WRITING FILTER PROCESSES FOR THE SAGA EDITOR

Peter A.Kirslis

Department of Computer Science

Unviersity of Illinois at Urbana-Champaign

Urbana, Illinois

July, 1985

Writing Filter Processes for the SAGA Editor

Peter A. Kirslis

July, 1985

1. Introduction

The SAGA editor provides a mechanism by which separate processes can be invoked during an editing session to traverse portions of the parse tree being edited. These processes, termed *filter processes*, read, analyze and possibly transform the parse tree, returning the result to the editor. By defining new commands with the editor's user–defined command facility, which invoke filter processes, authors of filters can provide complex operations as simple commands. A tree plotter, pretty printer, and Pascal tree transformation program have already been written using this facility. This document introduces filter processes, describes parse tree structure and the library interface available to the programmer, and discusses how to compile and run filter processes. Examples are also presented to illustrate aspects of each of these areas.

2. The SAGA Editor

The SAGA editor is a language–oriented editor based upon a table–driven LALR(1) parser. As the user inputs his program, the editor analyzes the input and interactively builds a parse tree internally. Modifications are incrementally reparsed. Since the data is stored in parsed form, it is a simple matter to make the parse tree available for additional analysis by other programs. These programs, using pre–

defined library routines, can walk the parse tree collecting data. They can modify some fields in the tree directly, and can transform the structure of the tree by writing a text file which is passed back to the editor to be parsed and inserted in place of some portion of the existing tree. The editor provides both user–defined command sequences and command files to facilitate the use of these programs. See the SAGA editor user manual for more information about the editor itself.

3. The Parse Tree

The parse tree which is built by the editor consists of three types of parse tree nodes, and a header record. The node types consist of *terminal, non–terminal* and *marker*. The header record contains the root node of the tree, how many syntax or semantic errors are present, and other information. Each of these tree components is described in more detail in the following sections.

3.1. Parse Tree Structure

The root node of the tree is a non–terminal, and corresponds to the start symbol in the grammar defining the language in use. Each non–terminal node in the tree represents a non–terminal token on the left hand side of a production rule in the grammar. The children of each non–terminal node correspond exactly to the terminal and non–terminal tokens on the right hand side of this production rule. Each parent node points to its leftmost child; each child points to its right sibling (the rightmost child has no sibling); and each child points to its parent.

Each node also contains a *rightmost descendant* (or *rdescend* pointer). For terminal nodes, this descendant is the node itself. For non–terminal nodes, this node is the rightmost terminal node in this non–terminal's tree.

Terminal nodes are also linked together in a doubly–linked list, with each terminal node pointing to both the terminal node just preceding it and following it.

Each node also contains a *left thread* (or *lthread*) field, which points to the node which was on the top of the parse stack just before this node was shifted onto the stack. This field is used by the editor to reconstruct intermediate stages in the parse when a modification is being made to this portion of the tree.

3.2. Fields Common to all Nodes

In addition to the above mentioned link fields, each node also contains the parse state of the parser after this node was shifted onto the parse stack, a set of Boolean flags, some formatting information for printing, and integers which order the node relative to others around it.

Although the parse tables are not directly available, the editor module which provides access to them can be retrieved and added to the filter process. Queries concerning the parse states of nodes can then be made, for example, in a parse tree consistency checking program.

The following flags are available in each node:

FPOINT	An editor pointer is set at this node,
FDELETE	this node has been deleted from the parse tree,
FMODIFIED	this node is new to the parse tree since the last parse tree difference was taken,
FMAKE	reserved for use by the SAGA make facility,
FNOTPARSED	this node has not been parsed,
FSHIFTREDUCE	this node contains no parse state, since a shift–reduce action was performed by the parser,
FSELECT	used to highlight portions of the parse tree,
FSEMDELETE	this node has been deleted from semantic tables,
FLEXERR	this node contains a lexical error,
FSYNERR	this node contains a syntax error,
FSEMERR	this node contains a semantic error,
FELIDE	this node is being elided (not printed),
FSUBELIDE	this node is nested in an elision.

The FPOINT and FSELECT flags are only set during an editing session. FDELETE, FNOTPARSED, FSHIFTREDUCE, FLEXERR, and FSYNERR are manipulated by the parser. FMODIFIED is used by the tree–differencing facility. FMAKE is used by the SAGA make facility [Badger, 84]. FSEMDELETE and FSEMERR are manipulated by the semantic analysis routines. FELIDE and FSUBELIDE are intended to guide the printing (and hiding of detail) of te parse tree; they are not fully implemented yet.

