-recursive sequence of productions that allow one to derive the replacement code from the generating non-terminal. A refinement step is said to be recursive otherwise. It is directly recursive if the length of the shortest recursive productions is 1, and indirectly recursive otherwise.

Given the above definitions, one can now describe different design paradigms in terms of refinement sequences. When all of the refinement steps used in the implementation of a code module are non-recursive, then the refinement sequence is said to be monotonic decreasing. That is, each time a stub is refined it is decomposed into stubs whose nonterminal classes correspond to smaller syntactic implementation tasks. Since there is no recursion here, each new stub must be at least one production application closer to implementation.

When all of the refinement steps used in the implementation of a code module are either directly recursive or non-recursive, the sequence is monotonic non-increasing. Here, the refinement of a stub will produce a stub of less than or equal syntactic complexity.

Both of the above categories of sequences place constraints on the size of the resulting nonterminal classes and are examples of constrained refinement sequences. Unconstrained refinement sequences contain at least one refinement step that is indirectly recursive. As a result, the syntactic complexity of the nonterminal classes produced as the result of a refinement do not necessarily have to decrease.

Intuitively, one expects that the complexity of the decision making process associated with an unconstrained refinement sequence should exceed that for the constrained case. In order to quantitatively describe the extent to which this is the case, a measure of decision making complexity for a grammar must be introduced. This is a graph theoretic measure that operates on a graph that contains the set of productions that can participate in the particular class of refinement sequences being assessed. For any given grammar, one can construct three separate graphs, one to describe the allowable productions for each of the three categories of design paradigms given above. The graph for the unconstrained refinement sequence will be discussed first, followed by that for the constrained non-monotonic case, and finally that for the monotonic decreasing case.

The graph theoretic model for the unconstrained case represents, for each nonterminal class in the language, the various decomposition products that can result based upon applications of productions in the grammar. In the unconstrained model all decompositions associated with productions in the grammar are allowed. The set of possible node labels for the graph in this corresponds to V U {.}. Each node labeled by an element from V represents either a terminal or nonterminal symbol that can be produced as the result of refining a stub associated with a particular nonterminal class. A directed arc from a node x to a node y exists if there is a production in the grammar which has the nonterminal label for node x on the left hand side and the terminal or nonterminal symbol for y on the right hand side. If a nonterminal is decomposed into a collection of more than one element from V, the broadcast symbol "." is used to denote the presence of multiple arcs that derive from the same production. The use of this symbol precludes the need to label each arc with the name of its associated production. An example of the unconstrained decomposition graph for a simple programming language is given in figure 1. This grammar is a subset of PASCAL and is taken from Backhouse [10]. The BNF

description for the grammar is given in table 1.

In order to use the grammar to support a monotonic non-decreasing refinement sequence, the designer must ignore all opportunities for indirect recursion. This is equivalent to removing from each production in the grammar those arcs supporting indirect recursion. For a production in the grammar this is done by removing any arc from the production to any term that labels a node lying along a path from the start node to the node from which the production emanates. It is replaced by an arc pointing to "nil". The resultant decomposition graph for the example grammar is given the figure. Five arcs were removed from the original unconstrained decomposition graph. The graph still contains five directly recursive productions since they are still allowed under the current constraints. This new graph is called the constrained decomposition graph for the grammar since it enforces the constraints on indirect recursion required by the monotonic nonincreasing approach to stub refinement at the syntactic level.

In order to support a monotonic decreasing refinement sequence a designer needs to ignore the recursive relations present in the graph as well. This entails the removal of any arc emanating from the production for a node that points back to the node. For the example grammar this results in the removal of five arcs (see figure). It is called the constrained decomposition DAG for the grammar since all cycles in the graph have been removed in order to support only a monotonic decreasing sequence of refinement steps at the syntactic level.

