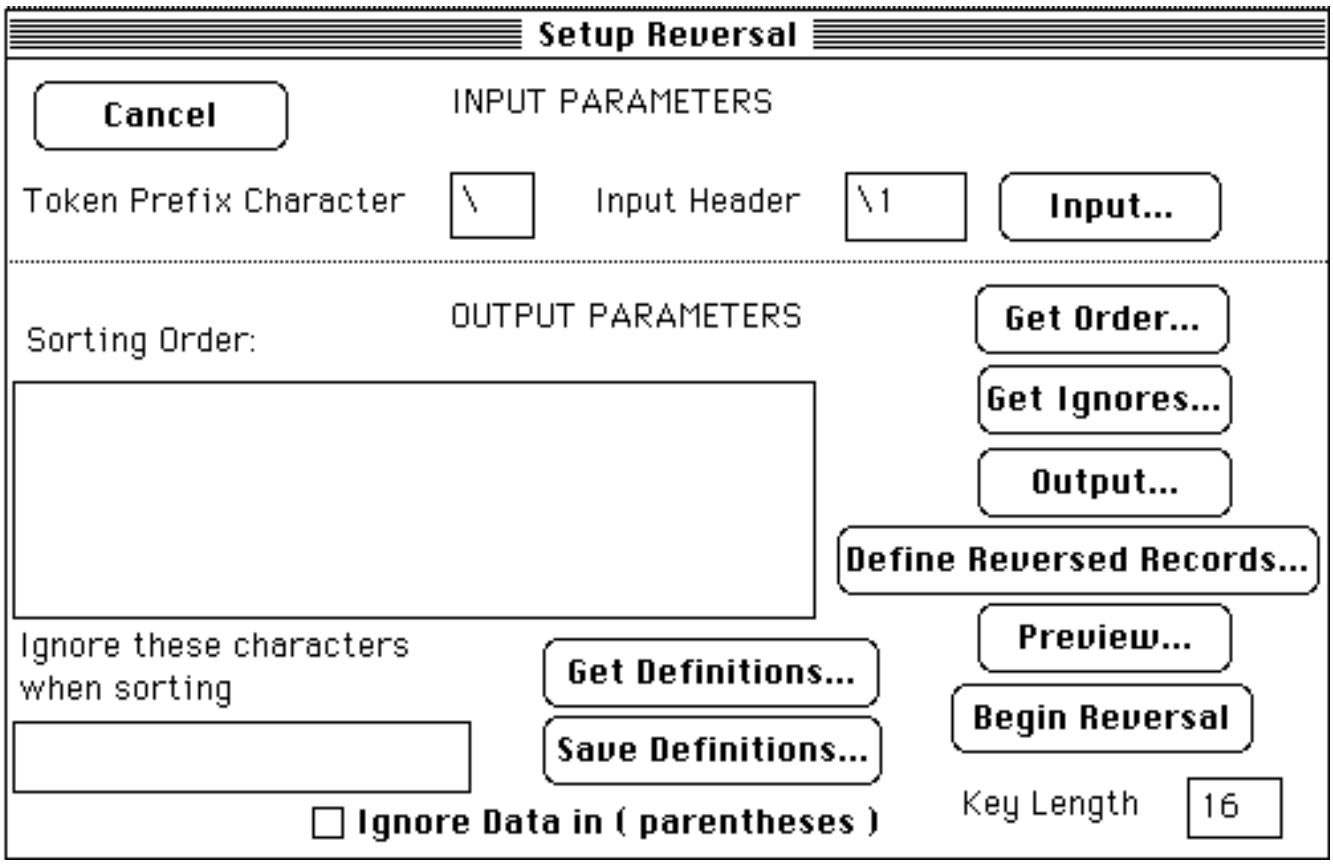
MacLex has a very powerful and flexible reversal facility. It will allow you to define how reversed records are to be built, to preview the records built by your definitions, to change the definitions and preview repeatedly until you are satisfied with the output, to save your definitions in a control file, retrieve them from there, and to initiate a full lexicon reversal when you are satisfied with the previewed output. Remember, reversing takes a lot longer than sorting - around ten times as long, or more.

To enter reversal mode, choose the **Reverse Lexicon** item in the Lexicon menu. After a short delay you will see the following dialog - remember the contents of some text boxes may be different for you.



Several items in this dialog are familiar from our previous discussion of lexicon mode and the dialog which comes up when you type **Load Lexicon...**, so we will not discuss them again here.

One thing you need to bear in mind is that everything below the dotted line near the top of the Setup Reversal dialog shown above applies to the output records, not to the input ones. That is, these items apply to the reversed records that MacLex builds in response to the definitions

you supply. It is therefore possible that you will want a different special sorting order, and perhaps different characters ignored, for the headers in your reversed records, and this is where you would do it.

One thing that MacLex assumes is that the same token will be used for the input record headers as is used for the output record headers[1] . NOTE CAREFULLY, **this does not mean that the input header field's data is what must appear in the output headers**. If that were the case we would not have a reversal, but just an export as has been discussed earlier. MacLex allows you to make any field type in the input records be the one used for the output headers; and it also permits you to specify a new name for the token to be used for that field in the output records. You can use this renaming facility to make the output records' header fields have the same token name as for the input records. We will see how to do this below.

The features of the Setup Reversal dialog are as follows:

The **Token Prefix Character**.

Type here the unique character you wish to use as the first character of every token in records, if different from the default (a backslash). The character in this box applies to input records, and is used by MacLex to divide each input record up into its fields.

The **Input Header**

Tells MacLex what token to look for in the input files in order to divide them up into records.

