ARIPUC 19, 1985, p. 1-42

SPL: A SPEECH SYNTHESIS PROGRAMMING LANGUAGE

Peter Holtse and

Anders Olsen*

This report describes the first version of a high level computer programming language for experiments with synthetic speech.

In SPL a context sensitive parser is programmed to recognize linguistic constructs in an input string. Both the structural and phonetic descriptions of the recognized structures may be modified under program control. The final output of an SPL program is a data stream capable of driving a parametric speech synthesizer.

The notation used is based on the principles known from Chomsky and Halle's "The Sound Pattern of English". This means that in principle all linguistic constructs are programmed in segmental units. However, in SPL certain macro facilities have been provided for more complicated units such as syllables or words.

1. INTRODUCTION

A special programming language for experiments with phonological rules in general and speech synthesis in particular is currently being tried out in our laboratory as part of a project to develop a Danish text-to-speech synthesis system (cf. Holtse(1982)).

---

*The Telecommunications Research Laboratory, Copenhagen

The first part of the present paper describes in broad terms the philosophy and concepts of the programming language while the last part contains the formal syntax definitions.

The idea of coding phonological rules directly into a digital computer is not new, of course. In some way or other special programming tools have been developed in most laboratories working with speech synthesis by rule. Thus, various rule testing programs have been described, e.g. Bobrow and Fraser (1968) and Basbøll and Kristensen (1974, 1975). Furthermore, several compiling and/or interpreting systems have been developed, for instance Hertz (1982), Carlson and Granström (1974), and Kerkhoff et al. (1984). These systems allow rules to be formulated in notations very like the style commonly used by linguists and phonologists.

While the early rule programs were mainly concerned with creating an environment for testing phonological rules, the later systems have tried to include facilities for direct manipulation of speech synthesizer control parameters via rule statements. Thus, the system described by Hertz (1982) is a complete interactive synthesis development system including a rule interpreter whereas Carlson and Granström describe a compiling language which can produce an entire text-to-speech conversion system.

The present paper defines a programming language (SPL for Synthesis Programming Language) very much like the one described by Carlson and Granström. In fact the whole project owes much to their ideas and experience. The language is, like most languages of this kind, based on the notation introduced by Chomsky and Halle (1968) in *The Sound Pattern of English*. The choice of this - maybe outdated - notational representation is not based on a firm conviction of the superiority of segmental descriptions over other more sophisticated ways of describing speech. But the segmental approach has the merit of being relatively widely accepted by linguists and phoneticians as one possible way of expressing phonological regularities, even if it is rarely the most elegant way. Also, since it is possible - with a bit of fiddling - to describe most phonologically relevant entities in terms of segments this approach is quite attractive as a basis for a general programming language so as to avoid hardwiring too many definitions into the language. And finally, the single level description is relatively easy to implement on a digital computer.

1.1. SCOPE OF THE LANGUAGE

SPL is intended as a tool for expressing regular phonological and phonetic rules. It could be used to write part of a text-to-speech system, or part of the hypothesis verification component(s) of a speech recognition system. Or it could be used directly for phonetic or phonological research.

Since an SPL program will accept normal orthography as input and is capable of directly driving a parametric speech synthesizer an entire text-to-speech system could, in principle, be coded in SPL. However, certain problems normally encountered in such systems are better dealt with in special orthographic preprocessors. Thus, since only regular expressions can be formulated in SPL single word exceptions to general rules or exceptionally weird orthographies are relatively expensive seing that they may need an entire rule to handle each word. Also, abbreviations, acronyms etc. can probably be dealt with more efficiently by general text preprocessors, although they may, of course, be handled via rules. Molbæk (1982) contains a detailed discussion of various strategies for handling these problems.

Finally, it should be noted that the SPL is not well suited for parsing larger linguistic units - primarily because it has no facilities for dictionary look up and cannot deal with even the most rudimentary kind of semantic information. This deficiency may turn out to cause difficulties with e.g. sentence stress in certain languages.

2. SURVEY OF THE LANGUAGE

This section gives a brief survey of the structure and concepts used in an SPL program. The chapter is intended as an informal description of the language - not a programmer's or user's manual. The complete formal definition of the SPL syntax is included as a separate section.

2.1. PROGRAM STRUCTURE

An SPL source program consists of two major components: A data declaration component and a rule description component. All data structures needed within an SPL program must be explicitly defined before they may be used either in rules or in other data declaration statements, i.e. forward references are not allowed. In the rule component the rules are described which transform an input text string first to a string of phonetic symbols and then to synthesis control parameters.

The actual transformation of a text string is effected in three stages:

During Stage One the input characters are translated to Phones - the internal representation used throughout the rule program. Then the Phones are loaded into a work buffer until all the characters of a complete sentence have been entered.

In Stage Two the rules of the rule component are applied in succession to the string of Phones in the work buffer. Each rule is applied to all the Phones of the buffer in a left to

right fashion before the next rule comes into play. The rules may add additional Phones to the buffer or delete Phones from the buffer, or they may change the descriptions of the Phones in the buffer. In this way the contents of the buffer is gradually changed into a more and more detailed description of the utterance originally entered as text input.

When all the rules have been applied, the buffer should contain the equivalent of the acoustic segments of the utterance, and the program enters Stage Three. During this stage the segmental chunks now contained in the work buffer are interpolated and reformatted, and a file is output which contains the control code necessary to make the target speech synthesizer produce the utterance in question.

2.2. DATA TYPES

SPL attempts to impose as few restrictions as possible on the way a user can describe his linguistic theory. Therefore, the language contains no built in notions of what, for instance, a syllable or a word should look like. The only predefined phonological units within SPL are Distinctive Features and Phones, i.e. the system is basically segmental. However, quite complicated segemental sequences may be described via *structure types* (q.v.) and later referred to as syllables of various kinds etc. so that, to some extent at least, the limitations of using a segmental environment are removed. The important point is that the proper definition of such units is left entirely to the user/programmer.

2.2.1. Features, Scalars, Parameters and Phones

The basic unit within an SPL program is the *Phone* which at the input end is a segmental entity roughly corresponding to a letter or a phonetic symbol. During Stage Two, application of the rules, this description is gradually refined so that at the output stage each Phone corresponds to a separate acoustic segment. Thus, an aspirated stop will typically consist of three Phones at the output stage: closure, explosion, and aspiration.

A Phone consists of a structural part and an optional segmental part. The structural part serves to describe primarily the phonological properties of the Phone, while the segmental part contains a description of the physical properties associated with the realisation of the Phone.

The structural properties of Phones are described in terms of *Distinctive Features* (or just Features). Features are binary entities which assume values of *plus* or *minus* to indicate the presence or absence of a certain property within the Phone. Examples of Features are *consonantal, vocalic, syllabic,* or *labial*. For instance, the consonants

b, *p*, or *m* might all be classified as [+labial] to indicate that they belong to the class of consonants articulated with lip closure.

Features are combined to form Matrices, e.g. [+voc, -cons, +syll]. Note, however, that while feature matrices in segmental phonology are traditionally written in columns, SPL Matrices are written in a linear fashion in reverence to the limitations imposed by most computer text editors.

