An Environment for Language Implementation called SIC

Mariza Andrade da Silva Bigonha[1]
Roberto da Silva Bigonha[2]
Valeska Gonçalves Russo[3]
Marco Rodrigo Costa[4]

Abstract

`SIC` is a programming tool whose purpose is to assist in the development of compilers by means of a special purpose language, also called `SIC` based on Pascal. `SIC` possesses facilities to specify the syntax of programming languages and to associate semantic routines with grammar productions. It also provides, without loss of eficiency, facilities to implement interactive or batch compilers organized in one or more passes, where each pass operates directly on the source code, requiring no intermediate language.

From the given syntax specification, the `SIC` tool produces a compressed `LALR(1)` table and a parser containing a language independent error handling and recovery routine. `SIC` also presents facilities to explicitly solve `LALR(1)` conflicts resulting from the use of ambiguous grammars.

`SIC` provides compilers in C and Pascal and runs on MS-DOS and WINDOWS.

[1]DSc (PUC/RJ/1994), MSc (UFMG/1985). Professor at DCC/UFMG. E-mail:mariza@dcc.ufmg.br
[2]PhD (UCLA/USA/1981), MSc (PUC/RJ/1975). Professor at DCC/UFMG. E-mail: bigonha@dcc.ufmg.br
[3]Bacharel em Ciência da Computação (UFMG julho/1996)
[4]Bacharel em Ciência da Computação (UFMG julho/1996)

1 Introduction

A programming language is, in general, very complex and full of details. For instance, to implement a compiler for a Pascal-like language without a specialized tool takes about four men-years. On the other hand, current research and experience with compilers have brought a good understanding of the compilation process and of the theoretical principles behind the techniques involved. In particular, the identification and formalization of several common phases of compilers, such as syntax analysis, lexical analysis and code generation, have permitted the automatization of the compilation process.

The idea to use tools to automatize, at least partially, the language implementation is very old. What makes the new systems different from the oldest ones is the technological advances used by them. There are actually a large number of helpful systems to support compiler construction, running on different environments. These tools vary from simple programs for automatic design of specific compiler tasks such as grammar checking and scanner generators, up to large systems capable of automatically generating a compiler from the syntax and semantics definitions of a source language.

A well known system used for parser generation is YACC [17]. It was proposed by S.C. Johnson, from Bell Laboratories. It runs on the UNIX operating system and produces compilers in the language C.

The methods of syntax analysis used in compiler implementations are based, in general, on context-free grammars. These methods are classified in two broad approaches: bottom-up and top-down. In a bottom-up method, the parsers build program parse trees in a bottom-up way, starting from the leaves and working up to the root, usually from left-to-right. If the root of the produced tree for the whole source program corresponds to the initial grammar symbol, the program is said to be syntactically correct. In top-down methods, the process used is the opposite, they build parse trees from the root to the leaves.

Among the bottom-up methods, the best known and most widely used is the `LR(1)` family [1, 2, 3, 20], originally developed by Donald E. Knuth [19]. These methods have been considered superior to the others used in compiler construction due, mostly, to their applicability, being applied to a large class of context-free grammars, including these programming languages of practical use. The most used tool, YACC, uses a `LALR(1)` method, an efficient variant of the LR(1) class.

The code generation phase in these systems follows a syntax-directed translation scheme, with semantic actions usually associated with individual productions of a context-free grammar. These routines are activated by a parser before a reduction action takes place [1]. The semantic actions, besides generating code, also manage synthesized attributes [2, 3, 1] attached in the parse tree nodes built by the parser. These attributes, associated with the grammar symbols, are used to propagate information in a bottom-up fashion, from the leaves to the root, on a parse tree. This information is used during code generation. Usually, to be more efficient, the parse tree is never effectively constructed. In its place, it is common to use a stack, which maintains only the nodes and its associated attributes whose roots have not yet been determined. As syntactical analyzes advances, nodes used and no more needed are dismissed.