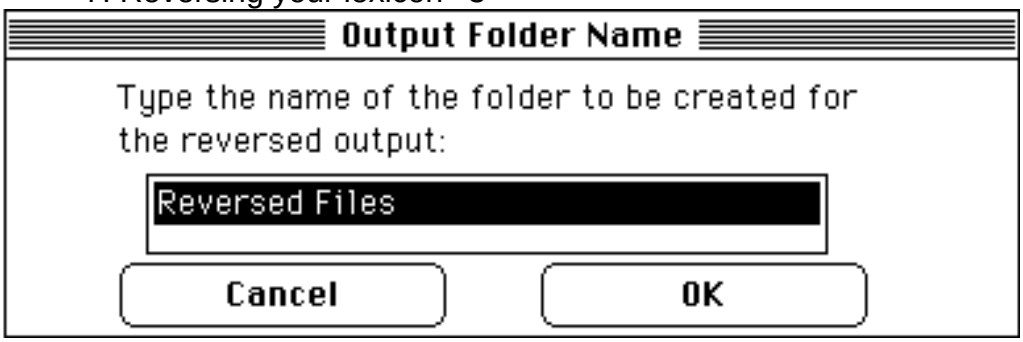
The **Input...** button

Puts up a standard file dialog which allows you to choose one of the files of your lexicon, and MacLex then uses this information to get the names of the other files in that folder. That is, it works the same as the **Locate Lexicon Files...** button in the lexicon setup dialog discussed earlier.

The **Output...** button

This button works a bit differently than for sorting mode. Instead of a file dialog you are presented with the following dialog:

[1] I could remove this requirement if users would prefer more flexibility.



This dialog does not ask you to nominate an existing folder which is to receive the reversed output files. Instead it just asks for a destination folder name. When MacLex is ready to create the final sorted output files, it will create a folder having whatever name you supply using this dialog, and send all output files to that folder.

MacLex always places that folder in the folder which has your input lexicon files. This is safe, because MacLex never looks into a nested folder when accessing the input files from the folder you specify. It has the advantage of keeping your output with your input, yet safely separated from it. It makes it possible for you to drag all your input files, reversed output files, and control files to another disk with just one folder drag; or to back up the lot with just one **Duplicate** choice at the desktop.

If you like, after reversal has finished, you can go to the desktop and drag the folder of output files to another folder, or to another disk.

The default output folder name is "Reversed Files". You can use this name repeatedly, but be aware of the following behaviour. If MacLex finds a folder of the same name as the one supplied by this dialog[2] already existing in the folder of input files, then before MacLex sends any output files to the folder it will first erase all of its existing files, if any are found within it.

So, if you want to do several reversals of different kinds based on the one input lexicon, then you will need to supply a different output folder name each time you do a reversal. If you don't, you will lose the earlier reversal files and have to do the reversal again. Of course, if you don't want to retain the earlier reversal files, then this behaviour will be exactly what you want.

The **Begin** button.

[2]irrespective of the folder name, whether Reversed Files or some other name

When you are satisfied that you are ready to initiate the full reversal process, then hit this button. The dialog will disappear, and you will get various messages in a status dialog as MacLex works its way through the reversal process.

Cancelling Prematurely

The reversal operation can be prematurely cancelled, safely, by typing the period (.) key while holding down the command key. All temporary files will be abandoned and removed from the disk, and the original state of the input files will be restored. However, any finished reversed files will be left in the output folder that MacLex creates. You can ignore these - they'll be removed automatically next time you send reversed output to that folder; or you can delete them manually if you wish.

For the curious:

Reversal is a three stage process. In stage one each input file is loaded, and reversed records build from each input record according to your instructions (see below). The output records are stored in temporary files with names TEMP0, TEMP1, TEMP2, etc. - one such file for each input file[3]. They are stored with your input files for as long as MacLex needs them, and deleted automatically at the end of the third stage.

Stage two, MacLex switches to sort mode, prepares an index table based on the contents of the temporary files, and sorts it.

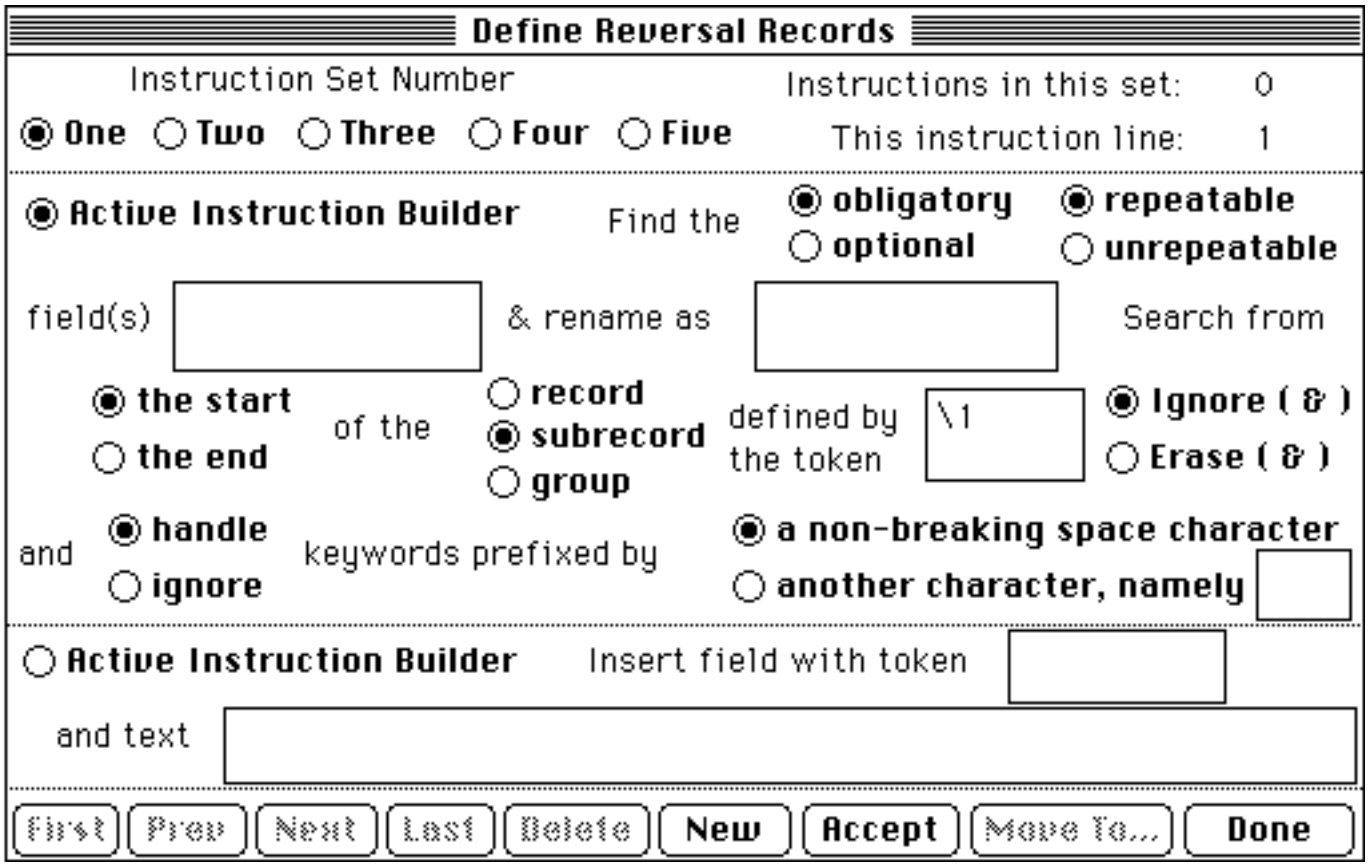
Stage three, MacLex uses the input table and the folder name your supplied at the **Output...** dialog to send the sorted reversed records to the output folder, creating the folder first if it does not already exist, and using the **Automatic filenames** option to construct appropriate filenames for the output files. It will create 50% more[4] output files than input files. (**Automatic filenames** is chosen for you, you cannot change that.)