Each Phone is, in principle, defined by a unique matrix of Distinctive Features. However, the values of certain Features may be irrelevant to a particular Phone. For instance, the value of the Feature "stress" could in some cases be considered irrelevant to consonants since stress may in some connections be regarded as a property associated with vowels. In cases like this the values of the irrelevant (or redundant) Features may be left undefined.

Certain properties of Phones cannot conveniently be expressed as binary values. To cope with these situations each Phone has associated with it a list of *Scalars*, which may be thought of as multivalued Features. The Scalars are, however, purely descriptive labels. They are not considered part of the definition of the Phone as such, i.e. two Phones may share the same combination of Scalar values, whereas each Phone must have a unique combination of Distinctive Features. *Duration* and *height* are examples of properties which could be expressed via Scalars. Technically, Scalars are integer variables capable of assuming the values of all integer numbers as defined by the implementation of the SPL compiler.

There are two classes of Scalars within an SPL program. The first class comprises the two predefined Scalars *DUR* and *RANK*. DUR specifies the duration of the Phone in time (expressed in milliseconds), while RANK is a control value used when the Phones are concatenated in the final output.

The second class comprises any user defined Scalars. The user defined Scalars have no direct influence on the physical characteristics of the output from the synthesis program. They may be used, as previously mentioned, to express multivalued structural conditions which can only with difficulty be expressed in binary distinctive features.

Both types of Scalars may be used in relational and logical expressions as part of phonological descriptions and conditions.

The actual acoustic phonetic realisation of the Phone is described in a table of *Parameters*.

Parameters are the physical control variables of the speech synthesizer which is eventually to use the output of the SPL program. The primary property of each Parameter is its

target. The target value is an integer number which indicates, for instance, the frequency of a certain formant or the amplitude of a gate. Each Phone contains one target value specification for each Parameter associated with the speech synthesizer in question. Additionally, two transition times are associated with each target value: An internal and an external transition time. The transition times are dynamic properties of the Phone. They specify the speed with which the target values should be reached during execution of the synthetic utterance. (Holtse(1974) contains a more detailed description of the general strategy used during interpolation of parameters.)

Scalars or Parameters which need to refer to the same integer value in many places within a program may do so via a *Constant* reference. A Constant is an integer with a name to it.

A Phone may serve only phonological purposes and therefore have no physical realization of its own. In such Phones the Scalar and Parameter specifications need not be supplied. These Phones are known as *Pseudo Phones*.

The following is an example of the code needed to define a vowel named "alpha":

```
feature cons, voc, high, low, back, round .......
.
.
phone <alpha [Sa2033Q]>
        [-cons, +voc, +low, -high, +back, ..... ]
        F1  450, F2 1100, F3 2800, A0 30
```

The first name in the triangular brackets defines the name by which the phone will be known in any following rules in this source module. The character string in the first square bracket is an alternative name. This string will be printed during debugging instead of the internal name. In this way it is possible to code the source of the rule program using an ordinary text editor on any dumb terminal while the final synthesis program will be able to drive a rather more sophisticated terminal by taking advantage of, for instance, a phonetic character generator.

The matrix of binary features is the unique definition of the phone. And the last line is a description of the parameter default target values of the phone.

2.2.2. Structures

The Structure is a special descriptional aid which has been provided to circumvent the basically segmental nature of SPL. In principle it is simply a sort of short hand for a more or less complicated sequence of Phones. Thus, the segmental setup of, for instance, a syllable need only be

declared once. From then on the declared name will automatically be expanded to the complete segmental syllable description every time the structure name occurs.

For example the following fragment of SPL code is one way of handling syllables:

```
feature cons. voc
.
.
phone <V [V]>  [-cons. +voc]
phone <C [C]>  [+cons. -voc]
.
.
structure  S  (C <0.3>) V (C <0.5>)
```

The first line declares the two names *cons* and *voc* to be of the type Feature. Then *V* and *C* are declared to be the names of two phones with the feature matrices contained in the square brackets. And finally *S* is declared to be a structure consisting of a vowel with from zero to three initial consonants and from zero to five final consonants. The expressions within the triangular brackets define the number of instances of the entity which are acceptable at that place.

Structure definitions may be used recursively so that the defintion above could be used in the declaration:

```
structure  W  (S<>) #
```

to declare that a word. *W*. is any number of syllables terminated by a word boundary symbol (which must, of course, also be defined as a Phone or a Structure). Please note that in the example above it is not necessary to compute the exact location of the syllable boundaries since all that is needed for the description to work is the "top" of each syllable.

2.3. DESCRIPTIONS OF TRANSFORMATIONS

Transformations are described in a context sensitive grammar and formulated in Rule Statements. Each Rule Statement contains a command word, a structural description of the string to be transformed: the Rule Kernel, a description of the context(s) in which the Kernel must occur for the Rule to apply, and a description of the changes to be made.

For example the rule:

```
change :  V / ["C", +lab] _____ # -> [+round]
```

could be one way of formulating that final vowels are rounded after labial consonants. As the example shows the Rule Kernel is separated from the Rule Context by a slash

while the place of the Kernel within the Context is indicated by an underline. The right arrow points to the changes to be made.

Space, tab, newline and form feed characters may be inserted anywhere to improve readability. Thus, the whole rule may be written on one line or newlines and tabs may be used to provide special visual effects as in the example below:

```
change :    V       /    ["C", +lab] ___  #
     ->  [+round]
```

The rule above will cause the synthesis program to find any occurrences of *V*, which presumably is a Pseudo Phone defined to match any vowel within the language being synthesized. Once a vowel has been located, the precontext, i.e. the context immediately preceding the Kernel, is scanned in reverse direction. In this example the precontext contains only one unit: *C*, which is probably any consonant within the language. Furthermore, the restriction is added that only *+lab* consonants are accepted.

When the precontext has been accepted the postcontext is scanned. The postcontext in this example consists of the (presumably) Pseudo Phone # - the usual sign for a word boundary.

If the structural descriptions of both pre- and postcontexts are matched the phone is changed as described in the last part of the statement. In this case the feature value *+round* is assigned to the vowel matching the Kernel, while all other feature values for that vowel are left unchanged. Furthermore, if the Distinctive Feature *round* is currently undefined for that vowel it will be marked as defined.

The Structures described above may be used to recognize more complicated conditions. For instance the rule:

```
changeall : S / ___   (S <2,2>)  #  -> [+stress]
```

would, using the definitions from above, add the value *+stress* to all segments of the last syllable but three in a word of three or more syllables.

Modifications of acoustical descriptions may be programmed as in the following example:

```
change :  ["V", +stress]
      /   _____          (S <num_syll= 0, 8>)   #

   ->    [(DUR = v_min +
                ( (DUR - v_min) / (num_syll+1) )
         )]
```

This fictitous rule will cause the number of following syllables in the current word, as previously defined, to be

evaluated and placed in the variable 'num_syll'. The duration (DUR) of the stressed vowel just recognized will be set equal to the sum of the minimal vowel duration allowed (v_min as defined by the user) plus a correction component depending on the number of succeeding syllables in the word. The correction component is computed as the difference between the inherent duration of the vowel and the minimal vowel duration divided by the number of succeeding syllables as computed above. (num_syll is incremented by one before the division to avoid dividing by zero in words with stress on the last syllable.)

Consider finally the rule:

```
change :  ["V", -stress]
          / ["V", +stress]  (C <num_c=0.7>) _____

          -> [(F1 += 0.1 * F1(-num_c-1))]
```

which causes a post tonic unstressed syllable to approximate the quality of the stressed syllable.