Besides the Pascal version, we present in this paper the most recent `SIC` version, which generates the compiler in the language C and runs on the WINDOWS environment. The

This paper is organized as follows: Section 2 presents the general characteristics of `SIC`. Section 3 lists the computational resources necessary to run `SIC` with MS-DOS. Section 4 illustrates the generation of Pascal compilers. Section 5 presents the computational resources necessary to run `SIC` with WINDOWS. Section 6 illustrates the generation of the C compilers. Section 7

describes the facilities of the SIC language. Section 8 describes the method of syntax analysis adopted by SIC, and Section 10 introduces the method of syntax error recovery used in SIC. Section 11 compares SIC with YACC. Finally, Section 12 concludes this paper.

2 General Characteristics of SIC

SIC, (Compiler Implementation System) is a tool to support the implementation of programming languages by automatization of some phases of the compilation process. SIC is essentially the compiler of a language of higher level than Pascal [16] or C [18]. This language, also called SIC, is an extended Pascal or C, whose purpose is to provide facilities for the implementation of compilers in these languages.

The SIC language possesses the following facilities[5]:

1. The full Pascal and C languages are supported.
2. Semantics actions can be expressed easily, with direct references to grammar symbols and their attributes.
3. Declaration and use of synthesized attributes are supported, with respect only to the grammar and with total abstraction of the corresponding parse tree.
4. A grammar description in a BNF like notation is accepted.
5. Permits to associate tokens or syntactic units of this type with the objective of establishing communication between the lexical analyzer defined by the user and the syntax analyzer generated by the system.

SIC possesses yet the following facilities:

1. It generates compilers in Pascal and C, i.e. it translates programs from the SIC language to Pascal or C.
2. It generates an LALR(1) parser with compressed tables [10].
3. It produces, upon an user's option, an LALR(1) parser with automatic syntax error recovery.
4. It produces, upon an user's option, an LALR(1) interactive parser embedded with a program editor or without automatic error handling routines in its Pascal version.
5. It verifies undefined and useless symbols of the input grammar, i.e. verifies if the grammar is reduced [2].
6. It produces several outputs: source program, grammar, undefined or useless symbols, grammar cross-reference, LALR(1) table, LR(0) machine [3, 1] and information about the generated parser.

Most of the constructions introduced have the objective of facilitating the expressing of the compilation mechanisms. Nevertheless some were incorporated in SIC trying to suppress Pascal and C drawbacks in relation to modular programming.

[5]Every statement in this paper is valid for both generated compilers, C and Pascal, unless otherwise explicitly indicated.

For instance, with SIC it is possible, when declaring global variables, to place them in the declaration section of the program taking into account their place to use. The same feature exists for other types of Pascal and C declarations, allowing the user to simulate modules, i.e. allowing to group several constants, types, variables, labels and procedure declarations, letting SIC do the job of recognizing these declarations and giving them the correct interpretation.

3 The Pascal Version running on MS-DOS

The SIC system was implemented in Turbo Pascal language on IBM-PC like, under an operating system compatible to Microsoft MS-DOS. It requires, at least, 265KB of memory to execute and a disk unit of 360KB. Adittional space on disk may be necessary depending of the application size. With a 256KB of internal memory, it is possible to generate a LALR(1) parser with about eight hundred states for the ADA language.

The SIC design and development begun in 1983, and its first version, 1.1, was released in 1985 with the presentation of a master thesis [4]. The first results of this work was published in [4, 10, 6, 5, 7]. Since that, new facilities were implemented and incorporated in the system. Additionally, its interface was completely re-projected. It adopted the interface package implemented by Roberto Bigonha [9].

4 The Generation of Compilers in Pascal

The compilation of a program in the SIC language is done in four independents passes. The first one receives as input a source program in the SIClanguage, stores it in a file with extension .SIC and from it produces output files with Pascal declarations, the grammar in its internal form, the production table and one table for terminals and nonterminals symbols. The Pascal declarations found in the source files are stored in separated files according to their kind: labels, constants, types, variables and procedures. Also are included in the procedure's file, the procedure produced by SIC, with the semantics routines, the procedure with the algorithm of the parser and the main body of SIC.

The second pass receives as input a symbol table and the grammar in its internal form produced in the first pass and produces as output a file containing the LR(0) table and another containing the LALR(1) table compressed.

The third pass gets as input the declaration files generated in the other passes and put everything together, producing as output a procedure or a complete program in Pascal.

