Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

R. Marti, T. Murer

GIPSY: A Generator for Incremental Programming Systems

TIK-Report
Nr. 7, June 1992

Institut für Technische Informatik und Kommunikationsnetze
Laboratory of Computer Engineering and Networks

Abstract

GIPSY (Generator for Incremental Programming Systems) is an experimental system, which produces incremental programming environments from a given language definition. Besides its tasks in a conventional environment, the compiler of a GIPSY generated system serves as a *program base* by making syntax and semantic information about a program accessible to other tools. The compiler manages a program internally in a general data model and offers well-defined editing operations on this data structure. In order to get short response times incremental compilation techniques are used. Therefore, the paper focuses on the incremental evaluation schemes used in GIPSY generated environments and discusses the steps taken during the translation of the language definition into the language-dependent part of the programming environments. Some experimental results will be shown, demonstrating the effect of incremental evaluation.

R. Marti, T. Murer:
GIPSY: An Experimental Programming System Generator
Juni , 1992
Version 0
TIK-Report Nr. 92-7

Laboratory of Computer Engineering and Networks,
Swiss Federal Institute of Technology (ETH) Zurich

Institut für Technische Informatik und Kommunikationsnetze,
Eidgenössische Technische Hochschule Zürich

Gloriastrasse 35, ETH-Zentrum, CH-8092 Zürich, Switzerland

Contents

1. Introduction

Most of today's Software Development Environments (SDEs) still use the *tool-kit* approach in providing a programming system by supporting users in the four traditional phases of software development: editing, translation from source to machine code, testing, and debugging. The tools' *interoperability* is established by using common data structures, e.g., text files, reference files. Figure 1.1 shows a typical configuration of a Modula-2 environment with four tools (text editor, compiler, linker, debugger) and their common data structures. Edges represent the producer-consumer relation.

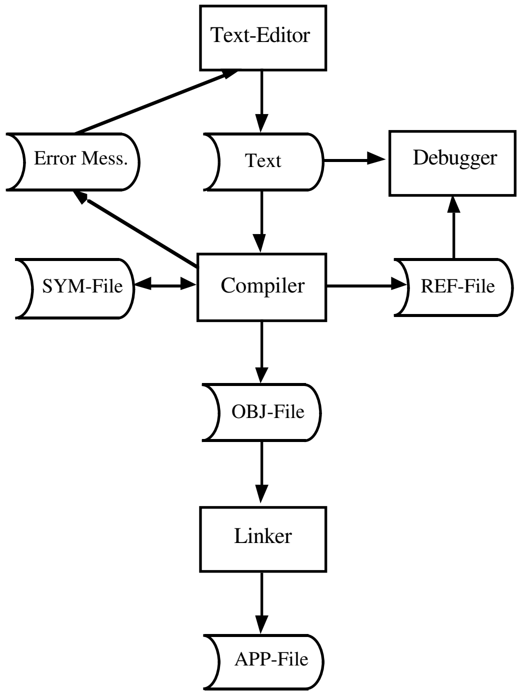

Fig. 1.1: Tools and files used by a Modula-2-System (MPW)

In such an SDE, the compiler generates nearly all common data structures. During the analysis pass, a Modula-2 compiler builds up internal data structures, which hold all interesting information about a program in a concise and compact way, e.g., a symboltable storing objects and their types. Before the compiler deallocates its internal data structures it stores parts of the data in external files, with each file serving a special purpose. The *symbol file*, for example, contains type information of exported objects used by the compiler during the compilation of another module, whereas *reference files* hold information about objects of an implementation module allowing the debugger to display the state of a module during program execution.

Because of this specialization or tool-orientation, the SDE can't be extended easily. New tools like browsers, inspectors or graphic program editors have to collect their information about a program from these files. Some tools even need to compile the source code into a new data structure, because no file holds the appropriate information or the file formats wasn't known to the tool designer. In order to achieve *consistent data structures*, after some changes have been made to the source code, every tool producing such data has to update them. This update-process can lead to a significant delay in the software development process.

Another problem in tool-kit based SDEs concerns the management and updating of *program documentation*. In most environments there is no possibility of establishing a connection between the program's source code and its external documentation that takes care of their consistency, e.g., object naming. On the other hand, documentation located only within a program isn't very satisfactory, because conceptual thoughts often concern several distant program parts, especially in object-oriented programming.

These few examples show some of the shortcomings of tool-kit based SDEs, especially their tool-oriented common data structures make them hard to extend. In the following section we show how to overcome these problems by using a centralized data-oriented, rather than a tool-oriented model, with incremental update.

1.1. Towards an Integrated Software Development Environment

The main idea behind GIPSY generated environments is the changed role of the compiler. Besides its usual tasks like syntax and semantic analysis and code generation, it serves as a *program base* by making syntax and semantic information about a program accessible to other tools. In contrast to tool-kit based SDEs, the compiler does more than export information in a tool-oriented fashion. Rather it manages a program internally in a general data model and offers well-defined operations on this data structure to external tools. This SDE model works similarly to the MVC-paradigm introduced in the Smalltalk-80 environment [Kras87]. The compiler represents the model, managing internal data, and the tools make this internal data visible to programmers and execute operations on the data caused by user actions, e.g., editing operations. Figure 1.2 shows the changed configuration.

Following the definitions of tool integration by [Thom92], GIPSY generated SDEs can be characterized as follows:

Interoperability between tools is established by the compiler (program base). All tools have a common *view of data* and there is no *redundancy* of information if the data model within the program base is free of redundancy. *Data consistency* is guaranteed, because there is only one place where information about the program is stored. To get a fast update process it is necessary that changes on the central data structure have a locally restricted effect. Therefore, incremental compilation and the design of the data model are important aspects of our compiler.

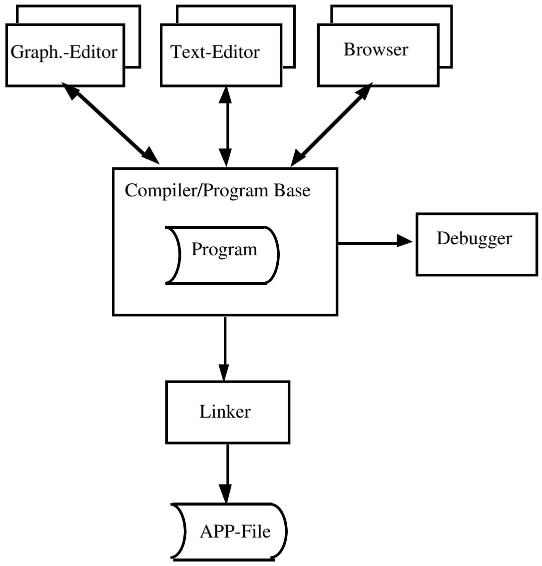

Fig. 1.2: Structure of an integrated SDE

1.2. Program Representation and Incremental Compilation

Because our aim was to implement an SDE-generator, we had to use a data model that is powerful enough to represent programs of a whole class of languages. GIPSY-generated environments use *attributed syntax trees* as their data model.

```
Statement   ::= Assignment | IfStatement | ... .
Assignment  ::= Designator ':=' Expression ';' .
Expression  ::= Term {['+'|'-'] Term} .
Term        ::= Factor {['*'|'/'] Factor}.
Factor      ::= Designator | Number | ... .
```

