A survey on attibute grammars.Part II review of existing systems

Pierre Deransart, Martin Jourdan, Bernard Lorho

INRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France

Tél. : (1) 39 63 55 11

Rapports de Recherche

N° 510

A SURVEY ON ATTRIBUTE GRAMMARS

PART II
REVIEW OF EXISTING SYSTEMS

Pierre DERANSART
Martin JOURDAN
Bernard LORHO

Mars 1986

A SURVEY ON ATTRIBUTE GRAMMARS
PART II
REVIEW OF EXISTING SYSTEMS

Pierre DERANSART, Martin JOURDAN, Bernard LORHO(*)
INRIA-ROCQUENCOURT
BP 105
78153 LE CHESNAY Cedex
FRANCE

(*)Also at : Université d'Orléans, Laboratoire d'Informatique,
45046 ORLEANS Cédex, FRANCE

Abstract : This review is the second part of a Survey on Attribute Grammars consisting of three parts :

- Main Results on Attribute Grammars
- Review of Existing Systems
- Classified Bibliography

Keywords : Attribute Grammars, Evaluation, Semantics, Systems

Résumé : Cette revue est la seconde partie d'une Etude sur les Grammaires Attribuées qui en comporte trois :

- Principaux Résultats sur les Grammaires Attribuées
- Revue des Systèmes Existants
- Bibliographie Classée.

Mots-Clés : Grammaires Attribuées, Evaluation, Sémantique, Systèmes.

Preliminary Notice : all the references quoted in this paper are listed in the third part of the whole survey, the Classified Bibliography.

CONTENTS

INTRODUCTION	01
APARSE	04
ATHEN'S SYSTEM	07
CIS	10
COPS and COPS-2	14
CWS	19
DELFT'S SYSTEM	24
DELTA	27
ELMA	33
FNC/ERN	36
FOLDS	44
GAG	48
HLP78	54
HLP/SZ	59
LILA	62
LINGUA	66
LINGUIST-86	70
MUG1	75
MUG2	79
NEATS	88
PARSLEY	93
SAGET	96
SSAGS	99
SUPER	103
SYNTHESIZER GENERATOR	106
TALLINN'S SYSTEM	113
TOKYO'S SYSTEM	117
VATS	123
YACC	129
OTHER SYSTEMS	133
CONCLUSION	137

INTRODUCTION

1) Aims of this review

Attribute Grammars (AGs) have proved to be a pleasant and efficient tool to describe syntax-directed computations, in particular compilers and translators (see the Classified Bibliography, sections "applications" and "languages"). However a specification with no corresponding implementation is rather useless. Much work has thus been done to supply programs (attributes evaluators) which could execute the computations described by AGs.

The theoretical part of this work is presented in the first part of this survey (Main Results on Attribute Grammars). In this second part we present the practical part of this work, i.e. the existing AG systems.

We intended to give as much practical information as possible, e.g. the size and speed of the generated evaluators, the implementation language, etc. Sometimes this kind of information was not available. We hope that the reader will not hold us reponsible for this situation.

The information we present here was gathered mostly from public communications (reports, conferences, etc..). We also sent a questionnaire to the authors of a number of systems ; most of them have answered, bringing interesting information.

We want this survey to be used as a catalogue of products by people who intend to use AGs (e.g. compiler writers), but we included as few specific comparisons between systems as possible. A general classification appears in the conclusion of this paper.

The information gathering for this paper spreaded over nearly two years ; it is thus possible that some information is not quite up-to-date.

2) What is in this paper

This paper is composed of the description of more than 20 systems which can be considered as attributes evaluators, plus a couple of others which are more loosely related to AGs. For the formers, each chapter is divided into the following parts :

- a header, indicating the name of the system, the head(s) of the project, and the address where one can get more information ;

- the list of the members of the project ; we apologize in advance to those we have forgotten ;

- the birthdate and deadline of the project ;

- the general features of the system ; since a very large majority of them are oriented towards compiler construction, the selected features are the lexical and syntactic analysis, the evaluation method, the class of AGs accepted by the system, the language used to describe AGs and the special features provided for code generation ;

- a more or less detailed schema of the internal organization of system, together with its implementation language and the machine(s) on which it runs ;

- the general comments : this part, which is generally the biggest, presents all the specific information about the system ; it has no really fixed organization ;

- the optimizations implemented in the system ;

- the applications and performances (when available) of the system ;

- the future projects;

- and the references.

The systems are classified in alphabetical order.

APARSE

Donn R. MILTON, Bruce R. ROWLAND
Bell Laboratories
Naperville, IL 60540
(U.S.A.)

1) Members of the project

D.R. MILTON , L.W. KIRCHHOFF, B.R. ROWLAND, T.P. BLUMER (implementation), C.N. FISCHER (adviser).

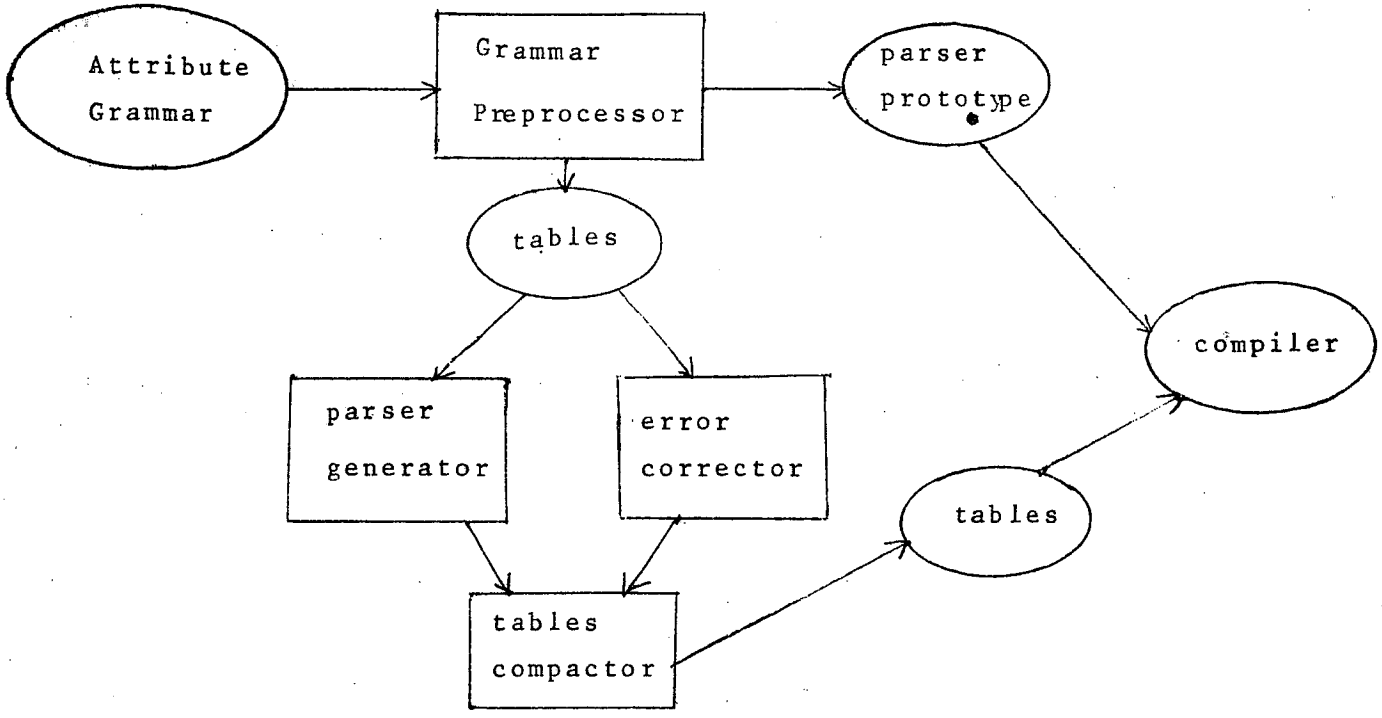
2) Birthdate : 1977. Deadline : ?

3) General Features

Lexical Analysis:	unspecified (LEX ?)
Syntactic Analysis:	ALL (1) (attributed LL) with error recovery.
Attributes Evaluation:	during parsing.
AG Class:	ALL (1)
AG Language:	based on C.
Code Generation:	no special feature provided.

4) Schema

APARSE is written in C, generates compilers written in C and runs on UNIX.



5) General Comments

APARSE uses the strong ALL (1) parsing and attributes evaluation algorithm devised in [Mil 77] and [Row 77]. Parsing is performed in a top-down predictive manner, and attributes are evaluated during parsing in a single left-to-right pass. The parse tree is not constructed, instead an attributes stack is maintained in parallel with the parsing stack.

Already evaluated attributes can influence parsing by means of disambiguating predicates. The system checks that such predicates are supplied for each LL(1) conflict, but cannot check (for obvious reasons) that the set of predicates for each conflict is complete (i.e, at least one of the predicates will always be true) and non-ambiguous (i.e,only one of the predicates will ever be verified).

The class of ALL(1) grammars is much larger than the LL(1) class. In addition, ALL(1) processing enables some of the syntactic specifications, (e.g. the priority of arithmetic operations) to be handled via attributes, which allows smaller AGs.

Moreover, ALL(1) grammars can be ambiguous (at the purely syntactic level).

The error-corrector used in APARSE is based on an insertion-only method; however attributes values do not influence this correction, except that the grammar specification supplies the values of the attributes of a terminal whenever it is inserted; this allows to "tune" somewhat the correction.

Productions and semantic rules are written in an input language very close to the one of YACC, from which many ideas are borrowed. The body of the semantic rules and the disambiguation predicates are written in C.Default rules are used for simple productions.

6) Optimizations

None required.

7) Applications

* An external symbol definition preprocessor for C.
* an infix notation desk calculator.
* a syntax analyzer and type checker for CHILL.
* other experimental applications.

We have no information about the size and performance of the system and the generated compilers.

8) Projects

Unknown

9) References

Mil 77, Row 77, MLR 79.

Athen's system
George K. PAPAKONSTANTINOU
Computer Center
Greek Atomic Energy Commission
Aghia Paraskevi Attikis
ATHENS
(GREECE)

1) Members of the project

George K. PAPAKONSTANTINOU

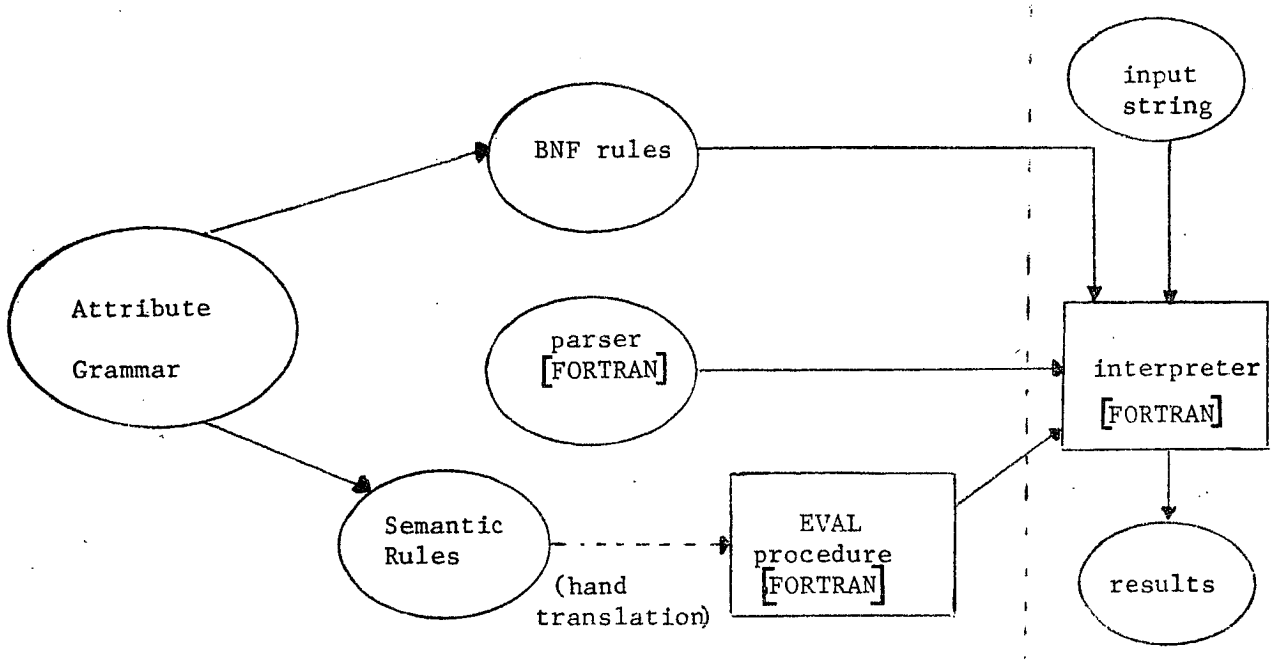
2) Birthdate : ? Deadline : ?

3) General Features

Lexical Analysis	: unspecified (hand-written ?)
Syntactic Analysis	: top-down non-deterministic parser
Attributes Evaluation	: in parallel with parsing
AG Class	: 1L-AGs
AG Language	: not specified
Code Generation	: no special feature provided

4) Schema

This system is written in FORTRAN and runs on any computer with a FORTRAN compiler; the author used an INTEL 8080-based microcomputer.



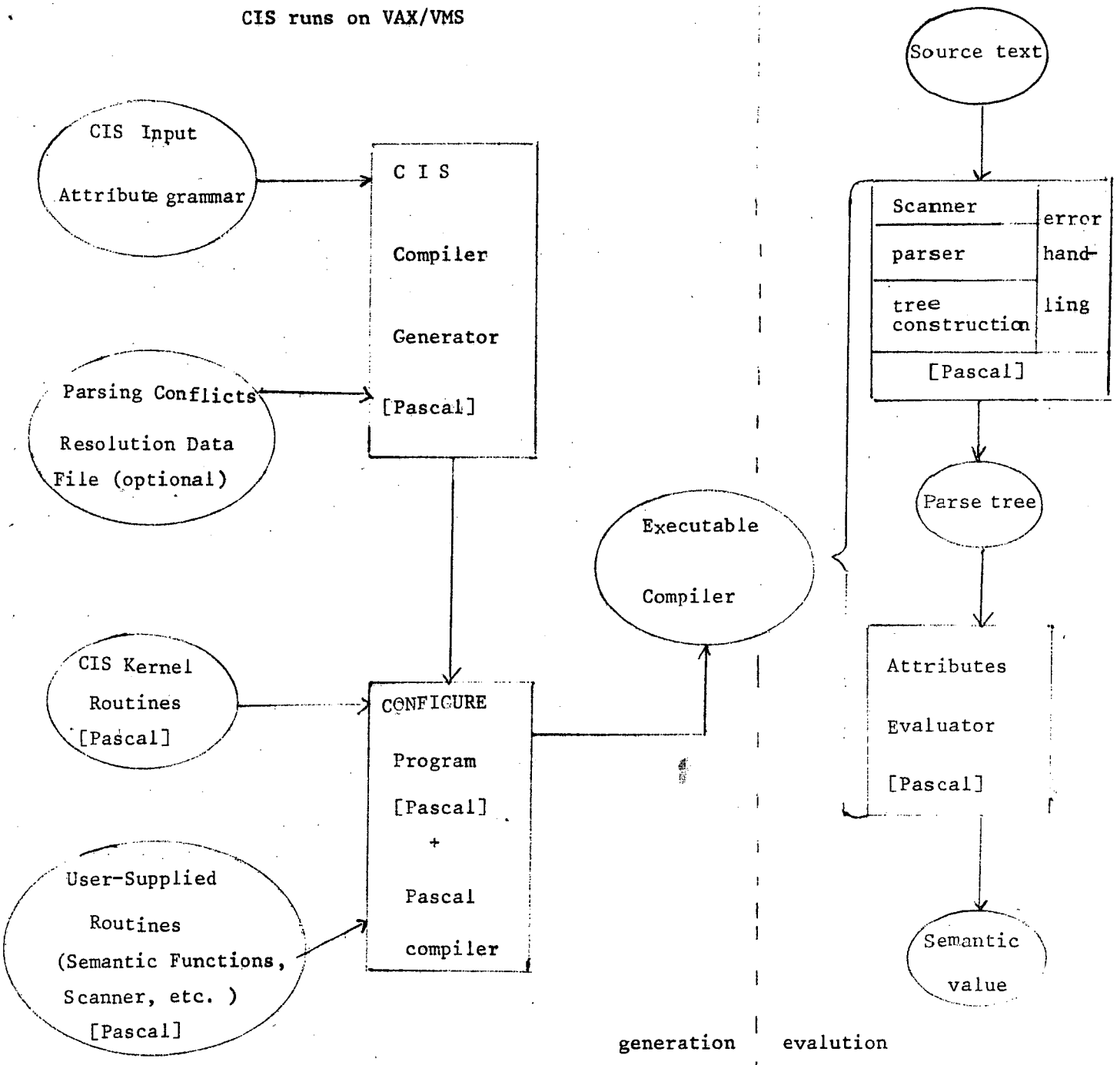
5) General Comments

The heart of this system is a top-down non-deterministic parser which interprets the BNF rules and which is able to backtrack to the next alternative of the current production. The usage of such an interpreter allows the grammar writer to express his specifications more conveniently; he may even write ambiguous grammars. However, as opposed to Earley's parser, this system will not construct all the possible derivation trees.

Attributes are evaluated in parallel with parsing. In consequence they must be evaluable in a single left to right pass. The grammar writer is responsible of writing the FORTRAN subroutine EVAL(G, I, J, M), where I denotes the rule, J the alternative of the rule, M the serial number of the current non-terminal in the rule, and G is a pointer to the location of the LHS non-terminal in the stack. You can derive from G the locations of the other non-terminals in the stack. These locations can be used to index the attributes stacks, in order to evaluate the attributes values. The construction of this EVAL subroutine from the AG can be performed in a mechanical way, e.g. by a preprocessor; however the system does not include such

4) Schema

CIS runs on VAX/VMS



5) General Comments

a) about lexical and syntactic analysis : the scanner must be written by hand in Pascal ; the CIS Compiler Generator supplies Pascal "include" files for the interface with the parser. The latter is an SLR(1) parser with an error recovery scheme based on the method by Pennello and de Remer ; new "error states" must be added, which increases the size of the parser, but this addition is automatic.

b) about attributes evaluation : the input AG is rather an "attribute scheme", with no "interpretation". The implementation of the attributes domain types and semantic routines must be written in Pascal and supplied separately. All Pascal consistency checks (type checking, etc) are performed by the Pascal compiler in module CONFIGURE. However the CIS Compiler Generator performs static type checking on the input AG. Non-normal AGs are accepted.

The goal of the attributes evaluator is to compute the "semantic value" of the source text, i.e. the list of the values of the (synthesized) attributes of the root of the parse tree (start symbol). This is done by (virtually) performing a depth-first lazy traversal of the compound dependency graph, interleaved with attributes evaluation : this eliminates automatically static dead ends (i.e. attributes instances not connected to the "root" of the dependency graph, and thus not contributing to the computation of the semantic value). The evaluation method is descendent.

Circularities are checked dynamically by marking each node in the graph as it is visited (a node corresponds to an attribute instance). When an attribute value is computed, it is stored in the tree and the subgraph issued from the corresponding node is (virtually) deleted to avoid reevaluation.

The compound dependency graph is not constructed. Rather the traversal uses the full parse tree and the dependency graphs of each production. The semantic rules are precompiled in an intermediate code.

As a special case, a (part of a) semantic rule can be a conditional of the form :

a preprocessor. The parser will call the EVAL subroutine at the right moments to perform attributes evaluation.

Already computed attributes values can influence the syntactic analysis by means of a variable called FLAG, common to the parser and the EVAL routine. After each call to EVAL, the parser will check the value of this variable, and if it is set to zero, the parser will backtrack to the next alternative of the current production. Of course FLAG is set to non-zero before each call to EVAL.

6) Optimizations

None

7) Applications and performances

The system has been used in problems of syntactic wave forms analysis.

The interpreter, not including the EVAL subroutine, consists of a few hundred FORTRAN statements and is thus simple and portable. On the 8080 microcomputer used by the author, it is stored in about 5 kbytes of ROM.

We have no information on the performances of the system, but using an interpreted as the parser cannot lead to tremendous performance.

8) Projects

Unknown.

9) References

On this system : Pap 81
On an earlier and very different system : Pap 79

CIS

Compiler Implementation System

Jean H. GALLIER
Dept of Computer and Information Sciences
Moore School of Electrical Engineering D2
University of Pennsylvania
PHILADELPHIA, PA 19104
(U.S.A)

1) Members of the project

Jean H. GALLIER (head, theoretical issues, attributes evaluation, incremental analysis), Fahimeh JALILI (now departed) (attributes evaluation, incremental analysis), John M. Mc ENERNEY Jr. (first implementation), John A. BIRCSAK (second implementation).

2) Birthdate : 06/1983. Deadline : none

3) General Features

Lexical analysis :	hand-written
Syntactic analysis :	SLR(1) with error recovery
Attributes evaluation:	recursive evaluation by need
AG class :	unrestricted (circularities are detected at evaluation time)
AG language :	resembles ALADIN (see GAG), AG "scheme" with "interpretation" supplied separately
Code generation :	no special feature provided

COND (predicate, truepart, falsepart)

In this case, at evaluation time, and according to the (dynamic) value of "predicate", only one of "truepart" or "falsepart" is evaluated, along with the attributes instances it references : thus dynamic dead ends are also eliminated.

6) Optimizations

Storage management uses a dynamic allocation technique.

7) Applications

* Toy languages
* A subset of Pascal
* The CIS Compiler Generator is not bootstrapped, but references [Gal83] and [Bir84] contain an AG in CIS input format describing the semantic rules and their translation into intermediate code.

8) Projects

* Incremental compiler : an incremental parser is already available, and the theory for an incremental attributes evaluator derived form the one of CIS is already developped [JG83].

9) References

Jal82, 83, JG83, Gal83, McE83, Bir84.

COPS and COPS 2
COmpiler Producing System
Jan BOROWIEC
Institute of Mathematical Machines MERA
Ul. Krzywickiego 34
02-078 WARSZAWA (Poland)

1) Members of the Project

Jan BOROWIEC (head)
R. KRZEMIEN, J. WITASZEK (lexical and syntactic analysis),
K. PIASECKI, D. KUPIECKI (Semantics)
T. PAPRZYCKI, M. KRISZWISKI, J. WINIEWSKI.

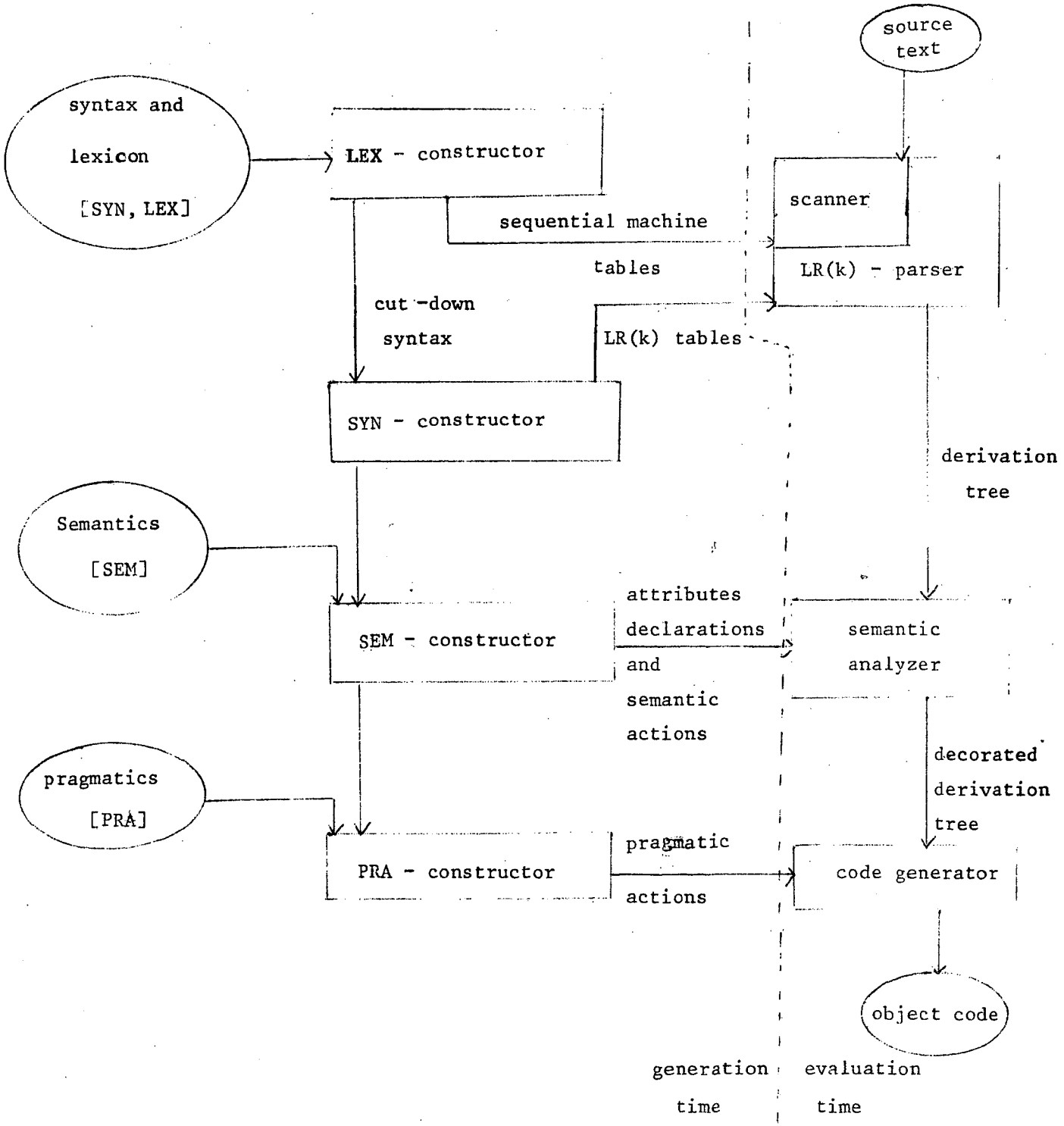
2) Birthyear : 1974; Deadline : unspecified.

3) General Methods

Lexical Analysis	: left linear grammars (BNF form)
Syntactic Analysis	: LR(k) (no error recovery)
Attributes Evaluation	: semantic rules (written in the metalanguage SEM) are structured in passes and processes. In each pass, processes are activated when necessary to compute the relevant attributes.
AG Class	: non-circular
AG Language	: LEX, SYN, SEM, PRA
Code Generation	: "pragmatics" is written in the metalanguage PRA and specifies the last traversal of the decorated tree. Similar to code templates.

4) Schema

COPS is written in PL/1 (4200 lines) and generates compilers written in PL/1. It runs on IBM 370.



5) General Comments

a) about the metalanguages

The SYN metalanguage is used to describe the syntax and lexicon of the source language, in a BNF-like form. The lexical part is thus a subset of the whole grammar, and must be written as a left-linear grammar. Special directives are used to define comment delimiters, separators, cross-references tables entries, generic terminals,etc. Each production alternative (either in the lexical part or the syntactic part) is named by an identifier, which will be referenced in the semantic and pragmatic part.

The semantic rules are written in the metalanguage SEM. The data types of SEM are roughly those of PL/1. Three kinds of attributes are defined in the system :

- local attributes are classical attributes, i.e. attached to a node throughout the whole evaluation process;
- global attributes, which are not attached to any node
- temporary attributes, which are attached to a node only during a given evaluation pass.

The direction (inherited or synthesized) of attributes is not demanded; it is not used by the SEM-constructor. Three attributes are defined by the system and attached to generic terminals.

SEM provides three simple statements (assignment, message, and empty) and three structured statements (conditional, loop, and selection). (Error) messages are defined separately.