For each of the three graphs described above it is possible to estimate the amount of decision-making effort needed to implement each syntactic class based upon its position in the graph. The estimate will represent the effort expended in the worst case for each situation. The measure of effort employed here is called refinement volume. It will be defined first in terms of the constrained decomposition DAG.

The constrained decomposition DAG contains only those productions that support the divide and conquer paradigm relative to a particular measure of effort, refinement volume. Refinement volume is computed in terms of two other metrics, refinement breadth and refinement depth. These three measures are called refinement metrics and will now be described. The first metric of concern will be refinement depth. Refinement depth for a given nonterminal corresponds to the depth of the subtree in the constrained decomposition DAG that supports the decomposition of a given nonterminal. This subtree is called the support subtree for that nonterminal. The support subtree is the set of all nodes and arcs that are traversed along at least one path from the root node of that tree to a leaf node. In order to describe the depth, a measure of distance is needed. The distance measure d is defined for two nodes in the graph, u and v, as the length of the shortest path between them, where the length of the path is the number of arcs traversed. In a connected graph such as this, distance is a metric [8]. This means that for all nodes u, v, and w: 1) $d(u,v) > 0$, with $d(u,v) = 0$ if and only if u=v. 2) $d(u,v) = d(v,u)$. 3) $d(u,v) + d(v,w) > d(u,w)$. The depth D of a connected graph is $\max(d(u,v))$ for all nodes u and v in the graph [8]. Given the structure of the support subtree for the nonterminal i, the depth must correspond to the path from the root node for the subtree to a terminal node of maximum length. Intuitively, this longest path corresponds to the maximum number of production applications required to implement the most complex syntactic subgoal associated with the implementation task for the nonterminal i.

Another parameter that will be of use in describing the support subtree for a given nonterminal i is refinement breadth. Refinement breadth corresponds to the diameter of the support subtree and is defined as the set of unique productions associated with the support subtree of i. Intuitively this represents the set of unique productions that are reachable via the decomposition of a nonterminal class associated with a stub in the pseudocode.

Refinement breadth and depth can be combined to produce a measure of the amount of information that a programmer needs to provide in order to replace a stub of a given nonterminal class with completed code. The measure is called refinement volume and for a nonterminal i is expressed as: refinement depth(i) * log2(refinement breadth(i)). Intuitively this corresponds to the number of bits needed to encode the set of productions required to implement the longest sequence of productions associated with the implementation of a given nonterminal class. The number of bits required to uniquely identify each of the productions in the support subtree for i is represented as log2(refinement breadth(i)), and the number of productions is given by the refinement depth for i.

Another way to view refinement volume is to see it as an index of the amount of effort required to attain the goal of implementing a stub from the nonterminal class i. From this perspective refinement depth corresponds to the number of decisions that need to be made in the worst case, while the second term is a worst case estimate of the number of productions from which the designer can select from at each step. It can be seen as a worst case estimate since not all of the original productions will be candidates for selection at any given step.

The constrained decomposition DAG possesses certain regularities with respect to these measures (i.e. refinement depth, breadth, and volume for a nonterminal class decreases monotonically with distance down any path from the root to a leaf). These trends can be observed in the figure where each node is labeled by a triple of values, refinement depth, breadth, and volume respectively.

The refinement metrics can now be extended to describe the other two graphs in the following way. Any arc in either graph that extends from a node i to a node j, such that the refinement volume of node j in the constrained decomposition DAG is greater that or equal to that of i is labeled as nondecreasing. All other arcs are labeled decreasing. A stub of nonterminal class i can be replaced by a stub from class j if there exists a walk in the graph from i to j. The arcs along that walk can be of both the constrained and unconstrained type. A

walk in the graph that contains both decreasing and non-decreasing arcs is said to be a mixed walk.

Mixed walks are classified here in terms of the following features: 1) The number of non-decreasing arcs that occur in the walk. The number that occurs is termed the order of the walk; 2) The relative positioning of these non-decreasing arcs in the walk. For example, if all the productions corresponding to non-decreasing arcs occur only at the beginning of the sequence , the walk is said to be in prefix normal form.

