Tool Support for Data Structures

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Tool Support for Data Structures

Josef Grosch

Nov. 8, 1989

Report No. 17

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

Tool Support for Data Structures

Josef Grosch
GMD Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe, Germany
Tel: +721-6622-26
E-Mail: grosch@karlsruhe.gmd.de

Abstract Linked records are a general mechanism to build data structures like lists, trees, and graphs. Most high-level programming languages only provide the definition of record types, an operator for component selection, and allocation of record storage. We propose to specify complete graph structures by context-free grammars. A tool can be used to transform such a specification into a set of record type declarations and program code for features like denotations for record values, input and output for record values and complete graphs, or interactive browsers for data structures. We describe such a tool called *ast* (generator for abstract syntax trees), its specification language, the advantages of this approach, and our current experiences. Currently, the main application is the specification of attributed abstract syntax trees within compilers. We finally discuss the relationship to related work.

1. Introduction

Linked records are a general mechanism to build data structures like lists, trees, and graphs. Most high-level programming languages only provide the definition of record types, an operator for component selection, and allocation of record storage. Therefore, the treatment of compound data types in most high-level languages can be considered to be quite "low-level". Exceptions are very-high-level languages like e. g. SETL [SDD86] which provides denotations (aggregates) and input/output operations for values of all data types, even compound ones like tuples, arrays, or sets.

We propose to raise the level of conventional languages somewhat by improving the declarations of data structures and by extending the set of operations available for compound data types. Declarations should not merely describe single records but also the relationships among them. Additional operations include denotations for record values (aggregates) as well as input and output for record values or complete data structures like graphs. Moreover, it is desirable to have commonly used operations for general data structures. These could range from reversing the elements of lists to interactive browsers for graphs which allow the inspection of the values of all fields of the nodes in a user-driven dialogue.

The structure of graphs can be specified conveniently by context-free grammars. A grammar rule describes a node type and a nonterminal a set of node types.

The above features could be incorporated into existing or future languages. This would of course be the kind of realization to prefer. However, today we have to live with languages like Modula-2 or C without those features. Therefore, a tool could produce a program module written in the concrete target language which defines the specified data structure by a set of record declarations and which implements the additional operations by generated procedures. This has the advantage that no changes to existing languages are necessary.

This paper presents such a tool called *ast*: generator for *a*bstract *s*yntax *t*rees [Gro91]. The tool's name is derived from its main application in compiler construction where it is used for attributed abstract syntax trees. *ast* is implemented in Modula-2 as well as in C under UNIX and generates Modula-2 or C source modules. We describe the specification language of the tool, its output, its advantages, and our experiences. We also discuss related approaches. In the following we talk only about the data structure directed graph because lists and trees are special cases thereof. The examples use Modula-2 as target language.

2. Specification Language

The structure of directed graphs is specified by a formalism based on context-free grammars. However, we use the classical terminology for graphs in defining the specification language. Its relationship to context-free grammars is discussed later.

2.1. Node Types

A directed graph consists of *nodes*. A node may be related to other nodes in a so-called *parent-child* relation. Then the first node is called a *parent* node and the latter nodes are called *child* nodes. Nodes without a parent node are usually called *root* nodes, nodes without children are called *leaf* nodes.

The structure and the properties of nodes are described by *node types*. Every node belongs to a node type. A specification of a graph describes a finite number of node types. A node type specifies the names of the child nodes and the associated node types as well as the names of the attributes and the associated attribute types.

2.2. Children

Children are distinguished by *selector* names which have to be unambiguous within one node type. The children are of a certain node type.