Semantic actions are structured, from coarser to finer, into passes, semantic rules, and processes. Passes will be performed one after the other to decorate the tree; temporary attributes are attached to passes, and vanish after the corresponding pass ends. A semantic rule is a set of processes attached to a given production. A process is an atomic evaluation action, but it can define several attributes at a time; temporary attributes retain their values only in a particular execution of a process.

Code generation ("pragmatics") is described in the metalanguage PRA. A pragmatic rule is attached to a production, and is a list of units which may be of three kinds :

- output some text or value,
- evaluate some (global) attribute,
- and visit a son.

These units can be embedded in compound statements as in SEM.

These metalanguages are completed by a macrolanguage and its macrogenerator.

b) about the evaluation strategy

The evaluation method used in COPS is similar to the one devised by Fang [Fan72] and used in FOLDS (q.v.) except that the attribute grammar is split into passes. No circularity test is performed.

c) COPS 2

We have only fragmentary information about COPS2 [KKW82]. The basic improvement is an enhanced modularization of the generated compiler. After the construction of original (abstract) syntax tree, performed by the scanner and parser, and during which some attributes can be computed in a bottom-up manner, several successive phases can occur. Each phase is composed of a semantic analyzer (an attribute evaluator using Kastens' OAG method), of optionnally an optimizer, and of a transformer. The output of the transformer is a partially decorated abstract tree to be input to the next phase, or, for the last phase, linear object code.

6) Optimizations

None, as far as we know. The authors find that PL/1 is a large memory consumer.

7) Applications

COPS : 10 to 15 small languges, ASPLE, a subset of COBOL, MODULA enhanced with real numbers.

COPS2 : a compiler of a subset of COBOL to MERA 400 assembly language (6500 lines of description), a compiler of MODULA (+ reals) to INTEL 8080/8085 assembly language (8300 lines).

8) Projects

Unknown.

9) References

Bor76, Bor77, Bor78, KKW82.

CWS1 and CWS2
Compiler Writing Systems

Gregor V. BOCHMANN
Département d'Informatique
Université de Montréal
Case Postale 6128
MONTREAL 101
(CANADA)

1) Members of the project

G.V. BOCHMANN, O. LECARME, P. WARD, M. LOUIS-SEIZE, D. GURTNER, and others.

2) Birthdate : 1972 Deadline : 1977

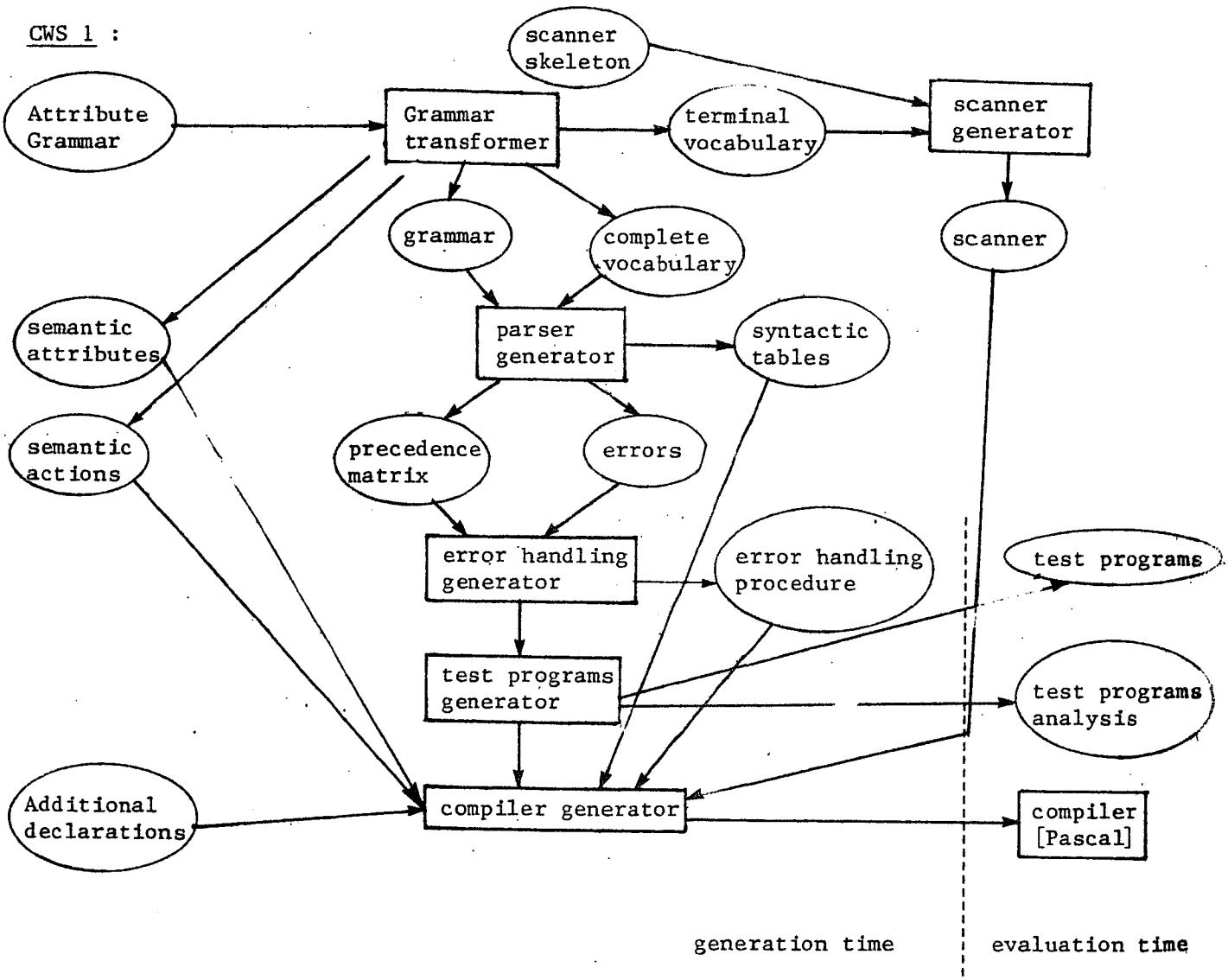
3) General Features

Lexical Analysis :	fixed, sightly parametrizable scanner
Syntactic Analysis :	CWS1 : weak precedence (bottom-up)
	CWS2 : LL(1) with regular expressions (top-down)
Attributes Evaluation :	during parsing
AG Class	CWS1 : purely synthesized
	CWS2 : L - AGs
AG Language	based on Pascal
Code Generation	no special feature provided.

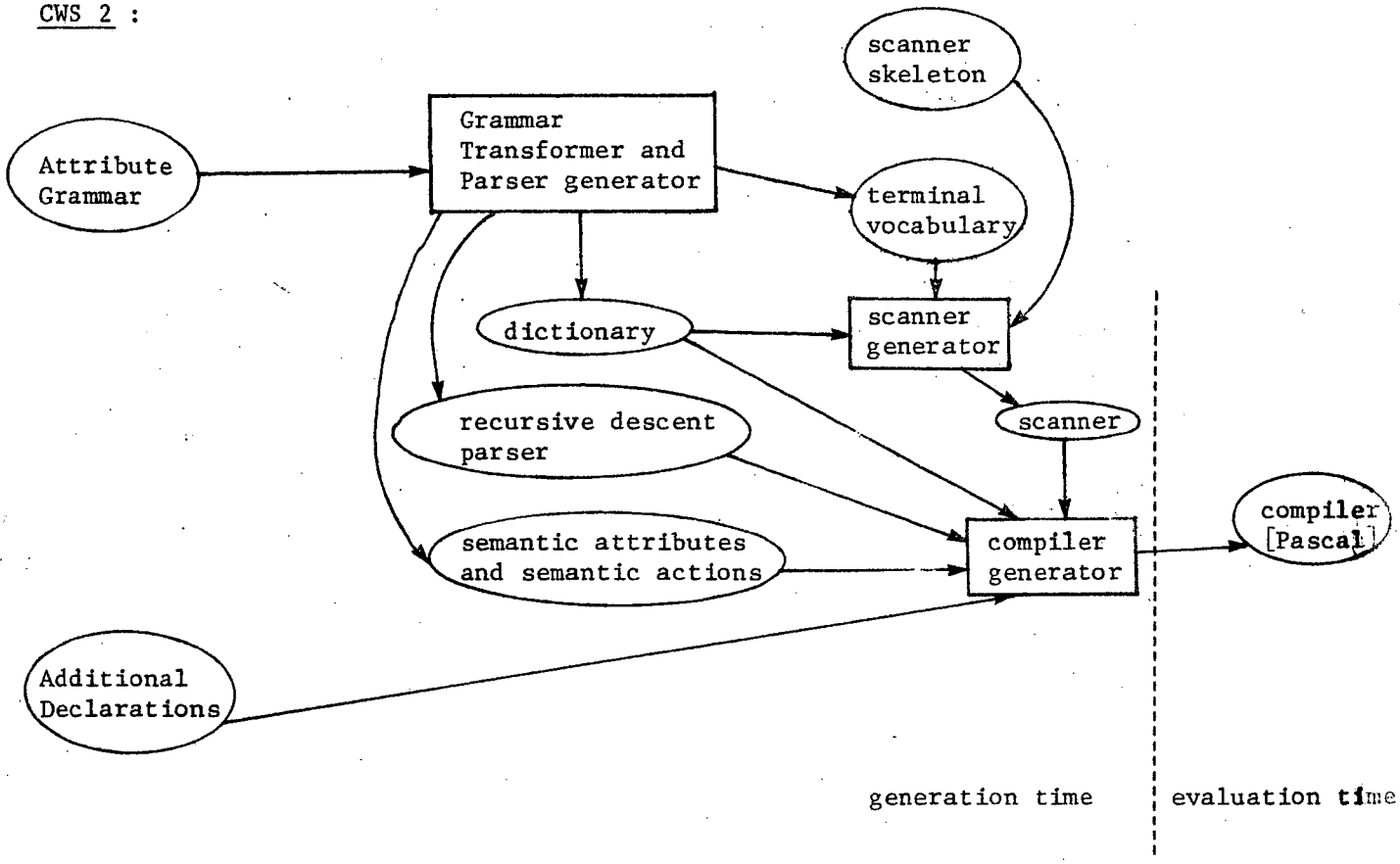
4) Schemas

Both systems are written in Pascal, produce compilers written in Pascal and run on CDC Cyber 74 and Xerox Sygma family.

CWS 1 :



CWS 2 :



5) General Comments

a) about the metalanguage

The two systems accept, in addition to "normal" semantic rules involving attributes occurrences, the specification of semantic actions to be executed upon reduction of each production. Semantic rules and semantic actions are embedded in the RHS of the productions themselves, as for YACC. It is possible to define local attributes, i.e. variables local to a production. The semantic rules language is based on Pascal, and semantic actions are written in Pascal. The AG, called here the "integrated description", is completed by additional declarations of types, variables, procedures, ..., written in Pascal.

b) about CWS 1

The main module of the generated compilers is the parser. Semantic routines are called upon each reduction to perform attributes evaluation and semantic actions. Attributes values are maintained on a stack. The test program generator constructs a

set of syntactically correct programs involving every production.

c) about CWS2

The generated compiler is a recursive descent one : a procedure is associated to each non-terminal, with value parameters for inherited attributes and variable parameters for synthesized ones. Attributes evaluation and semantic actions are inserted in the body of those procedures at appropriate places. Local attributes are implemented as local variables of these procedures.

The syntactic part in written in extended BNF, allowing regular expressions in the RHS. This permits to overcome the impossibility to write left-recursive productions in LL(1) grammars. The addition of attributes to extended BNF grammars is discussed in [Boc 75 a] . CWS 2 also provides for attributes-influenced parsing.

d) other remarks

Portability and usability (user-friendliness, efficiency, etc..) of the systems were the main design goals.

To generate the scanner, the grammar writer picks the relevant tokens among a fixed list (identifiers, numbers, strings, etc.) and sets a number of options.

6) Optimizations

None required.

7) Applications

Teaching.

8) Projects

None : the team has disintegrated.

9) References

CWS 1 : LB 74a, LB 74b, Lec 75, Lec 77, BL 73.

CWS 2 : BW 75, BW 78

Delft's System
J. VAN KATWIJK
Delft University of Technology
Dept. of Mathematics and Informatics
132 Julianalaan
DELFT
(The Netherlands)

1) Members of the project

J. VAN KATWIJK and perhaps others.

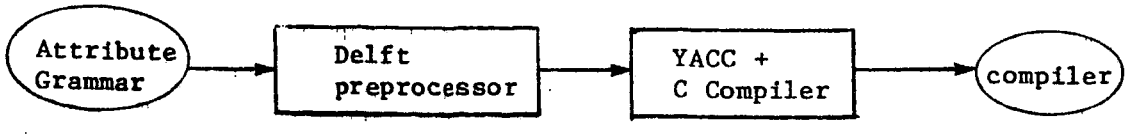
2) Birthdate : ? Deadline : ?

3 General Features

Lexical Analysis :	unknown (LEX?)
Syntactic Analysis :	LALR(1) with disambiguating rules and precedence rules (YACC).
Attributes Evaluation :	during parsing
AG Class :	1 L-AGS
AG Language :	based on C and YACC input language
Code Generation :	no special feature provided.

4) Schema

Delft's system is written in C and runs on UNIX



5) General comments

The Delft system has limited ambitions : its goal is to augment the power of YACC with a capability to deal with simple forms of attributes.

In fact, attributes are considered as parameters of non-terminals -inherited attributes being input parameters, and synthesized ones output parameters -, and no output parameter can depend (in the outside context) on an input parameter. That is, the AG must be evaluable in one left-to-right pass (1L-AG).

The basic ideas and the notations used in the system are much closer to affix grammars [Kos 71] or extended AGs [WH77, WH79, WM83] than to "normal" AGs. The evaluation strategy is derived from [Wat74].

The parse tree is not constructed ; rather an attribute stack is maintained in parallel with the parsing stack. Before recognizing a non-terminal, all its inherited attributes must be available on top of stack ; after the recognition of that non-terminal, those inherited attributes must be topped by its synthesized attributes. The order of attributes in a reference to a non-terminal is important since it is also the order in which those attributes (identifiers) are placed on the stack. To achieve this, it is possible that one has to insert an attribute mover, i.e. a new non-terminal deriving the empty string the only purpose of which is to provide an action hook to move attributes values around on the top of the stack. Those attribute movers are generated automatically by the system, but it is possible that the new grammar be not LALR(1) any more.

The input language is a mixture of affix grammars, YACC metalanguage and C. The semantics rules are written in C.

The system is to be used as a preprocessor for YACC and is about 3000 lines of C long. It is available from the University of Delft.

6) Optimizations

None required.

7) Applications

A compiler (front-end?) for ALGOL60 and an other for a subset of Ada.

We have no information about the performance of the system and the generated compilers.

8) Projects

Unknown.

9) References.

Vak 83.

DELTA

DEfinition des Langages et de
leurs Traducteurs par Attributs

Bernard LORHO
Institut National de Recherche
en Informatique et en Automatique
BP 105
78153 LE CHESNAY Cédex
(FRANCE)

1) Members of the project

Bernard LORHO, F. BLAIZOT, L. BLAIZOT, P. BOUCHENEZ (attributes processing), Pierre BOULLIER (lexical and syntactic analysis).

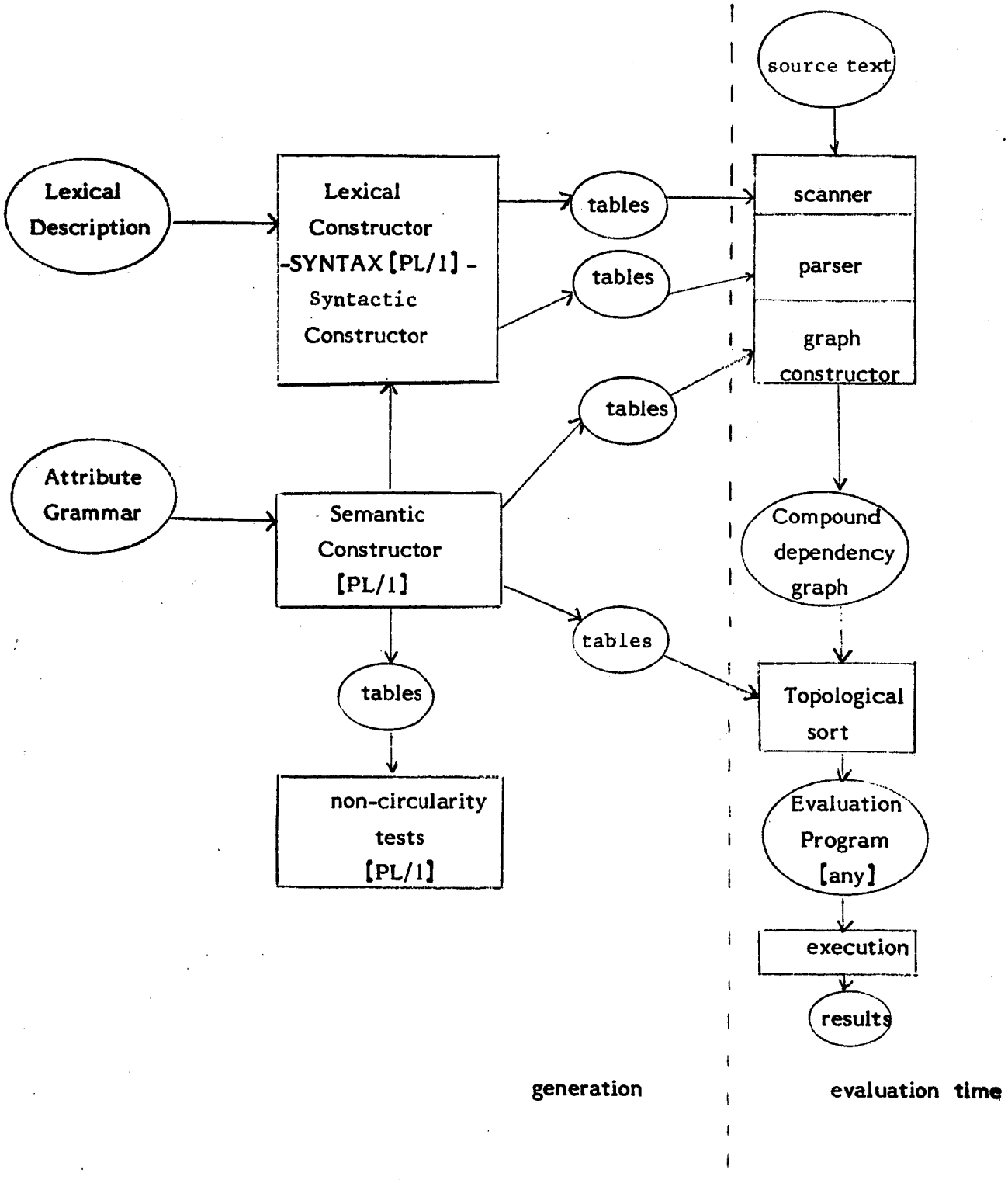
2) Birthdate : 1972 Deadline : 1976

3) General Features

Lexical Analysis	: regular expressions with user-defined predicates and actions, and error recovery.
Syntactic Analysis	: SLR(k), LALR(1), LR(1) with error recovery
Attributes Evaluation	: construction and descendent evaluation of the compound dependency graph
AG Class	non-circular
AG Language	any
Code Generation	no special feature provided

4) Schema

DELTA is written in PL/1 and runs on Multics.



5) General Comments

The system used for the lexical and syntactic parts is SYNTAX*. See the chapter devoted to the FNC/ERN system for a description of SYNTAX.

Semantic rules can be written in any language, and DELTA will produce, for each source text, an evaluation program written in that language. Each semantic rule is a statement in that language specifying what to do with the different attributes instances (considered as variables) : they may be assigned, passed as parameters to subprograms, etc. The description of an AG is composed of :

- a (maybe empty) header; this header is divided in three parts which will be inserted in the produced program, respectively at the beginning, between the declarations and the start of the program, and at the end ;

- The list of attributes declaration ; each attribute is declared with its kind (see further), the list of non-terminals to which it is attached, and maybe its type (in the sense of the semantic rules language) ;

- the list of syntactic productions and associated semantic rules ;

- a (maybe empty) list of "key terminals" to be used in error recovery.

DELTA predefines a number of lexical attributes, the values of which is set at parsing time.

DELTA provides facilities to ease the writing of an AG : a large number (more than 60% in practical AGs) of semantic rules (copy rules) need not be written because the system uses a set of default rules to generate them. DELTA recognizes four attribute "kinds" :

- synthesized ;
- inherited ;

(*) trademark of INRIA

- global : like an inherited but intended to be carried unchanged over a whole subtree ;

- compound : a pair of a synthesized and an inherited attributes with the same name, attached to the same non-terminals and intented to circulate from left to right on list-like productions.

DELTA also accepts synchronizing attributes, which merely add new dependencies to a semantic rule, and empty semantic rules for attributes occurrences which will never be evaluated but which must be defined (for the AG to be "well formed").

The attributes evaluation method is as follows. In parallel with parsing, the compound dependency graph is constructed : the tables produced by the semantic constructor contain information about the dependencies between attributes occurrences. This graph is complete except that copy rules are collapsed : the two (or more) attributes instances are represented by the same node. The graph is then traversed in a depth-first manner (topological sort) to determine the attributes instances to be evaluated and their evaluation order. Variable names, with the corresponding type (in the semantic rules language sense), are generated as placeholders for the attributes values. Dead ends, i.e. attributes instances not connected to the root of this graph (defined as the attributes of the start symbol) are not reached by this traversal and are thus eliminated from evaluation. This process produces a program (in the semantic rules language) which can be interpreted or compiled and run to give the desired results (this must be programmed in the semantic rules). When an attribute instance becomes useless (at a certain point in the evaluation program), the corresponding variable may be reused for another attribute instance, if necessary. Copy rules are not generated, because the corresponding attributes instances are represented by the same variable.

Non-circularity tests are provided too, but they are optional. However running one of the tests is important because, if there is a circularity in the dependency graph, the topological sort loops. The tests take as input the attributes tables. The main theoretical effort in DELTA has been to improve those tests so that they remain feasible, even for large AGs [LP75, DJL84]. The main improvement is the notion of "covering" [LP75]. There is also a program to check strong (absolute) non-circularity in polynomial time.

There is also a simplified version of DELTA, which accepts only purely synthesized AGs. The semantic rules can be written in Pascal (TABACT) or in PL/1 (TABPL1). Attributes values are computed bottom-up during LR parsing, and stored on a stack parallel to the parsing stack. The Pascal programs generated by TABACT, together with the Pascal version of SYNTAX analyzers, are portable ; however the constructors must be run on Multics. A version of TABACT also runs on SM90/SMX.

6) Optimizations and performance

Strictly speaking of attributes evaluation, all the optimizations have been described in the previous section.

DELTA is a pionneering work, and as such will stay in an experimental state. However, it is an important timestamp, on both theoretical and practical levels.

The full generality of DELTA - non-circular AGs, any programming language for the semantic rules - is the cause of its inefficiency.

DELTA is written in PL/1 and has the following size :

SYNTAX constructors	12 000 lines (8000*)
Semantic constructor	3 700
Non-circularity tests	
Full non-circularity	1 700
Strong non-cricularity	1 100
SYNTAX analyzers	5 000
Dependency constructor	800
Attributes Ordering	900
TOTAL	25 600 (21600)

TABACT and TABPL1 are much smaller (3000 lines each) ; the produced evaluators are composed of the SYNTAX analyzers and the generated programs.

(*) 8000 lines if restricted to the LALR(1) constructor.

7) Applications

DELTA was the first AG system in France, and was available rather early (1974) : it has thus been heavily used for a large number of research purposes, and for teaching purposes. It was also used in the first release of Perluette (q.v.)

TABACT and TABPL1 are used for more "industrial" purposes. TABPL1 is used in the European Ada Compiler to build the abstract syntax tree. TABACT is used in the RHIN project (Agence de l'Informatique) to develop a compiler for the PDIL language, devoted to communication protocols description.

8) Projects

Transport of TABACT to UNIX

9) References

Lor74, Lor77, LP75, DJL83, 84.

ELMA

M.B. MERISTE
Tartu University
TARTU, Estonia
(U.S.S.R.)

1) Members of the project

Merik B. MERISTE.

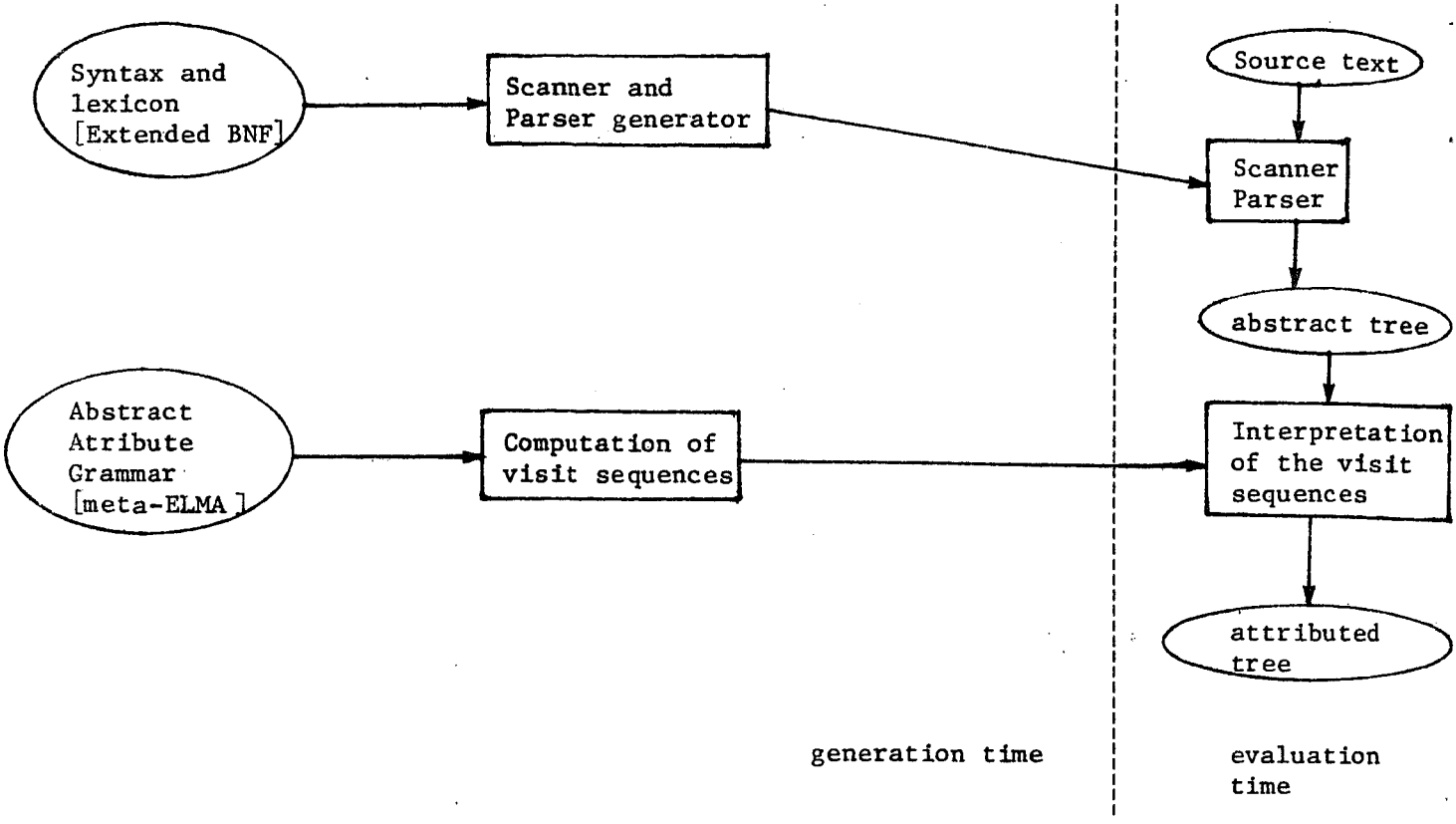
2) Birthdate : ? Deadline : ?

