Automatic Speech Data Processing with Praat[1]
Lecture Notes

Ingmar Steiner
steiner@coli.uni-sb.de

Fall Semester 2006/2007

[1] www.praat.org

Contents

List of Figures

List of Tables

Listings

Chapter 0

A Short Preview

This chapter will showcase a short test run in Praat, which demonstrates a few of the things yet to come by explaining a simple script and what it does. It requires nothing from the reader except an open mind, and a willingness to postpone full comprehension until later chapters, where everything will be explained from the ground up.

0.1 Automating Praat

We start Praat by executing the `praat` binary (or `praat.exe` under Windows), which brings up the Praat *Object Window*, as well as the *Picture Window*. Both are, for now, empty, and since we don't need the Picture Window yet, we can simply close it (it will open again as required).

To load a `wav` file, we use the command `Read from file...` from the *Read* menu, which opens the usual file selection dialog window. Once we select a file, that file is loaded into the *object list* (unless of course the file is of a type that Praat can't recognize, in which case we get an error message instead). For now, let's assume I want to load the file `aufnahme_1.wav`.

0.2 The Script Editor

Let's do the same thing again, by running a script.

Select the command `New Praat script...` from the *Praat* menu to open a fresh *Script Editor* Window. The Script Editor is nothing but a simple text editor, which we will use to develop our scripts. There is a *history mechanism* in Praat that keeps track of all commands issued and objects selected, which is accessible via the `Paste history` command in the *Edit* menu of the Script Editor. Using this command, we see that the two lines correspond exactly to what we just did, i.e. open a `wav` file and open the Script Editor. In fact, the commands now in the script are *precisely* what the commands in the menu of the Object Window are called, all the way to the ... at the end of the file opening command! This means that the `Read from file...` command takes an *argument*, namely the absolute path of the file it opened.

We can run the script with the `Run` command from the *Run* menu, and *voilà!* it loads the Sound file again and opens another Script Editor (which we close

Figure 0.1: The Praat Object Window in Linux/KDE, with a Sound loaded

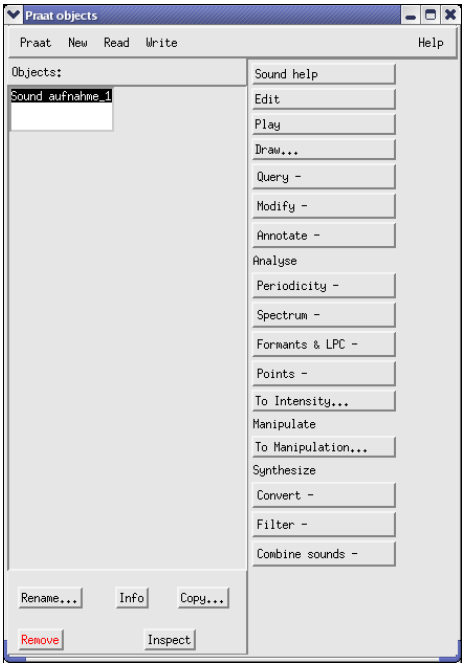
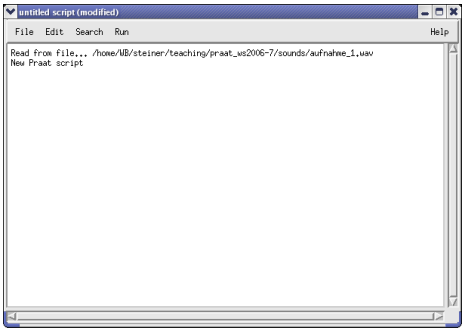


Figure 0.2: The Script Editor window



again, since we already have one).

Let's start a new script by using the `New` command from the *File* menu of the Script Editor (selecting "Discard & New" when prompted). For reasons that will be explained later, let's save this script as `testrun.praat` (using the `Save as...` command from the *File* menu), which will allow us to use relative paths.

0.3 Batch open script

Checking the `sounds` directory, we have four `wav` files that we can load in this fashion. So let's open them all at once (as one "batch"), because having to click on `Read from file...` and selected one file multiple times is just plain annoying.

0.3.1 Repeating commands

We could write a script like this:

Listing 0.1: `batchOpen1.praat`

```
Read from file... sounds/aufnahme_1.wav
Read from file... sounds/aufnahme_2.wav
Read from file... sounds/aufnahme_3.wav
Read from file... sounds/aufnahme_4.wav
```

0.3.2 `for` loop

But that's not really elegant, because we're doing things repeatedly that differ only in a single number. So instead, we could do this:

Listing 0.2: `batchOpen2.praat`

```
for number from 1 to 4
  Read from file... sounds/aufnahme_'number'.wav
endfor
```

This involves a `for` *loop*, which takes a counter variable called `number`, sets it to the value given after the `from` (here, `1`), and does everything until the `endfor` line, at which point it adds `1` to the value of `number` and checks whether `number` is less than or equal to the number we supplied after the `to` (here, `4`). If yes, then it repeats everything between the `for` and `endfor` lines (increasing the value of `number` again), if not, the loop is finished (and the rest of the script is processed).

This means that the `Read from file...` command is actually run four times. As the argument to the command, we've used the variable `number` again, and by enclosing it in 'single quotes', we ensured that its value (first `1`, then `2`, and so on), rather than its name (`"number"`) is used, so that the argument to the `Read from file...` command (the filename) actually *changes* every time we go through the loop.

0.3.3 `Strings` file list

What if we have different names for the files? What if we want to open all `wav` files in a directory, regardless of their names?

Praat has a type of object called `Strings`, which is essentially a list of *strings*, each string being a list of characters (letters, numbers, etc.). There is a command called `Create Strings from file list...`, which looks at the contents of a directory and returns all files matching a given pattern as the strings of a `Strings` object. Once we have a `Strings` object, we can use commands like `Get number of strings` and `Get string...` to print information about the object (and its contents) to the *Info Window*.

Let's write a script like this:

Listing 0.3: `batchOpen3.praat`

```
Create Strings as file list... wavList sounds/*.wav
numberOfStrings = Get number of strings
for stringCounter from 1 to numberOfStrings
  select Strings wavList
  filename$ = Get string... 'stringCounter'
  Read from file... sounds/'filename$'
endfor
```

First, we create a `Strings` object called *wavList* (we could just as well call it something else, though), which contains all filenames in the `sounds` directory ending in `.wav`. Since we can't be sure how many there will be and have to tell the `for` loop how many times we want it to go around, we use the `Get number of strings` command from the *Query* menu of the `Strings` object's *dynamic menu* (to the right of the object list). The output of this query command is redirected into another variable, which we call `numerOfStrings` (again, this could be anything, but we want to use names that make sense).

Then comes the loop. Inside the loop, we'll skip over the first line for now, and look at the `Get string...` command (again, from the *Query* menu). This one takes an argument (remember? that's what the ... means), namely the index of the string we want to know. An argument of `1` returns the first string, `2`, the second, and, `'numberOfStrings'`, the last one (in this script, anyway). Since we want a different string each time the loop goes through, we use `'stringCounter'` as the argument (because `stringCounter` is our loop's counter variable). But again, we redirect this query command's output (the `stringCounter`th string, i.e. filename) into a variable, which we call, for the sake of transparency, `filename$`. The reason there is a `$` at the end of this variable's name is that it is a *string variable*, not a *numeric variable*, which is the type of output of the `Get string...` command. And finally, we use that string variable in the `Read from file...` command as before.

One pitfall we've avoided is that once the first `wav` file is loaded, the selection in the object list changes so that only that `Sound` object is selected. However, the next time the script goes through the loop, the `Get string...` command will cause an error, because that command only works when a `Strings` object is selected. This error can be avoided if we explicitly select the `Strings` object containing our file list in the loop, before we use the `Get string...` command. This is done with the `select` command, which takes either an object's numeric *ID* or, as here, its *class* and *name*. In this case, we know the class (`Strings`), and the name as well (`wavList`), because we just assigned it. Usually however, using the ID number is preferable.

0.3.4 Simple dialog windows

What if we want to use this script for a different directory and open other files there? Wouldn't it be nice to have a way for Praat to ask which directory it should look inside and open all files of a specific type out of?

Let's go through the following script:

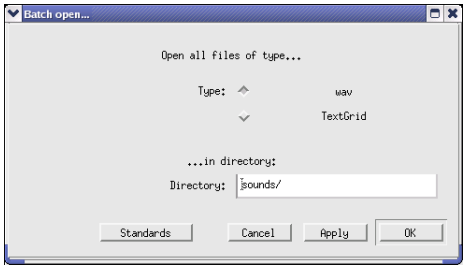
Listing 0.4: `batchOpen4.praat`

```
form Batch open...
  comment Open all files of type...
  choice Type: 1
    button wav
    button TextGrid
  comment ...in directory:
  sentence Directory sounds/
endform

Create Strings as file list... 'type$'List 'directory$'*.'type$'
numberOfStrings = Get number of strings
for stringCounter from 1 to numberOfStrings
  select Strings 'type$'List
  filename$ = Get string... 'stringCounter'
  Read from file... 'directory$''filename$'
endfor
```

The first part of the script consists of something that looks like a `form` loop, but it actually defines a dialog window that Praat will display when the script is run, which prompts the user for certain arguments which will be used during the second part of the script.

See if you can figure out what the lines between the `form` and `endform` do:

Figure 0.3: Dialog window of `batchOpen4.praat`



The `choice` and `sentence` lines are the actual point of this `form`, since they provide variables whose values are filled in by the user. So once the user clicks the "OK" button in the dialog window, the script continues with two new variables, `type$` and `directory$`[1], which contain either `"wav"` or `"TextGrid"`, and `"sounds/"` (or whatever the user entered into the text field), respectively. The details of the `form` loop will be explained later.

The second part of the script is basically the same as the script in the previous section, except that references to a "hard-coded" directory `sounds/` have been replaced with `'directory$'`, which is whatever the user entered in the dialog window, and similarly for references to `wav` as the file type.

0.3.5 Good scripting practices

It's generally advisable to make a script as robust as possible, with portability and scalability in mind. This means that we should add a few things to that last script.

For instance, it is quite possible that the user will accidentally input a directory in the dialog window that does not exist or is not readable. In this case, the script will simply terminate with an error generated by Praat directly, which we couldn't do any better.

On the other hand, if the directory exists (and a `Strings` object is successfully created), but contains no files of the selected type, the `Strings` will be empty, and no files will be loaded. It would be nice for the user to receive some information about this, so we'll add a *condition* with `if...endif` and cause an error window of our own to pop up, using `exit`.

Another potential problem is that the user might not put a trailing slash at the end of the directory name, which would cause the script to try and create a `Strings` from files matching the pattern `someOtherDirectory*.wav`, which would not look into the directory at all and, again, create an empty `Strings` object.[2] So, to make sure there is exactly one slash where we need it, we add another condition involving the *string function* `right$()`, which returns a substring of a

[1] Actually, *three* new variables: the selection of `Type` is additionally stored in the numeric variable `type`, which contains the *number* of the selected `button`, in this case, `1` or `2`.

[2] We could of course have added a slash ourselves, as in `Create Strings from file list... 'type$'List 'directory$'/*.'type$'`, but that would cause the reverse problem if the user were to enter `"someOtherDirectory/"` in the dialog window.

given length from a string. If the last character of `directory$` is *not* a slash, we simply add one through *string concatenation*.

And finally, after the script is finished, we no longer need the `Strings` object, so we simply remove it. However, to be really sure we get the right object (in case there happens to be another object of the same class with the same name in the object list), we'll use the `Strings` object's numeric ID, which we get with the `selected()` function, select it (with `select` or `plus`), and use the `Remove` command, which is actually just a button in the Praat Object Window, below the object list.

Just to be explicit, we'll also finish the script by selecting all of the objects it loaded, so that the user knows immediately what happened (after all, the script will tend to run supraliminally fast!). For this, we'll store all of the objects' IDs in an *array* as they are loaded. This is a tricky, but important part of Praat scripting, but it won't be explained in detail until later.

This is our new script (several *comments* have been inserted to explain the new parts, these are lines starting with a `#`):

Listing 0.5: `batchOpen5.praat`

```
form Batch open...
  comment Open all files of type...
  choice Type: 1
    button wav
    button TextGrid
  comment ...in directory:
  sentence Directory sounds/
endform

# add trailing slash to directory$, if there isn't one already
if right$(directory$, 1) <> "/"
  directory$ = directory$ + "/"
endif

Create Strings as file list... 'type$'List 'directory$'*.'type$'
stringsID = selected("Strings")
numberOfStrings = Get number of strings
for stringCounter from 1 to numberOfStrings
  select Strings 'type$'List
  filename$ = Get string... 'stringCounter'
  Read from file... 'directory$''filename$'
  # populate array with object IDs
  file_'stringCounter'_ID = selected()
endfor

# cleanup Strings object
select stringsID
Remove

# check if Strings is empty
if numberOfStrings = 0
  exit No 'type$' files were found in directory 'directory$'!
endif

# select all files loaded by this script
select file_1_ID
for fileNumber from 2 to numberOfStrings
  plus file_'fileNumber'_ID
endfor
```

Chapter 1

Scripting Fundamentals

Before we begin, a note concerning reference: This introduction assumes no familiarity with programming in general or Praat scripting in particular. However, the reader is strongly encouraged to consult the Praat Manual for reference, which is available via the "Help" function within Praat, or online at http://www.fon.hum.uva.nl/praat/manual/Intro.html.

1.1 My first program

Traditionally, the first step in learning any programming language is to cause the words "Hello World!" to appear on the screen. We'll do this using Praat, because that's what this course is about. Since Praat can be considered a scripting language, we need two things for this example to work: the main Praat program (called `praat` under Linux or `praatcon` under Windows) and a text file containing our instructions in a format that Praat can understand.

The text file is what we will refer to as our script, and can be created with any text editor. Using our favorite editor, let's create a script file called `helloWorld.praat`. (The `.praat` part at the end, sometimes referred to as the *file extension*, is not necessary and could just as well be something else, such as `.script`, `.psc`, `.txt`, or whatever. It's not important because the file is just a text file, and Praat will check its contents for well-formedness when we tell it to run the script.)

This script file should contain only the following line:

Listing 1.1: "Hello World!" in Praat

```
echo Hello World!
```

That's it!

Before we get into explanations, let's run the script (from the command line) and make sure it works:

```
$ praat helloWorld.praat
Hello World!
```

Great! So what just happened? Well, we invoked the `praat` program and gave it the script as an *argument* by typing a space followed by the script filename.

This caused Praat to open the script file, and starting from the top, carry out the instructions, line by line.

Our script consists of only a single instruction, which works much in the same way as what we did to run the script. There is one command, `echo`, followed by an argument. The `echo` command takes exactly one argument, so everything after the first space is treated as that argument (even if there is another space before the end of the line), and prints that argument to the output, which is just what we wanted.

To put things into perspective, other programming and scripting languages (the distinction is irrelevant here) can be much more complicated, as the following examples illustrate:

Listing 1.2: "Hello World!" in C++

```
#include <iostream.h>
main()
{
  cout << "Hello World!" << endl;
  return 0;
}
```

Listing 1.3: "Hello World!" in Java

```
import java.io.*;
class HelloWorld{
  static public void main(String args[]){
    System.out.println("Hello World!");
  }
}
```

Listing 1.4: "Hello World!" in Scheme

```
(define helloworld
  (lambda ()
    (display "Hello World!")
    (newline)))
(helloworld)
```

Of course, none of this is relevant here, except to illustrate how simple by comparison the Praat scripting language is!

1.2 Scripting elements

Apart from the `echo` command, there are of course many other commands that we could write into a script file as instructions. However, each instruction must reside on its own line, since Praat will assume everything to the end of the line to belong to one instruction. We can, however, have spaces and/or tabs ("whitespace") at the beginning of the line, before the instruction. This means we can make our script code more readable by indenting lines that belong together.

If a line becomes too long, we can break it into more than one line, if we begin each continuation line with a ..., and Praat will treat them as a single instruction.

The following three (!) instructions are all well-formed:

```
echo Hello World!
              echo Hello World!
echo This is output generated by a line so long that it was
... continued on a second line.
```

1.2.1 Comments

It is not only possible, but also considered good form to explain what we are doing in a script by providing *comments*. This not only helps others who might want to understand our code, but also ourself, once we go back to a script we wrote a few weeks ago. Trust me on this...

Comments should be on their own line, and that line should start with a `#`, `;`, or `!`. Some commands will also allow us to place a comment after the instruction on the same line, but others will cause problems when we try this, so it's safest to place comments on their own lines. Essentially, everything after this comment symbol is ignored by Praat. This also allows us to quickly disable certain lines when we're developing a script, in case we don't need them at the moment, or we're trying to find the source of an error ("debugging").

```
# This line is a comment.
! So is this one.; And this one as well.

# The last line was empty, and therefore ignored.
a = 1 + 2 ; we just did math, and this is another comment.

# The following does not work:
echo Hello World! ; this comment should not be printed, but will be!
```

1.3 Variables

Without variables, there could be no scripting.

A variable is a name by which Praat remembers the output of an instruction, with the purpose of reusing that output at a later time. Let's take a real-world example:

Let's assume that we want to run a pitch analysis, consisting of several steps, on some male voice data, and each of these steps depends on a certain predetermined value for pitch floor and ceiling. We could enter those floor and ceiling values by hand in each step, taking care to use the same values each time. While this would of course work perfectly well, let's imagine we want to run the same analysis on female voice data, where pitch floor and ceiling will be different. We would have to adjust those values in every single analysis step by hand, taking care not to forget to change any "male" values, or else our analysis would become invalid.

It would be far easier to define the floor and ceiling values once, and then use those values throughout the various analysis steps. This is exactly what variables are for.

So instead of using the following pseudo-script:

```
# male voice data

# pitch floor is 75 Hz
# pitch ceiling is 300 Hz

# analysis step 1, which involves the values 75 and 600
# analysis step 2, which involves the values 75 and 600
# analysis step 3, which involves the values 75 and 600
# analysis step 4, which involves the values 75 and 600

# female voice data
```

```
# pitch floor is 100 Hz
# pitch ceiling is 500 Hz

# analysis step 1, which involves the values 100 and 500
# analysis step 2, which involves the values 100 and 500
# analysis step 3, which involves the values 100 and 500
# analysis step 4, which involves the values 100 and 500
```

We could use the following, subtly different one:

```
# male voice data

pitch_floor = 75
pitch_ceiling = 300

# analysis step 1, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 2, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 3, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 4, involving 'pitch_floor' and 'pitch_ceiling'

# female voice data

pitch_floor = 100
pitch_ceiling = 500

# analysis step 1, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 2, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 3, involving 'pitch_floor' and 'pitch_ceiling'
# analysis step 4, involving 'pitch_floor' and 'pitch_ceiling'
```

Note how the lines with the actual analysis instructions (which would of course be more complex in a real script) are *exactly the same* for both two speaker analyses. This may seem trivial at first, but implies all the power of scripting with variables.

Now, let's look more closely at what the lines that are not comments do. The instruction `pitch_floor = 75` tells Praat to create a variable with the name `pitch_floor` and assign to it a value that is equal to whatever is on the right side of the `=`, in this case, the number `75`. After this instruction has been carried out, we can at any time refer to this number remembered as `pitch_floor` by using the variable name `pitch_floor`. In fact, this is exactly what is done in the analysis steps (except that, being comments, they don't do anything at all).

Once we get to the part where we look at the female voice data, we no longer need the pitch parameters of the male voice, so we *change* the values of the `pitch_floor` and `pitch_ceiling` variables. This is done simply by redefining them, which causes Praat to forget what their previous values (if any) were.

Before you wonder, once they have been created ("declared" or "instantiated"), variables remain available until the end of the script, even if their values change. There is no way to delete a variable or otherwise remove it from memory, but there should not be a need to, either.

Now, let's write a short script that instead of chewing through pitch analyses, simply outputs the pitch parameters for the male and female voice data:

Listing 1.5: `outputPitchParameters.praat`

```
#male voice data

pitch_floor = 75
pitch_ceiling = 300
```

```
echo Male voice:
echo Pitch floor is 'pitch_floor' Hz
echo Pitch ceiling is 'pitch_ceiling' Hz

# female voice data

pitch_floor = 100
pitch_ceiling = 500

echo Female voice:
echo Pitch floor is 'pitch_floor' Hz
echo Pitch ceiling is 'pitch_ceiling' Hz
```

This script actually *does* something when run:

```
$ praat outputPitchParameters.praat
Male voice:
Pitch floor is 75 Hz
Pitch ceiling is 300 Hz
Female voice:
Pitch floor is 100 Hz
Pitch ceiling is 500 Hz
```

1.3.1 Variable names

There are simple but important rules to follow when choosing names for our variables, namely they must

- start with a lower-case letter
- contain only letters (upper or lower-case), digits, and underscores
- *not* contain spaces, dashes, punctuation marks, umlauts, or anything not in the previous point

So `a`, `fooBar`, `number_1`, and `aEfStSgs3sWLKJW234` are all valid, legal variable names, while `Pitch`, `my-number`, `column[3]`, and `lösung` are not.

Furthermore, it is not entirely impossible to inadvertently choose a variable name that is the same as a function name or a predefined variable. If this happens, Praat will give us an error, at which point we may want to consider the possibility that a variable name may have caused this. Don't worry too much about this for now, though; we will soon learn more about function names and predefined variables, so that we can avoid the few that there are.

Finally, a word of advice on naming variables: choose names that are semantically transparent and that we will not confuse with others in our scripts. While we may have to press a few more keys to type `numberOfSelectedSounds` than `ns`, we will certainly know what the variable stands for. Remember, *cryptic code is not prettier!*

1.3.2 Variable types

There are actually two different types of variables in Praat scripts: *numeric variables* and *string variables*. The first type is what we've seen already, but

has an important restriction: numeric variables can only contain numbers. So, `4`, `-823764`, `0.03253`, and `6.0225e23` (6×10^{23}; Avogadro's number) are all possible values for a numeric variable, while `abc`, `All this belongs together`, `€ 78.56`,

```
Amplitude:
Minimum: -0.87652892 Pascal
Maximum: 0.83545512 Pascal
Mean: -8.5033717e-07 Pascal
Root-mean-square: 0.36832867 Pascal
```

, and everything else are not. They are *strings*.