Here, the refinement metrics will be extended to deal with first order mixed walks. First, refinement depth can be computed for a given nonterminal class i as follows. 1) First determine which productions in the grammar with nonterminal i on the left hand side have nonterminals on the right hand side with constrained refinement depths greater than or equal to i. 2) If this set is non-null then select the production with the nonterminal class, j, of greatest refinement depth in the constrained DAG. 3) Use of this production at the beginning of the sequence will produce a nonterminal that is farther from completion than i was. Since the remaining productions must be constrained, they will be taken from the support subtree for j in the constrained decomposition DAG. The resultant refinement depth for i will then be the refinement depth for j plus 1. The 1 corresponds to the application of the first production that generated j.

Refinement breadth and volume can be expressed in a similar fashion since they can be viewed as deriving from values for nonterminal class j. Refinement breadth is the refinement breadth for j plus the number of productions with i on the left that have j on the right. This latter term corresponds to the number of ways that one can derive j in a decomposition from i. Refinement volume for i just uses the above values in the same expression as for the constrained case.

Given the above definitions, refinement metrics can be calculated for both the constrained and unconstrained refinement graphs. The resultant values in each case are displayed in the figure. A statistical summary of the refinement metrics values for each of the three situations using the example grammar is given in table 2. These statistics include the minimum, maximum, mean, and standard deviation for each metric relative to the 29 nonterminals in the language. Note that while the maximum depth of a nonterminal is 17 (for the nonterminals of class program) in all three graphs, the average depth is considerably less. Also, as constraints on the productions are relaxed the values for the metrics do increase as expected. The magnitude of the increase is rather small in this instance due to the lack of substantial recursion in this simple grammar.

3. Conclusion.

This paper introduces a new level in the description of the stepwise refinement process. The traditional level, normally called the semantic level, is viewed as the decomposition of a stub associated with a high-level implementation goal into a set of new stubs, where each resultant stub is related to a smaller goal than its parent. The new level introduced is the syntactic level. In this level the task concerns the implementation of a nonterminal class associated with a given stub. The effort of implementing the task for a nonterminal can be expressed in terms of the production rules in the underlying grammar. The refinement metrics directly reflect this effort.

The refinement metrics are currently being applied to commersially avaialbe programming languages to assess thier support of the stepwise refinement process. Also an empirical study of refinement sequences are being made to determine how much human programmer may follow these paradigms.

Refinement metrics are also currently being used as a heuristic to drive the PM plan compiler system [9]. The goal of this system is to acquire program planning knowledge from existing program libraries.

4. References.

[1] H. Mills, "Stepwise Refinement and Verification in Box-Structured Systems", IEEE-Computer, Vol. 21, No. 6, 9 June 1988, pp. 23-37.

[2] D. Berry, N. Yavne, M. Yavne, "Application of Program Design Language Tools to Abbott's Method of Program Design by Informal Natural language Descriptions", Journal of Systems and Software, Vol. 7, No. 3, Sept., 9 1987, pp. 221-247.

[3] B. MacLennan, "Simple metrics for Programming Languages", Information Processing and Management, Vol. 20, No. 1/2, January, 1984, pp. 209-221.

[4] J.D. Gannon and J.J. Horning, "Language Design for Programming Reliability", IEEE-SE, Vol. 1, No. 2, 1975, pp. 179-191.

[5] J. D. Gannon, "An Experimental Evaluation of Data Type Conventions", CACM, Vol. 20, No. 8, 1977, pp. 584-595.

[6] B. Schneiderman, Software Psychology:Human Factors in Computer and Information Systems, Winthrop, Cambridge, Mass., 1980.

[7] J. E. Sammet, "Problems in, and a Pragmatic Approach to Programming Language Measurement", AFIPS Fall Joint Computer Conference, 1971, pp. 243-251.