Example:

```
If        = Expr: Expr Then: Stats Else: Stats .
While     = Expr: Expr Stats: Stats .
```

The example introduces two node types called *If* and *While*. A node of type If has three children which are selected by the names *Expr*, *Then*, and *Else*. The children have the node types *Expr*, *Stats*, and *Stats*. If a selector name is equal to the associated name of the node type it can be omitted. Therefore, the above example can be abbreviated as follows:

```
If        = Expr Then: Stats Else: Stats .
While     = Expr Stats .
```

2.3. Attributes

As well as children, every node type can specify an arbitrary number of *attributes* of arbitrary types. Like children, attributes are characterized by a selector name and a certain type. The descriptions of attributes are enclosed in brackets. The attribute types are given by names taken from the target language. Missing attribute types are assumed to be int or INTEGER depending on the target language (C or Modula-2). Children and attributes can be given in any order. The type of an attribute can be a pointer to a node type. In contrast to children, *ast* does not follow such an attribute during a graph traversal. All attributes are considered to be neither tree nor graph structured. Only the user knows about this fact and therefore he/she should take care.

Example:

```
Binary    = Lop: Expr Rop: Expr [Operator: INTEGER] .
Unary     = Expr [Operator] .
IntConst  = [Value] .
RealConst = [Value: REAL] .
```

2.4. Extensions

To allow several alternatives for the types of children an *extension* mechanism is used. A node type may be associated with several other node types enclosed in angle brackets. The first node type is called *base* or *super* type and the latter types are called *derived* types or *subtypes*. A derived type can in turn be extended with no limitation of the nesting depth. The extension mechanism induces a subtype relation between node types. This relation is transitive. Where a node of a certain node type is required, either a node of this node type or a node of a subtype thereof is possible.

Example:

```
Stats        = <
   If        = Expr Then: Stats Else: Stats .
   While     = Expr Stats .
> .
```

In the above example *Stats* is a base type describing nodes with neither children nor attributes. It has two derived types called *If* and *While*. Where a node of type Stats is required, nodes of types Stats, If, and While are possible. Where a node of type If is required, nodes of type If are possible, only.

Besides extending the set of possible node types, the extension mechanism has the property of extending the children and attributes of the base type. The derived types possess the children and attributes of the base type. They may define additional children and attributes. In other words they *inherit* the structure of the base type. The selector names of all children and attributes in an extension hierarchy have to be distinct. The syntax has been designed this way in order to allow single inheritance, only.

Example:

```
Stats        = Next: Stats [Position: tPosition] <
   If        = Expr Then: Stats Else: Stats .
   While     = Expr Stats .
> .
```

Nodes of type *Stats* have one child selected by the name *Next* and one attribute named *Position*. Nodes of type *While* have three children with the selector names *Next*, *Expr*, and *Stats* and one attribute named *Position.*

A node of a base type like *Stats* usually does not occur in an abstract syntax tree for a complete program. Still, *ast* defines this node type. It could be used as placeholder for unexpanded nonterminals in incomplete programs which occur in applications like syntax directed editors.

2.5. Modules

The specification of node types can be grouped into modules. This feature can be used to structure a specification or to extend an existing one. If a node type has already been declared the given children, attributes, and extensions are added to the existing declaration. Otherwise a new node type is introduced.

Example:

```
MODULE my_version

Stats        = [Env: tEnv] <                       /* add attribute */
   While     = Init: Stats Terminate: Stats .      /* add children  */
   Repeat    = Stats Expr .                        /* add node type */
> .

END my_version
```

2.6. Properties

Children and attributes can be given several properties by attaching keywords like INPUT or REVERSE. *Input* attributes receive a value at node-creation time, whereas non-input attributes may receive their values at later times. Input attributes are included into the parameter list of the node constructor procedures (see section 3). As a shorthand, every list of children and attributes may contain the symbol '->' to separate input from non-input items. The property *reverse* specifies how lists should be reversed. It is discussed in the next section.

2.7. Reversal of Lists

Recursive node types like *Stats* in the abstract grammar of the example below describe lists of subtrees. There are some cases where it is convenient to be able to easily reverse the order of the subtrees in a list. The facility provided by *ast* is a generalization of an idea presented in [Par88].

Using LR parsers, one might be forced to parse a list using a left-recursive concrete grammar rule because of the limited stack size. The concrete grammar rules of the following examples are written in the input language of the parser generator *lalr* [Gro88, GrV88] which is similar to the one of YACC [Joh75]. The node constructor procedures within the semantic actions are the ones provided by *ast* (see section 3).