3) General Features

Lexical Analysis :	?
Syntactic Analysis :	precedence
Attributes Evaluation :	visit-sequences
AG Class :	n-visit AGs
AG Language :	meta-ELMA
Code Generation :	no special feature provided.

4) Schema

ELMA is written in FORTRAN and runs on RYAD machines. A version runs on APPLE II.



5) General Comments

With ELMA, attributes are defined on abstract trees, i.e. on an abstract syntax described in BNF extended with regular expressions. The usual notation for attributes occurrences is extended to take into account any sequence of non-terminal symbols. Meta-ELMA is an extension of FORTRAN. The class of acceptable AGs lies between the multi-pass AGs and the OAGs.

The evaluation strategy seems to be the simple multi-visit one : the attributes are partitionned into passes, and for each pass the tree-walk is directed by an order which depends on local heuristics.

6) Optimizations

Unknown.

7) Applications

About fifty languages.
We have no information about the performances of ELMA

8) Projects

Optimizations such as the ones of HLP (q.v.).

9) References

Mér 80, Mér 81, VM 82.

FNC/ERN

Fortement Non-Circulaire

Evaluation Récursive par Nécessité

Martin JOURDAN

Institut National de Recherche

en Informatique et en Automatique

BP 105

78153 LE CHESNAY Cedex

(FRANCE)

1) Members of the Project

Martin Jourdan (attributes processing), Pierre Boullier (lexical and syntactic analysis), Bernard Lorho, Bruno Courcelle, Pierre Deransart (advisers).

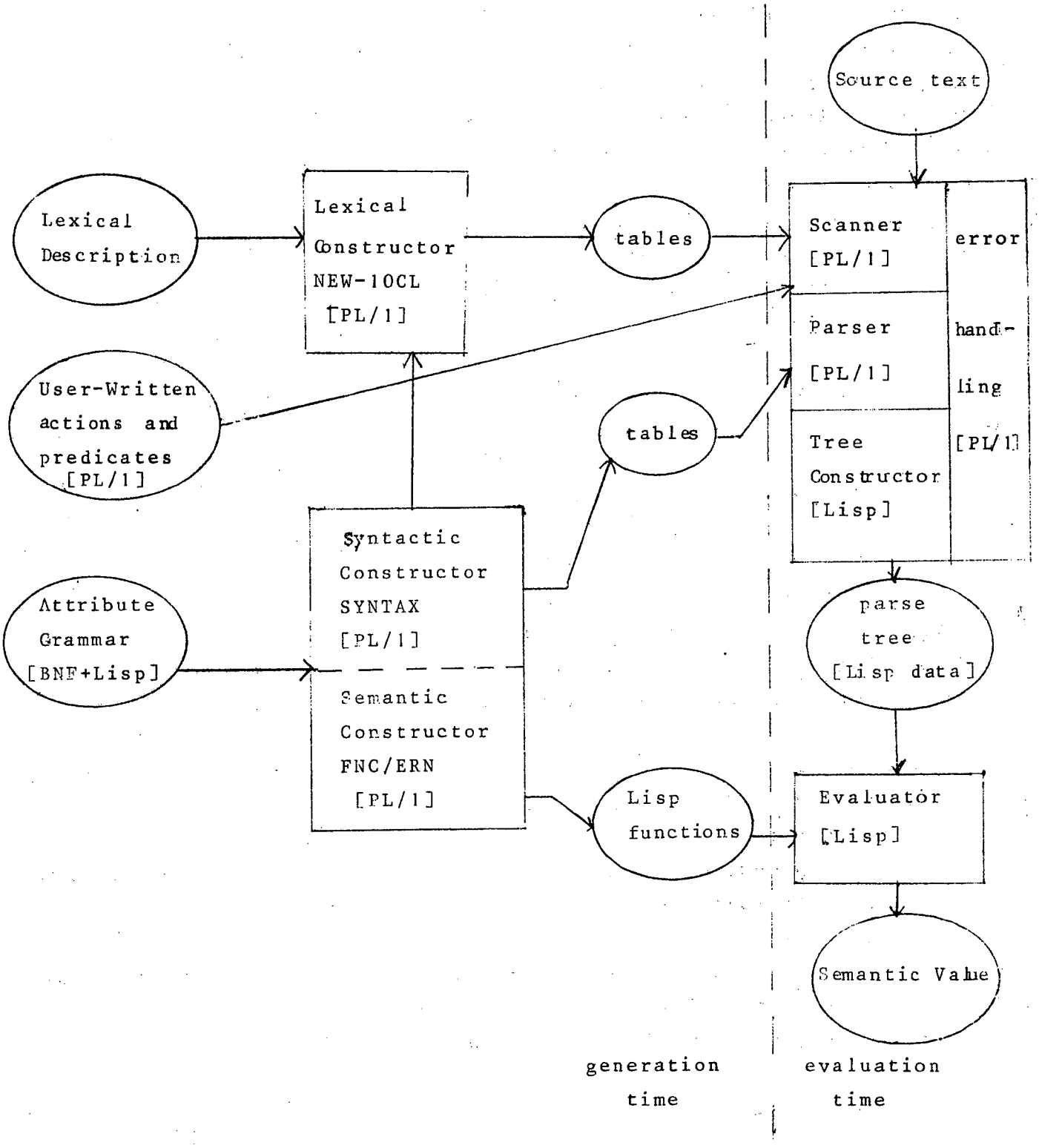
2) Birthdate: 02/1982. Deadline: none.

3) General Features

Lexical Analysis:	regular expressions with user-defined predicates and actions, and error recovery.
Syntactic Analysis:	SLR(k), LALR(1), LR(1) with error recovery.
Attributes Evaluation:	FNC: primitive recursive schemes [CF 82]. ERN: recursive evaluation by need [Jou 84 a,c,e] Both methods are descendent.
AG Class:	FNC: strongly (absolutely) non-circular. ERN: unrestricted (circularities are detected at evaluation time).
AG Language:	based on Lisp.
Code Generation:	no special feature provided.

4) Schema

FNC/ERN runs on Multics:



5) General Comments

a) about lexical and syntactic analysis: The underlying system is SYNTAX*. The scanner and parser are table-driven. The syntactic analysis uses LR methods. SYNTAX provides in particular intensive optimizations of the tables size, a very efficient and powerful error recovery mechanism, and facilities for user-defined error messages. The scanner tables are constructed from regular expressions using LR-like methods, producing directly the deterministic automaton; this method allows to use an unbounded number of characters in look-ahead to solve the conflicts. User-defined predicates and actions allow to solve conflicts and to add "context-sensitive" information to the scanner. The scanner has the same error recovery mechanism as the parser.

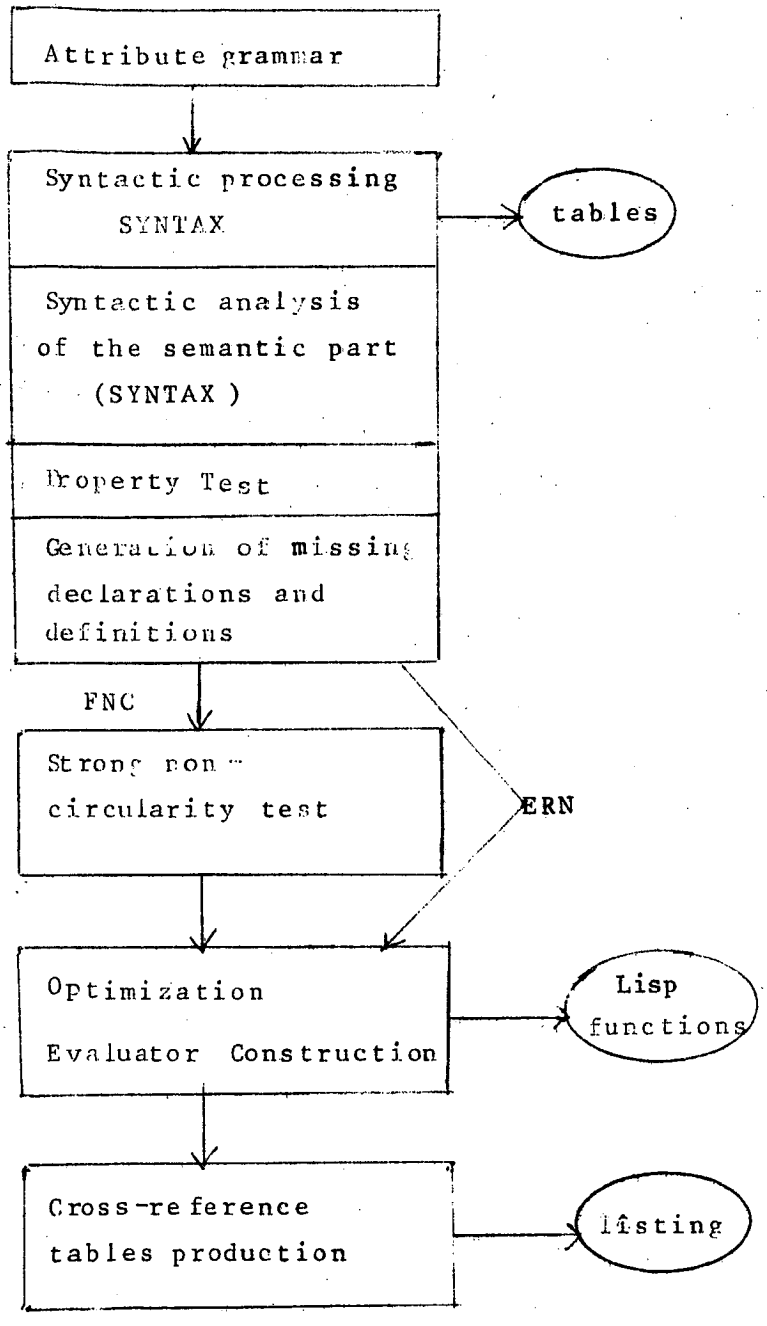
b) about attributes evaluation:

FNC and ERN accept the same attribute grammar as input, and share many modules (see schema). The module called "Property test" builds an internal form of the semantic rules and checks that the AG is "well-formed"(e.g. no inherited attribute of a LHS is defined). The next module generates, through a set of "default rules", the missing attribute declarations and semantic rules. More than 60% of the semantic rules in practical AGs can thus be omitted; this eases very much the design and writing of an AG. The default rules for generation of missing semantic rules are borrowed from the system DELTA (q.v.), together with its four attribute types. The generation of missing attribute declarations is entirely original.

FNC/ERN produces evaluators written in Lisp, and the attribute definitions language is based on Lisp. The description of an AG is composed of:

- a (maybe empty) header, which is a list of Lisp forms which will be copied directly with the produced evaluator; these forms are usually auxiliary function definitions;

* Trade mark of INRIA



- a (maybe empty) list of attribute declarations; each attribute is declared with its type (in the DELTA sense, not in the Lisp sense) and the (maybe partial) list of the non-terminals to which it is attached;

- a (maybe empty) list of default values for synthesized attributes, to be applied at "error nodes"; these error nodes appear in the parse tree when an unrecoverable syntactic error has occurred; it thus possible to evaluate attributes on syntactically correct portions of text, and get the behaviour of a real compiler;

- the (never empty!) list of syntactic productions and associated semantic rules;*

- a (maybe empty) list of "key terminals" to be used in error recovery.

The system predefines a number of lexical attributes, the value of which is computed at parsing time. Also a number of fuctions are predefined, e.g. to produce error messages.

* As with DELTA, it is possible to have "synchronizing attributes" and empty definitions; the latter produce run-time errors if they are used (implicity or explicity);

FNC uses the evaluation method introduced in [CF82] and later improved [Jou 83, Jou84 a,b,d,e]. After the common processing, FNC checks the strong non-circularity of the AG (with an improved algorithm [DJL83, Jou84e, DJL84]), computes the "argument selector" and builds (after the optimization phase) the set of Lisp evaluation functions, one for each synthesized attribute.

ERN uses the evaluation method presented in [Jou 84 a,c,e]. After the common processing and the optimization phase, ERN builds the set of Lisp evaluation functions, one for each attribute (either synthesized or inherited).

Both methods are applicative, recursive and implement a dynamic evaluation by need. The control flow of the semantic rules is mixed with, and influences, the evaluation process. The result of the evaluation is in both cases the "semantic value" of the source text, defined as the list of the values of the synthesized attributes of the root of the parse tree.

6) Optimizations and performance

a) Space management

Lisp, as the basis of the run-time evaluator, allows an efficient automatic memory management, since every datum is accessed and manipulated through pointers. Morcover large data structures such as symbol tables may easily be shared among many attributes instances. However the management of this sharing is left to the grammar writer .

As for parse tree, a concrete parse tree is built, with all the terminals (to keep the system simple). However, simple productions do not appear in the tree, and this saves much space.

As for attribute values, two optimizations are performed automatically:

- attributes such that all their instances are used only once (this is computed statically at generation time [Jou 84e]) are not stored in the tree; they exist only as function return values;

- useless subtrees (i.e. subtrees such that all the synthesized attributes of their root have been computed) are dynamically deleted.

The size of the function calls stack is bounded by:

- for FNC, the height of the tree;
- for ERN, the height of the compound dependency graph, which is much larger.

b) use and performance of the system.

The use of Lisp allows to take full advantage of the dynamic evaluation by need: you can use and write functions which do not compute all their arguments, which is impossible with other languages.

The height of the function calls stack is larger for ERN than for FNC, and the function calls mechanism is much more called upon; so, although theoretically optimal, ERN is practically less efficient than FNC, and restricted to the analysis of small texts. However, since it avoids the expensive non-circularity test, ERN is well suited to the development of a new AG.

The FNC/ERN constructor produces a number of cross-reference tables which give much information about the AG. The FNC/ERN runtime system provides many options to control the evaluation, the processing of the results and the debugging of the AG.

As for the size of the system, the constructor part of SYNTAX is 12000 PL/1 lines (8000 if restricted to only the LALR (1) constructor), and the FNC/ERN constructor is about 4500 PL/1 lines and 200 "SYNTAX" lines. The runtime system is about 500 Lisp lines, plus 200 PL/1 lines for the interface, plus 5000 PL/1 lines for the analyzers part of SYNTAX.

7) Applications

* Toy languages.

* FNC is included in the Perluette system (q.v.), both as the first phase of the generated compilers and as the constructor for a number of inputs to Perluette.

* A full ISO Pascal to P-code compiler which compares honourably with a hand-written one: on Multics, it runs only 2.7 times slower, and when processing a 6000+ lines Pascal program, it builds a tree 1300000 words large and the garbage collector messages show a maximal use of 2 350 000 words, which seems acceptable.

* A Pascal to Ada translator [BDJ 83, 84]

* A compiler for the LOM language (data processing), developed in Toulouse.

8) Projects

* Transport to UNIX.

* Implementation of the OAG/ visit-sequences evaluation method [Kas 80], to run useful comparisons with the FNC method.

* An interactive tool to trace circularities statically.

* A new SYNTAX constructor for RLR (Regular LR) grammars [Bou 84 a].

* A full compiler writing system à la MUG2 (q.v.), with abstract syntax trees, attributed tree transformations, and special features for code generation.

* An Ada version, written in Ada and producing Ada evaluators.

* A programming environment generator (à la Synthesizer Generator (q.v.)), providing incremental analysis and compilation.

9) References

CF80 , CF82, Jou82, Jou83, Jou84 a,b,c,d,e, DJL83, DJL84, BDJ83, BDJ84.

about SYNTAX:

[Bou 84a] P.Boullier: "Contribution à la Construction Automatique d'Analyseurs Lexicaux et Syntaxiques" (in French) thèse d'Etat, Université d'Orléans, Orleans (1984).

[Bou 84b] P.Boullier: "Lexical and Syntactic Analysis", in [Lor 84], pp 7-44.

FOLDS
FOrmal Langage Definition System
Isu FANG
Computer Science Dept
Stanford University
STANFORD , Ca
(U.S.A)

1) Members of project

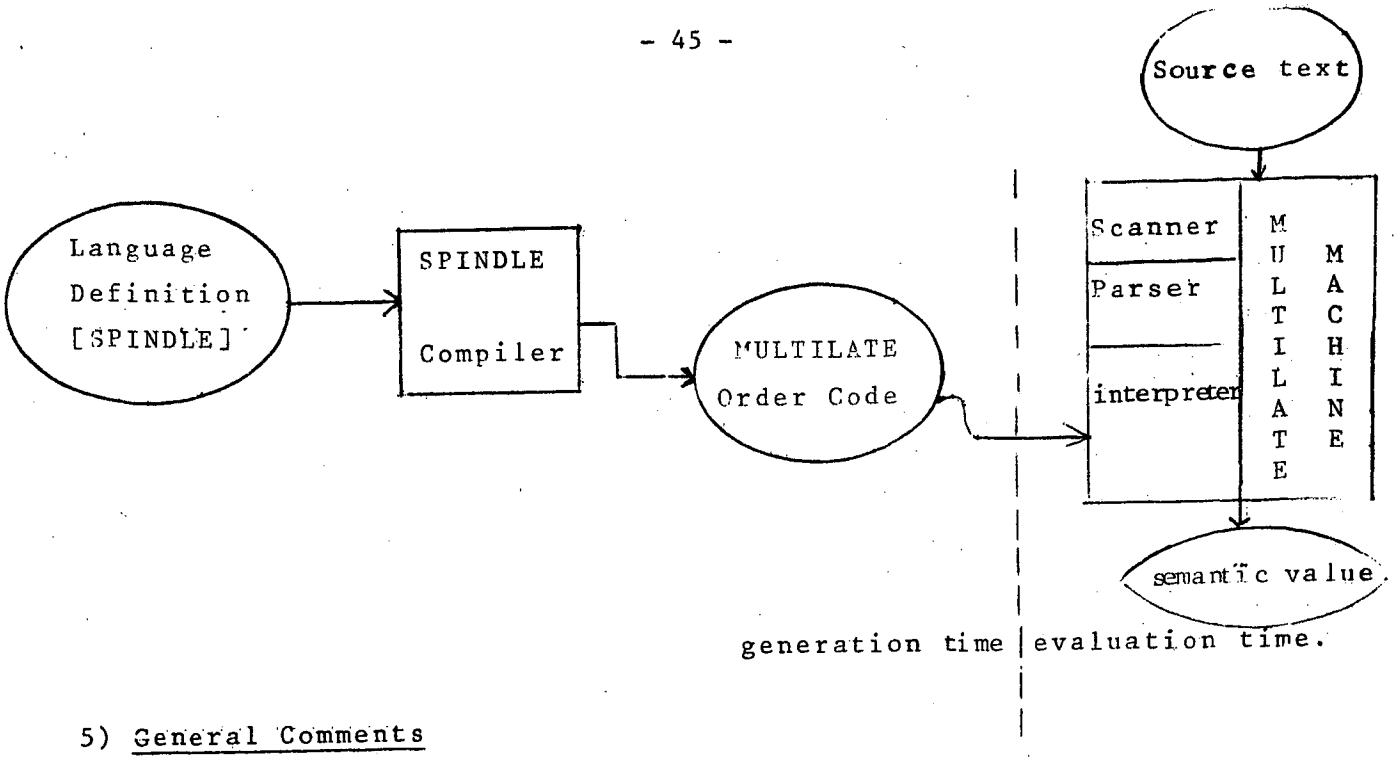
Isu Fang. The project was realized in part at Faculdade de Economica e Administraca da Universidade de Sao Paulo (Brazil).

2) Birth date :1971 Deadline : 1972.

3) General Features

Lexical Analysis :	fixed scanner recognizing tokens of Algol - like languages
Syntactic Analysis :	Earley non -deterministic parser.
Attributes Evaluation :	concurrent processes
AG Class :	non-circular
AG Language :	SPINDLE
Code Generation :	no special feature provided.

4) Schema



5) General Comments

FOLDS is the first historical implementation of Knuth's attribute grammars. It is essentially an experimental system to be used to design new languages.

There is no way to specify the lexical analysis. FOLDS provides a fixed scanner which recognizes identifiers, numbers, strings, etc .

FOLDS accepts ambiguous grammars. In that case the Earley non-deterministic parser used in FOLDS produces the set of all the derivation trees of the source text, in time $\mathcal{O}(n^3)$. If the grammar is not ambiguous, or if a particular source text has only one derivation tree, the time complexity is $\mathcal{O}(n)$. Syntactic ambiguities can be resolved by disambiguating predicates involving attributes values. In that case all the possible derivations trees are constructed, and only those which verify the predicates (hopefully only one) are kept; the others are discarded. However the test of the predicates can be performed only when the attributes are available, i.e. not during parsing.

Semantic rules are written in SPINDLE (Semantic Preparatory INput LanguagE). In fact SPINDLE provides for defining processes which can be executed in parallel. Normally each semantic rule is a separate process, but a process may also compute several attributes values at a time. Apart from that, SPINDLE is an Algol-like language with data structures a la VDL (Vienna Definition Language).

Copy rules need generally not be written, which can save up to 50% of the total number of semantic rules to be defined. It is also possible to define local attributes, i.e. variables local to a process.

The language definition in SPINDLE is translated into code for the parallel machine MULTILATE (Machine UnderLying The Interpretive LAnguage To be Executed). This machine generates the processes corresponding to the attributes instances of a derivation tree and executes them in parallel according to the following scheme:

- a semantic rule activates the corresponding process;
- the processes have one of three possible states:
  - inactive if never yet activated;
  - passive if awaiting a datum (attribute value) not yet defined (the corresponding attribute(s) being it- (them-) self(-ves) undefined);
  - else, active;
- the whole scheme is initiated by activating the processes corresponding to the (synthesized) attributes of the root of the derivation tree(s).

With such a strategy, only the attributes necessary to compute the attributes of the root (the "semantic value" of the source text) are evaluated. Furthermore the system can accept any non-circular AG and even evaluate trees of circular AGs involving no circularity, like the FNC/ERN system (q.v.) or the LINGUA system (q.v.)

However, the performance of such a system implemented on a monoprocessor machine cannot be good (and is not, actually), because the number of passive processes can be very large and the search of processes to activate can be long, dispite the use of chain links and stacks. Furthermore you must add the load of the interpretation of MULTILATE code on a conventional machine.

6) Optimizations

Space for storing attribute values is allocated dynamically.

However different instances having the same value (through a copy rule) imply distinct storage, except for complex values. A garbage collector recollects the storage freed by processes which terminate and by trees from ambiguous derivations which are discarded.

7) Applications

* TURINGOL
* a subset of SIMULA-67 (rewritten from [Wil 71])

No information is available about the size and the performances of the system

8) Projects

None (terminated).

9) References

Fan 72, Fan 73 .

GAG

Generator based on Attribute Grammars

Uwe KASTENS
Institut für Informatik II
Universität Karlsruhe
Postfach 6380
D-7500 KARLSRUHE 1
(F.R.G)

1) Members of the project

Uwe KASTENS (now departed to Paderborn University), Brigitte HUTT-ASBROCK, Erich ZIMMERMANN, P. DENKER.

The present correspondent is E. ZIMMERMANN in Karlsruhe.

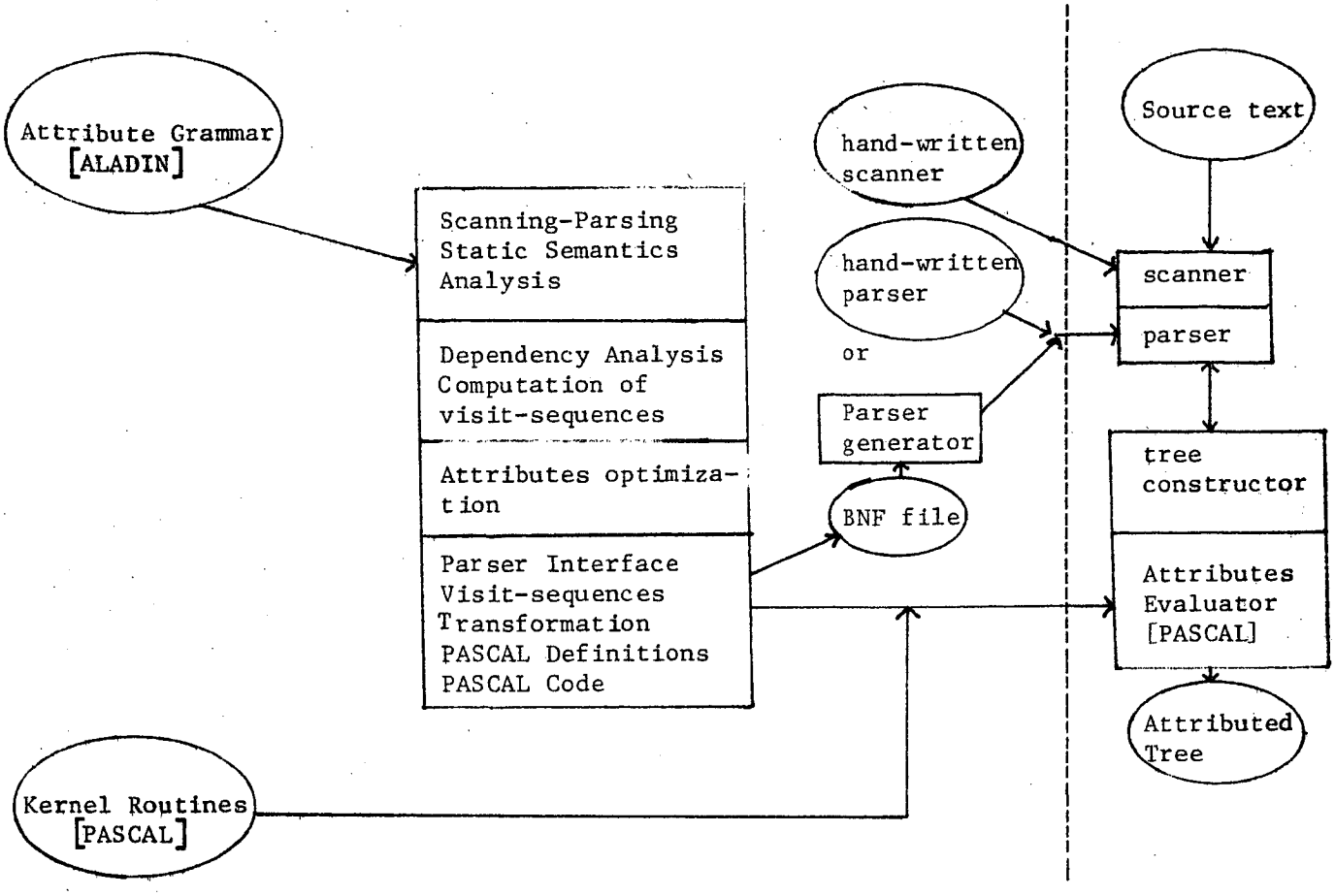
2) Birthdate : 1975 Deadline : 1981

3) General Features

Lexical analysis :	not part of GAG
Syntactic analysis :	
Attributes evaluation :	visit sequences (ascendent evaluator)
AG class :	ordered, or automatically arranged orderly
AG language :	ALADIN
Code generation :	see below.

4) Schema

GAG is written in standard Pascal and runs on a large number of machines.



5) General Comments

a) about lexical and syntactic analysis : this part is not included in GAG. From the input AG, GAG produces a file containing the syntactic part of the AG (BNF productions) plus tree-building procedure calls to be executed at each reduction. An LALR(1) parser generator named PGS is available from Karlsruhe. The tree constructor itself is generated by GAG.

b) about the input AG : it must be written in ALADIN (A Language for Attributed DefINitions). ALADIN is a strongly typed, applicative language. The possible types are basic types, subranges, sets and structured types. Predefined types are INT, BOOL, CHAR, STRING and SYMB (terminal symbols of the language). Other basic types are enumeration types a la Pascal, and subranges of scalar types. Structured types include union types a la Algol 68, structure types (records without variants) and (linear) lists. A number of predefined operators and functions are supplied with their usual meaning (e.g. HEAD and TAIL). The grammar writer can define constants,

types and (recursive) functions. Since ALADIN is an applicative language, there is no control flow structure but rather value-producing selection structures : these are the CASE "statement" and the IF-THEN-ELSE conditional. ALADIN is completed by a LET clause a la Lisp, types conversions and type and symbol tests.