Strings can be assigned to string variables. These work exactly like numeric variables, but their names have a `$` at the end. This means that the numeric variable `foo` is not the same as the string variable `foo$`, and both may occur in the same script.

Whenever a string is to be used in a place where an (unevaluated) string variable is expected, the string must be enclosed in double quotes `""`, for example when declaring a string variable:

```
stringVariable$ = "the string contents"
```

One reason for the distinction between numeric and string variables will become apparent later, when we learn about operators. For now, let's leave it at this simple explanation: *numeric variables are variables we can do math with, and string variables aren't.*

Predefined variables

Incidentally, Praat provides a number of predefined variables, which will come in handy later on. For now, we should just have a quick look.

Table 1.1: Predefined variables

Name	*Value*
`pi`	`3.141592653589793`
`e`	`2.718281828459045`
`newline$`	"line break" character
`tab$`	"tab" character
`shellDirectory$`	the current working directory
`date$()`	current time and date (format example: `Mon Jun 24 17:11:21 2002`)
`environment$(`*key*`)`	*value* of environment variable *key*[a]

[a] This is specific to the operating system. In Linux, environment variables can be listed with the `env` command; in Windows, the corresponding button is found in the "System Properties".

Note: `date$()` and `environment$()` are actually *functions*, cf. Section 1.4.

Special characters in strings

To create a string containing special characters, such as tabs and line breaks, the apropriate predefined variables should be used. A double quote *within* a string must be doubled:

Listing 1.6: Double quotes in strings

```
quotedString$ = """string"""
echo quotedString$ = 'quotedString$'
```

```
$ praat doubleQuote.praat
quotedString$ = "string"
```

1.4 Operators and functions

We've already seen one operator, the *assignment operator* `=` that takes whatever is to its right side and assigns it to the variable to its left. There are of course others, but they share the syntax to use them, which is,

```
OPERAND1 operator OPERAND2
```

On the other hand, there are also *functions*, which for scripting purposes do similar things as operators, but tend to involve parentheses. Functions use the following syntax (brackets denoting optionality),

```
function ( ARGUMENT1 [, ARGUMENT2 [, ARGUMENT3 [,...]]] )
```

As we can see, the function takes a number of arguments (the number and individual type of the arguments is specific to the function), separated by commas and enclosed in parentheses.

Spaces around operators, parentheses, and commas are almost always optional, but increase the legibility of script code.

There are quite a number of operators and functions available in Praat, but they are divided into those that work on numbers and numeric variables, and those that work on strings and string variables. The former are commonly used for mathematical operations while the latter are sometimes collectively referred to as "string handling".

1.4.1 Mathematics

A short selection of commonly used mathematical operators and functions, along with some examples, follows:

Table 1.2: Mathematical operators and functions (selection)

		Example	*Outcome*
`+`	addition	`1 + 2`	`3`
`-`	subtraction	`3 - 2`	`1`
`*`	multiplication	`2 * 3`	`6`
`/`	division	`6 / 3`	`2`
`^`	exponentiation	`2 ^ 3`	`8`
`div`	division, rounded down	`10 div 3`	`3`
`mod`	modulo (remainder of `div`)	`10 mod 3`	`1`
`abs()`	absolute value	`abs(-1)`	`1`
`sqrt()`	square root	`sqrt(9)`	`3`
`round()`	nearest integer	`round(0.5)`	`1`
`floor()`	next-lowest integer	`floor(1.9)`	`1`
`ceiling()`	next-highest integer	`ceiling(0.1)`	`1`
`sin()`	sine	`sin(pi)`	`0`
`cos()`	cosine	`cos(pi)`	`-1`

The full selection of mathematics operators and functions can be found in the Praat Manual, under "Formulas 2. Operators" and "Formulas 4. Mathematical functions", respectively.

Of course, all operators and functions can be nested, i.e. used as *arguments* of others. Parentheses can and should be used to modify the priority as intended. An example:

```
abs(5 - (1 / (cos(2 * pi) + sqrt(4))) ^ -2) ; outcome: 4
```

Just for fun, the above instruction is the same as $\left|5 - \left(\frac{1}{\cos 2\pi + \sqrt{4}}\right)^{-2}\right|$.

In some situations (such as when working with `while` loops, cf. Section 1.5.2) we will find it convenient to know that there is a shorthand to writing `a = a + n` (where n is a number), namely the *increment* operator, which does exactly the same thing, but is written as `a += n`.

Note that there is also a *decrement* operator, `-=`, as well as `*=` and `/=`, which work analogously.

1.4.2 String handling

A string is, in effect, a list of characters, and this sequence can be queried and modified. An important concept is that of a *substring*, which is essentially a *part* of a string, or more formally, a contiguous sublist of the list of characters in a string. It sounds more complicated than it really is, as illustrated by these examples:

```
hello$ = "Hello World!"

# substring of hello$ containing the first 5 characters:
# "Hello"

# substring of hello$ containing the last 6 characters:
# "World!"

# substring of hello$ containing characters 3 through 7:
# "llo W"
```

There are a number of handy functions in Praat for doing things with strings, the first three of which do just what the last example implied. Functions with a `$` at the end of their name return a string, the others return a number. Note that the number of arguments, as well as their sequence and type (string or numeric), is important!

Listing 1.7: String function examples

```
helloWorld$ = "Hello World!"

# first 5 characters
hello$ = left$(helloWorld$, 5)
echo 'hello$'

# last 6 characters
world$ = right$(helloWorld$, 6)
echo 'world$'

# characters 3 through 7, i.e.
llo_W$ = mid$(helloWorld$, 3, 5)
echo 'llo_W$'
```

Table 1.3: String functions (selection)

	Returns
`left$(string$, length)`	first `length` characters of `string$`
`right$(string$, length)`	last `length` characters of `string$`
`mid$(string$, start, length)`	substring of `length` characters from `string$`, starting with the `start`th character
`index(string$, substring$)`	starting position ("index") of first occurrence of `substring$` in `string$` (`0` if not found)
`rindex(string$, substring$)`	starting position ("index") of last occurrence of `substring$` in `string$` (`0` if not found)
`startsWith(string$, substring$)`	`1` if `string$` starts with `substring$`, `0` otherwise
`endsWith(string$, substring$)`	`1` if `string$` ends with `substring$`, `0` otherwise
`replace$(string$, target$, replacement$, howOften)`	`string$` with the first `howOften` instances of `target$` replaced by `replacement$` (for unlimited replacement, set `howOften` to `0`)
`length(string$)`	number of characters in `string$`
`extractWord$(string$, pattern$)`	substring of `string$` starting *after* the first occurrence of `pattern$` and ending before the next space or `newline$` or at `string$`'s end (returns empty string if `pattern$` is not found in `string$`; empty string as `pattern$` returns the first word)
`extractLine$(string$, pattern$)`	as `extractWord$()`, but returns substring from `pattern$` to end of line or `string$`
`extractNumber(string$, pattern$)`	as `extractWord$()`, but returns number immediately following `pattern$` (returns `--undefined--` if no number after `pattern$` or if `pattern$` not found)

```
# starting position of first "l"
firstL = index(helloWorld$, "l")
echo 'firstL'

# starting position of last "l"
lastL = rindex(helloWorld$, "l")
echo 'lastL'

# does helloWorld$ start with "H"?
firstCharIsH = startsWith(helloWorld$, "H")
echo 'firstCharIsH'

# does helloWorld$ end with "d"?
lastCharIsD = endsWith(helloWorld$, "d")
echo 'lastCharIsD'

# replace first "Hello" with "Goodbye"
goodbyeWorld$ = replace$(helloWorld$, "Hello", "Goodbye", 1)
echo 'goodbyeWorld$'

# replace all "l"s with "w"s
hewwoWorwd$ = replace$(helloWorld$, "l", "w", 0)
```

```
echo 'hewwoWorwd$'

# length of helloWorld$
helloLength = length(helloWorld$)
echo 'helloLength'
```

```
$ praat simpleStringFunctions.praat
Hello
World!
llo W
3
10
1
0
Goodbye World!
Hewwo Worwd!
12
```

It is also quite simple to *concatenate* strings. This is accomplished using the + operator, which works differently with strings than numbers. Observe:

```
helloWorld$ = "Hello" + " " + "World!"

# outcome: "Hello World!"
```

Similarly, the - operator also works on strings, removing a substring from the end of a string, but *only* if the string indeed ends with the substring in question:

```
helloWorld$ = "Hello World!"

hello$ = helloWorld$ - "World"

# outcome: "Hello World!"
# why? because helloWorld$ doesn't end in "World", but in "World!"

hello$ = helloWorld$ - "World!"

# outcome: "Hello "
```

As with mathematical functions and operators, string functions can be nested. For instance, to get everything *except* the first 3 characters from a string, we could do this:

```
helloWorld$ = "Hello World!"

from3$ = right$(helloWorld$, length(helloWorld$) - 3)

# outcome: "lo World!"

# which is the same as

from3$ = mid$(helloWorld$, 4, length(helloWorld$) - 3)
```

1.4.3 Variable evaluation

The crucial part of working with variables is the ability to use either their names or their values. This means that in some situations, we will type the variable's

name, but we want Praat to interpret it as if we had typed the variable's current *value*. This is called *evaluating* (or "substituting" or "expanding") the variable. In Praat, this is done by enclosing the variable's name in single quotes (as in `'myVariable'`). Figuring out when to evaluate a variable, and when to just use its name is one of the tricky parts of writing Praat scripts.

However, a few examples should shed light on this mystery. We've already used evaluation several times, in combination with the `echo` command. However, as we saw in our very first script, the `echo` command simply outputs whatever follows it on the same line.

```
echo This is a sentence.

# output: This is a sentence.
```

If we have a variable called `numberOfFiles` and assign it the number `4`, then output this variable using `echo`, we have to use variable evaluation. Observe:

```
numberOfFiles = 4
echo numberOfFiles

# output: numberOfFiles
# however:

echo 'numberOfFiles'

# output: 4

# or, more verbosely:

echo number of files: 'numberOfFiles'

# output: number of files: 4
```

As we've also seen, we can freely mix normal output text and evaluated variables, all as the argument to the `echo` command.

So what happens when a variable is evaluated that has not been instantiated yet? Observe:

```
echo 'noSuchVariable'

# output: 'noSuchVariable'
```

(This may happen to you fairly often as you learn how to write Praat scripts, and is usually caused by mis-typing variable names.)

As a rule of thumb, every variable in single quotes is evaluated before the line itself is interpreted by Praat.[1]

Evaluating string variables works the same way, except that we use the string variable's name (i.e. `echo 'myString$'`).

This raises an intriguing possibility.

Evaluating variables *within* strings

Since variables can be evaluated *anywhere* in a Praat script, we can use this to evaluate a variable *within a string*! This means that the following is possible:

```
a$ = "is"
b$ = "sentence"
```

[1] Cf. Paul Boersma's explanation in the Praat User List.

```
c$ = "This 'a$' a 'b$'."

# outcome: "This is a sentence."

# by the way, this is the same as...

c$ = "This " + a$ + " a " + b$ + "."

# ...but slightly more intuitive!
```

In fact, this feature is the basis of Praat's mechanism for arrays (cf. Section 1.6).

Additionally, this is also how we can "convert" a numeric variable into a string, and vice versa:

```
a = 1
a$ = "'a'"

# outcome: "1"

a = 'a$'

# outcome: 1
```

Note that the conversion from string variable to number only works if the contents of `a$` can be interpreted as a number.

1.4.4 Comparison operators

Finally, there are a few comparison operators, which are used almost exclusively in condition statements (cf. Section 1.5.1), which return either "true" or "false". This is called a *truth value* (also referred to as a *Boolean* value). Praat has a healthy, inherently binary, notion of truth values in that "false" is always `0` and "true" is `1` (usually), or more generally, `not 0`.

Table 1.4: Comparison operators

Returns `1` *iff*	
`x`	`x` is not `0`
`not x`	`x` is `0`
`x and y`	`x` and `y` are both not `0`
`x or y`	either `x` or `y` is not `0`
`x = y` (*or* `x == y`)	`x` and `y` are the same
`x <> y` (*or* `x != y`)	`x` and `y` are different (same as `not x = y`)
`x < y`	`x` is smaller than `y`
`x <= y`	`x` is smaller than or equal to `y`
`x > y`	`x` is greater than `y`
`x >= y`	`x` is greater than or equal to `y`
... and `0` *otherwise*	

} works for strings, too!

As usual, these operators can be combined to allow complex conditions such as `(a = 2 and not b <= 10) or c$ <> "foo"`. You are strongly encouraged to use parentheses to ensure proper grouping of multiple subconditions.

Note that the concepts "smaller" and "greater" are in fact applicable to strings as well as numbers, but refer to alphabetical ordering, i.e. `"a" < "b"` is true. In the same sense, upper-case letters are "smaller" than lower-case letters.[2]

1.5 Flow control

1.5.1 Conditions

Rather often in a script, there are instructions that should only be carried out if certain circumstances are met, and not if they aren't. This is what *conditions* (also referred to as "jumps") are for. Let's look at an example:

Listing 1.8: `if...endif`

```
condition = 0
echo 'condition'

if condition
  echo Condition has been met!
else
  echo Condition has not been met!
endif

condition = 1
echo 'condition'

if condition
  echo Condition has been met!
else
  echo Condition has not been met!
endif
```

```
$ praat ifThenElse.praat
0
Condition has not been met!
1
Condition has been met!
```

Notice how in the first `if...endif` block, only the first instruction was carried out, and in the second, only the second instruction. While the blocks themselves are identical, the value of `condition` changed, which caused the *condition* given after the `if` to evaluate to `0` in the first case, and `1` in the second.

In case we only want to do something if a certain condition is met, but nothing if it isn't, we can omit the `else` part.

On the other hand, if we want to differentiate between several cases if the first condition is not met, we can use the `elsif`[3] command, as in:

```
if not value
  echo Value is 0
elsif value < 0
  echo Value is negative
elsif value <= 10
  echo Value is greater than 0 but no greater than 10
```

[2] This is because the values that are actually compared are the values of the *ASCII* codes of the letters. Look it up!

[3] Instead of `elsif`, we can also write `elif`.

```
else
  echo value must be greater than 10
endif
```

Only one of the echo commands will be carried out, depending on the value of `value`. Note that if more than one condition evaluates to true, only the first one will be applied.

1.5.2 Loops

The magic key to automating repetitive tasks are *loops*. Loops keep performing instructions until a *break condition* (also referred to as an "exit condition" or "terminating condition") is met. There are three different flavors of loops in Praat, `repeat...until`, `while...endwhile` and `for...endfor` loops. They all share a dangerous pitfall: if the break condition is never, ever met, the script will continue to run until the Praat *task* is ungracefully terminated by hand.[4] This is called an *infinite loop*, and Praat cannot help us detect one in advance. It's our responsibility to avoid these when using loops.

`repeat` loops

In a `repeat...until` loop (which we'll call a `repeat` loop for brevity's sake), all instructions between the `repeat` and `until` lines are carried out *repeatedly until* the break condition, supplied *after* the `until`, evaluates as true. This usually means that we need some sort of conditional variable, whose value is checked by the break condition.

Listing 1.9: `repeat` loop

```
counter = 10

echo Countdown:

repeat
  echo 'counter'...
  counter = counter - 1
until counter = 0

echo Blastoff!
```

```
$ praat repeatUntil.praat
Countdown:
10...
9...
8...
7...
6...
5...
4...
3...
2...
1...
Blastoff!
```

[4] In Windows, this is done with the *Task Manager*; in Linux, using the `kill` command.

Note that even if the break condition is true from the start, the `repeat` loop is still performed at least once.

`while` loops

The `while` loop works just like the `repeat` loop, except that the break condition is defined at the beginning of the loop, right after the `while`. This means that if the break condition is true from the start, the `while` loop is not performed at all.

Listing 1.10: `while` loop

```
sentence$ = "This is a boring example sentence."
searchChar$ = "e"

echo The sentence...
echo "'sentence$'"

numberFound = 0

while index(sentence$, searchChar$)
  firstPosition = index(sentence$, searchChar$)
  numberFound = numberFound + 1
  sentence$ = right$(sentence$, length(sentence$) - firstPosition)
endwhile

echo ...contains 'numberFound' "'searchChar$'"s.
```

```
$ praat whileEndwhile.praat
The sentence...
"This is a boring example sentence."
...contains 5 "e"s.
```

If `searchChar$` is not in `sentence$` at all, the loop will be skipped.

`for` loops

As we will soon come to see, the most common type of loop by far involves an *iterator* variable, while the break condition is simply a value this iterator must not exceed.

This could easily be accomplished with a certain type of `while` loop:

Listing 1.11: `for` loop using `while`

```
iterator = 1
while iterator <= 5
  echo 'iterator'
  iterator += 1
endwhile
```

```
$ praat whileFor.praat
1
2
3
4
5
```

However, because it is so common, a streamlined syntax has been provided for this type of loop, namely:

Listing 1.12: `for` loop

```
for iterator from 1 to 5
  echo 'iterator'
endfor
```

```
$ praat forEndfor.praat
1
2
3
4
5
```

The `for` loop takes the variable whose name is provided after the `for`, sets it to the value provided after the `from`, performs all instructions between the `for` and `endfor`, increases the value of the variable by `1`, and repeats, until the value becomes larger than the value provided after the `to`.

In fact, `from 1` is implicitly assumed, so we can even omit that bit if we want to start iterating from `1`. And just as with the `while` loop, if the break condition is true from the start (e.g. `for i from 2 to 1` or something similar), the loop will not be executed even once.

1.6 Arrays

Combining `for` loops with what we learned in Section 1.4.3, we have everything we need for another important concept in Praat scripting: *arrays*.

An array is essentially a group of variables that have names with numbers in them. These variables are usually created within a `for` loop, and later used in another loop. The punchline, however, is that in creating and accessing the variables, the loops' iterators are used *within* the variable names!

So we might have several variables called `value_1`, `value_2`, `value_3`, and so on, and while this in itself is nothing new, it would allow us to do the following:

```
numberOfValues = 3
sumOfValues = 0
for i to numberOfValues
  sumOfValues += value_'i'
endfor
```

So what's going on? In the first iteration of the loop, `sumOfValues` is increased by the value of `value_1`, in the second iteration, by the value of `value_2`, and in the third and final iteration, by the value of `value_3`.

There are two important limitations here. The first is that we need some variable (such as `numberOfValues` in the example) to keep track of how many variables like `value_1` there are. We have to know this, because we need this number in the break condition of the `for` loop. If we were to try and access something like `value_4`, and that variable had not been previously set, we would tend to get an error.

The second limitation has not been shown, but would have become apparent if we had tried to *output* the respective value within the loop, using `echo`. We

have to evaluate the variable in the argument to the `echo` command, but we would have to *nest* one evaluated variable within another. However:

Listing 1.13: Evaluation nesting problem

```
value_1 = 1
value_2 = 2
value_3 = 3

for i to 3
  echo 'value_'i''
endfor

# let's make things interesting:

value_ = 99

for i to 3
  echo 'value_'i''
endfor
```

```
$ praat nestingProblem.praat
'value_1'
'value_2'
'value_3'
99i''
99i''
99i''
```

As we can see, none of this worked as we hoped. The only solution is to assign the desired variable to a "placeholder" variable, which we then output.

In fact, we can easily create and access "multidimensional" arrays by using loops within loops. Observe:

Listing 1.14: A table of products

```
# create the array

for x to 7
  for y to 7
    product_'x'_'y' = x * y
  endfor
endfor

# access the array to build the table

table$ = ""
for x to 7
  for y to 7
    # this is the placeholder:
    thisProduct = product_'x'_'y'
    table$ = "'table$''thisProduct''tab$'"
  endfor
  table$ = table$ + newline$
endfor

# output the table

echo 'table$'
```

```
$ praat tableOfProducts.praat
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
```

In this script, we additionally see that we can store a long string containing several lines in a single string variable, which is then output with a single `echo` command.[5]

Note that although the variables composing an "array" cannot be addressed as a single entity (unlike in many other programming languages), we will nevertheless uphold the custom of referring to such variables as *elements* of an array, although the array itself is just a mental construct and has no concrete manifestation in the Praat scripting language itself.

1.7 Procedures

Sometimes we will come across a portion of code in a script that occurs several times in the script. It would be desirable to only have to write this code *once* and then refer to it again as needed. This is where *procedures* come in.

A procedure is essentially a block of several instructions that are defined and named, and which can then be *called* whenever needed. A call to a procedure simply executes all lines of the procedure at the point where the call is made. Observe:

Listing 1.15: Procedures

```
# define an array of squares
for x to 10
  square_'x' = x ^ 2
endfor

# define a procedure to output this array
procedure output_array
  for x to 10
    square = square_'x'
    printline 'square'
  endfor
endproc

# call the procedure simply with
call output_array
```

No matter where in the script the procedure is defined, it can always be called, before or after the definition, which allows us to banish all tedious procedures to the end of the main script. In fact, it we can even "outsource" blocks

[5] Alright, I admit that we could have computed and output the respective product in a single pass through a double loop, but I was trying to demonstrate array usage, and in the real world, single passes will not always be possible. Just bear with me here!

of code that deal with one aspect of our script into individual procedures and have a very elegant "main" script:

Listing 1.16: More procedures

```
# begin main
call define_array
call output_array
# end main

procedure define_array
  for x to 10
    square_'x' = x ^ 2
  endfor
endproc

procedure output_array
  for x to 10
    square = square_'x'
    printline 'square'
  endfor
endproc
```

1.7.1 Arguments to procedures

Procedures can have *arguments* of their own. These are defined along with the procedure simply by adding them to the `procedure` line. These arguments act as variables in their own right, defined when the procedure is called.

