SCYLA: SPEECH COMPILER FOR YOUR LANGUAGE (*)

S. Lazzaretto(**), L. Nebbia(**)

ABSTRACT

The rules on which a text-to-speech system is based are often hard to be checked, since the programming language used is far away from phonetic and linguistic conventions. Moreover, the program is usually written in a user dependent style and, as a consequence, it is not very readable. Some of these problems could be solved by a speech synthesis oriented programming language, as close as possible to linguistic terminology. The high-level programming language described in this paper has exactly these features, since it is expressly designed for the development of text-to-speech conversion rules and, in addition, the final output of the system is not an object but a source program. One of the aims of the project is to provide a flexible software tool as to be able to deal with rules for linguistic processing at various levels (e.g. phonetic transcription, word stress assignment, phonological phenomena, prosodic control, etc.) up to the acoustic level. Nevertheless, though the language is text-to-speech oriented, it could be used in all kinds of text processing involving contextual rules, since input and output data structures are completely user-definable.

INTRODUCTION

SCYLA is a high-level programming language expressly designed for the development of text-to-speech rules, and, generally speaking, for linguistic processing involving contextual rules at various levels (e.g. phonetic transcription, prosodic control, phonological phenomena, word and phrase stress assignment, etc.) The name "SCYLA" stands for "Speech Compiler (for) Your Language"; nevertheless the system is based on a translator (rather than a compiler) which reads a SCYLA program and produces a "C" source program. This is not the first attempt to define a language expressly designed for automatic text-to-speech conversion: the same idea was approached for example by the MIT group (KLATTalk Klatt -1982) (ref. 1), the Royal Institute of Technology in Stockholm (Carlson Granstrom 1976) (ref. 2), S. Hertz (SRS 1979 DELTA 1982) (ref. 3), the Institute of Phonetics and Telecommunications Research Laboratory of Copenhagen (SPL Holte Olsen 1982) (ref. 4). Nevertheless we think that SCYLA has some specific features, which make it different from other languages, for example the very simple and straightforward way in which the rules can be written and the portability of SCYLA's output to any processor supporting "C" language. This can be useful when moving the synthesis system from the development stage to a µP based system. SCYLA can work with multi-level structures, that is it can manipulate high level units (e.g. phrases, word, syllables) and low level units (e.g. diphones, phonemes or their fragments) at the same time. Linguistic levels, which rules refer to, are completely user-defined. In the same rule it is possible to switch automatically from a level (e.g. graphemes)

(*) Work partially supported by EEC contract in ESPRIT project n.64 "SPIN".

(**) CSELT SpA - Via G. Reiss Romoli, 274 - 10148 TORINO - ITALY
to another (e.g. morphs). The synchronization of different levels is entirely user-transparent.

DESCRIPTION OF SCYLA LANGUAGE

The first part of a SCYLA program necessarily consists in a declaration block, in which all levels have to be entirely specified. This "level" specification consists of a "name-statement", that is the list of the tokens belonging to that level, and one or more "attribute definition statements", for creation of subsets which can be referred to within the rules. The following example shows a typical declaration of "grapheme" level for the Italian language:

```
level grapheme;
  name: a,b,c,d,e,f,g,h,i,j,l,m,n,o,p,q,r,s,t,u,v,z,
  " ",",","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","",
  letter(a,b,c,d,e,f,g,h,i,j,l,m,n,o,p,q,r,s,t,u,v,z);
  vowel(a,e,i,o,u);
  punctuation_mark( ",",",",",","",","",","","");
end grapheme;
```

In the example above (1), three subsets (letter, vowel, punctuation_mark) are defined. Rules can refer to all tokens belonging to a subset, simply with a reference to the name of the subset. In all programs produced by SCYLA, a quite simple lexical analyzer scans the input text from left to right in order to separate tokens and to detect input errors, such as undefined tokens. The second part of a SCYLA program consists of one or more procedures in the following form:

```
proc <name>(<input_level>,<output_level>)[<qualifier>];
  <rule>
  end <name>;
```

