

Tree-affix dendrogrammars for languages and compilers

by

Frank DeRemer and Richard Jullig
University of California
Santa Cruz, California 95064

ABSTRACT

Research in progress is reported regarding a variation on attribute and affix grammars intended for describing the "static semantic" or "context-sensitive syntactic" constraints on programming languages. The grammars are oriented toward abstract-syntax trees, rather than concrete-syntax strings. Attributes are also trees (only) and predicates are simply nonterminals, defined just as other nonterminals are, NOT in some extra-grammatical way. Moreover, trees are allowed as decorations on trees. Thus, the formalism is completely self-contained. The grammars are proposed for language specifications in reference manuals and for the automatic generation of practical compiler modules. Such a module is given an abstract-syntax tree, analyses it, and produces the checked, decorated tree as its result.

Key words and phrases: static semantics, context-sensitive syntax, attribute grammars, affix grammars, abstract syntax, concrete syntax, language specification, compiler generation, translator writing system.

Introduction

Brief history. In 1968 Donald Knuth invented attribute grammars to describe the semantics of context-free languages [Knu 68]. Basically these are context-free grammars (CFGs) extended by the addition of "attributes" to the nonterminals. Each attribute may take on any value of some given data type, and the attribute values in any given production p must be related as specified by some equations associated with p . These equations may involve arbitrary functions on combinations of the attribute data types involved.

The attribute dependences imply a graph associated with each derivation tree of the underlying CFG. The problem of deciding whether a given attribute grammar defines circular graphs, and thus sentences with undefined semantics, is very difficult [JOR 74]. However, given that the grammar has been confirmed not to produce circularities, the attribute dependencies imply a deterministic processor to compute the attributes and thus the semantics of any given sentence. In effect such a processor flows attribute values along the edges of the dependency graph until all values are computed and all equations have been satisfied. Attributes that flow down the tree are called "inherited"; those that flow upward are called "synthesized" or "derived". Functions involved in evaluating attributes are defined outside the grammar.

It can be determined, by analysis of the grammar, how many passes around the derivation tree of a general sentence will be necessary to evaluate the attributes [Boc 76]. Moreover, under stringent conditions it is even possible to compute the attributes during parsing [Wat 74]. This, in turn, has led to the (anti-modular) idea of letting the attributes influence the parsing [Wat 74] [W&M 79] [M&J 80].

In 1971 Kees Koster invented affix grammars, based on two-level grammars [Van 75]. Affix grammars are much like attribute grammars but were intended primarily for the purpose of describing the context-sensitive aspects of programming languages [Kos 71], called by some the "static semantics". Affix grammars have parameters, called "affixes", associated with the nonterminals. The effects of the "semantic equations" of attribute grammars are achieved via "predicate nonterminals" that generate the empty string while imposing a constraint, defined outside the grammar, on the values of the affix variables.

The affix values are specifically restricted to flow from left to right by restricting the affixes to depend only on others to their left in each production. With the addition of an $LL(k)$ constraint on the underlying CFG, it is always possible to implement the grammar as a recursive-descent

parser with value parameters in place of "inherited affixes" and result parameters for "derived affixes". The parser simply calls predicates at appropriate times in the parsing process to enforce context-sensitive constraints and to compute the affix values. Watt has extended this idea to LR(k) parsers [Wat 74].

In 1979 Watt and Madsen combined some of the best ideas from attribute, affix, and two-level [Van 75] grammars to form a very civil "extended attribute grammar" (EAG) [W&M 79]. Two advantages of EAGs are that they are easy to conceptualize as generative systems and that they are relatively compact and easy to follow. However, they do not differ sufficiently for our purposes here to warrant a description.

Our point of view. It is our thesis that language descriptions can and should be modularized, just as large programs can and should be, by the principles of structured programming. Proper modularization will occur when the "natural fracture planes" are found in language descriptions, and correspondingly, in compiler structure. Not all such boundaries have yet been found. Indeed, all practical compilers and language definitions to date are either large and monolithic, or some or all of their components have messy interfaces and/or far too many interconnections.

We also believe that restrictive formalisms help define such boundaries. For example, lexical grammars define scanners based on finite-state technology, and phrase-structure grammars define parsers based on deterministic pushdown technology. The natural fracture plane between these two is characterized by a language of token sequences. The modularization is effective at reducing the total number of states in the compiler "front end" and at enhancing the comprehensibility of the total language definition. The limited technology at each level restrains us from being overly ambitious at that level.

It must be emphasized, however, that these are only restraints. It is our opinion that even now CFGs are being overused. For example, the Algol 60 grammar tried to describe type checking, a thoroughly context-sensitive issue, and became ambiguous for its inadequate efforts. This "error" is repeated in all too many grammars modeled on that one. In general, we find that so many minor restrictions are typically "wired" into CFGs that they are much more difficult to comprehend than is necessary. The Ada grammar is a recent case in point [Ada 79].

We propose instead to use CFGs primarily to generate all desired programs and to associate with each the appropriate phrase structure. Never mind undesired programs at this level. The formalism is not powerful enough for such screening, nor should it be. It should restrain our ambition and focus the design. But the problem is bigger than that.