When such call is made, these arguments must be passed to the procedure, and the number and type (number or string) of the arguments must match the procedure definition. Of course the arguments passed can also be variables, but we should realize that they are different from the variables used within the procedure!

Listing 1.17: Procedures with arguments

```
call define_array "squares" 10
call output_array "squares" 10

procedure define_array array_name$ array_size
  for x to array_size
    'array_name$'_'x' = x ^ 2
  endfor
endproc

procedure output_array array_name$ array_size
  for x to array_size
    square = 'array_name$'_'x'
    printline 'square'
  endfor
endproc
```

It is important to remember that every string argument *except the last* must be enclosed in double quotes. This may be slightly confusing, especially when string *variables* are passed to a procedure as arguments, but we should keep in mind that string arguments in a procedure *call* expect *strings*, not string *variables*. Hence:

Listing 1.18: Procedures with arguments passed from variables

```
name_of_array$ = "squares"
size_of_array = 10
call define_array "'name_of_array$'" size_of_array
call output_array "'name_of_array$'" size_of_array

procedure define_array array_name$ array_size
  for x to array_size
    'array_name$'_'x' = x ^ 2
  endfor
endproc

procedure output_array array_name$ array_size
  for x to array_size
    square = 'array_name$'_'x'
    printline 'square'
  endfor
endproc
```

This is equivalent to:

Listing 1.19: Procedures in "plain text"

```
name_of_array$ = "squares"
size_of_array = 10

# this mimics calling the first procedure
array_name$ = "'name_of_array$'"
array_size = size_of_array
for x to array_size
  'array_name$'_'x' = x ^ 2
endfor

# at this point, we have an "array" of 10 variables:
#  squares_1
#  squares_2
#  ...
#  squares_10

# this mimics calling the second procedure
array_name$ = "'name_of_array$'"
array_size = size_of_array
for x to array_size
  square = 'array_name$'_'x'
  printline 'square'
endfor
```

Quoting string arguments

So what happens if we *don't* wrap the string arguments in double quotes? Praat makes assumptions about spaces, which are potentially not what we had in mind. Observe:

Listing 1.20: Procedures with string arguments

```
procedure greet greeting$ name$
  printline 'greeting$',
  printline 'name$'!'newline$'
endproc

# This works, but only because the first string contains no space
call greet Hello Mr. President
```

```
# This no longer works
call greet Happy birthday Mr. President

# Now with too many double quotes
call greet "Happy birthday" "Mr. President"

# Finally, this works just as intended
call greet "Happy birthday" Mr. President

# now the same with variables, which doesn't work
happyBirthday$ = "Happy birthday"
mrPresident$ = "Mr. President"
call greet happyBirthday$ mrPresident$

# because they must be evaluated
call greet 'happyBirthday$' 'mrPresident$'

# but as before, the first, and not the second, in quotes
call greet "'happyBirthday$'" 'mrPresident$'
```

```
$ praat procedures6.praat
Hello,
Mr. President!

Happy,
birthday Mr. President!

Happy birthday,
"Mr. President"!

Happy birthday,
Mr. President!

happyBirthday$,
mrPresident$!

Happy,
birthday Mr. President!

Happy birthday,
Mr. President!
```

1.7.2 Local variables

Normally, all variables declared within a procedure (starting with the procedure-"internal" variables in the procedure definition) are available in the script, as soon as the procedure has been called for the first time. This works just like with normal variables, and these normal variables are referred to as *global* variables.

Within procedures, however, it is possible to declare and use *local* variables, which means that they can be used only within the procedure. Outside the

procedure itself, these variables are unavailable. In Praat, local variables have names that begin with a . (dot).[6]

Listing 1.21: Procedures with local variables

```
name_of_array$ = "squares"
size_of_array = 10
call define_array "'name_of_array$'" size_of_array
call output_array "'name_of_array$'" size_of_array

procedure define_array .array_name$ .array_size
  for .x to .array_size
    '.array_name$'_'.x' = .x ^ 2
  endfor
endproc

procedure output_array .array_name$ .array_size
  for .x to .array_size
    .square = '.array_name$'_'.x'
    printline '.square'
  endfor
endproc
```

Note that the reverse is also true: local variables declared in the "main" part of a script are not accessible within procedures. In fact, this entails that a local variable in the main script and a local variable *with the same name* within a procedure will not overwrite each other and could be used side-by-side, as shown here:

Listing 1.22: Mutually "invisible" local variables

```
.foo$ = "foo"
echo '.foo$'
call bar
echo '.foo$'

procedure bar
  .foo$ = "bar"
  echo '.foo$'
endproc
```

```
$ praat procedures8.praat
foo
bar
foo
```

1.8 Arguments to scripts (part 1)

Just as procedures can receive arguments, the entire script itself can also take arguments, which are provided from the command line exactly as was detailed in the preceding section for procedure calls. This is done with a `form` block.

`form` blocks work slightly differently from the rest of Praat script syntax.[7]

[6] This is the only exception to the rule that variable names in Praat begin with a lower-case letter and consist only of letters, digits and underscores.

[7] This is due to the fact that they seem to have been designed primarily as a means to create custom dialog windows in the graphical version of Praat. We will return to this in a later chapter.

The `form` itself must be followed by a space.[8] Between the `form` and `endform` lines, there may not be any empty lines or comments, only a series of argument ("parameter") declarations. Each consists of the type of the argument (`real` or `text`, for numbers or strings, respectively), a space and the name of variable the argument will have in the script. Since the type is defined by the first part of the declaration, the name of a string variable does not end in a `$`. Let's have an example:

Listing 1.23: Script arguments

```
form
  real howMany
  text greeting
  text name
endform

echo 'howMany' 'greeting$'s, 'name$'!
```

```
$ praat form.praat 100 "Happy birthday" Mr. President
100 Happy birthdays, Mr. President!
```

Quotes around string arguments are handled similarly, but not identically, because the arguments are first split according to the operating system's rules for command line arguments, and then passed to the Praat script. This should not create insurmountable problems, though; if in doubt, just try it out.

1.9 External scripts

Apart from using procedures, there are two other ways to re-use code in Praat scripts: *including* another script and *executing* it.

1.9.1 `include`

The `include` command takes as its only argument the name of another script file. This other script file is then "inserted" into the including script at run time, just as if all lines in the included file had been typed into the including script at the point where the `include` command was issued.[9] Of course, it is possible to include multiple scripts. Note that Praat will perform `include` commands before anything else in the script, so we cannot use a variable to provide the filename of the included script.

Global variables in included scripts will count as global variables in the including script, so take care to check which variable names are used in scripts before you include them, or you might inadvertently overwrite variables in the including script...

The most effective way to use the `include` command is to use it with scripts that contain nothing but procedures, thereby *providing* these procedures to

[8] ... followed by the dialog window's title, which is ignored in command line use.

[9] Praat's behavior in the regard goes as far as counting lines in the including script as if all lines of the included script were actually present in the including script. This means that if Praat gives an error message about something that happens in the including script *after* the `include` command, we will have to subtract the number of lines in the included script from the line number of the error to find the actual line number of the offending command in the including script.

the including script without actually doing much at `include` time. Combining this approach with the use of local variables makes it rather safe concerning accidental variable overwriting.

1.9.2 `execute`

Another way to have one script use another one is the `execute` command. In contrast to the `include` command, this simply runs the executed script from start to finish, then returns control to the executing script and continues with it. No variables are shared or overwritten.

If the executed script takes any arguments (using `form...endform`), these must be provided along with the `execute` command. Passing these arguments works syntactically exactly as passing them from a procedure call (cf. Section 1.7.1) or from the command line.

1.10 File operations

Praat provides a limited number of functions and commands to query, read and write files. But first, a word about paths.

1.10.1 Paths

If a Praat script is to access any file (even another script) that is not in the same directory as the script itself, we have to supply the *path* to the file, either as an absolute or relative path. The exact format of *absolute* paths depend on the operating system under which we're running the script. Table 1.5 gives a few examples. What these absolute paths have in common is that they are fixed; if we move our script to another directory and run it from there, files given with absolute paths will still be found.

Table 1.5: Examples of absolute paths

Windows	`"C:\Documents and Settings\John Doe\Desktop\praat"`
Linux *or* MacOS X	`/home/jdoe/Desktop/praat` *or* `~jdoe/Desktop/praat`
MacOS ≤9	`"My Disk:Desktop:praat"`

However, it is usually preferable to use *relative* paths. These take the script's directory as the base, and work from there. So if we have our script in a directory, along with a subdirectory called `"Sounds"` containing some sound files (e.g. `abc.wav`) which we want to access with our script, we would simply precede references to these files with the name of the directory, followed by a forward slash `/` (e.g. `Sounds/abc.wav`).

The main advantage of using relative paths is *portability*. We can move the script and the relevant subdirectories to another location (directory or disk), and everything will work just as before. Also, since relative paths in Praat scripts always use forward slashes, scripts are even portable across different operating systems.

1.10.2 File I/O

File input and output ("I/O") is extremely easy in Praat scripts. The only thing we need is a string variable and the relevant I/O operator, `<`, `>`, or `>>`.

Reading a file

To read the entire contents of a text file into a string variable, use the `<` operator.

Listing 1.24: A text file

```
This is a text file containing several "sentences",...

...an empty line, and some numbers, separated by tabs:
        123     456.67  89000
```

Listing 1.25: Praat script to read a text file

```
foo$ < foo.txt

# The following expression is now true:
foo$ == "This is a text file containing several ""sentences"",..."
... + "'newline$''newline$'...an empty line, and some numbers, "
... + "separated by tabs:'newline$''tab$'123'tab$'456.67'tab$'89000"
```

Writing a file

To write the contents of a string variable to a text file, use the `>` operator instead. Be careful; if the file already exists, its contents will be deleted first!

Appending to a file

Appending to a file uses the `>>` operator and works just like writing, with one exception: if the file already exists, the contents of the string variable is *added* at the end of the file.[10]

Another way to append text (not just string variables) to a file is the `fileappend` command. This command is followed by the filename, and everything after that (to the end of the line) is treated as the string to be appended. This works similarly to the `echo` command. If the filename is stored in a string variable, that variable must be evaluated *and enclosed in double quotes*.

```
greet$ = "Hello"
fileappend hello.txt 'greet$' World!
```

```
$ cat hello.txt[11]
Hello World!
```

[10] In fact, `>` and `>>` behave exactly as the respective output redirection commands in Windows/DOS and Linux.

[11] `cat` is a Linux tool that can print the contents of files to the screen. The equivalent Windows/DOS command is `type`.

1.10.3 Deleting files

A file can be deleted simply by using the `filedelete` command, followed by the name of the doomed file. If the file does not exist, the command has no effect. `filedelete` can be useful in combination with `fileappend`, in case we want to write more text than just the contents of a string variable to a file, but don't want that file's previous contents (if any) to survive.

1.10.4 Checking file availability

Sometimes it is important to know whether a certain file exists. For instance, trying to read a file that isn't there will usually cause an error. In such cases, we can use the `fileReadable` function to have our script check for the file's existence first. The only argument to this function is the filename (as a string; a string variable should not be evaluated here!), and the function returns a boolean (i.e. `1` if the file can be read, `0` otherwise).[12] See Section 1.11.1 for an example.

1.11 Refined output

The `echo` command is not the only way to print text to the screen. There is also the `printline` command, which is essentially equivalent as long as we are using Praat scripts from the command line.

If we don't want to have the automatic line break at the end of an output command, we can use the `print` command. This allows us to print some text to the screen, then do something else, and print some more text *into the same line* as the last text we printed. Hence, `printline hello` is equivalent to `print hello'newline$'`.

1.11.1 Controlled crash with `exit`

If we want to abort the script for any reason, we can issue the `exit` command. Any further text in the same line will be printed to the screen, in addition to Praat's standard error message. This allows us to terminate a script early on, before a more serious error can occur, which can be a good thing e.g. in case a script argument is not what we intended. It also allows us to inform the user about the reason for the `exit` command.

Listing 1.26: Catching an exception with `exit`

```
# filename argument received from command line
form
  text filename
endform

# no filename received?
if filename$ = ""
  exit no input file specified!
# filename reveived, but file not found?
elsif not fileReadable(filename$)
  exit input file "'filename$'" not found!
endif
```

[12] As the function's name implies, `fileReadable` will also return `0` if the file exists, but we don't have permission to read it, which can occur on Linux type filesystems.

```
# read file
filetext$ < 'filename$'

# just print file contents to screen
print 'filetext$'
```

```
$ praat exit.praat
Error: no input file specified!
Script "exit.praat" not completed.
Praat: command file "exit.praat" not completed.

$ praat exit.praat noFile
Error: input file "noFile" not found!
Script "exit.praat" not completed.
Praat: command file "exit.praat noFile" not completed.
```

If all we want to do is make sure the script does not continue unless a certain condition is met, we can use the much shorter command `assert`. This command is followed by a statement, and if that statement is false, Praat will terminate the script with a standard error message. Using `assert` is much quicker than checking for conditions explicitly and using `exit`, but the tradeoff is that we cannot change the format of the error message:

Listing 1.27: Catching an exception with `assert`

```
form
  text filename
endform

assert filename$ <> ""
assert fileReadable(filename$)

filetext$ < 'filename$'

print 'filetext$'
```

```
$ praat assert.praat
Error: Script assertion fails in line 5 (false):
   filename$ <> ""
Script "assert.praat" not completed.
Praat: command file "assert.praat" not completed.

$ praat assert.praat noFile
Error: Script assertion fails in line 6 (false):
   fileReadable(filename$)
Script "assert.praat" not completed.
Praat: command file "assert.praat noFile" not completed.
```

1.12 Self-executing Praat scripts

It is possible to have scripts run by themselves without explicitly calling the `praat` command and passing the script as the first argument. Depending on the operating system, the procedure to set this up can vary.

Note that this is essentially a cosmetic feature and intended only for advanced users.

1.12.1 Linux

Under Linux and similar operating systems, we need two steps to make a script self-executing:

1. add a special line at the top of the script[13] containing the path to the `praat` program
2. make the script file executable by modifying its file permissions

Below is an executable version of `helloWorld.praat`:

Listing 1.28: "Hello World!" in Praat, executable

```
#!/path/to/praat
echo Hello World!
```

```
$ chmod +x helloWorldExe.praat
$ ./helloWorldExe.praat
Hello World!
```

1.12.2 Windows

In Windows, we can make Praat scripts self-executing by configuring the *file association* of "PRAAT Files" (i.e. files whose name ends with `.praat`, the "filename extension") so that they are automatically opened with the `praatcon.exe` program. The exact procedure depends on the version of Windows, as well as several other factors too Windows-specific to be listed here, but usually involves double-clicking a script file and taking it from there.

Note that while we should now be able to run a Praat script simply by double-clicking it, it will open a command prompt window to run the script and close this window again automatically (configuring Windows to keep the window open for review can be tricky.)

However, we *can* now simply enter the script filename on the command line, and Windows will automatically use `praatcon.exe` to run the script:

```
> helloWorld.praat
Hello World!
```

Note that Windows classifies files exclusively by filename extension, so if you use a different extension for Praat script files (such as `.psc` or `.script`), you will have to modify your file type settings accordingly.

[13] This must indeed be the first line of the script and consist of a `#!`, followed by the absolute path to the `praat` binary. This works exactly as with bash, perl, python, and similar scripts.

1.13 System calls

The following is also relevant only to advanced console jockeys.

It is possible to have Praat make a *system call* to the operating system, executing a command that would normally only be usable on the command line. Since this depends entirely on the operating system under which the Praat script is being executed, the possibilities are far beyond the scope of this introduction. The command for making such system calls is `system`, the rest of the line being interpreted by the operating system. In case a system call could return an error, we can instead use the `system_nocheck` command to keep the Praat script from terminating at that point.

As an afterthought, there is also a way to make Linux-type environment variables available to a Praat script, by using the `environment$()` function, which takes a single string argument, the name of the environment variable, and returns its value. So under Linux, `environment$("PWD") == shellDirectory$`.

Chapter 2

Praat GUI

While we can theoretically accomplish a lot with command line use of Praat scripts, the full set of Praat features is available only through the Graphical User Interface ("GUI"). Praat is obviously much more than a script interpreter; its main focus lies in phonetic analysis, and for this, we need visualization and editing capabilities. In fact, there are hundreds of Praat commands that only make sense when we work with object selection, which is entirely hidden and non-interactive if we use Praat from the command line. The only way to discover these commands (and their arguments) is to work with Praat graphically, and even if a script is designed to be run from the command line, it is almost always developed graphically first.

We should keep in mind, though, that calling scripts from the command line is more efficient (i.e. faster) when processing large amounts of data or complex computations, and so such "batch processing" scripts should be designed with command-line use in mind.

2.1 Object Window

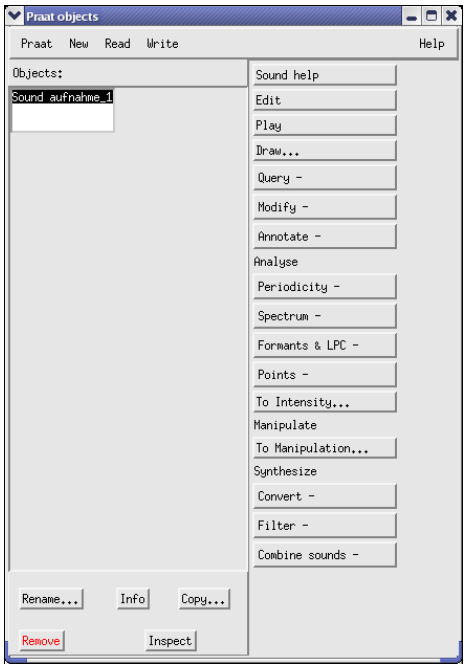
The graphical interface of Praat is started by executing the `praat` program with no argument. Under Windows, it is actually a different program, `praat.exe`, as opposed to the command line only version, `praatcon.exe`.

When Praat starts, we see *two* windows, the *Object Window* ("Praat objects") and the *Picture Window* ("Praat picture"). For now, we will ignore the Picture Window. In fact, we can close that window for now.

There are essentially four areas of the Object Window which demand explanation:

1. the *menu bar* at the top of the Object Window, consisting of the *Praat*, *New*, *Read*, and *Write* menus
2. the *object list*, entitled "Objects", is where objects can be added, selected, and removed
3. the *dynamic menu* to the right of the object list, containing a number of buttons and button menus; its contents changes according to type and number of objects selected in the object list (if none are selected, the dynamic menu will be empty)

Figure 2.1: Praat Object Window in Linux/KDE, with a Sound loaded



4. the area below the object list, which has no proper name, but always contains the buttons `Rename...`, `Info`, `Copy...`, `Remove`, and `Inspect`, which can be applied to all types of objects

2.1.1 Menu bar

The entries in the menu bar are all Praat commands, and mostly *static*. This means that (with the exception of the *Write* menu) they can be used regardless of the contents and state of the object list. Those that cannot be used at a given time will be visible, but disabled ("grayed out").

2.1.2 Objects

All objects in Praat appear in the object list until they are removed or Praat is closed. Each object entry consists of that object's *class* and its *name*. The class of the object can be `Sound` or `TextGrid` or something else. The name can consist of any sequence of letters, digits, and underscores.[1] Any other character supplied as part of an object name will be converted to an underscore. It is possible, though potentially confusing, to have more than one object with the same name, even when the class is the same.

For this reason, Praat uses unique internal *ID numbers* to keep track of the objects in the object list. The first object placed in the list after Praat has been

[1] Unlike scripting variables, object names *can* begin with an uppercase letter, digit, or underscore.

started gets the ID 1, the second, 2, and so on. If an object is removed, that object's ID is *not* freed up for re-use; Praat's internal counter assigning IDs is never reduced.

It is fairly obvious that objects can be renamed with the `Rename...` button and duplicated with the `Copy...` button. What is not so obvious is that the *order* of objects in the list can never be modified. This entails that an object will always have a higher ID than objects above it in the list, and a lower ID than objects following it.

Object selection

In the Object Window, objects are *selected* by clicking on them with the mouse. Any previous selection is *deselected*. We can also "drag" the mouse pointer over several objects to select them all. Alternatively, holding the Shift key while clicking an object will select that objects, as well as all other objects between that object and the current selection, while holding the Ctrl key and clicking an object will add only the clicked object to the current selection. Holding these keys can of course be combined with dragging the mouse pointer.

All currently selected objects are collectively referred to as the current *selection*.

Removing objects from the object list is done with the `Remove` button, which removes all currently selected objects.

2.1.3 Dynamic menu

The contents of the dynamic menu depends entirely on the current selection. Selecting a single object will show all available commands for that class of object, but selecting multiple objects will usually decrease the number of available commands, in many cases down to none. Sometimes, however, certain commands will become available only if a specific combination of objects is selected. In Section **??**, we will see how this specification works when we learn how to manipulate the dynamic menu and add custom buttons. If no object is selected, the dynamic menu will also be empty.

2.2 Script Editor

By choosing the command `New Praat script` from the *Praat* menu, we can open a fresh *Script Editor* window. This is where scripts are developed and run in the graphical version of Praat.

The Script Editor is a simple text editor, lacking many of the fancy features present in full-fledged editors but containing a few features specific to Praat.

We can write a new script, save it, or load a previously saved script from a file (using the appropriate command from the *File* menu). The `Where am I?` and `Go to line...` commands in the *Search* menu return the number of the line the cursor is on, or send the cursor to the specified line, respectively.

2.2.1 Running scripts

To have Praat execute the script currently in the Script Editor, select the `Run` command from the *Run* menu. Additionally, we can also select only a portion

of the script and use `Run selection` command to have Praat execute only the selected lines of the script, ignoring all others.[2]

2.2.2 Command history

A unique feature of the Script Editor is its access to Praat's *command history*. Praat records every click on an object, button or menu entry, and they can all be retrieved with the Script Editor's `Paste history` command, found in the *Edit* menu. Note that the entire command history will be inserted at the current cursor location and usually contains many more commands than we need, many of them selection commands. We can, however, use the `Clear history` at any time to erase all recorded commands and begin anew.

