Language extension in the HLP/SZ system

By E. SIMON

1. Introduction

It is the intent of this paper to describe a practicable modification of the HLP lexical metalanguage for specification of the language extension. Programmers should never be satisfied with languages which permit them to program everything. Therefore, there is a need for languages which help the programmer to find the most natural representation for the data structures he uses and the operations he wants to perform on them. This is clearly illustrated by the current trends in the evaluation of programming languages [4].

In order to achieve flexibility and power of expressions in programming languages, we must pay the price of greater complexity. In the 70's there was a tendency to retrench towards simpler languages such as PASCAL, even at the price of restricted flexibility and power of expressions. An alternative solution was the development of *extensible languages* too. The classification of language extension can be made on the basis of the stage of the translation process during which the definition of an extension is processed and the augment text is converted into a text in the base language.

In compiler-writing systems based on attribute grammars, two natural means are given to introduce extensions. In the first case the augment text is processed *during* the *lexical analysis,* and therefore the extension mechanism is a part of the generated lexical analyzer. Information which controls the recognition of the augment texts and which prescribes the generation of the definition texts can be obtained from the lexical metalanguage description. In this case the extension mechanism is generally similar to macrogenerators. Those lexical analyzers which have an extension mechanism must contain special stack automata to store parameters computed during isolation of a token. Extension mechanisms are generally implemented by recursive procedures. Adapted from [6], the extension executed during the lexical analysis is called the type-A extension. From the user's point of view, the type-A extension can be useful for a large class of problems, but there are some other classes where delay of the extension is needed until the translation process. For example, the recursive procedure call in BASIC language can not be introduced by a type-A extension, because the stack manipulations must be implemented at the target code level. It should be noted that the other type of language extension is based on *attributed tree transfor-*

mations. These are executed during syntax/semantic analysis of the token stream generated by the lexical analyzer. The attributed tree transformations are controlled by those attributes which are computed before transformations. In practice the application of both techniques is suitable.

In the case of the type-A extension there are two different methods to give definitions of the augments. In our HLP/SZ system the recursive definitions are given by generator expressions in lexical metalanguage, while the augments are contained by source programs. Hence, in all cases a new lexical analyzer must be generated on the basis of the new lexical description. If the generated lexical analyzer contains the total extension mechanism, including the parser generator, definition texts are given in the source programs. An extension executed by attributed tree transformations is now under development.

In the next part of this paper the modified lexical metalanguage is presented through examples. It is followed by a short description of our implementation based on SIMULA 67 language. In the following, a knowledge of the original HLP system cf. [1], [2]) is assumed.

2. Extension description in lexical metalanguage

A lexical description of a prgamming language defines the way the programs are written in that language. The streams of characters are divided into syntactically meaningful entities called *tokens.* Token classes are defined by regular expressions which are built up from character sets, terminals and tokens. In our system, five predefinite character sets are introduced with conventional meaning. These character sets are: LETTER, DIGIT, ANY, ENDOFLINE and SPACE. These names can be overdefined in the set declaration list. An essential difference between the original and the HLP/SZ system is that the *transformations* are related to the character set names and token class names only. The transformations are given after the token class description in the following way:

```
TRANSFORMATIONS ARE
identifier⇒; or                                    (1)
identifier_1⇒terminal_symbol;
END OF TRANSFORMATIONS.
```

In the first case, character sets assigned to the "identifier" are deleted during isolation of a token. In the second case, characters are changed by the terminal_symbol. If "identifier_1" is defined as the character set name, then the first character of the terminal_symbol will be inserted after deletion of the last input character. Applications of transformations can be found in the examples of this section.

In order to extend a language it is enough to create one or more generator expressions without any knowledge of the original parts of the lexical description. For introducing the generator expression, let us take the terminology of macrogenerators into consideration.