Example:

concrete grammar (with tree building actions):

```
Stats:                          {$$ := mStats0 ();      } .
Stats: Stats Stat ';'           {$$ := mStats1 ($2, $1);} .
Stat : WHILE Expr DO Stats END {$$ := mWhile  ($2, ReverseTREE ($4));} .
```

abstract grammar:

```
Stats         = <
   Stats0     = .
   Stats1     = Stat Stats REVERSE .
> .
```

Without the call of the procedure ReverseTREE and the property REVERSE a parser using the above concrete grammar would construct statement lists where the list elements are in the wrong order, because the last statement in the source would be the first one in the list. The WHILE rule represents a location where statement lists are used.

To easily solve this problem, *ast* can generate a procedure to reverse lists. The specification has to describe how this should be done. At most one child of every node type may be given the property *reverse*. The generated list reversal procedure ReverseTREE then reverses a list with respect to this indicated child. The procedure ReverseTREE has to be called exactly once for a list to be reversed. This is the case at the location where a complete list is included as subtree (e. g. in a WHILE statement).

2.8. Target Code

An *ast* specification may include sections containing *target code*. Target code is code written in the target language which is copied unchecked and unchanged to certain places in the generated module. Target code can be used for import or export statements, for the declaration of global variables or procedures, and for statements to initialize or finalize the declared data structures.

2.9. Type Specific Operations

Procedures generated by *ast* apply seven operations to attributes: initialization, finalization, ascii read and write, binary read and write, and copy (see Table 1). *Initialization* is performed whenever a node is created. It can range from assigning an initial value to the allocation of dynamic storage or the construction of complex data structures. *Finalization* is performed immediately before a node is deleted and may e. g. release dynamically allocated space. The *read* and *write* operations enable the readers and writers to handle the complete nodes including all attributes, even those of user-defined types. The operation *copy* is needed to duplicate values of attributes of user-defined types. By default, *ast* just copies the bytes of an attribute to duplicate it. Therefore, pointer semantics is assumed for attributes of a pointer type. If value semantics is needed, the user has to take care about this operation.

The operations are type specific in the sense that every type has its own set of operations. All attributes having the same type (target type name) are treated in the same way. Chosing different type names for one type introduces subtypes and allows to treat attributes of different subtypes differently. Type operations for the predefined types of a target language are predefined within *ast*. For user-defined types, *ast* assumes default operations (see Table 1). The procedures yyReadHex and yyWriteHex read and write the bytes of an attribute as hexadecimal values. The procedures yyGet and yyPut read and write the bytes of an attribute unchanged (without conversion). The operations are defined by a macro mechanism. TYPE is replaced by the concrete type name. *a* is a formal macro parameter referring to the attribute. It is possible to redefine the operations by including new macro definitions written in *cpp* syntax.

Table 1: Type specific operations

		default macro	
operation	macro name	C	Modula-2
initialization	beginTYPE(a)		
finalization	closeTYPE(a)		
ascii read	readTYPE(a)	yyReadHex (& a, sizeof (a));	yyReadHex (a);
ascii write	writeTYPE(a)	yyWriteHex (& a, sizeof (a));	yyWriteHex (a);
binary read	getTYPE(a)	yyGet (& a, sizeof (a));	yyGet (a);
binary write	putTYPE(a)	yyPut (& a, sizeof (a));	yyPut (a);
copy	copyTYPE(a)		

3. Generated Program Module and its Use

A specification as described in the previous section is translated by *ast* into a program module consisting of a definition and an implementation part. Only the definition part is sketched here. The definition part contains primarily type declarations to describe the structure of the graphs and the headings of the generated procedures.

Every node type is turned into a constant declaration and a record (struct) declaration. That is quite simple, because node types and record declarations are almost the same concepts except for the extension mechanism and some shorthand notations. All these records become members of a variant record (union) used to describe graph nodes in general. This variant record has a tag field called *Kind* which stores the code of the node type. A pointer to the variant record is a type representing graphs. Like all generated names, this pointer type is derived from the name of the specification. Table 2 briefly explains the exported objects. Their generation is requested by simple command line options.