[3]In the event of power failure during the reversal process, you might find these files on your disk. MacLex deletes them when it has finished with them, if allowed to run to completion. Just delete them.

[4]Because reversal can produce multiple output records from each input record it usually is the case that the reversed lexicon is larger in size than the input one. I have found the blowup factor to be about 150%, so adding an extra 50% more files keeps file sizes roughly the same as the input ones. Not a very important point, but if your input files are close to 32kb in size then this might keep the output files less than 32kb in size also, and make them able to be opened later by MacLex for inspection using the **Open...** item in the **File** menu. Do not use MacLex or MacSort to open a file larger than 32kb; if you do, be sure not to try to save it - you would lose data.

14. The **Define Reversed Records...** button.

This button allows you to teach MacLex how to construct your reversed records. Pressing it brings up the following dialog.



The first thing to notice is that the dialog is divided by dotted horizontal lines into four sections.

The first section.

The first sections has five radio buttons, only one of which can be selected at any time. Each of these buttons corresponds to a single set of instructions which you define with the rest of the dialog. One set of instructions gives MacLex the information necessary to construct reversed records according to one pattern[5].

Since there are five buttons, there are therefore five different ways or patterns that you can teach MacLex for building reversed records from a single input record. MacLex always tries every instruction set (ie. every

[5]Note carefully, this does NOT mean that one instruction set will produce just one output record. One instruction set may possibly produce dozens of output records, as we shall see. Each instruction set merely gives MacLex an ordered set of rules for building output records according to the pattern you supply.

pattern) on every input record, building as many as possible reversed records which are consistent with your instructions.

The right hand side of the first section of the dialog shows two numbers: the top number tells you how many instructions are in the current set - it is updated each time you add a new instruction to the set. The bottom number tells you the number of the instruction you are currently looking at in sections two or three of the dialog. The first instruction is numbered 1, and is always the instruction which builds an output record's header field (and hence is the field used for sorting the output records into correct order).

One instruction line is, with just a few exceptions discussed later, an instruction to MacLex telling it how to build a single output field of the reversed record. Thus, if an instruction set has six lines, you are giving it the capability of building output records with up to six[6] fields, in the order of the instructions in the set. So instruction one will build the first field of the reversed record, instruction two will build its second field, instruction three its third field, and so forth. This is somewhat of an oversimplification as we shall see below, but it gives the general idea.

MacLex also allows you to alter the order of the instructions you have supplied, thereby altering the order of fields in the output records. We will discuss this when we come to discuss the buttons at the bottom of the dialog.

Section two.

This is the main "instruction builder". It is comprised of a button at the top left which is labelled **Active Instruction Builder**. When this button is selected, MacLex uses the instructions in this section for building the next line of the output record. You inform MacLex to use this instruction by first filling out the boxes and clicking the buttons you want, then hitting the **Accept** button at the bottom of the dialog. MacLex will not remember an instruction unless you tell it to **Accept** it.

The rest of section two is the stuff which tells MacLex how to go about building that particular output field. We will defer out discussion of the details till after we have discussed the four sections of the dialog.

[6]Whether six fields are in every output record or not will depend on which, if any, fields are optional - which is something you can control. See the discussion of optional fields.

One thing to note: when the dialog comes up, the **record** button in the middle of this section will be selected (that is the default setting), and the text to the right and its following text box will be hidden. These are needed only for subrecords or groups, and hence are hidden when not relevant.

Section three.

This section also has an Active Instruction Builder button at the top left. Click it if you want MacLex to build the current output field using the instructions you supply in this section, rather than in the one above.

Section four.

This contains the buttons which allow you to manoeuvre around an instruction set, to accept or delete instruction lines, to move instruction lines, to ask for a new (blank) instruction line, and to exit the dialog when done.