The reason CFGs are overused is precisely that we lack acceptable formalisms for capturing the context-sensitive constraints and, separately, the dynamic semantics, in a modular way. Stated another way, given only tools for part of the language design process, we typically start off with an entirely wrong focus.

It is furthermore our thesis that the abstract syntax of a language [McC 62] is THE place to start when designing or learning it, and that the "language" of abstract-syntax trees (ASTs) characterizes this level quite naturally. Thus, since an AST, or a linearization of it, is the natural output of a parser, we argue that context-sensitive constraints should be addressed to ASTs, rather than to concrete-syntax strings. The concrete syntax is complex enough already, where operator precedence, bracketing, noise words, and the like, are enough to contend with. Rather than bog down an already overloaded CFG, we propose to adapt the attribute/affix technology to trees, ASTs. What is needed is an "attributed dendrogrammar" that generates a "dendrolanguage". (Greek "dendron" means "tree".)

Finally, it is our thesis that "decorated" trees serve well as the intermediary between the context-sensitive syntactic and dynamic semantic levels of language specification/processing [Cul 73]. By "decorations" we mean, for examples, explicit links from uses of identifiers back to the subtrees that represent their declarations, links from calls to the procedures called, from returns to the procedure returned from, from exits to the loops exited from, etc. In general, the idea is to make explicit all the implicit or symbolic references.

Thus, we propose a kind of grammar for describing decorated ASTs, to capture the context-sensitive syntax of programming languages, from which a practical compiler module can easily and directly be constructed. This module would primarily drive the declaration table mechanism of the compiler, enforcing scopes of definitions and type compatibility rules, resolving and explicitly recording nonlocal references. It would take the AST from the parser and deliver the decorated AST to the code generator. Correspondingly, we would argue that any formal specification of the (dynamic) semantics of the language should be based on the decorated AST as the starting point, although we will not pursue that position here.

We emphasize that we are NOT looking for a universal solution to the language design/specification/implementation problem across all levels. Indeed, we are addressing one isolated subproblem, in the style of structured programming. The goal is the modularization of the design process, language specification (definition), and its implementation, and most importantly, better programming languages.

Criteria for a good context-sensitive formalism. There follows a list of criteria for a good formalism aimed at the context-sensitive syntax (CSS) level of programming languages:

(1) It should encourage the designer of the language to ask just the right questions and consider all aspects of the CSS. Indeed, it should guide him toward a good design and provide a good notation for recording design decisions.

(2) It should encourage, and indeed help define, clean boundaries between the context-free syntax and the CSS on the one hand, and between the CSS and the (dynamic) semantics on the other. Thus, it should contribute to modularity in the language design process, in the language descriptions, both formal and informal, and in the implementation.

(3) It should be based on a data type that is natural to the CSS problem: e.g. abstract-syntax trees rather than concrete-syntax strings, and furthermore, decorated ASTs as the result, just as ASTs are the "result" of context-free (transduction) grammars.

(4) It should preferably model accurately what the human reader does when he reads and debugs a program, so that it will be most useful as a reference.

(5) It should be totally self-contained, not needing supporting definitions outside the grammar.

(6) It should be automatically implementable, just as CFGs are, resulting in a practical compiler module. Preferably this should not involve any problems as difficult as the circularity problem for attribute grammars, although this is a minor issue. More importantly, it should be possible to detect and report meaningfully any inconsistencies or circularities in a given CSS specification.

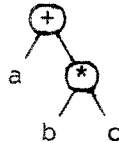
(7) Finally, to include some motherhood and apple pie, the notation for writing a CSS specification should be concise, but not to the point of obscuration, and simple, yet powerful.

Preview. The general idea of our proposed CSS notation is presented in the next section, primarily via a small sample language from elsewhere. Then comes a summary of our current approach to formalizing the notation as a grammar. Next the pragmatics of using the notation in a reference manual, and automatically generating a compiler module from it, are discussed briefly. Finally, a brief summary and evaluation of the notation is made relative to the above criteria.

The results reported here are from the Masters and Ph.D. thesis work, in progress, of the second author, under the supervision of the first.

The general idea of tree-affix dendrogrammars (TADGs)

Underlying dendrogrammar. The general idea has already been given away in the introduction. TADGs are based on "dendrogrammars", essentially context-free grammars that generate trees, rather than strings [Rou 70]. Greek "dendron" means "tree". Actually, what is generated is a direct string representation of a tree, namely "Cambridge Polish" [McC 62]. Thus, the tree



is represented by "<+ a <* b c>>". In our dendrogrammars the symbols "<" and ">" are meta-symbols, and terminal symbols are distinguished from nonterminals by quoting the former and using "identifiers" with the first letter capitalized for the latter. Thus, a "dendroproduction" with left part E and right part indicating a '+' node with two E subtrees is written "E -> <'+' E E>". When there are several right parts for one left part, we factor out the left part; and we terminate rules with a semicolon; e.g.

```

E -> <'+' E E>
    -> <'neg' E>
    -> '<INTEGER>';
  
```

The latter alternative illustrates a "pseudo-terminal", a terminal that has subrosa information, namely the actual text of the symbol. In this case literal integer numbers are intended. The angle brackets included in the quotes are intended to suggest that, although it is a terminal as far as the dendrogrammar is concerned, there is additional subrosa information, which could be viewed as subtrees, i.e. one leaf per character (digit, here).