[8] F. Harary, Graph Theory, Addison-Wesley Press, Reading, Mass., 1969.

[9] R. Reynolds, J. Maletic, S. Porvin, "PM: A Metrics Driven Plan Compiler", In the Proceedings for IEEE Workshop on Tools for A.I., Computer Society Press, Washington D.C. 1989.

[10] R. C. Backhouse, Syntax of Programming Lanugauges. Theroy and Practice, Prentice-hall International, 1983.

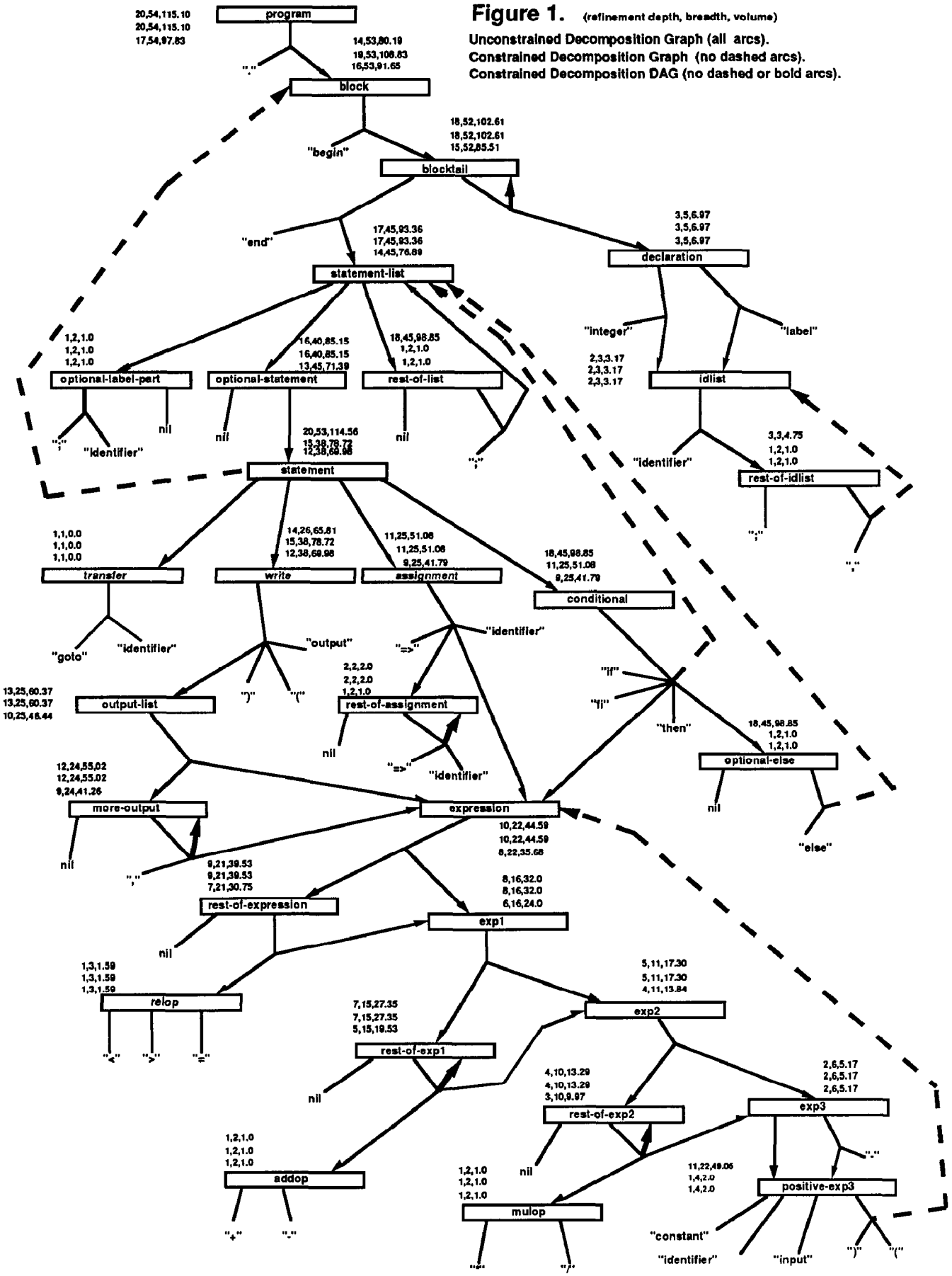
Figure 1. (refinement depth, breadth, volume)
Unconstrained Decomposition Graph (all arcs).
Constrained Decomposition Graph (no dashed arcs).
Constrained Decomposition DAG (no dashed or bold arcs).
program
block
blocktail
statement-list
declaration
optional-label-part
optional-statement
rest-of-list
idlist
statement
rest-of-idlist
transfer
write
assignment
conditional
output-list
rest-of-assignment
optional-else
more-output
expression
rest-of-expression
exp1
relop
rest-of-exp1
exp2
rest-of-exp2
exp3
addop
mulop
positive-exp3
"begin"
"end"
"integer"
"label"
"identifier"
"goto"
"output"
"=>"
"if"
"then"
"else"
"constant"
"input"
nil