First, an unstressed vowel following a stressed vowel with from zero to seven intervening consonants is recognized, and the number of intervening consonants is placed in the variable *num_c*. Then the frequency of the first formant of the stressed vowel is obtained (F1(-num_c-1)): One segment further to the left than the number of consonants found. This frequency is multiplied by 0.1 and finally added to the frequency of the first formant of the unstressed vowel.

2.3.1. Rule Types

Various types of rules are recognized in SPL. In the examples above the difference between *change* and *changeall* rules has been shown: In an ordinary change rule each segment in the Change Field applies to a corresponding segment in the Kernel, while in a changeall rule the modifications described in the Change Field apply to all the segments of the Kernel - irrespective of the number of segments contained within the Kernel.

A third type of rule is the *replace* rule which has the form:

```
replace :  x y z  /  A  ___  B  ->  w q t
```

This type of rule will, under the conditions specified, replace the entire sequence of segments of the Kernel, irrespective of whether they are described in terms of Structures or Phones, by the sequence contained in the Change Field.

Thus, the change rule is used to modify the values of existing Phones in the buffer while the replace rule is used when entire Phones are to be substituted. Also, in a replace rule

the number of Phones may differ in the Kernel and Change Fields so as to allow deletion and addition of Phones from the buffer.

Special cases of the replace rule are the *delete* and *insert* rules which are of the form:

delete : x y z / A ___ B ->

and

insert : / A ___ B -> x y z

These two types of rules require special command words as shown in the examples in order to improve error diagnosis.

2.4. INPUT CONTROL

Since the only unit recognized within an SPL program is the Phone any ordinary characters input to an SPL coded synthesis program must immediately be translated into an appropriate string of Phones. This translation is controlled via *graph statements.* The graph statement is of the form:

graph a : a1

which means that when the compiled program meets the character *a* in its input stream it must be translated to the Phone *a1* which must be a properly defined Phone or Pseudo Phone.

Alternatively, input may be complete matrix and parameter tables obtained from another SPL program. This facility allows the different phases of a complete rule system to be coded in independent programs in order to facilitate debugging.

2.5. OUTPUT CONTROL

Output from an SPL program may be provided in two ways. Either via a *print* command or via a *speak* command. The print command will cause the current contents of the buffer, including all feature matrices and parameter tables, to be output to the designated output stream. From here it may be redirected to a terminal or other printing device for inspection, or it may be used as input for another SPL program as explained above.

The speak command immediately causes the parameter tables of the work buffer to be interpolated. Interpolation is performed using a strategy very similar to the one described by Holtse (1974). This data stream is in a format acceptable for a parametric speech synthesizer.

2.5.1. Speech Synthesizers

SPL as such makes few assumptions about the type of synthesizer for which it is producing output and it may in fact be configured for a wide variaty of synthesizers - hardware or software implementations. Furthermore, SPL will recognize all the more usual parameter names, at least for synthesizers of the formant type, but it will, of course, only produce code for the synthesizer for which it is actually targeted. In the current version of SPL a separate compiler must be produced for each target synthesizer. This may, however, be changed.

2.6. PROGRAM DEBUGGING

SPL includes a *trace* facility which, when turned on, will print the output of any rule which has applied successfully. With proper use of the alternate phone representations, as mentioned in the section dealing with Phones, quite a detailed view of the actions of the synthesis program may be obtained.

3. IMPLEMENTATION

A first version of an SPL compiler has been developed as a joint effort between the Institute of Phonetics and the Telecommunications Research Laboratory, both of Copenhagen.

Currently, most of the defined facilities of the compiler and its corresponding run time system have been implemented on the VAX-11/750 computer under a VMS operating system at the Telecommunications Research Laboratory and on the PDP-11/60 computer under a UNIX operating system at the Institute of Phonetics. Our intentions are to keep the two versions as closely compatible as possible.

Also, various support facilities have been developed such as a special parameter editor which will allow very detailed interactive control of the synthesis control parameters. This is found to be a necessary tool for the development of proper Phone descriptions. Furthermore, special device drivers capable of producing phonetic script and detailed plots of control parameter traces are being developed.

Finally, driver tables for different speech synthesizers are under construction.

FORMAL DEFINITION OF SPL

First a note of warning: Readers with a linguistic or phonetic background should observe that in this report words like *syntax* or *semantics* and their derivatives will refer to the syntax or semantics of the SPL language - *not* the syntax or semantics of any natural language.

4. NOTATION, TERMINOLOGY, AND VOCABULARY

The grammar of SPL is described in an adapted sort of Backus-Naur form. Thus, non-terminal constructs are denoted by English words surrounded by angular brackets: < and >. Terminal symbols are written in **bold** characters.

The production rule for non-terminal symbols consists of the non-terminal symbol itself followed by the symbol ::= (two colons and an equal sign). After this follows one or more terminal or non-terminal symbols.

Repetition of constructs is indicated by curly brackets: { and }. Alternative productions are separated by a vertical rule: |. Constructs surrounded by square brackets are optional.

The symbol <empty> denotes a sequence of zero symbols.

Non-terminal symbols may include an underlined part. The underlined part is an indication of a semantic subcategory - not part of the context-free syntax description. (For instance <feature name> means a <name> of the Type "Feature".

4.1. VOCABULARY

SPL programs are represented using the full set of printing ASCII characters as follows:

<letter> ::=

A | B | C | D | E | F | G | H | I | J | K | L
| M | N | O | P | Q | R | S | T | U | V | W |
X | Y | Z | a | b | c | d | e | f | g | h | i
| j | k | l | m | n | o | p | q | r | s | t |
u | v | w | x | y | z

<digit> ::=

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special character> ::=

any printing ascii character not mentioned above!

The following characters and character sequences are reserved symbols with special meaning to the compiler. They may not be redefined by the user:

<reserved symbol> ::=

feature | phone | integer | scalar | real | constant | structure | graph | change | changeall | replace | delete | insert | obl | opt | tg | ti | tx | trace | on | off | print | speak | file | include | identity | -> | /*

4.2. NAMES AND CONSTANT NUMBERS

Names denote variables, features, scalars, functions, constants, output units, identifiers, or parameters. The special class of Names used to identify Phones and Structures within rule formulations is known as Unitnames. Their syntax is identical to the syntax of ordinary Names, except that a Unitname may also contain special characters.

Each Name or Unitname must be unique and the reserved symbols previously mentioned may not be used.

<name> ::=

<letter> { <letter> | <digit> | _ }

<unitname> ::=

any sequence of printing ascii characters not containing the characters **(** *(left bracket).* **)** *(right bracket).* **[** *(square bracket begin).* **]** *(square bracket end).* **/** *(slash).* **<** *(angular bracket begin). or* **,** *(comma).*

Furthermore, a Unitname may not consist entirely of underline characters.

Examples of Names:

```
tot_dur
tempo
wb2
```

Examples of Unitnames:

```
p_asp
p
#
!8
t*
:
```

Numbers are decimal numbers. SPL supports integer and real constant numbers in the usual notation.

<integer> ::=

```
<digit> { <digit> }
```

<real> ::=

```
<integer>.<integer>
| .<integer>
| <integer>.
```

Examples of Integers:

```
117
2467
```

Examples of reals:

```
0.0
123.
117.99999
```

Lexical entries are delimited by the first character which is not a legal part of the entry.