Affixes, constraints. TADGs follow affix grammars in that they specifically admit of only right-to-left affix dependencies, and thus only left-to-right flow of attribute values. This flow, however, is relative to the grammar notation itself, not the tree generated. Indeed, a TADG can explicitly specify several passes around any given subtree.

Latin "affix" means "attachment". Nonterminals have affix variables attached to them through which context-sensitive constraints are implemented. Conceptually, TADGs flow information around ASTs that they generate, rather than the derivation trees of generated strings.

Moreover, TADGs allow TREES ONLY as affix or attribute values, so nonterminals can serve the purpose of predicates as well as generators. Thus, the formalism is totally self-contained, a property that neither affix nor attribute grammars have. It is also possible to name subtrees, pass them as affix values, and thus specify multiple traversals of them. Finally, it is possible to "decorate" one subtree with another. Let us consider an example:

```
S                               : Statement ▼ Env
-> <'while' E S>                {E: Expression ▼ Env ▲ 'boolean'}
```

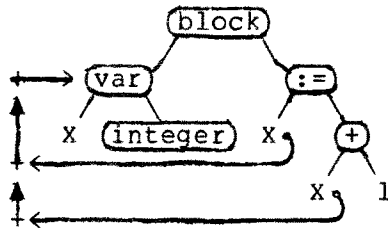
If the constraints to the right are ignored, there remains a simple dendroproduction indicating the abstract syntax of a while loop. The first constraint indicates that the left part S denotes or generates members of the domain Statement [Ten 76], the definition of which is sensitive to the context or environment Env, an inherited (▼) affix. The recursive occurrence of S in the right part abides by this same restriction, in the same environment Env, by implication, although this could have been explicitly overridden. The second constraint indicates that the subtree E is in the domain of Expression(s) of type 'boolean' in the context Env. The "type" is specified by the derived (▲) affix, which must be the tree (leaf) 'boolean' only. Now consider:

```
E                               : Expression ▼ Env ▲ Type
-> <'+' E E>                     ▲ 'integer'
                                {E: Expression ▼ Env ▲ 'integer'}
-> <'INTEGER>'                   ▲ 'integer'
-> <N> dec D                     ▲ T
                                {N: From ▼ Env ▲ D}
                                {D: <'var' <M> <T>>};
```

The usual constraint on E is overridden under the '+' node so that those subtrees (subexpressions) can be required to be of type 'integer'. The derived (▲) type of the sum is specified on the line with the '+' to be 'integer'. The literal integer case is straightforward.

The case of an identifier N is the most difficult. The surrounding angle brackets indicate that N is a "parameter", a place holder for ANY arbitrary subtree. N is restricted to an identifier by the nonterminal From, which we regard as predefined and system implemented though it can be defined in TADG notation, and which retrieves from the environment the subtree D associated with N. The association must have been established by a declaration, as we shall see shortly. D is required to be a subtree with 'var' as the name of its root and with two subtrees, M and T, with no constraints on them. (In fact, M will be the same identifier as N.) T is the derived type in this case.

Finally, we note that D is also specified as the decoration on N ; thus, from N there is a direct link to its declaration. A resulting decorated tree might look like:



Declarations are specified by the nonterminal D as being in the domain Declaration in context Env, upon which there is a side effect, namely the name N is associated with the declaration (D) by the predefined function Into:

D : Declaration ∇ Env
 \rightarrow <'var' <N> <T>> {N: Into ∇ Env ∇ D};

A substantial example. The next two pages contain two grammars adapted from the extended attribute grammar of Watt and Madsen [W&M 79]. The little language described is roughly a subset of Pascal, except that, like Algol 60, mutually recursive procedures do not require a forward declaration. Surprisingly, at least one variable declaration is required at the head of each block, and each procedure (and call) must have at least one parameter. Our grammars faithfully adhere to these conventions, although it is easy to allow zero in each case.

The first grammar describes the context-free concrete syntax and the translation to abstract-syntax trees, while the second describes the contextual constraints on ASTs and their decoration. Of course Watt and Madsen did not specify ASTs and their decoration. Nonetheless, the second grammar is believed to impose exactly the same context-sensitive restrictions as their grammar.

Concrete syntax. The first grammar below, G1, is a regular right part, string-to-tree transduction grammar [DeF 74], i.e. a context-free grammar with (extended) regular expressions in right parts of productions, and optionally, a tree part with each right part. The tree part, if present, is preceded by "=>" and indicates what node name is to parent the subtrees associated with the nonterminals and pseudo-terminals of the right part. A pictorial version of a dendrogrammar PDG generating the same ASTs generated by G1 is presented in comment form following G1.