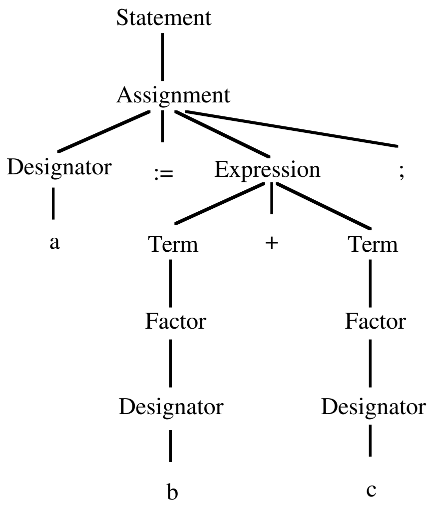

Fig. 1.3: context free grammar and syntax tree for 'a := b + c;'

Attributes, which are assigned to syntax tree nodes, are used to define additional context rules of a language, e.g., declaration rules, type rules. *Attribute Grammars* were introduced by Knuth [Knuth68] and are used in various compiler-generators for formal language definition [Kast82][Möss90]. A good bibliography on attribute grammars can be found in [Dera88]. In the last few years a lot of work has also been done on incremental evaluators for attribute grammars [Reps83] [Filé86] [Jour90]. Incremental evaluation in this context means the reevaluation of attribute values after a basic operation on the syntax tree.

1.3. Model of Editing

It is obvious that incremental updating of a syntax tree and its attributes depends on the *model of editing*, which is determined by the set of basic operations defined on the syntax tree. For example, Synthesizer Generator based environments [Reps88] use a model of editing that consists of *subtree replacement* operations. Modifying a program entails reorganizing a structure tree (syntax tree without terminal leaves) by *pruning* and *grafting* subtrees. An editing session is seen as a sequence of replacement operations and cursor motions starting from the structure tree with only one unexpanded node, labeled with the starting non-terminal of the underlying grammar.

In Synthesizer Generator environments, and in most other incremental programming systems, this model of editing not only defines an abstract way of transforming a structure tree, but also represents their user interface paradigm. *Structure-oriented* editiors primarily serve the needs of novice programmers by making the task of entering and editing programs easier and less error prone. These editors guarantee that the program fragment under construction is syntactically correct. On the other hand, professional programmers have little difficulty with syntax, they need highly interactive systems without restricted editing capabilities. This fact explains why structure-oriented program editors haven't been successful yet. Therefore, in GIPSY generated SDEs we use a fully *text-oriented* model of editing, found in every modern text-processing system, enhanced with structure expansion capabilities.

2. GIPSY Generated Environments

As mentioned above, the additional task of the compiler as a program base leads to an integrated, extensible SDE. Because the compiler now offers transformation as well as access operations on the internal data structure (attributed syntax tree), it can be seen as an abstract data type (ADT), which can be implemented, for example, as a class. In fact, in GIPSY generated environments the compiler is implemented with the class *ContextDocument* . Starting with the class *Document*, which will be explained below, the class *ContextDocument* is derived by several extension steps (*Document* > *TokenDocument* > *ErrorDocument* > *StructureDocument* > *ContextDocument*) (see fig 2.1).

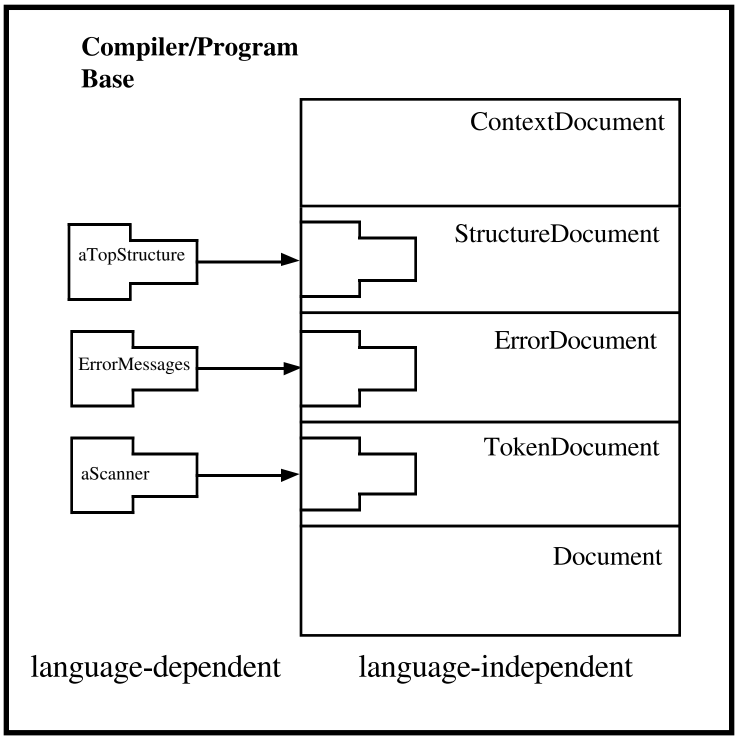

Fig. 2.1: Implementation structure of the incremental compiler

Each class implements a certain task of the incremental compilation scheme (lexical, syntax and semantic analysis). We call this class hierarchy the *language-independent kernel* of the compiler, because every class performs its task without the knowledge of the concrete language. In order to obtain a compiler for a specific language, some abstract classes (*Scanner*, *SynStructure*) have to be extended, and during the compiler's initialization phase instances of these classes have to be 'plugged' into the language-independent kernel. Therefore, only these *language-dependent* classes have to be generated from a formal language definition. In the following sections we discuss some implementation details about the language-independent kernel and the translation of a language definition into the language-dependent classes.

2.1. Concepts of the Language-independent Kernel

The language-independent kernel primarily serves two purposes:

(1) managing the internal data structure representing the program;
(2) exporting syntax and semantic information of the program.

