An implementation of the HLP

By T. GYIMÓTHY*, E. SIMON*, Á. MAKAY**

Introduction

The Helsinki Language Processor (HLP) system was designed originally [7] for description of programming languages and for automatic generation of compilers. Saving the descriptional metalanguages different implementations have a great freedom with respect to the applications of parsing, semantic evaluation and software generation technics. Our implementation chooses SIMULA 67 [1] as base language which influences the collection of semantic functions usable for description of semantic features and the structure of the generated compiler too.

This text follows the steps of the generating process. A source language L is assumed which has a lexical description on the lexical metalanguage and a syntactic-semantic description on that metalanguage of the HLP.

There are two hand-written lexical analyzers for the metalanguages. One of them receives the lexical description of L and produces the input for the generator of the lexical analyzer of L. This will be constructed as a finite automaton. The other works on the syntactic-semantic description of L fundamentally in the form of an attribute grammar, producing the input for the semantic evaluator and for the pure syntax constructor. Because there may be different token class names and terminal strings in the lexical and syntactical description, unification of the symbol table of the generated lexical analyzer must be executed after that two lexical analysis. In the semantic description of L we can use attributes as SIMULA types involving simple types, classes, expressions, functions, statements and predefined standard procedures.

Having the pure syntax of L, the parser generator checks the grammar being of type LR (1) [2]. If it is so it constructs the table of the optimal parser of type LR (1), LALR (1), SLR (1) or LR (0).

We can choose one of the modified strategies ASE [3] or OAG [4] for computing the necessary passes and order of the evaluation of the attribute values in the generated compiler. For each syntax rule a SIMULA class is constructed, which contains the actions of parsing and evaluation decomposed to passes, anywhere this rule is applied in a derivation. One-pass compilation is possible if we have only synthesized attributes and it means, that the values of all attribute occurences are evaluable during parsing bottom up. This is a sufficient condition and so a proposition with respect to the formulation of the grammar.

By this means we have all components of the combined parser and semantic evaluator. Working under the control of the parsing table new objects of types predefined in the above SIMULA classes are created and connected as the derivation tree. Subsequent passes are executed by reactivating and deactivating the objects as the inner structures of the classes prescibe the evaluation order.

The generators have the same structure as the generated compiler so there are possibilities to generate new variations of the lexical analysers and the parser by the system itself. We have written these parts of the HLPin the metalanguages of its own.

Structure of the generated compiler

The nucleus of the generated compiler (GC) consists of a parser based on a grammar G from the class or subclass of the LR (1) grammars. It constructs the derivation tree in the grammar G from the token stream produced from the incoming text $p \in L(G)$ by the generated lexical analyzer GL. The nodes of the derivation tree are the SIMULA objects of types (SIMULA classes) representing the rewriting rules in the grammar G. Local pointers inside the objects ensure the connections — edges — toward the nodes on a lower level of the tree.

The objects contain the local variables of the attribute occurrences too together with the calling sequence, which represents the attribute evaluation strategy predefined from the attribute dependencies of the grammar G by one of the algorithms ASE or OAG. During parsing, when a new object is activated not only a new node is generated in the derivation tree (bottom up) but those attributes are evaluated, which depend on previously evaluated attributes. After that the object — the procedural part of the object — detaches itself while accessing the contents of the variables of the attribute occurrences just evaluated is possible. These are usable by the objects on a higher level of the derivation tree. Reactivating an object a new package of attribute occurences not evaluated yet is evaluable. Of course during evaluation this object activates other objects too going up or down in the tree in the order of the strategy. After finite number of activating-deactivating action pairs an object together with all the objects on the lower levels have no attribute occurences not evaluated. This part of the tree is unnecessary so it is destroyed. Finally we have only the root of the derivation tree together with one or more attributes of the initial nonterminal of the grammar G. Generally these attributes serve the purposes of the target code generation.