An *action block* consists of a collection of screen clauses in the original lexical metalanguage. Each screen clause denotes an identifier which is declared as a token class name in the lexical description. Generator expressions are assigned to token

class names in action blocks. Character strings belonging to the token classes are treated as augment texts or *macro calls. Macro definitions* are given by generator expressions. Let us assume that, in the definition of a token class assigned to a generator expression, token class names occur. Character strings isolated to these token class names are called *parameters.* References to these parameters in the generator expressions are given by the following syntax rules.

formal_parameter = ≠↓≠ unsigned_integer; (2)
formal_parameter = ≠↓≠generator_expression;

In the first case, parameters are *positionated,* that is the "unsigned_integer" designates the serial number of a parameter. Token class names occuring on the left-hand side of token class definitions are regarded as 0th parameters. Numbering of parameters on the right-hand side is defined from left to right. The serial number of the leftmost token class name, as parameter, is equal to 1. In the second case, the generator expression generates a token class name, which is called *keyword parameter.* We suppose that the generated token class name can be found on the right-hand side of the token definition assigned to the generator expression in which the original formal parameter reference occurred. It should be noted that the token class definition above does not contain the same token class names twice.

In the original lexical metalanguage, the control can be transferred to an other block using GOTO. After a token is processed according to a screen clause, the GOTO part appearing on its left-hand side indicates the block where the next token will be recognized. The default rule states that the block is not changed. In the case of an extension description, the GOTO part can appear after the token class name assigned to a generator expression. The GOTO part occuring behind the generator expression indicates the block where the generated definition text will be recognized.

Generator expressions are built up from *terminal symbols, parameter references* and *functions.* The functions typed as string are the following: BLANK, SUC, PRED and IT_IND. The first function has no argument, while SUC and PRED have one string argument which must be converted into an integer. The function SUC is assumed to be an incrementing function, and PRED to be a decrementing one. The meaning of the function IT_IND can be found in the following part of this section.

In order to build up a generator expression from terminals parameters and functions, string operations are needed. Similarly to regular expressions, the descriptive power of generator expressions is based on the operations *iteration* (∗), *concatenation* (juxtaposition) and *McCarthy expression.* The operator ∗ has the highest precedence and the McCarthy expression the lowest. The use of subexpressions enclosed in parentheses is permitted.

For example, we could have the description to be found in Figure 1. The name of this description is FACTORIAL and no character set definitions are needed. The strings belonging to the token class FACT begin with terminal *FACT.* The left parenthesis is followed by 1—2 digits and the right parenthesis. After the right parenthesis some blank characters can follows, which are deleted during recognition of the token (augment text) called FACT. During evaluation, the generator expression assigned to the token class name FACT, ↓1 contains the substring passed to INT_PARAM as integer parameters of the factorial call. Parameter reference ↓0 assigns the total augment text without last spaces. It can be seen that the extension is *recursive.*

```
LANGUAGE EXTENSION AT LEXICAL LEVEL

THE AUGMENT TEXT IS: FACT( INT_PARAM )

LEXICAL DESCRIPTION FACTORIAL
    TOKEN CLASSES
    NUMBER          =DIGIT+ [4];
    MULT            =≠*≠;
    FACT            =≠FACT≠ ≠(≠ INT_PARAM ≠)≠ SPACE*;
    INT_PARAM       =DIGIT+ [2] ;
    END OF TOKEN CLASSES
    TRANSFORMATIONS ARE
    SPACE           ⇒ ;
    END OF TRANSFORMATIONS
BLOCK:BEGIN
    FACT [BLOCK]    →(↓1 EQ ≠1≠ →→ ≠1≠ /
                      ↓0 EQ  ↓0  →→  ↓1  ≠*≠ ≠FACT(≠ PRED(↓1)
                                                    ≠)≠)
                    [BLOCK_1] ;
    END OF BLOCK
BLOCK_1:
    BEGIN
    NUMBER          ⇒NUMBER ;
    MULT [BLOCK]    ⇒MULT_OP ;
    END OF BLOCK_1
END OF LEXICAL DESCRIPTION FACTORIAL.

    FINIS
```

Figure 1

A part of the generated text will be recognized by the automata assigned to the token class FACT. During the processing of an augment text (macro call), a sequence of tokens called NUMBER and MULT_OP are generated in action block BLOCK_1. After the processing, the next active block will be the BLOCK by GOTO transfer assigned to token class name FACT.