4.3. CONSTANTS

Certain frequently used integers may be declared constant and referred to by name instead of the usual sequence of digits.

<constant> ::=

constant <name><integer> { , <name><integer> }

A constant declaration consists of the reserved word *constant* followed by a Name and an Integer.

4.4. SPACES AND COMMENTS

Any number of white spaces, i.e. space, tab, or newline characters, may be inserted between lexical entries to improve readability and to separate entries which would otherwise flow together and cause syntactic ambiguities.

Comments are surrounded by sequences of **/*** and ***/**. Anything between (and including) these two symbols will be treated as a sequence of white spaces by the compiler.

5. PROGRAMS

A program consists of three parts: A data declaration part, an input part (the character conversion), and the program body which contains the Rule Statements and Auxiliary Commands.

<program> ::=

<data declaration> <character conversion>
<program body>

<program body> ::=

{ <rule statement> | <auxiliary statement> }

A Statement may occupy as many lines as desired. Blanks, tabs, or newlines may be used as previously explained to improve readability.

5.1. COMPILER DIRECTIVES

Directives look like ordinary statements. However, they are commands controlling the workings of the compiler itself whereas ordinary commands are commands to be incorporated in the program produced by the compiler.

Directives may be placed anywhere within a program.

5.1.1. Include Directive

<include directive> ::=

include " <filename> "

The Include Directive causes the compiler to include the named file of source text as if it had been part of the original input file at that point. After having read the included file, input will again be taken from the original file.

Filename may be any character string representing a legal filename within the operating system on which the compiler is implemented.

Include Directives may be nested to a reasonable depth.

5.1.2. Identity Directive

The Identity Directive is a sort of mock Data Declaration (q.v.), but it defines no new data structures.

<identity directive> ::=

identity <identity pair> { , <identity pair>}

<identity pair> ::=

(<name> , <name>) |
(<unitname> , <unitname>)

This directive causes the two names to point to the same data structure. The first name of a pair must be a previously defined Name or Unitname. The last Name of the pair will throughout the program be another way of writing the first Name.

Examples of Identity Directives:

```
identity (pluk, p)
identity (#, WB), (labial, lab)
```

6. DATA DECLARATIONS

Data types within an SPL program must, in principle, all be explicitly defined via a Data Declaration statement before they may be referenced in any other statement. However, certain types are predefined within the compiler. The predefined types are not part of the definition of the language as such, but they are implementation dependent

since they are mostly concerned with the interface between the programming language and a specific type of hardware speech synthesizer.

<data declaration> ::=

{ <basic type declaration> }
{ <complex type declaration> }

6.1. BASIC TYPES

<basic type declaration> ::=

<basic type identifier> <name> { , <name>}

<basic type identifier> ::=

feature | scalar | real | integer

Examples of Basic Type Declarations:

feature front, back, high, low
scalar height
real tempo
integer contour

6.1.1. Features

Features are binary entities capable of assuming the values plus or minus to designate the presence or absence of a certain property in each Phone. Declaring a new Feature causes no direct storage allocation but reserves space for one binary Feature in all Phones defined later in the program.

6.1.2. Scalars

Scalars are integer variables associated with the Phones defined later in the program. Declaring a Scalar name causes no direct storage allocation, but storage will be allocated in connection with all Phone declarations.

Two Scalars, *RANK* and *DUR* are predefined within the compiler and cannot be redefined.

6.1.3. Reals

Reals are variables which may hold any real valued number within the range defined by the implementation.

6.1.4. Integers

Integers are variables which may hold any whole number within the range defined by the implementation.

Storage for Reals and Integers is allocated as they are declared.

Reals and Integers are known collectively as *variables* or - since they are accessible from all parts of a program as - *globals* or *global variables.*

6.1.5. Parameters

The fifth basic data type is the Parameter. Parameters are the physical control variables of the target synthesizer. Since, at least in the current version of SPL, a separate compiler must be generated for each target synthesizer, the Parameter Names are predefined in the compiler and cannot be changed by the user program (although synonyms may be created through Identity Directives).

However, since it is also desirable to allow the same SPL source code to be compiled for different target synthesizers, a rather generous supply of Parameter Names are known beforehand to the SPL compiler - irrespective of the actual target synthesizer. All such predefined Parameter Names will be accepted by any SPL compiler, i.e. no message of "unknown identifier" etc. will be produced. However, only the Parameters that are physically present in the target synthesizer will be reflected in the compiled object module. Non-active Parameters may be redefined so as to produce an identity between active and non-active Parameters in order to avoid having the SPL compiler ignore the statement containing the non-active Parameter.

The following Parameters are known to all SPL compilers:

F0 - Pitch or fundamental frequency of voice source.
F1 - Frequency of first formant.
L1 - Amplitude of first formant.
B1 - Bandwidth of first formant.
F2 - Frequency of second formant.
L2 - Amplitude of second formant.
B2 - Bandwidth of second formant.
F3 - Frequency of third formant.
L3 - Amplitude of third formant.
B3 - Bandwidth of third formant.
F4 - Frequency of fourth formant.

L4 - Amplitude of fourth formant.
B4 - Bandwidth of fourth formant.
C1 - Frequency of first consonant formant.
C2 - Frequency of second consonant formant.
FN - Frequency of nasal formant.
BN - Bandwidth of nasal formant.
FZ - Frequency of spectral zero.
BZ - Bandwidth of spectral zero.
A0 - Overall amplitude.
AV - Amplitude of voicing.
AS - Amplitude of sinusoidal voicing.
AH - Amplitude of hiss noise.
AF - Amplitude of fricative noise.
AN - Amplitude through nasal branch.
AB - Bypass path amplitude.
VO - Voicing switch.

This allowance of Parameter names ought to allow communication with the more usual formant synthesizers.

6.1.6. Characters

The set of all printable ASCII characters may be considered a sixth Basic Type. The character type cannot be declared, however, since it is an existing and closed corpus. Furthermore, characters may only appear in character conversion statements within an SPL program, i.e. they are removed as soon as they are brought into the program.

6.2. COMPLEX TYPES

The complex types are data types made up of combinations of other types. The two complex types are *Phones* and *Structures*.

<complex type declaration> ::=

<phone type> |
<structure type>

6.2.1. Phones

A Phone consists of two or three parts: A name, a matrix of Distinctive Features and an optional segmental part. A Phone without segmental description, i.e. without any direct acoustic manifestation, is known as a *Pseudo Phone*.

<phone type> ::=

phone <p-name><matrix>
[<segmental description>]

⟨p-name⟩ ::=

< ⟨unitname⟩ [⟨script⟩] >

⟨matrix⟩ ::=

[["⟨phone unitname⟩,"] ⟨feature bundle⟩]

⟨feature bundle⟩ ::=

⟨feature expression⟩ {, ⟨feature expression⟩}

⟨feature expression⟩ ::=

⟨feature value⟩ ⟨feature name⟩

⟨feature value⟩ ::=

+ | - | ?