To satisfy Turbo Pascal restrictions, the compiler produced is divided in four parts and stored in files with the extensions .DCL, .SEM, .PRS and .PRO respectively. Each file may contain one source of approximately 61.000 bytes. It is the user responsibility to join these files.

Finally, the fourth pass treats the compilation of the Pascal program to produce an executable module with extension .EXE, that together with a LALR(1) parsing table plus the production table produced before forms the desired compiler.

5 The C Version running on Windows

The C version of the compiler generated by SIC uses the language C [18] and was developed between August/1995 and June/1996. It runs on WINDOWS on a 16 or 32 bits PC. The original C version was implemented in Delphi 1.0 on a 16 bits PC under the WINDOWS operating system. Today it runs also on Delphi 2.0, on 32 bits PC under the WINDOWS. It requires for

execution at least version 3.1 of the WINDOWS operating system, a 80386 processor, 4MB of RAM and 1MB of disk space to work [8]. In this case, adittional space on disk may be necessary depending on the application size.

6 The Generation of Compilers in C

To generate a compiler in C we follow the same procedure shown in Section 4 except for the files generated in the third pass. The result of this pass is a complete program in C. This program is divided in four files produced in the following way: the declaration files are combined in a single file with extension ".h" . In this file are included the function and procedure prototypes defined by the user, besides the ones generated by `SIC`. The files containing the semantic actions plus the other procedures are already in a separated file, both with extension ".c" . The four file names are constructed using the following criterion: the first three characters of the compiler name given by the user are followed by the suffix: `GLB.H, SEM.C, PRO.C` and `PRS.C` where:

- `XXXGLB.H` represents the global definitions.
- `XXXSEM.C` represents the semantic routines.
- `XXXPRO.C` represents the procedures and functions.
- `XXXPRS.C` represents the parser file and the main body of the compiler.

Finally, the fourth pass treats the compilation of the C program to produce an executable module with the extension .EXE, that together with a `LALR(1)` parsing table plus the production table produced before forms the desired compiler.

7 The Language `SIC`

A `SIC` program possesses a header followed by several sections and finishes by the keyword `%%END`. These sections may, in principle, occur in any order, it must only follow the rule that a declaration must always precede its use. Each section begins with a proper keyword and is valid until the beginning of the next section. The keywords of `SIC` always begin with %%. The sections of `SIC` have the following functions:

1. To specify the type of a compiler.
2. To specify the number of passes.
3. To define the tokens.
4. To declare the attribute symbols.
5. To declare labels, constants, variables, procedures and functions.
6. To define scope-map.
7. To define non-terminals for handling error recovery.
8. To specify grammar and semantic actions.
9. To solve conflicts in syntax analyis.
10. To specify the main body of the compiler.

7.1 Header

The header, which begins with the keyword `%%COMPILER`, specifies if the desirable compiler is a program or a procedure. Examples:

1. To generate a main program in Pascal:
   `%%COMPILER` program MR `%%BATCH`:
   ...
   `%%end`.

2. To generate main program in C:
   `%%COMPILER` main `%%BATCH`:
   ...
   `%%end`.

3. To generate a main program in C:
   `%%COMPILER` `%%BATCH`:
   ...
   `%%end`.

4. To generate a procedure in Pascal:
   `%%COMPILER` procedure MR `%%BATCH`:
   ...
   `%%end`.

5. To generate a procedure in C:
   `%%COMPILER` MR `%%BATCH`:
   ...
   `%%end`.

7.2 Kinds of Compiler

This section specifies the options that can be used to generate different kinds of compilers. In the C version the only option available is `%%BATCH`. In the Pascal version the options are:

1. `%%INTERACTIVE NOREC`
   Indicates that an interactive compiler should be generated together with a text editor but without automatic syntax error recovery.

2. `%%INTERACTIVE REC`
   Indicates that an interactive compiler should be generated together with a text editor and syntax error handler.

3. `%%BATCH`
   Indicates that a "batch" compiler should be generated together with or without automatic syntax error handler.

4. `%%INCREMENTAL`
   Indicates that an incremental interactive compiler should be generated together with a text editor.

7.3 Number of Passes

This section is used to indicate the number of passes a generated compiler should have and to identify each pass:

`%%FIRST PASS`
: Indicates that the compiler will have several passes and delimits the start of the identification or first pass.