Concrete syntax -- G1

#

parser Program:

Program -> Block '.';

Block -> 'var' (Vdcln ';'')+ (Pdcln ';'')*
'begin' Stmt list ';' 'end' => 'block';

Vdcln -> Name ':' Type => 'var';

Pdcln -> 'procedure' Name '(' Fparm list ';'')
Block => 'proc';

Fparm -> 'var' Name ':' Type => 'ref'
Name ':' Type => 'value';

Type -> < 'boolean' | 'integer' >
-> 'array' '[' Integer '..' Integer ']'
'of' Type => 'array';

Stmt -> 'begin' Stmt list ';' 'end' => ';' ;
-> 'if' Expn 'then' Stmt 'else' Stmt => 'if'
-> 'while' Expn 'do' Stmt => 'while'
-> Variable ':=' Expn => ':='
-> Name '(' Expn list ',' ')' => 'call';

Expn -> Sexp < '=' | '<>' > Sexp => 'rlnop'

Sexp -> Sexp < '+' | '-' > Term => 'addop'
-> Term;

Term -> < 'true' | 'false' > | Integer
-> Variable | '(' Expn ')';

Variable -> Variable '[' Expn ']' => 'subscript'
-> Name;

Name -> '<IDENTIFIER>'; # Lexical.

Integer -> '<INTEGER>'; # Lexical.


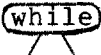


end Program

#

Abstract syntax -- PDG

Program: B



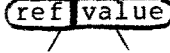
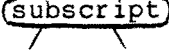
#

B:  S:    
D+ P* S+ S+ E S S E S V E N E+

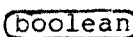
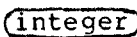
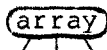

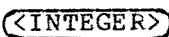
#

E:     I V
E  E E  E

#

P:  D:  F:  V:  N
N F+ B N T N T V E

#

T:    N:  I: 
I I T N: <IDENTIFIER> I: <INTEGER>

#

end of pictorial abstract syntax.

Contextual constraints -- G2

```
#
attributer Program:      #   'MtE' = empty environment.
  Program                : Pgrm
  -> B                    { : Open   ▼ 'MtE' ▲ EnvGlobal }
                        {B: Block   ▼ EnvGlobal};

  B                      : Block ▼ EnvLocal
  -> <'block' D+ P* S+>   {D: Dcln  ▼ EnvLocal}
                        {P: Dcln  ▼ EnvLocal}
                        then {P: Pbody ▼ EnvLocal}
                        {S: Stmt   ▼ EnvLocal};

  S                      : Stmt   ▼ Env
  -> <';' S+>
  -> <'if' E S S>         {E: Expn  ▼ Env ▲ 'boolean'}
  -> <'while' E S>        {E: Expn  ▼ Env ▲ 'boolean'}
  -> <':=' V E>           {V: Vrbl   ▼ Env ▲ Tcommon}
                        {E: Expn  ▼ Env ▲ Tcommon}
  -> <'call' <N> dec D E+> {N: From   ▼ Env ▲ D}
                        {D: <'proc' <X> <F>+ <B>>}
                        {F: <'ref'  <Y> <Tcommon>}
                        E: Vrbl   ▼ Env ▲ Tcommon
                        {F: <'value' <Y> <Tcommon>}
                        E: Expn  ▼ Env ▲ Tcommon};

  P                      : Pbody ▼ Env
  -> <'proc' <N> D+ B>    { : Open   ▼ Env ▲ EnvLocal}
                        {D: Dcln    ▼ EnvLocal}
                        then {B: Block ▼ EnvLocal};

  D                      : Dcln  ▼ EnvLocal
  -> <('var'|'ref'|'value'|
      'proc') <N> ...>  {N: Into   ▼ EnvLocal ▼ D};

  E                      : Expn  ▼ Env ▲ Type
  -> <'rlnop' E <Op> E>   ▲ 'boolean'
                        {E: Expn  ▼ Env ▲ 'boolean'}
  -> <'rlnop' E <Op> E>   ▲ 'boolean'
                        {E: Expn  ▼ Env ▲ 'integer'}
  -> <'addop' E <Op> E>   ▲ 'integer'
                        {E: Expn  ▼ Env ▲ 'integer'}
  -> ('true' | 'false')   ▲ 'boolean'
  -> <'<INTEGER>'>        ▲ 'integer'
  -> V                    ▲ Tvar
                        {V: Vrbl   ▼ Env ▲ Tvar};

  V                      : Vrbl   ▼ Env ▲ Type
  -> <'subscript' V E>    ▲ Telem
                        {V: Vrbl   ▼ Env ▲ T}
                        {E: Expn  ▼ Env ▲ 'integer'}
                        {T: <'array' <L> <U> <Telem>>}
  -> <N> dec D            ▲ T
                        {N: From   ▼ Env ▲ D}
                        {D: <('var'|'ref'|'value')<X><T>>};

end Program
```

The only two pseudoterminals in G1 are <IDENTIFIER> and <INTEGER>. Each occurrence of these, including the actual text of the token, is included in the AST by default. On the other hand, terminals are not included in the tree, except as they are encoded into the node name of the production in which they appear. The left part of a production is associated with the tree specified by its right part and tree part, if any. The four operators of the language, and the four key words, 'boolean', 'integer', 'true', and 'false', are surrounded by angle brackets, < and >, meaning to override the default and include these terminals as leaves in the tree.