Of course we can describe and so generate not only a compiler for a programming language by an attribute grammar — as the metalanguage of the system — but other special purpose systems based on context-free languages too: schemes of data bases, machine architectures, picture description and processing, and so on. The common feature of these tasks is, that there exist a class of very similar algorithms, each of which we can specify by a context-free grammar together with several special attributes. The result is, that we have a generated software system specialized to one task only and the gain is in time or space complexity. It is the case of a compiler too: GC has a parser for one grammar and one strategy for the evaluation of a given attribute set.

Although it is possible to describe the generation of the target code by an attribute in the metalanguage too, we recommend a final pass for it based on the other attribute values evaluated earlier. Several procedures well defined for this purpose can help the users in that — target language dependent — job. So far we have neglected this aspect because we need experiences in large-sized and complicated languages.

Lexical metalanguage

The lexical metalanguage is used to describe the lexical structure of the source language for automatic construction of the lexical analyzer which forms tokens from the character strings of the source program. A description on the lexical metalanguage consists of five parts. In the first part a collection of *character sets* is defined. Specification of *token classes* by regular expressions can be found in the second part. The description of *transformations* concerns characters and token classes too. Transformations are performed during the isolation of a character or tokens. In *action blocks* the scanning sequence, screening of keywords from token classes and the way the isolated tokens are sent to the syntax analyzer, are given.

To give an idea of what a lexical description looks like we refer to the description of a simple block structured language called BLOCK_HLP given in Appendix.

The syntactic and semantic metalanguage

The definition of an attribute grammar is divided into five parts. First the *inherited* and *synthesized attributes* must be defined by SIMULA types. It should be noted that the concept of global attributes was not implemented. Global attributes can be replaced by SIMULA objects. In *nonterminal declaration* those nonterminals are declared which appear in the production list as the left-hand side of at least one production. Each nonterminal declaration has a possibly empty attribute list associated with it. An attribute from this list is associated with all nonterminals appering in the nonterminal list. The third part of the description is the declaration of the *start symbol.* We assume the grammar to be reduced. The auxiliary SIMULA variables, classes, functions and procedures which are used in the semantic rules and code generation are declared in the *procedure declaration* part.

As in the original HLP system we employ BNF (Backus Naur Form) description method for the syntax of the source language. *Semantic rules* and *code generation* are built in the *productions.* Note that if the semantic part is empty for one production, then the use of ECF (Extended Context Free) description [5] on the right-hand side is allowable. According to the SIMULA features the original notation of an attribute occurence is modified. In a production, if an attribute is associated to the left-hand side nonterminal, then only the attribute name must occur. Other attribute occurences are denoted by

nonterminal name · attribute name.

In Appendix the syntactic semantic description of the language BLOCK_HLP can be found too.

Attribute grammars

An attribute grammar (AG) can be considered as an extension of a context free (CF) grammar with attributes and semantic rules defining values of attributes. These attributes serve to describe the semantic features of the language elements.

An AG is a 3-tuple

$$AG = (G, A, F),$$

where $G=(V_N, V_T, P, S)$ is a reduced CF grammar, V_N, V_T, P and S denote the nonterminals, terminals, productions and the start symbol of the grammar respectively.

A production $p \in P$ has the form

$$p\colon X_0 \to X_1 \dots X_{n_p}, \quad \text{where} \quad n_p \geqq 0, \quad X_0 \in V_N, \quad X_i \in V_N \cup V_T \quad (1 \leqq i \leqq n_p).$$

The finite set A is the set of attributes. There is a fixed set $A(X)$ associated with each nonterminal $X \in V_N$ denoting the attributes of X. For an $X \in V_N$, $p \in P$ and $a \in A(X)$ $X \cdot a$ denotes an attribute occurrence in p. An attribute can be either inherited or synthesized, so each $A(X)$ is partitioned into two disjoint subsets, $I(X)$ and $S(X)$.

The set

$$A_p = \bigcup_{i=0}^{n_p} \bigcup_{a \in A(X_i)} X_i \cdot a$$

denotes all attribute occurrences in a syntactic rule p.