`%%OTHER PASS`
: Indicates the start of this identification of other passes of the compiler. If this section is omitted, the system assumes that the compiler has a single pass. For each kind of compiler described in Section 7.2 there exists one specific parser. These parsers are implemented as procedures in Pascal or C read by `SIC` and incorporated in the generated compiler Pascal or C at the proper place.

7.4 Definition of Tokens

The goal of this section, which begins with the keyword `%%TOKENS`, is to establish the correspondence between the syntax analyzers generated by `SIC` and the lexical analyzer `YYSCAN` written by the user. This procedure is invoked by the parser of the generated compiler each time it needs a new token or a symbol of the source code is required to continue the compilation. For each token recognized, `YYSIMB` must return its type and, in some cases, additional information about it, e.g., its attribute values. Besides that, it is necessary that the parser recognizes each token type in order to establish the correspondence between each token that appears in the grammar and the corresponding type returned by `YYSCAN`. This section establishes the mapping of tokens and their types.

Example:

```
%%TOKENS
        "ident"  = TypeIdent;
        "const"  = TypeConst;
        "<"  = less;
        "eof"  = YYEOF;
```

The symbols at the left-hand-side of the equal sign are grammar token symbols and the identifiers on the right-hand-side are names of integer constants, whose definitions will be automatically generated by `SIC`, that represent the type of the associated symbols. There is no restriction in the name of identifiers used, except by the mandatory presence of a reserved type `YYEOF`, which identifies the end-of-file of the source program.

The procedure `YYSCAN` returns the type of each token read using an integer global variable `YYSIMB` pre-declared by the `SIC` system. The returning of additional information about the token is done by direct setting of token attributes, as shown in the next section.

7.5 Declaration of Attributes

This section, which begins with the keyword `%%STACK`, is used to declare the synthesized attributes of the terminal and non-terminal symbols of the grammar. Each attribute can be seen as a `Pascal` record, which is attached to a corresponding symbol node in a parse tree. In the C version, each atributte can be seen as a *typedef struct*. At this point occured the only syntax modification on the SIC language. This happens because in C the type identification of the

attributes may have more than one name, for instance, *expr = (r, type: unsigned int)*, while in Pascal the identification of the attribute type is composed by only one word like *char*, *integer*, *boolean*, etc.

From the attribute declaration, `SIC` creates in the compiler a stack whose elements may have any of the declared attributes. This stack is used to store the parse tree nodes that have not been completely processed, e.g., it stores the parse tree portion that must be processed.

Example:

```
%%STACK 200 OF ATTRIBUTES
        expr = (addr : integer; mode : TMode);
        "ident"  = (value : TValue) { "ident" .value := empty }
        "const"  = (value : integer) {"const" .value := 0 }
```

The integer value 200 corresponds to the stack size. For every grammar symbol that has an attribute associated with it, the user must specifies its name, the equal sign and, inside parentheses, its attributes with its respective types. The type of an attribute must be an identifier according to the language, C or Pascal.

Some terminal symbols have their attributes defined by semantic actions, or more frequently, by the procedure `YYSCAN` with an assignment of the form: "*ident*" *.value := StringRead.*

`SIC` should make sure that the attribute value defined above has been put in the node corresponding to "ident" in the parse tree when the parser is activated, and that the above token is found in the source code.

The text between braces shown in the example is optional and denotes initializations. These actions are executed only when the corresponding symbols are inserted in the source file as a result of a syntax error recovery action. This facility allows to assure that the stack contains only well defined values.

7.6 Declarations

This section is composed by subsections of Pascal declarations for labels, constants, variables, procedures and functions. Each of these subsections begins with a proper keyword: `%%LABELS` for labels, `%%CONSTANTS` for constant definitions, `%%VARIABLES` for variable declarations, `%%TYPES` for type definitions and `%%PROCEDURES` for Pascal procedure and function declarations. This is also valid for C, excluding the `%%LABELS` subsection, not present in C, and the prototypes of procedure declarations.

