Syntax™

USER COMMANDS
AND
C LIBRARY FUNCTIONS

Pierre Boullier Philippe Deschamp
INRIA–Rocquencourt
BP 105
78153 Le Chesnay Cedex
France

Release 6
Edited 11 mai 2023.

NAME

bnf – process syntactic grammars, ignoring semantics.

SYNOPSIS

bnf [**–v**] [**–nv**] [**–sc**] [**–nsc**] [**–ls**] [**–nls**] [**–rhs** *nnn*] [**–ln** *name*] . . . [*filenames*]

DESCRIPTION

bnf processes the syntactic part of a language, described for the **SYNTAX**® system (the semantic description is skipped), into a file which will be used by other **SYNTAX** modules. *bnf* checks that the grammar is proper.

Each file named *filename* contains the (syntactic and semantic) description of a given language. A *filename* is of the form [*path/*] *name* [*.suffixes*] where *name* is the name of the described language and *suffixes* is the kind of semantics used to describe the language. Traditionnally, if the semantics is described by **actions** or if there is no semantics *suffixes*= **bnf**, if the semantics is described by **abstract tree** *suffixes*= **at**, if the semantics is described by **synthesised attributes** *suffixes*= **semc**, and for a **pretty-printer** *suffixes*= **paradis**.

If no *filenames* argument is present, the standard input is read.

If no argument is given, *bnf* prints a short synopsis of its usage and exits with a non null status code.

OPTIONS

Options may appear in any order as long as they appear **before** the *filenames*. Only the **last** occurrence of a given option is taken into account.
Default options are such that the command

```
bnf [filenames]
```

is equivalent to

```
bnf -v -sc -ls -rhs 10 [filenames]
```

–v, **–verbose**
Animate the user's screen by displaying cryptic information about what is going on. (Default)

–nv, **–noverbose**
Execute silently.

–sc, **–source**
Produce a file named *name***.bn.l**, containing a line and production numbered source listing together with possible error messages. (Default)

–nsc, **–nosource**
Suppress the source listing ; implies –*nolist*.

–ls, **–list**
Add to the listing file a **cross reference** table of the grammar symbols and the **follow** matrix of terminal symbols ; implies –*source*. (Default)

–nls, **–nolist**
Suppress the cross reference table and the follow matrix.

–rhs *nnn*, **-max_right_hand_side** *nnn*
Issue a warning message each time the length of the right hand side of a production exceeds *nnn* symbols. The default value of *nnn* is 10.

–ln *name*, **–language_name** *name*
Force the *name* of the language to process. The **–ln** option is mandatory if no *filename* is given.

FILES

*name***.bt**	internal grammar form (output)
*name***.bn.l**	listing (output)

SEE ALSO

semact(1), semat(1), paradis(1), semc(1) and the **"SYNTAX REFERENCE MANUAL"**.

DIAGNOSTICS

When the grammar is not proper, an error message is issued and error diagnostics are gathered in the listing (if any).

Exit status is 0 if everything is alright, 1 if only warnings are issued, 2 if error messages are issued, 3 for command line syntax errors.

NAME

csynt – LALR (1) constructor and optimiser

SYNOPSIS

csynt [**–v**] [**–nv**] [**–fc**] [**–nfc**] [**–a**] [**–na**] [**–luc**] [**–nluc**] [**–lsc**] [**–nlsc**] [**–lc**] [**–nlc**] [**–p**] [**–np**] [**–lr1**] [**–nlr1**] [**–ab**] [**–nab**] [**–fe**] [**–nfe**] [**–ll**] [**–nll**] [**–tspe**] [**–pspe**] [**–nspe**] . . . *language* . . .

DESCRIPTION

For each *language* argument, *csynt* builds an optimised LALR (1) or LR (1) (see the **–lr1** option) push-down automaton.

The LALR (1) or LR (1) conflicts (if any) are solved in the following way:
- if there exists disambiguating rules (specified via **prio**(1)) and if they are valid for the current conflict, they are used ;
- else the built-in disambiguating rules are used.

A single parsing action (shift or reduce) is chosen anyway, and the resulting automaton is deterministic.

If no argument is given, *csynt* prints a short synopsis of its usage and exits with a non null status code.

OPTIONS

Options may appear in any order as long as they appear **before** the *language* names. Only the **last** occurrence of a given option is taken into account.
Default options are such that the command
csynt language
is equivalent to
csynt –v –fc –nlr1 –na –lsc –nab –nfe –np language

–v, **–verbose**
Animate the user's screen by displaying cryptic information about what is going on. (Default)

–nv, **–noverbose**
Execute silently.

–fc, **–force**
Force the optimisation phase, even when LALR (1) or LR (1) conflicts have been solved using system rules. (Default)

–nfc, **–noforce**
Stop the execution before the optimisation phase when LALR (1) or LR (1) conflicts have been solved using system rules: the push-down automaton is not produced.

–lr1
If the grammar is not LALR (1), determine for each conflictual state whether this conflict could be resolved by a LR (1) constructor; if so, the conflictual path is split and we obtain a (partial) conflict-free non canonical LR (1) automaton.

–nlr1, **–nolr1**
Do not perform the non canonical LR (1) automaton construction. (Default)

–a, **–automaton**
Add to the listing file, named *language***.la.l**, the LR (0) automaton and the LALR (1) look ahead sets of the reduce items involved in non-LR (0) states. If the **–lr1** option is set, the resulting non canonical LR (1) automaton is printed.

–na, **–noautomaton**
Do not print the resulting automaton. (Default)

–luc, **–list_user_conflicts**
Add to the listing file, named *language***.la.l**, the LALR (1) or LR (1) conflicts which have been solved using the user's disambiguating rules given via a **prio**(1) specification.

−nluc, **−nolist_user_conflicts**
Suppress the report of LALR (1) or LR (1) conflicts solved using the user's disambiguating rules. (Default)

−lsc, **−list_system_conflicts**
Add to the listing file, named *language***.la.l**, the LALR (1) or LR (1) conflicts which have been solved using the system disambiguating rules. (Default)

−nlsc, **−nolist_system_conflicts**
Suppress the report of LALR (1) or LR (1) conflicts solved using the system disambiguating rules.

−lc, **−list_conflicts**
Add to the listing file, named *language***.la.l**, all the LALR (1) or LR (1) conflicts.

−nlc, **−nolist_conflicts**
Suppress the report of LALR (1) or LR (1) conflicts.

−p, **−path**
Add to the listing file, named *language***.la.l**, for each state *s* in which a conflict is detected, a sample path through the automaton from the initial state until *s*. Also print a rightmost derivation showing the propagation of the offending terminal from the point where an occurrence is spontaneously generated, until the conflictual state. If the conflict is also an LR (1) conflict, a message is printed in terms of the theoretical LR (1) definition. Furthermore, some case of ambiguity are detected, and the corresponding message shows two different rightmost derivations leading to the same sentential form.

−np, **−nopath**
Suppress the corresponding report. (Default)

−ab, **−abstract**
Add to the listing file, named *language***.op.l**, statistics about the optimisation phase.

−nab, **−noabstract**
Suppress the corresponding report. (Default)

−fe, **−floyd_evans**
Add to the listing file, named *language***.op.l**, the optimised push-down automaton coded with Floyd-Evans Productions ; implies **−ab**.

−nfe, **−nofloyd_evans**
Suppress the Floyd_Evans Productions listing. (Default)

−ll, **−long_listing**
Add to the listing file, named *language***.la.l**, the non optimised push-down automaton coded with Floyd-Evans Productions. Add to the listing file, named *language***.op.l**, the tables coding the optimised push-down automaton ; implies **−ab**b. This option is mainly used for debugging purposes.

−nll, **−nolong_listing**
Suppress the previous listings. (Default)

−pspe, **−partial_single_productions_elimination**
Perform only a partial elimination of single productions ; implies options **−tspe** and **−nspe** are reset. (Default)

−tspe, **−total_single_productions_elimination**
Perform a total elimination of single productions ; implies options **−pspe** and **−nspe** are reset. May be used when analysis speed is at a premium.

−nspe, **−no_single_productions_elimination**
Do not perform any elimination of single productions ; implies options **−pspe** and **−tspe** are reset. Mainly used for measuring purposes.

FILES

language.**bt**	internal grammar form (input)
language.**dt**	user's disambiguating table rules (input)
language.**pt**	parser tables (output)
language.**la.l**	LALR (1) constructor listing (output)
language.**op.l**	optimisation results listing (output)

SEE ALSO

bnf(1), semact(1), semat(1), paradis(1), semc(1), prio(1), tables_c(1) and the **"SYNTAX REFERENCE MANUAL"**.

DIAGNOSTICS

The diagnostics are intended to be self-explanatory to a user familiar with LR theory.
Exit status is 0 if everything is alright, 1 if only warnings are issued, 2 if error messages are issued, 3 for command line syntax errors.

NAME

lecl – process lexical specifications.

SYNOPSIS

lecl [**–v**] [**–nv**] [**–ot**] [**–not**] [**–sc**] [**–nsc**] [**–tb**] [**–ntb**] [**–ob**] [**–nob**] [**–ls**] [**–nls**] [**–hl** *nnn*] [**–sks** *nnn*] [**–ln** *name*] . . . [*filenames*]

DESCRIPTION

lecl translates the lexical specification part of a language, described for the **SYNTAX**® system, into a file mainly coding a deterministic finite state automaton which will be used by other **SYNTAX** modules.

Each file named *filename* contains the lexical description of a given language. A *filename* may be of the form *name*. **lecl** where *name* is the name of the described language.

If no *filenames* argument is present, the standard input is read.

If no argument is given, *lecl* prints a short synopsis of its usage and exits with a non null status code.

OPTIONS

Options may appear in any order as long as they appear **before** the *filenames*. Only the **last** occurrence of a given option is taken into account.
Default options are such that the command

lecl [filenames]

is equivalent to

lecl -v -ot -sc -ob -hl |{keywords}| -sks 037777
[filenames]

–v, **–verbose**
Animate the user's screen by displaying cryptic information about what is going on. (Default)

–nv, **–noverbose**
Execute silently.

–ot Optimise the automaton in the sense that, whenever possible, a lexical token is recognised without looking at the next input character. (Default)

–not Use at least one character of look ahead for each lexical token recognition.

–sc, **–source**
Produce a file named *name***.lc.l**, containing a line numbered source listing together with possible error messages. (Default)

–nsc, **–nosource**
Suppress the source listing ; implies –*nolist,* –*notable*and –*noobject*.

–tb, **–table**
Add to the listing file the **symbol table** (this is mainly a debugging option) ; implies –*source*.

–ntb, **–notable**
Suppress the **symbol table** listing. (Default)

–ob, **–object**
Add to the listing file the **generated code** and the **perfect hashing function** ; implies –*source*. (Default)

–nob, **–noobject**
Suppress the **generated code** and the **perfect hashing function** listing .

–ls, **–list**
Add to the listing file the **symbol table**, the **finite state automaton**, the **generated code** and the **perfect hashing function** ; implies –*source*.

–nls, **–nolist**
Suppress the **symbol table**, the **finite state automaton**, the **generated code** and the **perfect hashing function** ; (Default)

−hl *nnn*, **-hash_length** *nnn*

Build, when relevent, a perfect hashing function whose buckets number is close to *nnn*. The default value of *nnn* is the number of keywords in *name*.

−sks *nnn*, **-scramble_kind_set** *nnn*

Try the strategies specified by *nnn* to compute a primary hash_code function (called scramble) from character string to integer. 14 strategies are currently implemented and are tried one after the other until one succeeds. The strategy number *n* is tried iff no strategy less than *n* succeeds and the bit number *n* of *nnn* is set. Bits are counted from right to left starting at 1. The default value of *nnn* is 037777.

−ln *name*, **−language_name** *name*

Force the *name* of the language to process. The **−ln** option is mandatory if no *filename* is given.

FILES

*name***.bt**	internal grammar form (input)
*name***.st**	scanner tables (output)
*name***.lc.l**	listing (output)
/tmp/sx*	scratch files

SEE ALSO

bnf(1), semact(1), semat(1), paradis(1), semc(1), st_to_c(1), and the **"SYNTAX REFERENCE MANUAL"**.

DIAGNOSTICS

When the specification is erroneous, error and warning messages are issued and diagnostics are gathered in the listing (if any).
Exit status is 0 if everything is alright, 1 if only warnings are issued, 2 if error messages are issued, 3 for command line syntax errors.

NAME

paradis – process syntactic grammars with pretty printer specification.

SYNOPSIS

paradis [**–v**] [**–nv**] [**–sc**] [**–nsc**] [**–ls**] [**–nls**] [**–rhs** *nnn*] [**–ln** *name*] . . . [*filenames*]

DESCRIPTION

paradis processes the syntactic and pretty printer specification parts of a language, described for the **SYNTAX**® system, into a file which will be used by other **SYNTAX** modules. *paradis* checks that the grammar is proper and translates the pretty printer specification into tables.