To decompose the management of the whole data structure, each class of the kernel holds a part of it and offers the operations corresponding to the model of editing. Additionally, each class makes information of its part of the data structure accessible to other tools. In the following table each class is described in a few words:

Class	Task	Part of data model
Document	Abstract class defining the basic operations on the data structure (model of editing)	no
TokenDocument	Recognizes incrementally lexical elements of the language (tokens). Needs a scanner instance to become language dependent.	token list = two dimensional linked list
Scanner	Abstract class defining the behaviour of a scanner. Has to be extended and a concrete instance has to be 'plugged' into the TokenDocument instance.	no
ErrorDocument	Manages errors and error messages which are associated with tokens and have been detected during syntax or semantic analysis.	error list
StructureDocument	Manages the structure tree consisting of instances of subclasses of SynStructure. Uses an incremental parsing scheme. Needs the top structure (start symbol) to become language dependent.	structure tree
SynStructure	Abstract class implementing parser construction methods. Has to be extended. Concrete instances are used as nodes of the structure tree and attributes are stored as instance variables.	no
ContextDocument	Implements the incremental attribute evaluation mechanism.	attribute dependency graph

Table 1: Description of kernel classes

2.1.1. Model of Editing

As mentioned above, GIPSY generated environments have a fully text-oriented model of editing. Consequently, the basic set of operations on the internal data structure is the same as in a text processing system. The abstract class *Document* defines this set, which is extended by method overriding in every subclass, managing a data structure (*TokenDocument*, Er*rorDocument*, *Structure-Document*, *ContextDocument*). The following class definition shows the set of basic operations written in Object-Modula-2:

```
Document = OBJECT (Object)

    workingPos: Pos;
        (* working position: row, column *)

    METHOD SetPos (to: Pos);
        (* Set the working position *)

    METHOD GetPos (VAR pos: Pos);
        (* returns the working position *)

    METHOD InsertChar (ch: CHAR);
        (* insert a character at the working position and
           move working position one character forward *)

    METHOD Cut (to: Pos; VAR text: Text);
        (* cut a text range starting from the working position to the
           given position and return it *)

    METHOD Copy (to: Pos; VAR text: Text);
        (* copy a text range starting from the working position to the
           given position and return it *)

    METHOD Paste (text: Text);
        (* insert the given text at the working position and move the
           working position behind the inserted text *)

END;
```

Fig. 2.2: Definition of class *Document*

2.1.2. Incremental Lexical Analysis

Incremental lexical analysis is implemented in the class *TokenDocument*. A *TokenDocument* instance holds the lexical part of the data model called a *token list* . A token stands for a lexical element of the underlying language, e.g. identifier, number (fig. 2.3).

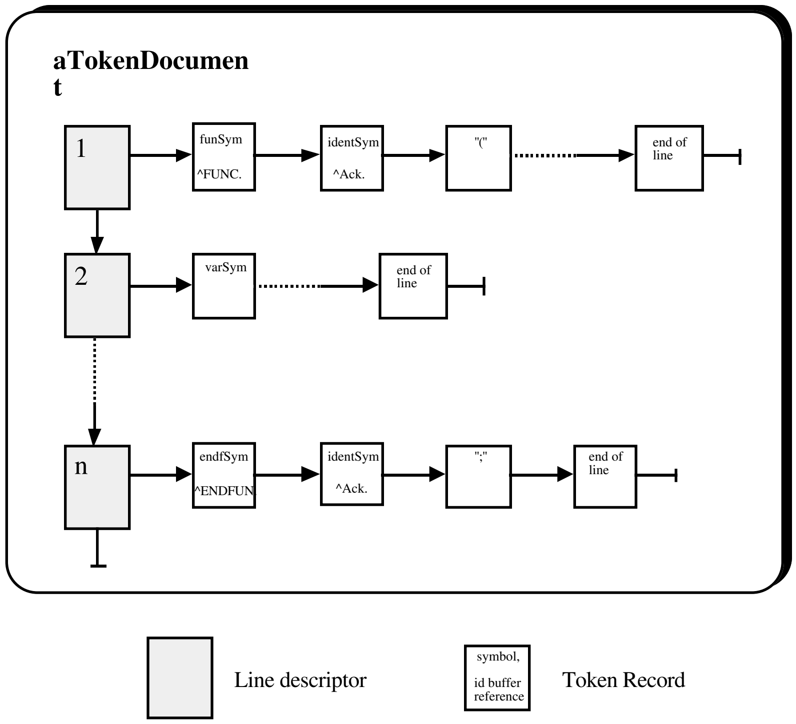

Fig. 2.3: Example of a token list

During each basic operation the token list is updated in the following manner:

(1) Tokens at the boundaries of the affected text-range are split. The affected text range is operation dependent and defined in table 2.

(2) The operation is executed directly on the prepared token list, e.g. insertion of a token consisting of one character (InsertChar) or a token range is cut out (Cut).

(3) Finally, some tokens at the boundaries of the affected text-range have to be rescanned, because the split operation or the insertion of a token range (Paste) could have changed their lexical type. The token ranges, which have to be rescanned are determined by *separator tokens* found to the left and right of split points. Separator tokens have the following property: their lexical type is independent of the character sequence to their left (right-separators) or to their right (left-separators). Thus, separator tokens restrict the effect of changes. The set of lexical types fullfilling the separator property can be identified during the analysis of the formal language definition (see section 2.2).

Operation	Affected text-range
InsertChar, Paste	The affected range consists only of the working position. Consequently, only the token holding the working position has to be split

Cut	The affected range is defined by the working position and the 'to' parameter. One token at the working position and one token at the 'to'-position has to be split.
Copy	no affected text-range

Tab. 2: Affected text range

Figure 2.4 shows an update process during a paste operation. Suppose the working position is located inside a token with lexical type *x*. During the first step token *x* is split at the working position into two tokens *x1*, *x2* with undefined lexical types. Second, the paste operation is executed by inserting a token list at the working position. Third, starting at the insertion points the rescan range is determined by searching the left and right separator token of both ranges. In our example the tokens *a1, a2* and *b1, b2* are separator tokens. Both rescan ranges are then unscanned into strings. Finally, the scanner recognizes the lexical elements and the *TokenDocument* instance inserts them as tokens into the token list. More implementation details about the incremental lexical analysis can be found in [Marti91].