The parameters of the procedures *m<node type>* have to be given in the order of the *input* attributes in the specification. Attributes of the base type (recursively) precede the ones of the

Table 2: Generated objects and procedures

object/procedure	description
<node type>	named constant to encode a node type
tTREE	pointer type, refers to variant record type describing all node types
TREERoot	variable of type tTREE, can serve as root (additional variables can be declared)
MakeTREE	node constructor procedure without attribute initialization
n<node type>	node constructor procedures with attribute initialization according to the type specific operations
m<node type>	node constructor procedures with attribute initialization from a parameter list for *input* attributes
ReleaseTREE	node or graph finalization procedure, all attributes are finalized, all node space is deallocated
ReleaseTREEModule	deallocation of all graphs managed by a module
WriteTREENode	ASCII node writer procedure
ReadTREE	ASCII graph reader procedure
WriteTREE	ASCII graph writer procedure
GetTREE	binary graph reader procedure
PutTREE	binary graph writer procedure
ReverseTREE	procedure to reverse lists
TraverseTREETD	top down graph traversal procedure (reverse depth first)
TraverseTREEBU	bottom up graph traversal procedure (depth first search)
CopyTREE	graph copy procedure
CheckTREE	graph syntax checker procedure
QueryTREE	graph browser procedure
BeginTREE	procedure to initialize user-defined data structures
CloseTREE	procedure to finalize user-defined data structures

derived type. The procedures *TraverseTREETD* and *TraverseTREEBU* visit all nodes of a graph. At every node a procedure given as parameter is executed. An assignment of a graph to a variable of type *tTREE* can be done in two ways: The usual assignment operators '=' or ':=' yield pointer semantics. The procedure *CopyTREE* yields value semantics by duplicating a given graph.

The procedure *QueryTREE* allows to browse a graph and to inspect one node at a time. A node including the values of its attributes is printed on *standard output*. Then the user is prompted to provide one of the following commands from *standard input*:

parent	display parent node
quit	quit procedure QueryTREE
<selector>	display specified child

Unfortunately, the typing rules of *ast* (see section 2.4.) can not be mapped to every target language. For example the subtype relation can not be expressed in Modula-2. A subtype has to be compatible with its base type. Two subtypes of one base type have to be incompatible. As a compromise, all node types without base types could be implemented by different pointer types. Extensions of a base type would be mapped to the same pointer type as the base type. This solution would implement half of *ast's* typing rules through static typing of the target language. For a full implementation, target languages with subtypes such as Oberon or C++ are necessary.

The current implementation of *ast* omits static type checking. It offers dynamic type checking through the procedure *CheckTREE*. This procedure has to be called explicitly to check if a graph is properly typed. In case of typing errors the involved parent and child nodes are printed on *standard error*.

The remainder of this section explains how to use the generated objects, presents the advantages of this approach, and reports early experience with the method.

Trees or graphs are created by successively creating their nodes. The easiest way is to call the constructor procedures m<node type>. These combine node creation, storage allocation, and attribute assignment. They provide a mechanism similar to record aggregates. Nested calls of constructor procedures allow programming with (ground) terms as in Prolog or LISP. The type of a node can be retrieved by examination of the predefined tag field called *Kind*. Children and attributes can be accessed using two record selections. The first one states the node type and the second one gives the selector name of the desired item.

Example:

abstract syntax:

```
Expr         = [Position: tPosition] <
   Binary    = Lop: Expr Rop: Expr [Operator: INTEGER] .
   Unary     = Expr [Operator] .
   IntConst  = [Value] .
   RealConst = [Value: REAL] .
> .
```

tree construction by a term:

```
CONST Plus = 1;
VAR t: tTREE; Pos: tPosition;

t := mBinary (Pos, mIntConst (Pos, 2), mIntConst (Pos, 3), Plus);
```

tree construction during parsing:

```
Expr: Expr '+' Expr {$$.Tree := mBinary ($2.Pos, $1.Tree, $3.Tree, Plus);} .
Expr:      '-' Expr {$$.Tree := mUnary  ($1.Pos, $2.Tree, Minus);        } .
Expr: IntConst      {$$.Tree := mIntConst ($1.Pos, $1.IntValue);         } .
Expr: RealConst     {$$.Tree := mRealConst ($1.Pos, $1.RealValue);       } .
```