The buttons are dimmed if their function is unavailable for some reason. For example, the **First** button will be dimmed if you are currently looking at the first instruction line. If you were at another instruction line, then it would not be dimmed and hitting it would take you to the first instruction line (wherupon it would become dim).

15. Concepts

Before we can discuss the instruction builders in sections two and three, we need a few more concepts. The discussion will now get somewhat long and detailed. I'm sorry about that. MacLex has so much power built into the reversal function that it is not possible to discuss it all in a page or two. Fortunately most people will need only a small fraction of the power which is there, and so don't hesitate to skim over any parts of this section which get a bit heavy to digest.

Firstly, we will review records, subrecords and groups.

Records

A record are a bundle of fields which start with a header field having a unique token which you specify to MacLex in a dialog somewhere. Sort mode, lexicon mode, and reversal mode each have a text box in a dialog where you can specify the header's token, or accept the supplied default token.

The only way MacLex can recognise a record is by its header field's token.

Subrecords

A subrecord is a bundle of fields which start with a header field having a unique token which you specify to MacLex in a dialog somewhere. You do the specification in section two of the dialog - more about that later. A subrecord's token must be different from a record's.

The only way Maclex can recognise a subrecord is by its header field's token.

A record may have zero, one, two, or more subrecords, without limit; except that the total number of fields in a record should not exceed 300. If a record has more, MacLex will ignore the extra ones beyond that limit when it builds reversed records.

Subrecords cannot be nested within records. They may only follow a record, and may only follow each other. A subrecord cannot be nested within another subrecord either.

For example, if \w is a record's header token, and \sw is a subrecord's token, then the following would be a valid record having two subrecords:

```
\w ...
   other fields
\sw ...
  other fields
\sw ...
  other fields
```

A subrecord finishes when one of the following happens: (a) another subrecord starts (whether of the same or different kind), or (b) another record starts.

Notice that the definition of a subrecord says nothing about the fields in it other than the header's token. A subrecord may have the same or different field types as are found in a record; their tokens can be unique or the same as corresponding fields in a record. MacLex doesn't care[7].

[7]This is not completely true. It is possible to exercise a little bit more control of the reversal process if subrecords have field types not found in records. If you use identical field types in both records and subrecords, then you have to use more of the high powered features of MacLex to prevent it doing such meaningless things as grabbing fields from two different subrecords and putting them in the one reversed record. MacLex is intelligent and has ways to prevent that from happening in most circumstances, so you need not be greatly worried. If our bundles are subrecords or groups then MacLex will block attempts to drag together things from different subrecords or different groups. See the next footnote for an example of how this can be done.

Groups

A group is a bundle of fields which start with a header field having a unique token which you specify to MacLex in a dialog somewhere. You do the specification in section two of the dialog - more about that later. A group's token must be different from a subrecord's and from a record's.

The only way Maclex can recognise a group is by its header field's token.

A group may have one, two, or more fields, without limit; except that the total number of fields in a record should not exceed 300. MacLex will ignore any extra ones beyond that limit when it builds reversed records.

Groups cannot be nested within groups. They can, however, be nested within subrecords, and also within records.

A group finishes when one of the following happens: (a) another group starts (whether of the same kind or a different kind); or (b) another subrecord of any kind starts, or (c) another record starts.

There can be more than one type of group in a subrecord, or in a record, or in both.

For example, if \g1 and \g2 are group header tokens, and \sw is a subrecord's token, and \w a record's token, then the following would be a valid subrecord having three groups in the order shown:

```
\sw ...
   other fields
\g2 ...
  other fields
\g1 ...
  other fields
\g1 ...
  other fields
\w ... (the beginning of the next record)
```

Notice that the definition of a group says nothing about the fields in it other than the header's token. A group may have the same or different field types as are found in a record or in a subrecord or in both; their

tokens can be unique or the same as corresponding fields in records and/or subrecords. MacLex doesn't care.

There are two further concepts we will need.

Firstly, the concept of a **bundle**. This is just a cover term for a defined group of fields. We have already discussed the kinds of bundles that MacLex recognises: **records**, **subrecords**, and **groups**.

However there is one more reason for having a distinct word for these three. When MaxLex goes looking in an input record to try and match a certain field type that you specified in your record building instructions, it does so only within a single bundle. You can specify the bundle to be the whole input **record**, or a **subrecord**, or a **group**. Section two of the dialog has three radio buttons where you specify the bundle type. The essential point to grasp is that the bundle limits the search for a matching field type to within a single bundle of whatever kind you nominate via the dialog.

One of the powerful features of MacLex's reversal capabilities is that once a bundle has been located for record building purposes, every instruction line in the current instruction set which refers to a bundle of that same type will try to find the nominated output field only within that exact same bundle of fields. Lets try make this clearer with an example.

Suppose \g1 and \g2 are group header tokens as before, and let \d be another field which occurs in each such group. Suppose part of the input record has the following structure (three groups):

\g2 ...
\d ...
\g1 ...
\d ...
\g2 ...
\d ...

Now suppose we told MacLex to make a \d field the output record's header, we would need one instruction line of an instruction set to do that. Suppose in the same set of instructions we told MacLex that the second output field is to be the record's header field (which might be dozens of fields earlier than the groups shown above); and then for the third output field we tell MacLex to grab a \g2 field from a \g2 group. In the example shown, there are two \g2 groups, so which \g2 field will MacLex grab?

It will always grab the one which is in the same bundle as the \d field that it had already grabbed when obeying an earlier instruction of the instruction set. How does it know how to do this?

MacLex analyses input records; it knows what subrecords they contain, what groups they contain, and what subrecords each group belongs to, and how extensive every subrecord and group is[8]. It also remembers which particular group and or subrecord it is working in as it builds output records, and so whenever an instruction specifies a bundle of a type used earlier MacLex knows exactly where to go to ensure that field matching is done within the same bundle. (That is, not just within a bundle of the same type, but within the actual same bundle as was used earlier in building the current record.) It does this automatically for you, you don't have to worry about it.