Each file named *filename* contains the (syntactic and semantic) description of a given language. A *filename* may be of the form [*path/*] *name*.**paradis** where *name* is the name of the described language.

If no *filenames* argument is present, the standard input is read.

If no argument is given, *paradis* prints a short synopsis of its usage and exits with a non null status code.

OPTIONS

Options may appear in any order as long as they appear **before** the *filenames*. Only the **last** occurrence of a given option is taken into account.
Default options are such that the command

paradis [filenames]

is equivalent to

paradis -v -sc -ls -rhs 10 [filenames]

–v, **–verbose**
Animate the user's screen by displaying crypting information about what is going on. (Default)

–nv, **–noverbose**
Execute silently.

–sc, **–source**
Produce a file named *name***.bn.l**, containing a line and production numbered source listing together with possible error messages. (Default)

–nsc, **–nosource**
Suppress the source listing ; implies *–nolist*.

–ls, **–list**
Add to the listing file a **cross reference** table of the grammar symbols, the **follow** matrix of terminal symbols and the **generated code** corresponding to the pretty printer specification ; implies *–source*. (Default)

–nls, **–nolist**
Suppress the cross reference table, the follow matrix and the **generated code** corresponding to the pretty printer specification .

–rhs *nnn*, **-max_right_hand_side** *nnn*
Issue a warning message each time the length of the right hand side of a production exceeds *nnn* symbols. The default value of *nnn* is 10.

–ln *name*, **–language_name** *name*
Force the *name* of the language to process. The **–ln** option is mandatory if no *filename* is given.

FILES

*name***.bt**	internal grammar form (output)
*name***.ppt**	internal pretty printer tables (output)
*name***.bn.l**	listing (output)
/tmp/sx*	scratch files

SEE ALSO

bnf(1), semact(1), semat(1), semc(1) and the **"SYNTAX REFERENCE MANUAL"**.

DIAGNOSTICS

When the grammar is not proper or when the specification of the pretty printer is incorrect, error messages are issued and error diagnostics are gathered in the listing (if any).
Exit status is 0 if everything is alright, 1 if only warnings are issued, 2 if error messages are issued, 3 for command line syntax errors.

NAME

prio – process the disambiguating rules associated with a syntactic grammar.

SYNOPSIS

prio [**–v**] [**–verbose**] [**–nv**] [**–noverbose**] [**–sc**] [**–source**] [**–nsc**] [**–nosource**] [**–ln** *name*] [**–language_name** *name*] [**–listing**] [**–nolisting**] . . . [*filenames*]

DESCRIPTION

prio processes the disambiguating rules associated with a syntactic grammar, written for the **SYNTAX®** system. *prio* records for each specified terminal symbol and for each specified grammar rule a precedence and an associativity which will be used later on by csynt(1) in order to help resolution of **LALR**(**1**) conflicts.

Each file named *filename* contains the description of the disambiguating rules. A *filename* may be of the form [*path/*] *name*.**prio** where *name* is the name of the processed language.

If no *filenames* argument is present, the standard input is read.

If no argument is specified, *prio* prints a short synopsis of its usage and exits with a non null status code.

OPTIONS

Options may appear in any order as long as they appear **before** the *filenames*. Only the **last** occurrence of a given option is taken into account.
Default options are such that the command
  prio [filenames]
is equivalent to
  prio -v -sc [filenames]

–v, **–verbose**
Animate the user's screen by displaying cryptic information about what is going on. (Default)

–nv, **–noverbose**
Execute silently.

–sc, **–source**
Produce a file named *name***.pr.l**, containing a line numbered source listing together with possible error messages. (Default)

–nsc, **–nosource**
Suppress the source listing .

–ln *name*, **–language_name** *name*
Force the *name* of the language to process. The **–ln** option is mandatory if no *filename* is given.

–listing
Display (using a human-readable form) in file *name***.pr.l** the contents of the binary tables generated in file *name***.dt**. This option is helpful for debugging and has only an effect if option **–source** is set.

–nolisting
Do not display in file *name***.pr.l** the contents of the tables. (Default)

FILES

*name***.bt**	internal grammar form (input)
*name***.dt**	internal disambiguating tables (output)
*name***.pr.l**	source listing (output)

SEE ALSO

bnf(1), semact (1), semat(1), paradis(1), semc(1), csynt(1) and the **"SYNTAX REFERENCE MANUAL"**.

DIAGNOSTICS

When the specification is incorrect an error message is issued and error diagnostics are gathered in the listing (if any).
Exit status is 0 if everything is alright, 1 if only warnings are issued, 2 if error messages are issued, 3 for

command line syntax errors.

NAME

recor – process error recovery specifications for SYNTAX.

SYNOPSIS

recor [**–v**] [**–nv**] [**–sc**] [**–nsc**] [**–ln** *name*] . . . [*filenames*]

DESCRIPTION

recor translates the lexical and syntactic error recovery specification part of a language, described for the **SYNTAX**® system, into tables which will be used by other **SYNTAX** modules.

Each file named *filename* contains the error recovery description for a given language. A *filename* may be of the form [*path/*] *name*.**recor** where *name* is the name of the described language.

If no *filenames* argument is present, the standard input is read.

If no argument is given, *recor* prints a short synopsis of its usage and exits with a non null status code.

OPTIONS

Options may appear in any order as long as they appear **before** the *filenames*. Only the **last** occurrence of a given option is taken into account.
Default options are such that the command
recor [filenames]
is equivalent to
recor -v -sc [filenames]

–v, **–verbose**
Animate the user's screen by displaying crypting information about what is going on. (Default)

–nv, **–noverbose**
Execute silently.

–sc, **–source**
Produce a file named *name*.**rc.l**, containing a line numbered source listing together with possible error messages. (Default)

–nsc, **–nosource**
Suppress the source listing .

–ln *name*, **–language_name** *name*
Force the *name* of the language to process. The **–ln** option is mandatory if no *filename* is given.

FILES

name.**bt**	internal grammar form (input)
name.**st**	scanner tables (input)
name.**rt**	error recovery tables (output)
name.**rc.l**	listing (output)

SEE ALSO

bnf(1), semact(1), semat(1), paradis(1), semc(1), lecl(1), tables_c(1), and the **"SYNTAX REFERENCE MANUAL"**.

DIAGNOSTICS

When the specification is erroneous, error and warning messages are issued and diagnostics are gathered in the listing (if any).
Exit status is 0 if everything is alright, 1 if only warnings are issued, 2 if error messages are issued, 3 for command line syntax errors.

NAME

semact – process syntactic grammars with semantics described by actions.

SYNOPSIS

semact [**–v**] [**–nv**] [**–sc**] [**–nsc**] [**–ls**] [**–nls**] [**–rhs** *nnn*] [**–ln** *name*] . . . [*filenames*]

DESCRIPTION

semact processes the syntactic part of a language described for the **SYNTAX®** system with semantics described by actions into a file which will be used by other **SYNTAX** modules. *semact* checks that the grammar is proper and records, for each grammar rule, the action number specified in the source file.

Each file named *filename* contains the (syntactic and semantic) description of a given language. A *filename* may be of the form [*path/*] *name*.**bnf** where *name* is the name of the described language.

If no *filenames* argument is present, the standard input is read.

If no argument is given, *semact* prints a short synopsis of its usage and exits with a non null status code.

OPTIONS

Options may appear in any order as long as they appear **before** the *filenames*. Only the **last** occurrence of a given option is taken into account.
Default options are such that the command

semact [filenames]

is equivalent to

semact -v -sc -ls -rhs 10 [filenames]

–v, **–verbose**
Animate the user's screen by displaying cryptic information about what is going on. (Default)

–nv, **–noverbose**
Execute silently.

–sc, **–source**
Produce a file named *name***.bn.l**, containing a line and production numbered source listing together with possible error messages. (Default)

–nsc, **–nosource**
Suppress the source listing ; implies –*nolist*.

–ls, **–list**
Add to the listing file a **cross reference** table of the grammar symbols, the **follow** matrix of terminal symbols and a table of non null **actions** ; implies –*source*. (Default)

–nls, **–nolist**
Suppress the cross reference table, the follow matrix and the action table.

–rhs *nnn*, **-max_right_hand_side** *nnn*
Issue a warning message each time the length of the right hand side of a production exceeds *nnn* symbols. The default value of *nnn* is 10.

–ln *name*, **–language_name** *name*
Force the *name* of the language to process. The **–ln** option is mandatory if no *filename* is given.

FILES

*name***.bt**	internal grammar form (output)
*name***.bn.l**	listing (output)
/tmp/sx*	scratch files

SEE ALSO

bnf(1), semat(1), paradis(1), semc(1) and the **"SYNTAX REFERENCE MANUAL"**.

DIAGNOSTICS

When the grammar is not proper or when the specification of an action is incorrect, an error message is issued and error diagnostics are gathered in the listing (if any).
Exit status is 0 if everything is alright, 1 if only warnings are issued, 2 if error messages are issued, 3 for

command line syntax errors.

NAME

semat – process syntactic grammars with semantics described by abstract tree.

SYNOPSIS

semat [**–v**] [**–nv**] [**–sc**] [**–nsc**] [**–ls**] [**–nls**] [**–rhs** *nnn*] [**–c**] [**–pascal**] [**–ll** *nnn*] [**–ln** *name*] . . . [*filenames*]

DESCRIPTION

semat processes the syntactic part of a language, described for the **SYNTAX**® system with semantics described by abstract tree. *semat* checks that the grammar is proper and translates the abstract tree specification into tables which will be used by other **SYNTAX** modules, and into a semantic pass program skeleton, written in *C* or *pascal*, which must be completed with the user's attributes computations.

Each file named *filename* contains the (syntactic and semantic) description of a given language. A *filename* may be of the form [*path/*] *name*.**at** where *name* is the name of the described language.

If no *filenames* argument is present, the standard input is read.

The semantic pass program skeleton is written on the standard output.

If no argument is given, *semat* prints a short synopsis of its usage and exits with a non null status code.

OPTIONS

Options may appear in any order as long as they appear **before** the *filenames*. Only the **last** occurrence of a given option is taken into account.
Default options are such that the command
semat [filenames]
is equivalent to
semat -v -sc -ls -rhs 10 -ll 128 [filenames]

–v, **–verbose**
Animate the user's screen by displaying cryptic information about what is going on. (Default)

–nv, **–noverbose**
Execute silently.

–sc, **–source**
Produce a file named *name***.bn.l**, containing a line and production numbered source listing together with possible error messages. (Default)

–nsc, **–nosource**
Suppress the source listing ; implies –*nolist*.

–ls, **–list**
Add to the listing file a **cross reference** table of the grammar symbols, the **follow** matrix of terminal symbols and a table which gives for each production the name of the abstract tree node and the names of its sons ; implies –*source*. (Default)

–nls, **–nolist**
Suppress the cross reference table, the follow matrix and the printing of the nodes hierarchy.

–rhs *nnn*, **-max_right_hand_side** *nnn*
Issue a warning message each time the length of the right hand side of a production exceeds *nnn* symbols. The default value of *nnn* is 10.

–c Indicate that the semantic pass program skeleton is written in *C*. This option is exclusive with the **–pascal** option. (Default)

–pascal
Indicate that the semantic pass program skeleton is written in *pascal*. This option is exclusive with the **–c** option.

–ll *nnn*, **-max_line_length** *nnn*
Try to keep the comments generated in the semantic pass program skeleton into lines whose length does not exceed *nnn* columns. The default value of *nnn* is 128.

−ln *name*, **−language_name** *name*

Force the *name* of the language to process. The **−ln** option is mandatory if no *filename* is given.

FILES

name.**at**	grammar and abstract tree specification (input)
name.**bt**	internal grammar form (output)
name.**att**	internal abstract tree tables (output)
name.**bn.l**	listing (output)

SEE ALSO

bnf(1), semact(1), paradis(1), semc(1) and the **"SYNTAX REFERENCE MANUAL"**.

DIAGNOSTICS

When the grammar is not proper or when the specification of the abstract tree is incorrect, an error message is issued and error diagnostics are gathered in the listing (if any).
Exit status is 0 if everything is alright, 1 if only warnings are issued, 2 if error messages are issued, 3 for command line syntax errors.

®" :1: Command not found. semc – process syntactic grammars with semantics described by synthetized attributes written in C.

SYNOPSIS

semc [**–v**] [**–nv**] [**–sc**] [**–nsc**] [**–ls**] [**–nls**] [**–so**] [**–nso**] [**–df**] [**–ndf**] [**–rhs** *nnn*] [**–ln** *name*] . . . [*filenames*]

DESCRIPTION

semc processes the syntactic part of a language, described for the **SYNTAX**® system with semantics described by synthetized attributes written in C. *semc* checks that the grammar is proper and translates the attributes specification into a semantic evaluator written in C.

Each file named *filename* contains the (syntactic and semantic) description of a given language. A *filename* may be of the form [*path/*] *name*.**semc** where *name* is the name of the described language.

If no *filenames* argument is present, the standard input is read.

The semantic evaluator is written on the standard output.

If no argument is given, *semc* prints a short synopsis of its usage and exits with a non null status code.