The set F consists of semantic rules associated with syntactic rules too. A semantic rule is a function type defined on attribute occurrences as argument types. For each attribute we have a set of attribute values (the domain of attribute) and for each semantic rule a semantic function defined on the sets which are related to its type. Formally, let us denote by F_p the rules associated with syntactic rule p, then

$$F = \bigcup_{p \in P} F_p .$$

We classify the set A_p into an output attribute occurrence set

$$\mathrm{OA}_p = \{X_i \cdot a \mid (i = 0 \quad \text{and} \quad a \in S(X_i)) \quad \text{or} \quad (i > 0 \quad \text{and} \quad a \in I(X_i))\}$$

and an input attribute occurrence set [6].

$$\mathrm{IA}_p = A_p - \mathrm{OA}_p$$

We assume, that for each $X_i \cdot a \in \mathrm{OA}_p$ there is exactly one semantic rule $f \in F_p$ the function related to it defines the value of $X_i \cdot a$. An AG is in normal form provided that only input occurrences appear as arguments of the semantic rules.

Evaluation of attribute values

Denote by t a derivation tree in the grammar G. If a node of t is labeled by X, then we can augment it by the attribute occurrences of X and their semantic rules defined by two syntactic rules. One of these is applied on the level over X in t and defines the inherited occurrences, while the other on the level under X determines the synthesized ones. Naturally, the root has no inherited and the leaves have no synthesized occurrences. (Leaves have attributes defined by the lexical analyzer which can be considered as synthesized ones.) A rule p, so an attribute occurrence may occur several times in t. We distinguish them and if it would be confusing we say occurrence in a tree t. Denote by T_{AG} the set of the augmented derivation trees in AG.

Let be given the set of semantic functions $\tilde{F}$ associated with F. The value of each attribute occurrence in t is computed by one of these functions and it is computable only if the argument values are computed. Therefore we have dependencies among attribute occurrences in the tree t. We denote by $(X_i \cdot a \to X_j \cdot b)$ the fact, that the function defining the value of $X_j \cdot b$ in t has the value of $X_i \cdot a$ as an argument. We say that $X_j \cdot b$ depends on $X_i \cdot a$ in t. By this relation the set F and the tree t induce a dependency graph D_t. If D_t has no cycle it determines an evaluation order for the computation of the values of all attribute occurrences in t. An attribute grammar is noncircular if there is no derivation tree with dependency graph containing a cycle. The decision whether an AG is noncircular requires algorithms of exponential complexity.

To determine the dependency graph to each derivation is time-consuming during compilation. For several subclasses of AG's it is possible to determine an evaluation strategy based on the grammar only. Such a strategy consists of an ordering of the attribute occurrences in the rules of the grammar in the form of a dependency graph D and means, that wherever an occurrence $X \cdot a$ appears in any tree t, it is computable if the occurrences on which it depends by D are already evaluated. The problem is determine D from the AG. During compilation we have to follow the evaluation order defined by D for each derivation tree. Naturally it is a tree traversal strategy and one travers may be seen as a pass of the compilation.

Two subclasses of AG's are considered in our system in accordance with them two algorithms, ASE and OAG serve to generate evaluation strategies.

ASE

The ASE algorithm is based on a fixed tree traversal strategy. An AG is ASE if any $t \in T_{\mathrm{AG}}$ is evaluable during m alternating depht-first, left-to-right (L—R) and depht-first right-to-left (R—L) tree traversal passes.

The attribute evaluation during an L—R traversal can be illustrated for a syntactic rule p: $X_0 \to X_1 \dots X_{n_p}$ as follows.

```
PROCEDURE TRAVERSE ($X_0$);
BEGIN
FOR  $i := 1$ STEP 1 UNTIL $n_p$ DO
```

```
    BEGIN EVAL ($I(X_i)$); TRAVERSE ($X_i$) END;
  EVAL ($S(X_0)$);
  END OF TRAVERSE;
```

During an $R—L$ pass the **FOR** statement above has the form

```
  FOR $i:=n_p$ STEP $-1$ UNTIL 1 DO
```

The ASE algorithm makes a membership test for an AG by this traversal procedure, and assigns attributes to passes. By the EVAL procedure we denoted the computation of the values of the attributes. The different instances of the same attribute is evaluated during the same pass.