The history mechanism can be quite useful and instructive to scripting beginners, because it outputs everything as a well-formed script which, if run, does *exactly* what the user did up to the point of the `Paste history` command. The drawback is that the power of such scripts is very limited. The history's contents is simply a batch of commands, one after the other, and makes no use whatsoever of variables, loops, or more advanced techniques. Therefore, a script "written" exclusively with the history mechanism will seldom enhance productivity compared to doing everything manually. On the other hand, if in doubt of the correct syntax for a command with many different arguments, the easiest solution is to use the command once and then noting the command history's last entry.

2.3 Output

Since we can no longer receive output on the terminal ("standard out") in the Praat GUI, there are other analogous strategies, and even some new ones, to output information.

2.3.1 Info Window

A window that is initially not visible but that will appear when needed is the *Info Window* ("Praat: Info"). It looks just like another text editor window, and you can even type into it and delete text and so forth, but this window is where Praat directs almost all of its output. Whenever a command is used that returns output, that output will appear in the Info Window. Note that every time this happens, the previous contents of the Info Window will be deleted.

The contents of the Info Window can also be cleared by hand (usind the `Clear` command from the *File* menu), or saved as a text file, or copied, etc. The Info Window can also be closed; it will reappear as required.

In scripts, the Info Window can be cleared with the `clearinfo` command. The Info Window is also where the output of `echo`, `printline`, and `print` will be displayed in the Info Window as well. This is also where the difference between `echo` and the two `print` commands is finally explained; the former will clear the

[2]Note that any variables declared before the selection start will not be available, so this approach is of limited use. To debug a script, make liberal use of comments to disable various lines.

Info Window before writing to it; the latter two will only append to it. This means that

```
echo
```

is equivalent to

```
clearinfo
printline
```

or

```
clearinfo
print 'newline$'
```

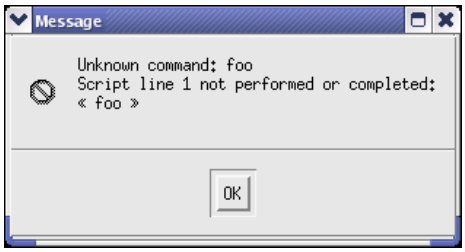
Beware of accidentally overwriting your script's output with multiple `echo` commands; this can become the cause of a lengthy and frustrating bug hunt! Conversely, if you use only `print` commands, you may end up not seeing your script's output as it becomes appended below the visible edge of the Info Window. We can avoid this with a single `clearinfo` at the beginning of the script.

The contents of the Info Window can also be appended to a text file with the `fappendinfo` command, which works similarly to the `fileappend` command (cf. Section 1.10.2).

2.3.2 Error messages

Not all output is written to the Info Window. The other way Praat can give us feedback is through *messages*. These appear as small pop-up windows and usually give us some sort of warning or error message. This is how Praat tells us about errors in a script, for instance. If we use the `exit` command (cf. Section 1.11.1) in a script, it will also generate such a message window.

Figure 2.2: Error message about faulty scripting command



2.3.3 Other forms of output

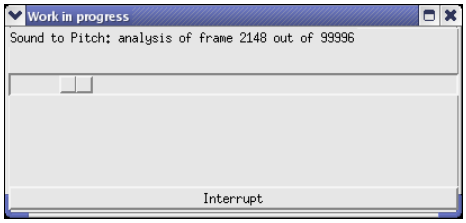
Another way to give feedback to the user during a script is to use the `pause` command, which works similarly to `exit`, but simply displays our text, along with two buttons, "Continue" and "Stop". As expected, the former will let the script continue, the latter will abort. This raises interesting possibilities in script usability design but should not be overused. Note that this command is ignored in command-line use.

Some commands in Praat are expected to take relatively long to complete. For instance, creating a Pitch object from a Sound will take longer, the more

Figure 2.3: Error message about faulty Praat command

samples must be processed. In such cases, Praat will show a *Progress Window* which allows some estimate of how long the command will take to complete. There is also an *Interrupt* button in the Progress Window, which allows us to abort the process (which is useful in case we e.g. want to modify some command parameters to decrease processing complexity).

Figure 2.4: Progress Window showing `To Pitch...` process



2.4 Objects in scripts

A Praat script can select objects and run available commands ("buttons") just as easily as if we used the mouse to do everything by hand, but very much faster! In fact, most scripts will perform such "actions" in the blink of an eye.

2.4.1 Object selection commands

To select an object with a script, we use the `select` command, which is equivalent to clicking on the object. Of course we have to supply an argument to the command specifying which object should be selected. This can either be the object's class *and* name (separated by a space), or its ID. So if we have a Sound object named `My_Recording` in the object list, we can select it with a script with the command `select Sound My_Recording` . Of course, nothing prohibits another Sound with the same name from existing in the object list, and in cases of ambiguity, the *last* object will always be selected.

For this reason, it is generally preferable to use the `select` command with object *IDs* instead of names, in which case the object class is omitted. So if the `Sound` named `My_Recording` that we want to select has the ID `44` (being the

44^{th} object placed in the object list since program start), we can have the script select it simply with the command `select 44`.

To select more than one object at once, we must *add to* an existing selection, using the command `plus`, which otherwise works just like `select`. If the object happens to be already selected, `plus` does nothing. To *remove* an object from the *selection*, use the `minus` command. Again, if the specified object is not selected anyway, `minus` does nothing. Note that we can use `minus` to deselect the last object in the selection, thereby clearing the selection. Likewise, we can use `plus` even if no object is currently selected.

To simply select all objects in the object list at once, use the command `select all`.

2.4.2 Querying selected objects

So how do we find out the name of an unknown object, let alone the internal ID (for which there doesn't seem to be a proper command)? We use one of two functions, `selected$()` or `selected()`. Notice how the first returns a string and the second, a number. These return values will be the selected object's class and name, or ID, respectively.

There's more to these functions, however. If the selection contains more than one object, we can pass either, or both, of two arguments. The first is the class of the object we're interested in (passed to the function as a string), in which case `selected$()` will return only the object's name, and the other is a number. This number n returns the name or ID of the n^{th} object in the selection, starting from the top.[3] If we want to count from the bottom, we simply specify a negative n argument.

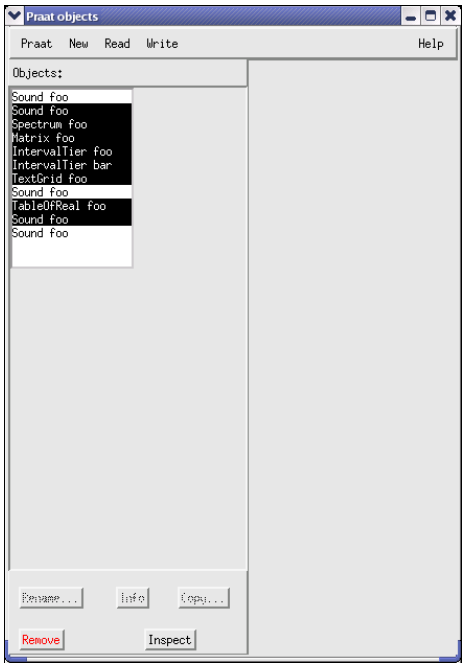
To get the number of selected objects, use the function `numberOfSelected()`, and to get only the number of selected objects of a certain class (presumably from a selection also containing objects of other classes), provide this function with the desired class as a string argument.

Time for a few examples (which assume we have a selection corresponding to Figure 2.5 and no objects have been removed since Praat was started):

```
name$ = selected$()
# outcome: "Sound foo"

id = selected()
# outcome: 2

secondObject$ = selected$(2)
# outcome : "Spectrum foo"

secondID = selected(2)
# outcome: 3

secondSoundName$ = selected$("Sound", 2)
# outcome: "foo"

secondSoundID = selected("Sound", 2)
# outcome: 10

lastIntervalTierName$ = selected$("IntervalTier", -1)
# outcome: "bar"
```

[3] In fact, `selected()` is simply shorthand for `selected(1)`.

Figure 2.5: Praat Object Window with various objects selected




```
thirdToLastObject$ = selected$(-3)
# outcome: "TextGrid foo"

firstIntervalTierID = selected("IntervalTier")
# outcome: 5

secondToLastIntervalTierID = selected("IntervalTier", -2)
# outcome: 5

seventhObjectClass$ = extractWord$(selected$(7), "")
# outcome: "TableOfReal"

numberOfSelectedObjects = numberOfSelected()
# outcome: 8

numberOfSelectedSounds = numberOfSelected("Sound")
# outcome: 2
```

Applying this to what we already know about arrays, we could easily store the IDs of all selected object in an array, to later recall the initial state of the selection:

Listing 2.1: Store IDs of selected objects in array

```
obj_num = numberOfSelected()
for o to obj_num
  obj_'o'ID = selected(o)
endfor
```

2.5 Praat command syntax

Notice how all menu commands and buttons in the various Praat windows begin with a capital letter or digit. This is the exact opposite of the scripting commands we have seen so far, which all begin with a lower-case letter. In general, the scripting commands are only available in scripts while the Praat commands beginning with a capital letter (or digit) can also be clicked on by hand when using Praat graphically and interactively.

2.5.1 Praat commands in scripts

We can use *all* of Praat's commands in scripts. However, we have to make sure that the command is available (i.e. visible and not grayed out) at the point in the script where it is used. Otherwise we will get an error message about the command's unavailability (cf. Section 2.3.2).

When we use such a command, we have to take special care to type it on its own line in the script, *exactly* as it appears on the button or in the menu. That means we have to pay extra special attention to capitalization, spaces, and other characters (such as parentheses, numbers, etc.). Otherwise, we'll get an error.

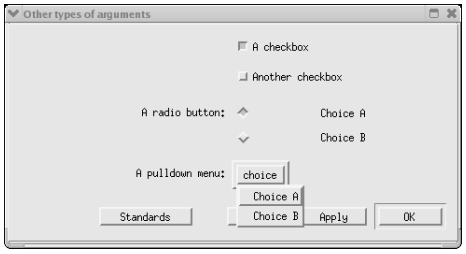
Arguments to Praat commands

There are many Praat commands that pop up *Dialog Windows*, asking for arguments of certain types. These commands invariably end in ... (three dots), which is Praat's indication that arguments must be supplied. When such a command is called in a script, the arguments must be given after the command, in the same line, separated by *single* spaces. This works similarly to arguments to procedures (cf. Section 1.7.1), with a few differences regarding double quotes and variable evaluation:

- Numeric arguments to Praat commands may, but don't have to be enclosed in double quotes.
- Numeric variables supplied as numeric arguments may, but don't have to be evaluated.
- String arguments to Praat commands may, but don't have to be enclosed in double quotes, with two exceptions:
  1. string arguments containing a space *must* be quoted;
  2. the last argument must *never* be quoted, even if it is a string containing a space!
- Variables supplied as string arguments (or parts of string arguments) to Praat commands must always be evaluated.

Some Praat commands may require other types of arguments, namely checkboxes, radio buttons, or pulldown menus:

Figure 2.6: Example of other argument types



A checkbox is essentially a boolean, either on or off, true or false, and hence, a checkbox argument can be supplied as either `1` or `0`.[4] However, we can also use `yes` and `no` instead, respectively.

Radio buttons and pulldown menus are essentially identical, except in appearance. Their arguments are strings and must be passed as exactly as the respective buttons or menu entries are presented in the dialog.

Assuming there were a Praat command called `Other types of arguments...` and Figure 2.6 displayed its dialog and the accompanying arguments, the following example illustrates its syntax in a script:

```
# this works
Other types of arguments... 1 0 "Choice A" Choice B

# this works as well
Other types of arguments... yes no "Choice A" Choice B

# this would NOT work
Other types of arguments... 1 0 "Choice A" "Choice B"
# because there is no pulldown menu item ""Choice B""

# and neither would this
Other types of arguments... 1 0 Choice A Choice B
# because the radio button would receive the string argument
# "Choice" and the pulldown menu "A Choice B"
```

If you have trouble figuring out the correct scripting syntax for a command with complex arguments, remember the Command History (cf. Section 2.2.2)!

Redirecting output into variables

Every Praat command that outputs some form of information to the Info Window can have its output *redirected* and assigned to a variable. This variable will be a string, except if it begins with a number. In this case, it can also be assigned to a numeric variable, but everything after the number (usually a unit of measurement in the command output) will be removed.

```
duration$ = Get total duration
# outcome: "5 seconds"

duration = Get total duration
# outcome: 5
```

[4] Note that the distinction is not just between `0` and `not 0` as with scripting booleans, but between `0` and `1`; any other numeric value is not allowed here.

Trying to assign non-numeric output to a numeric variable will result in an error.

Suppressing warnings and progress dialogs

Sometimes Praat will display a warning or error message, or a progress window. Assuming we know what we are doing, we may find it undesirable to have this kind of output during execution of a script. If a command might output a warning message, we can prefix the command with the `nowarn` command. To suppress an error message, use `nocheck`. And to suppress a progress window, use `noprogress`.

```
# stereo files read normally issue a warning and are read as mono
nowarn Read from file... mySoundWhichMightBeStereo.wav

# no progress window regardless of how long this will take
noprogress To Pitch... 0 75 600

# even if there is no object selected
nocheck Remove
```

`nocheck` can cause serious problems if used incorrectly. Do not use it unless you can be sure of what will happen, and that the error is something non-critical. Even then, there might be better ways to accomplish it.

2.6 Editor scripting

The only Praat commands easily available to a script are those in the Object and Picture Windows. This means that initially, all commands in the various Editor Windows are unavailable. Thankfully, there is a way for a script to "enter" an Editor Window and use all commands available there. This is accomplished via an `editor` block.

Listing 2.2: Enter and use Sound Editor window

```
# make sure we have exactly one Sound selected
assert numberOfSelected() == 1
assert extractWord$(selected$(),"") == "Sound"

# remember the Sound's name...
soundName$ = selected$("Sound")

# create the Editor Window
Edit

# enter the Editor Window named for the Sound
editor Sound 'soundName$'

  #
  # do things in Editor Window
  #

  # close Editor Window
  Close
endeditor
```

The `editor` statement takes two arguments, the class and name of the object being edited. These can be easily seen in the title bar of the Editor Window

itself, but for a script to use these dynamically, we have to query the object as described in Section 2.4.2.

Note that while in the `editor` block, *only* the commands in the Editor Window are available for scripting; Praat commands in the Object and Picture Windows are not available again until after the `endeditor` statement. Also note that editor scripting is not possible when running Praat scripts from the command line.

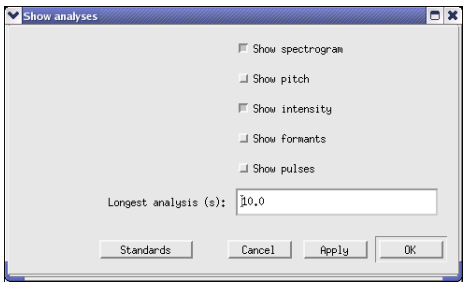
2.6.1 Sound Editors

Commands in Editor Windows that display a Sound's oscillogram ("waveform") and (optionally) its spectrogram, intensity, pitch, formants, and glottal pulses can be difficult to use in scripts. This is due to the fact that only the visible analysis components are available to the commands, while the commands usually depend on the current position of the *cursor*. This means that three things play a role here:

Visibility of analysis

To ensure that a certain analysis is visible, we can use the `Show analyses...` command from the *View* menu with appropriate arguments.

Figure 2.7: `Show analyses...` dialog



Additionally, the "Longest analysis (s)" argument determines the maximum length of the viewed part of the Sound. If the current view shows more than this, *none* of the analyses will be visible, and commands such as `Formant listing` will fail with an error message.

Zoom

To make sure we view an appropriate part ("window") of the Sound (and that the current view is not longer than the "Longest analysis (s)" (cf. previous Section), we can use the `Zoom...` command from the *View* menu, or commands like `Zoom to selection` (cf. next Section). `Zoom in` is probably not specific enough.

Cursor position and selections

We can also move the cursor to a specified position with the `Move cursor to...` command from the *Select* menu, or we can specify a selection with the `Select...` command. There are several related commands in the *Select* menu that could

be useful in this regard. What is important is that we can control the cursor and selection, which determines the output of other commands such as `View spectral slice` or `Extract visible pitch contour`.

It is important to realize that almost all analyses and extraction commands of an Editor Window are also available as similar commands in the Object Window, usually in *Query* or *Modify* submenus in the dynamic menu. For scripting, it is generally easier and more precise to use the Object window's commands and avoid using the Editor Windows.

2.7 Picture Window

The *Picture Window* is one of the powerful, but commonly underestimated features of Praat. It allows us to produce graphics and illustrations (usually, but not necessarily, based on Objects), which can be helpful for data analysis, and additionally be exported as vector-based image files for insertion into research papers and reports.

2.7.1 Picture Window basics

The Picture Window is essentially an (initially) empty canvas measuring 4×4 squares (delimited by yellow lines), each 3 inches on each side, as indicated by the *rulers* at the canvas edges (which are labeled from 0 to 12). By default, only the left half and top three quarters of this canvas are visible.

In addition, there is a pink *selection* rectangle, which can be created by dragging the mouse. Note that it is not possible to modify this selection by dragging its edges, so the selection behaves much like a selection in a Sound Editor, albeit in two dimensions.

The selection actually consists of two rectangles, the *outer viewport* and the *inner viewport*. It is the area *between* these two viewports that is filled in pink.[5] The inner viewport is where most of the graphics should be created, while the outer viewport serves as an outer guideline for axis labels, titles and things of the sort. The behavior of the mouse with regards to viewport creation, as well as the obligatory precise commands `Select inner viewport...` and `Select outer viewport...` are found in the *Select* menu.

Before we continue, let's have an example of how the viewport determines what will be drawn in the Picture Window. With the default viewport (6×4), the script

Listing 2.3: Create 1kHz sine and draw its spectrum

```
Create Sound... sine_1kHz 0 1 22050 1/2 * sin(2 * pi * 1000 * x)
To Spectrum... no
Draw... 0 0 0 0 yes
```

results in the Picture Window contents shown in Figure 2.10:

The `Draw...` command available for `Spectrum` objects has a number of parameters (cf. Figure 2.8) that determine which portion of the spectrum will be drawn, as well as the scale. The "Garnish" option adds the frame along the

[5] The difference in size between the inner and outer viewports is determined by the currently selected *font size*, see below.

Figure 2.8: `Draw...` dialog

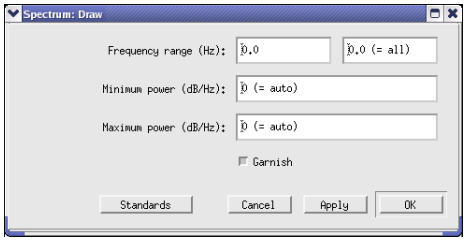
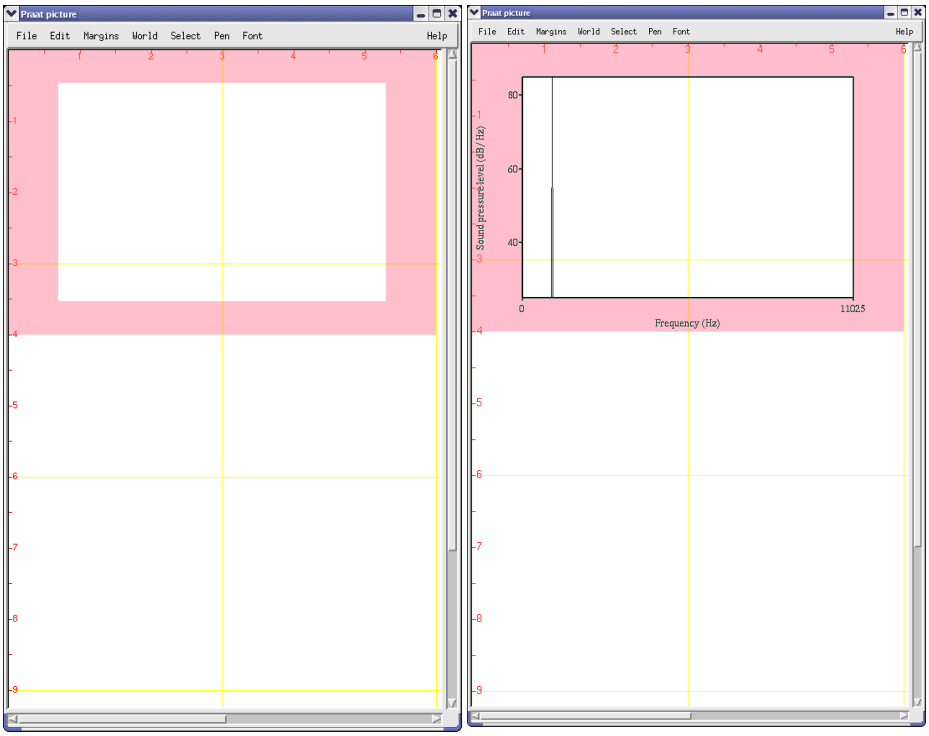


Figure 2.9: Empty Picture Window

Figure 2.10: Result of Listing 2.3



inner viewport edge, as well as the axes' labelings. Notice how these labels were drawn into the area between the inner and outer viewports.

There are many commands such as `Draw...` available for the various object classes, and not many of them leave anything to be desired. Remember that these graphics are not meant to rival the editors, but to present a possibility of exporting analysis data in a perfect format.

Don't use screenshots!

If you ever want to export anything visual from Praat to be included in a research paper or other publication, *do not use screenshots* of an editor window or anything of the sort. Doing so will create a pixel-based image with a res-

olution no higher than that of the screen from which it was captured. Print resolution will almost always be *much* higher, so the image will be blocky or blurry, depending on how it was processed, but never look good.

Also, pixel-based images tend to consume rather large amounts of memory (each pixel is stored individually), unless compression is used. One of the most common types of image compression is JPEG, which, when configured improperly, will introduce artifacts along high-contrast edges. Programs such as Microsoft Word tend to make the worst of such images when it comes to printing.