This keeps things which are logically related to each other together during the reversal; allowing MacLex to do such things as reordering the fields in a group without losing track of which group is the currently 'active' one. The same principles apply to subrecords, and of course to records as well since we only deal with one input record at a time.

Okay, now to the final concept. There are actually two kinds of groups. The kind we have so far been talking about is the kind which restricts a field-matching search operation to within the group's boundaries. Also, as we discussed above, one instruction line can initiate a search for one field type within a bundle which may be a group, subrecord, or record.

Suppose we wanted to tell MacLex to search not for a single field in a given bundle type, but rather for a collection (or 'group') of fields, the fields being in a certain order, and within the one bundle.

Such a collection of 'target fields' would be another kind of grouping of fields. The conceptual difference here is that such a grouping does not specify a limit for the search operation; that is done by the bundle specification, instead it merely says:

> "Okay MacLex, look for these fields within the bundle and you can expect to find them in the following order".

[8]It knows all this only so long as it can find enough information about what tokens are subrecord headers, and what ones are group headers. If your instruction sets don't specify those things, then MacLex won't do as good a job on the analysis of your records. You could improve MacLex's analysis skills by defining some instruction sets that you know will never produce any output records, and in those sets refer to all the subrecord and group tokens that MacLex should know about. This will become clearer later in our discussion.

Can MacLex do this? Yes it can. This is one of the ways that a single instruction line can produce more than one field in a reversed record. It's very useful when you don't want to change the order of fields as you transfer them to output records.

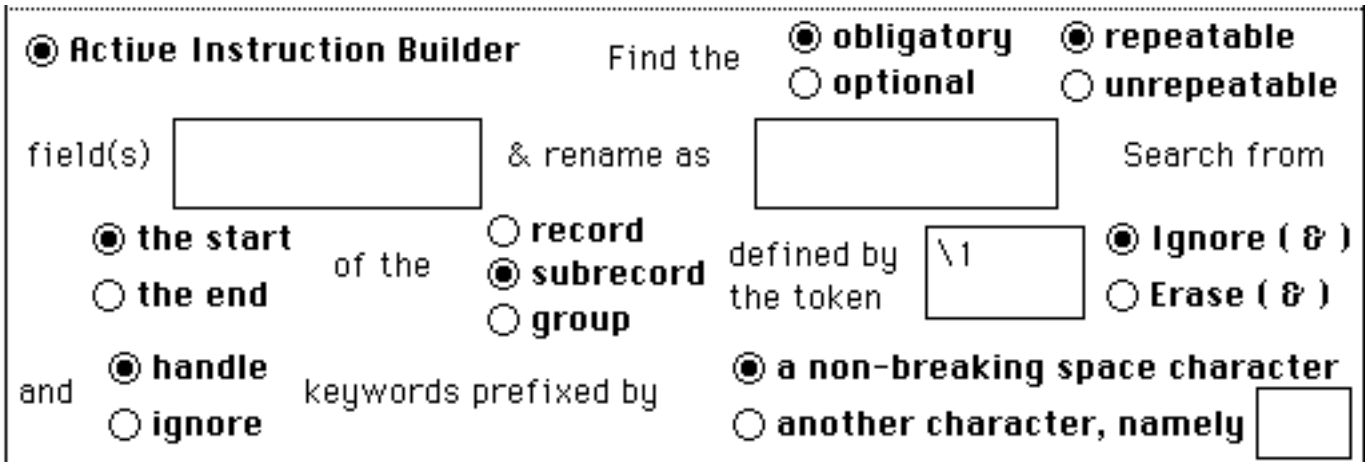
How can we do this? It's very simple; whenever you want MacLex to try to match more than a single field, you supply more than one field token in the first text box of the dialog. MacLex will start with the first token you type, and try to match each one in turn - moving in the one direction (either backwards or forwards) and confining its search to the bundle you specify. The fields do not have to be contiguous in the input record, but they do have to be in the order specified for the search to be judged by MacLex to have succeeded.

We need a way to talk about this kind of conceptual grouping of fields. Let us call them a "multifield set"; they are a 'group' to be searched for, not a 'group' specified in order to limit a search to a certain bundle of fields. I hope that is clear.

Okay, now we have the basic concepts, and if you have followed this far, then the rest should be easy. If you are having difficulty, don't be too concerned. MacLex has a very useful previewing feature which allows you to play with definitions and see what kind of reversed records come out the other end. Even if you don't understand all the details of our discussion here, you can play around in preview mode to your heart's content, knowing that everything you do there is completely safe and in no way affects your valuable data; and when you are satisfied with what is coming out, then just tell MacLex to go and make a proper job of it while you take a coffee break.

16. The primary instruction builder.

Let us look now in more detail at the instruction builder of section two of the dialog. It is the one you will mostly use, and it is very powerful. We will repeat it here for your convenience.



Let us start our discussion with the text boxes.

The first text box is where you type the token of the field type that you want MacLex to search for in whatever bundle you specify. If you want it to search for a \d field, just type \d in the box. We shall refer to the contents of the first box as the 'target' token, or by extension, the 'target field type', since this token (or tokens) is what MacLex tries to find in each input bundle.

If you want MacLex to search for a multifield set, then type the tokens which comprise the set of fields in the order you want, going from left to right. If MacLex is searching forwards, it tries to find each successive field type further down in the bundle (ie. closer to the bundle's end); if it is searching backwards, then it tries to find the first field (ie. the one you typed first in the box), and then attempts to find each successive field in the box by searching further up in the bundle (ie. closer to the bundle's start).

Hence if you typed \aa \bb \cc in the box, then a successful match will be made when searching in the forward direction if the input record's bundle has the fields in the order:

\aa ...
  other fields (or none)
\bb ...
  other fields (or none)
\cc ...