Examples:

```
%%LABELS1000, 9999
%%CONSTANTS
            MaxId = 8;
            MaxTs = 500;
%%TYPES
        TMode = (ExprConst, ExprVar);
        TValue = array[1..MaxIdent] of char;

%%VARIABLES
             X : integer;
             Y : real;
```

7.7 Scope Map

This section begins with the keyword `%%SCOPEMAP`. It shows the delimiters of nested constructions of a source language, such as procedures, blocks, parentheses expressions, etc. The knowledge of these delimiters allows the implementation of a sophisticated mechanism of automatic syntax error recovery [4, 5, 11, 12]. This section is optional.

Example:

```
%%SCOPEMAP
        "(" : ")" ;
        "begin" : "end" ;
        "while" : "do" ;
```

7.8 Non-terminals for Error Recovery

This section, which begins with the keyword `%%NTMAP`, allows to identify the non-terminal symbols of the grammar which can be used in the syntax error recovery phase as candidate for insertion at the point of error, substituting the erroneous symbol. If this section is omitted only terminal symbols are allowed as insertion or exchange symbols.

Example: `%%NTMAP` Expr, Decl, Cmd.

7.9 Grammar and Semantics Routines

This section begins with the keyword `%%GRAMMAR`. It allows the definition of the grammar rules of the source language and the semantic actions associated with them. From the grammar, `SIC` generates parse tables which directs the `LALR(1)` algorithm of parser. The semantic actions, thenselves, are collected, translated to Pascal or C and merged with the generated compiler, to be activated in the moment an associated production is used in a reduction action during the syntax analysis. The semantic rules enclosed in braces following each production are composed by a sequence of Pascal or C statements, which allows qualified references to symbols that appear in the production associated. Ambiguous references to symbols that occur more than once in some productions are solved by indexing, which distinguish between the desired occurrence of the symbol. The i-th occurrence of a symbol in one production must be indexed by an integer greater than zero.

Example:

```
%%GRAMMAR  prog AND SEMANTICS
        prog = expr;
        expr = expr "+"  expr
                    { expr.addr := GenTemp;
                    Gen("ADD" , expr[1].addr, expr[2].addr, expr[3].addr)
                    if (expr[2].mode = ExprConst) and (expr[3].mode = ExprConst)
                    then expr.mode := ExprConst
                    else expr.mode := ExprVar; }
```

In this example, `expr.addr, expr[1].addr` and `expr.mode` indicate the attributes `addr` and `mode` of the symbol `expr` which appears on the left-hand-side of the corresponding rule;

`expr[2].addr, expr[2].mode` indicate the attributes of the second `expr`; `expr[3].addr, expr[3].mode` denote the attributes `addr` and `mode` of the third occurrence of `expr` in the rule. Note that `expr.addr` is equivalent to `expr[1].addr`.

7.10 Conflict Resolution

Context-free grammars are very useful to define with precision the syntax of a programming language and provide an effective way for the generation of deterministic syntax analyzers, for instance `LALR(1)`. There are situations where programming language construct would be more compactly and naturally specified if an ambiguous context-free grammar could be used. In this case, the `LALR(1)` method will indicate conflict actions when the grammar is ambiguous and there is no way to solve these conflicts looking only at the information provided by the grammar. On the other hand, some conflict actions may be solved directly by the user, which has additional information, such as knowledge of context and operator priorities.

`SIC` permits to provide an `LALR(1)` with information about priority and associativity of binary operators, and the ordering of certain actions, allowing the use of ambiguous grammars. So, as soon as the conflicts are detected, the `LALR(1)` generator uses this information to solve the conflicts, giving preference to high priority actions. The priority of shift action [3, 1] is given by precedence and associativity of the read symbol, if the user had specified them. The precedence of a reduction action [3, 1] is the same as that of the rightmost terminal symbol in the corresponding production.

The declaration of precedence and associativity are made by clauses beginning with the keywords: `%%RIGHT`, `%%LEFT` and `%%NONE`. The keyword `%%RIGHT` declares an operator to be right associative; `%%LEFT` declares an operator to be left associative, and `%%NONE`, when association is not possible. In the conflict resolution section, which is identified by the keyword `%%CONFLICTS`, the clauses bellow must be specified, giving precedences in the order in which they appear in the declaration, lowest first.