OPTIONS

Options may appear in any order as long as they appear **before** the *filenames*. Only the **last** occurrence of a given option is taken into account.
Default options are such that the command
  semc [filenames]
is equivalent to
  semc -v -sc -ls -rhs 10 -nso -df [filenames]

–v, **–verbose**
Animate the user's screen by displaying cryptic information about what is going on. (Default)

–nv, **–noverbose**
Execute silently.

–sc, **–source**
Produce a file named *name***.bn.l**, containing a line and production numbered source listing together with possible error messages. (Default)

–nsc, **–nosource**
Suppress the source listing ; implies *–nolist*.

–ls, **–list**
Add to the listing file a **cross reference** table of the grammar symbols, the **follow** matrix of terminal symbols and a table which gives for each production the name of the abstract tree node and the names of its sons ; implies *–source*. (Default)

–nls, **–nolist**
Suppress the cross reference table, the follow matrix and the printing of the nodes hierarchy.

–rhs *nnn*, **-max_right_hand_side** *nnn*
Issue a warning message each time the length of the right hand side of a production exceeds *nnn* symbols. The default value of *nnn* is 10.

–ln *name*, **–language_name** *name*
Force the *name* of the language to be processed. The **–ln** option is mandatory if no *filename* is given.

–df, **–default**
Produce in the source listing (if any) the attributes definition that *semc* deduces by default. (Default)

–ndf, **–nodefault**
Remain silent on the attributes definition deduced by default.

−nso, **−nosem_out**
Remain silent on the attribute grammar. (Default)

−so, **−sem_out**
Produce in the source listing (if any) some information about the attribute grammar : what attributes apply on each non-terminal symbol, etc...

FILES

name.**bt**	internal grammar form (output)
name.**bn.l**	listing (output)
/tmp/sx*	scratch files

SEE ALSO

bnf(1), semact(1), paradis(1), semat(1) and the **"SYNTAX REFERENCE MANUAL"**.

DIAGNOSTICS

When the grammar is not proper or when the specification of the abstract tree is incorrect, an error message is issued and error diagnostics are gathered in the listing (if any).
Exit status is 0 if everything is alright, 1 if only warnings are issued, 2 if error messages are issued, 3 for command line syntax errors.

NAME

st_to_c – generate a scanner in C.

SYNOPSIS

st_to_c *language*

DESCRIPTION

st_to_c reads the file *language***.st**, built by a previous execution of **lecl**(1) and produces, on its standard output, the corresponding scanner, written in C, which can replace the standard interpreter (i.e. the **SYNTAX** scanner, see sxscanner(3)).

If no argument is given, *st_to_c* prints a short synopsis of its usage and exits with a non null status code.

FILES

*language***.st**	scanner tables (input)

SEE ALSO

lecl(1) and the **"SYNTAX REFERENCE MANUAL"**.

NAME

tdef – name the internal values of terminal symbols.

SYNOPSIS

tdef [**–v**] [**–nv**] [**–sc**] [**–nsc**] [**–c**] [**–pascal**] [**–ln** *name*] . . . [*filenames*]

DESCRIPTION

tdef produces from its input an include file containing definition of constants to be used wherever pertinent in the user's *C* or *PASCAL* sources.

Each file named *filename* contains the input description for a given language. A *filename* may be of the form [*path/*] *name*.**tdef** where *name* is the name of the described language.

Each input file contains a list of assignments:
lhs = *rhs* ;
where *lhs* is a *C* or *PASCAL* identifier and *rhs* is a terminal symbol of the *name* language.

If no *filenames* argument is present, the standard input is read.

The **#define** macros are written on the standard output.

If no argument is given, *tdef* prints a short synopsis of its usage and exits with a non null status code.

OPTIONS

Options may appear in any order as long as they appear **before** the *filenames*. Only the **last** occurrence of a given option is taken into account.
Default options are such that the command
tdef [filenames]
is equivalent to
tdef -v -sc [filenames]

–v, **–verbose**
Animate the user's screen by displaying cryptic information about what is going on. (Default)

–nv, **–noverbose**
Execute silently.

–sc, **–source**
Produce a file named *name***.td.l**, containing a line numbered source listing together with possible error messages. (Default)

–nsc, **–nosource**
Suppress the source listing .

–c Indicate that the semantic pass program skeleton is written in *C*. This option is exclusive with the **–pascal** option. (Default)

–pascal
Indicate that the semantic pass program skeleton is written in *pascal*. This option is exclusive with the **–c** option.

–ln *name*, **–language_name** *name*
Force the *name* of the language to process. The **–ln** option is mandatory if no *filename* is given.

FILES

*name***.bt**	internal grammar form (input)
*name***.td.l**	source listing (output)

SEE ALSO

bnf(1), semact(1), semat(1), paradis(1), semc(1), and the **"SYNTAX REFERENCE MANUAL"**.

DIAGNOSTICS

When the specification is erroneous, error and warning messages are issued and diagnostics are gathered in the listing (if any).
Exit status is 0 if everything is alright, 1 if only warnings are issued, 2 if error messages are issued, 3 for

command line syntax errors.

NAME

sxsmp, sxbrother, sxson – abstract tree traversor for SYNTAX.

SYNOPSIS

```
#include "sxunix.h"

SXVOID sxsmp (root, pass_inherited, pass_derived)
        SXNODE *root;
        SXVOID  (*pass_inherited) (),
                (*pass_derived) ();

SXNODE *sxbrother (node, n)
        SXNODE *node;
        int     n;

SXNODE *sxson (node, n)
        SXNODE *node;
        int     n;
```

DESCRIPTION

sxsmp is the module which performs the depth-first traversal of the [sub-]tree rooted at *root*. Each node is visited twice, first an inherited visit before its sub-tree traversal and second a synthesised (derived) visit to complete its sub-tree walk. At each inherited visit of a node the procedure *pass_inherited* is called while *pass_derived* is called at each derived visit. This tree traversal stops after the derived visit of *root*.

sxbrother returns (a pointer to) the *n*_th son of the father of *node* or NULL if it does not exist.

sxson returns (a pointer to) the *n*_th son of *node* or NULL if it does not exist.

This module provides also a set of macros and variables which may be used during a semantic pass. These macros and variables are declared in the include file **sxunix.h** and the declarations are close to the following :

```
struct sxtt_state {
   SXNODE       *visited                          /* pointer to the visited node*/;
   SXBOOLEAN  visit_kind                          /* SXINHERITED or SXDERIVED*/;
   SXBOOLEAN  last_elem_or_left                   /* last son or left brother pointer*/;
   } sxtt_state;


#define SXVISIT_KIND       sxtt_state.visit_kind
#define SXVISITED          sxtt_state.visited
#define SXLEFT             SXLAST_ELEM
#define SXLAST_ELEM        sxtt_state.last_elem_or_left
```

The variables in *sxtt_state* are easilly accessed via the above macros. During a tree traversal on a *SXVISIT_KIND* pass, the current node is pointed by *SXVISITED*. Its attributes can be found in the structure **SXVISITED* (see **sxatc** (3). If (and only if) the visit is inherited, *SXLEFT* is (a pointer to) its left brother if any else NULL. If (and only if) the visit is derived *SXLAST_ELEM* is (a pointer to) its rightmost son if any else NULL.

FURTHER DESCRIPTION

```
struct sxnext_visit {
   SXNODE      *visited    /* next node to be visited */;
   SXBOOLEAN  visit_kind /* in this way            */,
              normal     /* if SXFALSE             */;
```

```
    } sxnext_visit;


#define sxat_snv(kind, node_ptr)                    \
                      (sxnext_visit.normal = SXFALSE,   \
                       sxnext_visit.visit_kind = kind,       \
                       sxnext_visit.visited = node_ptr)
```

The depth-first tree traversal strategy may be altered in the following way: the macro *sxat_snv* (*s*pecify *n*ext *v*isit) sets the variables of *sxnext_visit* in such a way that the next node to be visited will be *node_ptr* during a *kind* pass. From *node_ptr* the tree walk restarts in a depth-first manner until the next *sxat_snv* occurrence. The walk stops, as usual, after the derived visit of *root*.

SEE ALSO

sxunix(3), sxscanner(3), sxparser(3), sxatc(3), and the *SYNTAX Reference Manual*.

NAME

sxatc – abstract tree constructor for SYNTAX.

SYNOPSIS

```
#include "sxunix.h"

struct sxtables  sxtables ;

SXNODE ***sxatc_stack ()

SXVOID sxatc (what, sxtables)
        int                  what ;
        struct sxtables *sxtables ;
```

DESCRIPTION

sxatc is the module which performs the construction of an abstract tree (see semat(1)) in parallel with the syntax analysis (see sxparser(3)). This module provides also a set of macros and variables which may be used during a semantic pass. These macros and variables are declared in the include file **sxunix.h** and the declarations are close to the following :

```
#ifndef SXNODE
/* by default SXNODE ==> struct sxnode_header_s */
#define SXNODE struct sxnode_header_s
#endif

/* to have the definition of sxnode_header_s look like normal C */
#define SXVOID_NAME

#define SXNODE_HEADER_S \
    SXNODE       *father            /* pointer to the father */,\
                 *brother           /* pointer to the right brother            */,\
                 *son               /* pointer to the leftmost son             */ ;\
    SXINT         name              /* node name                               */,\
                  degree            /* node arity                              */,\
                  position          /* positionth son of its father            */ ;\
    SXBOOLEAN                        is_list                                   /* is this node a list*/,\
                  first_list_element /* is this node the first element of a list */,\
                  last_list_element  /* is this node the last element of a list  */ ;\
    struct sxtoken token             /* lexical information                     */

struct sxnode_header_s {
    SXNODE_HEADER_S SXVOID_NAME;
    };
```

Each node in an abstract tree is a structure whose leading components must be (macro expanded from) "**SXNODE_HEADER_S** SXVOID_NAME;". For a node *father* is a pointer to its father, *brother* is a pointer to its right brother (or NULL), *son* is a pointer to its first son (or NULL), *name* is (the internal code for) its name, *degree* is its arity i.e. its number of sons (0 for a leaf), *position*, only valid for a non root node, is the rank as son of its father (1<=*position*<=*father*->*degree*). If the node represents a list *is_list* is **SXTRUE**. If the father of a node is a list and if *position* is 1 then *first_list_element* is **SXTRUE**. If its father is a list and if *position* is equal to *father*->*degree* then *last_list_element* is **SXTRUE**. If the node stands for a (generic) terminal symbol, *token* contains its lexical attributes. If the node stands for a non terminal symbol, the only valid fields of *token* are *source_index* and *comment*. *source_index* contains the coordinates of the leftmost terminal of the sentence recognised by this non terminal or the coordinates of its look-ahead terminal symbol if the sentence is empty.

```
struct sxatc_local_variables {
     SXNODE        *father                      /* internal use                   */ ;
     SXNODE        *abstract_tree_root          /* abstract tree root node pointer */ ;
     SXINT          node_size                   /* sizeof(SXNODE) */ ;
     SXATC_AREA *head                           /* internal use                   */;
     SXINT          areaindex                   /* internal use                   */;
     SXBOOLEAN   abstract_tree_is_error_node /* SXTRUE if ERROR node            */;
     SXBOOLEAN   are_comments_erased         /* internal use                   */;
     };
```

abstract_tree_root is (a pointer to) the **root** of the abstract tree. Usually the attribute evaluation starts at inherited visit of its first (leftmost) son. *node_size* is the size of an abstract tree node. *abstract_tree_is_error_node* is set if and only if there is (at least) one error node in the whole abstract tree, that means a syntactic error involving a global recovery had occurred.

```
struct sxatcvar {
     struct T_tables                   T_tables /* abstract tree tables */;
     struct sxatc_local_variables atc_lv     /* local variables        */;
     } sxatcvar;
```

T_tables are the tables which contain the informations about the abstract tree construction. *atc_lv* are the local variables which may be used by the user's semantic pass.

FURTHER DESCRIPTION

sxatc is the main entry of the module to be called for constructing an abstract syntax tree for given source text. Its possible uses are:
sxatc (*SXOPEN*, *sxtables*) opens the tables, initialises (some of) the local variables when a new language is going to be processed and calls the entry point "SXOPEN" of the (user's) semantic pass.
sxatc (*SXINIT*, *sxtables*) called by the parser just before any new source text syntax analysis. It initialises (the rest of) the local variables and, upon its first call allocates its semantic (tree node) stack.
sxatc (*SXACTION*, *action_no*) called by the parser on each reduction. Puts an other brick in the tree.
sxatc (*SXERROR*, *sxtables*) called by the parser each time a global syntax error recovery occurs. It builds an **ERROR** node.
sxatc (*SXFINAL*, *sxtables*) called by the parser just after the analysis of a source text. The variable *sxatcvar.atc_lv.abstract_tree_root* is set and the semantic stack is freed if there is no more *sxatc* call activ.
sxatc (*SXSEMPASS*, *sxtables*) calls the user's semantic pass and frees the abstract tree.
sxatc_stack () allows the user of syntactic actions or predicates to access to the [sub-]tree constructed so far. For example let us consider the grammar rules:

```
     <term>     = <variable> @i ;
     <variable> = %ID          ; "ID"
```

The following portion of C code allows to access the fields of the ID node

```
     case i:
       (*sxatc_stack ()) [SXSTACKtop ()]->...
```

Due to possible reallocations, only the value returned by sxatc_stack (and not the object pointed to) is garanteed not to vary during the abstract tree construction.