Alternatively, if searching backwards then for a successful match the fields in the bundle would need to be in the order:
\cc ...
  other fields (or none)
\bb ...
  other fields (or none)
\aa ...

MacLex requires that at least one token be typed in the first text box, otherwise it will refuse to **Accept** the instruction line, and alert you to the fact.

The second text box allows you to specify a new name for the token for whatever field is matched and sent to output. If you specify a multifield set in the first box, you can specify in the second box a new name for each token.

Note, you do not have to type anything in the second box. If left blank, MacLex will give the output field the old token's name. In the case of a multifield set, you need not supply a new name for them all. Any left unspecified will have their old token name used in the output. However, be careful. If you want, say, the last one of a multiset to have a new name, then you have to explicitly type the names of every token preceding it - even if they are unchanged.

The third box is where you specify the bundle type. For a record as the bundle there is no point typing anything, since the input record is the only possible candidate. But for subrecords or groups, you must type the bundle's defining token in this box. If you type more than one token, only the first will be used.

We will discuss the first two pairs of radio buttons last. There is more to them than meets the eye. Let's finish the easy stuff first.

You specify the search direction by the radio buttons **the start** and **the end.** If you specify the former, the search starts at the start of the bundle and proceeds to its end. If the latter, then it starts at the end of the bundle and proceeds backwards to its start.

The buttons **Ignore (&)** and **Erase (&)** allow you to instruct MacLex to erase the opening and closing parentheses bracketing data to be ignored for sorting purposes at the start of a field. It does this if you hit the **Erase (&)** button, but only provided the opening parenthesis is the first character of the data in the matched field. Parentheses elsewhere in the field, if there are any, remain unaffected. Remember, only the opening parenthesis and its matching closing parenthesis are erased, the data between them is left untouched.

The final section of this instruction builder tells MacLex whether or not to do certain predefined operations, but on a header field only. You can change the settings of the **handle** versus **ignore** buttons, and the ones which follow it (and the final text box), for any field you like. However, when MacLex builds output records, it uses these particular instructions only for the output header. For any other field they are ignored.

What do they do? Well, consider the following field which we will assume is part of an input record.
\ms be happy

As it stands, if you asked MacLex to use this field as a reversed record's header field, then it would ultimately be sorted with the 'b' entries, which is probably not what you would want. It would be much better to have it with the 'h' entries.

We can accomplish this by using a unique character, which we will call a **keyword prefix**, in the input lexicon to mark a keyword in a field's data. It only makes sense to do this if you later intend to use that field to form headers in reversed records. And only do so if the keyword is not the first word of the field's data.

You can use any character you like, but it should be unique (that is, not appear elsewhere in your data). I forgot to make MacLex a bit more clever at this point; it should accept as a keyword prefix only those which occur following carriage return, tab or space. The present version doesn't check for this, but just looks for the specified character and matches the first one it finds in the field's data regardless of context. Because of this it really does need to be unique.

The best character to use as the keyword prefix is a non-breaking space, which you can get in any font by typing "option space" (ie, hold down the option key and type the space bar). On the screen it looks like a regular space, but is actually a different character. MacLex looks for this one if you use the default setting. In the dictionary data used in this documentation you will notice that I use an asterisk to make it

more visible. This is just for instruction purposes, you don't have to use an asterisk; you'll get a much nicer display if you use non-breaking space.

What does MacLex do with keyword prefixes when it finds them? It takes the word which follows it, moves it to the front of the data, inserts a colon and space following it, and then copies the rest of the data after the inserted space, but placing two underline characters where the keyword formerly was - to act as a placeholder.

MacLex takes only the first keyword in a field if there is more than one. If you mark the first first word as a keyword, MacLex will just ignore the keyword prefix since the field will get sorted to the right place just as it stands.

Whenever MacLex does the above kind of rearrangement, it omits the keyword prefix from the output field's data. (Any subsequent keyword prefixes are not removed - but this is not great problem since they could thereafter be removed by the Export function.)

Hence, if the field were instead:
\ms be *happy
and we specified ***** as the keyword prefix, and told MacLex to **handle** it, then MacLex would produce the following output field (assuming the same token is used for the output field):
\ms happy: be __

17. **Obligatory** versus **Optional** fields.

Okay; now we come to the first two sets of radio buttons. Let us deal with the **obligatory** versus **optional** pair first.

If you specify that a field is **obligatory**, or that a multifield set is **obligatory**, then the target field (or target multifield set) MUST be matched within the nominated bundle. If it isn't, then that whole current output record is abandoned, no matter how many fields of that one record have been successfully build up to that point. A failed match for an obligatory field spells sudden death for that output record.

In the case of an **obligatory** mulitifield set, if just one field is not matched, then the whole set fails, and hence the whole record is abandoned. All must match, and match within the one bundle, and do so in the order specified in the dialog.

The first instruction of every set is **obligatory**, since it is the header and a valid output record is not produced if the header token is not

present. If you try to make a header optional, MacLex will just switch the radio button back to **obligatory**, preventing you from making such a mistake.

On the other hand, if you specify a field as **optional**, then MacLex will try to match it in the specified bundle, but if the match does not succeed, the whole record is not abandoned; instead, there is nothing placed in the output record from that instruction line.

Specify a field, or multifield set, as **optional** when the target fields may not be present in every input record, but whenever they are, you want them in the reversed records as well.

In the case of an **optional** multiline set, then it is permitted for one or more fields to be unmatched within the nominated bundle. Those that are successfully matched are sent to output. If none are matched, then nothing is sent to the output record.

18. **Repeatable** versus **Unrepeatable** fields

Now we come to the **repeatable** versus **unrepeatable** buttons. Perhaps the most important choice of all, and having the most complicated behaviour.