The syntactic part of the AG is written in Extended BNF, i.e. RHSs can be regular expressions. However alternatives are forbidden (they must be separate productions) and repetition clauses may not be nested. A number of laws apply for attributes of symbols appearing in a repetition clause, involving list processing.

All the attributes, their ALADIN type and their attachement to the grammar symbols must be declared, but declaring their direction (synthesized or inherited) is optional.

Semantic rules are normal rules, transfer rules or context conditions. A transfer rule is an abbreviation for multiple copy rules. A context condition is a boolean expression referencing attributes occurrences ; this expression must be true at evaluation time, else an error message is generated and, maybe, a default value is returned. Context conditions can stand alone, i.e. apply to a whole production, or be included in "normal" semantic rules. Some context conditions are generated implicitly upon use of certain ALADIN constructs.

ALADIN also provides for non-local attributes access, with the two clauses INCLUDING (refering to attributes of nodes upward in the tree) and CONSTITUENT (refering to attributes of nodes downward in the tree).

c) about the evaluation method : GAG is a straightforward implementation of the evaluation method by Kastens [Kas 80] ; the tree traversal and semantic rules evaluation are directed by visit sequences. The only (but important) differences stand in the storage optimizations which will be discussed in the next section.

d) about the results of the evaluation : the normal result is the completely attributed tree. However GAG supplies predefined ALADIN output routines for attribute values, which ease interfacing with some back-end.

6) Optimizations and performance

The optimizations are numerous, and represent probably the most attractive feature of GAG.

a) about the evaluation strategy : the visit-sequences are encoded efficiently in a table. It is possible that two or more visit-sequences share some table entries. Any sequence of "evaluate" operations is combined with one of the tree-walking operations. The typical size of such a table is 1 Kbyte.

GAG also accepts some AGs which are not directly ordered. With a more careful analysis it is possible that the AG can be "automatically arranged orderly" ; if so, GAG can compute the corresponding visit-sequences [Kas 80, Kas 84].

b) about the tree representation : the tree is compacted and "abstracted" by eliminating "chain productions" (simple productions) and useless terminals. The remaining tree nodes are linked together by two pointers, one to the leftmost son and one to the right brother. Experience shows that accessing a descendent with this structure is not slower than with a direct (e.g. indexed) access, and the size of the tree is reduced by 25 %.

c) about the mapping of ALADIN types to Pascal types : since ALADIN is a strictly applicative language with no side effects, structured types are implemented using pointers. Thus a STRUCT or UNION type becomes a pointer to a Pascal record (with variants for UNION), and a LIST is implemented by a pointer to a chain of Pascal records linked together. Actually the anchor of a list is composed of two pointers, one to the head of the list and one to its last element ; this allows an efficient implementation of list concatenation. Using pointers, copying structured values reduces to copying pointers ; moreover, list structures can often be shared among several attributes values. ALADIN simple types are mapped 1 : 1 to Pascal simple types.

d) about attributes values storage : GAG performs a complete lifetime analysis to determine if a given attribute :

- can be stored in a global variable (pairwise disjoint attributes instances lifetimes),
- can be stored in a global stack (property included lifetimes),

- or must be stored nodewise (any other case).

This lifetime analysis is based on the visit sequences and the rules concerning the computation and use of attributes values. It is described in [Asl 79, KHZ 82, Kas 84]. Furthermore it is possible to store two or more different attributes in the same global variable or global stack, provided that the lifetimes agree and the Pascal types are compatible.

As an example, for a Pascal front-end with 140 attributes occurrences, 59 are stored nodewise and the remaining are stored in 11 global variables and 18 global stacks. The total attributes storage is reduced to 25 % of the tree size. However the storage of dead attributes stored in the heap is not reclaimed because Pascal does not provide an effective mean therefore.

e) about the ALADIN to Pascal translation : common compilation techniques are applied during this phase, including common subexpressions elimination (stored and considered as attributes...), tail recursion elimination (mandatory because the "loop" concept is absent from ALADIN, a purely applicative language), and inline coding of a number of predefined functions.

f) about the system : GAG is implemented in standard Pascal. Portability was one of the design requirements, and GAG comes with a configuring preprocessor, named PROPP, to adapt it to each particular implementation. The generated compiler is written in Pascal.

The whole GAG system is 44000 Pascal lines long, but is very modular. This figure includes the support modules, in particular PROPP, but not the parser generator. GAG has been widely distributed over the world.

GAG supplies extensive statistics about its operation either at generation time or at evaluation time.

7) Applications and Performances

In [KHZ 82] a complete AG describing a Pascal front-end is given. This 3000 lines AG is processed by GAG in 124 secs (*), producing a 10700 lines compiler

---

(*) All the figures are drawn from [KHZ 82], running on a SIEMENS 7.760 machine (similar to IBM 360,370).

with 4 visits maximum to any symbol. This (pure) front-end analyzes source texts at at a speed of about 350 tokens/sec. The full front-end with scanner and parser (generated by PGS) runs at 250 tokens/sec. The space needed to analyze a 10.000 tokens text is roughly 500 Kbytes of heap storage plus 50 Kbytes of stack space. These figures are, according to the authors, very close to those of other compilers using a tree as internal structure.

The generated Pascal front-end compiles to a 150 Kbytes code after all the Pascal checks have been removed. These checks are useless since most of them are performed by GAG itself on the ALADIN text.

Apart from toy languages, front-ends were generated for Pascal [KHZ 82], Ada [UDP 82], LIS and PEARL [DIN 80].

The Pascal front-end has been interfaced with a code generator produced automatically by the CGSS system developped in Karlsruhe.

8) Projects

None : the project is terminated.

9) References

Kas 80, Kas 84, KHZ 82, Asb 79, DIN 80, UDP 82.

HLP 78

Helsinki Language Processor

Martti TIENARI
University of Helsinki
Dept of Computer Science
Tukholmankatu 2
SF-00250 HELSINKI 25
(FINLAND)

1) Members of the project

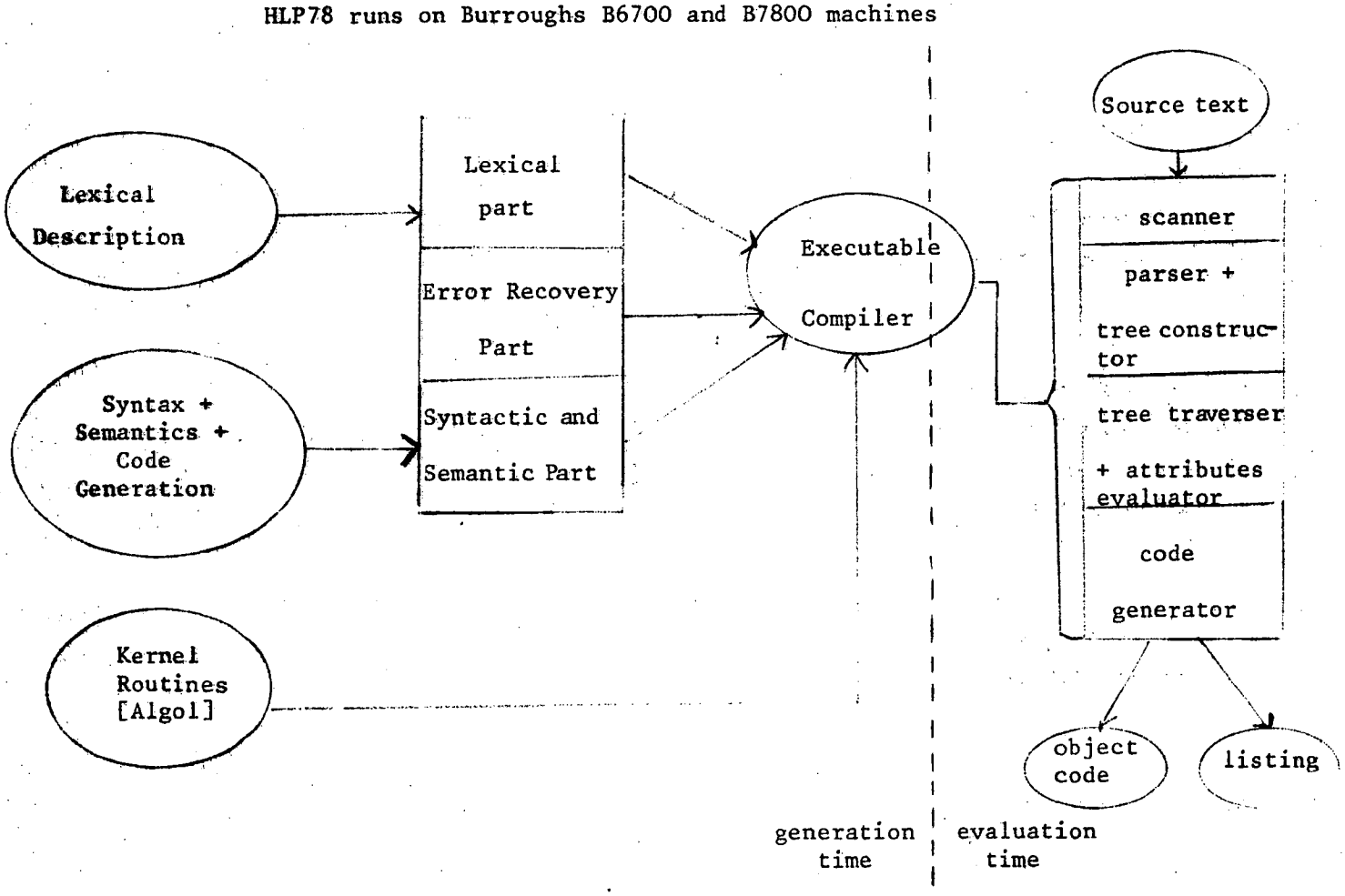
Martti Tienari (head), Mikko Saarinen (lexical part), Seppo Sippu, Eljas Soisalon-Soininen (syntactic part), Kari-Jouko Räihä, Matti Sarjakoski (attributes evaluation, code generation, control part), Lassi Juntinen, Kai Koskimies, and many students.

2) Birthdate : 1975 Deadline : 1982

2) General Features

Lexical Analysis	:	regular expression + screening
Syntactic Analysis	:	LALR(1) with error recovery
Attributes Evaluation	:	Alternating Semantic Evaluator
AG class	:	simple multi-pass
AG language	:	based on Burroughs Extended Algol + features for code generation
Code generation	:	"code templates"

4) Schema


HLP78 runs on Burroughs B6700 and B7800 machines
Lexical Description
Syntax + Semantics + Code Generation
Kernel Routines [Algol]
Lexical part
Error Recovery Part
Syntactic and Semantic Part
Executable Compiler
Source text
scanner
parser + tree constructor
tree traverser + attributes evaluator
code generator
object code
listing
generation time
evaluation time

5) General Comments

a) about attributes evaluation

HLP78 uses the most well-known attributes evaluation method : evaluation in passes, derived from the Alternating Semantic Evaluator (ASE) [JW75]. The first pass is carried out during parsing, so it can contain only synthesized attributes and may thus be empty. The algorithm used to assign attributes to passes is an improved version given in [RU81].

b) about code generation : code generation is performed as the last traversal of the tree. Code generation instructions are of two kinds : "visit a subtree" and "write some code onto some file under some condition" (code templates). The grammar writer has total control over the tree walk, which is similar to those performed in "one sweep" evaluation.

c) about the input metalanguages : there are two metalanguages to write the inputs to HLP78, one for the lexical description and one for all the remaining : syntax, attributes, code generation.

Both are designed such that their aspect is close to English. As a consequence they look rather verbose, but they are easier to read.

An attribute grammar to be input to HLP78 is composed of :

- (optional) memories (i.e. constants) declarations ;
- attributes declarations : each attribute must be declared with its kind (direction) and (Algol) data type (simple type or array) ;
- non-terminals declarations (since nothing can differentiate a non-terminal from a terminal in the productions) : each non-terminal must be declared with its attributes ;

- start symbol declaration ;
- (optional) Algol formats declarations ;
- (optional, but usually required) procedures and functions declarations (written in Algol) ;

- production descriptions ; each production comes with its associated semantic rules and code generation instructions.

Some semantic rules and the code generation instructions may be omitted ; the system will apply default rules to generate the missing ones. For code generation the default is to visit the subtrees in left-to-right order, and emit no code.

Normal semantic rules are either assignments statements or procedure calls. Those procedure calls must specify which are the output parameters (attributes occurrences) ; input parameters are expressions made of constants, attributes occurrences, usual operators and an applicative if-then-else. Procedure calls may compute simultaneously (i.e. in a single semantic rule) several attributes occurrences.

Predefined "attributes" and subroutines allow to retrieve the text of a token or its "unique number", to emit error messages refering to the line/column coordinates of the error point, to abort the translation, etc.

6) Optimizations and performance

a) tree-walk optimization : if, in a given pass, a given subtree contains no attribute to be evaluated in this pass, it is skipped. This explains why the different passes are implemented by recursive procedures.

b) storage management : since the only structured data type available in HLP78 is arrays, copying such structures can be very expensive ; such copies can be involved either in normal semantic rules (through assignments) or inside procedures. HLP78 provides a mean to avoid such copies inside procedures when they are not necessary : each procedure may start with a "copy block" describing those copies. The system will then automatically check each call of those procedures to determine whether the copy is actually necessary, i.e. whether the corresponding "old" attribute instance is not yet useless.

The parse tree does not contain unit (i.e. simple) productions with trivial semantics, nor proper terminals (e.g. keywords).

Attributes values storage is reduced by a very efficient dynamic allocation technique [Raï 79]. Attributes values are accessed through pointers, and attributes instances that are given the same value by copy rules point to the same storage area. Moreover the system performs at evaluation time (during parsing) an analysis to find out the moment after which each such area becomes useless (i.e. all the attributes instances pointing to this area become useless). Then the allocator can reuse the area for another class of attributes instances.

c) performances : on the Burroughs B7800, front-ends generated by HLP78 run at a speed of 5000 to 6000 lines per minute, i.e. 3 to 4 times slower than hand-written compilers. As for space, the analysis of a Pascal text 1000 lines and 4700 tokens long requires 250kbytes of workspace [KRS82].

The size of HLP78 is about 35000 Algol lines.

7) Applications

HLP78 has generated compilers or front-ends for more than ten real languages, including PL360, Simula, Pascal, Ada and Euclid. It has also been heavily used as an educational tool.

8) Projects

The HLP78 project is terminated.

The same team has started in 1981 a new project to develop another translator writing system based on AGs, named HLP84. Ils main goals are portability, ease of use and efficiency. It will be written in Pascal and will generate compilers written in Pascal. Il will provide a variety of parsers (LR(1), SLR(1), LALR(1), and LL(1)), and many shorthand notations will ease the task of specifying typical compiler operations. For efficiency, the AG class will be restricted to thoses for which it is possible to evaluate all the attributes during parsing [Tar82, KR83]

9) References

JW75, RU81, KRS82, Kor80, Raï76a, Raï80a, Raï84, RSS78, RSS83, Tie79, Tie80

HLP / SZ

Helsinki Language Processor / Szeged

Tibor GYIMOTHY, Endre SIMON
Research Group on Theory of Automata
Hungarian Academy of Science
Somogyi u.7
H-6720 SZEGED
(HUNGARY)

1) Members of the project

Endre SIMON, Tibor GYIMOTHY, Arpad MAKAY, Janos TOCKZI, Tamas GARAI, Ferenc KOCSIS.

2) Birthdate : 1979 Deadline : none

3) General Features

Lexical Analysis :	regular expressions + screening
Syntactic Analysis :	ELR(1), ELALR(1), ESLR(1) and ELR(0)
Attributes Evaluation :	1) Alternating passes 2) Visit sequences
AG Class :	1) simple multi-LR 2) OAG
AG Language :	those of HLP 78 (q.v.)
Code Generation :	no special feature provided.

4) Schema

The same as HLP 78 (q.v.). In fact HLP/SZ is another implementation (and an extension) of HLP 78.

HLP/SZ is implemented in SIMULA-67 and generates compilers written in SIMULA-67. It runs on a CDC-3300 computer with 64 k words (of 24 bits) of main memory.

5) General Comments

The choice of the parsing method (ELR(1), ELALR(1), ESLR(1), ELR(0)) is done automatically by the system.

The two attributes evaluation methods correspond to the use of alternating passes [JW 75] and visit-sequences for ordered AGs [Kas 80].

6) Optimizations

a) ASE method : in a given pass a subtree will not be visited if it contains no attributes to be evaluated during that pass. Also after each pass subtrees such that all their (synthesized) attributes are evaluated are deleted (the SIMULA run-time system includes a garbage-collector). Lastly, the first evaluation pass is combined with (bottom-up) parsing, and computes purely-synthesized attributes.

b) OAG method : this method is modified to accept a larger class of non-circular AGs. The modification deals with the manner in which the dependencies are transformed into a total order [GSM 83].

7) Applications

- the two metalanguages of HLP/SZ
- a Pascal subset (100 productions)
- Toy languages
- a front-end for a block-structured language with concurrent blocks.

The total size of HLP/SZ is about 15 000 SIMULA-67 lines. No information is available about its performances.

8) Projects

A new project started in 1983, aiming to implement a compiler writing system in Pascal, following the general lines of HLP/SZ but with both LL and IR parsing techniques.

9) References

GSM 83, Sim 84.

LILA
Language Implementation LAboratory
Johan LEWI
Computer Science Dept.
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3030 LEUVEN
(BELGIUM)

1) Members of the Project

Johan LEWI, Karel DE VLAMINCK, Jean HUENS, Michel HUYBRECHTS, Eric STEEGMANS, and others.

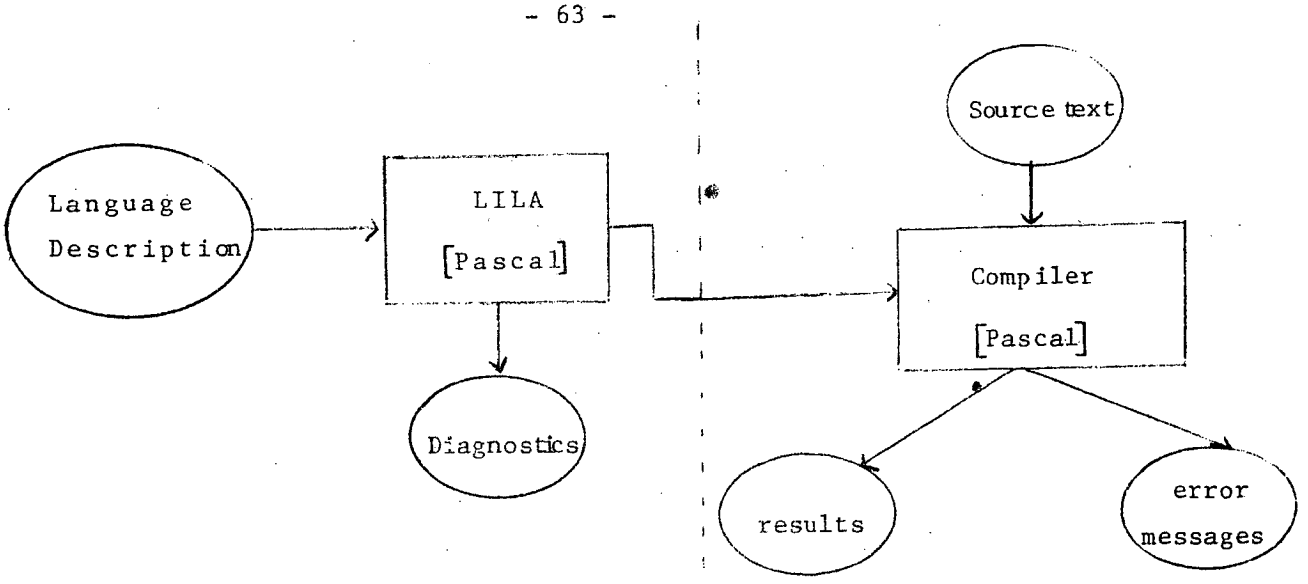
2) Birthdate : 1973 Deadline: ?

3) General Features

Lexical Analysis	: regular expressions
Syntactic Analysis	: ELL(1) with error recovery
Attributes Evaluation	: during the recursive descent parsing
AG Class	: 1L-AGs
AG Language	: based on Pascal
Code Generation	: see below

4) Schema

LILA is written in standard Pascal and thus runs on a large number of machines. The generated evaluators are written in Pascal.



5) General Comments

Basically LILA is a parser generator which has been extended to deal with (semantic) actions and attributes. The input to LILA is what the authors call a generalized attributed extended context-free grammar (AECF-grammar), i.e. a grammar written in extended BNF (meta-operators are exclusive and inclusive alternance, non-negative and positive repetition, and grouping), with semantic actions calls embedded in the productions. These actions, which are described in Pascal just after the production, can reference the (unique) attribute of each (terminal or non-terminal) symbol appearing in the production. However, since each attribute can be a record containing several fields, this is the same as several attributes per symbol. Some of the fields represent inherited attributes and the others synthesized ones; LILA checks whether they are used consistently.

Given an AECF-grammar, LILA checks whether it belongs to the AELL(1) class, that is :

- the underlying grammar is an Extended LL(1) grammar;
- the attributes can be evaluated in a single left-to-right pass (1L-AG).

If the grammar passes the test, LILA produces a recursive descent parser, with a procedure for each non-terminal; the attribute of a given non-terminal is represented by a parameter passed by reference. The semantic actions are implemented by procedures local to each non-terminal-procedure.

Apart from the input (terminal) and non-terminal vocabularies, LILA provides for defining an output vocabulary. Each output symbol may have an attribute, and LILA produces a procedure to output those symbols together with the value of their attribute. However the grammar writer may supply his own output routine, which

acts as a (very simple) code generation part.

The lexical part is defined in exactly the same way, except that the grammar must be an extended type-3 grammar, i.e. it must not use recursion. From this grammar, LILA produces a finite-state scanner recognizing the language. The output symbols of this grammar are the input tokens of the syntactic grammar; their attributes are evaluated in the scanner and may be used in the syntactic grammar. LILA supplies a standard "read-in" procedure for the scanner, but the user may also redefine it.

LILA has two error recovery mechanisms, one for interactive input (Skip, Undo and Redo) and one for batch input (Skip, Recognize and Synchronize). The grammar writer must state in his grammar at which points and how the synchronization will work.

6) Optimizations

None

7) Applications and performances

* teaching compiler construction
* toy languages
* desk calculators
* compilers or front-ends for Atlas, REGTRAL, Alto, Chill and Ada

We have no information on the size and performances of LILA. Its authors claim that LILA and the generated programs are "simple, reliable, adaptable, portable and efficient".

8) Projects

The industrial development of LILA, now called MIRA, has been taken over by :

Expert Software Systems n.v.
Software Engineering Product Division
Building "De Schelde"
Moutstraat 100
B-9000 GHENT
(BELGIUM)

MIRA runs on every computer which has a standard Pascal compiler (IBM PC-AT or XT under MS-DOS and compatibles, APOLLO, DEC, SUN, VAX (VMS or UNIX), HP3000, IBM 3033 and 3081 (under MVS and VMS), Siemens 7500 family (under BS2000), etc.). It is able to produce compilers in Pascal, C or Ada, provided that the semantic actions are written in that target language.

9) References

LDH75, LDH77, LDH79

LINGUA
Georg Heeg, Enno de Vries
Abteilung Informatik
Universität Dortmund
Postfach 500500
D-4600 DORTMUND 50
(F.R.G.)

1) Members of the project

P. WILMES, G. HEEG, E. DE VRIES.

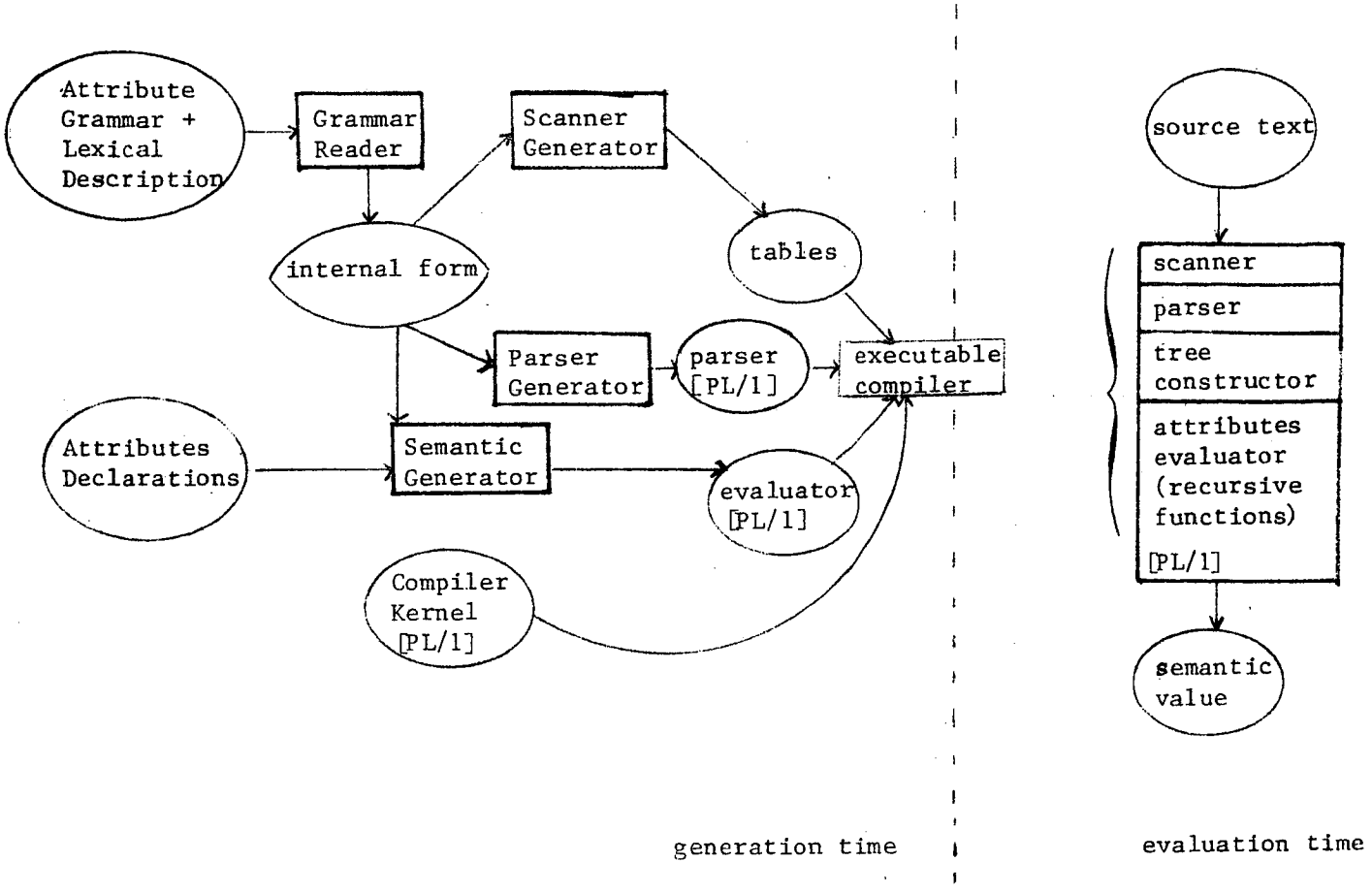
2) Birthdate : 1973 Deadline : 1980 (?)