If the pictures following G1 do not make the string-to-tree correspondence obvious, the reader should review prior work [DeR 74]. However, it may be useful to review the meanings of the regular operators: "list" means a list of that to its left separated by the delimiter to its right, "+" means one or more occurrences of that to its left, "*" means zero or more, "?" means zero or one, i.e. optional, and "|" means either that to its left or that to its right. Terminals are in single quotes in these grammars. Nonterminals are just standard identifiers. Meta-symbols are unquoted, e.g. ->, ;.

Dendrogrammar. Assuming that the reader has a firm grasp of the simple abstract syntax of this little language, we proceed to G2 and the context-sensitive constraints on the language. In this notation regular operators are used on trees, really on string representations of trees, just as they are used on strings in the concrete-syntax realm. Thus, "<'block' D+ P* S>" denotes a 'block' node whose subtrees consist of: one or more D subtrees, then zero or more P's, then one or more S subtrees.

Grammar G2 is laid out such that, if all the constraints to the right are covered up, all that remains is the "underlying dendrogrammar" UDG2 with decorations indicated by the key word 'dec'. Thus, each variable name gets decorated with a D subtree, namely the one representing its declaration, as we shall see; and each procedure name is decorated with the corresponding procedure definition subtree, as will be deduced from the constraints.

Note, however, that UDG2 is not equivalent to the pictorial dendrogrammar PDG of the page preceding it. In particular, each name N in UDG2 is left unspecified. Furthermore, the nonterminal I has been back substituted away (it was in PDG only as an abbreviation anyway); T does not appear at all in UDG2; and D, F, and P of PDG have been merged into just D in UDG2, although P also remains in UDG2, and the production for D contains the "... notation, indicating that ANY sequence of subtrees or "orchard" is derivable following the first subtree N of a 'var', 'ref', 'value', or 'proc', node.

The 'block' case is interesting because it specifies some explicit dependencies, and thus sequencing for the parser. The D and P subtrees can be derived or parsed in any order, but, as indicated by the key word 'then', this must happen before the P's are confirmed also to be in the subdomain Pbody ∇ EnvLocal, and independently, that the S's are in Stmt ∇ EnvLocal. This ordering is necessary because our model of an environment is one with side effects, as discussed below. Thus, the grammar mirrors what the reader must do to read a program top-down: in each block he must scan the declarations and procedure headers, to find out what is defined, but not how procedures work, BEFORE investigating procedure bodies and the statements that are the block body.

Note that procedure bodies are derived in a definite order also. First a new "scope of definition" is opened in the environment, resulting in a local version. Then the formal parameter declarations are derived, as indicated by the attribute dependency: the EnvLocal derived by Open is inherited by Dcln. Then the block can be derived, as indicated by the keyword 'then'.

The nonterminal D (promiscuously) generates declarations of both variables and formal parameters, as well as procedure declarations. Its only real purpose is to enter the name N "Into" the current environment, associating it there with the very declaration D just derived (parsed).

Thus, when V generates a name N, it is constrained to be an identifier by "From" which also retrieves the associated declaration subtree D From the environment. The name N is decorated with D, which is constrained to be a 'var', 'ref', or 'value' subtree, not a 'proc', with two subtrees X and T. T is used to represent the type of the variable for economy's sake, and is the derived attribute of Vrbl in this case. It happens that we know that X will be the same identifier as N, by definition of the environment mechanism, but this need not be checked here (which it could be by changing $\langle X \rangle$ to $\langle N \rangle$).

The 'subscript' case derives the V and E subtrees independently, requiring that E be of type 'integer' and that V be of type 'array' with element type Telem, which is the derived Type of Vrbl in this case, as indicated by " Δ Telem" to the right of '<subscript' ... >'. The L and U subtrees are not restricted by this grammar, but are known from grammar G1 to be '<INTEGER>' nodes, i.e. leaves representing literal integer constants. (We could have replaced "<L> <U>" with "... " since they are unrestricted.)

Such mergers and missing restrictive specifications mean that UDG2 derives a superset of the ASTs generated by G1. This is perfectly all right, since redundant restriction is pointless in this modular language specification. Note also that, had it been useful, some further restriction could have been achieved by designing UDG2 to generate a subset of some portion of the tree language generated by G1, getting the effect of intersecting two context-free languages. For example, type checking could have been achieved in this trivial language by generating Boolean expression subtrees only where allowed, etc. We chose not to do that here to keep a close correspondence with the watt grammar and because that technique does not work for more complex languages in which the programmer can define new types.

Finally for UDG2, note that the grammar is ambiguous in E, because of the two identical right parts defining relational expression subtrees. Also, V is ambiguous, since the parameter <N> derives ANY subtree, including the 'subscript' alternative. These ambiguities will be eliminated by constraints. Thus, a deterministic "tree parser" will be constructable to "recognize" the AST and impose the constraints.