When a field, or multifield set, is specified as **unrepeatable**, MacLex looks for only one instance of it in the nominated bundle. The first such matching field it finds in the bundle is the one it sends to output. That's all there is to know about **unrepeatable** fields.

For **repeatable** fields, the rules are very much more complex. So here goes....

When a field is specified as **repeatable**, the basic behaviour that MacLex follows in every instance is to match that field as many times as possible within the nominated bundle. **Repeatable** basically means

> "do it over and over again, for as long as you can find another matching field".

However, there are a couple of complicating factors. Firstly, it makes a difference whether or not that instruction is an instruction to build an output header, or to build a non-header field. In each of these situations MacLex behaves differently (see below).

The other complication occurs when the instruction is for building an output record's header field. If the instruction line says to match the repeatable field within a specified bundle, then not only does MacLex look for the field as many times as possible in the one bundle, but it also repeats the process for all such bundle types of that kind in the record. If the bundle is the record, then only one such bundle can exist in each input record, which simplifies things a bit. But when the bundle is a subrecord or group, things get a bit more complex (but fully predictable) as we shall see below.

Let us take all the possibilities in order and describe what happens. But first, let us give a sample Djinang record with which we can illustrate what MacLex does in different situations.

```
\w djalgi
\p adj
\c ma wu
\d salty
\ds
\d angry
\ds
\d strong *taste
\z
\d bad-tempered
\v djalkngi
\sw djalgidjidji
\sp v3 it
\sc ma
\sd be *hurting
\ds
\sd become angry
\sz
\sd be in pain
\ds
\sd hit oneself in *grief
\ds
\sd be *sour
\sz
\sd be mildly *poisonous
\ds
\sd be *rash from anger
\sv djalkngdjidji
\sw djalgidjnyirdjigi
\sp v1 t
\sd cause to be in *pain
\sz
\sd wound
\ds
\sd make *wild
```

Case 1.

Trying to match a **repeatable** non-header field, in a given bundle type. (Note, 'non-header' means the instruction line is not line 1 of the instruction set currently being used.)

MacLex will scan the bundle in the specified direction, matching the target field as many times as possible, each time sending the matched field to output - thereby building output fields in the order they are encountered in the input record.

Suppose our instruction specified \d as the repeatable second field of the output record, with the whole record (ie. \w) as the bundle. Then MacLex would build the following fields as the second, third, forth, etc. until all \d fields are matched. The fields would be consecutive in the output.

\d salty
\d angry
\d strong *taste
\d bad-tempered

In this instance, if our instruction set has a third instruction, it would not produce the numerically "third" output field, but rather the sixth, since there have already been five others produced before it (ie. the header and then the four fields shown above). Each new field matched and sent to output goes at the end of whatever has already been sent. In this way a few non-header instruction lines, if their target fields are nominated as **repeatable**, can produce quite a number of output fields in a single output record.

Case 2.

Okay, the next thing to grasp about a **repeatable** field is its behaviour when you specify it as the header field for the output record. (That is, it is the target field type specified in your first instruction line of the instruction set currently in use.)

Again MacLex will match as many instances of it as possible within the specified bundle, but with one very important difference... EACH TIME A SINGLE FIELD IS MATCHED, A WHOLE NEW OUTPUT RECORD IS BUILT USING THE REST OF THE INSTRUCTIONS IN THAT INSTRUCTION SET.

Consider the Djinang record given above. Suppose we had the following two instructions comprising a single instruction set:

Instruction line one: (the header instruction):
" Find the **obligatory repeatable** field **\d** & rename as **\w** Search from **the start** of the **record. Ignore (&)** and **handle** keywords prefixed by **another character, namely *** "

Instruction line two:
" Find the **obligatory unrepeatable** field **\w** & rename as **\d** Search from **the start** of the record. **Erase (&)** and ... (this latter part is irrelevant since this field is not an output header) "

The sequence of events goes as follows:

a. MacLex matches the first \d field, which will be "\d salty".
b. now Maclex pauses in its search for \d fields, and uses the rest of the instruction set to build this particular output record before proceeding further with the output headers. So line two matches "\w djalgi". This is the last instruction of the set, so one complete output record has now been built, and it will appear as follows:

    \w salty
    \d djalgi

c. MacLex now takes up where it left off with the **repeatable** header, and looks for another field with that token, starting from the field immediately following the last \d matched. Hence it finds "\d angry" and sends it to output, with appropriate token name changes.
d. Again MacLex halts the search for \d fields, and uses the rest of the instruction set to complete that particular record. So again it matches "\w djalgi" and sends it to output. The output now looks like the following:

    \w salty
    \d djalgi
    \w angry
    \d djalgi

e. MacLex takes up again with its search for a \d header field. Since there are four of them in the record, this process will not stop until all four have been matched and therefore four separate output records will be produced. What the other two will be should be obvious from what has already been produced - each one has \d djalgi as its second field.

Case 3.

Okay, what happens if you specify a **repeatable** multifield set as the header? The answer is: "nothing that you would not expect". It works exactly like specifying a single field type as the **repeatable** header; except that all the fields must be matched each time (since headers are forced to be obligatory), and matched in the specified order as was typed in the dialog box. Of course, only the first of the matched fields actually becomes the header field in each output record.

For example, suppose we nominated \d as the bundle header token; then if we nominated the fields \d and \ds as the **repeatable** multifield set for the first instruction line (and left the \ds token's name unchanged), we would get three output records produced as follows:

    \w salty
    \ds
    \d djalgi

7. Reversing your lexicon 23

\w angry
\ds
\d djalgi
\w bad-tempered
\ds
\d djalgi

In this instance we don't get four output records, because the third \d field is followed by a \z field and a new group begins immediately after that, causing a failed match for the \ds field (the \ds you recall has to be matched in the same bundle as the \d field), and thereby causing the third output record to be abandoned.