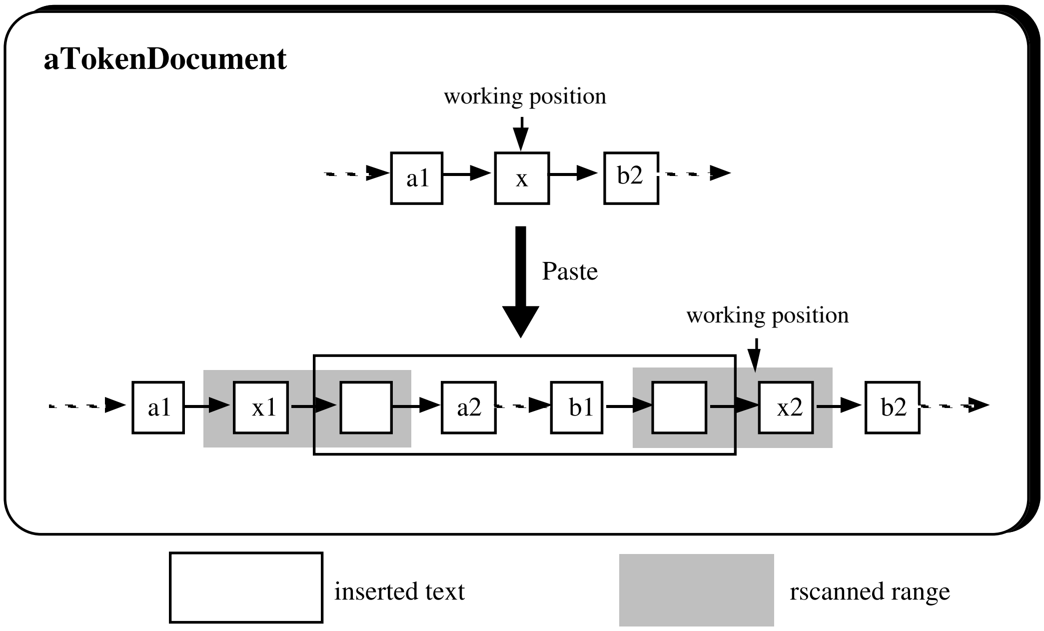

Fig. 2.4: Token list before and after a paste operation

2.1.3. Incremental Syntax Analysis

The class *StructureDocument* is responsible for executing the operations on the syntactic part of the data model represented by a *structure tree* (syntax tree without terminals). The nodes of the structure tree are instances of subclasses of the abstract class *SynStructure*. These classes correspond to nonterminals of the underlying *context-free grammar* and are generated from a formal language definition. They all have implemented a *Reparse*-Method representing their nonterminal's syntax rule. An example of a syntax rule and its implementation is given in figure 3.5.

```
IFStatement ::= 'IF' Expression 'THEN'
                    {Statement ';'}
                'ELSE'
                    {Statement ';'}
                'END'
```

```
METHOD IfStatement.Reparse;
BEGIN
    SELF.ExpectToken (ifSym, ifErr);
    (* Expression *)
    IF SELF.ExpectedSet (exprFirst, exprError) THEN
        SELF.ExpectStructure (exprType);
    END;
    SELF.ExpectToken (thenSym, thenError);
    (* Statements *)
    WHILE SELF.ExpectedSetWithStop (statemtFirst, statemtStop) DO
        SELF.ExpectStructure (statemtType);
        SELF.ExpectToken (';', semError);
    END;
    SELF.ExpectToken (elseSym, elseError);
    (* Statements *)
    WHILE SELF.ExpectedSetWithStop (statemtFirst, statemtStop) DO
        SELF.ExpectStructure (statemtType);
        SELF.ExpectToken (';', semError);
    END;
    SELF.ExpectToken (endSym, endError);
END Reparse;
```

Fig. 2.5: Syntax rule and its implementation in Object Modula-2

The class *SynStructure* offers *parser-construction methods*, which are used to implement Reparse-Methods. Parser-construction methods contain the whole incremental parsing scheme. Therefore, Reparse-Methods could be written by hand and look like procedures of a *recursive descent parser* [Waite85]. In fact, the incremental parsing scheme, which is invoked after performing the operation on the lexical level (see section 2.1.2), works similar to a recursive descent parser. The Reparse-process starts at the structure tree root by invoking its Reparse-Method and moves down the tree by calling appropriate parser construction methods within each Reparse Method. In contrast to a recursive descent parser, our parser tries to match the *current tree node*, identified by a *node-pointer*, with the *expected node type*. This is done by Method *ExpectStructure(expectedType)*. In case of a match, the node is skipped. However, if the node covers the affected text range (see table 2) it is reparsed. If the match fails, *ExpectStructure* inserts a new node into the structure tree and immediately calls its Reparse-Method. Obsolete nodes are removed during tree traversal.

A parsing mechanism like this leads to a tree traversing scheme similar to the problem of 'finding a node in a tree with a given key', because the parser visits those nodes, which cover the affected text range. Other nodes are usually matched because of the local effect of most changes (see section 3).

After detecting a syntax error, a parser usually tries to synchronize with the input token stream, so that it can proceed to find further errors (error recovery). Various error recovery strategies are known from literature [Aho86]. *Panic-mode* error recovery, for example, is based on the idea of skipping tokens on the input until a token in a selected set of synchronizing tokens appear. Its effectiveness depends on the choice of the synchronizing set. On the other hand, an incremental parser based on *structure tree matching* can use the tree to recover. Good synchronization points, for example, are *start tokens* or *end tokens* of tree nodes which are located textually behind the incorrect tree node. In fact, GIPSY generated environments uses such a recovery strategy.

2.1.4. Incremental Semantic Analysis

Semantic analysis within the language-independent kernel is based on incremental attribute evaluation. Several incremental attribute evaluators have been proposed in literature [Reps83][Filé86]. The current version uses an *incremental topological evaluation* scheme [Reps83], whereby a *dependency graph*, defining the attributes' dependencies, is used to propagate attribute changes. Attributes are stored as instance variables of the structures tree nodes (instances of subclasses of class SynStructure). The class *ContextDocument* is used to update the dependency graph during a basic operation and to reevaluate the attributes.The advantage of this evaluation scheme is its minimal set of attributes which have to be reevaluated. On the other hand, the storage overhead used by the dependency graph is a clear disadvantage. However, to keep this overhead small, it is possible to build up only those parts of the dependency graph belonging to the subtree, which the user currently manipulates. This technique leads to a *demand driven* attribute evaluation scheme with only a partially evaluated structure tree. If the user changes its focus to a different subtree, the structure tree has to be reevaluated until the newly focused subtree's attributes are up-to-date.

2.1.5. Accessing Syntax and Semantic Information

Each class of the language-independent kernel, that holds a part of the data model, offers some *access-* and *registration*-methods for external tools. Access-methods are used to make information about the current state of the data structure accessible from outside, e.g., accessing a syntactic structure (tree node) and its attributes. By contrast, registration-methods are part of the *notification mechanism* providing consistency between the data model and its external representation, e.g., the program text displayed by an editor.

2.2. Generating the Language-dependent Part

In GIPSY, languages are specified by using ESL (Environment Specification Language), which is based on the compiler specification language Cocol/R [Möss90]. Thereby, token classes, which describe the lexical part of the language, are defined by *regular expressions* . The syntax is specified by a context-free grammar and has to satisfy the LL(1) requirement.

In order to obtain an incremental programming system, GIPSY translates the language specification into the language-dependent part consisting of a scanner module, a module with error messages and modules containing the structure classes.