Our experiences show that the ASE subclass is large enough and can be applied well in a compiler writing system. But it needs some modification in the original algorithm to use it in a practical system. For example we need not traverse a subtree during the ith pass if there is no evaluable attribute in this subtree. It can be decided by the following test.

Denote $H(X)$ the set of nonterminals which can be derived from an $X \in V_N$. It is easy to generate these sets by the transitive closure using P.

Let $K(X)= \bigcup_{Y \in H(X)} A(Y)$, and denote by A_j the set of attributes which can be evaluated during the jth pass.

If $(K(X) \cup S(X)) \cap A_j = \emptyset$, then for an $X_i=X$ we will not call the TRAVERSE (X_i) during the jth pass.

In the ASE algorithm the tree traversal and the attribute evaluation starts from the root of the derivation tree. In our system we use bottom-up tree constructor and many synthesized attributes can be evaluated interleaved with the construction of the derivation tree. These synthesized attributes can be easily assigned by the TRAVERSE procedure often decreasing the number of evaluation passes of an AG.

We can ensure an efficient space management technique for a generated compiler by using an extended version of ASE algorithm. We test for each $p \in P$ whether after the ith pass the attiibutes of the subtrees which can be derived from p are computed or not. If each of them are computed we generate a statement for the rule p which releases these subtrees. This technique is based on a garbage collector and is very efficient, because large parts of a derivation tree are released during the construction of the tree.

The ASE algorithm is pessimistic in the sense that it considers all dependencies for an attribute a. E.g. there are dependencies for an attribute a in the rules p and q, but there is no derivation tree containing the rules p and q together. Generally this does not occur in practical programming languages but it causes problems in some types of languages. Whether there is a derivation tree containing the rules p and q together may be decided by a simple algorithm using the sets $H(x)$.

OAG

In this section we give a short description of the OAG algorithm using some notations of [4]. We modified this algorithm, so an attribute evaluation strategy is given for a larger subclass of noncircular AG's. The time needed for the modified algorithm does not significantly differ from the time needed for the original algorithm.

As opposed to ASE algorithm in the OAG algorithm there is not a predefined tree traversal strategy. For each $AG \in OAG$ an attribute evaluation strategy is generated, and all derivation trees of the AG can be evaluated by this strategy. The OAG algorithm for each $X \in V_N$ constructs a partial order over the set $A(X)$, such that in any derivation tree containing X its attributes are evaluable in that order.

Denote by $\mathrm{DS}(X)$ the partial order over the $A(X)$, and let

$$\mathrm{DS} = \bigcup_{X \in V_N} \mathrm{DS}(X)$$

be the set of these partial orders.

We define dependency graphs over the attribute occurrences of syntactic rules and over the attributes of nonterminals, finally we construct DS using these graphs.

The dependency graph DP_p contains the direct dependencies between attribute occurrences associated to a syntactic rule p.

$$\mathrm{DP}_p = \{(X_i \cdot a \to X_j \cdot b) \mid \text{there is an } f \in F_p \text{ defining } X_j \cdot b \text{ depending on } X_i \cdot a\}$$

$$\mathrm{DP} = \bigcup_{p \in P} \mathrm{DP}_p$$

The dependency graph IDP can be constructed from the DP

$$\mathrm{IDP}_p = \mathrm{DP}_p \cup \{(X_i \cdot a \to X_i \cdot b) \mid X_i \text{ occurs in rules } p \text{ and } q,\ (X_i \cdot a \to X_i \cdot b) \in \mathrm{IDP}_q^+\},$$

where IDP_q^+ denotes the nonreflexive, transitive closure of IDP_q.