Additionally, window borders distract from the analysis you're trying to show with your image, and if you want your readers to know that you used Praat, you should state it in the text. Showing additionally that you were running e.g. Windows XP with the "Energy Blue" Theme is not desirable, and the names of files or objects you analyzed are details that are usually irrelevant.[6]

The solution to these issues is to export the contents of the Picture Window to a file that recreates it using *vector graphics*. One such format is *Encapsulated PostScript*, created with the `Write to EPS file...` and its variants, which can be easily converted to any other vector-based format using appropriate software. Another is Microsoft's *Enhanced Metafile* format, which is well suited for insertion into Microsoft Office documents. The required command, `Write to Windows metafile...`, however, is available only in the Windows version of Praat.

Vector images can be enlarged arbitrarily without reducing edges or introducing artifacts, because their components are essentially *continuous functions*, which are sampled and redisplayed optimally whenever they are rendered.[7] Since these components in most cases take up very little memory, vector images are also very efficiently stored. (The exception is a pixel image *within* a vector image, which is, of course, a series of colored squares.)

In fact, the contents of the Picture Window displayed in Figure 2.10 could be exported as an `eps` file and inserted into a LaTeXdocument such as this one directly, with code like this:

```
\begin{figure}
  \includegraphics{spectrum1kHz}
\end{figure}
```

Observe:

2.7.2 Custom drawing commands

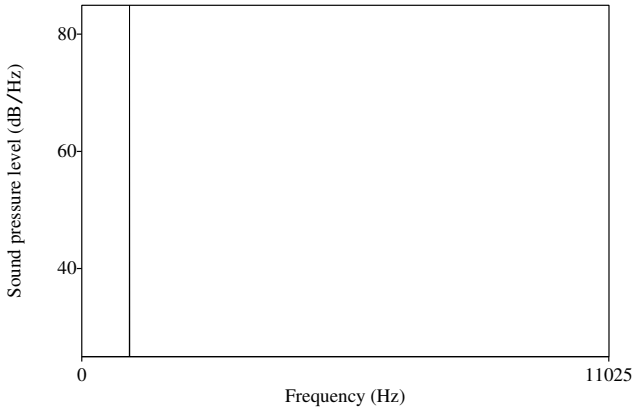
Besides exporting graphics to files for insertion into documents, we can of course draw arbitrary graphics into the Picture Window. There are a number of commands at our disposal, and scripting makes them efficient to use.

Preliminaries

Similar to the Info Window, drawing commands will not clear the Picture Window, so to start with a blank canvas, we can issue the `Erase all` command in the *Edit* menu.

[6] I realize that I'm ranting against everything I've done myself in this document, but I'm trying to focus on the interaction with Praat itself, not the data!

[7] Incidentally, this is conceptually quite similar to digitization of audio signals!



We can try out various drawing commands by hand, and whenever we make a mistake, we can use the `Undo` command (also in the *Edit* menu), which can come in handy.

Most commands that draw lines, shapes are modified by the current settings in the *Pen* menu. These include the line *type* (solid, dotted or dashed) and *width*, controlled with the commands `Solid line`, `Dotted line`, `Dashed line`, and `Line width...`, respectively.

Likewise, Text printed to the Picture Window can be controlled with respect to font *size* (`Font size...`) and *family*: `Times`, `Helvetica`, `New Century Schoolbook`, `Palatino`, and `Courier`, all in the *Font* menu. Several common font sizes can also be specified directly, with the commands `10`, `12`, `14`, `18`, and `24` (which may look strange in a script, on a line all by themselves, but are nevertheless valid Praat commands).

Furthermore, lines, shapes, and text can be colored with the following palette:

Axes and scale

While the rulers along the edges of the Picture Window aid in selecting the viewport's proportions, they have nothing to do with the actual coordinates used to draw objects in the Picture Window. The coordinate system is defined using the command `Axes...` (found both in the *Margins* and *World* menus). This can be arbitrary, and redefined as desired; in fact, the left margin does not necessarily have to be smaller than the right margin, and likewise for top and bottom.

The `Axes...` command takes four numeric arguments, the left, right, bottom, and top values for the coordinate system enclosed by the *inner viewport*. This means that after clicking *OK* in the dialog shown in cf. Figure 2.11, the lower left-hand corner of the inner viewport is the point of origin of a coordinate system spanning to the upper right-hand corner of the inner viewport, which has the position $(1, 1)$.

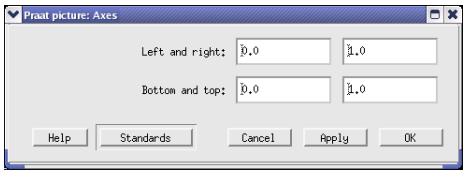
This can easily be illustrated by executing the following commands, which results in Figure 2.12:[8]

[8]It would be tedious to explain every drawing command's arguments from here on, so

Table 2.1: Color commands and their colors

Command	*Color (Linux)*	*Color (Windows)*
`Black`		
`White`		
`Red`		
`Green`		
`Blue`		
`Yellow`		
`Cyan`		
`Magenta`		
`Maroon`		
`Lime`		
`Navy`		
`Teal`		
`Purple`		
`Olive`		
`Silver`		
`Grey`		

Figure 2.11: `Axes...` dialog




```
Marks bottom every... 1 0.1 yes yes yes
Marks left every... 1 0.1 yes yes yes
Draw inner box
```

Now, a few simple drawing commands could be to paint a blue circle with a diameter of 0.2 right into the center of the viewport, then print the text “Earth” in 18pt Courier in the lower right-hand corner and draw an arrow from the text to the circle:

```
Paint circle... Blue 0.5 0.5 0.1
18
Courier
Text... 0.25 Centre 0.25 Half Earth
Draw arrow... 0.3 0.3 0.4 0.4
```

This enriches the Picture Window to look like this:

We could just as well select the viewport to have a different aspect ratio and redefine the axes:

```
Select outer viewport... 0 6 0 6
Axes... -1 1 -1 1
```

please refer to the Praat program to see what the arguments mean.

Figure 2.12: Coordinate system from $(0, 0)$ to $(1, 1)$

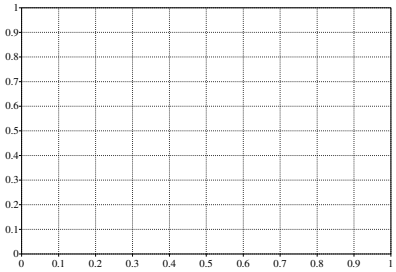
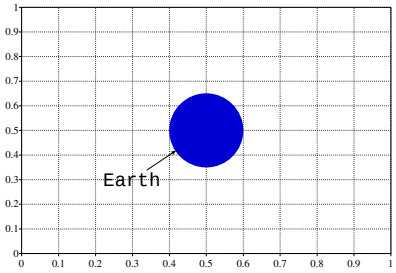


Figure 2.13: A few things drawn in




```
Marks bottom every... 1 0.1 yes yes yes
Marks left every... 1 0.1 yes yes yes
Draw inner box

Paint circle... Blue 0.5 0.5 0.1
18
Courier
Text... 0.25 Centre 0.25 Half Earth
Draw arrow... 0.3 0.3 0.4 0.4
```

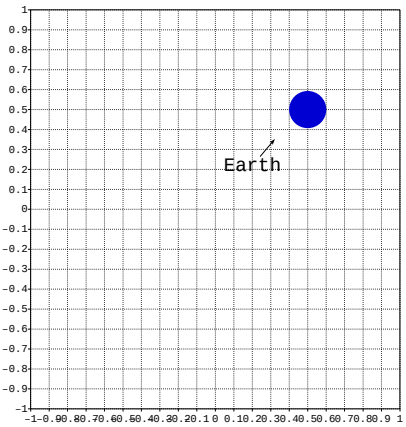
Which results in:

The point of being able to define and redefine the axes at will is that various datasets can be drawn without having to first scale the values to some fixed coordinate system.

Note that even though the axes are defined with reference to the inner viewport, things can still be drawn outside of the inner viewport, but tend to look messy.

So now we know everything we need to put the Picture Window to good use!

Figure 2.14: Same as Figure 2.13, but with a different scale



2.7.3 Data analysis with the Picture Window

Where Praat's analysis commands don't offer what we want, we can easily make our own.

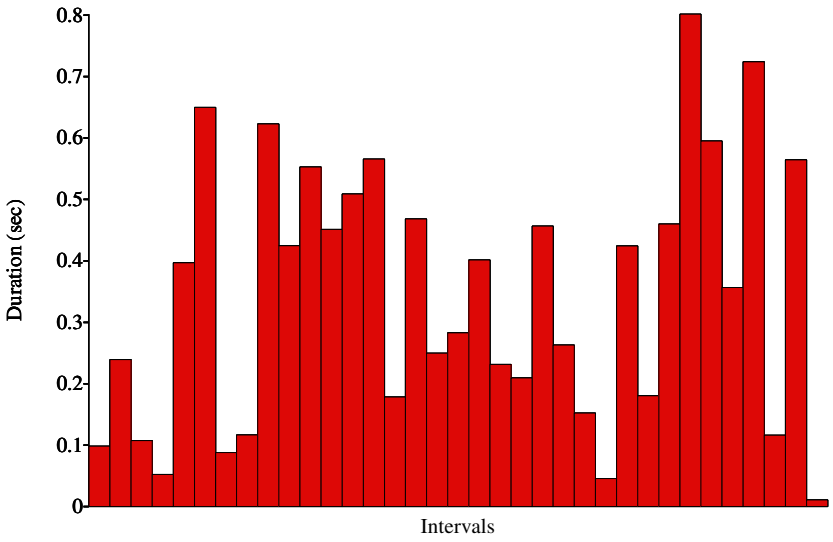
As an example, we will have Praat draw a histogram with the duration of each interval on the first tier ("Word") of `festintro.TextGrid` (cf. Section 3.1).

Listing 2.4: Duration histogram of `festintro.TextGrid`

```
# open TextGrid file (modify as appropriate)
Read from file... festintro.TextGrid

# read interval durations into array
numIntervals = Get number of intervals... 1
for i to numIntervals
  start = Get starting point... 1 i
  end = Get end point... 1 i
  interval_'i'_Duration = end - start
endfor
Remove

# for the vertical dimension, we need to know the maximal duration
maxDuration = 0
for i to numIntervals
  if interval_'i'_Duration > maxDuration
    maxDuration = interval_'i'_Duration
  endif
endfor

Axes... 0 numIntervals 0 maxDuration
for i to numIntervals
  x_left = i - 1
  x_right = i
  y_bottom = 0
  y_top = interval_'i'_Duration
  Paint rectangle... Red x_left x_right y_bottom y_top
  # to make it look nice, draw an outlined rectangle over that
  Draw rectangle... x_left x_right y_bottom y_top
```

```
endfor

# garnish
Draw line... 0 0 0 maxDuration
Marks left every...  1 0.1 yes yes no
Text left... yes Duration (sec)
Text bottom... no Intervals
```

This produces the following Picture Window contents:



Chapter 3

Scripting Techniques

3.1 TextGrid processing

A *TextGrid* is Praat's standard format for labeling Sounds. Apart from the obvious benefit of being able to segment speech into segments, this allows us to analyse portions of longer Sounds without having to extract these portions first.[1] The approach is always the same: Use the TextGrid to mark boundaries (or points in time), then select the actual data (Sound, Pitch, or whatever) and perform analysis on the Intervals or Points marked in the TextGrid.

A TextGrid consist of one or more *tiers*. Each tier is either an *IntervalTier*, which marks *spans* of time ("intervals"), or *TextTier*, which marks *points* in time ("points"). Intervals and points can have *labels* attached to them. The process of marking intervals and/or points and attaching labels to them is known as *labelling*.

There are several possible scripting approaches to use time information stored in a TextGrid for analysis of other objects. For illustrative purposes, we will attempt to analyse a single recording of synthesized speech.[2] The spoken text is the utterance

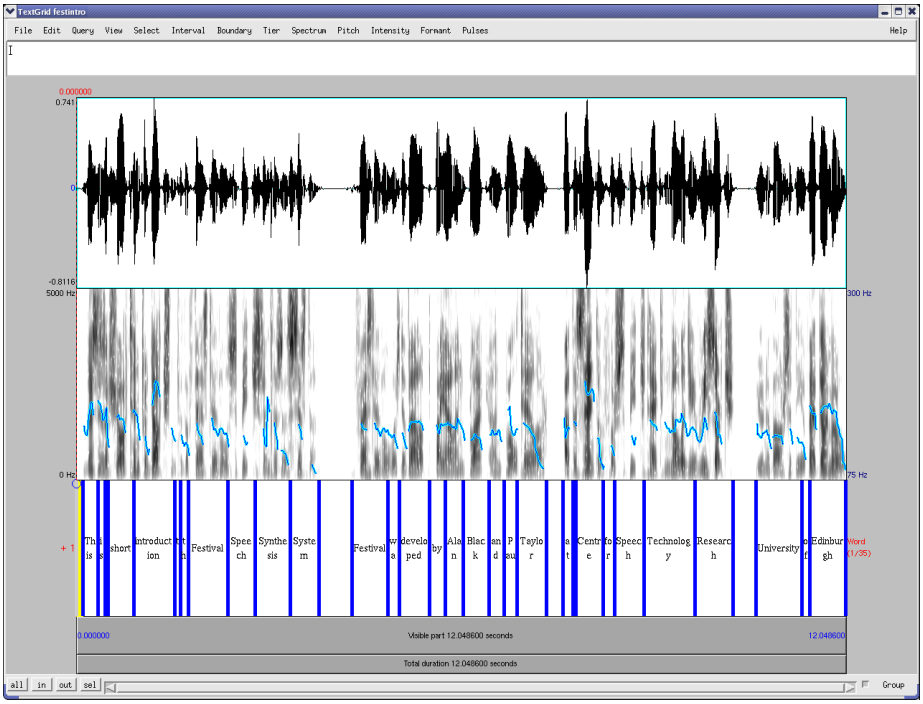
> This is a short introduction to the Festival Speech Synthesis System. Festival was developed by Alan Black and Paul Taylor, at the Centre for Speech Technology Research, University of Edinburgh.

In this Section, we will measure the mean pitch during each word of the recording. Starting point is that we have loaded both the Sound (`festintro.wav`) and the TextGrid (`festintro.TextGrid`) into Praat and selected both. To avoid octave jumps, the pitch ceiling will be lowered to 300Hz, but all other settings remain at their defaults.

[1] This is especially useful where the extraction itself would change data, or where analysis is not possible, e.g. at the extraction edges.

[2] The data we will use here is the result of the `(intro)` command in Festival 1.95:beta, using the `cstr_us_awb_arctic_multisyn` voice. Festival and the voice data are available at festvox.org. The synthesized word boundaries were converted automatically to Praat TextGrid format using in-house scripts.

Figure 3.1: Festival Intro



Manual analysis

The easiest, and most time-consuming, approach to measure the mean pitch for each word of the TextGrid is to click the `Edit` button and use the TextGrid Editor. For this, we would simply click into each non-empty interval on the TextGrid's first (and only) tier, and use the `Get pitch` command from the *Pitch* menu. However, as outlined in Section 2.6.1, we initially have to make sure that the `Show pitch` command is checked. We also have to make sure that the "Longest analysis" setting is larger than the longest interval to be measured and zoom in until the pitch contour is visible.

For every measurement, we must note the result down and can finally type up our results. This can be streamlined slightly by using the keyboard shortcuts Alt+→ (`Select next interval`) and F5 (`Get pitch`).

Using the TextGrid Editor

We can mimic the procedure described in the last Section using the following script:

Listing 3.1: Using the TextGrid Editor

```
clearinfo
textGridName$ = selected$("TextGrid")
Edit
editor TextGrid 'textGridName$'
```

```
  # setup pitch options
  Show analyses... 0 1 0 0 0 5
  Pitch settings... 75 300 Hertz "Intonation (AC method)" automatic

  # select first interval
  Move cursor to... 0
  firstEnd = Get end point of interval
  Select... 0 firstEnd

  # then, for each interval
  repeat
    Zoom to selection
    label$ = Get label of interval

    # if label is not empty
    if label$ <> ""
      pitch = Get pitch
      printline 'label$''tab$''pitch:2' Hz
    endif
    Select next interval
    end = Get end point of interval
  until end == firstEnd
  Close
endeditor
```

The `repeat` loop's break condition exploits the fact that using the `Select next interval` command at the end of a tier will "wrap" back to the first interval on that tier. Note that this script will fail with an error if an interval is longer than the "Longest analysis" argument of the `Show analyses...` command (here, 5 seconds).

Using the Object Window

A much faster and more robust alternative is to use commands from the Object Window, which makes no use of Editor Windows at all:

Listing 3.2: Using the Object Window

```
soundID = selected("Sound")
textGridID = selected("TextGrid")
minus textGridID

# quietly create Pitch object
noprogress To Pitch... 0 75 300
pitchID = selected()

# back to the TextGrid
select textGridID
numIntervals = Get number of intervals... 1
clearinfo

# for each interval
for i to numIntervals

  # if label is not empty
  label$ = Get label of interval... 1 i
  if label$ <> ""

    # get start and end times
    start = Get starting point... 1 i
    end = Get end point... 1 i
```

```
    # get mean between start and end from Pitch
    select pitchID
    pitch = Get mean... start end Hertz

    # back to TextGrid
    select textGridID
    printline 'label$''tab$''pitch:2' Hz
  endif
endfor

# cleanup
select pitchID
Remove
select soundID
plus textGridID
```

Note that the pitch analysis is done by repeatedly querying a Pitch object created from the Sound, retrieving the time parameters from the TextGrid. While this script is much faster than the one in the preceding Section, the "jumping" back and forth between the Pitch and TextGrid objects is rather cumbersome. We could instead use an array to store the times:

Listing 3.3: Using the Object Window and arrays

```
soundID = selected("Sound")
textGridID = selected("TextGrid")
minus soundID

# store intervals in arrays
numIntervals = Get number of intervals... 1
for i to numIntervals
  interval_'i'_label$ = Get label of interval... 1 i
  interval_'i'_start = Get starting point... 1 i
  interval_'i'_end = Get end point... 1 i
endfor

# quietly create Pitch object
select soundID
noprogress To Pitch... 0 75 300
clearinfo

# for each interval
for i to numIntervals

  # if label is not empty
  if interval_'i'_label$ <> ""

    # get mean between start and end
    pitch = Get mean... interval_'i'_start interval_'i'_end Hertz
    label$ = interval_'i'_label$
    printline 'label$''tab$''pitch:2' Hz
  endif
endfor

# cleanup
Remove
select soundID
plus textGridID
```

Direct access via `TableOfReal`

There is another way to access the interval data without creating arrays from the TextGrid. We can exploit the fact that if a `TableOfReal` object is created from an `IntervalTier`, that `TableOfReal` will be a table containing the start and end times, as well as duration, of each interval:

Table 3.1: `TableOfReal` of `festintro.TextGrid` (excerpt)

	Start	End	Duration
	0.	0.0984407914312	0.0984407914312
This	0.0984407914312	0.337637603283	0.23919681185180003
is	0.337637603283	0.445062607527	0.10742500424399998
a	0.445062607527	0.497505914081	0.05244330655399998
short	0.497505914081	0.894330280168	0.396824366087
introduction	0.894330280168	1.54408713749	0.6497568573220001
⋮	⋮	⋮	⋮

An advantage of using this approach is that we can "trim" the `TableOfReal` to contain only those intervals we are interested in, based on their label. We can do this when creating the `TableOfReal` with `Down to TableOfReal...`, or use much more powerful commands such as `Extract rows where label...` on an existing `TableOfReal`.

While we could then query the `TableOfReal` much in the same way as a `TextGrid`, the real advantage is that we can use a special syntax in scripts to access a `TableOfReal` directly, *without selecting it* in the object list!

We can either use the expression `TableOfReal_Foo` to access the `TableOfReal` named "Foo" or, assuming that its ID is e.g. `5`, use the more robust expression `Object_5`. Table 3.2 gives a quick overview of relevant syntax.[3]

Table 3.2: Standard Praat commands vs. direct object access (`TableOfReal`)

`Get number of rows`	`Object_'id'.nrow`
`Get number of columns`	`Object_'id'.ncol`
`Get row label... r`	`Object_'id'.row$[r]`
`Get column label... c`	`Object_'id'.col$[c]`
`Get value... r c`	`Object_'id'[r, c]`
`r = Get row index... Foo` `Get value... r c`	`Object_'id'["Foo", c]`
`c = Get column index... Bar` `Get value... r c`	`Object_'id'[r, "Bar"]`

`r = Get row index... Foo`
`c = Get column index... Bar`
`Get value... r c`
	`Object_'id'["Foo", "Bar"]`

Using this knowledge, we can write a script that queries the `Pitch` object, referring to the `TableOfReal` for time parameters:

[3] For the standard commands to work, the `TextGrid` with ID `id` must be selected. This is not required for direct object access.

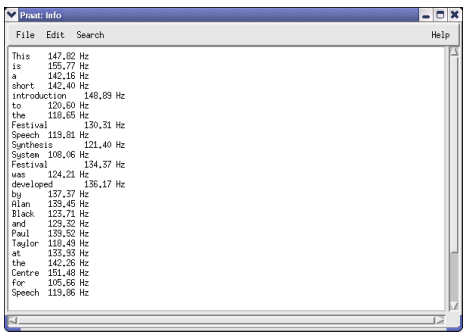
Listing 3.4: Using direct `TableOfReal` access

```
soundID = selected("Sound")
textGridID = selected("TextGrid")
minus soundID

# create and trim TableOfReal
Extract tier... 1
intervalTierID = selected()
Down to TableOfReal (any)
tableOfReal1ID = selected()
Extract rows where label... "is not equal to" 
tableOfReal2ID = selected()

# quietly create Pitch object
select soundID
noprogress To Pitch... 0 75 300
clearinfo

# for each interval (i.e. TableOfReal row)
for i to Object_'tableOfReal2ID'.nrow

  # get mean between start and end
  start = Object_'tableOfReal2ID'[i, "Start"]
  end = Object_'tableOfReal2ID'[i, "End"]
  pitch = Get mean... start end Hertz
  label$ = Object_'tableOfReal2ID'.row$[i]
  printline 'label$''tab$''pitch:2' Hz
endfor

# cleanup
plus tableOfReal2ID
plus tableOfReal1ID
plus intervalTierID
Remove
select soundID
plus textGridID
```

[4]

All of these scripts yield the same results:[5]

Figure 3.2: Pitch analysis script output



[4]Note the space at the end of this line! It means that the *second* command argument is the empty string.