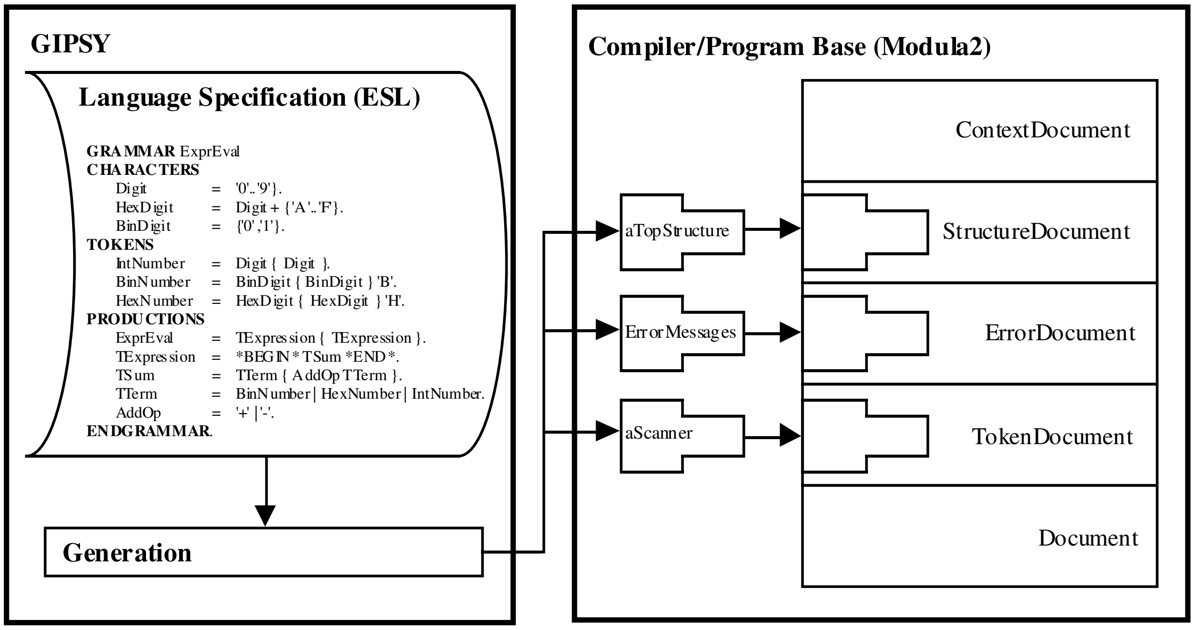

Fig. 2.6: Translation Scheme

Scanners can be implemented efficiently by using a *Reduced Automaton* (RA). GIPSY constructs a RA from a set of Regular Expressions by performing the following steps:

(1) Set of Regular Expressions -> Nondeterministic Finite Automaton (NFA)
(2) NFA -> Deterministic Finite Automaton (DFA)
(3) DFA -> RA.

The separator property of lexical types can be determined with the following considerations:

An automaton recognizing a lexical type can be represented by a directed transition graph. If two lexical types t1 and t2 recognize the same string or if t1 recognizes a substring of t2, then t1's transition graph is isomorphic to a subgraph of t2. Each lexical type with separator property must not recognize a substring of a string recognized by another lexical type. Consequently, the separator property of a lexical type can be found by matching its transition graph with the graphs of all other lexical types.

2.3 Tools

Figure 2.7 shows different tools of a GIPSY generated environment, e.g., a system browser including a program editor, a browser with a graphic view and a hypertext editor. Every tool displays parts of the same program in different ways.

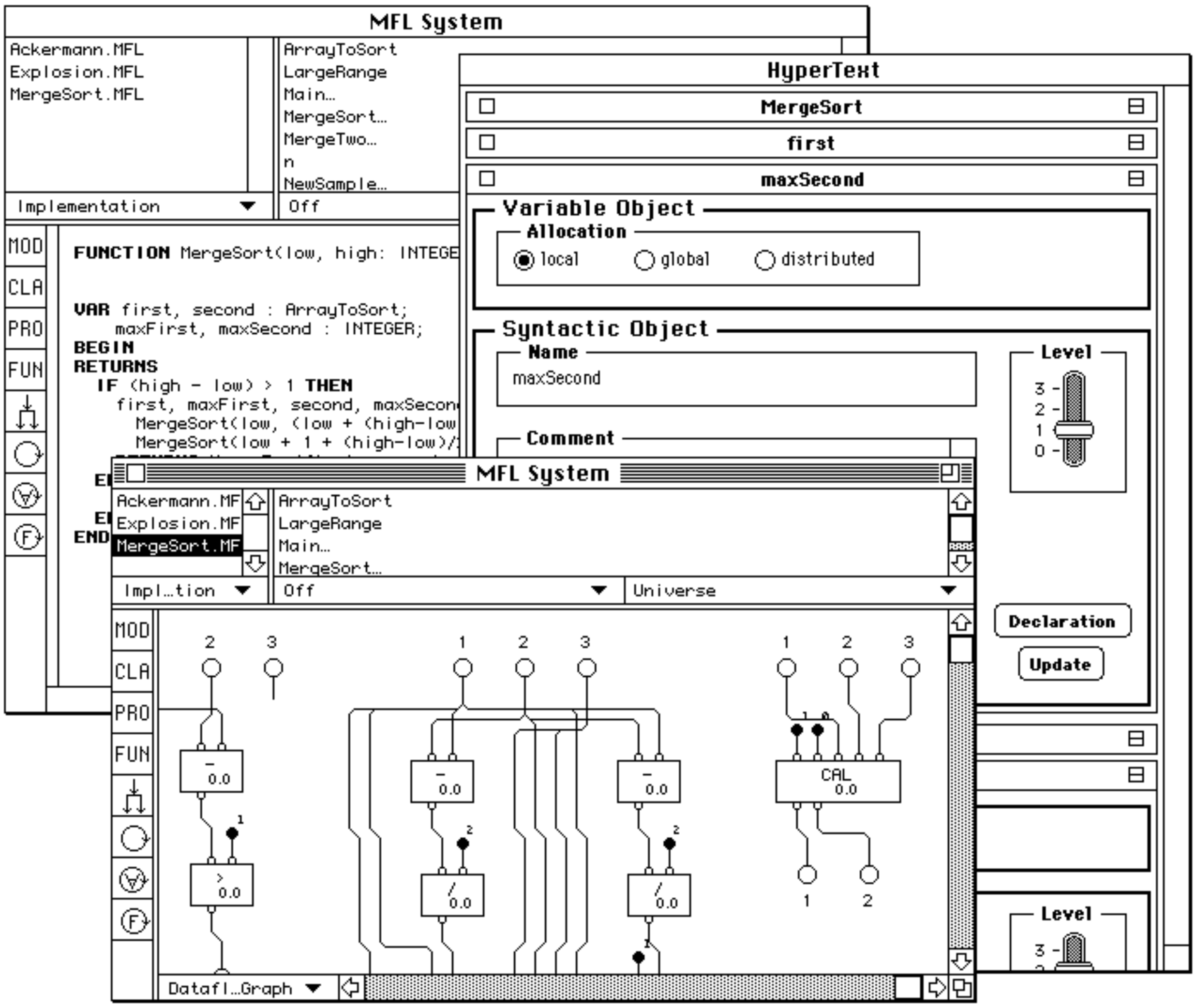

Fig. 2.7 : Different tools of a GIPSY generated environment (FOOL)

In order to o test GIPSY generated environments, special graphic views are provided by a prototype configuration.

The two browsers in Figur 4.2 display the same program fragment in three different ways (text, token list, structure tree). The token list and the structure tree are part of the compiler's data model.