$$\mathrm{IDP} = \bigcup_{p \in P} \mathrm{IDP}_p$$

The graph IDP comprises the direct and indirect dependencies of attribute occurrences. For an $X \in V_N$ the dependency graph $\mathrm{IDS}(X)$ contains the induced dependencies between attributes of X

$$\mathrm{IDS}(X) = \{(X \cdot a \to X \cdot b) \mid \text{there is an } X_i = X \text{ in a rule } p \text{ and } (X_i \cdot a \to X_i \cdot b) \in \mathrm{IDP}_p\}$$

$$\mathrm{IDS} = \bigcup_{X \in V_N} \mathrm{IDS}(X).$$

The set DS can be constructed using IDS. For an $X \in V_N$ the set $A(X)$ is partitioned into disjoint subsets $A(X)_i$, and $\mathrm{DS}(X)$ defines a linear ordering over these subsets. The sets $A(X)_i$ are determined such that for an $a \in A(X)_i$ if $(X \cdot a \to X \cdot b) \in \mathrm{IDS}(X)$ and $b \in A(X)_k$, then $k \leq i$. The sets $A(X)_i$ consist of either synthesized or inherited attributes only. The $\mathrm{DS}(X)$ defines an alternating sequence of the synthesized and inherited sets $A(X)_i$.

$$\mathrm{DS}(X) = \mathrm{IDS}(X) \cup \{(X \cdot a \to X \cdot b) \mid X \cdot a \in A(X)_k, X \cdot b \in A(X)_{k-1}, 2 \leq k \leq m_x\},$$

where m_x is the number of the sets $A(X)_i$. The extended dependency graph EDP is defined by IDP and DS.

$$\mathrm{EDP}_p = \mathrm{IDP}_p \cup \{(X_i\cdot a \to X_i\cdot b) \mid (X\cdot a \to X\cdot b)\in \mathrm{DS}(X),$$
$$X_i = X \text{ and } X_i \text{ occurs in rule } p\}$$
$$\mathrm{EDP} = \bigcup_{p\in P} \mathrm{EDP}_p.$$

A given AG is an OAG iff the EDP is noncircular. We implemented the OAG algorithm as a part of our compiler writing system. We have favourable experiences using the algorithm, but we have found simple attribute grammars (occurring in practical applications, see Fig. 1), where the IDP is noncircular but the EDP is circular. We modified the OAG algorithm so that in these cases we generate a new EDP.

The graphs DP, IDP, IDS, DS are computed using the original algorithm. In the next step for each $X\in V_N$ and $(X\cdot a\to X\cdot b)\in \mathrm{DS}(X)$—$\mathrm{IDS}(X)$ we add $(X\cdot a\to X\cdot b)$ to IDP_p, if X occurs in rule p, and construct IDP_p^+. If a $(Y\cdot c\to Y\cdot d)$ is induced in IDP_p^+, then

(a) if $(Y\cdot d\to Y\cdot c)\in \mathrm{DS}(Y)$—$\mathrm{IDS}(Y)$, then we add $(Y\cdot c\to Y\cdot d)$ to $\mathrm{IDS}(Y)$ and generate a new $\mathrm{DS}(Y)$ using the modified $\mathrm{IDS}(Y)$,

(b) if $(Y\cdot d\to Y\cdot c)\in \mathrm{IDS}(Y)$, then the algorithm is finished and the given AG is not an OAG,

(c) otherwise we have $(Y\cdot c\to Y\cdot d)$ out of consideration.

If each $(X\cdot a\to X\cdot b)\in \mathrm{DS}(X)$—$\mathrm{IDS}(X)$ is added for an $X\in V_N$, then the set $\mathrm{DS}(X)$ is not changed later on.

In Fig. 1 we show an AG which is neither ASE nor OAG but for which an attribute evaluation strategy can be generated using the modified OAG algorithm. We denote by ○ an inherited attribute and by ● a synthesized one.