access of tag field, children, and attributes:

```
CASE t^.Kind OF
| Expr  : ... t^.Expr.Position             ...
| Binary: ... t^.Binary.Operator           ...
          ... t^.Binary.Lop                ...
| Unary : ... t^.Unary.Expr^.Expr.Position ...
END;
```

ast can be used not only for abstract syntax trees in compilers but for every tree or graph like data structure. In general the data structure can serve as interface between phases within a program or between separate programs. In the latter case it would be communicated via a file using the generated reader and writer procedures.

Generated tree respectively graph modules have successfully been used in compilers e. g. for MiniLAX [WGS89] and UNITY [Bie89] as well as for a Modula -> C translator [Mar90]. The modules for the internal data structure of *ast* itself and the attribute evaluator generator *ag* [Gro89] have also been generated by *ast*. Moreover, the symbol table module of the Modula -> C translator has been generated.

The advantage of this approach is that it saves considerably hand-coding of trivial declarations and operations. Table 3 lists the sizes (numbers of lines) of some specifications and the generated modules. Sums in the specification column are composed of the sizes for the definition of node types and for user-supplied target code. Sums in the tree module column are composed of the sizes for the definition part and for the implementation part. The large sizes of the tree modules are due to the numerous node constructor procedures and from the graph browser in the case of *ag*. These procedures proved to be very helpful for debugging purposes as they provide readable output of complex data structures.

Table 3: Examples of *ast* applications

application	specification	tree module
MiniLAX	56	202 + 835 = 1037
UNITY	210	207 + 962 = 1169
Modula -> C	240	583 + 3083 = 3666
ag	78 + 347 = 425	317 + 1317 = 1634
Symbol table	82 + 900 = 982	399 + 1431 = 1830

The realization of the presented concepts by a preprocessor leads to the mixture of generated and hand-written program code. The debugging of such a program may be problematic. Of course, the pure generated parts are correct. With the possibility to insert target code and type specific operations errors might be introduced. These are detected by the compiler or during run time and reported with respect to the generated program code instead of the specification. Therefore, errors in this situation are hard to debug. This problem could be solved by incorporating the concepts into a language instead of implementing them by a preprocessor.

4. Related Research

4.1. Variant Records

ast specifications and variant record types like in Pascal [JWM85] or Modula-2 [Wir85] are very similar. Every node type in an *ast* specification corresponds to a single variant. In the generated code, every node type is translated into a record type. All record types become members of a variant record type representing the type for the graph nodes.

The differences are the following: *ast* specifications are shorter than directly hand-written variant record types. They are based on the formalism of context-free grammars (see section below). The generator *ast* automatically provides operations on record types which would be simple but voluminous to program by hand. The node constructor procedures allow programming with record aggregates and provide dynamic storage management. The reader and writer procedures supply input and output for record types and even for complete linked data structures such as trees and graphs.

4.2. Type Extensions

Type extensions have been introduced with the language Oberon [Wir88a, Wir88b, Wir88c]. The extension mechanism of *ast* is basically the same as in Oberon. The notions extension, base type, and derived type are equivalent (see Table 4). *Type tests* and *type guards* are not supported by *ast*. They can be programmed by inspecting the tag field of a node. *ast* does not provide assignment of subtypes to base types in the sense of value semantics or a projection, respectively. The tool can be seen as a preprocessor providing type extensions for Modula-2 and C.

The second example in section 2.4. illuminates the relationship between *ast* and Oberon. The node type *Stats* is a base type with two fields, a child and an attribute. It is extended e. g. by the node type *While* with two more fields representing children.

4.3. Context-Free Grammars

As already mentioned, *ast* specifications are based on context-free grammars. *ast* specifications extend context-free grammars by selector names for right-hand side symbols, attributes, the extension mechanism, and modules. If the features are omitted we basically arrive at context-free grammars. This holds from the syntactic as well as from the semantic point of view. The names of the node types represent both terminal or nonterminal symbols and rule names. Node types correspond to grammar rules. The notions of derivation and derivation tree can be used similarly in both cases. The selector names can be seen as syntactic sugar and the attributes as some kind of terminal symbols. The extension mechanism is equivalent to a shorthand notation for factoring out common rule parts in combination with implicit chain rules.