Terminal nodes and non–terminal comment tree nodes contain *skipline* and *skipcol* fields to guide the printing of the node. The skipline field stores the number of newline characters to be output before the ascii string representing the token is printed, while the skipcol field stores the number of space characters to be output. The actual character string to be printed for the node can be retrieved with a call to one of the library routines to be presented later.

3.3. Terminal Nodes

Terminal nodes contain the token code of the terminal, a pointer to the print name of the terminal, and the length of the print name (not including preceding newlines and spaces). All relevant information described earlier is also present.

3.4. Non–terminal Nodes

Non–terminal nodes contain the token code of the non–terminal, the number of the production rule in the grammar for which this node represents the non–terminal on the left hand side of the production, and the leftmost child (first token on the right hand side of the production) of this node. All relevant information described earlier is also present.

3.5. Marker Nodes

If at any time during a parse, the parser encounters an error or an incomplete surrounding tree in its environment, the parser will suspend the parse. When it does so, it leaves a discontinuity in the tree. In order to be able to resume the parse at a later time, a marker node is inserted into the tree at the point of the error. This node stores a pointer to the node currently on the top of the parse stack, a pointer to the node which caused the error (if any), and a pointer to the "next" marker token in the parse tree. By next is meant the marker for the most recently occurring previous error. The first old terminal node following the new input (and any of its ancestors whose left thread pointer are identical) also have their left thread pointers reset to point to this marker node, so that later reparses that happen to reach this area of the tree will detect this discontinuity.

In general, it is not recommended that filter processes traverse parse trees containing discontinuities, since the tree structure will be incomplete. The fact that a parse tree has syntax errors (and semantic errors) can be detected by querying the *status*, *synerror* (and *semerror*) fields in the parse tree header record.

4. Filter Process Structure

Now that the reader has some idea about the structure of the parse tree, we will describe the structure of a filter process (program), and how it accesses the filter library of node access routines. The library itself will be described in the next section.

Two steps are necessary to use the filter library in a Pascal program. First, the filter library header file must be included. The Pascal program does this via the following *include* statement:

```
program myfilter(output);
#include "../../../src/filterlib/hdr/filterlib.h"
end. (* myfilter *)
```

The path given above assumes that the filter program resides in the filter process directory within the SAGA source code directory hierarchy. You will want to adjust it if the program source resides elsewhere.

This include file in turn references several include files used by the SAGA editor. These files define the interfaces used by the routines which access the parse tree nodes, many of which are used by the SAGA editor itself.

The include file which declares the routines which access the parse tree is *src/editor/hdr/nodeaccess.h.* The routines in this file are the ones described in the next section of this document.

The second step needed to use the filter library occurs at compilation time. The compiled object (relocatable binary) is linked and loaded together with the filter library as follows:

```
pc -c myfilter.p
pc -o myfilter myfilter.o ../../../src/filterlib/filterlib.
```

(The filter library may instead be stored in *saga/lib/filterlib.*) The compiled program may be used either in conjunction with an editing session, or stand alone on a parse tree file produced earlier by an editor. This latter ability can be helpful when debugging filter programs. When used with the editor, the filter process is invoked with some standard command line arguments, including the name of the directory containing the parse tree and related files. These conventions will be described in detail in a later section.

5. The Filter Library

The filter library consists of a number of functions and procedures which read (and some which modify) certain fields within nodes of the parse tree. These routines are divided into several categories, and presented in the following order: opening and closing the parse tree file, retrieving information from the header record, accessing pointers which connect nodes, accessing fields common to all node types, accessing fields specific to each node type, and modifying selected parse node fields.

5.1. Opening and Closing the Parse Tree Files

The following functions are provided to establish a connection to an existing parse tree file:

```
function Ninitialize          (* open a parse-tree directory *)
   (var pathname: charbuf;    (* name of directory *)
    var parsefile: filerange; (* return: file tag of parse tree *)
    var stringfile: filerange (* return: file tag of string table *)
   ): integer;                (* return 0 if o.k., -1 for error *)
   external;
```

The pathname parameter is the name of the directory containing the parse tree files. It is supplied to the filter process as the first argument on the command line; using the *argc* and *argv* Pascal system routines, this string can be retrieved and passed to *Ninitialize*. The second and third parameters are returned by *Ninitialize*, and are passed to other filter library routines.