The dependencies in rule 2 show that AG $\notin$ ASE. We construct the sets $A(Y)_i$ and $A(Z)_i$ using the rules 1, 3, 5. The sets $A(Y)_1=\varnothing$, $A(Y)_2=\{e, g\}$, $A(Y)_3=\{f\}$ and $A(Z)_1=\{f\}$, $A(Z)_2=\{e\}$ imply that $(Y\cdot f\to Y\cdot e)\in \mathrm{DS}(Y)$ and $(Z\cdot e\to Z\cdot f)\in$ $\in \mathrm{DS}(Z)$. If we construct EDP_3 by $\mathrm{DS}(Y)$ and $\mathrm{DS}(Z)$ it will be circular, so AG $\notin$ OAG. Using the modified algorithm, if we add $(Y\cdot f\to Y\cdot e)$ to IDP_3, then $(Z\cdot f\to Z\cdot e)$ is induced in $\mathrm{IDS}(Z)$. The new $\mathrm{DS}(Z)$ is constructed from the sets $A(Z)_1=\varnothing$, $A(Z)_2=\{e\}$, $A(Z)_3=\{f\}$ and the EDP is generated by this $\mathrm{DS}(Z)$ will be noncircular. It is easy to prove that for an AG $\in$ OAG the modified algorithm does not change the set DS and graph EDP. The OAG algorithm for each $p\in P$ generates a visit-sequence VS_p using the graph EDP_p. Each VS_p is linear sequence of node visits and attribute evaluations and it is easy to generate an attribute evaluation strategy using the sets VS_p.

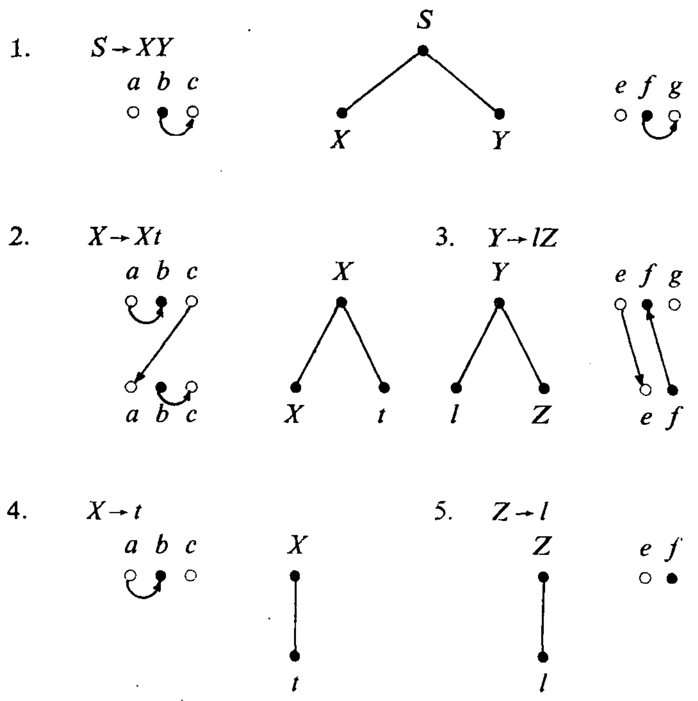

Fig. 1

Construction of the parser

In the present paragraph the logical description of parsing autòmata constructor modul is given. This modul serves to compute the state transitions for finite state and stack automaton too. The definition of the token classes by regular expressions and the description of an ECF grammar are coded uniform manner. Consequently the procedure which computes the parsing states can be controlled at the job control level to generate finite, ELR (1), ELALR (1), ESLR (1) or ELR (0) [2] states too. The states are represented by SIMULA objects based on the following declarations.

```
CLASS ITEM (NO, DOT, RSET);
INTEGER NO, DOT; REF (SET) RSET;
  BEGIN
  REF (ITEM) LINK;
  END ITEM;
CLASS SET (BOUND);
INTEGER BOUND;
  BEGIN INTEGER ARRAY TSET [0: BOUND];
  END SET;
```

It is easy to see that this representation has two advantages. The finite and LR (0) states which have no *follower set* can be stored uniform manner. Secondly, those items which have the same follower sets store only one SIMULA reference to an object in which the followers have been written. If the computed state is equal to a state which has been computed earlier then the SIMULA run-time system releases the space by calling the garbage collector. For each computed state there is a table which contains a set of ordered pairs. The first element describes the state number from which this state has been derived. The second element contains the symbol code used to compute the considered state.