In order to create macro calls which have other calls in their own parameter list, the following agreement is reached. If none of the token classes currently under consideration has a right-hand side from which a token class name TO can be derived, the string generated by TO will be normally processed; otherwise, the generated string will be strored *in the stack.*

As an example, let us consider the description of HANOI TOWERS in Figure 2. In this case the string generated by PARAM must be stored in the stack as a parameter of the augment text HANOI. Naturally, theere is a posibility to continue the example found in Figure 2 by creating an attribute grammar to synthesize the original HANOI call on the basis of the token stream generated by the lexical analyzer HANOI_TOWERS.

```
LEXICAL DESCRIPTION HANOI_TOWERS
CHARACTER SETS
    ABC         = ≠ABC≠ ;
END OF CHARACTER SETS
TOKEN CLASSES
    IDENTIFIER  = LETTER (LETTER/DIGIT)* ;
    INTEGER     = DIGIT+ [2] ;
    SPACES      = SPACE+ (/ENDOFLINE) ;
    DOT         = ≠.≠ ;
    NUMBER      = DIGIT+ ;
    FROM        = ABC ;
    TO          = ABC ;
    MOVE        = ≠XMOVE≠ ≠(≠ NUMBER ≠,≠ FROM ≠,≠ TO ≠)≠;
    SER_NUMB    = DIGIT+ ;
    PARAM2      = ABC ;
    PARAM1      = ABC ;
    PARAM       = ≠PARAM≠ ≠(≠ PARAM1 ≠,≠ PARAM2 ≠)≠ ;
    ARGUMENT    = (ABC/PARAM) ;
    HANOI       = ≠HANOI≠ ≠(≠ SER_NUMB (≠,≠ ARGUMENT)*
                                                    ≠)≠ ;
END OF TOKEN CLASSES
SCREEN_A:
BEGIN
    IDENTIFIER ⇒ KEYSTRINGS ;
    INTEGER    ⇒ INTEGER ;
    SPACES     ⇒ ;
    DOT        = DOT ;
    MOVE       → ≠MOVE≠ ≠ ≠ ≠THE≠ ↓1≠TH≠ ≠ ≠ ≠DISC≠ ≠ ≠
                  ≠FROM≠ ≠ ≠ ≠THE≠ ≠ ≠ ↓2 ≠.≠ ≠ ≠
                  ≠PLACE≠ ≠ ≠ ≠TO≠ ≠ ≠ ≠THE≠ ≠ ≠ ↓3 ≠.≠
                  ≠ ≠ ≠PLACE≠ ;
    HANOI      → (↓1 EQ ≠1≠ −→ ≠XMOVE≠ ≠(≠ ↓1 ≠,≠ ↓2 ≠,≠ ↓3
                  ≠)≠ /
                   ↓0 EQ ↓0 −→
                   ≠HANOI≠ ≠(≠ PRED(↓1) ≠,≠ ↓2 ≠,≠
                            ≠PARAM≠ ≠(≠ ↓2 ≠,≠ ↓3 ≠)≠ ≠)≠
                   ≠XMOVE≠ ≠(≠ ↓1 ≠,≠ ↓2 ≠,≠ ↓3 ≠)≠
                   ≠HANOI≠ ≠(≠ PRED(↓1) ≠,≠
                            ≠PARAM≠ ≠(≠ ↓2 ≠,≠ ↓3 ≠)≠ ≠,≠ ↓3
                                                    ≠)≠) ;
    PARAM      → (↓1 EQ ≠A≠ −→
                  (↓2 EQ ≠B≠ −→ ≠C≠ / ↓0 EQ ↓0 −→ ≠B≠) /
                   ↓1 EQ ≠B≠ −→
                  (↓2 EQ ≠A≠ −→ ≠C≠ / ↓0 EQ ↓0 −→ ≠A≠) /
                   ↓0 EQ ↓0 −→
                  (↓2 EQ ≠A≠ −→ ≠B≠ / ↓0 EQ ↓0 −→ ≠A≠)) ;
END OF SCREEN_A
END OF LEXICAL DESCRIPTION HANOI_TOWERS.
```

```
FINIS
```