Function *Nopen* is provided for completeness, should the filter process wish to define its own paged data structure, or open and reference a second parse tree file in addition to the first one opened above. If a second tree is to be accessed, both the parse tree file and string table file need to be explicitly opened. If only one tree is to be accessed, and no other files referenced, then this call need not be used.

```
function Nopen                (* open an existing paged file *)
   (var pathname: charbuf;    (* name of paged file *)
    var recsize: integer;     (* return: record size in bytes *)
    var recperpage: integer   (* return: records/page (for Nusebuffer()) *)
   ): integer;                (* return filetag if o.k., -1 if error *)
   external;
```

Note that the pathname parameter to *Nopen* refers to the actual file to be opened, not just the directory which contains the file. The remaining parameters are returned, and are to be passed to *Nusebuffer* to

assign to the file a buffer in which to place the file's data.

Nusebuffer assigns a buffer to the file to be paged into memory. The *bufaddr* parameter should be declared as a pointer to an array from 1 to n of records, where the record type is the record being paged. This pointer needs to be passed to the routine as ord(<pointer>) (so that *Nusebuffer* can be used for many record types). The *reccount* parameter specifies the number of records in the array, which must be an exact multiple of the page size returned by the *Nopen* call. If only *Ninitialize* called, *Nusebuffer* need not be called either, since the code in *Ninitialize* declares a buffer to contain the paged data, and also makes a call to Nusebuffer *itself.*

```
function Nusebuffer          (* assign data buffer to paged file *)
   (filetag: integer;        (* assign buffer to this file *)
    bufaddr: integer;        (* memory address of buffer (ord(b)) *)
    reccount: integer        (* record size of buffer *)
   ): integer;               (* return 0 if o.k., -1 for error *)
   external;
```

The *Nclose* routine should be called when the filter process is finished. If the parse tree file was only read, this call is not strictly necessary. However, if any fields were modified by the filter process, this routine must be called in order to write out the remaining data in memory and close the file, otherwise information may be lost.

```
function Nclose              (* close an open paged file *)
   (ft: filerange            (* file tag *)
   ): integer;               (* return 0 if o.k., -1 if error *)
   external;
```

5.2. Retrieving Information from the Header Record

These routines are provided to retrieve information from the header record: The most useful of these are *Nroot*, which returns the root node of the parse tree, and *Nstatus*, which returns the parse tree status, either COMPLETE or SUSPEND. COMPLETE will be returned only if the tree contains neither syntactic nor semantic errors. The

```
function Ndelete             (* get no. of explicitly deleted nodes *)
   (ft: filerange): nodeindex; external;
```

```
function Nmodified              (* get parse tree modified flag *)
  (ft: filerange): boolean; external;

function Nreadonly              (* get parse tree readonly flag *)
  (ft: filerange): boolean; external;

function Nroot                  (* get parse tree root node *)
  (ft: filerange): nodeindex; external;

function Nsemerror              (* get parse tree semantic error count *)
  (ft: filerange): integer; external;

function Nstatus                (* get parse tree status *)
  (ft: filerange): statuskind; external;

function Nsynerror              (* get parse tree syntax error count *)
  (ft: filerange): integer; external;

function Ntreesynlist           (* get parse tree .treesynlist pointer *)
  (ft: filerange): nodeindex; external;
```

5.3. Accessing Pointers which Connect Nodes

The parse tree nodes can be thought of as being stored as an array of nodes from 1 to n. Each node has an integer assigned to it which is used to reference it. This index is stored in and used by other nodes as well. These routines are presented below, with associated comments.

```
function Nf                     (* get next node on frontier of tree *)
  (ft: filerange; n: nodeindex): nodeindex; external;

function Nleftson               (* get leftmost child of non-term node *)
  (ft: filerange; n: nodeindex): nodeindex; external;

function Nlthread               (* get node beneath this one on "stack" *)
  (ft: filerange; n: nodeindex): nodeindex; external;
```

```
function Np                    (* get previous node on frontier of tree *)
   (ft: filerange; n: nodeindex): nodeindex; external;

function Nparent               (* get parent node *)
   (ft: filerange; n: nodeindex): nodeindex; external;

function Nrdescend             (* get rightmost terminal in this tree *)
   (ft: filerange; n: nodeindex): nodeindex; external;

function Nsibling              (* get right sibling *)
   (ft: filerange; n: nodeindex): nodeindex; external;
```