3) General Features

Lexical Analysis :	left linear grammars (regular expressions)
Syntactic Analysis :	LALR(k)
Attributes Evaluation :	Recursive evaluation by need
AG Class :	unrestricted
AG Language :	based on PL/1
Code Generation :	no special feature provided.

4) Schema

LINGUA is written in PL/1, produces compilers written in PL/1 and runs on SIEMENS BS2000 and IBM 360/370 under OS2.



5) General Comments

The attributes evaluation strategy is a recursive evaluation by need, very close to the one used in the ERN system (see the FNC/ERN system) : each attribute is turned into a function, taking as parameter a tree (or tree node) and returning the value of that attribute at that node. Each function recursively calls other functions to evaluate other attributes if needed. An attribute value, once computed, is stored in the tree in order to avoid reevaluating it. The process is started by calling the functions corresponding to the (synthesized) attributes of the tree root, which form the "semantic value".

This strategy is efficient because it reduces the number of attributes instances to be evaluated to those really necessary to compute the semantic value. However this "necessity" is purely static here, as opposed to the ERN method.

True circularities are detected dynamically, by marking each attribute instance which is needed but not yet computed (that is, during its computation). However, if the grammar proves to be absolutely non-circular, which is optionally tested at generation time, this dynamic test is disabled.

The metalanguage is rather classical. Semantic rules are written in PL/1 and the attributes data types are PL/1 types (those which can be returned by a function, which still leaves a great choice). One of the remarkable features of this language is that it is completely parametrizable at the lexical and syntactic levels ; that is, the grammar writer can tailor the appearance of his AG to his own tastes. Another feature is that the lexical description is a BNF left-linear grammar, augmented with an ALLBUT construct.

The system provides extensive information about the generation and evaluation processes.

The parser is generated as a tailor-suited PL/1 program rather than as a set of tables for a fixed parser, which is the case for the scanner.

6) Optimizations

None.

7) Applications and Performances

. a pilot implementation of the control language NICOLA.

. a compiler of a subset of Pascal to P-code consisting of 110 (syntactic) productions.

For the last application, the scanner generator took a few CPU seconds, the parser generator (the AG is LALR(2)) 15 seconds, and the semantic generator 180 seconds. The PL/1 compiler took 250 seconds. The generated compiler analyzes 120-lines Pascal texts in 5 seconds and uses 236 kbytes of (dynamic) memory. It consists of about 400 functions.

We have no information about the size of the system.

8) Projects

The authors speak of an unknown LINGUA II.

9) References

HV 80.

LINGUIST - 86

Rodney FARROW
Intel Corporation

1) Members of the project

Rodney FARROW (now departed), Sue OJEDA, Tuyen QUOC, Al HARTMANN, Tom WILCOX.

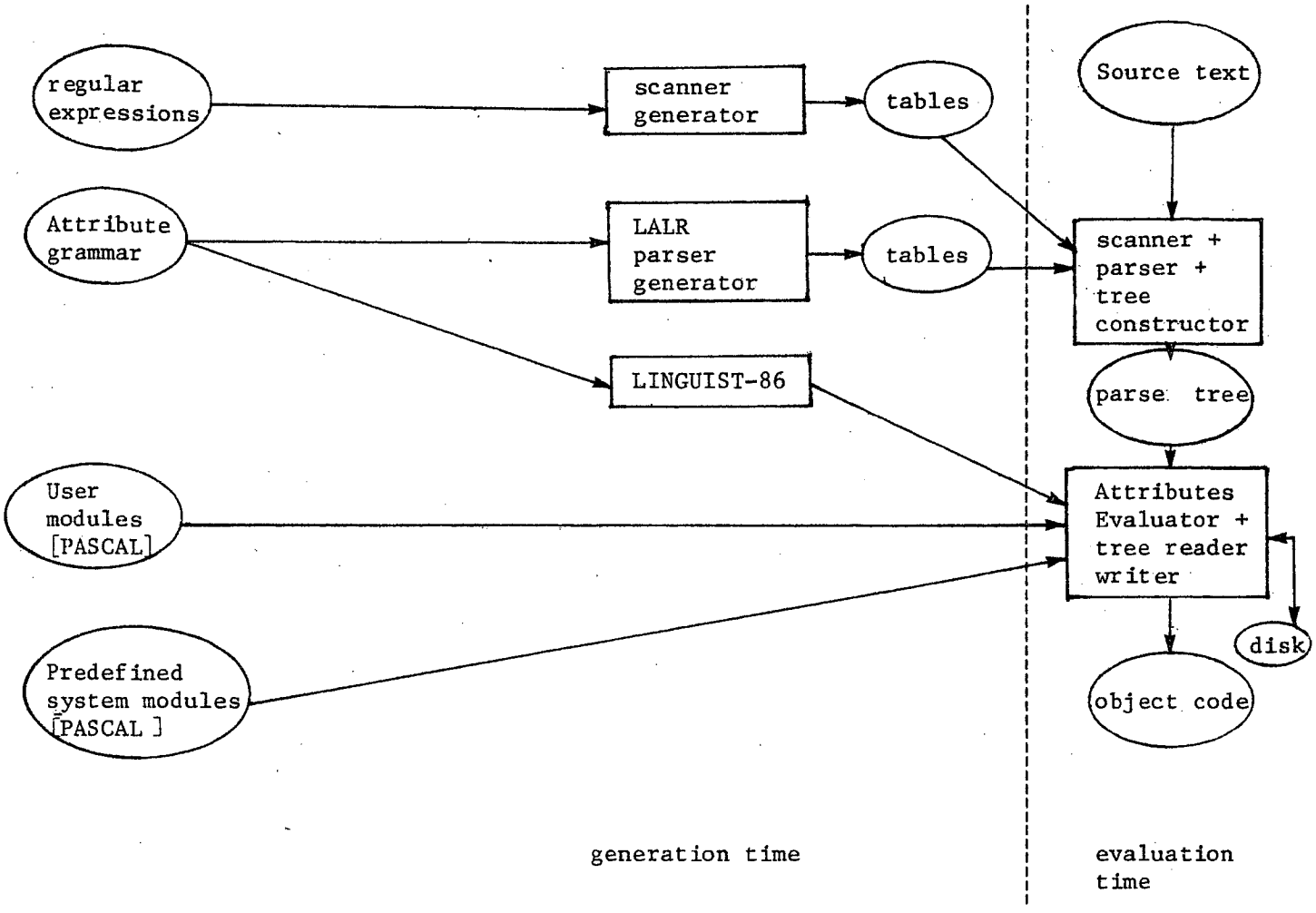
2) Birthdate : ? Deadline : 1983

3) General Features

Lexical Analysis :	regular expressions
Syntactic Analysis :	LALR or LL
Attributes Evaluation :	alternating passes
AG Class :	simple multi-LR
AG Language :	based on Pascal
Code Generation :	no special feature provided.

4) Schema

LINGUIST-86 runs on Intel 8086-based systems with floppy disk or hard disk secondary memory. It is written in Pascal and produces compilers written in Pascal.



5) General Comments

a) about the evaluation strategy

The evaluation stragtegy is basically an alternating passes evaluation, where the attributed parse tree is stored mainly in secondary memory (disks). This method is derived from [Sch76]. In a left-to-right pass for instance, a node is read from the file onto the stack, its inherited attributes (i.e. those for the corresponding pass) are evaluated, and then, for each of its sons (from left to right), the following process is repeated : read the node onto the stack, evaluate its synthesized attributes, write it onto the file. Then evaluate the synthesized attributes of the former node, write it on the file and pop its sons from the stack. When the pass is ended, the file is ready for being input to a right-to-left pass (which implies that the passes are strictly alternating).

This process is implemented through a set of recursive procedures, one for each production and each pass. Each procedure takes as parameter the node corresponding to the LHS of the production, which has been read and allocated onto the stack by the caller.

The alternating-pass test is polynomial, and the whole is easy to implement. It is very efficient in space since main memory is limited to 48 kbytes on the host system (8086-based) and yet everything runs ! However the whole process consumes much time and space in secondary storage.

LINGUIST-86 can generate an evaluator to be interfaced with either a bottom-up parser (in which case the first pass, which is combined with parsing, must be R-to-L) or a top-down (first pass L-to-R). The choice is left to the grammar writer. However the system comes with only an LALR (bottom-up) parser generator.

b) about the use of the system

The semantic rules are written in an expression language based on Pascal and augmented with a functional if-then-else. The system predefines two attributes for terminal symbols.

Each non-simple production has also a "limb symbol" which is the third type of grammar symbol (in addition to terminals and non-terminals). It is used to construct the name of the production-procedures, but it may also have attributes which serve as local names inside the production, e.g. to store common subexpressions.

LINGUIST-86 has also a facility to generate some of the missing semantic rules.

A semantic rule may define several attributes occurrences at a time. This is useful for instance if their values depend on the same condition, which will then be tested only once.

6) Optimizations

They are all related to space management.

The most efficient space-saving optimization is called "static subsumption". Its effect is to store (whenever possible) "temporary" attributes (defined as attributes which are used only in the current pass) into global variables rather than nodewise or onto the stack. The effect is to reduce the size of the tree nodes (secondary storage) and the size of local storage on the stack (main memory). It also saves some code in the evaluation procedures (by 13 to 20 %) since it eliminates a large number of copy rules. Static subsumption is related to the investigation of the storage allocation problem by Ganzinger [Gan 79b] and to some optimizations implemented in the GAG system (q.v) (see [Far 82b, FY 85]).

Other space-saving optimizations are not to store in the tree temporary attributes, and to loosen somewhat the order of attributes evaluation in each production-procedure.

7) Applications and performances

LINGUIST-86 was designed to be used by INTEL Corp. to develop language translators and other processors for their machines.

The two applications quoted in [Far 82b] are a Pascal front-end and LINGUIST-86 itself, which is bootstrapped as a 1800-lines AG evaluable in 4 passes, first R-to-L.

LINGUIST-86 is about 55 kbytes of code and processes its input at 350 to 500 lines per minute (front-end only), which is competitive with hand-written compilers running on the same machine (400 to 900). Experience shows that most of the processing time of LINGUIST-86 and the generated compilers is spent reading and writing the tree from and to secondary storage.

8) Projects

Unknown. Rodney Farrow is now at Columbia University in New York.

9) References

Far 82a, Far 82b, FY 85

MUG 1

Modularer Übersetzer Generator

J. EICKEL

Institut Für Informatik

Technische Universität München

Arcisstrasse 21 - Postfach 202420

D-8000 MÜNCHEN 2

(F.R.G.)

1) Members of the project

J. EICKEL (head), R. WILHELM, K. RIPKEN, H. GANZINGER, J. CIESINGER, W. LAHNER, R. NOLLMANN.

2) Birthdate : 1972 Deadline : 1975

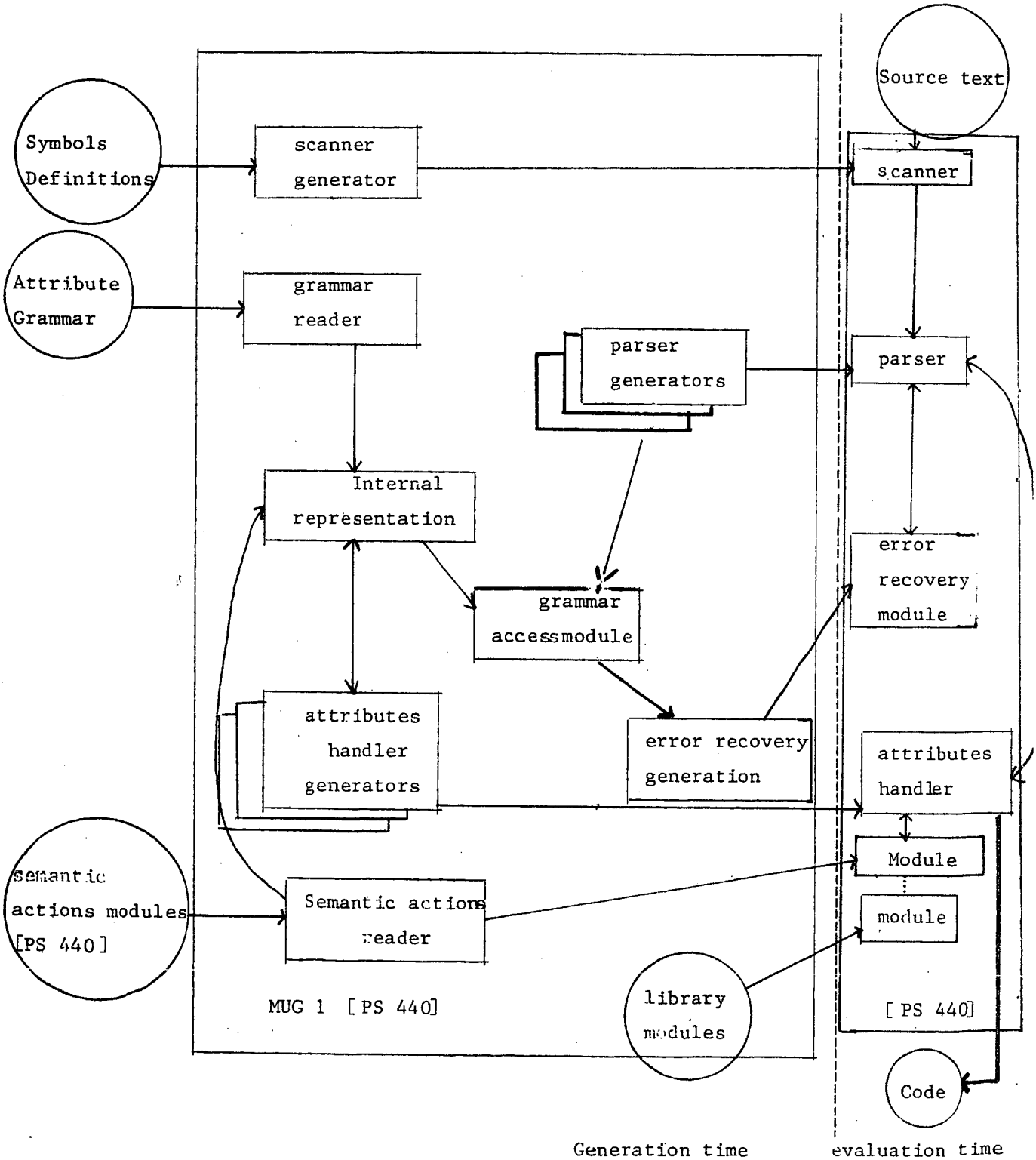
3) General Features

Lexical Analysis :	regular expressions
Syntactic Analysis :	LL(k), LR(k), LALR(k) with error recovery
Attributes Evaluation :	in parallel with parsing
AG Class :	IL - AGs
AG Language	calls to PS 440(*) procedures embedded in the CF-grammar
Code Generation	see below

(*) PS 440 is the system implementation language of the Telefunken TR 440 computer

4) Schema

MUG 1 runs on the Telefunken TR 440 machine.



5) General Comments

MUG1 generates one-pass compilers. The semantic rules are written as procedure calls embedded in the CF-grammar. Each procedure has in-arguments (inherited attributes) and out-arguments (synthesized attributes). Non-terminals may also have such in- and out-arguments. In fact attribute grammars in the sense of MUG1 are very close to affix grammars [Kos 71b]. The AG is restricted to be an 1L-AG.

Attributes evaluation is carried out in parallel with syntactic analysis. In the top-down case, the attributes handler manages the stack(s) of attributes values and the calls to the procedures. In the bottom-up case, the underlying CF-grammar must be transformed in order to force the parser to do a top-down simulation in some cases where inherited attributes are to be transfered. This transformation is performed automatically by MUG1, but the resultant grammar may not be LR(k) any more because of the introduction of empty-RHS productions.

Special routines are provided to translate intermediate code to machine code ; those are not part of the AG.

One of the main goals of MUG1 was modularity and efficiency of the generated compilers. This explains those strong restrictions.

6) Optimizations

None.

7) Applications

. Toy languages

. A subset of Algol 60, which served as a basis for a student compiler laboratory project.

. It was intended to "bootstrap" MUG1 in some way in order to transport it and the generated compilers onto other machines (PDP11).

. MUG1 was used for several years for teaching compiler courses

MUG 2

Modularer Übersetzer Generator 2

J. EICKEL (head), H. GANZINGER,
R. GIEGERICH, K. RIPKEN, R. WILHELM
Institut Für Informatik
Technische Universität München
Arcisstrasse 21-Postfach 202420
D-8000 MÜNCHEN 2
(F.R.G.)

Preliminary Notice :

MUG 2 was originally developped in Munich as a successor of MUG1. However MUG2 is considered by its authors as a research tool in which to implement theoretical ideas rather than as a production-quality system. This explains in part why the design and implementation of MUG 2 are so fuzzy and change quickly. We shall describe its state as of late 1983.

An other point is that the original team in Munich has split into three teams located in Munich, Dortmund and Saarbrücken, and each team concentrates on different aspects of the compilation process; thus MUG 2 is now a three-headed system called respectively MUG 2, M200 and OPTRAN, written in different languages and running on different systems.

MUG 2 in Munich is written in Pascal and runs on a Siemens machine under VM/CMS. M200 is written in Modula-2 and runs on Unix. OPTRAN is written in Pascal and runs on Unix.

Addresses :

. Abteilung Informatik
Universität Dortmund
Potfach 500500
D-4600 DORTMUND 50 (F.R.G.)

. Fachberich 10 - Informatik
Universität des Saarlandes
D-6600 SAARBRÜCKEN (F.R.G.)

1) Members of the project

In Munich : J. Eickel, G. Bartmuss*, G. Jochum*, A. Liebl, K. Ripken*, M. Storz, S. Thürmel, F. Welty*, W. Willmertinger*, H. Wittner.

In Dortmund : H. Ganzinger+, M. Vach.
In Saarbrücken : R. Wilhelm+, I. Glasner, U. Möncke, B. Weisgerber.

2) Birthdate : 1975 Deadline : 1985 ?

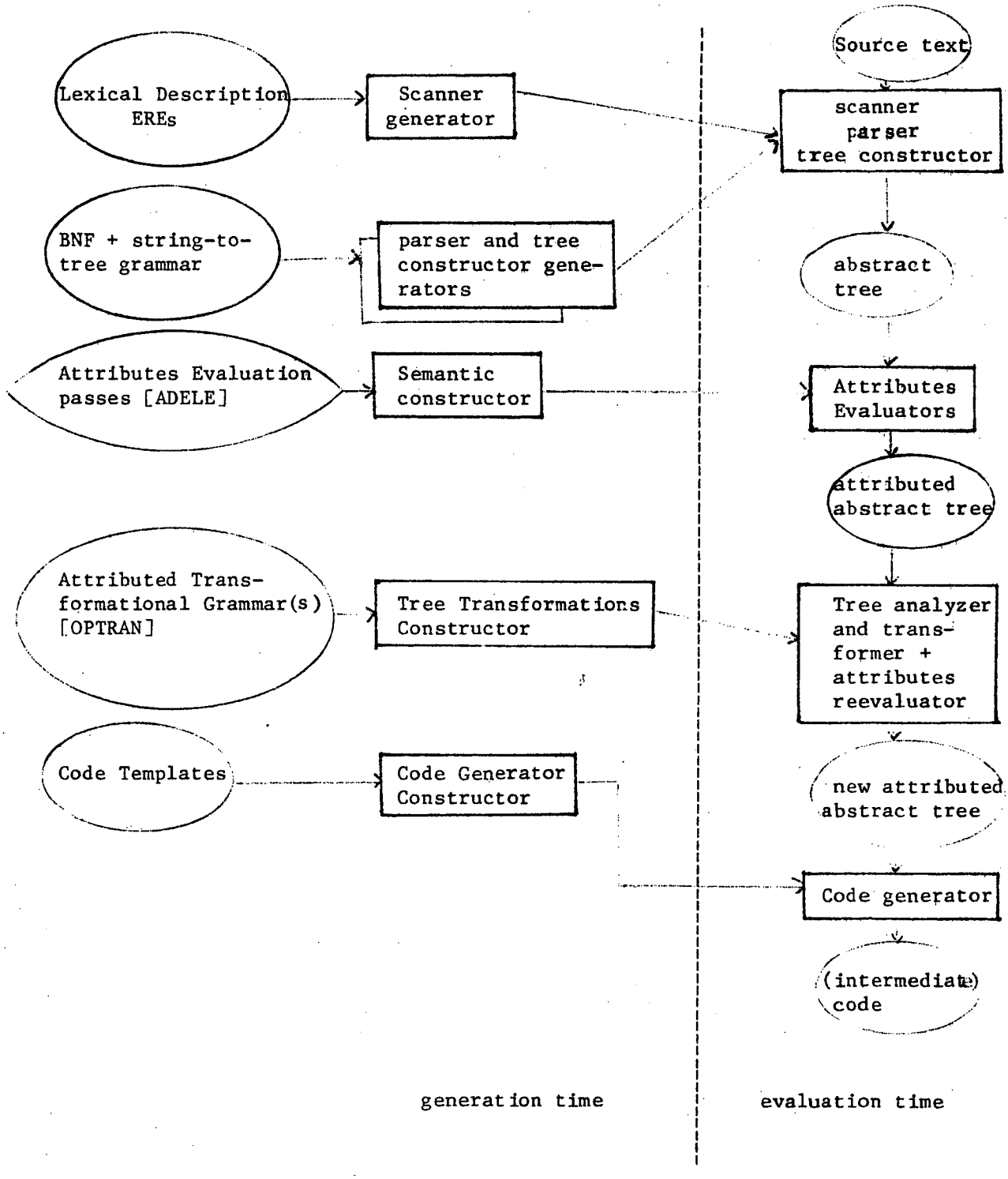
3) General Features :

Lexical Analysis	:	extended regular expressions
Syntactic Analysis	:	LL(k), LALR(k) with error recovery and abstract tree construction
Attributes Evaluation	:	several one-sweep passes
AG Class	:	modified simple multi-sweep
AG Language		ADELE
Code Generation	:	optimization through attributed transformational grammars (OPTRAN) + code templates.

* : left to industry

\+ : formerly in Munich.

4) Schema


Lexical Description EREs
Scanner generator
BNF + string-to-tree grammar
parser and tree constructor generators
Attributes Evaluation passes [ADELE]
Semantic constructor
Attributed Transformational Grammar(s) [OPTRAN]
Tree Transformations Constructor
Code Templates
Code Generator Constructor
Source text
scanner parser tree constructor
abstract tree
Attributes Evaluators
attributed abstract tree
Tree analyzer and transformer + attributes reevaluator
new attributed abstract tree
Code generator
(intermediate) code
generation time
evaluation time

5) General Comments

MUG2 is a complete compiler generator based on attribute grammars. Its main goals are modularity (both in the input descriptions and in the generated compilers), expressive power, efficiency and versatility.

As for pure attributes evaluation, considered as the operation of decorating some tree with some values, the two peculiar features of MUG2 are the use of abstract trees rather than concrete syntax trees, and attributed tree transformations.

a) about the "syntactic" part

This part contains the lexical analysis, the syntactic analysis and the abstract tree constructor.

The operators (nodes) of the abstract trees have a fixed arity. The same subtree may be duplicated to become the son of two or more different nodes [Gie79]. The abstract tree is constructed in parallel with parsing.

b) about attributes evaluation

The modularity in attributes evaluation is implemented in ADELE and in the generated compilers by the splitting of the AG into one or more passes (i.e. subgrammars), each corresponding to an evaluation sweep. This decomposition into passes is left to the grammar writer, and each pass must be evaluable by the "one-sweep" strategy [EF81c]. All passes operate on the same (abstract) tree, which will be decorated step by step.

An attribute grammar in ADELE is composed of the following parts [Gan82]:

- definition of the abstract syntax ;
- types and constants ;
- attributes declarations ;
- functions specifications ;
- definition of passes, including local attributes declarations, functional

attributes specifications and semantic rules (called "attributes statements").

The definition of the abstract syntax is mandatory because an ADELE AG can stand alone, without refering to a particular string-to-tree grammar. MUG2 will check that a given string-to-tree grammar indeed generates correct trees w.r.t. the given abstract syntax definition. The main characteristics of abstract grammars in ADELE are :

- ambiguity is allowed ;
- terminal symbols are automatically distinguished by not appearing as the LHS of a production ;
- simple productions are also distinguished (by the grammar writer) : they will not appear in the tree and cannot have associated semantic rules, apart from simple copy rules.

The types and constants part specifies Pascal-like and Pascal-compatible types and constants. In addition to those a concept of functional types is provided. Their implementation is some Pascal type about which nothing is known to the grammar writer. Functional types can be (indirectly) recursive.

Attributes declarations specify the ADELE type and kind (i.e. synthesized or inherited) of each attribute. The attachment of attributes to nodes will be infered from the attributes statements. Two predefined "lexical" attributes are provided.

Functions specifications declare the profile of external functions and/or procedures, which must be supplied separately.

Attributes statements (semantic rules) are associated with a particular (abstract) production. They are written as assignments of expressions to attributes occurrences as in any AG. An expression can be either a Pascal expression, another attribute occurrence, or the body of a Pascal procedure. Within such bodies it is possible to refer to other attribute occurrences and to access subcomponents of those if they have a structured type (record or array). Using a Pascal body it is possible to define several attributes occurrences in a single attribute statement.

A functional attribute is defined as an attribute (sub-) grammar. In ADELE any attribute (sub-) grammar "AG" defines the function "AG of X" for any node X

which is labelled with (one of) the start symbol(s) of "AG" in the tree. "AG of X" is then seen as a function mapping a tupel I of values for the inherited attributes at X to a tupel S of results as follows :

- initialize the inherited attributes at X with I ;
- evaluate the "AG" - attributes in the subtree rooted at X ;
- deliver the final values S of the synthesized attributes at X as the result. The attributes instances of "AG" evaluated by this "call" disappear after the results have been delivered.

Functional attributes values are computed by invoking such a process by the "eval" clause. They can be passed as normal attributes values to other nodes in the tree. A typical use of such functional attributes is to implement symbol table entries ; each entry will then be the subtree corresponding to the declaration, which can be traversed when "evaluating" those functions.

An (outer) attributes evaluation pass may contain local attributes and local functional types. Attributes which are declared locally cannot be accessed from other evaluation passes, and are therefore not stored in the tree but rather (because of the 1-sweep strategy) on a stack.

An evaluation pass may also contain circular attributes dependencies if they are defined as "iterative". If so, the attributes evaluation will be repeated iteratively until all the attributes instances values do not change during one pass (i.e. until "convergence"). Circular dependencies are used (by the keywords last and initially) to supply old values to the new iteration.This feature, which is only a "formalization" of what happens in real compilers, is very useful for e.g. data flow analysis.

It is also possible, in each semantic rule, to reference non-local attributes occurrences (of an ascendant node) with the enclosing construct. Because of the 1-sweep strategy, it is possible to store those enclosed attributes in global variables.

c) about attributed tree transformation

Tree transformations are a tool to specify and perform source level optimizations, after some evaluation passes have computed relevant information, e.g. data flow information. In MUG2, tree transformations are written in OPTRAN [GMW80, GGM82, MWW84]. They are a sequence of transformation units (t-units). A t-unit is a set of transformation rules composed of :

- an input template describing the original shape of a subtree ;
- a predicate, which is a boolean expression involving attributes of the nodes of the input template, and which tells whether the transformation is applicable ;
- an output template describing the final shape of the subtree ;
- attributes reevaluation rules, which tell how to compute the attributes of the new nodes in term of the old ones.

The t-unit is completed by a specification of the strategy to use when traversing and transforming the tree. OPTRAN offers several fixed strategies (e.g. bottom-up).

Tree transformations may map the input tree language onto itself (pure transformations) or onto another language (transfer transformations, e.g. to produce intermediate code).

After each application of a transformation rule the attributes of the restructured tree must be reevaluated to guarantee the consistency of the attributed tree. Reevaluation is not needed at all if the safeness of the transformation phase can be proven [GMW 80]. The reevaluation algorithm used in MUG2 is described in [MWW 84] and uses a table-driven automaton. The authors claim it is more efficient than the one of Reps [Rep 82b].