Constraints. Now consider the right half of G2, the constraints. Each left part now contributes to a syntactic (tree) domain, parameterized by zero or more inherited or derived attributes. Thus, S no longer generates just any statements, but only those that are meaningful in a particular environment Env (or context, or in compiler terms, a declaration table), as indicated by the new left part: "S : Stmt ∇ Env". The domain Stmt is the union of some subdomains (not partitions), each of which is the set of statements derivable in a particular environment. In G2 S is the only nonterminal with Stmt as its constraint in the left part, therefore it is the only contributor to the domain Stmt, and so S derives the entire domain. Such uniqueness is always the case in this simple grammar. Similarly, expressions (Expn) are derived by E in an (inherited) Env and with a (derived) Type. And so on.

Consider the assignment statement, the '=' node. The V subtree must be a variable (Vrbl) in the inherited Env and with some derived type, called here Tcommon. The E subtree must be an Expn in the inherited Env with the SAME derived type Tcommon. It matters NOT what order the two subtrees are derived or "parsed" in; they are independent -- in fact, ALL dependencies must be stated explicitly in these grammars. Of course, the assumed Env is inherited from the left part.

Next consider the 'while' statement. The E subtree must be an Expn in Env with the leaf 'boolean' as its derived type. No restriction is stated for the S subtree, so the constraint of the left part is recursively applied by default. This default applies twice in the 'if' right part, and one or more times in the ';' case, as needed.

The E productions should be easy to understand now, so let us consider the most complex case, 'call'. Here we get the procedure subtree D From the Env by constraint on N. D must not be a 'var', 'ref', or 'value' subtree, but a 'proc' with each formal parameter denoted F. Independently, each F is required to be a 'ref' with right subtree Tcommon AND (juxtaposition) the corresponding actual parameter is required to be a Vrbl in Env with the same type Tcommon, OR (|) alternatively, F must be a 'value' with right subtree Tcommon AND the actual must be an Expn in Env with the same Tcommon.

Environments. The derivation process is started with the goal or start symbol, whose constraint is not parameterized. In this case Program starts by Open-ing a new scope relative to an initial environment that is empty. If the language had had any pre-defined procedures or variables, they would have been entered in the environment at this point. Now the derivation continues with B, given the "global" Env.

We are still debating how much power the environment mechanism should have, and whether or not it should be side-effect-free. There seem to be clear advantages either way. Futhermore, it is not clear whether the environment formalism should be described in the TADG for each new language, or extra-grammatically, once and for all -- for all languages, that is. This question is related to the problem of formalizing and/or restricting the decoration of subtrees, so we leave the issue for further discussion below.

Formalization of TADGs

This is the area in which our research is the least complete and conclusive. The following page defines the abstract and concrete syntax of TADGs, just as was done in the previous section for the little programming language. That is, the page contains a regular right part grammar for TADGs. Soon to come will be a TADG for TADGs, i.e. a self-describing grammar on the context-sensitive level. Thus, initially we are treating TADGs as a programming language, a notation for programming constraint-checking phases of compilers.

Our ultimate goal, however, is to devise a formal grammatical model as a foundation for TADGs. The main stumbling blocks currently are the very aspects of TADGs that make them novel, powerful, and convenient, namely, the decorations and the environment mechanism. It is desirable to harness this power and to restrict it to no more than is needed for this problem area. As it is, Turing machines can easily be simulated.

It may be possible to restrict the decorations so that they can be described generatively. This, no doubt, means disallowing redecorations, at least. Without any such limitation it is possible to use the decorations as memory cells and perform arbitrary iterative computations. Clearly, this is a misuse of what is intended to be a grammatical model. Relatedly, the ability to create new nodes, even whole subtrees, "dynamically" for attributes and decorations, which we currently allow, will have to be curtailed. Indeed, we have rarely used the facility in practical grammars, and even when we have, it could have been avoided by defining an extra node or two in the AST.