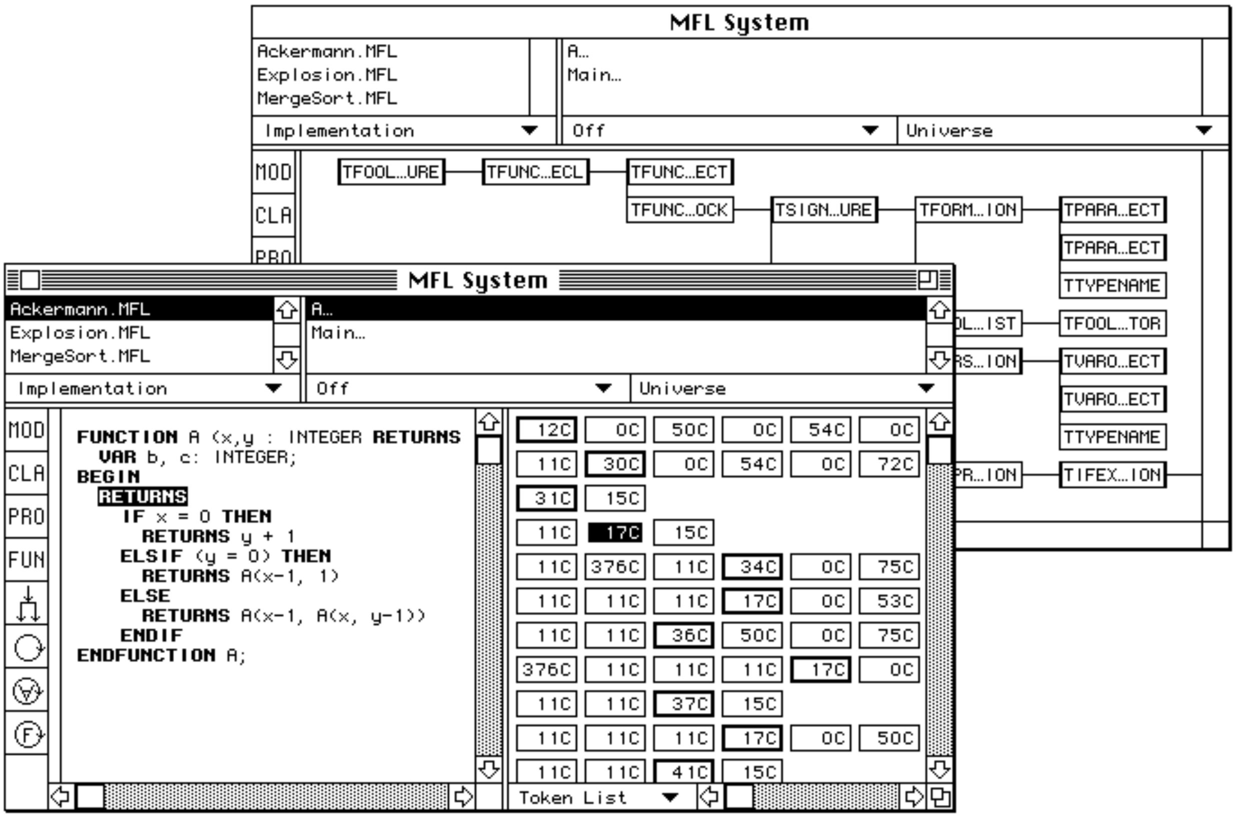

Fig. 2.8 :Views to test a GIPSY generated environment

3. Applications and Experimental Results

GIPSY has been used to produce several prototyping incremental environments but without the semantic analysis part. We are using these frontends to test the performance of the incremental lexical and syntax evaluation schemes. Complete programming systems have been developped for the FOOL language [Murer92a] and for GIPSY [Murer92b]. The FOOL system is an experimental programming system supporting users in writing programs for the ADAM multiprocessor [Maq90] developped at our laboratory. All generated environments consist of an incremental compiler, system browser and a hypertext editor and are implemented in Object-Modula2 [Henne86] on Apple Macintosh computers.

Some experimental results about the performance of incremental lexical and syntax evaluation are shown below. They are based on editing sessions in the GIPSY environment.

3.1 Incremental Lexical Analysis

The most interesting question in incremental lexical analysis concerns the size of the rescanned text during an operation. Figures 3.1 to 3.3 show some results of an editing session in the GIPSY environment. During the session, a new language definition of about two pages was entered. The x-axis represents the size of the rescanned text in characters and the y-axis shows the frequency.