[5]Using the Sound/TextGrid Editor for pitch analysis may produce slightly different measurements, since the default pitch settings can differ from when a `Pitch` object is created from a `Sound`. Also, the visible portion of the Pitch contour influences the measurements.

3.2 Batch processing

As we glimpsed in Chapter 0, it is possible to have Praat list all files in a directory, e.g. to perform an analysis on each one, etc. The key to such *batch processing* of many files in one script is the Praat command `Create Strings as file list...` in the *New* menu of the Object Window.

This command creates an Object of the `Strings` class, which contains all files in a given directory. It is possible to *mask* certain files, including only those in the `Strings` which match a certain condition. For instance, the masking expression `*.wav` will include only those filenames ending with `.wav`,[6] while the expression `a*` will match all filenames beginning with an `a`. The `*` is referred to as a *wildcard* character and stands for "0 or more characters". Another wildcard character is `?`, which stands for "any one character".

The `Strings` object

An object of class `Strings` is conceptually quite similar to an array of strings, or to true arrays in other programming languages. It is simply a number of strings of characters that can be addressed as a single object. Unlike a string array, its lifespan is not restricted to the runtime of a script, but it exists in the Object Window and it can be written to, or read from, a file. There are many useful commands available for `Strings` objects, the most common of which are `Get number of strings`, which speaks for itself, and `Get string...`, which takes a single numeric argument n and returns the n^{th} string in the `Strings` object.

Other useful commands for `Strings` objects are `Sort`, `Randomize`, and `Genericize`[7], all of which modify the stored strings *in place* (i.e. without creating a copy of the object first). It is possible to manipulate individual strings with the `Set String...` command, and replacement operations (allowing the use of *regular expressions*) can be performed with the `Change...` command. Finally, a new `Strings` object containing a contiguous subset of the strings stored in a `Strings` object can be created with the `Extract part...` command.

It is even possible to create a `Strings` object from a text file, with one string per line, using the `Read Strings from raw text file...` command from the *Read* menu. This is another way to read text files into Praat; the following two scripts are practically equivalent, except that the `Strings` object is *persistent* in the object list:

Listing 3.5: Read a text file into an array

```
text$ < file.txt
lines_num = 0
repeat
  lines_num += 1
  line_'lines_num'$ = extractLine$(text$, "") + newline$
  text$ = replace$(text$, line_'lines_num'$, "", 1)
until text$ == ""
```

Listing 3.6: Read a text file into a `Strings`

```
Read Strings from raw text file... file.txt
```

[6] Typically audio samples in PCM format with a RIFF header.
[7] This changes special characters into a 7-bit representation.

```
# optionally
lines_num = Get number of strings
for l to lines_num
  line_'l' = Get string... l
endfor
```

One disadvantage a `Strings` object has over an array of strings is that it cannot be created from scratch, and there is no straightforward way to add strings to such an object. On the other hand, it is comparatively trivial to view the actual strings stored in a `Strings` object, using the `Inspect` command below the object list, while array elements must be printed or otherwise explicitly output, using loops and placeholder variables.

3.2.1 Single directory processing

As mentioned above, to access a list of all files in a given directory (or only those matching a certain wildcard expression), we use the command `Create Strings as file list...`. This takes two arguments; the first is the name of the resulting `Strings` object, and the second is the path (and filename mask) of the directory to be listed. This can be an absolute or relative path.

Listing 3.7: List directory contents

```
form List directory contents
  sentence Directory
endform

echo 'directory$':
Create Strings as file list... fileList 'directory$'
numFiles = Get number of strings
for f to numFiles
  file$ = Get string... f
  printline 'file$'
endfor
```

Of course, we want to *do* something with these files; e.g. reading all `wav` files into the object list as `Sound` objects is a common task:

Listing 3.8: Read all `wav` files in specified directory

```
form Read all sounds in directory
  sentence Directory
endform

Create Strings as file list... wavList 'directory$'/*.wav
numSounds = Get number of strings
for s to numSounds
  sound$ = Get string... s
  Read from file... 'directory$'/'sound$'
  select Strings wavList
endfor
Remove
```

Note two common pitfalls here: First, if the script is not in the `directory$` directory, the files to be read must be preceded with the proper path, i.e. `directory$`. And second, if the `Strings` containing the file list is not selected when the `Get string...` command is used in the `for` loop, the script will fail.

Of course, nothing prevents us from read the contents of `Strings` file list into an array, then loading the files in a second loop:

Listing 3.9: Read all `wav` files in specified directory using an array

```
form Read all sounds in directory
  sentence Directory
endform

Create Strings as file list... wavList 'directory$'/*.wav
numSounds = Get number of strings
for s to numSounds
  sound_'s'$ = Get string... s
endfor
Remove

for s to numSounds
  sound$ = sound_'s'$
  Read from file... 'directory$'/'sound$'
endfor
```

3.2.2 Subdirectory processing

The `Create Strings as file list...` command ignores directory entries that are not files. This means that not only are files in subdirectories of the specified directory not processed, we don't even find these subdirectories in the list.

For this purpose, there is a different command, `Create Strings as directory list...`. It works analogously to `Create Strings as file list...`, but lists only subdirectories, and no files. Therefore, given a directory which contains a number of subdirectories, each of which in turn contains a number of files, we can nest a `Create Strings as file list...` loop in a `Create Strings as directory list...` loop, loading every files in every subdirectory:

Listing 3.10: Read all `wav` files in specified directory's subdirectories

```
form Read all sounds in directory's subdirectories
  sentence Directory
endform

Create Strings as directory list... subDirList 'directory$'
numSubDirs = Get number of strings
for d to numSubDirs
  subDir$ = Get string... d
  Create Strings as file list... wavList 'directory$'/'subDir$'/*.wav
  numSounds = Get number of strings
  for s to numSounds
    sound$ = Get string... s
    Read from file... 'directory$'/'subDir$'/'sound$'
    select Strings wavList
  endfor
  Remove
  select Strings subDirList
endfor
Remove
```

Dot files and directories

There is a caveat when using these commands to process files and directories: Depending on the operating system under which Praat is run, files and directories whose name begins with a `.` (these are sometimes called *dot files* or *dot directories*) may or may not be included in the list. This has two consequences:

1. Under Linux, dot files and directories will be hidden and cannot be accessed with `Create Strings as file list...` and `Create Strings as directory list...`, respectively. However, one possible workaround is to use a `bash` script to create a directory listing as a text file, then read this file as a `Strings` object and proceed normally:

Listing 3.11: Similar to Listing 3.7, but reads even dot files

```
include createStringsAsFileList.praat

form List directory contents
  sentence Directory
endform

echo 'directory$':
# procedure call instead of Create Strings as file list...
call createStringsAsFileList fileList 'directory$'
numFiles = Get number of strings
for f to numFiles
  file$ = Get string... f
  printline 'file$'
endfor
```

Listing 3.12: Procedure with embedded `bash` script

```
procedure createStringsAsFileList .stringsName$ .path$
  # make sure .path$ ends with "/"
  if not endsWith(.path$, "/")
    .path$ = .path$ + "/"
  endif
  # system call with embedded bash script...
  system
  ... for f in $(ls -AU '.path$');
  ... do
  ...   if [ -f '.path$'$f ];[8]
  ...   then
  ...     echo $f;
  ...   fi;
  ... done
  ... > '.stringsName$'
  # ...which only works under Linux, of course
  Read Strings from raw text file... '.stringsName$'
  filedelete '.stringsName$'
endproc
```

```
$ praat listFiles.praat bla
bla:
foo
bar
baz
$ praat listAllFiles.praat bla
bla:
foo
bar
baz
.hidden
.invisible
```

[8] To modify this `procedure` to list directories instead of files, change the `-f` to `-d`.

2. Under Windows, dot files and directories are shown and processed normally. However, this also means that every directory will contain two "special" directory entries, `.` (the directory itself) and `..` (the parent directory). These are part of the file system, but can cause problems in Praat scripts if they are treated as normal directories, since `Create Strings as directory list...` will include them as extra strings. It is fairly trivial to exclude them from being processed, however, by wrapping relevant lines in a condition (cf. Listing 3.13).

3.2.3 Recursive subdirectory processing

As illustrated in Section 3.2.2, processing subdirectories can quickly become rather awkward and even problematic, when the depth of the *directory tree* is not hard-coded into the script (or cannot be, because it is not known). The solution to processing trees of arbitrary depth is to use *recursion*, which in Praat can be accomplished using a procedure that *calls itself*:

Listing 3.13: Process each subdirectory recursively

```
# "initialize" array of directory names
num_Dirs = 0

# root of directory tree
basepath$ = "foo"

# preparations
depth = 0
call openDir 'basepath$'

procedure openDir .dir$
  # .listName$ is the name of each Strings, purely cosmetic
  .listName$ = "dirList"
  .dir_'depth'$ = .dir$
  # operation to be performed on every directory in the
  # tree goes here, e.g.
  call listDir '.dir$'
  # or just append to an array of directory names for
  # later processing:
  num_Dirs += 1
  directory_'num_Dirs'$ = .dir$
  # create Strings of subdirectories
  Create Strings as directory list... '.listName$' '.dir$'
  .numDirs_'depth' = Get number of strings
  # for loop is skipped if no subdirectories in this .dir$
  for .dir_'depth' to .numDirs_'depth'
    .nextDir$ = Get string... .dir_'depth'
    # under Windows, exclude "." and ".." entries
    if .nextDir$ <> "." and .nextDir$ <> ".."
      depth += 1
      # recursive procedure call
      call openDir '.dir$'/'.nextDir$'
      depth -= 1
    endif
    # reset .dir$, because recursive call has overwritten it
    .dir$ = .dir_'depth'$
    select Strings '.listName$'
  endfor
  Remove
endproc
```

```
procedure listDir .dir$
  for d to depth
    print
  endfor
  print '.dir$':'newline$'
  Create Strings as file list... fileList '.dir$'
  .numFiles = Get number of strings
  for .file to .numFiles
    .file$ = Get string... .file
    for d to depth
      print
    endfor
    print   '.file$''newline$'
    select Strings fileList
  endfor
  Remove
endproc
```

Since subdirectories with no subdirectories of their own are removed at the end pf the `openDir` procedure, the last `Strings` object in the object list is always the "current" directory being processed (i.e. the current *node* in the tree).[9] Note that in spite of the use of local variables, certain variables required for *backtracking* would be overwritten, which necessitates use of an array indexed by the `depth` counter (which represents the number of nodes in the tree dominating the current node).

[9]This exploits Praat's behavior of selecting the *last* candidate object in cases of ambiguity, such as when selecting by object name, as done here.

Chapter 4

Sound Editing

Several possibilities exist in Praat to create, filter, and otherwise manipulate Sounds, with special emphasis on speech. A few of the more common techniques will be discussed in this Section. While some of them can be used by hand (with Editors and Praat commands), others require certain amounts of scripting to achieve.

4.1 Editing with the Sound Editor

If you are familiar with "normal" sound editing software (such as Adobe Audition (formerly CoolEdit), Sound Forge, Audacity[1], etc.), you will not be surprised to find a few common commands in Praat's Sound Editor.

4.1.1 Sound clipboard

In the Sound Editor (*not* the TextGrid Editor[2]), we can perform a number of very basic manipulations on the Sound itself. Most of these involve the *Sound clipboard*, which stores a portion of a Sound and whose contents can be easily inserted into a Sound. To place a portion of a Sound in the Sound clipboard, select the appropriate part of the Sound in the Sound Editor and use the command `Copy selection to Sound clipboard` from the *Edit* menu. You can also simultaneously remove this selection from the Sound by using the `Cut` command instead.

To insert the contents of the Sound clipboard into the Sound displayed in the Sound Editor (which need not be the Sound it was originally taken from), use the `Paste after selection` command. If there currently is no selection, it will be pasted at the cursor position. This will not empty the Sound clipboard, so the insertion can be performed repeatedly.

Note that the Sound clipboard can only hold one contiguous Sound extraction. Any copying to the Sound clipboard will overwrite its previous contents (if

[1] audacity.sourceforge.net

[2] If we want to select intervals by clicking on them like in the TextGrid Editor, we can open a TextGrid Editor on the TextGrid *in addition to* the Sound in the Sound Editor. Then, if we make sure that the *Group* boxes in the lower right-hand corner of both editors are checked, we have synchronized scrolling, zoom level and selection between the two editors. By selecting an interval in the TextGrid Editor, we simultaneously select the corresponding range in the Sound Editor, which we can then copy to the Sound clipboard, etc.

any). Also note that the sampling frequency of the Sound clipboard's contents must match that of the Sound to be pasted into.

4.1.2 Other editing commands

The Sound Editor contains only two other commands for modifying the displayed Sound, and both are applicable only to a selection in the Sound:

- `Set selection to zero` will silence the current Sound selection;
- `Reverse selection` will modify the selection so that it will be played backwards.

Additionally, the `Undo` command can be used to restore the Sound to the state before the last editing command.

By the way, instead of copying a selection to the Sound clipboard which can hold only a single selection at once, we can alternatively extract the selection to the Object Window as a new Sound with the appropriate `Extract selection` command in the *File* menu of the Editors,[3] which brings us to the next Section.

4.2 Editing with the Object Window

Of course, for scripting purposes, it is preferable to perform such operations in the Object Window directly, which can be done with commands such as `Extract part...` (found in the *Convert* submenu) and `Concatenate` (in the *Combine sounds* submenu). A part of a Sound can also be silenced with the `Set part to zero...` command, and an entire Sound (not just a part, unfortunately) can be reversed with the `Reverse` command (both are found in the *Modify* submenu).

4.2.1 Extracting parts of Sounds

To extract part of a Sound as a new Sound, we use the `Extract part...` command mentioned above. It is essentially identical to the `Extract windowed selection...` command in the Sound Editor in that it allows us to specify how the extracted part should be windowed ("faded" in and out) at the edges. The point is to avoid "jumps" in the signal when two extractions are concatenated and do not join well. If they both join at zero, artifacts will be minimized. To disable such windowing, use `Rectangular` as the window type parameter.

A useful feature of Praat is the possibility of extracting intervals from a specified interval tier in a TextGrid as Sounds, provided both the TextGrid and corresponding Sound are selected in the object list. Each extraction begins and ends at the corresponding interval's boundaries. For additional transparency, these new Sounds will be named according to the interval's label (subject to the usual Object naming restrictions). The command to do this for all intervals indiscriminately is `Extract all intervals...`, while empty (i.e. unlabeled) intervals can be ignored using `Extract non-empty intervals...`. Precise control over which intervals should be extracted can be exerted with the `Extract intervals where...` command, which allows even *regular expressions*.

[3] This works in the TextGrid Editor, too!

4.2.2 Concatenating Sounds

If more than one Sound is selected in the object list, we can use the `Concatenate` command to combine them into one long Sound. The selection may of course be discontinuous (i.e. the Sounds do not have to be adjacent in the list), but they will be concatenated *in the exact order* in which they appear in the list. This means that to reorder Sounds differently than in the object list, some extensive use of the `Copy...` command may be in order. It is therefore recommended to bear this behavior in mind when extracting or creating Sounds, so that they are placed in the object list in the order in which they are to be concatenated finally.

An additional feature is Praat's ability to create a TextGrid along with the concatenated Sound, so that the original Sounds start and end times are preserved as interval boundaries. Furthermore, the Sound objects' names are the labels of these intervals. This is done with the `Concatenate recoverably` command.

4.2.3 Examples

Before we look at some example scripts which edit a file called `123.wav`, we will create a TextGrid for this Sound using *automatic segmentation*.

Autosegmenting with Praat

Using the Sound Editor, we will edit a short recording consisting of the words, "one", "two", and "three." To make things more easier, we will first create a TextGrid containing a lexical transcription.

A recent addition to Praat's features (stable since version 4.5.02) is the `To TextGrid (silences)...` command. It attempts to automatically create a TextGrid for a Sound, with boundaries at silent/non-silent transitions, based on the Sound's intensity contour.

Figure 4.1: Sound `123`'s waveform

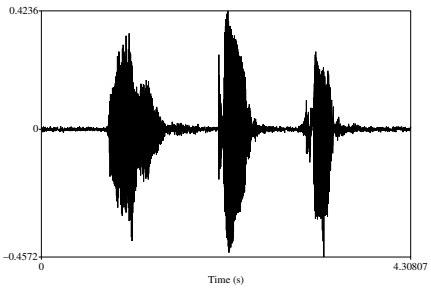
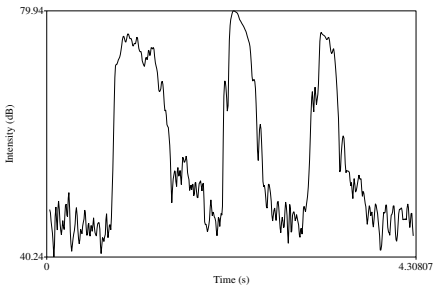


Figure 4.2: Sound `123`'s intensity contour



Using the `To TextGrid (silences)...` command with default values (except for the labels), and subsequent labeling of the words by hand produces a well-segmented TextGrid (Figure 4.4).

Figure 4.3: `To TextGrid (Silences)...` dialog

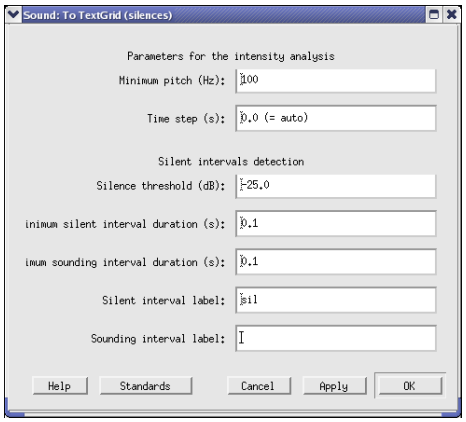
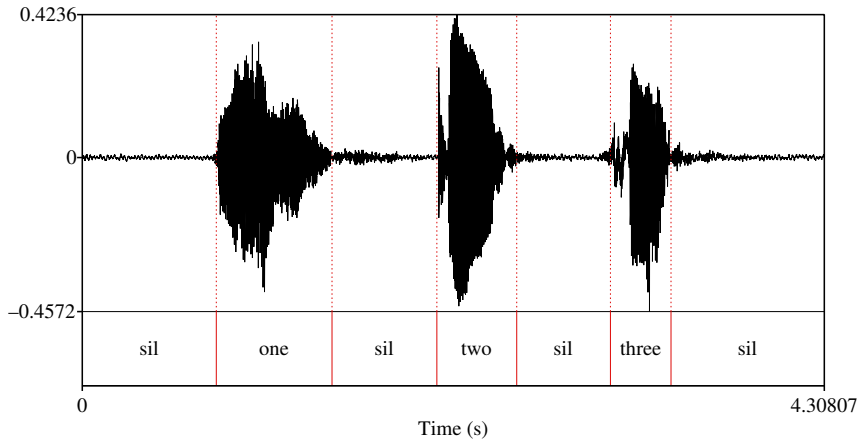


Figure 4.4: Sound and TextGrid `123`



Editing with a Sound Editor script

For this example, we want to load and edit `Sound 123` in such a way that all intervals labeled `sil` are silenced (eliminating recording noise), and the non-empty intervals are reordered, so that the utterance "three, two, one" is synthesized.

The first part, silencing the `sil`-labeled intervals, is given as Listing 4.1:

Listing 4.1: Editor script example

```
# store selection
soundID = selected("Sound")
soundName$ = selected$("Sound")
tgID = selected("TextGrid")
tgName$ = selected$("TextGrid")
```

```
# open editor windows
select soundID
Edit
select tgID
Edit

# get end time of TextGrid
xmax = Get end time
editor TextGrid 'tgName$'
Move cursor to... xmax

# zero "sil" intervals
repeat
  Select next interval
  end = Get end point of interval
  label$ = Get label of interval
  if label$ == "sil"
    endeditor
    editor Sound 'soundName$'
    Set selection to zero
    endeditor
    editor TextGrid 'tgName$'
  endif
until end == xmax

# reorder non-"sil" intervals
# HERE BE DRAGONS!

# cleanup
Close
endeditor
editor Sound 'soundName$'
Close
endeditor
plus soundID
```

For the second task, reordering the non-`sil` intervals, the most intuitive approach, using the Sound and TextGrid Editors, is, ironically, also the most harrowingly complex procedure, if only commands available from the Editor windows are to be used... Feel free to try it out by hand, but stay away from trying to write an editor script for this operation.[4]

Editing with the Object Window (Part 1)

For scripting purposes, it is much easier to use the Object Window. The first script we will study uses extraction commands, reorders by copying and finally concatenates.

Listing 4.2: Object Window script example 1

```
# store selection
soundID = selected("Sound")
tgID = selected("TextGrid")

# extract intervals as sounds and store IDs in two arrays
Extract all intervals... 1 0
```

[4]The various problems in creating such a script involve writing code to remove text from an interval, shifting all boundaries after it to the left, then shifting boundaries after the target position to the right (keeping the labels in the appropriate intervals!), and finally inserting the label text again. Repeat twice for every two intervals you want to switch...