By applying the next theorem from [2] to our parser constructor we can obtaine an useful conclusion. An ELR (0) language can be parsed by a finite state automaton iff there is no state which can be derived by a nonterminal from more than one state. Hence, in order to generate a finite state automaton the ELR (0) states are computed first. It is followed by performing the *finite state test*.

After computing the selected type of states (ELR (1), ELALR (1), ESLR (1) and ELR (0)) a membership test will be performed together with parser code generation. If it produces true then the next type of states will be computed from the last states. The test are performed from ELR (1) to ELR (0). Some simple optimization procedure are executed during the tests. In the present version of our implementation there is no automatic error recovery procedure. Ordering of states on the base [8] an efficient error correcting algorithm is under development.

The states and the internal code of the lexical analyzer as a finite automaton are generated by the same modul. Of course we need additional service routines working in the lexical analyzers. These are written for metalanguage purposes, but they can be used in the generated compilers in the same form too.

Appendix

```
% LEXICAL DESCRIPTION FOR A SIMPLE BLOCK STRUCTURED
% LANGUAGE
% CALLED BLOCK_HLP
LEXICAL DESCRIPTION BLOCK_HLP
 CHARACTER SETS
    LETTER_OR_DIGIT=LETTER/DIGIT;
END OF CHARACTER SETS
TOKEN CLASSES
    UNDERSCORE = ≠_≠;
    IDENTIFIER=LETTER (LETTER_OR_DIGIT/UNDERSCORE)* [16];
    PROPERTY  =DIGIT+ [2];
    COMMENT   =≠%≠ ANY* ENDOFLINE;
    SPACES    =SPACE* ENDOFLINE;
    SPACES    =SPACE+;
END OF TOKEN CLASSES
    TRANSFORMATIONS ARE
    UNDERSCORE=>;
    END OF TRANSFORMATIONS
ACT_BLOCK: BEGIN
    IDENTIFIER=>IDENTIFIER / KEYSTRINGS;
    PROPERTY  =>PROPERTY;
    COMMENT   =>;
    SPACES    =>;
END OF ACT_BLOCK
END OF LEXICAL DESCRIPTION BLOCK_HLP.
   FINIS
% SYNTACTIC-SEMANTIC DESCRIPTION OF BLOCK_HLP
ATTRIBUTE GRAMMAR BLOCKHLP

SYNTHESIZED ATTRIBUTES ARE
    REF (SYMB) SYMREF; REF (SDECL) SEREF;
    INTEGER ID, TYPE, EXTYPE;
END OF SYNTHESIZED ATTRIBUTES
INHERITED ATTRIBUTES ARE
    REF (SBL) SYMT;
END OF INHERITED ATTRIBUTES
NONTERMINALS ARE
    PROGRAM;
    BLOCK HAS SYMT, SYMREF;
    STATLIST HAS SYMT, SYMREF;
    STAT HAS SYMT, SEREF;
    IDECL HAS ID, TYPE;
    EXDECL HAS SYMT, EXTYPE;
END OF NONTERMINALS

% PROCEDURES AND CLASSES

$$$$

    CLASS SBL (A, B);
    REF (SBL) A; REF (SYMB) B;
    BEGIN
    END OF SBL;

    CLASS SYMB (A, B);
    REF (SYMB)A; REF (SDECL)B;
    BEGIN
    END OF SYMB;
    CLASS SDECL (A, B);
    INTEGER A, B;
    BEGIN
    END OF SDECL;

    PROCEDURE FIND (A, B, C);
    NAME A;
    INTEGER A, B; REF (SBL) C;
    BEGIN
```