Example:

```
%%CONFLICTS
        %%NONE "<" , ">" , "="
        %%LEFT "+" , "-"
        %%LEFT "*" , "/"
        %%RIGHT "**" ;
```

7.11 Main Section

This section, which begins with the keyword `%%PROGRAM`, defines the main body of the compiler. This body is a list of Pascal or C statements, containing necessarily the activation of the procedure `YYPARSER`, pre-defined by `SIC` and which represents a call to the parser.

8 The `LALR(1)` Table

An `LALR(1)` parser consists of an input, an output, a basic algorithm, a stack, and a parsing table. The algorithm is fixed, and each grammar has its own table. This table, in its normal form, is a sparse matrix, where the row indexes represent possible states names, and the column indexes represent terminal and nonterminal symbols of the underlying grammar. For a programming language of real size like `Pascal`, an `LALR(1)` parsing table may have 300 X 100 entries. Each entry represents one possible parsing action: shift, reduce, accept or error.

The generation of an LALR(1) parser consists in producing the parsing table from a given grammar. To do that, SIC uses an algorithm proposed by Kristensen [20], extended to manage ambiguous grammars. In this algorithm, conflict resolution is done by first looking at the LR(0) machine [3, 1, 20], and if the conflicts still remain, SIC uses information about priorities specified by the user in the conflict resolution section.

The LALR(1) table in its original, rectangle matrix form, spends too much space. In practice, most of the entries are empty, i.e. they denote error actions, which permits the application of a more efficient storage methods. The compactation method for LALR(1) parsing tables used in SIC, which is described in detail in [10], is based on intrinsic properties of the parsing method, and allows LALR(1) tables' needs of space to be substantially reduced without compromising table accessing time. The reduction in memory occupancy is claimed to be greater than 96% of the area of the original matrix representation. For instance, the compacted parsing table for the ADA language requires only 12Kbytes of memory space to encode all the 800 LALR(1) parser states.

9 The Program Editor

One kind of parser generated by SIC possesses a text editor embedded with it. This allows the generation of interactive compilers like Turbo Pascal [15], which, when a syntax error is detected the user automatically sees the source program at the error position in edition mode. The SIC text editor was implemented by Eduardo Costa e Silva [13], from the Borland Turbo Editor ToolBox [14], but it has not yet been integrated to the system.

10 Error Recovery

The automatic syntax error recovery method used on SIC is based in the ideas of Burke & Fisher [11, 12]. In this method, the best action of error recovery considered is that which permits the compiler to go further in the source file. This method is implemented using three strategies [4, 5]: initially, it tries to insert, delete or replace a symbol at the point of error or before it. Secondly, if it fails, it tries to close the nearest scope using information given in the section %%SCOPEMAP (Section 7.7). Finally, if it also fails, the third strategy consists of deleting pieces of source code starting with the symbol that detects the error in conjunction with insertion, deletion and replacement of symbols on the left-hand-side of the erroneous symbol. After all these tries, the system selects to correction the candidate which permits the parser to advance farther from the error position. Just one of the candidates is efectively used by parser and the other ones are shown to the user with the objective of facilitating the real identification of the error.

11 Comparison With Another Systems

The compiler implementation system that most approximates SIC is YACC [17]. Nevertheless, the SIC system is more powerful than YACC in the following aspects:

1. The syntax error recovery mechanism used in SIC is more transparent and produces better error messages. The recuperation method implemented in SIC does not make any restriction in the use of default reductions and its use proved that it does not disturb the error recovery mechanism [5].

2. The scope of SIC options is greater than YACC, it permits the generation of several kinds of compilers, for instance, an interactive compiler.

3. SIC, in its earlier versions, executed under operation systems compatible to SISNE of SCOPUS, which, at that time, was an advantage because its widspread in Brazil with respect to Unix. Today it still operates in MS-DOS, as well as under the WINDOWS environment.

4. The syntax table compression method of SIC is more efficient than the one use in YACC and the one proposed in [1].

5. The association of semantic routines with initialization of grammar symbols used in error recovery allows insertion or substitution of nonterminal symbols during error recovery.

The YACC system produces compilers written in the language C, while SIC produces compilers in Pascal or C, giving the user the option to choose the language in which the generated compiler is implemented. Taking to account that compilers in C have been implemented more efficiently, producing code of higher quality than Pascal, this certainly turns the C version of SIC more efficient than its Pascal version. Nevertheless, both the Pascal and C versions of SIC have proved to be very useful up to now, mainly, on compiler courses. Its output is more readable than that of YACC, what makes it easier to follow the LR(0) and LALR(1) parsing tables if one has to analyze the output. For instance, on the presence of conflict actions.