The output of a t-unit is an attributed tree which may be feeded to another t-unit or to other evaluation passes.

d) about code generation

Code generation is the part of MUG2 which is in the least advanced state. It is described by code templates : there is a code template for each operator (node) in the tree, and it is a sequence of actions which can be of three kinds :

- visit an operand (son) subtree ;
- compute some local values (e.g. labels) ;
- output some code using the node's attributes and those local values.

This method resembles the one used in HLP78 (q.v.). However those code templates may receive, use and pass some parameters.

Apart from that, only theoretical work has been undertaken, e.g. on peephole optimization.

6) Optimizations

The main optimizations are related to storage management.

The first point is that using abstract trees rather than concrete syntax trees reduces the storage needed for the tree, by getting rid of clumsy syntactic constraints.

As for attributes values, the splitting into one-sweep passes allows to apply the optimizations devised for evaluators in passes [Poz79, Räi79,...], and indeed they are. MUG2 allows for instance to determine (at generation time) the lifetime of attributes, distinguishing temporary (i.e. one-pass) attributes which can be stored on a stack [Gan 82].

7) Applications

. Toy languages.
. A sophisticated formatter for mathematical formulas.

No information is available on the size and actual organization of MUG2, nor its performances.

8) Projects

The future of MUG2 ([BT84]) concentrates on two main aspects :

- attributed tree transformations as described here;
- a new form of AGs, named Attribute Coupled Grammars [GG84], in which no distinction is made between syntactic objects (trees) and semantic objects (attributes values).

The last point will be further developed by Harald GANZINGER in Dortmund. As a by-product, the implementation will be carried out in Modula-2, and the possible evaluation strategies will include ASE and the Kenndy-Warren method. This implementation is planned to be completed by the end of 1985. The Munich version of MUG2 also incorporates Attribute Coupled Grammars.

9) References

BG 81, BT 84, Gan 82, GG 84, GGM 82, Gie 79, GMW 80, GRW 77, MWW 84, Rip 75, Wil 78.

NEATS

New Extended Attribute Translation System

Ole Lehrmann MADSEN

Computer Science Dept.

Aarhus University

NY- Munkegade

DK-8000 AARHUS C

(DENMARK)

1) Members of the project

O.L. MADSEN (head), P. JESPERSEN, M. MADSEN, H. RIIS (NEATS), S.H. ERIKSEN, B.B. KRISTENSEN (BOBS).

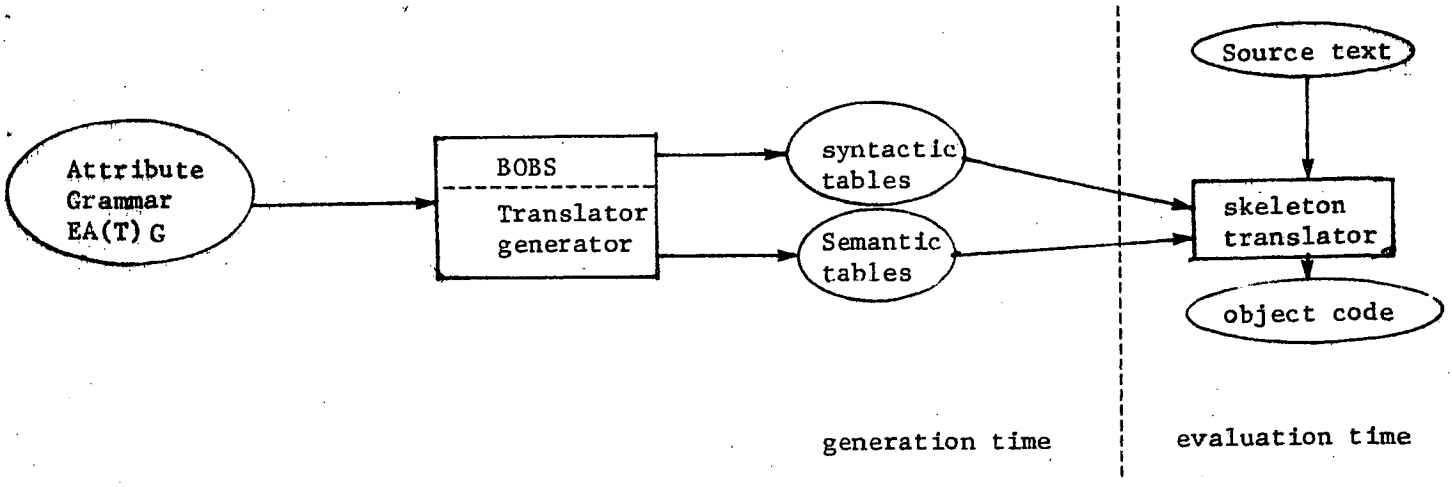
2) Birthdate : 1978 Deadline : ?

3) General Features

Lexical Analsyis :	unspecified (hand-written ?)
Syntactic Analysis :	LALR(1)
Attributes Evaluation :	DAG-evaluator [Mad 80a]
AG Class :	non-circular
AG Language :	EAG or EATG
Code Generation :	EATG

4) Schema

NEATS is written in Pascal and runs on DEC-10.



5) General Comments

The input to NEATS is an Extended Attribute Grammar (EAG) or an Extended Attribute Translation Grammar (EATG). Extended Attribute Grammars[Mad 80, WO 77, WM 79, WM 83] are a mixture of affix grammars [Kos 71] and attribute grammars. Attributes have no name ; they are just slots attached to each non-terminal (and some terminals). Synthesized and inherited attributes are distinguished. The semantic rules and constraints are embedded in the context-free productions, by filling the (attribute) slots with variables and expressions.

At the basic level, input attributes (i.e. the inherited ones of the LHS and the synthesized one of the RHS) are just variables. These variables can be used in the expressions defining output attributes. If all the input attribute variables are distinct, then no constraint is attached to the production. It is allowed to use the same variable for different input attributes, and even to use expressions. This means that the input attributes must have the shape described by those expressions, in a pattern-matching way. Similarly, the same variable for different input attributes means that those attributes must have the same value. These constraints are implicit in an EAG, which is a generative description rather than an analytic one. For an example of an EAG see [Wat 79].

Constraints can be enforced by introducing non-terminals deriving the empty string iff a given predicate is verified. Because of the generative nature of EAGs,

those productions can be derived only if the predicate is true. Those predicate non-terminals are used to describe computations which would otherwise be described by extra-grammatical user functions.

Extended Attribute Translation Grammars (EATGs) are to EAGs what attributed translation grammars [LRS 74] are to normal AGs, or what syntax-directed translation schemes [AU 73] are to CF-grammars. An output grammar is associated to the input grammar ; they have the same non-terminals but may have different terminals. The output grammar is also an EAG. To each input production a corresponding output production is associated. The attributes expressions of the output production can refer to the attributes variables of the input production but not vice-versa. Output terminals can have only inherited attributes, specifying e.g. what value is to be output for the terminal "INTEGER".

Attribute values have a domain (type). Basic domains are booleans, integers, names (strings) and Pascal-like enumerations. Domains can be combined with the following domain constructors : cartesian products (records), discriminated unions (variants), (partial) maps (to represent arrays or tables) with disjoint union and overriding, and sequences. Attribute values having a "map" domain can occur (in the generative sense) only if they are used where defined ; this implements constraints such as mandatory declarations.

An EA(T)G to be input to NEATS is composed of :

- a definition of the attributes domains ;
- a definition of user functions (there stay some !);
- a definition of non-terminals and terminals with their attributes domains ;
- a definition of attributes variables, with their domains ;
- a definition of the set of productions (input productions, and output ones for an EATG) with their semantic equations.

NEATs transforms such a description into an equivalent classical AG. This transformation is possible only if the EAG is well-formed [Mad 80a], i.e. non-circular. The syntactic part is processed by the BOBS LALR(1) parser generator. However predicate

non-terminals do not appear in the actual CF-grammar : they are translated into constraints.

Attributes evaluation is performed by a DAG-evaluator [Mad 80a] : during parsing the parse tree is not constructed, instead the compound dependency graph is built. This graph is then traversed in a depth-first manner, and attributes are evaluated upon return to a node. This depth-first search can also detect dynamically the true circularities. This evaluation strategy works for all non-circular AGs, and even for circular ones if a meaning is given to "circular attributes".

6) Optimizations

In the compound dependency graph, nodes (i.e. attributes instances) which are connected by a copy rule are collapsed. This saves a lot of space. The authors claim that all in one the size of the DAG is proportional to the size of the parse tree.

It is intended to implement a method to let attributes values influence the parsing (rule-splitting [Wat 80]). However this will change completely the evaluation strategy.

7) Applications

Actual applications are not known. The references present some examples of EAGs and EATGs ; in particular [Wat 79] is an EAG to express the static semantics of full Pascal ; [JMR 78] presents an EATG to translate the pL source language into the cL assembly language ; [Mad 80a] studies how to express predicate transformers, denatational semantics and operational semantics by means of EAGs.

We have no information about the sizes and actual performances of the system ; however it is considered as slow because of its important space consumption, even on the DEC-10.

PARSLEY

Milton E. BARBER
Summit Software
LOS GATOS, CA
(U.S.A.)

1) Members of the project

M.E. Barber and probably others.

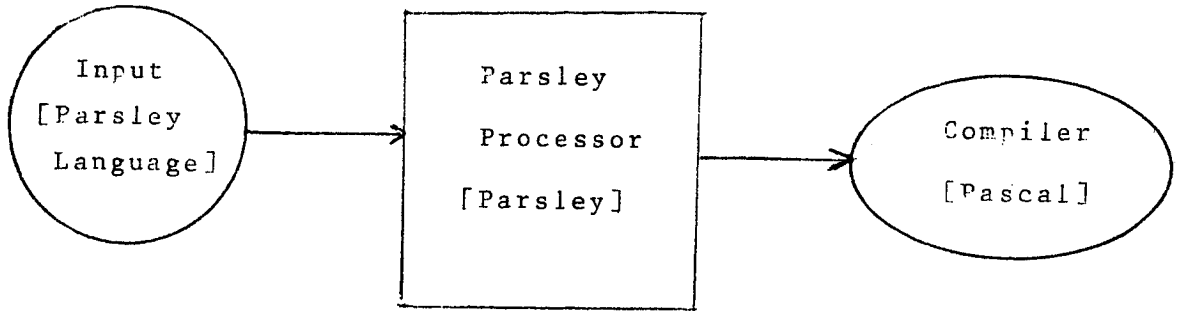
2) Birthdate ? Deadline ?

3) General Features

Lexical Analysis:	type-3 grammars (regular expressions).
Syntactic Analysis:	MD (1) (Mixed Directions Parsing).
Attributes Evaluation:	alternating passes
AG Class	simple multi-pass
AG Language:	Parsley language (strict extension of Pascal).
Code Generation:	no special feature provided.

4) Schema

Parsley is written in standard Pascal and thus runs on a large number of machines.



5) General Comments

Parsley is an attempt to provide a unified framework combining scanner generation, parser generation, attributes evaluation and standard Pascal programming. A summary of Parsley extensions over Pascal is as follows:

- tree structures become a data type; they are described by context-free grammars;
- an extended language is provided to express the computation of certain kinds of functions on trees; this language offers essentially AGs capability;
- a new kind of subprogram, called an "iterator", is provided; like a function, it yields values, however it yields a stream of values rather than a single value;
- a new statement, called "parse", is provided; it takes a stream of elementary objects, constructs the parse tree of that stream according to a specified grammar and then computes a specified set of functions on this tree.

Everything from a simple scanner to a complex multi-pass AG is presented to the user via a single model: the parse statement. The Parsley processor automatically recognizes the fact that, for a scanner, finite-state techniques are sufficient, and automatically determines the number of passes necessary to evaluate more complex AGs. It can also determine whether attributes evaluation can be conducted in parallel with parsing. The parse statement can be used as any Pascal statement in a Parsley program; this enhances the flexibility of the system.

MD parsing is a mixture of top-down (LL(1)) and bottom-up (LALR (1)) parsing, with a preference given to the former. It is based on the partitionning of a grammar into pieces.

Since the system is written in standard ANSI/ISO Pascal, it is highly portable.

6) Optimizations

Unknown.

7) Applications

Unknown, apart from the Parsley processor itself which is bootstrapped.

8) Projects

Unknown.

9) References

Bar 83

SAGET

V.P. MAKAROV
Gomel Division
Institute of Mathematics
Academy of Sciences of Bielorussia
(U.S.S.R.)

1) Members of the project

V.P. MAKAROV, V.G. PESHKOV.

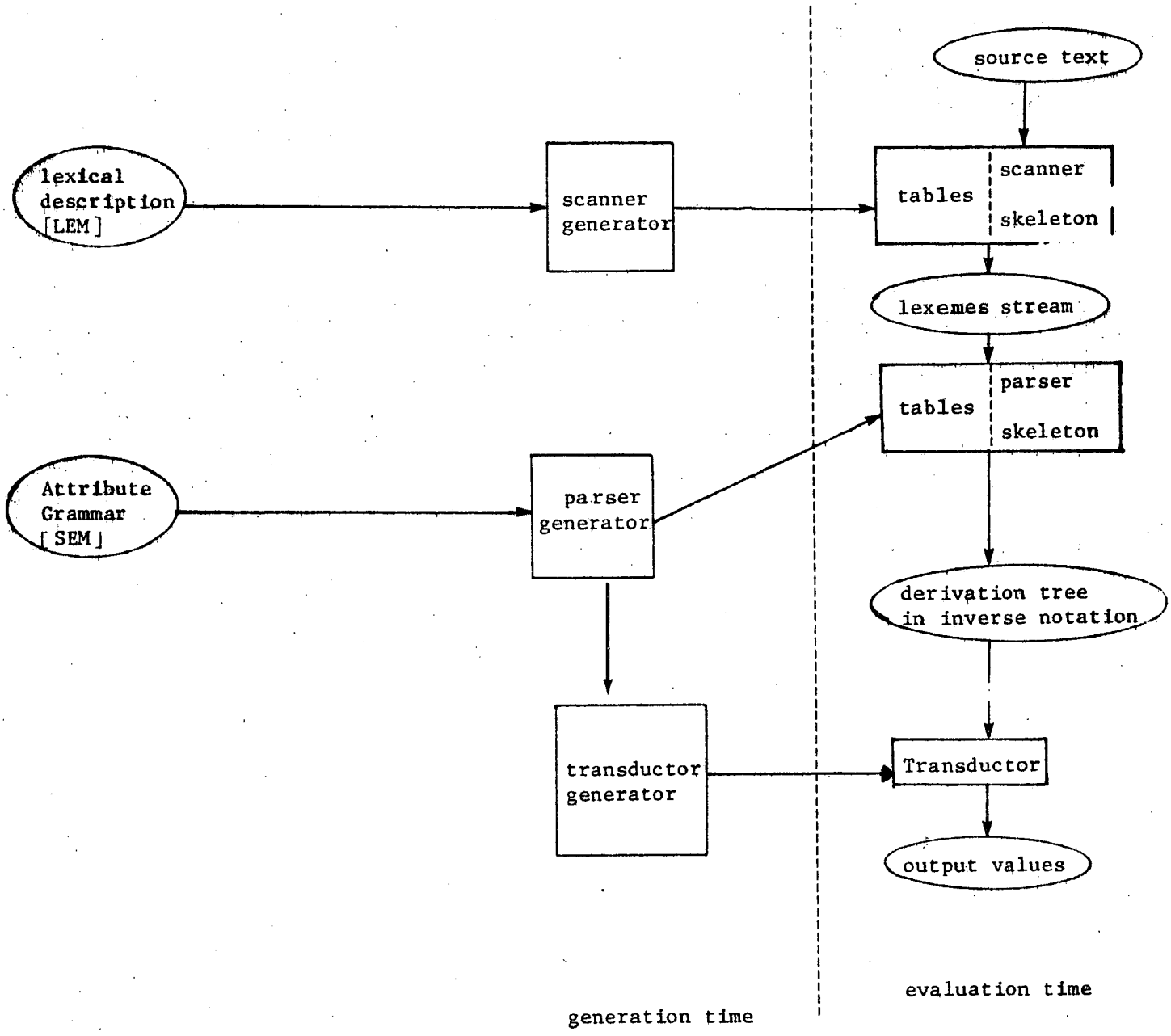
2) Birthdate : 1979 (?) Deadline : ?

3) General features

Lexical Analysis :	automatic generation but otherwise unknown.
Syntactic Analysis :	LR(1) with error recovery
Attributes Evaluation :	during parsing
AG Class :	purely synthesized
AG Language :	SEM
Code Generation :	no special feature provided.

4) Schema

SAGET is written in PL/1, produces compilers in PL/1 and runs on the soviet M4030 computer.



5) General Comments

The basic evaluation algorithm is the classical one used for evaluating synthesized attributes during a bottom-up syntactic analysis. Attributes values are stored on a stack parallel to the parse stack.

The only modification, which transforms such an AG into an attributed quasi-translation grammar, is the use of "working attributes" which act as local variables inside a production and propagate information from left to right. However this information cannot be used inside subtrees ; this is not inherited information.

It is also possible to use global attributes (global variables).

Semantic rules are written embedded in the productions. The metalanguage is called SEM and is based on PL/1. Synthesized and working attributes can have only the types bit-string or integer. Global attributes can have any PL/1 type. User-defined functions and procedures are written in PL/1.

The SAGET system consists of about 3500 PL/1 lines.

6) Optimizations

Unknown.

7) Applications

Pascal (syntactic part only).
We have no information about the performance of SAGET.

8) Projects

Unknown.

9) References

Mak 82, Mak 83, MP 80.

SSAGS

Syntax and Semantics Analysis and Generation System

Teri F. PAYTON
SDC, a Burroughs Company
Research and Development
Box 517, PAOLI, PA 19301
(U.S.A.)

1) Members of the project

Teri F. PAYTON, Steven E. KELLER, John A. PERKINS, S.P. MARDINLY, S. ROWAN

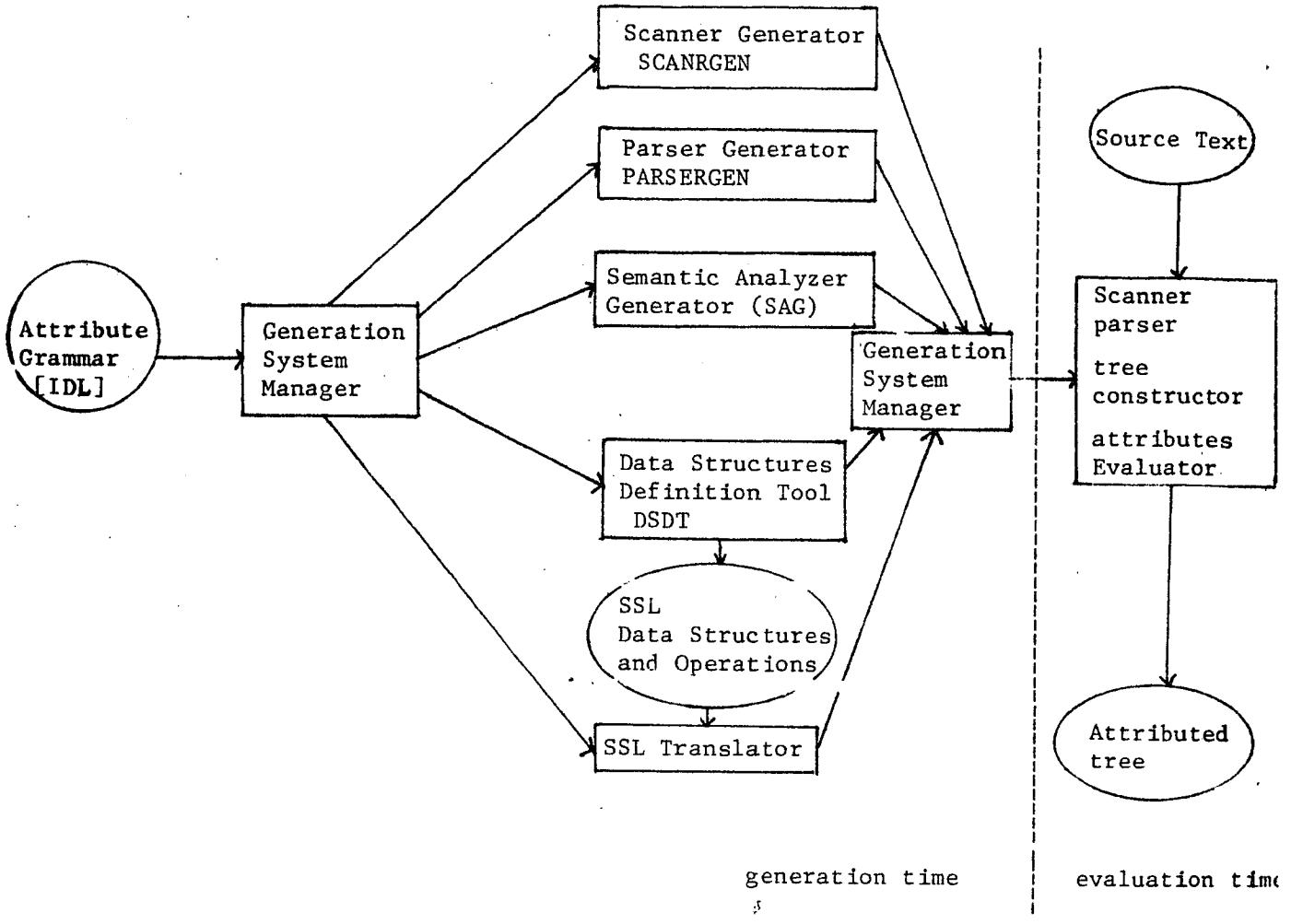
2) Birthdate : ? Deadline : ?

3) General Features

Lexical Analysis :	regular expressions
Syntactic Analysis :	SLR with error recovery
Attributes Evaluation :	Visit-sequences
AG Class :	OAG
AG Language :	IDL (Interface Description Language)
Code Generation :	No special feature provided.

4) Schema

SSAGS runs on Burroughs B6900 (in Algol 68) and soon on VAX/UNIX (in C).



5) General Comments

SSAGS was designed to be not only a pure attributes evaluator but mainly a program generation tool, applicable to the development of a wide range of software products.

Regarding the core of the system, SAG, which is the attributes evaluator generator, not much can be said : SAG uses well-known methods [Kas 80]. The OAG evaluation is performed by a set of recursive procedures. The output is the attributed tree.

The most interesting feature of SSAGS (and the most unusual with respect to this survey) is the data abstraction facility provided by the DSDT. The structure of the output attributed tree can be described by a high level specification (in IDL),

and packages which implement those data structures and the operations to manipulate them are produced automatically. Those packages (and in fact the whole generated front-end) are written is SSL (Semantics Specification Language) which is a subset of Ada "powerful enough to specify the semantics of a front-end, yet simple enough to facilitate easy translation to a variety of implementation languages such as Algol, Pascal or C " [PKP82a]. This later translation is the task of the SSLT, which currently produces Algol 68 (on the Burroughs B 6900) and C (on VAX/UNIX).

The other unusual feature of SSAGS is the use of AGs to implement tree-transformation grammars (TT-grammars) [KPP 84]. A TT-grammar specifies the mapping of trees of an input grammar to trees of an output grammar (and vice-versa in some cases). A number of different classes of TT-grammars are defined. SSAGS automatically translates a TT-grammar into an AG performing the translation, and generates the corresponding evaluator. The authors claim that this new formalism is much more pleasant than AGs to specify tree-to-tree transformations such as performed in a compiler front-end (from input language parse tree to an intermediate form). The loss of generality (TT-grammars do not have the full power of AGs) is balanced by the possibility to prove the "correctness" of the transformation.

6) Optimizations

None, as far as we know.

SSAGS produces an attribute dependencies cross-referencer and traces in order to help the grammar writer "debugging" OAGs' cycles.

7) Applications

. SSAGS itself : the system is bootstrapped.
. A FORTRAN format encoder.
. An Ada front-end (under progress).
. Ada to DIANA trees and DIANA trees to C trees.

8) Projects

Improvements of tree transformation techniques.

SUPER

V.M. KUROCHKIN
Computing Center of The
Academy of Sciences of the USSR
Varilova 40
117333 MOSCOW
(U.S.S.R.)

1) Members of the project

V.M. KUROCHKIN, A.N. BIRJUKOV, V.A. SEREBRYAKOV.

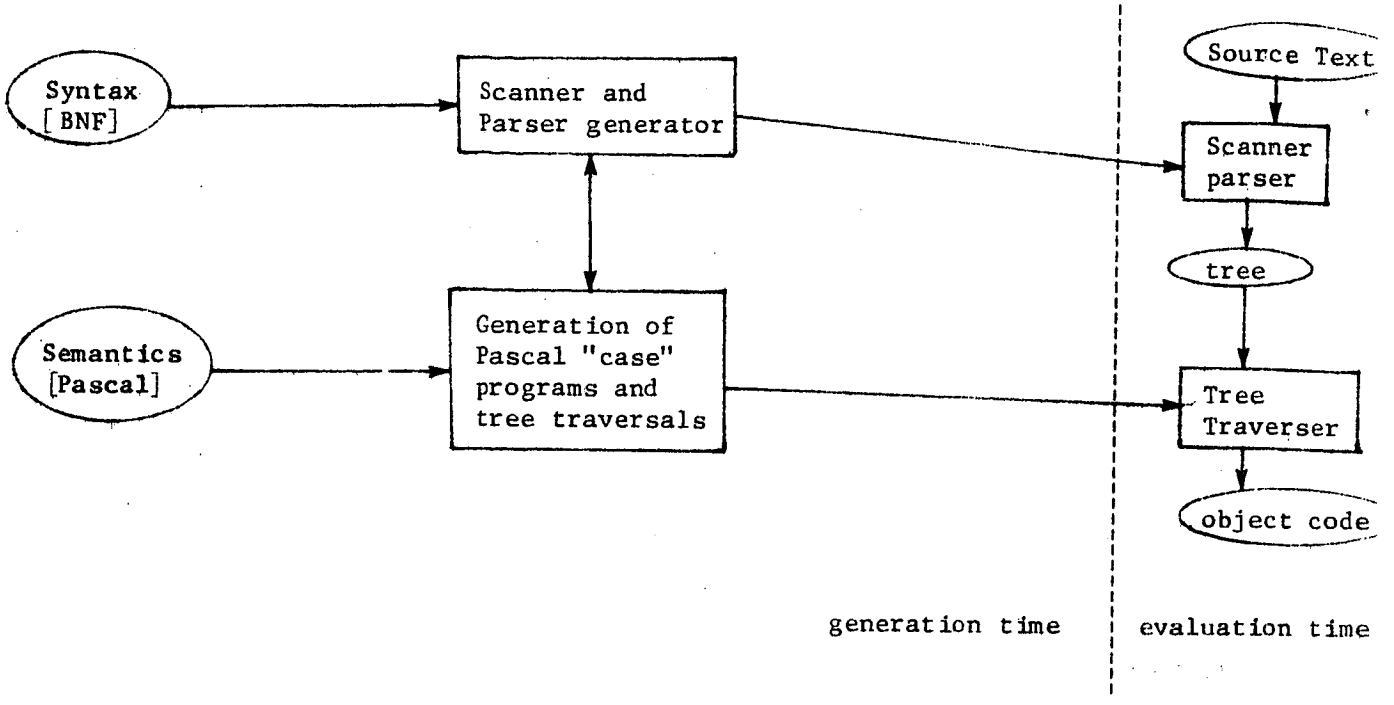
2) Birthdate : 1978 Deadline : ?

3) General Features

Lexical Analysis :	?
Syntactic Analsyis :	LR(1), LL(1)
Attributes Evaluation :	Left-to-right passes
AG Class :	1) Non-circular 2) One-visit
AG Language :	based on Pascal
Code Generation :	No special feature provided

4) Schema

The system is written in Pascal and generates compilers written in Pascal.