Again referring to the second example in section 2.4., *Stats* corresponds to a nonterminal. There are two rules or right-hand sides for *Stats* which are named *If* and *While*. The latter would be regarded as nonterminals, too, if a child of type *If* or *While* would be specified.

4.4. Attribute Grammars

Attribute grammars [Knu68, Knu71] and *ast* specifications are based on context-free grammars and associate attributes with terminal and nonterminals symbols. Additionally, *ast* allows attributes which are local to rules. As the structure of the tree itself is known and transparent, subtrees can be accessed or created dynamically and used as attribute values. The access of the right-hand side symbols uses the selector names for symbolic access instead of the grammar symbols with an additional subscript if needed. There is no need to map chain rules to tree nodes because of the extension mechanism offered by *ast.* Attribute evaluation is outside the scope of *ast*. This can be done either with the attribute evaluator generator *ag* [Gro89] which understands *ast* specifications extended by attribute computation rules and processes the trees generated by *ast* or by hand-written programs that use an *ast* generated module. In the latter case attribute computations do not have to obey the single assignment restriction for attributes. They can assign a value to an attribute zero, once, or several times.

4.5. Interface Description Language (IDL)

The approach of *ast* is similar to the one of IDL [Lam87, NNG89]. Both specify attributed trees as well as graphs. Node types without extensions are called nodes in IDL and node types with extensions (base types) are called classes. *ast* has simplified this to the single notion of a node type. Attributes are treated similarly in both systems. Children and attributes are both regarded as attributes, as structural and non-structural ones, with only little difference in between. Whereas IDL in general allows multiple inheritance of attributes, *ast* is restricted to single inheritance and uses the notion extension instead [Wir88a]. IDL knows the predefined types INTEGER, RATIONAL, BOOLEAN, STRING, SEQ OF, and SET OF. It offers special operations for the types SEQ OF and SET OF. *ast* really has no built in types at all, it uses the ones of the target language and has a table containing the type specific operations e. g. for reading and writing. Both *ast* and IDL allow attributes of user-defined types. In *ast* the type specific operations for predefined or user-defined types are easily expressed by macros using the target language directly. IDL offers an assertion language whereas *ast* does not. IDL provides a mechanism to modify existing specifications. The module feature of *ast* can be used to extend existing specifications. From *ast*, readers and writers are requested with simple command line options instead of complicated syntactic constructs. *ast* does not support representation specifications, because representations are much more easily expressed using the types of the target language directly. Summarizing, we consider *ast* to have a simpler specification method and to generate more powerful features like aggregates, reversal of lists, and graph browsers.

4.6. Object-Oriented Languages

The extension mechanism of *ast* is exactly the same as single inheritance in object-oriented languages like e. g. Simula [DMN70] or Smalltalk [Gol84]. The hierarchy introduced by the extension mechanism corresponds directly to the class hierarchy of object-oriented languages. The notions *base type* and *superclass* both represent the same concept. Messages and virtual procedures are out of the scope of *ast.* Virtual procedures or object specific procedures might be simulated with procedure-valued attributes. Table 4 summarizes the corresponding notions of trees (*ast*), type extensions, and object-oriented programming.

Table 4: Comparison of notions from the areas of trees, types, and object-oriented programming

trees	types	object-oriented programming
node type	record type	class
-	base type	superclass
-	derived type	subclass
attribute, child	record field	instance variable
tree node	record variable	object, instance
-	extension	inheritance

4.7. Tree Grammars

Conventional tree grammars are characterized by the fact that all right-hand sides start with a terminal symbol. They are used for the description of string languages that represent trees in prefix form. *ast* specifications describe trees which are represented by (absolute) pointers from parent to child nodes. If we shift the names of node types of *ast* specifications to the beginning of the right-hand side and interpret them as terminals we arrive at conventional tree grammars. That is exactly what is done by the tree/graph writer procedures. They write a tree/graph in prefix form and prepend every node with the name of its node type. That is necessary to be able to perform the read operation.