Finally, an approach to environments must be settled upon. Are the usual models, as used in attribute grammars [W&M 79] and denotational definitions [Ten 76], e.g., really appropriate for reference manuals, or are they just appropriate for compiler writers? Can one environment formalism be defined that will suffice for all well-designed languages? Or should a specialized mechanism be defined within the TADG for each new language? If the latter, then is it possible to deduce an efficient implementation for each? If so, how? These are some of our current research questions.

```

#
# Concrete syntax of TADGs --
#
parser TADG:
  TADG      -> 'attributer' Goal ':'
              Rule+
              'end' Goal          => 'attributer';
  Goal      -> Nontermnl;

  Rule      -> Leftpart ('->' Rightpart)+  => 'rule';
  Leftpart  -> Nontermnl ':' Predicate      => 'leftpart';
  Predicate -> Pred_name Inherits Derives  => 'predicate';
  Inherits  -> ('▼' Tree_expn)*             => 'inherits';
  Derives   -> ('▲' Tree_expn)*             => 'derives';
  Rightpart -> Tree_expn Derives Cnstrnts   => 'rightpart';

  Cnstrnts  -> Consgroup list 'then'        => 'sequence'?;
  Consgroup -> ('{' Cons_expn '}')*         => 'group'?;
  Cons_expn -> Cons_term_list '|'           => 'or'?;
  Cons_term -> Cons_prim+                   => 'and'?;
  Cons_prim -> Parameter?
              ':' (Predicate | Subtree)     => 'constraint'
              -> '(' Cnstrnts ')';

  Tree_expn -> Tree_term list '|'           => 'alternates'?;
  Tree_term -> Tree_fact*                   => 'catenate'?
              -> '...'                     => 'any_trees';
  Tree_fact -> Tree_fact 'dec' Parameter    => 'decorate';
              -> Tree_fact 'has' Parameter => 'decoration';
              -> Tree_prim 'is' Parameter   => 'labeled';
              -> Tree_prim '+'               => 'one_or_more';
              -> Tree_prim '*'               => 'zero_or_more';
              -> Tree_prim '?'               => 'zero_or_one';
              -> Tree_prim;
  Tree_prim -> '<' Parameter '>'             => 'parameter';
              -> '(' Tree_expn ')'
              -> Subtree | Nontermnl;

  Subtree   -> '<' Node_name Tree_fact* '>' => 'subtree';
              -> Leaf;
  Node_name -> '(' '<STRING>' list '|' ')' => 'one_of';
              -> '<STRING>';

  Leaf      -> '<STRING>';                  # Lexical.
  Parameter -> '<IDENTIFIER>';              # Lexical.
  Pred_name -> '<IDENTIFIER>';              # Lexical.
  Nontermnl -> '<IDENTIFIER>';              # Lexical.
end TADG

#
# "?" in tree parts means "do not build the node if there
#                               is only one subtree.
# Contextual constraints --
#
#       Soon to come:  a TADG for TADGs!
#

```


Dendrogrammars. The underlying dendrogrammars of TADGs are about as easy to formalize as are context-free grammars:

Definition. A (context-free) dendrogrammar G is a quadruple (T, N, S, P) where
 T is a finite set of "terminal" symbols (node names),
 N is a finite set of "nonterminal" symbols such that
 T , N , and $\{>, <\}$ are mutually disjoint sets,
 S is a member of N , called the "start symbol", and
 P is a finite subset of $N \times L(G_trees)$ where
 each "dendroproduction" in P is written $A \rightarrow w$,
 w is called the "right part" (a tree expression),
 A is called the "left part" (a nonterminal), and
 G_trees is a context-free grammar (Tt, Nt, St, Pt)
 where $Tt = T \cup \{>, <\}$, $Nt = \{St, Tree\}$, and $Pt =$
 $\{ St \rightarrow St\ Tree, Tree \rightarrow t \text{ for all } t \text{ in } T,$
 $St \rightarrow (\text{empty}), Tree \rightarrow < t\ St > \text{ for all } t \text{ in } T \}.$

Note that G_trees is a CFG that generates "tree expressions", namely, Cambridge Polish notation [McC 62] with angle brackets serving as meta-parentheses and with terminals in T serving as node names, both interior and leaf. Of course, $L(G_trees)$ is the language generated by G_trees , and $L(G)$ is the "dendrolanguage" generated by G , as usual for CFGs.

In general, the tree expressions denote sequences of trees, or "orchards", rather than just trees, so G generates orchards in general, too. The former is handy because it allows us to describe n -ary trees, or "bushes", which are rather more useful often than ranked trees (a fixed number of subtrees per node name). Relatedly, and even more useful, as the former TADGs have clearly demonstrated, is the idea of allowing regular expressions in the right parts of dendroproductions, resulting in a "regular right part dendrogrammar". The above definition is easily extended to "RRPDGs" by including the desired additional meta-symbols in G_trees appropriately. See for example the tree_expn subgrammar of the TADG concrete-syntax grammar.

We believe that further research will produce extensions of the above definition to describe first decorated trees and then affixes and constraints.

Use in reference manuals and compiler construction

Each reference manual should be organized around the abstract syntax of the language it describes. Thus, its major sections should correspond to the syntactic domains, plus a separate section for the lexicon and appendices for the individual, collected grammars and other terse summaries. At the least, there should be included a lexical grammar, a context-free phrase-structure grammar, a context-sensitive constraint grammar, and a formal definition of the semantics.

Each syntactic domain section, e.g. for declarations or statements or expressions or variables, should be subdivided according to individual language constructs, e.g. the 'while' statement, the 'loop' statement, including the corresponding 'exit', the 'procedure' definition, including 'call' and 'return', etc. Each construct description should look something like the following sample:

```

**** 'while' Statement ****#

Concrete syntax:  'while' Expression 'do' Block 'od'

Abstract syntax:  <'while' E S>

Constraints:      E: Expression ▼ Env ▲ 'boolean'
                  S: Statement  ▼ Env

Semantics:        ... (denotational definition?)

Discussion:        ... (informal description)
                  (special notes, etc.)

*****#

```

Construction of a practical compiler module from each language level description should be straightforward, though there is a long way to go toward that goal, especially in the semantics area. The goal has been attained in the lexical and phrase-structure areas and attainment is now eminent in the area of context-sensitive constraints. In the case of TADGs, we expect to know soon how to map each one into a practical set of recursive tree-traversing procedures that pass value parameters for inherited affixes and result parameters for derived affixes, decorating the tree as they proceed. We already perform this mapping manually, getting scope and type checking compiler modules that are quite satisfactory, in both size and speed. However, at least another year will be needed to get the TADGs well defined and the mapping to a compiler module automated.