⟨segmental description⟩ ::=

⟨segment field⟩ {, ⟨segment field⟩}

⟨segment field⟩ ::=

⟨scalar field⟩ |
⟨parameter field⟩

⟨scalar field⟩ ::=

⟨scalar name⟩ ⟨integer constant expression⟩

⟨parameter field⟩ ::=

⟨target definition⟩ [⟨transition definition⟩]

⟨target definition⟩ ::=

⟨parameter name⟩ ⟨target value⟩

⟨transition definition⟩ ::=

(⟨internal transition⟩,⟨external transition⟩)

⟨target value⟩ ::=

⟨integer constant expression⟩

⟨internal transition⟩ ::=

⟨integer constant expression⟩

⟨external transition⟩ ::=

⟨integer constant expression⟩

⟨script⟩ ::=

string of ascii characters

Examples of Phone Definitions:

```
phone <alfa[a2]> ["V". +low. -high. +back. -round]
        dur 10. rank 50. height 5.
        F1 650 (5.5).
        F2 1200. F3 2800. A0 60

phone <#[#0]> [-seg. +wb. +sylb]
```

The phone statement defines the properties to be associated with a given Phone. Each Phone has a Name (of the type Unitname) and a Script, i.e. phonetic transcription. The Name of the Phone is used in structural descriptions as an abbreviation for the complete feature matrix. For instance *p_asp* could be the Name of a special aspiration after [p].

The Script is a string of characters (including non printing characters in escape notation) to be used for debugging and other print out in symbolic form. It has no meaning to the internal workings of the SPL program but will be printed exactly as it is entered in the definition. This strategy allows the synthesis program to take advantage of any special character generators in printers or terminals while still retaining a measure of readability in the rule formulations.

The Feature Matrix is the combination of feature values which uniquely identifies that Phone. No two Phones may have the same combination of feature values.

The phone statement causes the Phone in question, together with the properties described in the statement, to be entered into the Symbol Table and the Phone Definition Table.

The data structure defined by of the matrix part of the Phone definition consists of two parts: A Definition Matrix in which the bit positions of the Features defined for that Phone are set and a Condition Matrix in which the bit positions of the Features having the value *plus* are set while Features having the value *minus* have their corresponding bits cleared.

When the matrix part of a Phone statement contains the name of a previously defined Phone the Definition and Condition Matrices are copied from that Phone and used as the basis for the new Phone. The special feature value *?* has the effect of *removing* a Feature from the definition of the Phone if it is already there. These facilities should save some typing efforts and errors in the definition part of the program.

When occurring in structural descriptions the Definition Matrix is used to mask out the undefined Feature positions so that only defined Features are matched to the input string. This strategy means that Features which are undefined for a given Phone cannot block the application of a rule.

The Segment Description defines the physical properties associated with the Phone. Thus each Parameter entry contains a target value and two transition times. The target is the frequency or amplitude to be reached during the Phone, while the transition times are the duration of the transitions external and internal to the duration of the Phone. (The duration of the Phone is contained in the Scalar *dur*.)

Transition times may be left undefined in a parameter field. In such cases they are by default set to zero.

Targets may be undefined for a given Parameter. In such cases the target value will be supplied from an internal default table depending on the target synthesizer.

While feature matrices may be "inherited" from previously defined Phones. Scalars and Parameter values must be explicitly declared for each segmental description.

6.2.2. Structures

The Structure concept is a sort of macro definition for commonly needed sequences of strings of Phones. Thus, Structure definitions may be used to simplify the formulation of complicated linguistic units such as syllables or words.

<structure type> ::=

structure <structure definition>
{ , <structure definition> }

<structure definition> ::=

<unitname> <structural sequence>

<structural sequence> ::=

{ <structural unit> |
(<structural sequence> <range expression>) }

<structural unit> ::=

<phone unitname> |
<structure unitname> |
<matrix>

Examples of Structure Definitions:

```
structure S (C<0.5>) V (C<0.5>)
structure W (S<ntsyl=1.>) #
```

The Structure definition statement consists of the reserved word *structure* followed by the description of one or more structures. Each description consists of a name and a listing of the units which make up the Structure. The elements of a Structure are Phones, feature matrices or other Structures.

The structure definition statement causes the list of elements for each Structure to be entered into the Structure Definition Table. In later rule formulations this list of structural descriptions are invoked every time the name of the Structure is used, i.e. it is a sort of macro facility for expressing commonly used complicated conditions. Technically, however, the Structure is expanded at compile time and therefore may not contain undefined or forward references.

The Range Expression is described in the chapter dealing with *Expressions.*

7. CHARACTER CONVERSION

Input to an SPL program is any string of ASCII characters. Internally in the program all operations are carried out on Phones - not on characters. The graph conversion rules

define what the SPL program must do with the input ASCII characters when they are encountered in the input stream.

<character conversion> ::=

graph <character-phone map>
{ , <character-phone map> }

<character-phone map> ::=

<character> : <phone unitname>

Examples of Character Conversion Statements:

```
graph a : a1, A : a1
graph b : b_luk, B : b_luk
```

Each character-phone map defines a unique conversion from a given ASCII character to a previously defined Phone. Upper and lower case characters are different identities. Two different characters may be mapped to the same Phone. It is an error to map the same character to two different Phones.

Input characters which are not mapped to Phones are deleted from the input stream, i.e. they cannot be accessed within the program.

8. RULE STATEMENTS

Changes to the contents of the Work Buffer are made via Rule Statements. Each rule statement describes a set of conditions under which the Phones currently in the Buffer are modified. The changes may be deletions or additions of Phones or they may be modifications to the properties of the Phones already residing in the Buffer.

<rule statement> ::=

<rule command> <rule head> : <context field>
-> <change field>

A rule statement consists of a command word indicating the Rule Type followed by a Rule Head which is terminated by a colon. Then follows a description of the structural context to which the Rule applies and a right arrow pointing to the description of the changes to be made to the Buffer.

8.1. RULE TYPES

There are five types of rules in SPL.

<rule command> ::=

change | **changeall** | **replace** | **insert** | **delete**

Change Rules are used to modify the current contents of one or several Phones already residing in the work buffer of the synthesis program. This type of rule must state explicitly how many Phones are affected by the modification and how each Phone affected is to be modified.

Changeall Rules are used to apply the same modification to a whole family of consecutive Phones in the work buffer.

Replace Rules replace one or several Phones in the work buffer with a sequence of Phones obtained from the definition tables.

Insert Rules are used to enter additional Phones into the work buffer. The Phones inserted are taken from the definition tables.

Delete Rules are used to remove one or more Phones from the work buffer.

8.2. THE RULE HEAD

The Rule Head contains two fields, both of which may be empty: A Label and a Type Declaration.

<rule head> ::=

[<rule label>] [<rule type>]

<rule label> ::=

<rule class>.<rule number>

<rule class> ::=

<integer constant>

<rule number> ::=

<integer constant>

<rule type> ::=

(<type indicator>)

<type indicator> ::=

obl | opt

Examples of Rule Commands:

```
change 26.5(opt): ........
changeall 5.2: .........
insert (obl): .....
delete : ......
```

8.2.1. Rule Label

The Rule Label is used entirely for debugging purposes during program development. Thus the class and number digits are printed out every time the Rule applies successfully to a form and the Trace function is turned on. The specific numbers used have no meaning to the SPL program as such and need not be unique.

8.2.2. Optional and Obligatory Rules

SPL rules are either Optional or Obligatory. If the Type field is empty the rule is Obligatory.