5.4. Accessing Fields Common to All Nodes

The following routines retrieve other information stored in each parse tree node. These fields are described in more detail in the parse tree description section earlier in this paper.

```
function Ndepth                (* get depth of node into tree *)
   (ft: filerange; n: nodeindex): integer; external;

function Nenum                 (* get ordering stamp of node *)
   (ft: filerange; n: nodeindex): integer; external;

function Nflagtest             (* test flag setting *)
   (ft: filerange; n: nodeindex; thisflag: short): boolean; external;

function Nnodetype             (* get type of parse tree node *)
   (ft: filerange; n: nodeindex): treenodetype; external;

function Npstate               (* get state of parser stored in this node *)
   (ft: filerange; n: nodeindex): staterange; external;

function Nskipcol              (* get skip column count for printing *)
   (ft: filerange; n: nodeindex): integer; external;
```

```
function Nskipline              (* get skip line count for printing *)
   (ft: filerange; n: nodeindex): integer; external;
```

5.5. Accessing Fields Specific to a Node Type

5.5.1. Fields Present in Terminal Nodes Only

Procedure *Nname* retrieves the print name of a terminal node. Both the parse tree file and string table file tags must be supplied to the routine. Calls to *Nskipline* and *Nskipcol* should also be made to retrieve the number of newlines and spaces to print before the token name, if these are needed.

```
procedure Nname                 (* get print name of token from string table *)
   (ftp, fts: filerange;        (* parse tree and string table file tags *)
    n: nodeindex;               (* node of interest *)
    var buf: charbuf;           (* return: print name of token *)
    var length: cbufindex       (* return: length of print name *)
   ); external;


function Ntoken                 (* get the token code of the terminal node *)
   (ft: filerange; n: nodeindex): tokenrange; external;


function Nvallength             (* get the length of the print name *)
   (ft: filerange; n: nodeindex): integer; external;
```

5.5.2. Fields Present in Non–terminal Nodes Only

The following routines are only meaningful when applied to non–terminal nodes. Note that a third routine *Nleftson*, mentioned earlier, is also only applicable to non–terminal nodes.

```
function Nntoken                (* get token code of non–terminal node *)
   (ft: filerange; n: nodeindex): tokenrange; external;


function Nrule                  (* get rule # of non–term node *)
   (ft: filerange; n: nodeindex): rulerange; external;
```

5.5.3. Fields Present in Marker Nodes Only

If the parser encounters an incorrect token during the parse, the parse will be suspended, a marker token inserted in the tree at this point, and the *badtoken* field of the marker set to point to this incorrect token. If, however, the parse is simply suspended (via a partial parse command in the editor for example), a marker will be inserted into the frontier of the parse tree at the point of the suspension, but no node will be assigned to the *badtoken* field.

```
function Nbadtoken          (* get offending node of marker node *)
  (ft: filerange; n: nodeindex): nodeindex; external;
```

Marker tokens are linked together in a list. The header record of the parse tree contains a pointer to the first marker token, and then each marker token contains a pointer to the next one. The *Nmarksynlist* routine is used to retrieve this pointer from a marker node.

```
function Nmarksynlist       (* get next error pointer in marker node *)
  (ft: filerange; n: nodeindex): nodeindex; external;
```

When a parse is suspended, the node on the top of the parse stack must be noted for later resumption of the parse. This node is stored in the *oldstacktop* field of the marker node, and can be retrieved by the *Noldstacktop* routine.

```
function Noldstacktop       (* get stack top stored in marker node *)
  (ft: filerange; n: nodeindex): nodeindex; external;
```

5.6. Modifying Selected Parse Node Fields

Presently, only the parse tree flags and format fields for printing of the nodes can be rewritten. Only those flags not maintained by the parser should be changed, or havoc will result. See the discussion of parse tree flags presented earlier in this paper for specific flag names.