5. Summary

We presented the tool *ast*, a generator for abstract syntax trees, which supports the definition and manipulation of graph-like data structures. The records which define a graph and their relationships are specified by a formalism based on context-free grammars. The data structures may be decorated with attributes of arbitrary types. The tool generates a program module containing a set of declarations to define the data structure and various procedures to manipulate it. There are procedures to construct and destroy nodes or graphs, to read and write graphs from (to) files, and to traverse graphs in some commonly used manners. The mentioned readers and writers process ascii as well as binary graph representations.

The advantages of this approach are: record aggregates are provided which allow a concise notation for node creation. It is possible to build trees by writing terms. The extension mechanism avoids chain rules and allows, for example lists with elements of different types. Input/output procedures for records and complete graphs are provided. The output procedures and the interactive graph browser facilitate the debugging phase as they operate on a readable level and know the data structure. The user does not have to attend to algorithms for traversing graphs. He/she is freed from the task of writing large amounts of relatively simple code. All of these features significantly increase programmer productivity.

References

[Bie89] F. Bieler, An Interpreter for Chandy/Misra's UNITY, internal paper, GMD Forschungsstelle an der Universität Karlsruhe, 1989.

[DMN70] O. Dahl, B. Myrhaug and K. Nygaard, *SIMULA 67 Common Base Language - Publication S-22*, Norwegian Computing Center, Oslo, 1970.

[Gol84] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, Reading, MA, 1984.

[Gro88] J. Grosch, Generators for High-Speed Front-Ends, *LNCS 371*, (Oct. 1988), 81-92, Springer Verlag.

[GrV88] J. Grosch and B. Vielsack, The Parser Generators Lalr and Ell, Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, Apr. 1988.

[Gro89] J. Grosch, Ag - An Attribute Evaluator Generator, Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Gro91] J. Grosch, Ast - A Generator for Abstract Syntax Trees, Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1991.

[JWM85] K. Jensen, N. Wirth, A. B. Mickel and J. F. Miner, *Pascal User Manual and Report*, Springer Verlag, New York, 1985. Third Edition.

[Joh75] S. C. Johnson, Yacc — Yet Another Compiler-Compiler, Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.

[Knu68] D. E. Knuth, Semantics of Context-Free Languages, *Mathematical Systems Theory 2*, 2 (June 1968), 127-146.

[Knu71] D. E. Knuth, Semantics of Context-free Languages: Correction, *Mathematical Systems Theory 5*, (Mar. 1971), 95-96.

[Lam87] D. A. Lamb, IDL: Sharing Intermediate Representations, *ACM Trans. Prog. Lang. and Systems 9*, 3 (July 1987), 297-318.

[Mar90] M. Martin, Entwurf und Implementierung eines Übersetzers von Modula-2 nach C, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1990.

[NNG89] J. R. Nestor, J. M. Newcomer, P. Giannini and D. L. Stone, *IDL: The Language and its Implementation*, Prentice Hall, Englewood Cliffs, 1989.

[Par88] J. C. H. Park, y+: A Yacc Preprocessor for Certain Semantic Actions, *SIGPLAN Notices 23*, 6 (1988), 97-106.

[SDD86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky and E. Schonberg, *Programming with Sets - An Introduction to SETL*, Springer Verlag, New York, 1986.

[WGS89] W. M. Waite, J. Grosch and F. W. Schröer, Three Compiler Specifications, GMD-Studie Nr. 166, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Wir85] N. Wirth, *Programming in Modula-2*, Springer Verlag, Heidelberg, 1985. Third Edition.

[Wir88a] N. Wirth, Type Extensions, *ACM Trans. Prog. Lang. and Systems 10*, 2 (Apr. 1988), 204-214.

[Wir88b] N. Wirth, From Modula to Oberon, *Software—Practice & Experience 18*, 7 (July 1988), 661-670.

[Wir88c] N. Wirth, The Programming Language Oberon, *Software—Practice & Experience 18*, 7 (July 1988), 671-690.

Contents