SEE ALSO

sxunix(3), sxscanner(3), sxparser(3), sxatc(3), sxat_mngr(3) and the *SYNTAX Reference Manual*.

NOTES

The contents of the structure *sxatcvar* must be saved and restored by the user when switching between different parsers.

NAME

sxatedit, sxateditinit – pretty-printer for abstract trees constructed by SYNTAX.

SYNOPSIS

```
#include "sxunix.h"

SXVOID sxatedit (node)
        SXNODE *node ;

SXVOID sxateditinit (file, md1, md2, pw)
        FILE *file ;
        long   md1, md2, pw ;
```

DESCRIPTION

sxatedit prints on some file the graphical representation of the abstract tree constructed by **SYNTAX** (see **semat**(1) and **sxatc**(3)) whose root node is *node*, using the generic tree pretty-printer **sxpptree**(3).

The output file and some other parameters must have been set beforehand by a call to *sxateditinit*, with parameters as follows:

file a pointer to the output file, which must have been opened for writing using **fopen**(3) ;

md1 the minimum number of spaces between two sibling nodes (see the meaning of *min_dist* in the description of **sxpptree**(3)) ;

md2 the minimum number of spaces between two non-sibling nodes (see the meaning of *min_dist_on_subtrees* in the description of **sxpptree**(3)) ;

pw the width of an output page (see the meaning of *page_width* in the description of **sxpptree**(3)).

The type of the nodes constituting the tree, denoted by *SXNODE* above, must be compatible with the type of bare nodes manipulated by SYNTAX, declared as **struct sxnode_header_s** in *sxunix.h* (see **sxatc**(3)). More precisely, it MUST contain a **struct sxnode_header_s** as first component. Note that this is the same general requirement as when you use the other tools provided by SYNTAX for manipulating those trees (see **sxsmp**(3) and **sxat_mngr**(3)).

The labels of the nodes appearing in the graphical representation are defined as follows: for a leaf node corresponding to some terminal token, it is the text of that token, otherwise it is the node name as defined in the language specification *language_name***.at** (see **semat**(1) and the *SYNTAX Reference Manual*).

IMPORTANT NOTE

sxatedit must be called in an environment in which the global variable *sxatcvar* (see **sxatc**(3) and **sxunix**(3)) contains the tables of the language to which the tree corresponds. This is because *sxatedit* uses these tables. Note that this is achieved automatically in most cases when you use the procedure **syntax**(3).

SEE ALSO

semat(1), sxpptree(3), sxatc(3), sxat_mngr(3), sxsmp(3) and the *SYNTAX Reference Manual*.

BUGS

Those of **sxpptree**(3).

NAME

sxba_calloc, sxba_resize, sxba_empty, sxba_fill, sxba_0_bit, sxba_1_bit, sxba_bit_is_set, sxba_cardinal, sxba_scan, sxba_1_lrscan, sxba_0_lrscan, sxba_1_rlscan, sxba_0_rlscan, sxba_copy, sxba_and, sxba_or, sxba_xor, sxba_minus, sxba_not, sxba_is_empty, sxba_is_full, sxba_first_difference, sxba_read, sxba_write, sxbm_calloc, sxbm_resize, sxbm_free – bits array and bits matrix processing for SYNTAX.

SYNOPSIS

```
#include "sxunix.h"

SXBA sxba_calloc (bits_number)
        int     bits_number ;

SXBA sxba_resize (bits_array, new_bits_number)
        SXBA  bits_array ;
        int     new_bits_number ;

SXBA sxba_empty (bits_array)
        SXBA  bits_array ;

SXBA sxba_fill (bits_array)
        SXBA  bits_array ;

SXBA sxba_0_bit (bits_array, bit)
        SXBA  bits_array ;
        int     bit ;

SXBA sxba_1_bit (bits_array, bit)
        SXBA  bits_array ;
        int     bit ;

SXBOOLEAN sxba_bit_is_set (bits_array, bit)
        SXBA  bits_array ;
        int     bit ;

int sxba_cardinal (bits_array)
        SXBA  bits_array ;

int sxba_scan (bits_array, from_bit)
        SXBA  bits_array ;
        int     from_bit ;

int sxba_1_lrscan (bits_array, from_bit)
        SXBA  bits_array ;
        int     from_bit ;

int sxba_0_lrscan (bits_array, from_bit)
        SXBA  bits_array ;
        int     from_bit ;

int sxba_1_rlscan (bits_array, from_bit)
        SXBA  bits_array ;
        int     from_bit ;

int sxba_0_rlscan (bits_array, from_bit)
        SXBA  bits_array ;
        int     from_bit ;

SXBA sxba_copy (lhs_bits_array, rhs_bits_array)
        SXBA  lhs_bits_array, rhs_bits_array ;

SXBA sxba_and (lhs_bits_array, rhs_bits_array)
        SXBA  lhs_bits_array, rhs_bits_array ;

SXBA sxba_or (lhs_bits_array, rhs_bits_array)
        SXBA  lhs_bits_array, rhs_bits_array ;
SXBA sxba_xor (lhs_bits_array, rhs_bits_array)
        SXBA  lhs_bits_array, rhs_bits_array ;

SXBA sxba_minus (lhs_bits_array, rhs_bits_array)
        SXBA  lhs_bits_array, rhs_bits_array ;

SXBA sxba_not (bits_array)
        SXBA  bits_array ;

SXBOOLEAN sxba_is_empty (bits_array)
        SXBA  bits_array ;

SXBOOLEAN sxba_is_full (bits_array)
        SXBA  bits_array ;

int sxba_first_difference (bits_array_1, bits_array_2)
        SXBA  bits_array_1, bits_array_2 ;

SXBOOLEAN sxba_read (file, bits_array)
        int   file ;
        SXBA  bits_array ;

SXBOOLEAN sxba_write (file, bits_array)

        int   file ;
        SXBA  bits_array ;

SXBA *sxbm_calloc (lines_number, bits_number)
        int   lines_number, bits_number ;

SXBA *sxbm_resize (bits_matrix, old_lines_number, new_lines_number, new_bits_number)
        SXBA  *bits_matrix ;
        int   old_lines_number, new_lines_number, new_bits_number ;

SXVOID sxbm_free (bits_matrix)
        SXBA  *bits_matrix ;
```

DESCRIPTION

The **bits array** module allows to manipulate arrays of bits, which may be considered as an implementation of sets. As all arrays of the C language, bits arrays are indexed starting with zero. They are implemented as arrays of *SXBA_ELT*s (which is a macro expanding to *unsigned long int*) ; the first element of each such array keeps the number of significant bits in the other elements, which hold the bits themselves. The **SYNTAX** error recovery modules (see for example **sxp_rcvr**(3)) is a good example of the use of bits arrays.

The **bits matrix** module allows to manipulate matrices (two dimensional arrays) of bits. These matrices are implemented as arrays of **bits array**. If *bm* is a **bits matrix** (allocated via *sxbm_calloc*), *bm* [*i*] is a **bits array** over which all *BA*s operations (except freeing and resizing) can be applied.

sxba_calloc allocates a memory zone suitable for holding *bits_number* bits, which are all initialized to zero. It returns a pointer to that zone.

sxba_resize reallocates the existing *bits_array*, so that it may afterwards hold *new_bits_number* bits. If the new number of bits is greater than the old one, the bits that become allocated are reset. The bits belonging to both the old and the new arrays are not changed.

sxba_empty resets all bits of *bits_array*, while *sxba_fill* sets them.

sxba_0_bit resets the bit numbered *bit* in *bits_array*, while *sxba_1_bit* sets it.

sxba_bit_is_set returns *SXTRUE* if the bit numbered *bit* is set in *bits_array*, *SXFALSE* otherwise.

sxba_cardinal returns the number of bits which are set in *bits_array*.

sxba_scan returns the index in a left to right scan of the first non-null bit following *from_bit.* If the remainder of the array is all zeroes or if *from_bit* is greater or equal than *bits_number-1* it returns *-1.* If *from_bit* is negative, the scan starts at the first bit.

sxba_1_lrscan is a macro expanding to *sxba_scan*.

sxba_0_lrscan returns the index in a left to right scan of the first null bit following *from_bit.* If the remainder of the array is all ones or if *from_bit* is greater or equal than *bits_number-1* it returns *-1*. If *from_bit* is negative, the scan starts at the first bit.

sxba_1_rlscan returns the index in a right to left scan of the first non-null bit following *from_bit.* If the head of the array is all zeroes or if *from_bit* is less or equal than *zero* it returns *-1*. If *from_bit* is greater or equal than *bits_number*, the scan starts at the last bit.

sxba_0_rlscan returns the index in a right to left scan of the first null bit following *from_bit.* If the head of the array is all ones or if *from_bit* is less or equal than *zero* it returns *-1*. If *from_bit* is greater or equal than *bits_number*, the scan starts at the last bit.

sxba_copy puts into its first argument a copy of its second argument. It returns its first argument.

sxba_and (*sxba_or*, *sxba_xor*, *sxba_minus*) puts into its first argument the result of the bitwise *AND* (*OR*, *XOR*, *MINUS*) of its two arguments. It returns its (modified) first argument.

sxba_not inverts all significant bits of its argument and returns it, modified.

sxba_is_empty (*sxba_is_full*) returns *SXFALSE* if any bit is set (reset) in its argument, *SXTRUE* otherwise.

sxba_first_difference returns the index of the first bit which is set in one of its arguments and reset in the other, or *-1* if its arguments hold the same bits.

sxba_write (sxba_read) writes (reads) on file *file* opened via *open* (see **open**(2)) or *creat* (see **creat**(2)) the **bits array** *bits_array*. Returns *SXTRUE* on success, *SXFALSE* otherwise.

sxbm_calloc allocates a **bits matrix** which is an array of *lines_number* lines (indexed from *0* to *lines_number-1*), each line is (a pointer to) a **bits array** holding *bits_number* bits, all initialized to zero.

sxbm_resize reallocates the existing *bits_matrix*, so that it may afterwards hold *new_lines_number* lines and *new_bits_number* bits. If *new_lines_number* is greater than *old_lines_number*, the corresponding **bits array**s that are allocated are initialized to zero. If the new number of bits is greater than the old one, the bits that become allocated are reset. The bits belonging to both the old and the new arrays are not changed.

sxbm_free allows to free the memory used to hold the **bits matrix** *bits_matrix*.

SEE ALSO

sxunix(3) and the *SYNTAX Reference Manual*.

WARNINGS AND NOTE

The user is welcome to take advantage of the implementation, but should be aware that some of the functions described here will not work correctly if the last element of the underlying C array is not suitably padded with zeroes (but everything goes well if the user does not interfere).

Unless otherwise stated, no check is performed on the validity of the arguments passed to these functions. In particular, when a function expects two bits arrays as parameters, it is the user responsibility to pass two correct bits arrays of the same length.

Bits arrays allocated via *sxba_calloc* may be freed through *sxfree* (see **sxmem_mngr**(3)).

NAME

sxincl_mngr, sxpush_incl, sxpush_recincl, sxpush_uniqincl, sxpop_incl, sxincl_depth, sxincl_size, sxincl_get, sxincl_retrieve, sxincl_depend, sxincl_depend_but – include manager for SYNTAX.

SYNOPSIS

```
SXVOID sxincl_mngr (what)
        int       what;

SXBOOLEAN sxpush_incl (pathname)
        char      *pathname;

SXBOOLEAN sxpush_recincl (pathname)
        char      *pathname;

SXBOOLEAN sxpush_uniqincl (pathname)
        char      *pathname;

SXBOOLEAN sxpop_incl ()

SXINT sxincl_depth ()

SXINT sxincl_max_depth ()

SXINT sxincl_get_depth (incl_index)
        SXINT     incl_index;

SXINT sxincl_size ()

char *sxincl_get (incl_index)
        SXINT     incl_index;

SXINT sxincl_retrieve (pathname)
        char      *pathname;

SXVOID sxincl_depend (f)
        FILE      *f;
        SXINT     order;

SXVOID sxincl_depend_but (f)
        FILE      *f;
        SXINT     order;
        SXINT     excluded_index;
```

DESCRIPTION

The **include manager** module provides the user with a set of functions which may help his handling of the include mechanism, in particular for storing the pathnames of the parsed files. By default, the pathnames are stored in the default string table *sxstrmngr* (see **sxstr_mngr**(3)). These functions must be called by the user actions of the lexical level.

sxincl_mngr is the first and last entry point of the module to be called for initialization and finalization. It has a variable number of arguments :

- when the first is *SXOPEN*, *SXINIT*, *SXFINAL* or *SXCLOSE*, no further argument is expected ;
- when the first is *SXSEPARATE*, two further arguments *table* and *pathname*, must be present, where *table* is a pointer of type (sxstrmngr_t *), possibly NULL, to a string manager table used to store the pathnames of the includes files, and *pathname* is the name of the principal file (i.e., the main file from which other files are included).