% The value of *A* will be the type of the variable *B*. This type is tried to find
% in the list of identifiers defined by $C \cdot B$. If *B* is not found in it, then *C* is replaced
% by $C \cdot A$. Repeating until having the type of *B* or being the list empty, the
% requested value is done or *A* is undeclared.

```
    END OF FIND

****

% END OF PROCEDURES AND CLASSES
  PRODUCTIONS ARE
%1%
```

```
PROGRAM = BLOCK;
DO

    BLOCK.SYMT :— NEW SBL (NONE, BLOCK.SYMREF);

END
%2%
BLOCK = STATLIST;
%3%
STATLIST = STATLIST STAT;
DO

    SYMREF :— IF STAT.SEREF =/= NONE THEN
                 NEW SYMB (STATLIST.SYMREF, STAT.SEREF) ELSE
                STATLIST.SYMREF;
END
%4%
STATLIST = STAT;
DO

    SYMREF :— IF STAT.SEREF == NONE THEN NONE
                 ELSE NEW SYMB (NONE, STAT.SEREF);
END
%5%
STAT = IDECL;
DO

    SEREF :— NEW SDECL (IDECL.ID,IDECL.TYPE);
END

%6%
STAT = EXDECL;
DO

    SEREF :— NONE;
END
%7%
STAT = BEGIN BLOCK END;
DO
    BLOCK.SYMT :— NEW SBL (SYMT,BLOCK.SYMREF);
    SEREF :— NONE;
END
%8%
IDECL = DECLARE IDENTIFIER PROPERTY;
DO
    ID := IDENTIFIER.VALUE;
    TYPE := PROPERTY.VALUE;
END
%9%
EXDECL = USE IDENTIFIER;
DO
```

```
    EXTYPE <= FIND (EXTYPE,IDENTIFIER.VALUE,EXDECL.SYMT);
END
END OF PRODUCTIONS
END OF ATTRIBUTE GRAMMAR

% SIMULA classes associated with two nonterminals and a production in the
% generated compiler
NODE        CLASS GRNODE 1;
            BEGIN COMMENT PROGRAM;
            END;
NODE        CLASS GRNODE 2;
            BEGIN COMMENT BLOCK;
            REF (SBL) SYMT;
            REF (SYMB) SYMREF;
            END ;
GRNODE 1    CLASS P 1;
            BEGIN COMMENT PROGRAM;
            REF (GRNODE 2) BLOCK;
           BLOCK: — POP QUA GRNODE 2;
           PUSH (GOTO (1), THIS NODE);
           DETACH;
           BLOCK.SYMT:— NEW SBL (NONE, BLOCK.SYMREF);
           CALL (BLOCK);
           DETACH;
            END ;
```

*RESEARCH GROUP ON THEORY OF AUTOMATA
HUNGARIAN ACADEMY OF SCIENCES
SOMOGYI U. 7
SZEGED, HUNGARY
H-6720

**DEPT. OF COMPUTER SCIENCE
A. JÓZSEF UNIVERSITY
ARADI VÉRTANÚK TERE 1
SZEGED, HUNGARY
H-6720

References

[1] DAHL, O. J., B. MYHRHAUG and K. NYGAARD, SIMULA 67 common base language, Norwegian Computing Center, Publication No. S-2, May 1968.
[2] HEILBRUNNER, S., A parsing automata approach to LR theory, *Theoret. Comput. Sci.*, v. 15, 1981, pp. 117—157.
[3] JAZAYERI, M. and K. G. WALTER, Alternating semantic evaluator, Proc. of the ACM 1975. Annual Conference, Oct. 1975, pp. 230—234.
[4] KASTENS, U., Ordered attribute grammars, *Acta Inform.*, v. 13, 1980, pp. 229—256.
[5] PURDOM, P. W., JR. BROWN and A. CYNTHIA, Parsing extended LR (*k*) grammars, *Acta Inform.* v. 15, 1981, pp. 115—127.
[6] RÄIHÄ, K. J., A space management technique for multi-pass attribute evaluators, University of Helsinki, Report A-1981-4, 1981.
[7] RÄIHÄ, K.J., M. SAARINEN, E. SOISALON—SOININEN and M. TIENARI, The compiler writing system HLP (Helsinki Language Processor). University of Helsinki. Report A—1978—2, 1978.
[8] RÖHRICH, J., Methods for the automatic construction for error correcting parsers, *Acta Inform.*, v. 13. 1980, pp. 115—139.

(Received Feb. 7, 1983)