Generally, the changes described in the Change Field of a Structural Rule are applied to any form which matches the structural conditions given in the Context field of the rule - This is an obligatory rule. If the Context Description of an *Optional* rule matches the input string the current state of the Work Buffer is saved in a special storage area - core or disk as the implementation prefers. The Optional Rule is then applied to the Work Buffer in the usual way, and execution continues as usual. When all the rules of the program have been applied the SPL run time system retreives the saved buffer version from its storage and applies all commands after the Optional Rule that caused the diversion. Thus two versions of the same input string are created: One with the effect of the Optional Rule included and another without.

Since every Optional Rule of a program may in principle cause a split of the Buffer this facility is not aimed at production versions of talking machines. Primarily, it is a research tool for trying out new rules. However, the ability to create several versions of the same utterance will

also be needed for hypothesis building within Automatic Speech Recognition algorithms.

8.3. CONTEXT DESCRIPTIONS

The structural context to which the Rule applies consists of two fields. the Rule Kernel and the Rule Context.

<context field> ::=

<rule kernel> <rule context>

The Kernel describes the structural conditions of the string that is to be modified. while the Rule Context describes the conditions that must be met in the surroundings of the Kernel for the Rule to apply.

The unit used in the description of structural contexts is the *Context Sequence.*

<context sequence> ::=

{ <context unit> |
(<context sequence> <range expression>) }

<context unit> ::=

<phone unitname> |
<structure unitname> |
<context matrix>

The Context Sequence consists of a list of units which may be either Names of Phones or Structures or Context Matrices. Structures will be interpreted as a short hand form of a string of Phones with or without range indications as defined in the appropriate structure declaration statement.

<context matrix> ::=

[["<phone unitname>",] [<feature bundle>]
[(<expression>)]]

The Context Matrix is a matrix describing the conditions which must be met within a single Phone for the Rule to apply.

If the Matrix contains the name of a Phone it will be interpreted as equal to the complete feature specification of that Phone. Any specific Feature values after the Phone name will override the Feature values of the definition. Thus the matrix ["p". +voice] means: All Features defined for 'p' except that 'p' is voiced here. irrespective of its original definition.

The special Feature value *?* has the effect of removing a Feature definition temporarily from a Phone. Thus the matrix *["p", ?voice]* means: All the Features defined for *p* except that *p* is undefined for voice in this context.

The *expression* field of a context Matrix may be used to test for specific values of Scalars or Parameters within the Phone - or in the neighbouring Phones if relative addressing is employed - or it may be used to test for certain global conditions.

8.3.1. Rule Kernels

The Rule Kernel describes a sequence of Phones to which the changes described in the Change Field must be made. Basically, the Rule Kernel is just a Context Sequence:

<rule kernel> ::=

<context sequence>

This definition holds unconditionally for *changeall*, *replace*, and *delete* rules. Since the number of Phones affected by a *change* rule must be explicitly stated in the formulation of the rule, *Structures* and *range expressions* are illegal in the Kernel of a change rule, i.e. only *phone unitnames* and *context matrices* are allowed in the Kernel of a change rule.

Furthermore, the Kernel of an *insert* rule must logically be empty.

8.3.2. Rule Contexts

The Rule Context describes the conditions which must be met in the environment of the Kernel for the rule to apply.

<rule context> ::=

<empty> |
/ <context description> ____ <context description>

<context description> ::=

<empty> |
<context sequence>

The rule context is signalled by a slash. Then follows the description of the Phones that must precede a certain Kernel for the rule conditions to be met, the *Pre-Context*. The place of the Kernel itself within the context is signalled by one or more underline characters. Then follows the

description of the Phones that must follow the Kernel, the *Post-Context.*

If the whole context field is empty, the rule applies unconditionally, i.e. regardless of the context in which the Kernel appears.

Context descriptions are evaluated from the Kernel and out - i.e. the Pre-context is evaluated from right to left while the Post-context is evaluated from left to right.

It should be noted that only Features, Scalars, and Parameters actually mentioned in a rule are taken into consideration when it is determined whether a given input string matches a particular rule. In any Matrix Features which are not mentioned are marked as "undefined". An already defined Feature may be declared as "undefined" through the *?* operator.

The following rules describe the conditions under which a description matches the description of a segment in the buffer:

(i) If a Feature in the description is marked as plus *(+)* or minus *(-)* the corresponding Feature in the segment under observation must also have a defined value.

(ii) If a Feature in the description is undefined, either because it has not been mentioned at all or thrugh an explicit "undefinition" *(?)* the corresponding Feature in the segment under observation may have any value.

Examples of Context Descriptions:

```
["C", -voice] ([+cons]<0.2>) V
(C) ["V", +back,+round]
(C <ntcons = 0, 4>) ["V", (dur>10)]
["V", -stress]
(["C", +dent (dur<20 || dur>100)] <ntc=0.3>)
["C", -voice] ["C", (dur < dur(-1))]
```

8.4. RULE CHANGE DESCRIPTIONS

The Change Field is the last part of the Rule Statement. It describes the modifications to be carried out in the work buffer.

<change field> ::=

{ <absolute element> }

<absolute element> ::=

<phone unitname> |
<absolute matrix>

<absolute matrix> ::=

[["<phone unitname>" ,] [<feature bundle>]
[(<assignment field>)]]

<assignment field> ::=

<assignment statement> {, <assignment statement>}

The Change Field consists of a list of *Absolute Elements.* As were the case with the Kernel field there are certain semantic restrictions to the Change field.

Thus, in a *change* rule the number of absolute elements must agree with the number of Phones in the corresponding Kernel field so as to state explicitly how each Phone is to be modified.

In a *changeall* rule only *one* Absolute Element must appear, since the same modifications will be applied to all the Phones of the Kernel.

Finally, in a *delete* rule the whole change field must be empty.

An Absolute Element may be just the Name of a Phone, or it may be an expression involving explicit values of Distinctive Features with or without a Phone Name. If the special feature value *?* is used it will *un-define* the feature for that Phone. The *assignment field* is used when specific values are to be assigned to Scalars, Parameters, or to global variables (integers and reals). The syntax of the assignments is described in the next chapter.

If the absolute elements of a Change Description is the name of a Phone, the Feature Matrix of that Phone replaces the Matrix of the appropriate Phone in the Work Buffer. If the absolute element contains an absolute matrix the feature values of that matrix replace the corresponding feature values of the appropriate Phone in the work Buffer.

If any of the features concerned are currently undefined in the Work Phone these features become defined for that Work Phone.

Features having the value *?* in the matrix field of a Change Description should become undefined in the corresponding Work Phone if they are already defined.

If an absolute matrix contains an assignment field the receiving location(s) (or lvalue(s)) refer either to global variables or to Scalars or Parameters of the appropriate Work Phone.

Since Scalar and parameter lists are not copied into the Work Buffer until they are modified, assigning into a Scalar or Parameter which is currently not residing in the Work Buffer, will cause the appropriate Phone to be mapped back onto the definition table and the corresponding Scalar or Parameter list to be copied into the Work Buffer before the assignment actually takes place.

Relative addressing may be used to obtain values of Scalars or Parameters for comparison or copying between neighbouring Phones in the Work Buffer. The semantics for evaluating relative addresses in these (and all other) cases is governed by two general principles:

(i) Relative addresses are evaluated within the entire Context Field, i.e. Rule Kernel and Rule Context, before any modifications are applied to the Work Buffer.