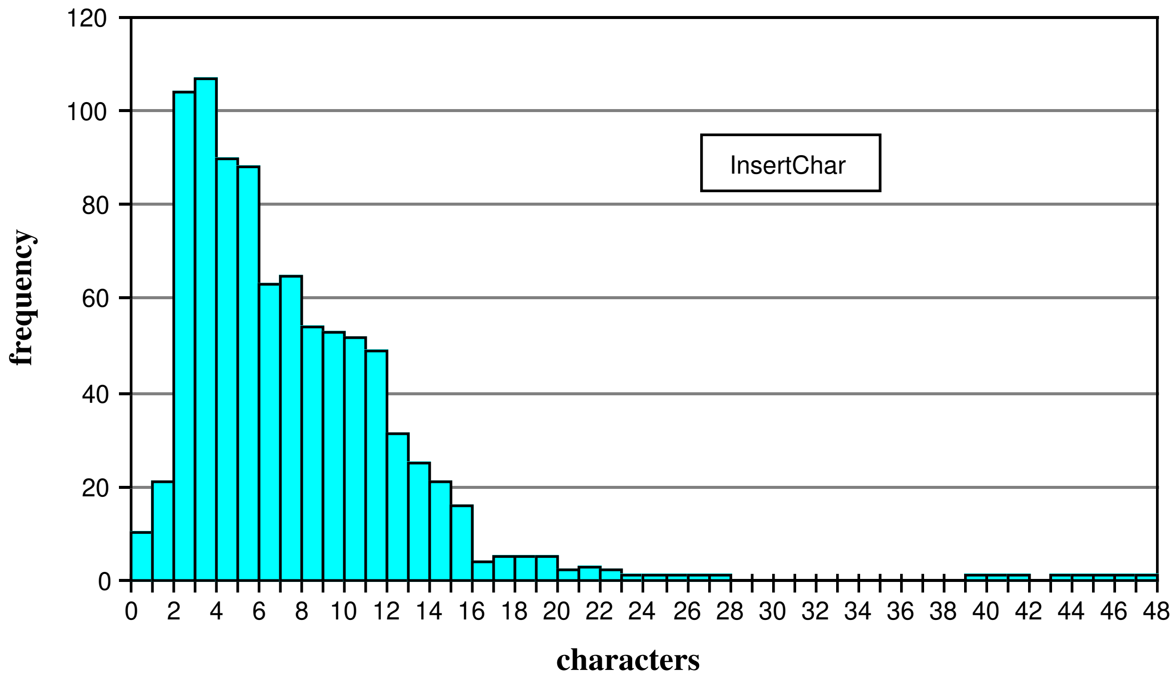

Fig. 3.1: Rescanned text during 'InsertChar' operations

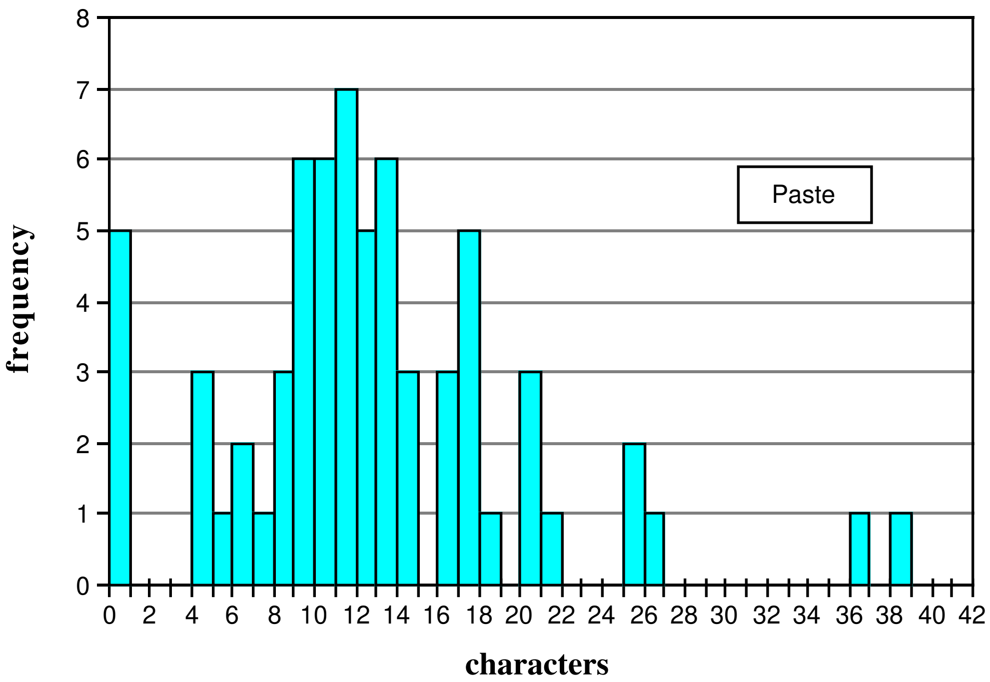

Fig. 3.2: Rescanned text during 'Paste' operations

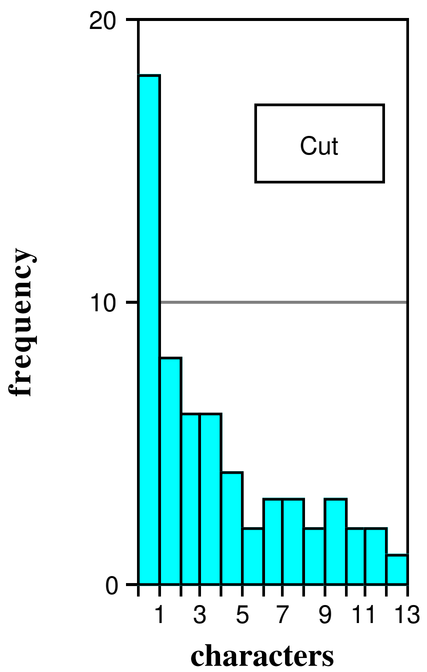

Fig. 3.3: Rescanned text during 'Cut' operations

As mentioned before, the rescan range is determined by the kind of operation and by the separator tokens. Therefore, it is important to identify every lexical type satisfying the separator property from a formal language definition. The incremental rescannig scheme lead to a rescan range of only three to fourteen characters for most 'InsertChar' operations, which is a rather good result. In very few cases, the range was about 40 to 50 characters. On the other hand, 'Cut' and 'Paste' operations caused a rescan range between zero and thirty characters.

3.2. Incremental Syntax Analysis

In order to measure the performance of our incremental parsing scheme, we recorded also the number of inserted and removed structures (structure tree nodes). In figure 3.4, the x-axis represents the number of inserted structures per operation and the y-axis shows the frequency.

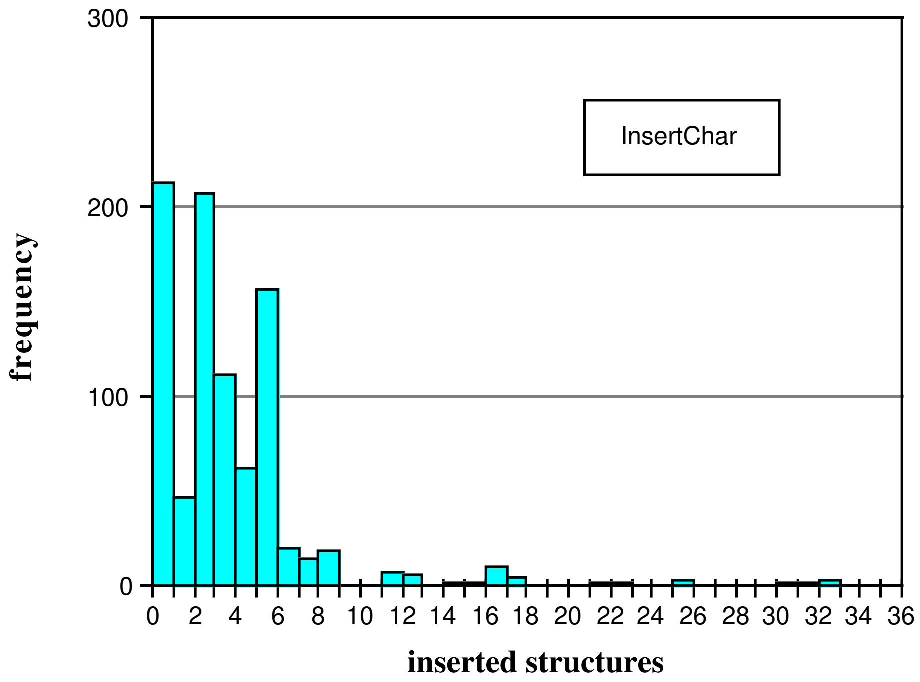

Fig. 3.4: Number of allocated structures during 'InsertChar' operations

The parser yields good results too, because most 'InsertChar' operations caused between zero and eight inserted structures only. The incremental nature of the parser can also be seen in figure 3.5. The ascending curve represents the total number of structures in the structure tree at a certain time (x-axis: operations). On the other hand, the curve near the x-axis shows the number of inserted structures during the specified operation. Peaks in this curve occured during

'Paste' operations. The figure shows that the number of inserted structures is not related to the total number of structures which is an important property of incremental systems.