Figure 2

HLP/SZEGED (0.2) **SOURCE TEXT**

```
 1
 2          EXTENSION OF AN ALGOLW SUBSET
 3            BY CASE AND FOR STATEMENTS
 4              5 OCTOBER 1983. SZEGED
 5
 6
 7  LEXICAL DESCRIPTION EXTENDED_ALGOLW_SUBSET
 8
 9  CHARACTER SETS
10**  PERCENT       = ≠%≠ ;
11    UNDERSCORE    = ≠_≠ ;
12    STMT_CHAR     = ANY - ≠;≠ / ENDOFLINE ;
13    END OF CHARACTER SETS
14
15  TOKEN CLASSES
16    IDENTIFIER    = LETTER (LETTER/DIGIT/UNDERSCORE)*
                      [16] ;
17    NUMBER        = DIGIT+ [8] ;
18    COMMENT       = PERCENT ANY* ENDOFLINE ;
19    SPECIALS      = DEFAULT TOKENS ;
20**  SPACES        = SPACE+ ;
21    LINE_SKIP     = SPACE* ENDOFLINE ;
22    CASE          = ≠XCASE≠ ≠ ≠INT_PAR≠ ≠ ≠OF≠ ≠ ≠
23                    (STATEMENT ≠;≠)* (SPACE/ENDOFLINE)*
                      ≠END≠ ;
24    FOR           = ≠XFOR≠ ≠ ≠ CYCLE_PAR ≠ ≠ ≠:=≠ ≠ ≠
                      FIRST_VALUE ≠ ≠ ≠STEP≠ ≠ ≠
25                    STEP_VALUE ≠ ≠ ≠UNTIL≠ ≠ ≠
26                    LAST_VALUE ≠ ≠ ≠DO≠ ≠ ≠
                      STATEMENT_1 ;
27    INT_PAR       = DIGIT+ [2] ;
28    STATEMENT     = STMT_CHAR* ;
29    CYCLE_PAR     = LETTER(LETTER/DIGIT/UNDERSCORE)* [16] ;
30**  FIRST_VALUE   = DIGIT+ [8] ;
31    STEP_VALUE    = DIGIT+ [8] ;
32    LAST_VALUE    = DIGIT+ [8] ;
33    STATEMENT_1   = STMT_CHAR* ;
34    STATEMENT_1   = ≠BEGIN≠ (ANY/ENDOFLINE)* ≠END≠ ;
35  END OF TOKEN CLASSES
36
37    TRANSFORMATIONS ARE
38    UNDERSCORE ⇒ ;
39    END OF TRANSFORMATIONS
40**
41  SCAN: BEGIN
```

```
42      IDENTIFIER      ⇒ IDENTIFIER/KEYSTRINGS;
43      NUMBER          ⇒ NUMBER ;
44      COMMENT         ⇒ ;
45      SPECIALS        ⇒ KEYSTRINGS ;
46      SPACES          ⇒ ;
47      LINE_SKIP       ⇒ ;
48      CASE            → (≠IF≠ ↓1 ≠=≠ ITIND ≠THEN≠ ↓SUC(ITIND)
49                        ≠ELSE≠)* [1] ≠;≠ ;
50 * *  FOR             → ≠BEGIN≠ ↓1 ≠:=≠ ↓2 ≠;≠ ≠LABEL:≠
51                        ≠IF≠ ↓1 ≠LE≠ ↓4 ≠THEN≠ ≠BEGIN≠ ↓5≠;≠
52                                ↓1 ≠:=≠ ↓1 ≠+≠ ↓3 ≠;≠
53                                 ≠GOTO≠ ≠LABEL;≠ ≠END≠ ≠;≠
54                        ≠END≠ ≠;≠ [SCAN] ;
55  END OF SCAN
56
57  END OF LEXICAL DESCRIPTION EXTENDED_ALGOLW_SUBSET.
58
59
```

```
60 * *  FINIS
```

Figure 3

In the following we define the format in which the number of iterations are given to an elementary subexpression of a generator expression. The number of an iteration is equal to 1 if the iteration specification is omitted. If it occurs, then it must be written in the following format:

iteration_number = ≠ * ≠ ≠[≠ generator_expression ≠]≠ ; or
iteration_number = ≠ * ≠ ≠[≠ unsigned_integer ≠]≠ ; (3),

assuming that the string generated by the "generator_expression" can be converted into an integer. It should be noted on the basis of (3) that *[2] and *[≠2≠] are (in the same way) syntactically correct items with different meanings. In the first case the value of the "generator_expression" designates the number of iterations to be executed. To illustrate the meaning of "iteration_number" defined secondly, let us consider a token class which is defined by iterations. Let us number the occurrences of iterations from left to right. In this case *[i] has the following meaning. The index i denotes a * symbol on the right-hand side of the token class definition, assigned to the generator expression in which *[i] occurs. In this way the item *[i] means the number of the i-th iterations performed by the automata during isolation of a token. In order to use the iteration number in the form *[i], practical additional features are required. Firstly, during execution of an iteration the elementary string expressions must be evaluated in each step. Secondary, we must introduce a new elementary string expression called IT_IND, which produces the value of the iteration index.