(ii) Relative addresses in change Fields of Insertion Rules are evaluated when all new Phones have been entered into the Work Buffer.

Therefore in the following example of an Insertion Rule:

insert 1.1: / V ["C", (dur>dur(+1))] ___ C -> V

a vowel (V) will be inserted between two consonants (C) if the duration of the first consonant is greater than the last.

Also in the Deletion Rule:

delete 1.2: V / ["C", (dur<dur(+1))] ___ # ->

a word final vowel (V) will be deleted if its duration is greater than the duration of a preceding consonant (C), i.e. no attempt will be made to read the duration of the word boundary pseudo phone (#).

Finally consider the Insertion Rule:

insert 1.3: / V(C) ___ # -> V ["C", (dur=dur(-1))]

which will insert an extra VC-sequence word finally (before '#') after a vowel (V) with an optional consonant following. The duration of the inserted consonant will be set equal to the duration of the defined duration of 'V'.

Modifications are carried out from left to right as described in the Change Field.

Consequently the rule:

change 2.1: V C / -> [(dur += 10)] [(dur = dur(-1))]

will cause the duration of *C* to be equal to the duration of *V* - including the added 10 ms.

Whereas the rule:

change 2.2: V C / -> [(dur=dur(+1))][(dur+=10)]

will cause *C* to be 10 ms longer than *V*.

8.4.1. Assignment Statements

Assignment Statements move values to specified data locations.

<assignment statement> ::=

<lvalue> <assignment operator> <expression>

<lvalue> ::=

<integer name> |
<real name> |
<scalar name> |
<parameter specification>

<parameter specification> ::=

<parameter name> [.**<parsub field>**]

<parsub field> ::=

<empty> | **tg** | **ti** | **tx**

Lvalues are the receiving locations in assignment expressions. They may be global variables or segmental Scalars or Parameters. It should be noted that a Parameter consists of three fields. its Target and two transition times. If the parameter sub-field is left empty the expression is assumed to refer to the Target field of the Parameter. thus allowing expressions like *F1 = 250* or *F1. = 250* to mean what they appear to say.

Assignment statements containing non-active Parameters are currently ignored.

<assignment operator> ::=

= | += | -= | *= | /= | %=

The basic assignment operator is the equal sign, which simply causes the result of the expression following the operator to be left in the receiving location - the lvalue.

The other five assignment operators are *arithmetic* assignment operators. Thus, the operator += causes the result of the expression following the operator to be *added* to the current contents of the lvalue, while the result of the addition is left in the same location.

The operations performed are: Addition (+=), subtraction (-=), multiplication (*=), division (/=), and the modulus operation (%=).

Real and integer values may be mixed in assignments. An assignment always converts to the type of the receiving location. Reals are converted to integer type by truncation.

Examples of Assignment Statements:

```
dur = ......
F1.tg *= ......
F2 = ......
```

9. EXPRESSIONS

In SPL there are two types of expressions: The ordinary *Logical* or *Arithmetic Expressions* and the special class of *Range Expressions.* Range Expressions are used within Context Sequences to compute the number of elements that satisfy the conditions described in the Sequence.

When expressions are evaluated over- or underflow is reported (except division by zero).

9.1. ARITHMETIC AND LOGICAL EXPRESSIONS

Constants, Scalars, Parameters, and global variables may be combined with operators in expressions to obtain new arithmetic or logical values.

Arithmetic expressions are used for ordinary computational purposes. However, the result of any arithmetic expression may be used as a logical value. Thus, any non-zero value has the logical value *true,* while a zero value is equal to the logical value *false.*

Real and integer values may be mixed in expressions. In these cases the contents of integer variables is converted to real before the result is computed. The result of a computation is always converted to the type of the receiving location. Reals are converted to integer type by truncation.

The order of precedence for the different operators is defined by the syntax:

<expression> ::=

[<logical sign>]<logical term> |
<expression> <alternative operator> <expression>

<logical term> ::=

<arithmetic expression> |
<arithmetic expression> <relational operator>
<arithmetic expression>

<arithmetic expression> ::=

[<sign>] <term> |
<arithmetic expression> <additive operator>
<term>

<term> ::=

<primary expression> |
<term> <multiplicative operator>
<primary expression>

<primary expression> ::=

<constant name> |
<integer name> |
<real name> |
<scalar expression> |
<parameter expression> |
(<arithmetic expression>) |
<function call>

<function call> ::=

<function name> (<expression list>)

<expression list> ::=

<empty> |
<expression> { , <expression> }

<scalar expression> ::=

<scalar name> [<relative location>]

<parameter expression> ::=

<parameter name> [.<parsub
[<relative location>]

<relative location> ::=

(<integer expression>)

The Relative Location is used to specify Parameter or Scalar values from neighbouring Phones in the Work Buffer. Thus, *F1.tg(-1)* means 'the target value of the first formant in the Phone immediately preceding this one in the Buffer'.

It is an error to attempt to access Phones outside the Work Buffer.

9.2. OPERATORS

Operators are divided into four classes according to their order of precedence.

9.2.1. Multiplicative Operators

The three operators of the multiplicative class perform multiplication, division, and the modulus operation.

<multiplicative operator> ::=

*** | / | %**

9.2.2. Additive Operators

Operators of the additive class perform addition and subtraction. A term may be preceded by a plus or minus to indicate sign identity or sign inversion.

⟨sign⟩ ::=

+ | -

⟨additive operator⟩ ::=

+ | -

9.2.3. Relational Operators

Operators of the relational class compare two arithmetic expressions and return values of *true* or *false.*

⟨relational operator⟩ ::=

== | != | < | <= | > | >=

The comparisons performed are: Equal to, Not Equal to, Less than, Less than or Equal to, Greater than, and Greater than or Equal to.

9.2.4. Alternative Operators

The two operators of this class combine the truth values of two or more (logical) expressions to produce one logical result of the operations Logical And, and Logical Or.

⟨alternative operator⟩ ::=

&& | ||

A logical sign may be prepended a logical term to negate the truth value of the term:

⟨logical sign⟩ ::=

!

9.3. FUNCTIONS

A limited set of the more usual mathematical functions are defined in SPL. Suggested list of basic arithmetic functions is: *log(), logn(), exp(), sqrt().* This list may, however, be expanded as the need arises. Also, a special class of system functions is being considered. Thus, a function *lenght()* returning the number of Phones currently in the Work Buffer is needed.

Furthermore, a set of special input functions that will accept input from sense lines and knobs on a control panel

are being considered. These could be used to make experiments with interactive modifications to parameter values.

9.4. RANGE EXPRESSIONS

Range Expressions are used to evaluate repetitions of constructs within structural descriptions.

<range expression> ::=

<empty> |
< [<lvalue> =] <minmax expression> >

<minmax expression> ::=

<empty> |
<min expression>, <max expression>

<min expression> ::=

<empty> |
<integer expression>

<max expression> ::=

<empty> |
<integer expression>

A Range Expression consists of two major fields: An assignment and a minimum-maximum expression field. The latter field defines the minimum and maximum number of repetitions of the structural entity in question which will satisfy the structural condition. The minimum and maximum number of repetitions allowed may not evaluate to a negative value.