When calling *sxincl_mngr (SXSEPARATE*, *table*, *pathname*), immediately after calling *sxincl_mngr (SXINIT)*, the pathnames are stored in *table*, and numbered contiguously from 0 (associated to the pathname of the main file from which other files are included) to *sxincl_size()*.

If *table* is equal to **NULL**, then a string manager table is automatically created and allocated inside the include manager to store the pathnames and it will be automatically destroyed by a call *sxincl_mngr (SXCLOSE)*.

sxpush_incl must be called at the beginning of each include file. It stores the current value of the structure sxsrcmngr (see **sxsrc_mngr**(3)) and opens a new input (include) file whose name is *pathname* and returns **SXTRUE** on success ; otherwise, it returns **SXFALSE** if the include file *pathname* cannot be opened or if it is a recursive call.

sxpush_recincl is similar to *sxpush_incl*, but accepts recursive inclusions.

sxpush_uniqincl is similar to *sxpush_incl*, but does not include any file more than once. When invoked on a file named *pathname* that has already been included, it ignores this file and returns **SXTRUE** (unless if the inclusion is recursive, in which case **SXFALSE** is returned). This function only works when *sxincl_mngr (SXSEPARATE, ...)* has been called before.

sxpop_incl must be call at the end of each include file. It closes the file and restores the context of its caller. Returns **SXTRUE** on success and **SXFALSE** otherwise (cannot close the include file or there is no caller).

sxincl_depth returns the current depth of the stack of included pathnames.

sxincl_max_depth returns the maximal depth of the stack of included pathnames.

sxincl_get_depth returns the maximal depth encountered so far for the pathname stored for the string table entry *incl_index.*

sxincl_size is only meaningful when *sxincl_mngr (SXSEPARATE, ...)* has been called before and returns the number of pathnames stored in the separate table.

sxincl_get returns the pathname stored for the string table entry *incl_index*.

sxincl_retrieve returns the index of the stored pathname *pathname*. If *pathname* is not stored in the string table, *sxincl_retrieve* returns the constant *-1*.

sxincl_depend prints to file *f*, in the order specified by *order*, the pathnames of all included files, i.e., all pathnames that have been passed as arguments to *sxpush_incl*. The file *f* must have been opened for writing before calling *sxincl_depend*. Currently supported values for *order* are *sxincl_order_reading*, *sxincl_order_inverse_reading*, *sxincl_order_increasing_depth*, and *sxincl_order_decreasing_depth*, the latter two being supported only when *sxincl_mngr (SXSEPARATE, ...)* has been called before.

sxincl_depend_put is similar to *sxincl_depend*, but does not print the pathname with index *excluded_index.*

SEE ALSO

sxunix(3), sxscanner(3) and the *SYNTAX Reference Manual*.

NAME

sxml – library for generating XML, YAML, or JSON code from a SEMC specification

SYNOPSIS

#include "sxml.h"

typedef char *SXML_TYPE_TEXT;

```
typedef ... SXML_TYPE_LIST;

void SXML_PRINT (FILE *OUTPUT, SXML_TYPE_LIST X);

SXML_TYPE_LIST SXML_L (SXML_TYPE_LIST L1);

SXML_TYPE_LIST SXML_LL (SXML_TYPE_LIST L1, L2);

SXML_TYPE_LIST SXML_T (SXML_TYPE_TEXT T1);

SXML_TYPE_LIST SXML_TLLT (SXML_TYPE_TEXT T1, SXML_TYPE_LIST L2,
                SXML_TYPE_LIST L3, SXML_TYPE_TEXT T4);

etc.

SXML_TYPE_LIST SXML_TTTLTLT (SXML_TYPE_TEXT T1, SXML_TYPE_TEXT T2,
                 SXML_TYPE_TEXT T3, SXML_TYPE_LIST L4,
                 SXML_TYPE_TEXT T5, SXML_TYPE_LIST L6,
                 SXML_TYPE_TEXT T7)
```

DESCRIPTION

SYNTAX is a compiler-generation system that provides for lexical and syntactic analysis, but it also suppports more "semantic" aspects, by means of three different processors: **semact**(1), **semat**(1), and **semc**(1).

These three processors are functional (and used to build SYNTAX itself by bootstrapping), but the two former are intrinsically linked with the C programming language, which is needed either to program semantic actions in **semact**(1) or to traverse the abstract syntax tree constructed by **semat**(1). Unfortunately, the C language is often too involved and error-prone for developing large compilers rapidly and efficiently.

For this reason, modern uses of SYNTAX tend to reduce, as much as possible, the proportion of compiler code written in C, and use other higher-level programming languages instead. There are currently two main approaches:

- The first approach [GLM02] consists in using, to describe and traverse the abstract syntax tree, the LNT (formerly known as LOTOS NT) language. LNT is a first-order functional language that is directly translated to C using the TRAIAN compiler [TRA]. LNT is strongly typed, and supports constructor types and pattern matching. It supports external functions written in C, which can, thus, perform side effects, such as modifying the abstract syntax tree and writing to a file. This approach has been intensively used by the VASY/CONVECS research teams of INRIA to build more than a dozen compilers (up to 50,000 lines of code) using SYNTAX.

- The second approach consists in using SYNTAX, and especially the **semc**(1) processor, to parse the input program and simultaneously print its abstract syntax tree to a file, in a tree-like format such as XML, YAML, or JSON. Once parsing is finished, the XML, YAML, or JSON file can be read and processed by any program that is fully independent from SYNTAX. This approach fits the needs of developers that prefer using other languages than C, such as Java, Python, etc.

The **sxml**(1) library was designed to support this second approach, and allows to build and print the abstract syntax tree. The types and functions exported by this library are meant to be invoked in the semantic actions of a **semc**(1) specification. Although the library is written in C, it can be used without deep knowledge of this language.

The main goal of the **sxml**(1) is to avoid the traditional multi-step approach, in which the abstract syntax tree is first specified (using type declarations), then built in memory (using calls to node constructors), and finally traversed and printed to a file. Instead, the library favors a much more concise approach, in which the abstract syntax tree is not specified, but dumped directly along each rule of the BNF grammar contained in the **semc**(1) specification.

Because SYNTAX is based on LR(1) or LALR(1) parsing, it is not possible to print the XML, YAML, or JSON code from left to right, by simple inserting "printf" statements in the semantic actions of the **semc**(1) specification. Indeed, each semantic action is executed *after* the corresponding syntax rule has been recognized. This does not allow to print XML terms from left to right, given that these terms are enclosed between <tag> and </tag> delimiters (only the closing delimiter </tag> can be printed when the rule is recognized).

Therefore, the solution is to assemble the XML, YAML, or JSON code fragments in memory, and print them to a file once parsing is complete. The **sxml**(1) provides the primitives for this. Its key data structure is a linked list of character strings (i.e., a "rope" data structure organized as a list rather than a binary tree). Each character string in the list may have a different length, as in the following list example:

```
"<sum>" -> "<var>a</var>" -> "<var>b</var>" -> "</sum>" -> NULL
```

Each list denotes the character string obtained by concatenating all its elements in sequence, e.g., for the above example:

```
"<sum><var>a</var><var>b</var></sum>"
```

The **sxml**(1) library therefore defines two types: **SXML_TYPE_TEXT**, which denotes a standard character string (i.e., "char *" in the C language), and **SXML_TYPE_LIST**, which denotes a pointer to a list.

To use the library properly, each non-terminal symbol "<t>" of the **semc**(1) specification should compute a synthesized attribute , noted "$LIST (<t>)", which is of type **SXML_TYPE_LIST** and denotes the fragment of XML, YAML, or JSON code produced for the non-terminal symbol "<t>".

Therefore, each syntax rule in the **semc**(1) specification, whose left-hand side defines a non-terminal symbol "<t>" should properly assign the attribute "$LIST (<t>)". This is done by an assignment of the following form:

```
$LIST(<t>) = SXML_... (...);
```

where SXML_... is one of the functions exported by the library: **SXML_L**, **SXML_LL**, **SXML_T**, **SXML_TLT**, etc. In the names of these functions, each letter 'L' corresponds to a parameter of type **SXML_TYPE_LIST**, and each letter 'T' corresponds to a parameter of type **SXML_TYPE_TEXT**.

For instance, the following syntax rule:

```
<expr> = %IDENTIFIER
```

may have a semantic action such as:

```
$LIST (<expr>) = SXML_T ($pste (%IDENTIFIER));
```

or:

```
$LIST (<expr>) = SXML_TTT ("<var>", $pste (%IDENTIFIER), "</var>");
```

Also, the following syntax rule:

```
<expr> = <expr> "+" <expr>
```

may have a semantic action such as:

```
$LIST (<expr>) = SXML_LL ($LIST (<expr>'), $LIST (<expr>''));
```

or:

```
$LIST (<expr>) = SXML_TLLT ("<sum>", <$LIST (<expr>'),
                 $LIST (<expr>''), "</sum>");
```

The "close" section of the specification will usually invoke the function **SXML_PRINT**() to print to a file the list generated for the axiom of the grammar.

SEE ALSO

semc(1) and the *SYNTAX Reference Manual*.

A complete example is given in directory "trunk/examples/lustre".

[GLM02] Hubert Garavel, Frederic Lang, and Radu Mateescu. *Compiler Construction using LOTOS NT*. Proceedings of the International Conference on Compiler Construction CC'2002 (Grenoble, France), April 2002. https://cadp.inria.fr/publications/Garavel-Lang-Mateescu-02.html

[TRA] *The TRAIAN compiler*. https://vasy.inria.fr/traian.

NOTES

1. The list of functions SXML_L...() and SXML_T..() implemented in the **sxml**(1) library is not exhaustive, and other functions can be added when needed for a particular language. Please send your extensions to Hubert.Garavel@inria.fr to get them added to the SYNTAX distribution.

2. The various lists computed by the $LIST attributes must be concatenated again and again, in a bottom-up manner, until the input program has been entirely parsed. To perform such concatenations efficiently (i.e., in constant time and space), the **sxml**(1) library implements circular linked lists, each list being referred to as a pointer to its last element. Such implementations details are opaque, meaning that the user of the library does not need to be aware of them while using the primitives provided by the library.

BUGS

Please report any problem to Hubert.Garavel@inria.fr.

NAME

sxparser, sxget_token, sxprecovery, SXSTACKtop, SXSTACKnewtop, SXSTACKreduce, SXSTACKtoken, sxparstack, sxpglobals, sxplocals – parser for SYNTAX.

SYNOPSIS

```
#include "sxunix.h"

struct sxtables  sxtables ;

SXVOID sxparser (what, sxtables)
          int                  what ;
          struct sxtables *sxtables ;

SXVOID sxpsature (nbt)
          int  nbt ;

struct sxtoken *sxget_token (tok_no)
          int  tok_no ;

SXBOOLEAN sxprecovery (what)
          int  what ;
```

DESCRIPTION

The **parser** is the module which performs the syntax analysis of source texts in the **SYNTAX** system. The **parser** calls the scanner (see **sxscanner**(3)) each time a new lexical token is needed. When the right hand side of a grammar rule is recognised the **parser** calls a "semantic" action. This action depends on the kind of semantics used. It may be a user action if the semantics is described "by actions" (see **semact**(1)), an abstract tree construction if the semantics is described "by abstract tree" (see **semat**(1) and **sxatc**(3)), a linear syntax tree for a pretty-printer (see **paradis**(1) and **sxatcpp**(3)) or the construction of a synthesised attributes evaluator (see **semc**(1)). This module also provides a set of procedures, macros and variables available to the semantic actions, i. e. just before a reduction. For a user point of view the main tools are the followings:

SXSTACKtoken (x_stack)

is a macro which expands into the token which is at the index *x_stack* in *parse_stack.* If *x_stack* refers to a non terminal symbol the *lahead* field of the token structure (**sxscanner** (3)) is 0.

SXSTACKtop ()

is a macro which designates, in the parse stack, the position of the rightmost symbol of the grammar rule, if any.

SXSTACKnewtop ()

is a macro which gives the index in the parse stack of the future left hand side non terminal. It designates the position of the leftmost symbol of the grammar rule, if any.
The tokens of the right hand side of a production lay in the parse stack between SXSTACKnewtop () and SXSTACKtop () and hence may be accessed in two ways: from left to right by *SXSTACKtoken* (*SXSTACKnewtop* ()), *SXSTACKtoken* (*SXSTACKnewtop* ()+1), ... or from right to left by *SXSTACKtoken* (*SXSTACKtop* ()), *SXSTACKtoken* (*SXSTACKtop* ()-1), ... For an empty production SXSTACKtop () is strictly less than SXSTACKnewtop (). The terminal symbols in lookahead can be accessed by *sxget_token* ().