<program>	::=	<block>.
<block>	::=	**begin** <blocktail>
<blocktail>	::=	<declaration> <blocktail> | <statement list> **end**
<declaration>	::=	**label** <idlist> | **integer** <idlist>
<idlist>	::=	<identifier> <rest of list>
<rest of idlist>	::=	; | , <idlist>
<statement list>	::=	<optional label part> <optional statement> <rest of list>
<optional label part>	::=	<empty> | <identifier>
<optional statement>	::=	<empty> | <statement>
<rest of list>	::=	<empty> | ; <statement list>
<statement>	::=	<assignment> | <transfer> | <conditional>| <write> | <block>
<assignment>	::=	<expression> => <identifer> <rest of assignment list>
<rest of assignment list>	::=	<empty> | => <identifier> <rest of assignment list>
<transfer>	::=	**goto** <identifier>
<conditional>	::=	**if** <expression> **then** <statement list> <optional else> **fi**
<optional else>	::=	<empty> | **else** < statement list>
<write>	::=	**output (** <output list> **)**
<output list>	::=	<expression> <more output>
<more output>	::=	<empty> | , <expression> <more output>
<expression>	::=	<exp1> <rest of expression>
<rest of expression>	::=	<empty> | <relop> <exp1>
<exp1>	::=	<exp2> <rest of exp1>
<rest of exp1>	::=	<empty> | <addop> <exp2> <rest of exp1>
<exp2>	::=	<exp3> <rest of exp2>
<rest of exp2>	::=	<empty> | <mulop> <exp3> <rest of exp2>
<exp3>	::=	<positive exp3> | - <positive exp3>
<positive exp3>	::=	**input** | <identifier> | <constant> | (<expression>)
<relop>	::=	< | > | =
<addop>	::=	+ | -
<mulop>	::=	* | /

Table 1. BNF of Simple Programming Languge.

		Minimum	Maximum	Mean	Standard Deviation
Refinement Volume	Constrained Decomposition Tree	0	96.907	31.943	31.415
	Constrained Decomposition Graph	0	96.907	32.689	31.746
	Unconstrained Decomposition Graph	0	98.725	39.617	34.215
Refinement Depth	Constrained Decomposition Tree	1	17	6.7586	5.3829
	Constrained Decomposition Graph	1	17	6.8966	5.4271
	Unconstrained Decomposition Graph	1	17	8.1034	5.6716
Refinement breadth	Constrained Decomposition Tree	1	52	19.241	16.771
	Constrained Decomposition Graph	1	52	19.517	16.879
	Unconstrained Decomposition Graph	1	52	22.966	18.064

Table 2.
Refinement metric values for each of the three graphs for the example grammar.