The assignment field assigns the number of repetitions actually found in a successful match to the location described in the assignment field. If no match is found the contents of the location is unmodified.

If the assignment field is empty it is an indication that the user program will not need the output of the count. Consequently it will not be made available.

If the whole min-max field is empty, it means that any number of repetitions will satisfy the conditions.

If the minimum sub-field is empty it means: Zero or more.

If the maximum sub-field is empty it means: Infinitely many.

If the entire Range Expression is empty the construct is interpreted by a special convention to mean: Zero or One repetition. Thus, allowing the use of parentheses in a structural description to simply signify an optional string as in: *V(C)#.*

When a Context Sequence involving Range Expressions is evaluated the program will find the longest match which satisfies the structural conditions.

Due to the way Context Sequences involving range expressions are expanded certain structural descriptions may be overlooked. Thus the structural descriptions:

/ X (Y<0.5>) ___ or / ___ (Y<>) X

will not be correctly matched if the structural descriptions of *Y* match the descriptions of *X*. The *X-Phone* will be used as part of the range of *Y* before any matching of *X* can be done.

This situation is considered a programming error and will not be caught by the SPL compiler or run time system.

Examples of Range Expressions:

```
<n = 1, 2>
<0,3>
<,>
< >
<i=>
```

10. AUXILIARY COMMANDS

The Auxiliary Commands perform various functions extraneous to the linguistic description which is contained in the Rule and Data Statements. The functions are mainly debugging and input/output control.

10.1. TRACE FUNCTION

The Trace Command controls the trace debugging function.

<trace command> ::=

trace <trace function>

<trace function> ::=

on | **off** | <empty>

When the Trace function is *on*, the produced program will print out the contents of the Work Buffer from the start of the Buffer up to (and including) the current position of the Rule Kernel every time a Rule has modified the contents of the Buffer. The Rule Label (if it is there) will be prepended to the produced output string.

An empty function field or the reserved word *on* causes the debugging function to be turned on. It will stay on until the next off-command - or until the end of the program.

10.2. OUTPUT CONTROL

Three commands are available for controlling the output from an SPL program. They are *File*, *Print*, and *Speak*. The File statement defines the destination of the output produced, and should properly be considered part of the data declaration. The Print command produces output in symbolic form, i.e. phonetic notation. And the Speak command produces output in a format capable of driving a parametric synthesizer.

10.2.1. File Statement

Since the File declaration interacts heavily with the operating system of the target computer of the SPL compiler the syntax described here may be considered a sort of guideline.

<file statement> ::=

file <output unit> , "<filename>"

<output unit> ::=

<name>

<filename> ::=

any legal filename

The File command causes the named output file to be associated with the output unit mentioned. If the file exists it will be opened for output. If it does not exist it will be created and prepared for output.

10.2.2. Print Command

The Print command produces output in symbolic form.

<print command> ::=

print <output unit>

The Output Unit must be a filename defined via a File statement.

The Print command causes the entire contents of the Work Buffer to be written to the appropriate output file in Symbolic Form.

10.2.3. Speak Command

The Speak command takes one optional argument.

<speak command> ::=

speak [<output unit>]

This command consists of the reserved word *speak* and an optional output destination. It causes the contents of the Work Buffer to be interpolated and the result to be output via the designated output unit. Using the Speak command without output designation causes output to be sent directly to the appropriate speech synthesizer. The format of the output is determined by the implementation.

The Speak command does not terminate the application of Rules, although the usual procedure will be to have a Speak command as the last Statement of the SPL program. If the program contains several Speak commands with one or more Rules between them, progressively more refined versions of the same input sequence will be produced.

In the UNIX implementation output from the Speak command is usually directed to the Standard Output Device from where it may easily be redirected to a file or piped to a driver program for a hardware synthesizer.

11. IMPLEMENTATION NOTES

11.1. Mapping the Contents of the Work Buffer

When a rule modifies the feature matrix of a Phone in the Work Buffer this modification will only affect the copy of the Phone in the Buffer - not the original definition of the Phone, which still remains intact in the definition table. At certain points during execution of the program the feature matrices will, however, need to be mapped back to the original definition table.

This happens first of all every time the buffer contents must be printed in symbolic form (through the trace or print command). In these cases the definition table is searched

for matches with the Phones of the Work Buffer. And for every match found the corresponding Symbolname is printed. When no match can be found a special default symbol must be output (or the complete list of feature combinations?).

A more problematic mapping occurs when the Phones of the Work Buffer are expanded from consisting of only feature matrices to their full segmental descriptions, i.e. containing Scalars and Parameters. This mapping should, of course, be delayed for as long as possible so that changing, for instance, a vowel from [+back] to [-back] will cause another table of Parameter values to be used.

The first point when the mapping becomes necessary is when a Scalar or Parameter value in the Work Buffer need to be modified. Ultimately, i.e. at interpolation time, all the Phones must, of course, be mapped.

At the time of mapping the definition table is searched for matches with the appropriate matrices in the Work Buffer. When a match is found the segmental description is copied into the Work Buffer and Parameter, or whatever it was, is modified. Thus, by the end of the program the Work Buffer will hold all the modified versions of the Phones while Phones which were not modified by any rule may be taken directly from the definition table.

If no match can be found at the final mapping it is an error condition of which the user must be duly notified.

It should be noted, however, that once the segmental part of a Phone has been mapped into the Work Buffer no further changes in the feature composition of the Phone can affect the segmental properties of that Phone.

REFERENCES

Basbøll, H. and Kristensen, K. 1974: "Preliminary work on computer testing of a generative phonology of Danish", *Ann. Rep. Inst. Phon. Univ. Cph.* 8, p. 216-226

Basbøll, H. and Kristensen, K. 1975: "Further work on computer testing of a generative phonology of Danish". *Ann. Rep. Inst. Phon. Univ. Cph.* 9, p. 265-292

Bobrow, D. G. and Fraser, J. B. 1968: "A phonological rule tester", *Communications of the ACM*, 11, 11, p. 766-772

Carlson, R. and Granstrom, B. 1974: "A phonetically oriented programming language for rule description of speech", *Preprints of the SCL-1974*, 2, p. 245-253

Chomsky. N. and Halle. M. 1968: *The sound pattern of English* (Harper and Row)

Hertz. S. R. 1982: "From text to speech with SRS". *JASA* 72. 4. p. 1155-1170

Holtse. P. 1974: "Preliminary experiments with synthesis by rule of standard Danish". *Ann. Rep. Inst. Phon. Univ. Cph.* 8. p. 239-251

Holtse. P. 1982: "Speech synthesis at the Institute of Phonetics". *Ann. Rep. Inst. Phon. Univ. Cph.* 16. p. 117-126

Kerkhoff. J.. Wester. J. and Boves. L. 1984: "A compiler for implementing the linguistic phase of a text-to-speech conversion system". *Proc. Inst. Phon.. Catholic University, Nijmegen.* 8. p. 60-69

Molbæk Hansen. P. 1982: "The construction of a grapheme-to-phone algorithm for Danish". *Ann. Rep. Inst. Phon. Univ. Cph.* 16. p. 127-136

Molbæk Hansen. P. 1983: "An orthography normalizing program for Danish". *Ann. Rep. Inst. Phon. Univ. Cph.* 17. p. 87-109