The *skipline* and *skipcol* fields of the parse tree are used by the display manager to format the tree for printing. These may be reset to any appropriate non-negative value. A filter process to pretty print the parse tree would use these fields to reformat the tree. Note that both non-terminal and terminal nodes have these format fields, but only the nodes along the frontier of the tree have their formats read by the

display manager. Thus the format fields in internal nodes can be used to store formats as inherited attributes of the parse tree nodes. Coding a program in this manner could simplify the bookkeeping which would otherwise be necessary.

```
procedure Newflagclear        (* clear flag *)
   (ft: filerange; n: nodeindex; thisflag: short); external;


procedure Newflagset          (* set flag *)
   (ft: filerange; n: nodeindex; thisflag: short); external;


procedure Newskipline         (* set newline count to print before name *)
   (ft: filerange; n: nodeindex; value: integer); external;


procedure Newskipcol          (* set space count to print before name *)
   (ft: filerange; n: nodeindex; value: integer); external;
```

5.7. Accessing Other Specialized Data: An Array of Shorts

This next section presents one other type of record array which is predefined along with the parse tree node: an array from 1 to n of short integers (two bytes of storage per number). The following routine retrieves these shorts.

```
function Nptshort             (* get ptshort field *)
   (ft: filerange; st: integer): short; external;
```

6. Executing a Filter Process: Command Line Arguments

The SAGA editor contains a *filter* command which takes the name of the filter process as an argument, and arranges to execute the program as a sub–process to the editor. This command automatically supplies the name of the parse tree directory as the first argument to the program, and optionally supplies a parse tree node number as a second argument if a sub–tree is selected by the user to be passed to the filter command. Any other arguments given to the filter command are passed along to the filter process after these initial arguments. Thus the filter process is executed with the following arguments:

<filtername> <parse-tree-directory> [<tree-node>] [<args to filter cmd>]

When the filter process begins execution, it should first pass its first argument to *Ninitialize* to open the parse tree and string table files. If the optional second argument is present, it should be used as the starting node in the tree to be processed. If it is absent, a call to *Nroot* will return the root node of the parse tree, which should be used instead.

Unless the process is prepared to deal with discontinuities in the parse tree, it is a good idea to call *Nstatus*, *Nsynerror* and/or *Nsemerror* to determine whether any exist. If this is the case, the process may wish to simply produce an error message and exit.

If a sub-tree has been specified to the filter process and discontinuities exist in the parse tree, it is possible to determine whether any exist within the subtree of interest. One approach is to traverse the frontier of the subtree, checking for the presence of a marker node or a node with the FNOTPARSED flag set. Alternatively, the *Ntreesynlist* and *Nmarksynlist* routines can be called to retrieve the first and successive marker tokens in the tree, respectively. The *Nenum* routine could check the enumeration field of each of these marker nodes or the *Nbadtoken* node associated with the marker to see whether it is in between the enumeration fields of the first and last terminal nodes in the sub-tree of interest. If none are found, the processing can go ahead.

7. Traversing the Parse Tree

Once the files are opened and the tree status determined, the *Nleftson* and *Nsibling* routines can be used to perform a pre-order, in-order, or post-order walk of the parse tree. Alternatively, starting at the first terminal node in the tree, the *Nsibling* and *Nparent* routines can be used to walk the tree in the same order as the canonical parse which constructed it. Starting at the first terminal node, the *Nf* routine could also be used to walk the frontier of the tree.

At each node in the tree, the appropriate library routine can be used to retrieve the fields of interest in the node.

Should it be desired to make modifications to the tree, two approaches may be used. Fields such as the skipline and skipcol fields can be queried and reset directly using the *Newskipline* and *Newskipcol* routines. To transform the tree, a text file should be created into which the new text to be inserted into the tree is placed. If the *filter* command in the editor is placed into a user–defined command sequence, then additional commands in this sequence can cause the deletion of the sub–tree which was passed to the filter followed by the insertion of the new text from this file.

For more complex modifications, the filter process can created a command file which contains a combination of editor commands and input data. The user–defined command sequence which executes the filter command can then invoke the editor's *exec* command on the file produced by the filter process; commands in this file will then guide the modifications to be made.

Note that if the filter process plans to modify the parse tree in any way, the filter command in the editor should be given as *filter -w ....* The –w option tells the editor to close the parse files and re–open and re–read their contents once the filter process has completed. Normally, all data in memory is written to disk before the filter process is invoked, but a copy is kept in memory for efficiency, and is reused when the editor continues execution.

8. Summary

This document has described the implementation of filter processes. Constructive comments, questions, and feedback concerning unclear or incomplete sections should be be directed to the author.