There are two modes in which the rules can be tested and applied, normal mode and cyclic mode. In normal mode, all the rules are tested, until the first applicable rule is encountered; all the remaining rules are ignored and a new token is read from input text until end of text is detected. In cyclic mode the whole text is processed and each rule is applied to each occurrence of the token it refers to; then the output is swapped with input and the process continues until no more changes are made. A typical rule can occur in the following form:

```
<input_sequence> -> <output_sequence>
  /<left_context>...<right_context>
  /<left_context>...<right_context>
  ....
  -> <output_sequence>
  /<left_context>...<right_context>
  /<left_context>...<right_context>
  ....
  -> ....
```

If the input sequence (one or more elements belonging to the input level) is found in the input text, the body of the rule is executed,
otherwise the rule is skipped away. The output sequence (one or more elements belonging to the output level) is written into output if at least one specified context occurs, otherwise the next set of contexts is tested, until at least one context matches the input text. The context for the last output sequence is optional: if not present, the sequence is intended as the one to be written when no contexts occur. For example:

\[
\begin{align*}
\text{a} & \rightarrow \text{b} \\
/\text{c} & \ldots /\text{d} \\
\rightarrow & /\text{e};
\end{align*}
\]  

(4)

means that "a" is translated into "b", if preceded by "c" and followed by "d", otherwise it's translated into "e". More complex rules are also possible, by using several kinds of logical structures, like "or", "and", "not", for example:

\[
\begin{align*}
\text{a} & \rightarrow /\text{b} \\
\{(\text{a}, \text{b}, \text{c}, \text{d})\} & \ldots [\text{vowel}, !\text{u}] \\
/\text{s} & \ldots /\text{consonant} \\
/ & \ldots /\text{h} \\
\rightarrow & /\text{d} \\
/\text{e} & \ldots /\text{d} \\
\rightarrow & /\text{g} \\
\end{align*}
\]  

(5)

Braces mean logical "or", square brackets mean "and", and exclamation mark means "not"; rule (5) means that the sequence "a" "b" has to be translated into "c" or into "d" or into "g" depending on its context. Going into detail, the first left context is "a", or the sequence "b" "c", or "d" \{(a,b,c,d)\}; the right context is an element of the subset "vowel" but not "u" \{[vowel],!u\}. The second context is "s" on the left, and a "consonant" on the right, while the third context requires only "h" on the right. If none of those context occurs, the rule checks if "d" is the correct output sequence, looking for "e" on the left, and "d" on the right. If this test fails too, the correct output sequence is "g". Rules within a procedure are applied in order of length of the sequence they refer to, independently of the order in which they are written, unless the qualifier "nosort" is specified. In our examples, rule (5) would be applied before rule (4), even if (4) is written first, because (5) refers to a two elements input sequence ("a" "b"), while (4) only refers to one element ("a"). It is possible to switch automatically from a level to another, in the same rule, with no synchronization problems; for instance:

\[
/\text{a} <\text{phoneme "b:"}>\ldots /\text{c} <\text{morpheme root}>
\]  

(6)

Left context refers to "a" in the current input level, followed by "b:" in the phoneme level; right context refers to "c" followed by "root" in the morpheme level. Note that an element has to be intended as belonging to the input level, if no other specifications are given; any part of left/right context enclosed in "<<" and ">>" has to be found in the level which is specified immediately after "<<". The complete syntactic specification for the rules available in SCYLA environment is too complex to be reported in this paper; nevertheless some remarkable features will be described in the following. SCYLA rules can also have many other forms, i.e. blocks of rules to be activated by a specified pre-condition. The system allows the mixing of SCYLA and C source statements, therefore "C"
pre-defined or user-defined functions and procedures are available. SCYLA can deal with "numerical" levels too, either integer or float; this can be useful in the treatment of numerical information, such as synthesis parameters (duration, pitch, etc.) A flexible dictionary for exception handling is presently being developed. In order to speed rule development and tuning, a powerful debugger has been implemented. This software tool, extremely useful especially when the number of rules is large, allows step by step execution and interactive display of each level and its runtime changes. It can also display the portion of the source listing containing the applied rule, and all rules partially matched.

REFERENCES