Summary and evaluation

Our research in this area is very much on-going. The form of TADGs presented here is different from what was presented at the Workshop on Semantics-Directed Compiler Generation in January, 1980. Just in the last week the second author has proposed a revised scheme that utilizes decorations so well that affixes are unnecessary and that has a user-oriented way of specifying scopes of definition, in place of the more compiler-oriented environment scheme usually used in attribute grammars and denotational definitions. On the other hand, it is not obvious how to deduce a compiler module in this new scheme.

An earlier version of TADG was used to describe the context-sensitive constraints of Pascal and to design those for a language of nearly the ambition of Ada. In the case of Pascal we found that some constraints were not clearly stated in any formal or informal definition that we checked. For example, the scope of definition of identifiers and the compatibility of types are ill-defined. In these cases we followed Steensgaard-Madsen [Ste 79].

In the case of the more ambitious language we found that the TADG did cause us to ask just the right questions and was a good tool for recording the decisions as they were made. It did, indeed, guide us toward a good design. It also contributed to modularity in the design process in that we delayed the design of the concrete syntax and the semantics until after the constraints on the abstract-syntax trees were finished.

Of course, we feel that TADGs are based on a natural data type for the constraint problem. In fact, we found the decorated tree to be ideal for input to the code generator, which was itself described in an extension of TADG notation.

What we are most dubious about is whether TADGs, or any other related descriptive method now available, model in a good way what the reader does when he reads the program. Given an error at a point of usage of an identifier, one wants to refer back to its declaration. The usual environment mechanism, ignoring the side effects used here, is at best a very implicit model of such a back reference and does not make use of the available tree structure.

TADGs are self-contained, in that the environment can be modeled via TADG rules, but this modeling seems to misuse the (re-)decoration capability. We are still looking for a better solution. The previous version of TADGs was directly implementable -- so much so that the implementation shined through the notation and made it more a programming language than a grammar. Finally, TADGs are clearly concise. They become obscure only when intricate nonlocal correspondences must be implemented via the environment.

References (Internal page references below each reference.)

- [Ada 79] Preliminary Ada reference manual. SIGPLAN Notices 14(6A), June 1979.
- [Boc 76] Bochmann, G.V.: Semantic evaluation from left to right. CACM 19, 55-62, 1976.
- [Cul 73] Culik, K.: A model for the formal definition of
(3) programming languages. IJCM A(3), 315-345, 1973.
- [DeR 74] De Remer, F.: Review of formalisms and notation.
(7,10) In: Compiler Construction - An Advanced Course (F. L. Bauer and J. Eickel, eds.), Lecture Notes in Computer Science 21, Springer-Verlag, N.Y., 1974.
- [JOR 75] Jazayeri, M., Ogden, W.F., and Rounds, W.C.
(1) Complexity of the circularity problem for attribute grammars. CACM 18(12), 697-706, December, 1975.
- [Knu 68] Knuth, D.E.: Semantics of context-free languages.
(1) Math. Sys. Theory 2(2), 127-145, 1968(errata 1971).
- [Kos 71] Koster, C.H.A.: Affix grammars. In: ALGOL 68
(1) Implementation (J.E. Peck, ed.), 95-109, North-Holland, Amsterdam, 1971.
- [M&J 80] Madsen, M. and Jones, N.: Letting the attributes
(1) influence the parsing. Workshop on Semantics-Directed Compiler Construction, Aarhus, Denmark, January, 1980.
- [McC 62] McCarthy, J., et al: LISP 1.5 Programmer's Manual.
(5,16) MIT Press, Cambridge, Mass., 1962.
- [Rou 70] Rounds, W.C.: Mappings and Grammars on Trees.
(5) Math. Sys. Theory 4(3), 257-287, 1970.
- [Ste 79] Steensgaard-Madsen, J.: Pascal clarifications and
(18) recommended extensions. Acta Inf. 12, 73-94, 1979.
- [Ten 76] Tennent, R.D.: The denotational semantics of programming languages. CACM 19(8), 437-453, Aug. 1976.
(6,14)
- [Van 75] Van Wijngaarden, A., et al: Revised report on the
(2) Algorithmic Language ALGOL 68. Acta Informatica 5, 1-236, 1975; Springer-Verlag, N.Y., 1976.
- [Wat 74] Watt, D.A.: LR parsing of affix grammars. Report
(1,2) 7, Comp. Sci. Dept., Univ. of Glasgow, Aug., 1974.
- [W&M 79] Watt, D., and Madsen, O.: Extended attribute grammars. DAIMI report no. PB-105, Comp. Sci. Dept.,
(1,2) Aarhus University, Aarhus, Denmark, November 1979.
(7,14)