SXSTACKreduce ()

is a macro which gives the number of the reduction.

FURTHER DESCRIPTION

The parser variables are declared in the include file **sxunix.h** and the declarations are close to the following :

```
struct sxparstack {
    struct sxtoken token /* lexical token */ ;
    SXSHORT      state  /* LR state       */ ;
    } ;
```

token contains a lexical token (see **sxscanner**(3)). *state* have no meaning for the user.

```
struct sxpglobals {
    struct sxparstack *parse_stack /* parse stack                */ ;
    SXSHORT          reduce       /* grammar rule number        */ ;
    SXSHORT          xps          /* parse stack top            */ ;
    SXSHORT          pspl         /* right hand side rule size  */ ;
    SXSHORT          stack_bot    /* parse stack bottom         */ ;
    } sxpglobals;
```

parse_stack is a pointer to the parse stack. Warning, this value may have changed from the previous call. *reduce* contains the number of the grammar rule which has been recognised. *xps* is the current parse stack top ; it refers to the rightmost symbol of the grammar rule. For an empty rule *xps* is meaningless. *pspl* is related to the size of the right hand side of production number *reduce,* it must not be used directly (see SXSTACKnewtop). *stack_bot* is the bottom index of the parse stack for the current activation of the parser, it is automatically managed upon recursive call of the parser.

```
struct sxplocals {
    struct P_tables P_tables  /* parser tables                          */ ;
    struct sxtables *sxtables  /* the whole tables                       */ ;
    SXSHORT         state     /* internal use                           */ ;
    struct sxtoken  *toks_buf  /* look ahead tokens buffer               */ ;
    int             atok_no   /* look ahead token number for actions    */,
                    ptok_no   /* token number for predicates            */,
                    Mtok_no   /* last token number                      */,
                    min       /* min toks_buf interval                  */,
                    max       /* max toks_buf interval                  */ ;
  } sxplocals ;
```

P_tables and *sxtables* are respectively the current parser tables and (a pointer to) the current language tables. *toks_buf* is a buffer which contains the look-ahead tokens already scanned. These tokens can be accessed via the *sxget_token(n)* function which returns a pointer to the *n*th token. These numbers (contrary to pointers) are independants from the manipulations (reallocation, displacement..) of *toks_buf.* From the user's actions and predicates point of view the *current* token is:

analyzer	case	current token	accessed by
parser	action	next token	sxget_token(sxplocals.atok_no)
	T_predicate	associated token	sxget_token(sxplocals.ptok_no)
	NT_predicate	next token	sxget_token(sxplocals.ptok_no)
scanner	post-action	post-action token	sxsvar.sxlv.terminal_token

Let *n* be a given token number. If *n* < *min* it cannot be accessed anymore. If *n* >= *min* and *n* <= *Mtok_no* it has already been scanned. In all cases where *n* >= *min* it can be accessed by *sxget_token(n)* even if *n* > *Mtok_no*; in such a case a sufficient number of calls to the *scanner* will be performed. If *c* is the number of the current token, the previous token is always accessible via *sxget_token(c-1).* The following relations are always true:

min < *atok_no* <= *ptok_no* <= *Mtok_no* <= *max* *sxparser*

is the main entry of the module to be called for parsing a given source text. Its possible uses are:
sxparser (*SXBEGIN*, *sxtables)* allocates the global variables which are language and source text independent (contained in sxpglobals).
sxparser (*SXOPEN*, *sxtables*) opens the tables and allocates the local variables when a new language is going to be processed (contained in sxplocals).
sxparser (*SXACTION*, *sxtables*) analyses a (new) source text written in the language corresponding to *sxtables*.
sxparser (*SXCLOSE*, *sxtables*) terminates all parsing for a given language and frees the local variables.
sxparser (*SXEND*, *sxtables*) frees the global variables.

sxprecovery

is the syntax level error processing module. It allows corrections and recoveries (see **recor** (1) and the *SYNTAX Reference Manual*).

SEE ALSO

bnf(1), **recor**(1), **sxunix**(3), **sxscanner**(3), **sxatc**(3), **sxatcpp**(3) and the *SYNTAX Reference Manual*.

NOTES

The contents of the structure *sxplocals* must be saved and restored by the user when switching between different parsers.

NAME

sxppp – pretty-printer of programs

SYNOPSIS

```
#include "sxunix.h"

struct sxppvariables  sxppvariables ;

SXVOID sxppp (what, sxtables)
        int         what ;
        struct sxtables  *sxtables ;
```

DESCRIPTION

sxppp is the end module of the **SYNTAX** paragraphing system **Paradis**. Once a program has been compiled into a tree, using **sxatcpp**(3), this module traverses that tree and interprets the pseudo-code produced by **paradis**(1) in order to output the pretty version of the program on the file *sxstdout*.

what may be:

SXOPEN to initialise the paragrapher, allocate internal structures and verify options ;

SXCLOSE
to free the internal structures ;

SXACTION
to produce the pretty version of the source program which has just been analysed and for which a tree has been built by **sxatcpp**(3).

Some of the pseudo-code directives make use of a set of options, which are all embedded in a *sxppvariables* structure; this structure contains the following variables, which may be modified by the user (the default value of each of these variables is obtained by setting it to 0 or NULL) :

kw_case
how should keywords be written (default: as in the grammar which has been fed into **paradis**(1)) ;

terminal_case
same as *kw_case*, but for each type of terminal (default: use the text the scanner returned) ;

kw_dark
should keywords be artificially darkened, by overstriking using backspaces (default: no!) ;

terminal_dark
same as *kw_dark*, but for each type of terminal ;

no_tabs
do not optimize spaces into tabs (default: any sequence of more than one space leading to a tabulation position is turned into a HT character (ASCII 9)) ;

tabs_interval
number of columns between two tab positions (default: installation dependent ; the value used is that of SXTAB_INTERVAL, usually defined in **sxunix**(3)) ;

line_length
maximal length of output lines (default: 79) ;

block_margin
do not preserve structure when deeply nested ; for a discussion on this option and the next one, see **ppc**(1) ;

max_margin
do not indent lines further than that (default: two-thirds of the line length) ;

sxppp is not usually called directly by the user, but instead through the *SXSEMPASS* entry of the **sxatcpp**(3) module. It is the user's responsibility to open and close the output file *sxstdout*, onto which goes all output.

FURTHER DESCRIPTION

The *sxppvariables* structure is declared in the include file **sxunix.h** ; other than for variables private to the paragrapher, that declaration is close to the following :

```
struct sxppvariables {
    SXCASE     kw_case ;         /* How should keywords be written               */
    SXCASE     *terminal_case ;  /* Same as kw_case, but for each type of terminal */
    SXBOOLEAN  kw_dark ;         /* Should keywords be artificially darkened     */
    SXBOOLEAN  *terminal_dark ;  /* Same as kw_dark, but for each type of terminal */
    SXBOOLEAN  no_tabs ;         /* Do not optimize spaces into tabs             */
    short int  tabs_interval ;   /* Number of columns between two tab positions  */
    SXBOOLEAN  block_margin ;    /* Do not preserve structure when deeply nested */
    short int  max_margin ;      /* Do not indent lines further than that        */
    short int  line_length ;     /* What it says                                 */
    SXBOOLEAN  is_error ;        /* SXTRUE if the pretty-printer detected an error */
    long int   char_count ;      /* Number of chars output by the pretty-printer */
} sxppvariables ;
```

When not NULL, *terminal_case* is a pointer to an array of *SXCASE*s, indexed by the internal codes assigned to each token by SYNTAX (see **tdef**(1)) ; each element of this array is either:

SXNO_SPECIAL_CASE

(i.e. 0), which means to use the grammatical form of the terminal if it is non-generic, else the text kept by the scanner ;

SXUPPER_CASE

as above, but alphabetical characters are upper-cased ;

SXLOWER_CASE

as above, but alphabetical characters are lower-cased ;

SXCAPITALISED_INITIAL

alphabetical characters are lower-cased when they occur after an alphabetical or numerical character, upper-cased otherwise ;

Note that a non-NULL *terminal_case* overrides any definition of *kw_case*.

terminal_dark and *kw_dark* are similar, only the information they hold are just boolean values stating for each terminal if it should be output with embedded overstrikes, to make it look darker on a line printer.

SEE ALSO

paradis(1), tdef(1), ppada(1), ppc(1), sxunix(3), sxatcpp(3), sxscanner(3) and the *SYNTAX Reference Manual*.

NOTES

The contents of the structure *sxppvariables* must be saved and restored by the user when switching between source languages for which *SXOPEN* has been done, but not *SXCLOSE*.

sxppp (SXACTION, ...) momentarily exchanges the contents of the FILE variables ***stdout** and ***sxstdout**, if they are not the same. This is an (admittedly crude) attempt at efficiency, as the paragrapher is very IO-bound. The user is usually not aware of this substitution, unless (s)he starts catching signals...

BUGS

If the global variable *sxverbosep* is not 0, an attempt is done at animating the user's screen. This results in much more IO and, in some situations, a messy screen.

NAME

sxscanner, sxscan_it, check_keyword, sxsrecovery, sxssrecovery, SXSCAN_LA_P, SXCURRENT_SCANNED_CHAR, sxttext, sxeof_code, sxnextsyno, sxkeywordp, sxgenericp, sxsource_coord, sxtoken, sxlv_s, sxlv, sxsvar – scanner for SYNTAX.

SYNOPSIS

```
#include "sxunix.h"

struct sxtables  sxtables ;

SXVOID sxscanner (what, tables)
        int                what ;
        struct sxtables *sxtables ;

SXVOID sxscan_it ()

int sxcheck_keyword (string, length)
   char *string ;
   int   length ;

SXBOOLEAN sxsrecovery (what, state_no, class)
         int                what,
                            state_no ;
         unsigned char *class ;

SXBOOLEAN sxssrecovery (what, state_no, class)
         int                what,
                            state_no ;
         unsigned char *class ;
```

DESCRIPTION

The **scanner** is the module which performs the lexical analysis of source texts in the **SYNTAX** system. The **parser** calls it each time a new lexical token is needed (see **sxparser**(3)). A token is described by a set of variables which are put together in a structure called **sxtoken**. Some of the scanner local variables (including sxtoken) are gathered in a structure called **sxsvar**. This structure achieved two different purposes: first it contains the variables which are pertinent from the user's actions and predicates point of view (see the *SYNTAX Reference Manual*) ; second, all these variables must be saved (and restore) by the user when switching between different (or even recursive) scanners. Moreover this module provides a set of procedures and macros which operates on these variables. These structures and macros are declared in the include file **sxunix.h** ; with declarations close to the following :

```
struct sxsource_coord {
   char              *file_name /* file name          */ ;
   unsigned long  line          /* line number       */ ;
   unsigned int    column       /* column number  */ ;
   } ;
```

sxsource_coord is a structure which contains the coordinates of a character in a source file: *file_name* is the source file name, *line* and *column* are the positions of the character in the source file.

```
struct sxtoken {
   int                          lahead              /* token code          */ ;
   unsigned int               string_table_entry /* string table index  */ ;
   struct sxsource_coord source_index       /* source coordinates */ ;
```

```
    char              *comment        /* comment          */ ;
    } ;
```

sxtoken is a structure which contains information about a terminal token: *lahead* is the internal code of the token, *string_table_entry* is an index to its character string representation (see **sxstrmngr**(3)), *source_index* is the source coordinates of its first character and *comment* is a (possibly NULL) pointer to a character string which is the catenation of the kept portions (see operator "-" of the chapter on *lecl* in the *SYNTAX reference manual*) of the comments **preceeding** the token.