We are now ready to give two extensions of an ALGOLW subset in HLP/SZ lexical metalanguage. The original lexical description [3] is increased by some token class definitions to introduce the CASE and FOR statements into the base language. Additional token class definitions, such as STATEMENT, INT_PAR etc., are

needed to describe the parameters of the augments. It seems to me that the definition of a statement list by iteration, which can be seen in the description of the CASE statement (cf. Figure 3), is very useful for giving the extension assigned to the CASE. The generator expression is a very good example for application of the function IT_IND too. The meaning of the elementary string expression ↓SUC(IT_IND) is not trivial. During the first step of the interation cycle prescribed by *[1], the value of the iteration index is equal to 1. On the other hand, the serial number of the first STATEMENT parameter is equal to 2.

3. Implementation

Our implementation is based on SIMULA 67 language and it is now running on the CDC 3300 computer [5]. The system can generate two types of lexical analyzer, such as a lexical analyzer with an extension and without an extension facility. The type of generated lexical analyzer is given at the job control level.

There is a further facility too. If the source text is given without augments, the lexical analyzer containing procedures to execute extensions can be controlled so that, during its running, additional information for extensions will not be created. The hand-written analyzer of the HLP/SZ lexical metalanguage has no extension facilities, because in this case the processing time was the primary point of view. The generated lexical analyzer has automatic error-correction routines, which discontinue "one distance" lexical errors if it is possible. Therefore, a restriction is needed for the first characters of augments. The reader may imagine what happens if, for example, the new tokens called IDENTIFIER and CASE are being recognized in the same action block and the next four input characters are CAPE. In this case it can not be decided whether an error correction must be made or not. In this manner, the first character of an augment text must differ from the first characters of the other tokens assigned to the same block. In exchange for this, the augment text can be more exactly recognized. Information, needed to the error-correcting, parameter-generating and iteration-calculating routines, can be computed during the generation of the final states. Generator expressions are embedded into the generated lexical analyzer in a special tree language, to promote the fast evaluation of these.

Abstract

The Hungarian version of the Helsinki Language Processor, HLP/SZ [1], consists of modules for the lexical, syntactic and semantic processing of programming languages. The lexical metalanguage of our system has been modified so as to introduce the possibility of language extension. Augment texts, expressed in terms of constructs which are not part of the base language, are defined by token classes. The generator expressions, which are assigned to the token class names in action blocks, generate the definition texts. Definition texts can contain additional augments in a recursive way, and it will be processed according to an optional action block. Generator expressions built up from terminals, parameters and functions are controlled by "McCarthy expressions". The system implemented in SIMULA 67 language is now running on the CDC 3300 computer.

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

References

[1] GYIMÓTHY, T., E. SIMON and Á. MAKAY, An implementation of the HLP, Acta Cybernetica, Tom. 6, Fasc. 3, pp. 315—327.
[2] RÄIHÄ, K. J., M. SAARINEN, M. SARJAKOSKI, S. SIPPU, E. SOISALON-SOININEN and M. TEINARI, Revised report on the compiler writing system HLP78, University of Helsinki, Report A—1983-1, 130 pp.
[3] SIPPU, S., Syntax error handling in compilers, University of Helsinki, Report A-1981-1, 100 pp.
[4] SIMON, E., Language design objectives and the CHANGE system, Computational Linguistics and Computer Languages, v. 15, 1982, pp. 229—247.
[5] SIMON, E. and T. GYIMÓTHY, Using attribute grammars to generate compilers, Információ Elektronika, v. 1984, 2, in Hungarian.
[6] SOLNTSEFF, N. and A. YEZERSKY, A survey of extensible programming languages, McMaster University Hamilton, Technical Report No. 71—7, 143 pp.

(Received Jan. 26, 1984)