12 Conclusion

A tool to help compiler implementation under WINDOWS and MS-DOS systems has been presented in this paper. Its most important contributions are related to the automatization of several aspects of the compilation process, the production of a support tool, the implementation of languages and the proposal of new techniques of language and compiler implementation systems.

The Pascal version of SIC is complete with all options listed in this paper. SIC can also produce compilers in C in %%BATCH mode. Developments are still necessary in the C version, for instance, to finish the generation of interactive compilers with or without error recovery; in particular, it is planned to convert the implementation fo SIC to C and to join the text editor to its C version. Additionally, the SIC system does not runs under UNIX operational system yet, but this will be in the near future.

References

[1] Alfred V. Aho, R. Sethi, and J. D. Ullman. *Compiler Principals, Techniques and Tools.* Addison Wesley Publishing Company, 1986.

[2] Alfred V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1 and 2. Prentice-Hall, Englewood Cliffs N.J., 1973.

[3] Alfred V. Aho and J. D. Ullman. *Principals of Compiler Design.* Addison Wesley Publishing Company, 1977.

[4] Mariza A. S. Bigonha. Sic - sistema de implementação de compiladores. Master's thesis, Universidade Federal de Minas Gerais, Julho 1985.

[5] Mariza A. S. Bigonha and Roberto S. Bigonha. Uma experiência na implementação de um recuperador de erro lr(1). In *Anais do V Simpósio sobre Desenvolvimento de Software Básico, Sociedade Brasileira de Computação*, pages 158–171, Belo Horizonte, Minas Gerais, 1985.

[6] Mariza A. S. Bigonha and Roberto S. Bigonha. Sic: Sistema de implementação de compiladores. Série de Monografias T02/86, Departamento de Ciência da Computação, UFMG, 1986.

[7] Mariza A. S. Bigonha and Roberto S. Bigonha. Sic: Um sistema de suporte à implementação de compiladores. In *Anais do VI Congresso da Sociedade Brasileira de Computação*, pages 429–442, Recife - Pernambuco, 1986.

[8] Mariza A. S. Bigonha et al. Sic: Sistema de implementação de compiladores - manual do usuário, versão c 1.0. Relatório Técnico 020/96, Departamento de Ciência da Computação, UFMG, 1996.

[9] Roberto S. Bigonha. Window 2.0: Um sistema básico de gerência de interface. Relatório Técnico 011/90, Departamento de Ciência da Computação, UFMG, 1990.

[10] Roberto S. Bigonha and Mariza A. S. Bigonha. Um método de compactação de tabelas lr(1). In *Anais do III Seminário sobre Desenvolvimento Software Básico, Sociedade Brasileira de Computação*, pages 141–157, Rio de Janeiro, RJ, 1983.

[11] M. Burke and J.A. Fisher. A practical method for syntatic error diagnosis and recovery. In *CACM*, 1982.

[12] M. Burke and J.A. Fisher. A practical method for lr and ll syntatical error diagnosis and recovery. In *TOPLAS*, April 1987.

[13] E. Costa e Silva. Um editor para ambientes de programação. Master's thesis, Universidade Federal de Minas Gerais, 1988.

[14] Borland International. *Turbo Editor Toolbox*. Borland, 1985.

[15] Borland International. *Turbo Pascal V.3.0*. Borland, 1985.

[16] N. Jensen, K. & Wirth. *User Manual and Report*. Springer-Verlag, 1974.

[17] S. C. Johnson. Yacc - yet another compiler compiler. Technical Report 32, Computer Science, AT & T Bell Laboratories, Murray Hill, New Jersey, 1975.

[18] Brian W. Kernighan and Dennis Ritchie. *The C programming language*. Prentice Hall., 2 ed., 1988.

[19] D. E. Knuth. On the transaction of languages from left to right. *Information and Control*, 8:607–639, 1965.

[20] O.L. Kristensen, B.B. & Madsen. Methods for computing lalr(k) lookahead. *ACM Transactions on Programming Languages and Systems*, 3(1):60–83, January 1981.