Note:

total structures (op) = total structures (op-1) + inserted (op) - removed (op)

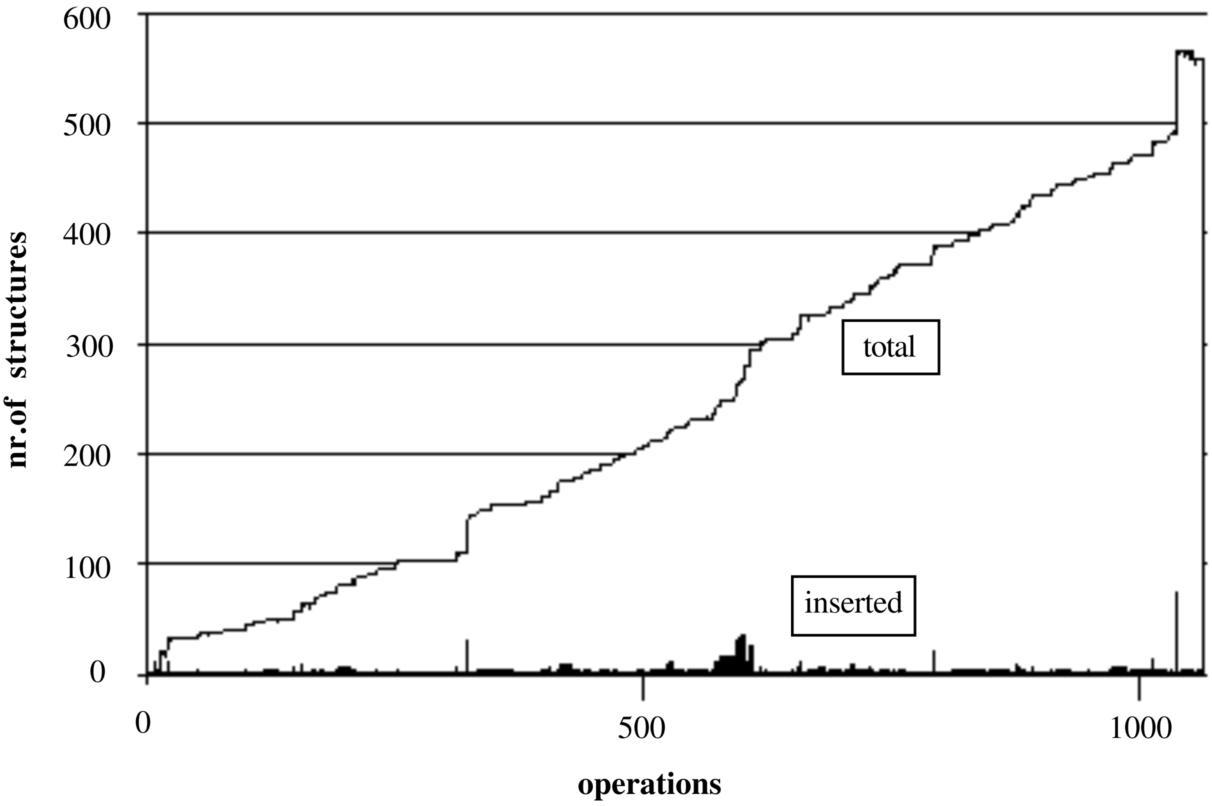

Fig. 3.5: Total and inserted structures over time

Currently, we are investigating the incremental evaluation schemes in different generated environments and with different programmers. However, the results we obtained so far are similar with the presented ones. Of course, additional experiments have to be done with the incremental attribute evaluation algorithm.

4. Conclusions

We presented some design goals and implementation details about GIPSY, a generator for incremental programming systems. In contrast to tool-kit based SDEs, a compiler of a GIPSY generated environment serves as a *program base*, by making syntax and semantic information about a program accessible to other tools. In our opinion, this different approach leads to an integrated SDE, because all tools have a *common view of data* and there is no problem of *data consistency*. *Incremental compilation* and the design of the *data model* are important aspects of such a new compiler. Also the model of editing, which is determined by the set of basic operations defined on the data model, plays an important role. We believe, that text-oriented models of editing are more appropriate for professional programmers than structure-oriented models. Experimental results showed the performance of the compiler's incremental evaluation schemes. However, further experiments based on different generated programming systems have to show the applicability of our concepts.

5. Literature

[Aho86] Aho, A., Sethi, R., Ullman, J. 1986. Compilers, Principles, Techniques and Tools. Addison Wesley.

[Dera88] Deransart, P., Jourdan, M., Lorho, B. 1988. Attribute Grammars; Definitions, Systems and Bibliography. LNCS vol. 323. Springer Verlag.

[Filé86] Filé, G. 1986. Classical and Incremental Evaluators for Attribute Grammars. 11th Colloquim on Trees in Algebra and Programming, Nice, France, March 1986, CAAP´86, LNCS vol. 214. Springer Verlag. pp. 112-126.

[Henne86] Henne, E., Schmidt, W., Wiedemann, A. 1986. p1 Modula2: Objekt-Orientierter Compiler für Apple Macintosh unter MPW. p1, München.

[Jour90] Jourdon, M., Parigot, D., Julié, C., Durin, O. & Le Bellec, C. 1990. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. Proc. of the ACM SIGPLAN´90 Conference on Programming Language Design and Implementation, White Plains, NY, 1990, pp. 209-222.

[Kast82] Kastens, U., Hutt, B., Zimmermann, E. 1982. GAG: A Practical Compiler Generator. LNCS vol. 141. Springer Verlag.

[Knuth68] Knuth, D.E. 1968. Semantics of Context-free Languages. Math. Systems Theory 2, 2, pp. 127-145. June 1968.

[Kras87] Krasner, G., Pope, St. 1987. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. ParcPlace Systems.

[Maq90] Maquelin, O. 1990. ADAM: A Coarse-Grain Dataflow Architecture that Addresses The Load Balancing and Throttling Problems. Proceedings of the Joint Conference on Vector and Parallel Processing Conpar'90. September 1990.

[Marti91] Marti, R. 1991. Design and Implementation of a Token-Editor. Proceedings of TOOLS'91, Technology of Object Oriented Languages and Systems, Paris, April 1991. pp. 349-359.

[Murer92a] Murer, S., Marti, R. 1992. The FOOL Programming Language: Integrating Single-Assignment and Object-Oriented Paradigms. Proceedings of EWPC'92, European Workshops on Parallel Computing. March 23-24. Barcelona, Spain. pp. 248-251.

[Murer92b] Murer, T. 1992. GIPSY: Eine Umgebung zur Spezifikation Inkrementeller Compiler-Frontends. Diploma Thesis. ETH Zurich, Computer Engineering Laboratory.

[Möss90] Mössenböck, H.P. 1990. Coco/R: A Generator for Fast Compiler Front-Ends. ETH Zurich. Departement Informatik, Institut für Computersysteme. Bericht 127.

[Reps83] Reps. Th. 1983. Generating Language-Based Environments. The MIT Press, Cambridge, Massachusetts.

[Reps88] Reps, Th., Teitelbaum, T. 1988. The Synthesizer Generator; A System for Constructing Language Based Editors. Springer Verlag New York.

[Thom92] Thomas, I. and Nejmeh, B. 1992. Definitions of Tool Integration for Environments. IEEE Software. March 1992. pp. 29-35.

[Waite85] Waite, W., Goos, G. 1985. Compiler Construction. Springer Verlag NewYork. Corrected second printing.