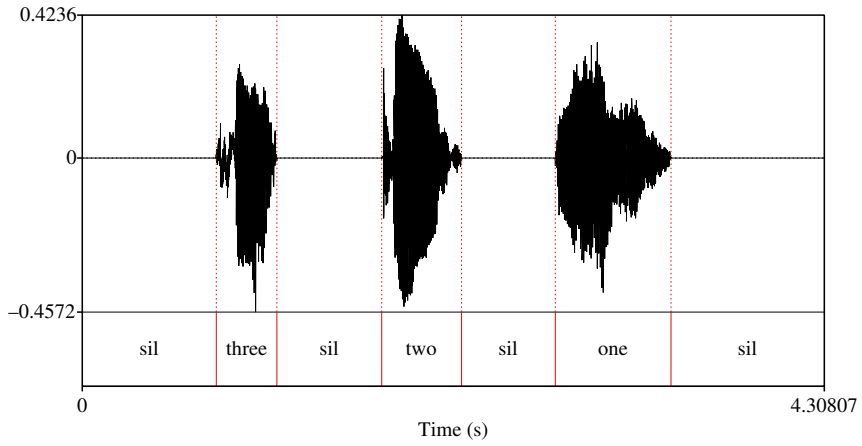
```
num_sil = 0
num_non_sil = 0
for s to numberOfSelected()
  if selected$("Sound", s) == "sil"
    num_sil += 1
    sil_'num_sil'ID = selected(s)
  else
    num_non_sil += 1
    non_sil_'num_non_sil'ID = selected(s)
  endif
endfor

# copy in final order ("sil"s in order, non-"sil"s in reverse order)
select sil_1ID
Copy... sil
firstSoundID = selected()
s = 2
n = num_non_sil
while s <= num_sil and n > 0
  select non_sil_'n'ID
  name$ = selected$("Sound")
  Copy... 'name$'
  n -= 1
  select sil_'s'ID
  Copy... sil
  s += 1
endwhile
lastSoundID = selected()

# select and concatenate
select firstSoundID
for s from firstSoundID + 1 to lastSoundID
  plus s
endfor
Concatenate recoverably
finalSoundID = selected("Sound")
finalTgID = selected("TextGrid")

# cleanup
select sil_1ID
for s from 2 to num_sil
  plus sil_'s'ID
endfor
for s to num_non_sil
  plus non_sil_'s'ID
endfor
for s from firstSoundID to lastSoundID
  plus s
endfor
Remove
select finalSoundID
plus finalTgID
```

The result of this script is shown in Figure 4.5.

Editing with the Object Window (Part 2)

Using a slightly different approach, we can accomplish the same goal more elegantly, storing the intervals in a TextGrid (whose rows are then rearranged as desired) and easily iterating over it to extract parts from the Sound in the final order.

Figure 4.5: Sound and TextGrid `123` after zeroing all `sil` intervals and reversing the order of the others (compare to Figure 4.4)



Listing 4.3: Object Window script example 2

```
# store selection
soundID = selected("Sound")
tgID = selected("TextGrid")

# create TableOfReal from TextGrid
select tgID
Extract tier... 1
itID = selected()
Down to TableOfReal (any)
torID = selected()

# extract non-"sil" intervals
Extract rows where label... "is not equal to" sil
tor_non_silID = selected()

# copy non-"sil" rows back into full TableOfReal, in reverse
select torID
for row to Object_'tor_non_silID'.nrow
  new_row = Object_'torID'.nrow - row * 2 + 1
  label$ = Object_'tor_non_silID'.row$[row]
  Set row label (index)... new_row 'label$'
  for col to Object_'torID'.ncol
    value = Object_'tor_non_silID'[row, col]
    Set value... new_row col value
  endfor
endfor

# extract parts from Sound
for r to Object_'torID'.nrow
  select soundID
  start = Object_'torID'[r, "Start"]
  end = Object_'torID'[r, "End"]
  Extract part... start end Rectangular 1 0
  sound_'r'ID = selected()
```

```
endfor

# select and concatenate
for s to Object_'torID'.nrow - 1
  plus sound_'s'ID
endfor
Concatenate recoverably
finalSoundID = selected("Sound")
finalTgID = selected("TextGrid")

# restore labels from TextGrid
select finalTgID
for i to Object_'torID'.nrow
  label$ = Object_'torID'.row$[i]
  Set interval text... 1 i 'label$'
endfor

# cleanup
select itID
plus torID
plus tor_non_silID
for s to Object_'torID'.nrow
  plus sound_'s'ID
endfor
Remove
select finalSoundID
plus finalTgID
```

Note that this script does not suffer from a certain shortcoming of Listing 4.2, namely that all characters in the original TextGrid's labels which not allowed for object names (i.e. everything except letters, digits, and underscores) will be replaced by underscores, since the final TextGrid's labels are derived from the concatenated Sounds' object names. In Listing 4.3, the labels are stored as row labels in the TextGrid and reinserted after concatenation.

4.3 Duration manipulation

The easiest way to change the duration of a Sound is to have it play more quickly or more slowly. This is easily accomplished by telling Praat to play the Sound in such a way that more or less samples of the Sound are played per second; for this, we can use the `Override sampling frequency...` command. E.g. to play a Sound at twice its normal speed (halving its duration), we simply pass twice that Sound's sampling frequency as the parameter of the `Override sampling frequency...` command.

An obvious side-effect of this technique is that the pitch is modified in correspondence to the shift in duration, i.e. a Sound with a pitch of 500 Hz sampled at 16 kHz will have its pitch lowered to 250 Hz if it is slowed down by halving its sampling frequency by 800 Hz.

4.3.1 PSOLA

Praat's specialty is speech, and so there are relatively sophisticated features available for manipulating speech Sounds. This includes modifying the duration of Sounds without changing the pitch (and vice versa) using an algorithm known

as *PSOLA* (Pitch Synchronous OverLap Add). The details of this algorithm will not be discussed here, but it works rather well within certain bounds.

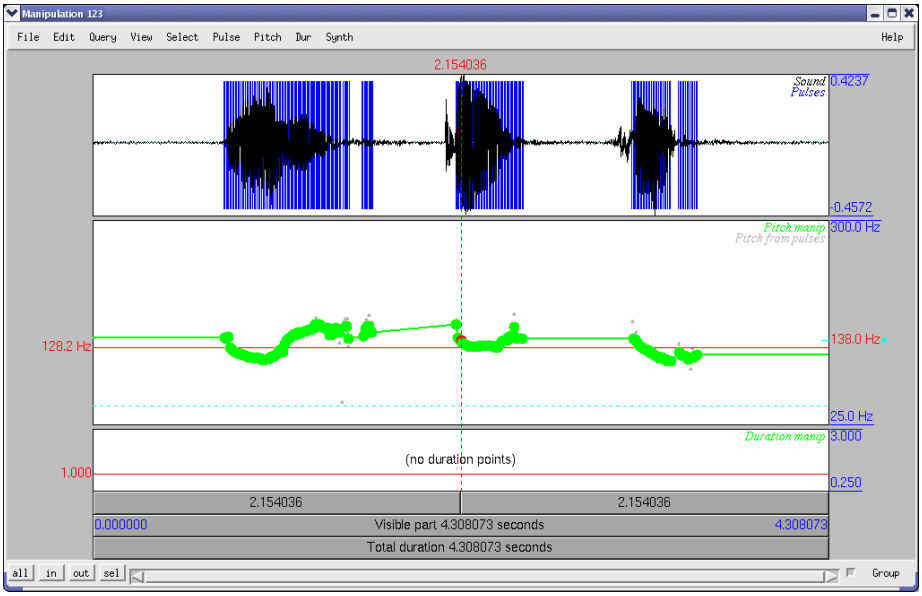
The easiest way to use PSOLA for duration manipulation is the `Lengthen (PSOLA)...` command. The *Factor* parameter determines the resulting duration, relative to the original.

Note that if an *annotated* Sound is lengthened (or shortened) in this way, the corresponding TextGrid will no longer match. Fortunately, we can apply uniform scaling so that the TextGrid matches the modified Sound. We simply select both the Sound and the TextGrid and apply the command `Scale times`.

4.3.2 The Manipulation object

To create a Manipulation object, use the `To Manipulation...` command on a Sound object. The resulting `Manipulation` object can be modified with the powerful Manipulation Editor (Figure 4.6). The top panel of this editor displays the waveform (with pulses) and is similar to the Sound Editor. The middle panel displays the *pitch tier* and the bottom panel, the *duration tier*. In this section, we will focus on the duration tier.

Figure 4.6: Manipulation Editor window



In a newly created Manipulation object, the duration tier will be empty. Adding points will define a duration *contour*, and this in turn will modify the Sound's *local* duration. For example, adding one point with a value of 1 at the beginning of a Sound, another one with a value of 1.5 in the middle, and a third point with a value of 1 at the end will result in a modified Sound that becomes increasingly longer (i.e. slower), up a maximal slowdown of 50% in the middle, and then speeds up again to normal speed at the end. Halfway between the beginning and the middle of the Sound, the duration contour will have a value

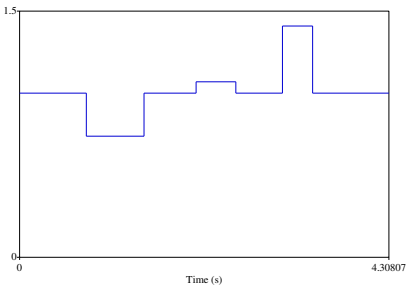
of 1.25 (even though there is no duration point there), which means that this part of the Sound will be 25% slower than normal.

4.3.3 Selective interval equalization

To have only *portions* of the Sound at different relative durations, and those evenly lengthened/shortened, we need a duration "contour" that is essentially a series of rectangles. The problem is that it is not possible to have two duration points above one another at the same time (as would be required for a perfect rectangle). We can approximate this, however, by creating two duration points that are only e.g. 0.0000000001[5] seconds apart, which is just as good for practical purposes.[6]

For example, to modify a Sound so that all non-`sil` intervals are equally long, we could use the following script, which generates the duration tier shown in Figure 4.7 and inserts it into the Manipulation. It then resynthesizes the Sound, which results in Figure 4.9.

Figure 4.7: Duration tier used to make `123`'s non-`sil` intervals equally long



Listing 4.4: Make non-`sil` intervals equally long (mean)

```
# store selection
soundID = selected("Sound")
tgID = selected("TextGrid")

# create TableOfReal from TextGrid
select tgID
Extract tier... 1
itID = selected()
Down to TableOfReal (any)
torID = selected()
Extract rows where label... "is not equal to" sil
tor_non_silID = selected()

# get mean non-"sil" interval duration
mean = Get column mean (label)... Duration

# create Manipulation from Sound
select soundID
```

[5] Remember: this can be represented as 1×10^{-10} and written in Praat as `1e-10`.

[6] Keep in mind that at a sampling frequency of 44.1 kHz, two adjacent samples are $\frac{1}{44100} \approx 0.0000226757$ seconds apart, which is *much* longer than the "slope" of such a near-rectangle!

```
To Manipulation... 0.01 75 600
manID = selected()

include equalizeDurationsEditor.praat
call equalizeDurationsEditor tor_non_silID mean

# make new TextGrid
newSoundID = selected()
To TextGrid... labels
newTgID = selected()
end = Get start time
for i to Object_'torID'.nrow
  label$ = Object_'torID'.row$[i]
  Set interval text... 1 i 'label$'
  if i < Object_'torID'.nrow
    if Object_'torID'.row$[i] == "sil"
      end += Object_'torID'[i, "Duration"]
    else
      end += mean
    endif
    Insert boundary... 1 end
  endif
endfor

# cleanup
select itID
plus torID
plus tor_non_silID
plus manID
Remove
select newSoundID
plus newTgID
```

Listing 4.5: Procedure for actual duration manipulation using the Manipulation Editor

```
procedure equalizeDurationsEditor .tableOfRealID .targetDuration
  .manName$ = selected$("Manipulation")
  Edit
  editor Manipulation '.manName$'
    for .i to Object_'.tableOfRealID'.nrow
      .duration = .targetDuration / Object_'.tableOfRealID'[.i, "Duration"]
      .start = Object_'.tableOfRealID'[.i, "Start"]
      Add duration point at... .start 1
      .start += 1e-10
      Add duration point at... .start .duration
      .end = Object_'.tableOfRealID'[.i, "End"]
      Add duration point at... .end 1
      .end -= 1e-10
      Add duration point at... .end .duration
    endfor
    Publish resynthesis
    Close
  endeditor
endproc
```

Figure 4.8: Sound and TextGrid `123` *before* running Listing 4.4

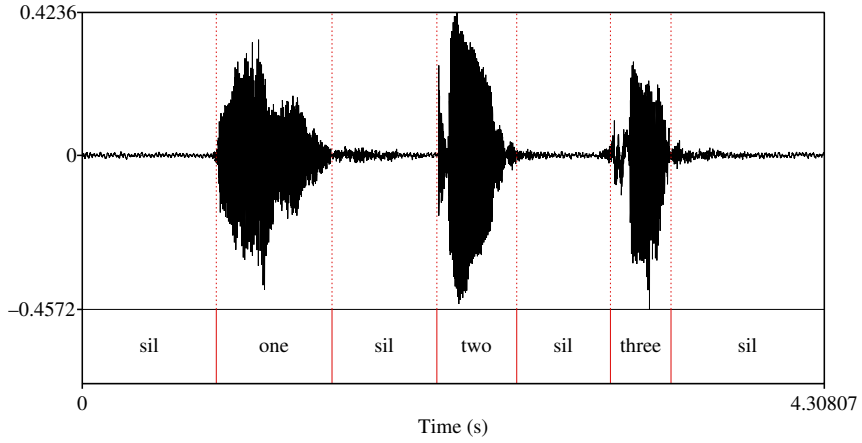
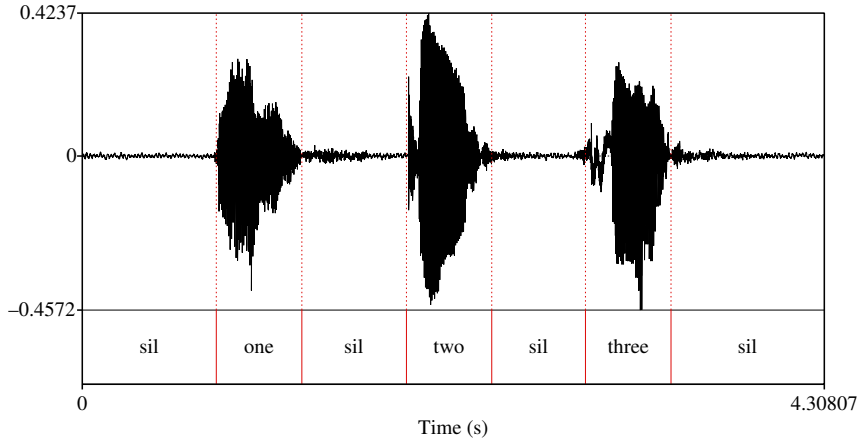


Figure 4.9: Sound and TextGrid `123` *after* making the non-empty interval equally long



The actual manipulation (i.e. performing operations on the Manipulation *object*) is accomplished through the procedure in Listing 4.5. As usual, this can also be done using only commands from the Object Window:

Listing 4.6: Procedure for actual duration manipulation *without* using the Manipulation Editor

```
procedure equalizeDurations .tableOfRealID .targetDuration
  Extract duration tier
  dtID = selected()
  for .i to Object_'.tableOfRealID'.nrow
    .duration = .targetDuration / Object_'.tableOfRealID'[.i, "Duration"]
    .start = Object_'.tableOfRealID'[.i, "Start"]
    Add point... .start 1
    .start += 1e-10
    Add point... .start .duration
    .end = Object_'.tableOfRealID'[.i, "End"]
    Add point... .end 1
    .end -= 1e-10
    Add point... .end .duration
  endfor
  plus manID
  Replace duration tier
  minus dtID
  Get resynthesis (PSOLA)
endproc
```

Note that the TextGrid is rebuilt because there is no straightforward way of time-shifting only some intervals of a TextGrid, keeping the other interval durations unchanged.

4.3.4 Selective interval equalization *without* Manipulation object

A possible alternative approach that does not make use of the Manipulation Editor is to use `Extract all intervals...`, then `Lengthen (PSOLA)...` (with varying factors) on only *some* of the extracted Sounds, and finally `Concatenate recoverably`:[7]

Listing 4.7: Make non-`sil` intervals equally long (mean) *without* a Manipulation

```
# store selection
soundID = selected("Sound")
tgID = selected("TextGrid")

# create TableOfReal from TextGrid
select tgID
Extract tier... 1
itID = selected()
Down to TableOfReal (any)
torID = selected()
Extract rows where label... "is not equal to" sil
tor_non_silID = selected()

# get mean non-"sil" interval duration
mean = Get column mean (label)... Duration
```

[7] Because the pitch extraction precalculations required for PSOLA are only performed on those parts of the original Sound that are actually lengthened, as opposed to the entire Sound, which is done when a Manipulation object is created, this approach is actually slightly *faster* than those that make use of the Manipulation.

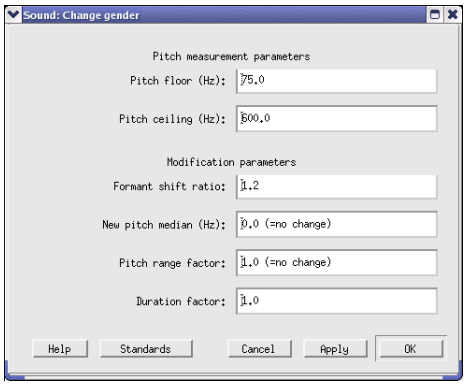
```
# extract intervals
select soundID
plus tgID
Extract all intervals... 1 0
for s to numberOfSelected("Sound")
  extractedSound_'s'ID = selected(s)
endfor

# copy or lengthen depending on label/name
for s to Object_'torID'.nrow
  select extractedSound_'s'ID
  soundName$ = selected$("Sound")
  if soundName$ == "sil"
    Copy... 'soundName$'
    newSound_'s'ID = selected()
  else
    factor = mean / Sound_'soundName$'.xmax
    noprogress Lengthen (PSOLA)... 75 600 factor
    Rename... 'soundName$'
    newSound_'s'ID = selected()
  endif
endfor

# concatenate
select newSound_1ID
for s from 2 to Object_'torID'.nrow
  plus newSound_'s'ID
endfor
Concatenate recoverably
finalSoundID = selected("Sound")
finalTgID = selected("TextGrid")

# cleanup
select itID
plus torID
plus tor_non_silID
for s to Object_'torID'.nrow
  plus extractedSound_'s'ID
  plus newSound_'s'ID
endfor
pause
Remove
select finalSoundID
plus finalTgID
```

4.4 Pitch manipulation

The PSOLA algorithm can of course also be used to change the pitch of a Sound while keeping the duration constant. This can easily be achieved in Praat with the command `Change gender...` in the *Convert* submenu. The command's dialog is shown in Figure 4.10.

Corresponding to the command's modification parameters, the formants can be shifted, the median pitch changed, the pitch range expanded or contracted and, as a "bonus", the duration modified (as with the `Lengthen (PSOLA)...` command; all of these parameters can be changed independently.

Figure 4.10: `Change gender...` dialog



4.4.1 Pitch manipulation with the Manipulation object

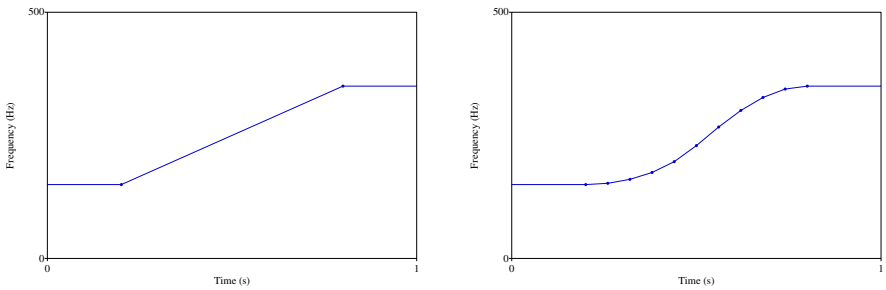
Of course, as foreshadowed in Section 4.3.2, precise control over the pitch manipulation can be achieved using a Manipulation object. For this, the procedure is essentially the same as with manipulation of duration, including the choice between using the Manipulation Editor or the Object window's commands. In the latter case, we either extract the *pitch* tier from the Manipulation object or create a new one from scratch. After modifying the resulting PitchTier object as desired (again, using either the PitchTier Editor or commands available from the Object Window), we select it together with the Manipulation object and run the `Replace pitch tier` command, before using the `Get resynthesis (PSOLA)` or `Get resynthesis (LPC)` command to render the manipulated Sound.

The resulting manipulated Sound will have a pitch contour almost exactly along the points defined in the pitch tier of the Manipulation, but only in areas where the signal *has* pitch (i.e. a periodic component in the appropriate frequency range) in the first place. Pitch-less portions such as voiceless fricatives, noise and silence remain largely unaffected by Praat's pitch manipulation.

A key difference to modifying the duration tier of a Manipulation lies in the fact that the pitch tier is initially already filled with the Sound's pitch contour, manifested as a number of points and of course influenced by the parameters used for the pitch extraction. In fact, if the parameters supplied to the `To Manipulation...` command are insufficient, we can use another command (such as `To Pitch (ac)...`) to extract the pitch to a *Pitch* object, modify that if desired, and then insert the resulting pitch contour into a Manipulation by converting it with the `Down to PitchTier` command and using the `Replace pitch tier` command as above.

Additionally, Praat offers the possibility of using *quadratic splines* to interpolate between two points in a PitchTier in such a way that a number of evenly spaced new points are inserted not along a line connecting the two initial points, but along a curve sloping smoothly from one point to the other. The command for this is `Interpolate quadratically...`, and its effect can be seen in Figure 4.11.

Figure 4.11: Effect of quadratic interpolation on a pitch contour



4.5 Formant manipulation

Using Praat's *source-filter synthesis* features, it is possible to modify the formants of a Sound. This involves three steps:

1. Create a *source* Sound from the Sound
2. Create a *filter* FormantTier
3. Filter the source

The first step is fairly straightforward. We first create an *LPC* object from the resampled original Sound[8] using the command `To LPC (burg)...` or an equivalent alternative from the *Formants & LPC* submenu. We then select the original Sound along with this LPC and apply the command `Filter (inverse)` which creates the source Sound.