The fourth \d field is not closely followed by a \ds field, but one is found much further down in the record, which would allow the third output record to be built, since there are no more \d fields to close off the group and thereby prevent that particular \ds field from being matched[9].

Case 4.

Okay, what happens if you specify an output record's header field as **repeatable**, and also specify its bundle as something less than the record - say the bundle is a **subrecord**, or maybe a **group**? MacLex does the following things:

---

[9]The observant reader will notice that the matching \ds field for the third output record comes from the fields following \sw - and the latter is clearly a subrecord header field. If we only had the one 2-line instruction set mentioned earlier, MacLex would not know that \sw starts a subrecord, and so there would be nothing to stop this 'spurious' match from being made.
Also note: in practice, we are permitted to designate a record's header token, in this case \w, as a subrecord bundle's token. And for Djinang records of the kind illustrated earlier we also will have a second instruction set which builds records from \sd fields as repeatable headers, and unrepeatable \sw fields as the second output field, with the subrecord \sw as the confining bundle. When the input record is analysed, MacLex detects the \sw field, knows that a subrecord starts, and therefore that the earlier 'subrecord' (ie \w) must finish there. When the output records are being built, MacLex will then not let a record-building instruction cause it to reach across the subrecord boundary to grab the \ds field following the \sd be *hurting field. This leads to a failed match of an obligatory field, and hence the"\w bad-tempered \d djalgi" record would not be built. Note therefore that it is permissible, and can be quite useful, to nominate the input record's header token as a subrecord bundle. MacLex is smart enough to not become confused by this dual use of the one token name. Another thing to notice is that it does not help MacLex if you specify fewer instruction sets rather than more. It is a mistake to think that to 'keep it simple' would be best. MacLex benefits greatly from each instruction set you specify; the more sets, the more information it can glean about how to analyse input records, and the more comprehensive and accurate will be the analysis it then can make of each one. A better analysis means that MacLex finds it easier to detect and reject spurious output.

8. The **Define Reversed Records...** button. 23

a. It finds the first bundle in the record which matches the specification in the instruction line.
b. It then looks for as many matches of the target field as it can find within this one bundle, making a single output record each time a match is made, and using the rest of the instruction set to fill out the rest of the record's output fields each time it does so.
c. When the first bundle is exhausted (ie. no more records can be built from it); MacLex then starts with the field immediately following that bundle and tries to match another bundle of the same type. If it finds one, it goes to step b. and produces as many output records from that bundle as possible. Then it repeats steps c. and b. until there are no more bundles of the nominated type in the record.

Clearly this process is capable of producing quite a lot of output records if your input records are structurally complex.

Case 5.

The equivalent of case 4, but with the target as a multifield set, works as you would expect by extending the above cases to this situation. Probably noone would need that much power, but it is there if you require it.

19. The static field instruction builder

Now we need to discuss the third section of the dialog, which contains the second instruction builder. This just has two text boxes. The first box is where you type the output field's token. The second box is where you can type the data which is to be in the field.

We call this kind of a field a 'static field', because its data does not change. It puts the same field, with the same data, in every record that is successfully built by the rest of the instructions in the set. Such a field might contain a date, for example. It can have anything in it you like; and up to 255 characters of data are permitted.

You can have none, one, or several such fields in a reversed record. If you specify one, it will always be there in the output, since there is no matching of tokens involved.

20. The command buttons

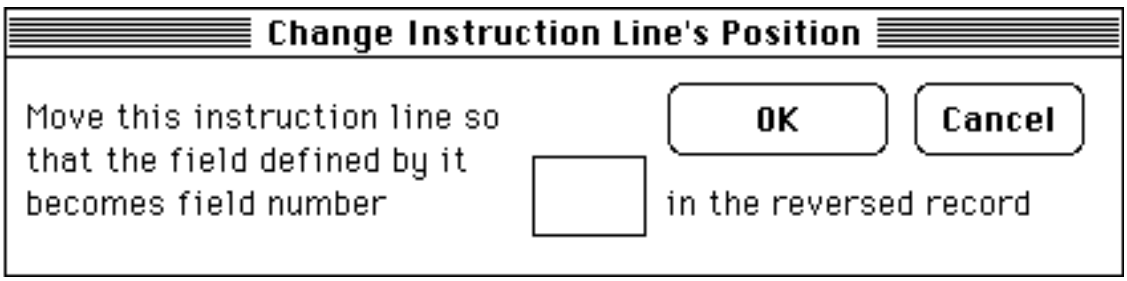
Finally, we come to the buttons at the bottom of the dialog. From left to right they have the following functions:

The first four are navigation buttons, they take you to the **first**, the **previous**, the **next** or the **last** instruction line of a given instruction set.

The **delete** button does exactly what is says - it deletes whatever instruction line is currently showing, and the deletion is unrecoverable. If you mistakenly delete, you'll have to fill out a blank instruction line, and possibly use the **Move To...** button to put it in the correct place in the set of instructions.

The **New** button gives you a new ('blank') instruction line for you to fill in. It becomes part of the instruction set only when you type the **Accept** button. If you fill out a **New** instruction line, and move to another line without **Accept**ing the **New** one just filled out, your instructions for that line will be forgotten.

If you change you mind about the order of fields in the output, you can rearrange the order of instruction lines quite easily using the **Move To...** button. First get the instruction line you want to move showing on the screen, then hit the **Move To...** button. You will get the following dialog.



In the text box just type the number of the instruction line that you want the current line to become. For example, if you want it to become the header, type 1. What you type there is not acted upon by MacLex until you press the output button, so you can change your mind while the dialog is up. Whatever number is in the box when you press OK is what is used.

Finally, when you have finished defining all your instruction sets, and you must define at least one set, then type the **Done** button. This will take you back to the Setup Reversal dialog. There are three more buttons in that dialog which we have not yet discussed, so let us do so now.