For internal reasons the scanner local variables are separated in two structures *sxlv* and *sxlv_s*.

```
struct sxlv_s {
    long          *counters      /* size = S_counters_size    */ ;
    int            include_no    /* include number            */ ;
    int            ts_lgth_use   /* size of token string buffer */ ;
    char          *token_string  /* token string buffer       */ ;
    } ;
```

The *sxlv_s* structure contains a part of the scanner local variables: the long integers of the array *counters* are manipulated via the "@Set", "@Reset", "@Incr" and "@Decr" actions and tested via the "&Is_Set" and "&Is_Reset" predicates. The counter numbered n can be accessed by sxsvar.sxlv_s.counters [n]. *include_no* is the current include number (0 for "main" source text). *ts_lgth_use* is the current maximum size of the buffer *token_string* which contains the string of the characters which have been kept (according the lexical specification) since the beginning of the current scanner call.

```
struct sxlv {
    int            ts_lgth             /* token string size          */ ;
    int            current_state_no    /* scanner state number       */ ;
    struct sxtoken terminal_token      /* terminal token structure   */ ;
    unsigned char  source_class        /* current source class       */ ;
    int            include_action      /* include post action        */ ;
    SXSHORT        previous_char       /* before the current token   */ ;
    struct mark {                      /* "@Mark" processing         */
        struct sxsource_coord source_coord
                                   /* source coordinates         */ ;
        int                       index
                                   /* token string index         */ ;
        SXSHORT                   previous_char
                                   /* before the marked character */ ;
        } mark ;
    } ;
```

The *sxlv* structure contains the other part of the scanner local variables: *ts_lgth* is the number of characters in *sxlv_s.token_string*. *current_state_no* is the current state of the scanner, mainly used in a predicate via the **SXSCAN_LA_P** macro to know if the current character has been read in look-ahead. *terminal_token* is the interface between the scanner and the parser. All these informations are only pertinent in a post action, however *sxlv.terminal_token.source_index* is always valid and *sxlv.terminal_token.comment* is valid when used in a terminal token recognition. *source_class* and *include_action* are not pertinent to the user. *previous_char* is the character before the first character of the current token. *mark* is a structure which contains the informations which are kept by the scanner on each "@Mark" action.

```
struct sxsvar {
    struct S_tables  S_tables  /* scanner tables          */ ;
    struct sxlv_s    sxlv_s    /* local variables (part I)  */ ;
    struct sxlv      sxlv      /* local variables (part II) */ ;
    struct sxtables *sxtables /* the whole language tables*/ ;
    } sxsvar ;
```

sxsvar contains the scanner variables which must be saved by the user on each recursive call to the scanner. *S_tables* are the scanner tables, *sxlv, sxlv_s* the local scanner variables and *sxtables* the pointer to the whole tables of the current language.

sxscanner
: is the main entry of the module to be called for scanning a given source file.
When a new language is going to be processed the call *sxscanner* (*SXOPEN*, *tables*) opens the tables from the scanner point of view and initialises a part of *sxsvar*. The call *sxscanner* (*SXINIT*, *tables*) initialises the rest of its local variables and read the first character of the source text. The call *sxscanner* (*SXACTION*, *tables*) is done by the parser each time it needs a new token ; this call is equivalent to *sxscan_it().* *sxscanner* (*SXCLOSE*, *tables*) terminates all scanning for a given language.

*(*sxsvar.SXS_tables.check_keyword)* (*string*, *length*)
: if *string* of length *length* represents a keyword returns its internal code as terminal symbol else 0. This function can only be used via the current *S_tables*.

sxsrecovery and *sxssrecovery*
: are the lexical level error processing modules ; the first is the standard one (correction and recovery) while the second is a simplified (hence compact) version which only deletes the erroneous character.

SXSCAN_LA_P
: is a (SXBOOLEAN) macro which may be used in the code of a user's predicate to know if the current character (i.e. the character of the source text whose class is associated with the predicate being processed) has been read in look ahead.

SXCURRENT_SCANNED_CHAR
: is a (char) macro which may be used in the code of a user's predicate to retrieve the current character ; it uses SXSCAN_LA_P.

sxttext (*sxtables*, *look_ahead*)
: is a macro which expands into a character pointer. This pointer refers to the name of the terminal symbol whose internal code is *look_ahead* in the language whose tables are *sxtables.*

sxeof_code (*sxtables*)
: is a macro which expands into an integer. This integer is the internal code of the token "End Of File" in the language whose tables are *sxtables.*

sxnextsyno (*sxtables*, *look_ahead*)
: is a macro which expands into a character pointer. This pointer refers to the name of the next synonym of *look_ahead* in the language whose tables are *sxtables.*

sxkeywordp (*sxtables*, *look_ahead*)
: is a macro which expands into a SXBOOLEAN value : SXTRUE if *look_ahead* is the code of a keyword in the language whose tables are *sxtables* else SXFALSE.

sxgenericp (*sxtables*, *look_ahead*)
: is a macro which expands into a SXBOOLEAN value : SXTRUE if *look_ahead* is the code of a generic terminal in the language whose tables are *sxtables* else SXFALSE.

SEE ALSO

lecl (1), **tdef** (1), **sxunix** (3), **sxsrcmngr** (3), **sxparser** (3) and the *SYNTAX Reference Manual*.

NOTES

The contents of the structure *sxsvar* must be saved and restore by the user when switching between different scanners.

NAME

sxsrcmngr, sxsrc_mngr, sxnext_char, sxnextchar, sxlafirst_char, sxlanext_char, sxlaback, sxX, sxsrcpush – source manager for SYNTAX.

SYNOPSIS

```
#include "sxunix.h"

struct sxsrcmngr  sxsrcmngr ;

SXVOID sxsrc_mngr (what, infile, file_name)
        int        what ;
        FILE       *infile ;
        char       *file_name ;

SXSHORT sxnext_char ()

SXSHORT sxlafirst_char ()

SXSHORT sxlanext_char ()

SXVOID sxlaback (backward_number)
        int        backward_number ;

SXVOID sxX (inserted)
        SXSHORT inserted ;

SXVOID sxsrcpush (previous_char, chars, coord)
        SXSHORT  previous_char ;
        char       *chars ;
        struct sxsource_coord  coord ;
```

DESCRIPTION

The **source manager** module is responsible for all accesses by other **SYNTAX** modules to the end user's source text. Opening and closing files are not considered as accesses and must be done elsewhere. This module provides a set of procedures and variables. The variables are contained in a *sxsrcmngr* structure, which contains at least the following, valid after initialisation via a call to *sxsrc_mngr* :

infile the current input stream ;

source_coord
the coordinates of *current_char* (initially *column* 0 of *line* 0) ;

previous_char
the character that was returned before *current_char* (initially *EOF*) ;

current_char
the last character returned by a call to *sxnext_char* (initially *SXNEWLINE*).

sxsrc_mngr is the first and last entry point of the module to be called for accessing a given source file. It has a variable number of arguments ; when the first is *SXINIT*, two other arguments must be present : a pointer *infile* to the FILE to be accessed by other functions of the module (obtained e.g. via a call to **fopen**(3S)), and a pointer *file_name* to the name of that file ; when the first argument is *SXFINAL*, no more interaction will be done on the current input stream, so that internal structures may be freed. It is the user's responsibility to close the file.

sxnext_char returns the next character, usually by calling *sxgetchar* (see **sxunix**(3)) ; *sxnextchar* is a rather complicated macro with the same semantics, defined for efficiency reasons.

FURTHER DESCRIPTION

The *sxsrcmngr* structure is declared in the include file **sxunix.h** ; that declaration is close to the following :

```
struct sxsrcmngr {
    FILE       *infile ;          /* stream being accessed              */
    struct sxsource_coord  source_coord ;
                                  /* coordinates of current_char        */
    SXSHORT    previous_char,     /* preceding current_char             */
               current_char ;     /* last character returned            */
    SXSHORT    *buffer ;          /* characters read in look-ahead      */
    struct sxsource_coord *bufcoords ;
                                  /* coordinates of characters in buffer */
    int        buflength ;        /* usable size of buffer              */
    int        bufused ;          /* index of last character read       */
    int        bufindex ;         /* index of current_char              */
    int        labufindex ;       /* index of looked-ahead character    */
} sxsrcmngr ;
```

When there is some *look-ahead* (information about this may be extracted from the scanner, see **sxscanner**(3)), more variables may be safely accessed :

buffer pointer to an array of *SXSHORT*s, which contains all characters read, from at least the current character (which may therefore be accessed either by *current_char* or by *buffer [bufindex]*), to the last character examined in the course of a *look-ahead* (note that this last, accessed by *buffer [bufused]*, may be beyond the current *looked-ahead* character, which is accessed by *buffer [labufindex]*) ;

bufcoords
pointer to an array containing the coordinates of all characters kept in *buffer* (so that those coordinates are correctly restored whatever occurs) ; this should be of little importance in most cases ;

buflength
size of *buffer* and of *bufcoords* ; this should be of no concern to the user ;

bufused
index in *buffer* of the last character placed there ; this is the last character ever read on *infile* ;

bufindex
index in *buffer* of *current_char* ;

labufindex
index in *buffer* of latest character examined in *look-ahead*.

sxnext_char and *sxnextchar* return the next character, whether by incrementing *bufindex* or by calling *sxgetchar* (see **sxunix**(3)) ; the variables *source_coord*, *previous_char* and *current_char* are updated.

sxlafirst_char returns the first **looked-ahead** character (i.e. the one immediately following *current_char*), without modification to those variables.

sxlanext_char returns the next **looked-ahead** character (the one following that returned by a previous call to *sxlafirst_char* or *sxlanext_char*), without those modifications either.

sxlaback goes back *backward_number* characters of *look-ahead* ; there must not have been less than that number of calls to *sxlanext_char* since the previous call to *sxlafirst_char*.

sxX should not be used except by a **SYNTAX** scanner, but is documented here for completeness purposes. It is used for error-recovery and inserts *inserted* before *current_char*, so that this same *current_char* will be the character returned by the next call to *sxnext_char* . . .

sxsrcpush pushes the characters of *chars* ahead of *current_char*, preceded by *previous_char* ; the first character of *chars* becomes the *current_char*, with coordinates *coord*. The follower of the last character pushed will be the character which was in *current_char* before the call, with its coordinates unchanged.

SEE ALSO

sxunix(3), sxscanner(3) and the *SYNTAX Reference Manual*.

NOTES

The contents of the structure *sxsrcmngr* must be saved and restored by the user when switching between source files. *sxsrc_mngr* must be called with *SXINIT* each time a new source file has been opened, with *SXFINAL* each time an old source file will be closed.

sxnext_char will try to read past an EOF if required to (and will usually succeed when input is from a terminal).

It is possible to make believe a C string comes from a file, by suitable use of *sxsrcpush*.

The *coordinates* managed by this module are not related to the builtin predicates of **sxscanner**(3).

BUGS

There is no simple way to start reading in the middle of a stream : the *source_coord* in particular cannot be correctly positioned except by horrendous means.

Since the second argument to *sxsrcpush* is a character pointer, inserting some funny characters may be invalidly done (on a machine with signed characters on eight bits, if EOF is defined as the integer value −1, insertion of "\377" will probably result in an insertion of EOF). Furthermore, insertion of a null character cannot be done that way.

NAME

sxstrmngr_t, sxstrmngr, sxstr_mngr, sxstr_retrievelen, sxstr_retrieve, sxstr_savelen, sxstr_save, sxstr_get, sxstr_length, sxstr_size, sxstr_dump, sxstr_save_keywords, sxstr2retrieve, sxstr2save, sxstrretrieve, sxstrsave, sxstrget, sxstrlen, SXSTRtop – string manager for SYNTAX.

SYNOPSIS

```
#include "sxunix.h"

typedef struct sxstrmngr sxstrmngr_t ;

SXVOID sxstr_mngr (what, strtable)
        int         what ;
        sxstrmngr_t *strtable ;
```

General primitives operating on any string table passed as 1st argument

```
unsigned int sxstr_save (strtable, string)
        sxstrmngr_t *strtable ;
        char      *string ;

unsigned int sxstr_savelen (strtable, string, strlength)
        sxstrmngr_t *strtable ;
        char      *string ;
        unsigned long  strlength ;

unsigned int sxstr_retrieve (strtable, string)
        sxstrmngr_t *table ;
        char      *string ;

unsigned int sxstr_retrievelen (strtable, string, strlength)
        sxstrmngr_t *table ;
        char      *string ;
        unsigned long  strlength ;

char *sxstr_get (strtable, string_table_entry)
        sxstrmngr_t *strtable ;
        unsigned int  string_table_entry ;

unsigned long sxstr_length (strtable, string_table_entry)
        sxstrmngr_t *strtable ;
        unsigned int  string_table_entry ;

unsigned int sxstr_size (strtable)
        sxstrmngr_t *strtable ;

SXVOID sxstr_dump (f, strtable)
        FILE     *f ;
        sxstrmngr_t *strtable ;

SXVOID sxstr_save_keywords (strtable, lang)
        sxstrmngr_t*strtable ;
        char      *lang ;
```

Traditional primitives that operate only on the global variable sxstrmngr

```
sxstrmngr_t sxstrmngr ;

char *sxstrget (string_table_entry)
        unsigned int  string_table_entry ;

unsigned long sxstrlen (string_table_entry)
        unsigned int  string_table_entry ;

unsigned int sxstrsave (string)
        char      *string ;
```

```
unsigned int sxstr2save (string, strlen)
        char      *string ;
        unsigned long  strlen ;

unsigned int sxstrretrieve (string)
        char      *string ;

unsigned int sxstr2retrieve (string, strlen)
        char      *string ;
        unsigned long  strlen ;

unsigned int SXSTRtop ()
```

DESCRIPTION

The **string manager** module allows to store strings once and subsequently refer to them with a unique positive integral number (‘‘string table entry’’). This module provides two sets of primitives (procedures and macro-definitions): the first set operates on any string table passed as first argument, whereas the second set operates only on the default string table, stored in the global variable *sxstrmngr* (of type *sxstrmngr_t*, the details of which are of no interest to the naive user). The **SYNTAX** scanner (see **sxscanner**(3)) is a good example of the use of the second set of primitives.

sxstr_mngr is the first and last entry point of the module to be called for using a string table. It may be called with four possible arguments :