5) General Comments

SUPER distinguishes global attributes, corresponding to large data structures (arrays, records, files,...), and other attributes.

An algorithm, linear in the size of the source text, organizes the informations so that attributes evaluation can be performed in a sequence of top-down left-to-right tree traversals. This tree-walk is computed by a coding of the parse tree during syntactic analysis.

SUPER comes in two versions, a general one for any non-circular AG, another limited to one-pass (1 visit) AGs.

6) Optimizations

As soon as an attribute instance becomes useless, it is deleted and its storage reclaimed.

7) Applications

. ASPLE

. SUPER itself (the semantic generation part), representing about 3000 Pascal lines.

We have no information about the performance of the system.

8) Projects

Unknown.

9) References

BKS 79a, BKS 79b, BKS 80, BKS 81.

The Synthesizer Generator
Thomas REPS, Tim TEITELBAUM
Dept. of Computer Science
Upson Hall
Cornell University
ITHACA, NY 14853
(U.S.A.)

1) Members of the project

Thomas REPS, Tim TEITELBAUM, Alan DEMERS , Susan HORWITZ, M. FINGERHUT, A. ZARING, K. MUGHAL, D. MOITRA, T.K. SRIKANTH.

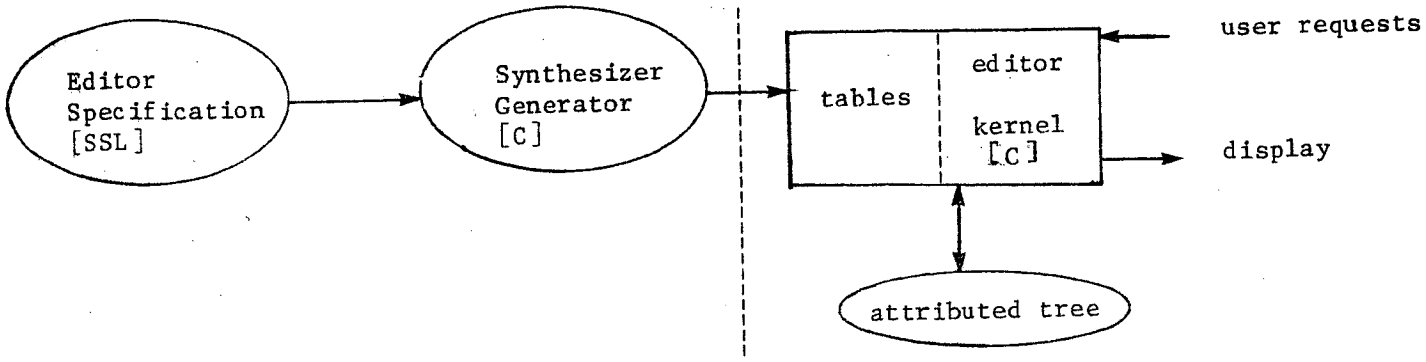
2) Birthdate : January 1980 Deadline : unspecified

3) General Features

Lexical Analysis :	regular expressions (LEX?)
Syntactic Analysis :	LALR(1) with disambiguation rules and precedence rules (YACC) or structural top-down derivation
Attributes Evaluation :	Incremental
AG Class :	1) Non-circular 2) OAGs
AG Language :	SSL (Synthesizer Specification Language)
Code Generation :	none.

4) Schema

The system is written in C and runs on VAX/UNIX.



5) General Comments

The Synthesizer Generator is a tool for specifying how objects may be edited in the presence of context-sensitive relationships. The editor designer prepares a specification which includes rules defining a language's abstract syntax, context-sensitive relationships, display format and concrete input syntax. From this specification the Generator creates a full-screen editor for manipulating objects (texts) according to these rules.

The Synthesizer Generator is particularly well suited for creating editors that enforce the syntax and static semantics of a particular language. Each object to be edited is represented as a consistently attributed (abstract) derivation tree. When these objects are modified some of the attributes may no longer have consistent values ; incremental analysis is performed to update attribute values throughout the tree in response to modifications. If an editing operation modifies an object in such a way that context-dependent constraints are violated, the attributes that indicate satisfaction of these constraints will receive new values, and can be used to annotate the display in order to provide the user with feedback about the existence of errors.

Editor specifications are written in the Synthesizer Specification Language (SSL), which is based on the concepts of a term algebra and an attribute grammar, although certain features are tailored to the application domain of language-based editors. SSL is described in [RT 84a, RT 84b] and summarized below.

The Synthesizer Generator has two components :

- a translator that takes an SSL specification as input, and produces grammar tables ;
- an editor kernel that consists of an attributed-tree data type [DRT 81, Rep 82] and a driver for interactively manipulating attributed trees ; the kernel takes input from the keyboard and executes appropriate operations on the current tree.

A "shell" program handles the details of invoking the translator and producing a language-based editor from the resulting tables.

A number of original ideas implemented in the system are discussed below.

a) Optimal-time incremental attributes updating

[Rep 82a, Rep 82b, RTD 83, Rep 84]

Each editor represents a program as an attributed tree, and programs are modified by tree operations such as prunning, grafting, deriving and parsing. A derivation tree modification directly affects the values of the attributes of the modification point, and maybe other values ; incremental analysis is performed by updating attributes values throughout the tree in response to modifications. After each modification to a program tree, only a subset of attributes instances, denoted by AFFECTED, is determined as a result of the updating process itself. The editor kernel of the Synthesizer Generator uses algorithms which require $O(|\text{AFFECTED}|)$ steps, where the application of a semantic rule is counted as an atomic step. Since $|\text{AFFECTED}|$ is the minimal amount of work required to update a tree after a modification, these algorithms are asymptotically optimal.

In each generated editor, one of two versions of the attribute updating algorithm is used : either one which works for arbitrary non-circular AGs or a much more efficient one which works for ordered AGs. The choice is done at generation time : if the grammar passes the OAG test, the editor is created with a version of the editor kernel which has the more efficient change propagation routine. In addition, this kernel uses a much more compact representation of tree nodes that leaves out some of the information needed by the more general algorithm.

These attribute updating methods also have applications in optimizing compilers for updating data flow information after an optimization has been applied (see the transformations in the MUG2 system).

b) Sublinear attributes evaluation

[Rep 82b, Rep 84, RD 85] A method for reducing the amount of primary storage an editor uses has been implemented in the Synthesizer Generator. It allows to make use of spill files in secondary storage. Syntactic subtrees outside the current focus of attention are linearized in a preorder representation and paged to secondary storage. The semantic effect of the subtree, viewed as a function from its inherited attributes to its synthesized attributes, is compiled into a linear code string that evaluates the function. The compilation method ensures that when the function is evaluated, only a sublinear number of temporary values will be retained at any one time. The compilation method is based on an algorithm for evaluating a distinguished attribute of an n-attribute tree that saves at most $O(\sqrt{n})$ attribute values at any stage of the evaluation.

c) The Synthesizer Specification Language

The core of an editor specification is the definition of an abstract syntax, specified as a collection of phyla and operators. An operator is a uniquely-named, possibly 0-ary, cartesian product of phyla. A phylum is a non-empty set of operators. A rule of the form :

$$phy_o : op\ (phy_1, phy_2, \ldots, phy_k)\ ;$$

declares the membership in "phy_o" of a k-ary operator "op" with arguments "phy_1", "phy_2",..., "phy_k", and is analogous to the context-free production :

$$phy_o \rightarrow phy_1\ phy_2 \ldots phy_k$$

with some differences.

Predefined primitive phyla include INT, CHAR, STR, FLOAT, DOUBLE and BOOL. For instance the INT phylum contains the nullary operators (constants) 0,1,-1,2,-2,...

The first declared operator of a phylum is termed the completing operator and is used, by default, at unexpanded occurrences of that phylum in the derivation tree (so it should be 0-ary).

Attributes and semantic rules are attached respectively to phyla and operators. The basic syntax of the corresponding attribute grammar borrows much from C and YACC input language. It is extended by the possibility to use local attributes in a "production", which permit defining a computation in one operator of a phylum withou imposing that definitions be provided in every production of the phylum.

In SSL, an attribute's type can be one of the built-in primitive types, or it can be a user-defined composite type. The same sort of rules are used precisely to define composite types as to define abstract syntax. Thus, the abstract syntax tree being edited and the attributes attached to it are all elements in the universal domai of terms.

SSL incorporates a notation that permits the specifications of separate aspects of a language to be placed in separate portions of a specification : AGs and declarations can be split in separate parts. It is allowed to :

- add a new attribute to an existing phylum ;
- add new semantic rules to existing operators, and
- add new operators and their semantics to an existing phylum.

The display of an object is defined by an unparsing scheme given for each production and consisting of a sequence of strings, names of attributes occurrences and names of right-side non-terminals (phyla). The display is generated by a left-to-right traversal of the tree that interprets these schemes. Formatting is defined by control characters that can be included in the string of an unparsing scheme. The same formalism is used to display composite attributes values. Local attributes such as error messages can be embedded in the display.

To specify the input interface, productions are given for a concrete input syntax (a la YACC), along with semantic equations that define a translation to abstract syntax. This translation can be context-dependent by allowing inherited attributes to be propagated from the current cursor position in the abstract tree into the tree being parsed. This allows to define e.g. macro-commands. The mechanism to translate input text to an abstract syntax tree provides the editor designer with the ability to define textual and structural interfaces in whatever balance is desired.

Because of the uniform treatment of syntactic and semantic domains, it is possible to permit attribute values to refer to and perform calculations on syntactic components. A second consequence of that uniform treatment is that denotable (attributes) values are themselves attributable tree-structured objects. The expression language of SSL permits forcing the attribution of a (previously unattributed) structure with "attribution expressions" of the form :

expression {equations}. attribute.

The value of such an expression is computed as follows :

- the expression is evaluated, yielding some attributable (but as yet unattributed) object S ;
- the inherited attributes of S are initialized by the given equations ;
- the value of S. attribute is computed by demand and returned.

6) Optimizations

The optimizations of the incremental evaluation algorithm are numerous and described in [Rep 82b, Rep 84]. A data structure (sharable 2-3 trees) allowing efficient operations such as insertion, deletion, retrieval, etc., while keeping storage usage low is also discussed.

7) Applications

The origin of the Synthesizer Generator is the Cornell Program Synthesizer (thus the name) which is a structural editor with static semantics checking, incremental compilation and execution capabilities. The Synthesizer Generator was designed to

be able to construct easily Program Synthesizers for other languages.

Applications are :

- a Pascal editor with full static-semantics checking ;
- an editor for partial-correctness programs proofs in Hoare-style logic [RA 84] ;
- a full-screen desk calculator ;
- a lambda-calculus editor with interpreter ;
- a text formatter ;
- a mathematical equations formatter ;
- incremental code generators of two varieties :
  - using Sethi-Ullmann register allocation, and
  - using continuation semantics ;
- a program editor that uses data-flow analysis to detect program anomalies such as uninitialized variables, computation of a value that is never used, conditions that are always true or always false, division by zero and unreachable code.

8) Projects

The Synthesizer Generator will be extended to allow the specifications of global relations whose values are to be maintained automatically as a function of the state of a program being edited. A relational query language available in generated editors will then provide a powerful tool for computing program properties.

The authors expect to distribute the Synthesizer Generator by late'84.

9) References

On the Synthesizer Generator itself : Rep 81, RT 84a, RT 84b, TR 80, TR 81, TRH 81.

On the incremental evaluation algorithms : DRT 81, Rep 82a, Rep 82b, Rep 83, Rep 84, RTD 83.

Tallin's System

Jaan PENJAM

Institute of Cybernetics

Academy of Sciences of Estonia

Academia Tee 21

TALLIN

(U.S.S.R.)

1) Members of the project

Enn TYUGU (head), Jaan PENJAM.

2) Birthdate : 1980 Deadline : ?

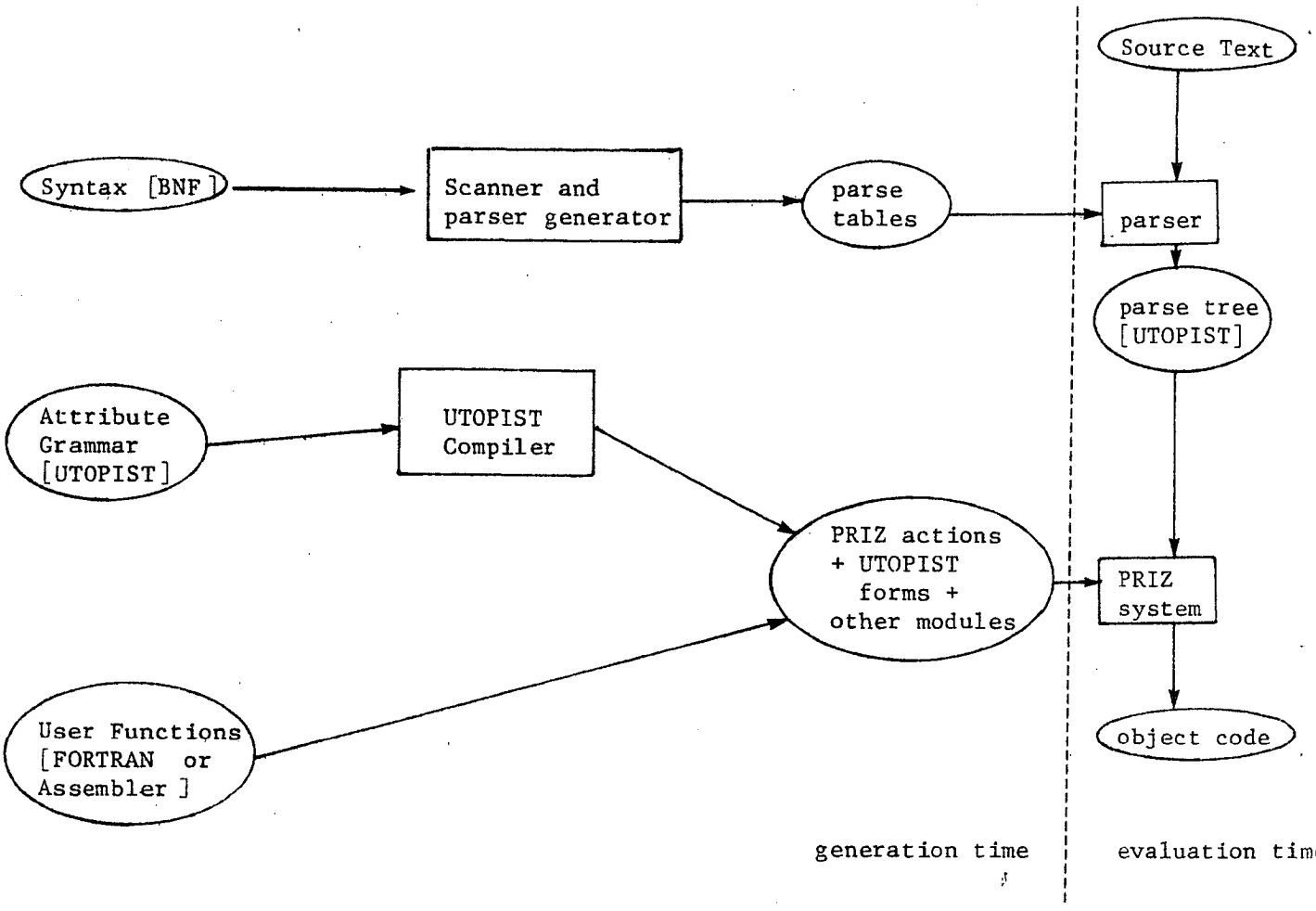
3) General Features

Lexical Analysis :	?
Syntactic Analysis :	precedence
Attributes Evaluation :	1) equation solver 2) recursive evaluation
AG Class :	1) Non-circular 2) SNC
AG Language :	UTOPIST
Code Generation :	No special feature provided.

4) Schema

The system is written in FORTRAN. A reduced version runs on APPLE II.

The following schema is applicable to the first version of the system, the one using the equation solver.



5) General comments

This (unnamed) system uses the WIRTH parser generator developed at Tartu University (USSR). Any other parser could be used, provided that it produces a tree in UTOPIST form, maybe using a linear transcoder.

The UTOPIST language and the PRIZ system were developped by E. TYUGU at Tallin University to implement automatic program synthesis systems. PRIZ is a "goal oriented problem solver". A particular problem can be considered as a relation over

typed variables, such that if some are known (input variables), some others can be computed (output variables). Such a problem, i.e. the computation of some output variables, when input variables are given, can be solved by decomposing the problem into subproblems. The UTOPIST language is interpreted by the PRIZ system which computes the transitive closure of the relations corresponding to a given problem, organizes the information flow and executes the computations necessary to solve the problem. In fact, the whole is close to a PROLOG system.

Thus, to each production in the AG is associated a relation over the attributes occurrences. The semantic rules are written in UTOPIST, using predefined and/or user operators, the latter being written in FORTRAN or assembler. The parse tree is itself a relation (called TREE) over nodes instances, labelled by non-terminals. Attributes instances are dynamically associated to those nodes instances because the non-terminals are typed by the associated attributes themselves (the declarations are written in UTOPIST). Then the PRIZ system has only to solve the computation of the output variables of TREE, which correspond to the synthesized attributes of the root of the parse tree. For more details see [Pen 80a].

The efficiency of the whole system is strongly dependent on the one of the PRIZ system. The strategy used here is similar to the one of DELTA (q.v.), since the computation of the information flow is separated from the variables evaluation proper.

This system is presented by E. TYUGU [Tyu 80] as an example of automatic program synthesis. This is both an originality and a drawback. The originality is to evaluate the attributes using a problem solver designed for other purposes. The drawback seems to be using a general purpose system to solve a problem which can be solved more efficiently with more specific methods.

The system comes in two versions : a dynamic one, described above, and a static one, more efficient, based on the FNC method (see the FNC/ERN system) and using the same kind of input, written in UTOPIST. Since this method avoids the dynamic ordering of the flow, the PRIZ system is used only to deal with user subprograms calls and parameter passing. The SNC test is performed at generation time.

6) Optimizations

Unknown.

7) Applications

The lack of practical efficiency seems to restrict the usage of the system to language design. Only small languages (e.g. MILAN) have actually been tested.

8) Projects

Unknown.

9) References

Pen 79a, Pen 80a, Pen 83a, Pen 83b, Tyu 77, Tyu 80.

Tokyo's System

Masataka SASSA
Dept. of Information Sciences
Tokyo Institute of Technology
Ookayama
Meguro-ku
TOKYO 152
(JAPAN)

1) Members of the project

Masataka SASSA, Junko TOKUDA, Tsuyoshi SHINOGI, Kenzo INOUE, S. VEHARA, H. TAZAKI, S. NAKAMURA, H. YOSHIDA.

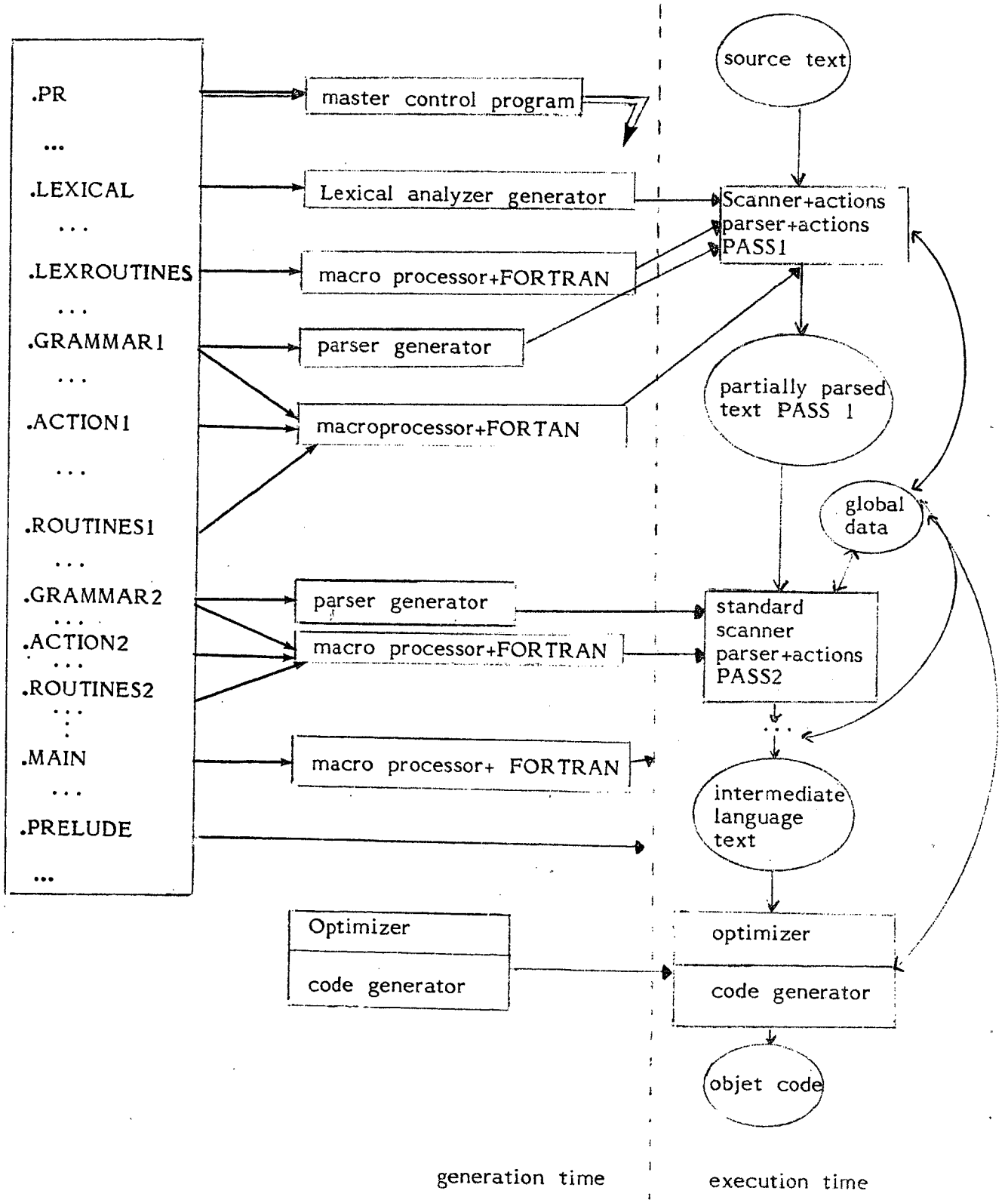
2) Birthdate : ? Deadline : ?

3) General Features

Lexical Analysis	:	regular expressions + user actions
Syntactic Analysis	:	multipass partial grammar SLR(1) or LALR(1) parsing.
Attributes Evaluation	:	in parallel with parsing + actions called at each reduction
AG Class	:	purely synthesized
AG Language	:	Algol-like (procedural)
Code Generation	:	standard intermediate language and optimizer and code generator (not part of the system).

4) Schema

Tokyo's system is written in FORTRAN and produces compilers written in FORTRAN.



5) General Comments

Tokyo's system aims at solving efficiently the problems caused by counter-one-pass languages features, such as identification of targets of a goto statement, identification of variables in procedure bodies (Algol 60 and 68) and redefinable operators precedence (Algol 68). This last problem is especially difficult because it influences the correct parsing of arithmetic expressions.

The solution to these problems, which has been implemented in Tokyo's system, is multipass partial grammar parsing. The idea is to make several passes on the source text, each pass parsing and analyzing only small portions of the text. For instance, in an Algol 68 compiler, the first pass would analyze mode and operator declarations, constructing a mode table and an operator table; the second pass would analyze variables and procedures declarations, constructing a symbol table; the third pass would analyze the rest of the text, performing (in a single pass) identification, type checking and translation into an intermediate language.

Another design goal of Tokyo's system is the efficiency of the generated compilers. To achieve it, the parsers do not construct parse trees. Rather, semantic analysis is carried out in parallel with parsing. The intermediate form produced by each pass and input to the next is composed of the partially parsed intermediate text and of global information computed by the semantic analysis and stored in tables.

The other design goals are :

* complete, readable, easily modifiable compiler description;
* efficient and usable compiler generator which can flexibly and partially regenerate a compiler whenever part of the description is changed;
* machine-independence considerations.

a) The lexical analyzer generator

The lexical analyzer specified by the user is responsible of scanning in one pass the whole source text; a standard scanner, "scanning" only an internal form of the intermediate texts, is supplied for the other passes.

The lexical description is composed of a set of character classes, a set of tables (for keywords, etc.), and a set of regular expressions describing the tokens. A user-defined procedure is called after the recognition of each regular expession. The return value of this procedure may be used to choose the terminal symbol to be passed to the parser.

b) Multipass partial grammar parsing

Basically it works as follows : the parser of each pass reads input text of the pass and copies it into output text of the pass. When it catches the starting position of the partial grammar to analyze in this pass, it enters the parsing mode, analyzes (a part of) the input text, and outputs the corresponding, goal symbol. The i-th pass parser is given a partial grammar G_i which is a subgrammar of the whole grammar G. The parsing is made only on partial portions of the i-th text which correspond to sentences in $L(G_i)$. The goal symbol S_i of the partial grammar G_i is output to the (i+1)-th text and will be treated by the (i+1)-th parser as a terminal symbol. The rest of the i-th text is merely copied to the (i+1)-th (in internal form).

In order to catch the starting position(s) for i-th pass parsing, two sets must be given : $PREC_i$, the set of terminal symbols preceding sentences of $L(G_i)$ in the text, and $FIRST_1^{G_i}(S_i)$, the set of first terminal symbols of sentences in $L(G_i)$. While the later can be computed automatically, the former cannot because it cannot be derived from G_i alone; thus it should be specified explicitly in the description of G_i. The i-th pass parsing is triggered whenever a symbol $a \in PREC_i$ is followed by a symbol $b \in FIRST_1^{G_i}(S_i)$ in the i-th text. Similarly, to determine the terminating position of a partial grammar parsing, the description of G_i must explicitly specify $SUCC_i$, the set of terminal symbols following $L(G_i)$.

Other features to specify exactly multipass partial grammar parsing include :

- extra goal symbols specification, when the combination of $PREC_i$ and $FIRST_1^{G_i}(S_i)$ is insuffficient to discriminate exactly what is needed;

- a mechanism to save and restore parser states, e.g. to skip initialization parts in variables declarations (Algol, C);

- a mechanism to replace input terminal symbols when information gained in the preceeding passes allows such discrimination, e.g. an identifier denoting a function may be replaced by the symbol "function identifier" after the identification pass;

- a mechanism to catch "range structures" such as begin... end, extended to catch e.g. if... then;

- a specification of the syntax of label definitions.

c) Description and evaluation of semantics

Semantics is described by a modified attribute grammar : symbols (non-terminals and terminals) may have only synthesized attributes which are implicitly transfered from bottom to top. This transfer may be overriden by user-specified actions, written after the production proper. These actions (semantic rules) are written in a procedure-oriented language which is macro-translated into FORTRAN. This language allows complex operations to be specified concisely, e.g. "ENV +:= DECS" adds a whole set of declarations DECS to an existing environment ENV. Semantic attributes are implemented by semantic stacks elements, as usual with bottom-up parsing. Global entities may also be used to replace inherited attributes.

d) Generator Organization

The generator is composed of a number of processors (see schema) so that (re)generation can be incrementally applied to each component of the input description.

e) The generated compilers

The generated compilers are also composed of a number of modules (see schema) which can be overlaid in a natural way.

The authors have designed a machine-independent intermediate language named IL. If the grammar writer chooses to use this intermediate language, he may use directly the IL optimizer and code generator. Presently the latter generates code for only the japanese machine FACOM 230-455, but it was written with special care for generality and portability, and thus should prove easy to retarget.

6) Optimizations

None required.