The next step is to create a filter in the form of a *FormantTier* object. To create a blank FormantTier, we use the `Create FormantTier...` command. To manipulate the original Sound's formants, however, we create a Formant object from the Sound, using the `To Formant (burg)...` command (or equivalent) and convert that to a FormantTier object with the command `Down to FormantTier`. This will serve as the filter.

Once the FormantTier has been modified as desired (see below), we simply select it along with the source Sound and apply the `Filter` command. However, we should bear in mind that this kind of LPC resynthesis significantly lowers the fidelity of the resulting signal, since the not all linear prediction coefficients have survived in the Formant object.

4.5.1 Selective formant manipulation

The actual manipulation consists of modifying the FormantTier before filtering the source Sound. Unfortunately, Praat has no FormantTier Editor, and each *point* in the FormantTier consists of several frequency-bandwidth pairs which cannot be individually manipulated. The solution is to store *all* points in the portion of the Sound to be modified, then remove those points and finally reinsert them again, but with certain changes, as desired.

[8] Resampling to twice the frequency of the highest forment increases the accuracy of the LPC analysis

The following script illustrates this process by switching all formants in all non-empty intervals of a certain tier one non-empty interval to the right. This changes 123 (Figure 4.12) to Figure 4.13.

Listing 4.8: Switch all formants in non-empty intervals

```
form Switch formants on tier
  natural Tier 2
  natural LPC_components 10
  integer Resample 10000 (=0 to disable)
  natural Formants 5
  boolean Cleanup 1
endform

# store selection
soundID = selected("Sound")
tgID = selected("TextGrid")

# store vowel boundaries on selected tier in arrays
minus soundID
numInts = Get number of intervals... tier
numV = 0
for i to numInts
  label$ = Get label of interval... tier i
  if label$ <> ""
    numV += 1
    vowel_'numV'_start = Get starting point... tier i
    vowel_'numV'_end = Get end point... tier i
  endif
endfor

# resample if desired
select soundID
if resample
  noprogress Resample... resample 50
  resampledID = selected()
endif

# extract source
noprogress To LPC (burg)... lPC_components 0.025 0.005 50
lpcID = selected()
if resample
  plus resampledID
else
  plus soundID
endif
Filter (inverse)
sourceID = selected()

# create filter and store all FormantTier points in array
select soundID
noprogress To Formant (burg)... 0 formants 5500 0.025 50
formID = selected()
Down to FormantTier
ftID = selected()
numP = Get number of points
for p to numP
  time = Get time from index... p
  for f to formants
    point_'p'_F'f' = Get value at time... f time
    point_'p'_B'f' = Get bandwidth at time... f time
  endfor
  point_'p'_time = time
```

```
endfor

# for each vowel, find relevant FormantTier points
for v to numV
  start = vowel_'v'_start
  end = vowel_'v'_end
  vowel_'v'_firstPoint = Get high index from time... start
  vowel_'v'_lastPoint = Get low index from time... end
endfor
for v to numV
  if v < numV
    otherV = v + 1
  else
    otherV = 1
  endif
  start = vowel_'v'_start
  end = vowel_'v'_end

  # switch points with next/other vowel
  firstPoint = vowel_'v'_firstPoint
  firstPoint_time = point_'firstPoint'_time
  lastPoint = vowel_'v'_lastPoint
  lastPoint_time = point_'lastPoint'_time
  timestep = (lastPoint_time - firstPoint_time)
  ... / (vowel_'otherV'_lastPoint - vowel_'otherV'_firstPoint)
  Remove points between... start end
  time = firstPoint_time
  for p from vowel_'otherV'_firstPoint to vowel_'otherV'_lastPoint
    formants$ = ""
    for f to formants
      freq = point_'p'_F'f'
      bndw = point_'p'_B'f'
      formants$ = "'formants$' 'freq' 'bndw'"
    endfor
    Add point... time'formants$'
    time += timestep
  endfor
endfor

# filter
plus sourceID
Filter
finalSoundID = selected()

# cleanup
if cleanup
  plus ftID
  plus formID
  plus sourceID
  plus lpcID
  if resample
    plus resampledID
  endif
  Remove
endif
select soundID
plus tgID
```

Figure 4.12: Sound and TextGrid `123` with two vowels marked

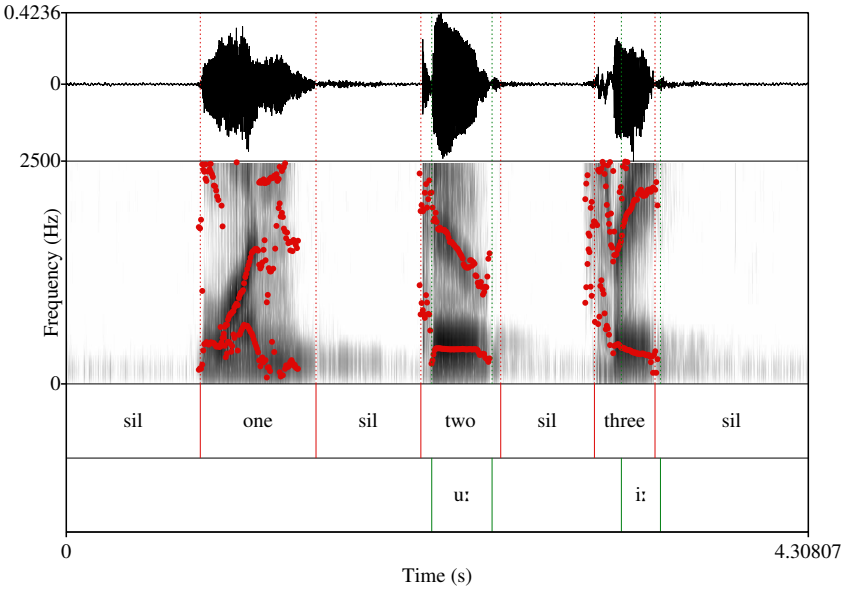
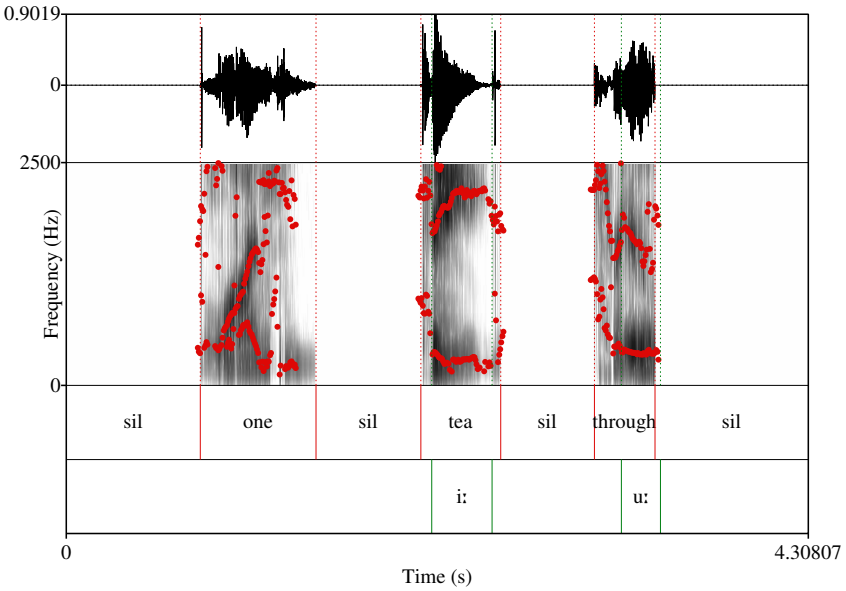


Figure 4.13: Sound and TextGrid `123` with formants switched (and `sil` intervals silenced)



4.6 Low-level sound manipulation

We can "bypass" Praat's commands and access the samples stored in a Sound object directly, even changing individual samples with the command `Set sample number...`. This *low-level* access is the only way to accomplish tasks for which no Praat command exists; examples follow in this Section. First, however, we will have a look at how to access a Sound directly, without using commands from the *Query* submenu, and then learn about using *Formulas*.

4.6.1 Direct Sound access

Similar to what we saw in Section 3.1, we can access a Sound in the object list directly, even when it is not selected, and use its attributes and data in scripts and numeric Praat command arguments. The syntax, just like with `TableOfReal` objects, is `Sound_foo`, where `foo` is the name of the Sound, or (more robustly for lack of potential ambiguity) `Object_n`, where `n` is the ID of the Sound in question. Table 4.1 gives a non-exhaustive overview of common Praat commands and their direct access equivalents.[9] Keep in mind that direct object access is *read-only*; to modify a Sound, the appropriate Praat commands must be used.

Table 4.1: Standard Praat commands vs. direct object access (`Sound`)

`Get start time`	`Object_'id'.xmin`
`Get end time`	`Object_'id'.xmax`
`Get total duration`	`Object_'id'.xmax - Object_'id'.xmin`
`Get number of samples`	`Object_'id'.nx` *or* `Object_'id'.ncol`
`Get sampling period`	`Object_'id'.dx`
`Get sampling frequency`	`1 / Object_'id'.dx`
`Get value at time... t Linear`	`Object_'id'(t)`
`Get value at sample number... s`	`Object_'id'[s]`

Note that when attempting to access values that are out of bounds (e.g. a value at a time after the end of the Sound), Praat will return `0`.

4.6.2 Formulas

Many Praat commands take a *formula* as one of their arguments. Such a formula is essentially a small, single-line script with restricted syntax. In fact, it is limited to expressions that return a numeric value. A number of special variables can be used in the formula, but no new variables may be declared, although when using a formula within a script, the script's variables may of course be used within the formula. Also, to allow the use of conditions, an abbreviated syntax can fit a simple condition into the formula:

`if` *`condition`* `then` *`value1`* `else` *`value2`* `fi`

where `condition`, `value1`, and `value2` are expressions returning a numeric value. Note that there is no provision for an `elif` block, and that the `else` block is mandatory.

[9] For the standard commands to work, the `Sound` with ID `id` must be selected. This is not required for direct object access.

The formula is applied to each sample in the signal, and the value it returns becomes the new value for the sample. The expressions used can be constants (i.e. numbers) or functions returning a constant. When applying a function to an object, the following formula-internal variables may be used to access available objects:

Table 4.2: Predefined variables in a Sound formula

`self`	value of current sample
`col`	number of current sample
`x`	time of current sample
`xmin`	time of first sample
`xmax`	time of last sample
`nx` *or* `ncol`	number of samples
`dx`	sampling period

4.6.3 Examples

Applying what we just learned, there are many possible uses for formulas and direct Sound access. Several examples follow.

Increasing/reducing amplitude

This is just for warming up. We can multiply every sample in a Sound by a constant factor, thereby increasing or reducing its amplitude (depending on whether the factor is larger or smaller than 1).

Listing 4.9: Multiply a Sound's sample values

```
form Multiply
  real Multiplication_factor 1.5
endform

Formula... self * multiplication_factor
```

This script merely imitates Praat's `Multiply...` command.

Adding echo

To add an echoing effect to a Sound, we simply cycle through each sample and add to it the value of the sample at a previous time, determined by a constant delay. Since the Sound is modified in place, the `form` describes the script as "inline".

Listing 4.10: Add echo to a Sound

```
form Add echo (inline)
  real Delay 0.25
  real Amplitude 0.5
endform

Formula... self + amplitude * self(x - delay)
```

Mixing two Sounds together

We can mix two Sounds by adding their samples together. Assuming both Sounds start at zero and have the same duration and sampling frequency (i.e. the same number of samples), we could insert one Sound's samples directly into the other Sound. By multiplying each Sound's samples by a certain factor, we can control the ratio and amplitude of the resulting Sound.

Listing 4.11: Mix two Sounds into one

```
form Mix two Sounds
  real Factor_Sound_1 0.5
  real Factor_Sound_2 0.5
endform

sound1 = selected("Sound", 1)
minus sound1

Formula... factor_Sound_1 * Sound_'sound1'[col] + factor_Sound_2 * self
```

We could also create a new Sound containing the mix, and make the procedure more robust by accounting for different source sampling rates, as well as different time domains, choosing appropriate values so that nothing is lost.

Listing 4.12: Create a new Sound as mix of two Sounds

```
form Mix two Sounds
  real Factor_Sound_1 0.5
  real Factor_Sound_2 0.5
endform

sound1 = selected("Sound", 1)
sound2 = selected("Sound", 2)

Create Sound... mix
... "if Object_'sound1'.xmin < Object_'sound2'.xmin
... then
...   Object_'sound1'.xmin
... else
...   Object_'sound2'.xmin
... fi"
...
... "if Object_'sound1'.xmax > Object_'sound2'.xmax
... then
...   Object_'sound1'.xmax
... else
...   Object_'sound2'.xmax
... fi"
...
... "if 1 / Object_'sound1'.dx > 1 / Object_'sound2'.dx
... then
...   1 / Object_'sound1'.dx
... else
...   1 / Object_'sound2'.dx
... fi"
...
... factor_Sound_1 * Object_'sound1'(x)
... + factor_Sound_2 * Object_'sound2'(x)
```

This script demonstrates an additional use of formulas in that it does not first query the original Sounds (start, end, and sampling frequency), store the values in variables and supply these as arguments to the `Create Sound...` command, but

inserts the relevant conditional code as formulas in the appropriate fields of the command directly. Note the use of double quotes to separate the command's arguments.

For legibility, the command's long list of formula-arguments has been liberally split into continuation lines.

Smoothing/noise reduction

We can reduce noise (i.e. randomness in the signal) by *smoothing* a Sound's samples. One method of doing this is by setting each sample to the mean of its own value and that of its two adjacent samples.

Listing 4.13: Smooth a Sound (3 samples, inline)

```
Formula... (self[col - 1] + self + self[col + 1]) / 3
```

If we want to increase the *window length* (the number of samples taken into account), the formula becomes longer.

Listing 4.14: Smooth a Sound (5 samples, inline)

```
Formula... (self[col - 2] + self[col - 1] + self + self[col + 1]
... + self[col + 2]) / 5
```

However, if we want to set a window length of n samples while avoiding modifying the signal before all means have been calculated, we very quickly run into several limitations: Formulas cannot be longer than 98 characters, and there are no loops in formula syntax. For such purposes, we would have to use a "real" script.

Listing 4.15: Smooth a Sound (n samples)

```
form Smooth
  natural Window_length 3
endform

sound = selected()
name$ = selected$("Sound")

Copy... 'name$'_smoothed
Set part to zero... 0 0 at exactly these times

for col to Object_'sound'.ncol
  window_total = 0
  for s from col - window_length to col + window_length
    window_total += Object_'sound'[s]
  endfor
  window_mean = window_total / (2 * window_length + 1)
  Set value at sample number... col window_mean
endfor
```

4.6.4 Creating Sounds from scratch

Apart from ready-made Sounds such as gamma and Shepard tones (`Create Sound from gamma-tone...` and `Create Sound from Shepard tone...`, respectively), we can use the `Create Sound...` command with formulas to produce audible[10] signals from mathematical functions. Several common examples follow, each with the

[10] Depending on the function, of course!

equation of the function, a portion of the signal's oscillogram and a script used to produce it. Of course, the possibilities are theoretically endless, but limited in practice by the maximal length of the string passed as the formula. For formulas longer than 98 characters, we will have to create a blank, "canvas" Sound and modify it using the `Set value at sample number...` command repeatedly in a loop.

Silence

Figure 4.14: Silence

0

Listing 4.16: Generate silence

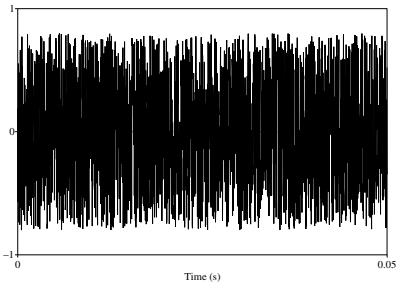
```
form Create silence
  sentence Name silence
  real Start_time_(s) 0.0
  positive End_time_(s) 1.0
  natural Sampling_frequency_(Hz) 44100
endform

Create Sound... 'name$' start_time end_time sampling_frequency
... 0
```

White noise

Figure 4.15: White noise

`randomUniform(`$-A$`, `A`)`

where A is the amplitude of the signal



Listing 4.17: Generate white noise

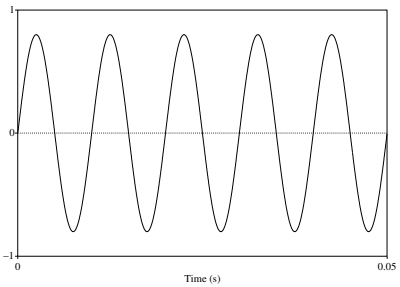
```
form Create white noise
```

```
  sentence Name white_noise
  real Start_time_(s) 0.0
  positive End_time_(s) 1.0
  natural Sampling_frequency_(Hz) 44100
  positive Amplitude_(Pa) 0.8
endform

Create Sound... 'name$' start_time end_time sampling_frequency
... randomUniform(-amplitude, amplitude)
```

Sine

$$A \sin(2\pi f x)$$

where f is the signal frequency



Listing 4.18: Generate a sine wave

```
form Create sine
  sentence Name sine
  real Start_time_(s) 0.0
  positive End_time_(s) 1.0
  natural Sampling_frequency_(Hz) 44100
  natural Frequency_(Hz) 100
  positive Amplitude_(Pa) 0.8
endform

Create Sound... 'name$' start_time end_time sampling_frequency
... amplitude * sin(2 * pi * frequency * x)
```

Square

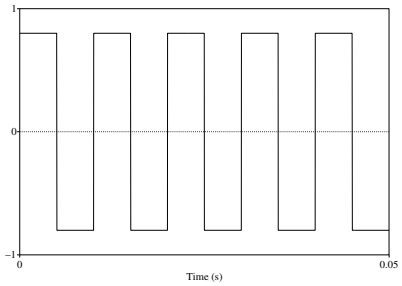
Listing 4.19: Generate a square wave

```
form Create square
  sentence Name square
  real Start_time_(s) 0.0
  positive End_time_(s) 1.0
  natural Sampling_frequency_(Hz) 44100
  natural Frequency_(Hz) 100
  positive Amplitude_(Pa) 0.8
endform

# period
p = 1 / frequency

Create Sound... 'name$' start_time end_time sampling_frequency
... if x mod p <= p / 2
... then
```

Figure 4.16: Square waveform (5 periods)

$$\begin{cases} A & \text{if } 0 \leq x \bmod T \leq \frac{T}{2} \\ -A & \text{if } \frac{T}{2} \leq x \bmod T \leq T \end{cases}$$

where T is the length of one signal period

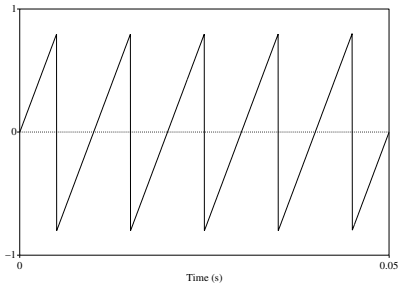



```
...   amplitude
... else
...   -amplitude
... fi
```

Sawtooth

Figure 4.17: Sawtooth waveform (5 periods)

$$\frac{2A}{T}\left(\left(x+\frac{T}{2}\right) \bmod T - \frac{T}{2}\right)$$



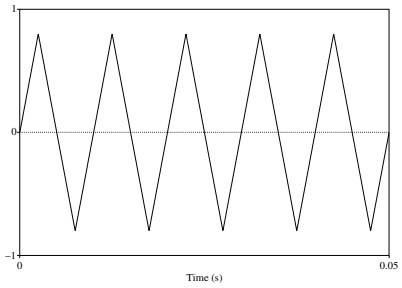
Listing 4.20: Generate a sawtooth wave

```
form Create sawtooth
  sentence Name sawtooth
  real Start_time_(s) 0.0
  positive End_time_(s) 1.0
  natural Sampling_frequency_(Hz) 44100
  natural Frequency_(Hz) 100
  positive Amplitude_(Pa) 0.8
endform

# period
p = 1 / frequency

Create Sound... 'name$' start_time end_time sampling_frequency
... 2 * amplitude / p * ((x + p / 2) mod p - p / 2)
```

Triangle

Figure 4.18: Triangle waveform (5 periods)

$$A\left(\frac{4}{T}\cdot\left|\left(x+\frac{3T}{4}\right) \bmod T-\frac{T}{2}\right|-1\right)$$



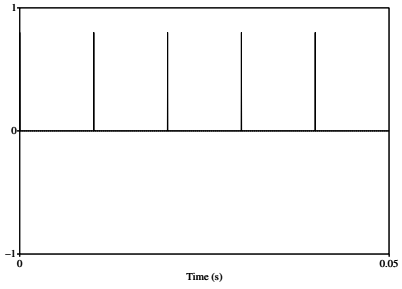
Listing 4.21: Generate a triangle wave

```
form Create triangle
  sentence Name triangle
  real Start_time_(s) 0.0
  positive End_time_(s) 1.0
  natural Sampling_frequency_(Hz) 44100
  natural Frequency_(Hz) 100
  positive Amplitude_(Pa) 0.8
endform

# period
p = 1 / frequency

Create Sound... 'name$' start_time end_time sampling_frequency
... amplitude * (4 / p * abs((x + 3 * p / 4) mod p - p / 2) - 1)
```

Pulse train

Figure 4.19: Pulse train waveform (5 periods)

$$\begin{cases} A & \text{if } x \bmod T = 0 \\ 0 & \text{if } x \bmod T \neq 0 \end{cases}$$



Listing 4.22: Generate a pulse train

```
form Create pulse train
```

```
  sentence Name pulse_train
  real Start_time_(s) 0.0
  positive End_time_(s) 1.0
  natural Sampling_frequency_(Hz) 44100
  natural Frequency_(Hz) 100
  positive Amplitude_(Pa) 0.8
endform

# period
p = 1 / frequency

Create Sound... 'name$' start_time end_time sampling_frequency
... if x mod p < 1 / sampling_frequency
... then
...   amplitude
... else
...   0
... fi
```