- *SXBEGIN* states that the default string table *sxstrmngr* should become initialized. If this table was already initialized and used, the call *sxstr_mngr (SXBEGIN)* reinitializes the table, the internal structures of which are cleared, without the need to call *sxstr_mngr (SXEND)* beforehand; in such case, it is possible and recommended to call *sxstr_mngr (SXCLEAR)* rather than *sxstr_mngr (SXBEGIN)* so as to better document the expected effect.
- *SXEND* states that the strings saved in the default table *sxstrmngr* will be no more accessed, so that internal structures may be freed.
- *SXOPEN strtable* states that the string table **strtable* should become initialized. If this table was already initialized and used, the call *sxstr_mngr (SXOPEN, strtable)* reinitializes **strtable,* without the need to call *sxstr_mngr (SXCLOSE, strtable)* beforehand.
- *SXCLOSE strtable* statea that the strings saved in the table **strtable* will be no more accessed, so that internal structures may be freed.

sxstr_get returns (a pointer to) the null-terminated string associated with *string_table_entry* in the table *strtable*. *sxstr_length* returns the number of characters in that string, not including the terminating null character. *sxstr_get* and *sxstr_length* are (side-effect free) macros.

sxstr_save saves the string *string* in the string table *strtable* and returns the associated unique number. Further calls with a string comparing equal with *string* will return the same number.

sxstr_savelen is similar ; the difference is that the number of characters in *string* is given (as *strlength),* and does not have to be computed. This allows to save strings that contain null characters and may be not null-terminated.

sxstr_retrieve is only different from *sxstr_save* in that it does not store the string *string* in the string table if it is not there, and returns instead the constant *SXERROR_STE* (see NOTES below).

sxstr_retrievelen is similar ; the difference is that the number of characters in *string* is given (as *strlength),* and does not have to be computed.

sxstr_size is a side-effect-free macro that returns the current number of elements (including the two special entries *SXERROR_STE* and *SXERROR_STE*) stored in the string table *strtable*.

sxstr_dump prints the contents of the string table *strtable* to file *f*, which has to be opened before calling *sxstr_dump*.

sxstr_save_keywords adds all the reserved keywords of the language *lang* to *strtable.* Currently, only the C programming language is supported (i.e., *lang* = "C").

The primitives of the second set of primitives differ from those of the first set only by the fact that they operate on the default string table, i.e., the global variable *sxstrmngr*:

- *sxsxtrget (...)* is similar to *sxstr_get (&sxstrmngr, ...)*
- *sxsxtrlen (...)* is similar to *sxstr_length (&sxstrmngr, ...)*
- *sxsxtrsave (...)* is similar to *sxstr_save (&sxstrmngr, ...)*
- *sxsxtr2save (...)* is similar to *sxstr_savelen (&sxstrmngr, ...)*
- *sxsxtrretrieve (...)* is similar to *sxstr_retrieve (&sxstrmngr, ...)*
- *sxsxtr2retrieve (...)* is similar to *sxstr_retrievelen (&sxstrmngr, ...)*
- *SXSTRtop()* is similar to *sxstr_size (&sxstrmngr)*

SEE ALSO

sxunix(3) and the *SYNTAX Reference Manual*.

NOTES

Two special constants are defined in the header file *"sxunix.h"*. *SXERROR_STE* is the string table entry associated with any generic token inserted by the parser during error recovery. *SXEMPTY_STE* is predefined as the string table entry associated with the empty string.

BUG

Because *sxstrget* and *sxstrlen* are implemented as macros, they should not be used directly with the result of a call to *sxstrsave* or *sxstr2save*, as that call may change the value of variables used in the macros, which might on some systems produce invalid results. Thus the user is urged not to write things as :

```
   ptr = sxstrget (sxstrsave (string)) ;
```

and to use an intermediate variable instead, as in :

```
   ste  = sxstrsave (string) ;
   ptr = sxstrget (ste) ;
```

NAME

sxsyntax – SYNTAX.

SYNOPSIS

#include "sxunix.h"

```
SXVOID sxsyntax (what, tables, ...)
        int        what ;
        struct sxstables *tables ;
```

DESCRIPTION

sxsyntax (often abbreviated as *syntax*) is the entry point function for invoking a syntactic/lexical analyzer built using **SYNTAX.**

This function can be used in two different modes:

- The normal mode, which covers 90% of the standard needs for building a compiler using **SYNTAX** and parsing one or many files specified, e.g., on the command line. The normal mode provides a high-level interface above the various modules provided by **SYNTAX**, such as **sxerr_mngr**, **sxincl_mngr**, **sxsrc_mngr**, **sxstr_mngr**, etc.

- The special mode, which is reserved to particular cases, such as parsing a file whose contents are written in different languages described by two or more syntactic grammars; for analyzing a file whose contents are described by a single grammar, the normal mode is sufficient and should be preferred. The special mode provides a low-level interface, which allows a fine-grained control of the various **SYNTAX** modules, but requires many additional function calls to manage these modules explicitly.

The value of the first argument *what* (7 possible values) determines the effect of the *sxsyntax* function.

The second argument *tables* always corresponds to the tables generated by other modules of **SYNTAX** and representing the language to be analyzed.

NORMAL MODE

In normal mode, the *sxsyntax* function must be invoked five times, with the following arguments in the following order:

- *sxsyntax (SXINIT, tables, use_include_manager),* where *use_include_manager* has type *SXBOOLEAN*, initialises an analyzer for the language described by *tables*. This call successively invokes *sxopentty()*, *sxstr_mngr (SXBEGIN)*, *(*(tables->analyzers.parser)) (SXBEGIN, tables)*, *syntax (SXOPEN, tables)*, and, if *use_include_manager* is equal to *SXTRUE*, it also invokes *sxincl_mngr (SXOPEN)*.

- *sxsyntax (SXBEGIN, tables, file, pathname)* where *file* has type *FILE** and *pathname* has type *char**, prepares the analysis of the file named *pathname*. *file* is a pointer to the file *pathname*, which should be opened before calling *sxsyntax (SXBEGIN, ...)*. This call successively invokes *sxsrc_mngr (SXINIT, file, pathname)*, *sxerr_mngr (SXBEGIN)*, and, if *use_include_manager* was equal to *SXTRUE* in the prior call to *sxsyntax (SXINIT, ...)*, it also invokes *sxincl_mngr (SXINIT)*.

- *sxsyntax (SXACTION, tables)* triggers the analysis of *file*, whose value was provided in the prior call to *sxsyntax (SXBEGIN, ...)*.

- *sxsyntax (SXEND, tables)* terminates the analysis of *file*, whose value was provided in the prior call to *sxsyntax (SXBEGIN, ...)*. This call successively invokes *sxsrc_mngr (SXFINAL)*, *sxerr_mngr (SXEND)*, and, if *use_include_manager* was equal to *SXTRUE* in the prior call to *sxsyntax (SXINIT, ...)*, it also invokes *sxincl_mngr (SXFINAL)*.

- *sxsyntax (SXFINAL, tables, delete_str_mngr)* where *delete_str_manager* has type *SXBOOLEAN*, terminates all the **SYNTAX** modules of the analyzer, except perhaps the **str_mngr** (keeping the string table may be useful if data structures referring to entries in this table are still in use). This call successively invokes *syntax (SXCLOSE, tables)*, *(*(tables->analyzers.parser)) (SXEND, tables)*, and, if *use_include_manager* was equal to *SXTRUE* in the prior call to *sxsyntax (SXINIT, ...)*, it also invokes *sxincl_mngr (SXINIT)*. Finally, if *delete_str_manager* is equal to *SXTRUE*, the call also invokes *sxstr_mngr (SXEND)*.

SPECIAL MODE

In special mode, the *sxsyntax* function must be invoked three times with the following arguments:

- *sxsyntax (SXOPEN, tables)*

- *sxsyntax (SXACTION, tables)*

- *sxsyntax (SXCLOSE, tables)*

Using the special mode, many additional function calls are required.

The following examples show how the standard mode can be used. See also the examples in the "examples" directory of the **SYNTAX** distribution and in the **SYNTAX** processors (themselves written using **SYNTAX** and bootstrapped), some of which illustrate the special mode (search for "syntax (SXOPEN" and "syntax (SXCLOSE"). For instance, the **semc** processor uses the special mode to analyze files mixing BNF grammars and C code. Also, the C and Ada pretty-printers *examples/ppc* and *examples/ppada* illustrate the use of two different grammars, one for parsing command-line options, and another one for parsing the program text.

EXAMPLE 1

The following code illustrates the simple case of an analyzer that reads a program in a file named "pathname", written in a language defined by one single grammar, without using the include manager:

```
FILE *file;
file = fopen ("pathname", "r");
if (file == NULL) /* error */
sxsyntax (SXINIT, &sxtables, SXFALSE);
sxsyntax (SXBEGIN, &sxtables, file, "pathname");
sxsyntax (SXACTION, &sxtables);
sxsyntax (SXEND, &sxtables);
sxsyntax (SXFINAL, &sxtables, SXTRUE);
(void) fclose (file);
```

EXAMPLE 2

The following code illustrates the case of an analyzer that reads from the standard input a program written in a language defined by one single grammar, without using the include manager. The strings encountered during the analysis and stored in the string manager table are kept available after the analysis (e.g., because they might be referenced by the abstract syntax tree constructed during the analysis) until the string manager table is destroyed explicitly:

```
sxsyntax (SXINIT, &sxtables, SXFALSE);
sxsyntax (SXBEGIN, &sxtables, stdin, "");
sxsyntax (SXACTION, &sxtables);
sxsyntax (SXEND, &sxtables);
sxsyntax (SXFINAL, &sxtables, SXFALSE);
... /* here, the strings are still available */
sxstr_mngr (SXEND);
... /* here, the strings are no longer available */
```

EXAMPLE 3

The following code illustrates the case of an analyzer that successively parses a list of files specified on the command line. Before analyzing each next file, the string table must be explicitly purged. Notice that this example is a simplification of *src/sxmain.c*.

```
int i;
FILE *file;
sxsyntax (SXINIT, &sxtables, SXFALSE);
for (i = 1; i < argc; i++) {
    file = sxfopen (argv[i], "r");
    if (file == NULL) {
        fprintf (sxstderr, "%s: cannot read \"%s\"\n",
                argv[0], argv[i]);
    } else {
        sxsyntax (SXBEGIN, &sxtables, file, argv[i]);
        sxsyntax (SXACTION, &sxtables);
        sxsyntax (SXEND, &sxtables);
        /* purge the string table */
        sxstr_mngr (SXCLEAR);
        (void) fclose (file);
    }
}
sxsyntax (SXFINAL, &sxtables, SXTRUE);
```

EXAMPLE 4

The following code shows additional calls to control the format of error messages, to store the pathnames of the included files in a separate table, and to use the source manager in absolute mode.

```
FILE *file;
file = fopen ("pathname", "r");
if (file == NULL) /* error */
/* the include manager will be used */
sxsyntax (SXINIT, &sxtables, SXTRUE);
sxsyntax (SXBEGIN, &sxtables, file, "pathname");
/* creation of a table to store pathnames of included files */
sxstr_mngr (SXOPEN, &INCLUDE_TABLE);
sxincl_mngr (SXSEPARATE, &INCLUDE_TABLE, "pathname");
/* selection of a custom (Gcc-like) error format */
sxerr_mngr (SXFORMAT, SXERR_FORMAT_CUSTOM, "%s:%lu: ");
/* use of the source manager in absolute mode */
sxsrc_mngr (SXABSOLUTE, "apparent pathname", firstline);
sxsyntax (SXACTION, &sxtables);
sxsyntax (SXEND, &sxtables);
sxsyntax (SXFINAL, &sxtables, SXFALSE);
...
```

```
/* deletion of the two string tables */
sxstr_mngr (SXEND);
sxstr_mngr (SXCLOSE, &INCLUDE_TABLE);
(void) fclose (file);
```

EXAMPLE 5

The following code shows how to redirect the messages written by **SYNTAX** on the standard output and standard error. For instance, the messages written on the standard output will be redirected to a file *pathname*, while the messages written on the standard error will be redirected to the standard output. These instructions should be placed before invoking *sxopentty()*, or before invoking *syntax (SXINIT, ...)*, which itself invokes *sxopentty()*.

```
FILE *file;
file = fopen ("pathname", "r");
if (file == NULL) /* error */
sxstdout = file;
sxstderr = stdout;
sxsyntax (SXINIT, ...); /* invokes sxopentty() */
```

DEFINITION OF THE scanner_act FUNCTION

The skeleton of the function *scanner_act* defining the effect of the scanner actions is as follows:

```
void scanner_act (ENTRY, ACTION_NUMBER)
int ENTRY;
int ACTION_NUMBER;
{
   switch (ENTRY) {
   case SXOPEN:
   case SXCLOSE:
   case SXINIT:
   case SXFINAL:
      return;
   case SXACTION:
      switch (ACTION_NUMBER) {
      case 1:
          ...
          break;
      case 2:
          ...
          break;
      default:
          /* error message */
          return;
      }
   default:
      /* error message */
      return;
   }
}
```

SEE ALSO

sxunix(3) and the *SYNTAX Reference Manual*.