7) Applications and performances

* Toy languages
* Pascal compiler
* compiler for a subset of Algol
* compiler for a subet of Ada

We have no information about the size and performances of the system. As for the generated compilers, comparisons have been done between a two-pass parser and a one-pass parser; the former runs with CPU time increased by 5% w.r.t. the latter, but space is reduced by 20% using overlay.

Experience shows that multipass parsing simplifies very much the description of counter-one-pass language features.

8 Projects

Unknown.

9 References

STS79, STS80.

VATS
Visible Attributed Translation System

P.G. SORENSON
Dept. of Computational Science
University of Saskatchewan
SASKATOON, SASKATCHEWAN S7N 0WO
(CANADA)

1) Members of the Project

A.Berg, D.A.Bocking, D.R.Peachery, P.G.Sorenson, J.P.Tremblay J.A.Wald.

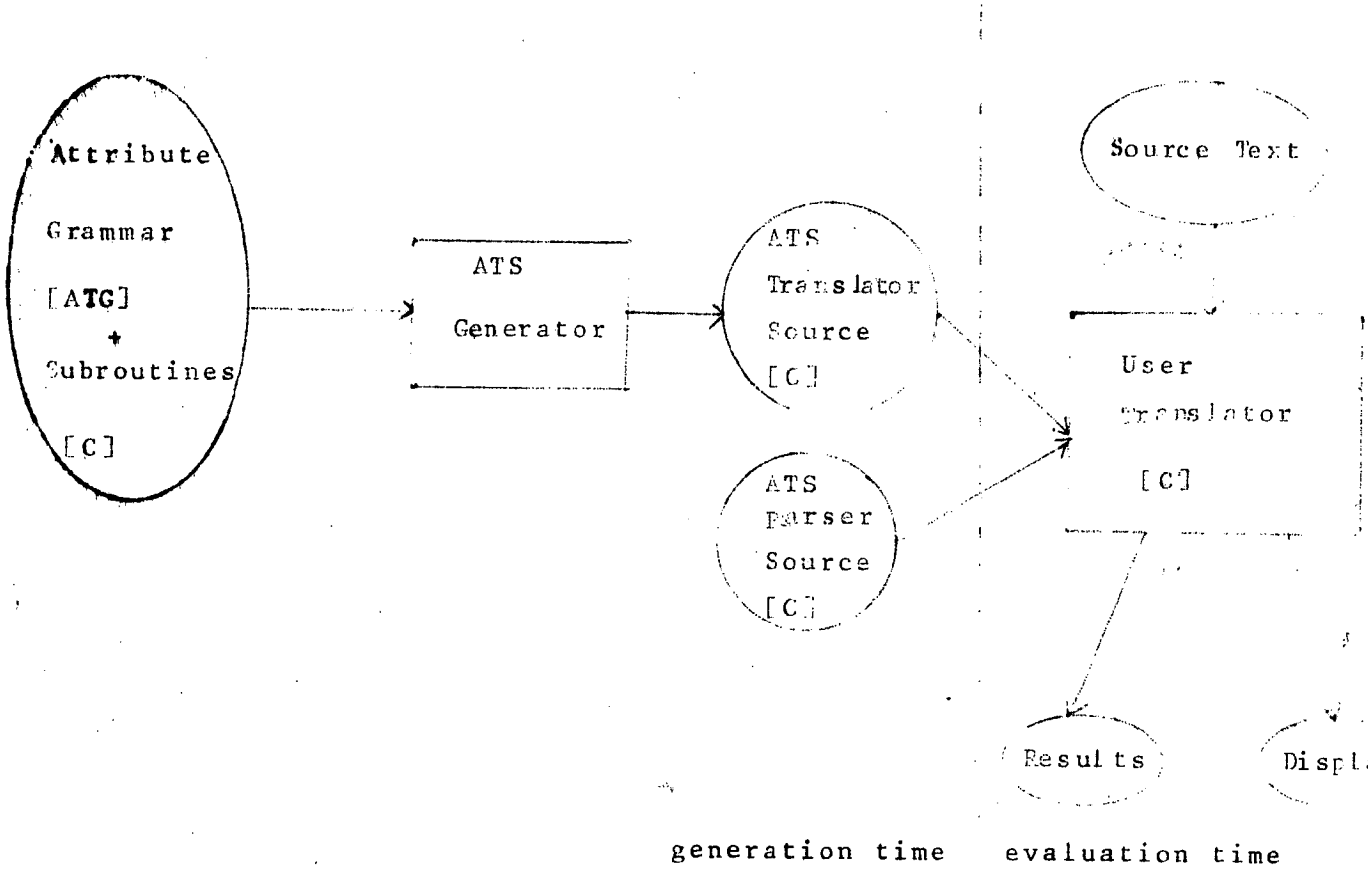
2) Birthdate:? Deadline:?

3) General Features

Lexical Analysis:	hand-written scanner, or generated by Lex .
Syntactic Analysis:	LL(1) with error recovery
Attributes Evaluation:	in parallel with parsing
AG Class:	L-ATGs (Attributed Translation Grammars [LRS74])
AG Language:	ATG + C
Code Generation:	No special feature provided.

4) Schema

VATS is written in C and runs on a number of small machines under UNIX (VAX-11, Zilog ZEUS, DECSystem 2060, Perkin-Elmer 3200 serie, SUN-2) or PC-DOS.



5) General Comments

The input to VATS is an Attributed Translation Grammar (ATG) [LRS74].

A Translation Grammar is a context-free grammar in which RHS of productions are composed of terminals, non-terminals and action symbols which can be seen as calls to user-defined procedures. These calls are performed when the preceding terminals and/or non-terminals are recognized, which implies that the parsing method must be predictive (descendent). An ATG is a Translation Grammar where symbols (terminals, non-terminals and action symbols) have attributes. The notation used in VATS for a symbol is thus:

symbol<i1,i2>s1,s2

where i1 and i2 are the inherited attributes of "symbol" and s1 and s2 are its synthesized attributes. Attributes have no names: they are rather positional slots in which you write attributes variables (identifiers). The semantic rules are written separately using those attributes variables.

ATGs accepted by VATS are restricted to be L-ATGs in simple assignment form. An L-ATG is an ATG whose attributes can be evaluated in a single left to right pass. An L-ATG is in simple assignment form when all the semantic rules are copy rules. This is achieved by introducing action symbols where non-trivial semantic rules are evaluated. When an ATG is in simple assignment form, the explicit assignments (copy rules) can be made implicit by renaming (in the productions) the attributes occurring on the LHS of an assignment with the name of the attribute occurring in the RHS of the assignment. An ATG to be input to VATS must be an L-ATG in simple assignment form with implicit assignments.

An input file to be input to VATS must contain:

- an ATG as previously defined;
- for each action symbol @X, the definition of the corresponding procedure _X; such procedures are parameterless, the values of inherited attributes are retrieved from the stack one at a time by calling the procedure llinh, and synthesized attributes are pushed back on the stack one at a time by calling the procedure llsyn;

- a number of required procedures (main, llscan,...);
- other auxilliary routines.

The ATS generator converts this file into a C program named the ATS translator source; in particular the ATG is replaced by a set of parsing tables. This translator source is then compiled and linked with the (fixed) ATS parser source to form the executable user translator.

When a syntactic error is encountered, a local correction of the source text is attempted, by successively inserting a new symbol, deleting the offending symbol, and replacing it with a new one. For each of these corrections, a trial parse (a parse limited to the next few symbols) is performed (ignoring the action symbols) and if it is successful, the correction is validated and parsing resumes with the new text. If none of the corrections is valid, the offending symbol is deleted and the parser is restarted; however, ultimately, recovery will probably fail and the parser will give up.

Since VATS has been heavily used for educational purposes, a special feature has been added to make the parser operation "visible". In this mode the display screen is divided in a number of dedicated zones:

- the input zone, where the current input line is displayed, with the most recently scanned token marked by a cursor;
- the parser area which provides a narrative descriptions of the actions of the parser ;
- the recovery area which describes the actions taken by the syntax error recovery algorithm;
- the stack zone displaying the content of the primary parse stack (terminals, non-terminals and actions).

As the source text is analyzed, messages are emitted in the relevant zones, so that the configuration of the parser is maintained up to date. This feature proved very useful for teaching purposes.

6) Optimizations

None.

7) Applications and performances

* Teaching purposes.
* A data base query language
* A number of systems related to distributed applications (PANEL), form design (PICDRAW), data bases (SPSL/SPSA), and end users facilities (PICASSO).

No information is available about the size and performance of the system, but the authors claim VATS has "substantial capabilities", even on microcomputers.

8) Projects

* Global attributes
* automatic generation of error recovery tables
* enhancements to the "visible" operating mode
* acceptation of ambiguous grammars.

9) References

Since this system was made known to us after the publication of the bibliography, the next reference is not included in it:

[BBP84] A.Berg, D.A.Bocking, D.R.Peachey, P.G.Sorenson, J.P. Tremblay, J.A.Wald: "VATS- the Visible Attributed Translation System", report 84-19, Dept of Computational Science, University of Saskatchewan, Saskatoon. To appear elsewhere.

YACC
Yet Another Compiler-Compiler

Stephen C. JOHNSON
AT&T Bell Laboratories
600 Mountain Avenue
MURRAY HILL, NJ07974
(U.S.A)

1) Members of the Project

Stephen C. JOHNSON and others

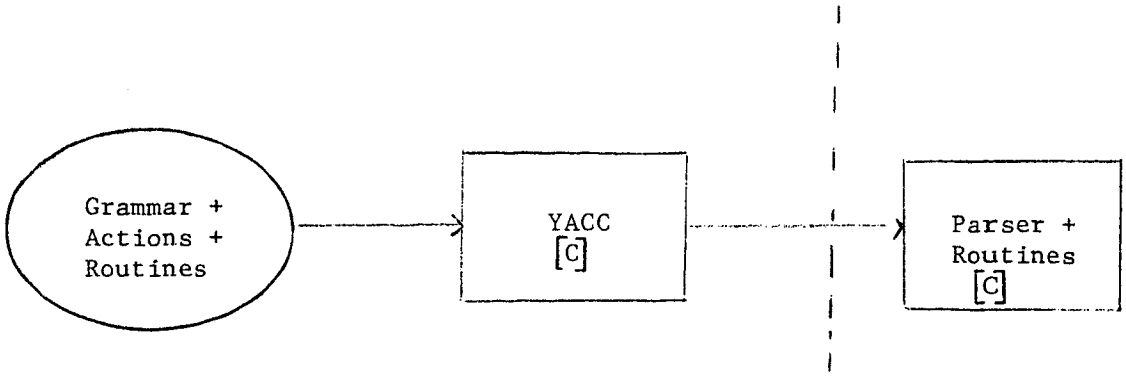
2) Birthdate : 1973(?) Deadline : 1978

3) General Features

Lexical Analysis	: hand-written or generated by LEX (regular expressions)
Syntactic Analysis	: LALR(1) with disambiguating rules and some error recovery
Attributes Evaluation	: in parallel with parsing
AG Class	: purely synthesized
AG Language	: C
Code Generation	: no special feature provided

4) Schema

YACC is written in C and produces "compilers" written in C. YACC is a standard compronent of any UNIX environment.



5) General Comments

Although YACC was not designed with the attribute grammar method in mind, it has a mechanism which empowers it with the same capabilities as other systems described in this survey (S-DELTA, SAGET, CWS1). Moreover its widespread usage makes YACC a strong reference point to which other systems can be compared. It was thus impossible not to include YACC in this survey.

YACC is basically an LALR(1) parser generator. The productions are written in a dialect of BNF. Terminal tokens names must be declared; one-character tokens may also appear literally in the rules. The other names appearing in the rules are assumed to be non-terminals.

YACC has a mechanism which allows the user to control the resolution of shift/reduce or reduce/reduce conflicts when building the parser : a shift/reduce conflict is resolved by favoring the shift, and a reduce/reduce one is resolved in favor of the (textually) earlier production. The user may also associate a precedence level and an associativity kind with some tokens, which facilitate very much the description of e.g. arithmetic expressions.

As for error handling, YACC provides only a very crude mechanism. The user must add some productions involving the reserved token "error" to suggest places where errors are expected and where recovery might take place. Upon detection of an error, the parser pops its stack until it enters a state where the token "error" is legal. It then shifts this immaterial token and restarts in that state. It remains in error state until three tokens have been successfully read and shifted; until then the detection of another error produces no message.

As for semantics, YACC provides for the execution of some actions upon reduction to a given production. These actions are written in C and are embedded in the description of the productions themselves. They may also appear inside the productions rather than at the end; however YACC internally translates such cases into new non-terminals and productions to execute the action upon reduction of these new productions.

Those actions may have a "return value", which is actually a single synthesized attribute. The attribute of the LHS is denoted by $$ and those of the RHS symbols are denoted by $1, $2, etc... Terminals have also an attribute which must be computed by the scanner. Inside the actions, those attributes can be used as any variable. In fact they are references to elements of a stack which runs in parallel with the parse stack, as is usual with bottom-up parsing.

Having only one attribute may seem rather restrictive, but this attribute may be a structure, with many fields, thus obviating the problem. It is also possible to have this attribute have a type which depends on the symbol to which it is attached.

The result of the "evaluation" is the attribute of the start symbol. However, since the actions are any block of C statements, the user can store some intermediate results when desired.

6) Optimizations

None required.

7) Applications and performances

Applications of YACC are numerous. Among the ones developped for the UNIX environment at Bell Labs, let us quote :

- compilers for C (the Portable C Compiler), APL, Pascal, RATFOR, etc.
- a C program checker (lint),
- a mathematical texts typesetter (eqn),
- a Fortran debugging, system,
- several desk calculators,
- a document retrieval system.

The usage of YACC spreads continuously as the number of UNIX sites grow up.

YACC executable code is 200 kbytes on a 68000 based machine. Th produced parsers are very efficient and reasonably compact.

8) Projects

None.

A revised version of YACC (EYACC), including a much improved error recovery was developped in University of California in Berkeley and runs on Berkeley UNIX.

9) References

S.C. Johnson : "YACC. Yet Another Compiler-Compiler", report CS-TR-32, Bell Laboratories, Murray Hill, NJ (July 1975). Also in "UNIX Programmer's Manual", Volume 2 (any edition).

OTHER SYSTEMS RELATED WITH ATTRIBUTE GRAMMARS

1) HFP

Name : HFP (Hierarchical and Functional Programming)

Author(s): Takuya KATAMAYA, Yutaka HOSHINO (1980-?)

Address : Dept. of Computer Science, Tokyo Institute of Technology TOKYO 152 (Japan)

Description: HFP is an application of AGs to a programming methodology. The basic idea is to develop programs as attribute grammars. Programs are decomposed hierarchically into modules ; each module is characterized by its inputs and its outputs. To such a description is associated an attribute grammar : the module names are the non-terminals of a context-free grammar describing the caller/callee relationships between modules. The semantic rules correspond to the body of each module. To each "production" of the grammar is associated a decomposition predicate telling whether this decomposition is applicable ; these predicates involve the values of the inputs of the LHS module, i.e. its inherited attributes (deterministic case, the only one used in HFP).

Thus the derivation tree is a function of the inputs to the root module. HFP uses absolutely non-circular AGs, which are translated into procedures [Kat 80]. A proof method based on assertions is associated to HFP [KH 81] ; it has been proved sound and complete in the sense of program schemes, under some restrictive conditions.

References : Kat 80, Kat 81a,b, KH 81, Kat 84

2) TAG

Name : TAG (Tester for Attribute Grammars)

Author(s) : Mehdi JAZAYERI, Diane POZEFSKY (1975-1980)

Address : Dept. of Computer Science, University of Noth Carolina, Chapel Hill, NC 27514 (USA)

Description:TAG is a tester which performs the ASE (Alternating Semantic Evaluator) test on AGs. The grammars which have been tested include PL 360, an ALGOL subset, SIMULA, PASCAL, S-FORTRAN. The authors have also studied, from a theoretical point of vue, two optimizations applicable to evaluation in passes : elimination of the parse tree and dynamic allocation of attribute values.

References: JW 75, JP 77a,b, JP 79, JP 80a,b, JP 81, PJ 78a,b, Poz 79, KMP 75.

3) PAULSON'S SYSTEM

Name : undefined : sometimes called CGSG (Compiler Generator for Semantic Grammars) [Pau 81] or PSP [Ple 84]

Author(s) : Lawrence PAULSON (1979-1982)

Address : Dept. of Computer Science, Stanford University, Stanford CA 94305 (USA). Now at Computer Laboratory, University of Cambridge, Cambridge CB2 3QG (UK)

Description: CGSG is a compiler generator for languages described by a semantic grammar. A semantic grammar is a combination of (extended) attribute grammars, for describing the syntax and static semantics of the language, and of denotational semantics [SS 71, Gor 79] for describing the dynamic semantics. The compiler generator consists of the grammar analyzer, which converts a semantic grammar into a language description file, the universal translator, which reads that file and then compiles programs into stack machine (SECD) instructions, reporting semantic errors, and the stack machine which reads the program's input, executes the instructions and prints the outputs.

Attributes evaluation is performed in the universal translator, and uses the DAG evaluation devised by Madsen [Mad 80a,b,c,] : the parse tree is not constructed, instead the compound dependency graph is built. The graph is then traversed in depth-first manner, computing attributes upon return. This allows the dynamic detections of circularities. The result is the attributed DAG itself, which is then processed by the simplifier and the code generator.

The grammar analyzer is rather efficient, but the universal compiler translates Pascal programs 25 times slower than a conventional compiler, and the stack machine executes them 1000 times slower [Pau 82] . Because of storage needs, the universal translator cannot compile programs more than 20 pages long. The system is written in standard Pascal and is 4400 + 3900 + 1300 lines long. It has produced compilers for Pascal, Fortran and a number of smaller languages.

The main interesting point of CGSG is the clean description of run-time semantics by means of denotational semantics.

References: Pau 81, Pau 82, Pau 84, Ple 84.

4) PERLUETTE

Name : Perluette

Author(s) : M.C. GAUDEL , Ph. DESCHAMP, M. MAZAUD, R. RAKOTOZAFY (1978-1983)

Address : INRIA, BP. 105, 78153 LE CHESNAY Cedex (France).

Description : Perluette is a compiler generator based on abstract data types (ADTs). Source and target languages are described by ADTs and the translation is defined as a morphism from the algebras defined by the source ADT to algebras defined by the target ADT. This is the second phase of the generated compilers. The first phase is a translation of the source program into the corresponding term of the source ADT, and is described by an attribute grammar. The third phase is the generation of target code from the target ADT term, and is described by code templates. This modularization of the generated compiler is reflected in the modularization of the inputs to Perluette: the source language and target machine can be described

completely separately, and it is possible to try several translation choices of a given source language into a given machine code. These translations can also be proved correct in an algebraic framework.

The first phase of Perluette and its generated compilers is a variation of the FNC system (q.v.) ; in fact FNC was designed to replace DELTA (q.v.) in Perluette. The variations consist of special constructs to deal with terms of the source ADT, especially to provide a better type-checking . FNC was also used to generate a large part of Perluette itself.

The generated compilers are written in Lisp and are reasonably efficient. A number of toy languages have been described and implemented. However, theoretical problems have arisen when trying to describe Pascal, and are not yet resolved.

The strong points of Perluette are its modularity, both for its inputs and its outputs, and the possibility to prove the correctness of the generated compilers.

References : Gau 80a,b, Gau 81, Des 82.

CONCLUSION

The different systems presented in this paper have considerably different ambitions : some are very simple, performing little more than syntax-directed translations, others aim at providing a tool for generating a whole compiler. The efficiency of the generated evaluators is generally the inverse of their expressive power (the class of accepted AGs), but some systems break this rule (e.g. GAG). Some can run on micro- or minicomputers (e.g. LINGUIST-86), others need a large mainframe.

Figure 1 classifies the different systems according to their expressive power. Figure 2 gives a two-dimensional classification, according to the expressive power and the (approximative) date. It shows a tendency to accept larger classes, because of their larger power which eases the writing of AGs by getting rid of implementation details, and because much work made the generated evaluators more efficient in time and in space. However there exist exceptions in both ways: the new HLP84 is of restricted power but seeks much efficiency. Conversely, the first available systems, FOLDS and DELTA, are very general.

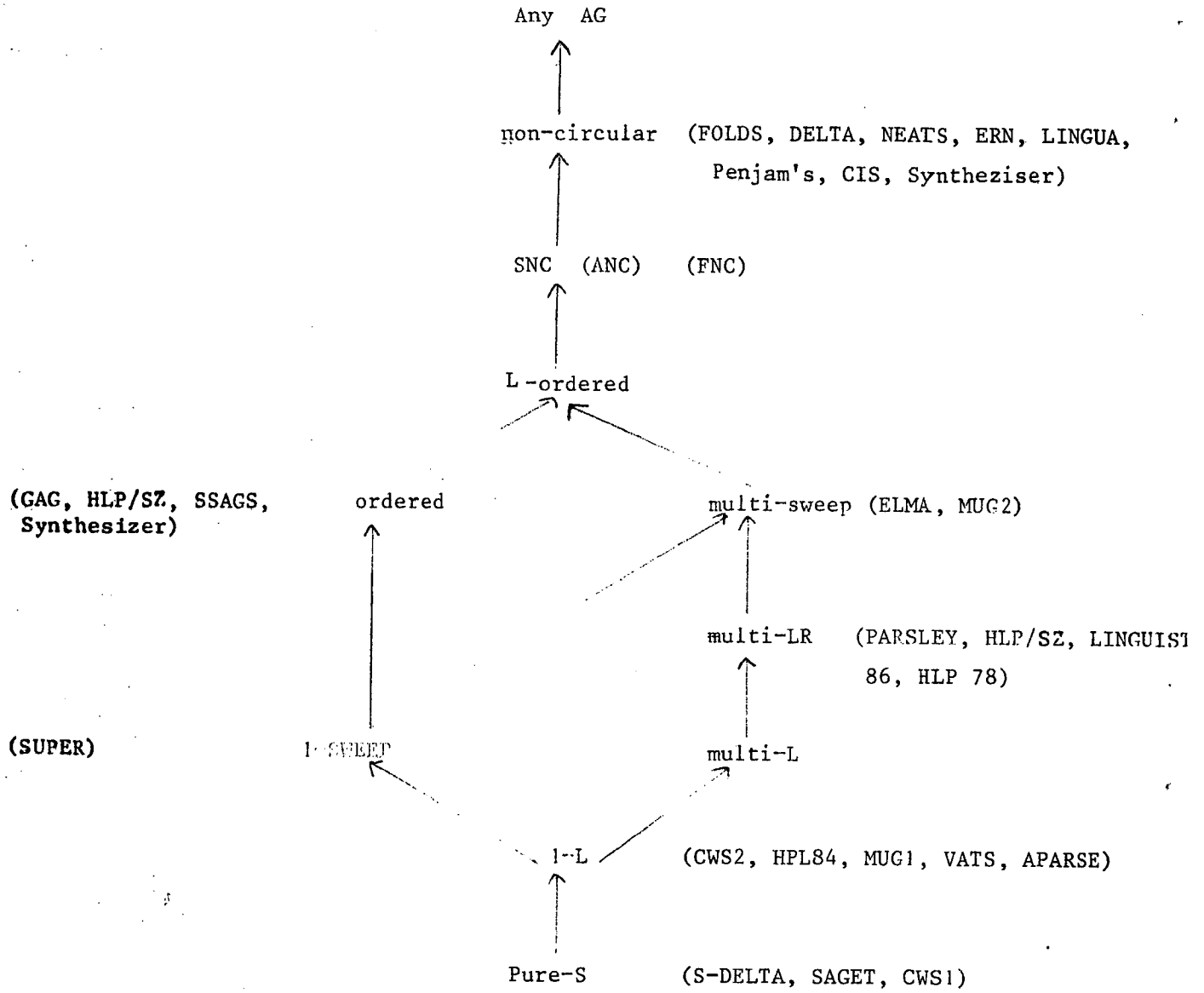


Figure 1: Expressive power of the systems

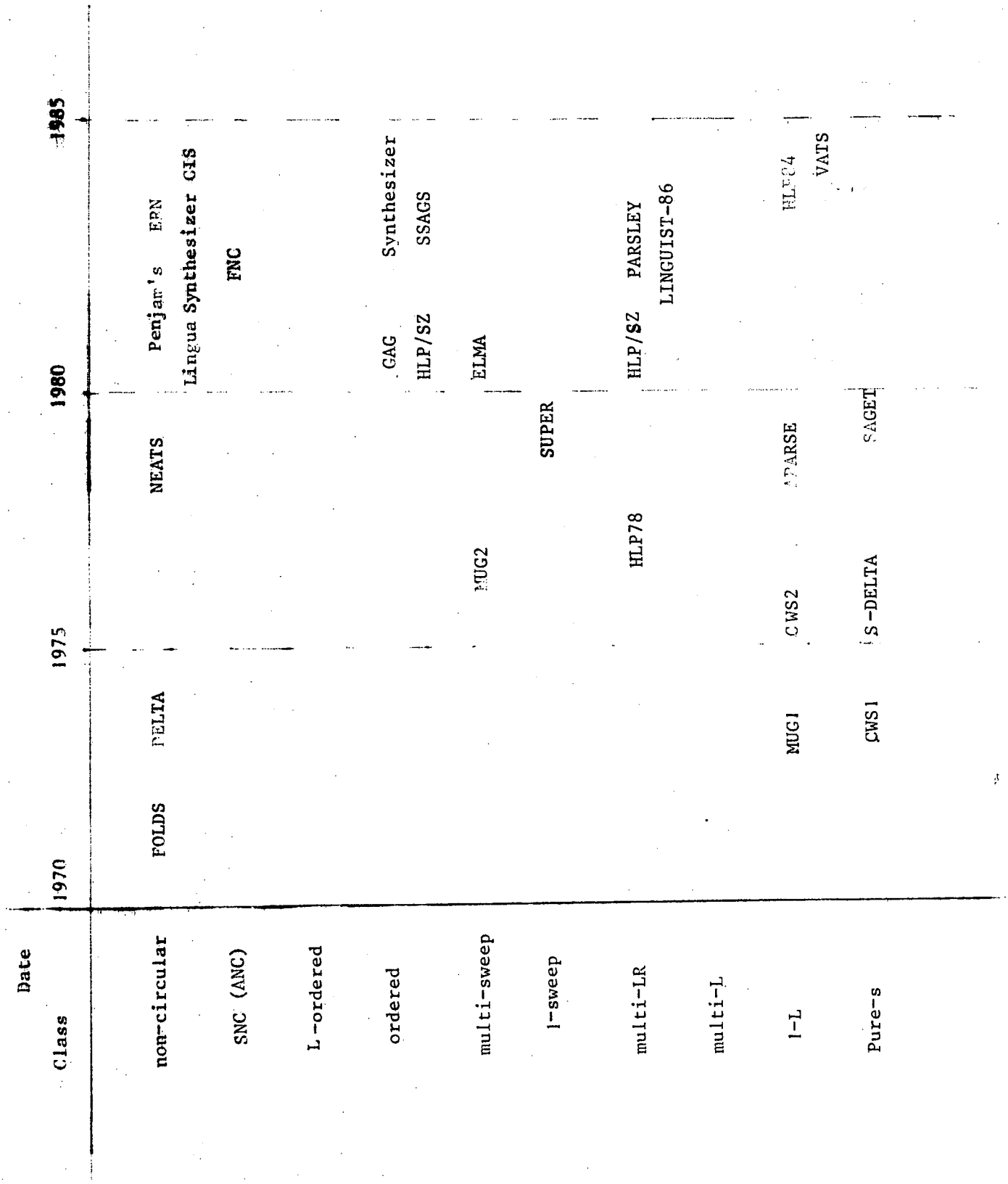


Figure 2 : Historical evolution

We hope that we have achieved our goal: provide an exhaustive catalogue of practical systems based on attribute grammars.

Acknowledgements

We would like to thank all the people who answered our questionaire.

Many thanks also go to Nelly Maloisel for her careful and patient typing of this enormous work.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique