Ammunition (Reusable packages) - C interface

Vladimir Makarov, vmakarov@gcc.gnu.org Oct 10, 2014

This document describes ammunition (reusable packages written in C/C++).

Contents

1 Introduction

This document describes ammunition (reusable packages written in C/C++). The packages are oriented towards creation of compiler and cross-compiler. Currently there are the following packages (their names

and tasks):

allocate

allocating and freeing memory with fixing some allocation errors

vlobject

work with variable length objects

objstack

work with stacks of objects

hashtab

work with hash tables

position

work with source code positions

errors

output of compiler messages

commline

work with command line

bits

work with bit strings

arithm

implementing host machine-independently arbitrary precision integer numbers arithmetic

IEEE

implementing host machine-independently IEEE floating point arithmetic

ticker

timer

spset

Expandable sparse set

cspset

Compact expandable sparse set

YAEP

Earley parser

2 Package for allocating memory with fixing some allocation errors

Developing modern compilers it is necessary to avoid arbitrary limits on the length or number of any data structure by allocating all data structures dynamically. Here the package 'allocate' which implements allocation and freeing memory with automatic fixing allocation errors is suggested.

The package automatically calls action on situation 'no memory'. The action never returns control back because after calling function which processes allocation error the function 'abort' is always called. Therefore the function which processes allocation error should not return control back.

The interface part of the package is file 'allocate.h'. Defining macro 'NDEBUG' (e.g. by option '-D' in C compiler command line) before the package macros usage disables fixing some internal errors and errors of usage of the package. The implementation part is file 'allocate.c'. The interface contains the following external definitions and macros:

Function 'change_allocation_error_function'

```
'void change_allocation_error_function
      (void (*error_function) (void))) (void)'
```

is used for changing up action on the situation 'no memory'. The function also returns former function which was early action on the situation 'no memory'.

Function 'default_allocation_error_function'

```
'void default_allocation_error_function (void)'
```

is default action of the package on the situation 'no memory'. This action consists of output message '*** no memory ***' to standard error stream and calling function 'exit' with code equals to 1.

Macro 'MALLOC'

```
'MALLOC(ptr, size)'
```

is analogous to ANSI C library function 'malloc'. But the macro has two parameters. The first is pointer variable which is set up by address of allocated memory. The second is size of needed memory.

Macro 'CALLOC'

```
'CALLOC(ptr, nel, size)'
```

is analogous to ANSI C library function 'calloc'. But the macro has three parameters. The first is pointer variable which is set up by address of allocated memory. The last two parameters have the same sense as in standard function 'calloc'.

Macro 'FREE'

```
'FREE(ptr)'
```

is analogous to ANSI C library function 'free' but can accept nil pointer value. In this case macro does nothing.

Macro 'REALLOC'

```
'REALLOC(new, old, size)'
```

is analogous to ANSI C library function 'realloc' but has three parameters. The first parameter is variable which is set up by new address of reallocated memory. The second is old address of reallocated memory. And third is new size of reallocated memory.

3 Package for work with variable length objects

The package 'vlobject' implements work with variable length object (VLO) and uses package 'allocate'. Any number of bytes may be added to and removed from the end of VLO. If it is needed the memory allocated for storing variable length object may be expanded possibly with changing the object place. But between any additions of the bytes (or tailoring) the object place is not changed. To decrease number of changes of the object place the memory being allocated for the object is longer than the current object length.

Because arguments of all macros which return a result ('VLO_LENGTH', 'VLO_BEGIN', 'VLO_BOUND', and 'VLO_END') may be evaluated many times no side-effects should be in the arguments.

The package uses package 'allocate'. The interface part of the package is file 'vlobject.h'. Defining macro 'NDEBUG' (e.g. by option '-D' in C compiler command line) before the package macros usage disables fixing some internal errors and errors of usage of the package. The implementation part is file 'vlobject.c'. The interface contains the following definitions and macros:

Type 'vlo_t'

describes a descriptor of variable length object. All work with variable length object is executed by the following macros through the descriptors. Structure (implementation) of this type is not needed for using variable length object. But it should remember that work with the object through several descriptors is not safe.

Macro 'VLO_DEFAULT_LENGTH'

has value which is default initial size of memory is allocated for VLO when the object is created (with zero initial size). Original value of the macros is equal to 512. This macro can be redefined in C compiler command line or with the aid of directive '#undef' before any using the package macros.

Macro 'VLO_CREATE'

```
'VLO_CREATE(vlo, initial_length)'
```

is used for creation of VLO with initial zero length. Initial memory allocated for VLO whose descriptor is given as the first macro parameter is given as the second parameter. If the second parameter is equal to zero then the initial allocated memory length is equal to 'VLO_DEFAULT_LENGTH'.

'VLO_DELETE'

‘VLO_DELETE(vlo)’

is used for freeing memory used by VLO whose descriptor is given as the macro parameter.

Macro ‘VLO_NULLIFY’

‘VLO_NULLIFY(vlo)’

makes that length of VLO whose descriptor is given as the macro parameter will be equal to zero (but memory for VLO is not freed and not reallocated).

Macro ‘VLO_TAILOR’

‘VLO_TAILOR(vlo)’

makes that length of memory allocated for VLO whose descriptor is given as the macro parameter becomes equal to VLO length.

Macro ‘VLO_LENGTH’

‘VLO_LENGTH(vlo)’

returns current length of VLO whose descriptor is given as the macro parameter.

Macros ‘VLO_BEGIN’, ‘VLO_END’, ‘VLO_BOUND’

‘VLO_BEGIN(vlo)’, ‘VLO_END(vlo)’, ‘VLO_BOUND(vlo)’

return pointer (of type ‘void *’) to correspondingly the first, the last byte of VLO whose descriptor is given as the macros parameter, and pointer to the last byte plus one. Remember that the object may change own place after any addition.

Macro ‘VLO_SHORTEN’

‘VLO_SHORTEN(vlo, n)’

removes n bytes from the end of VLO whose descriptor is given as the first parameter. VLO is nullified if its length is less than n.

Macro ‘VLO_EXPAND’

‘VLO_EXPAND(vlo, length)’

increases length of VLO whose descriptor is given as the first parameter on number of bytes given as the second parameter. The values of bytes added to the end of VLO will be not defined.

Macro ‘VLO_ADD_BYTE’

‘VLO_ADD_BYTE(vlo, b)’

adds byte given as the second parameter to the end of VLO whose descriptor is given as the first parameter.

Macro ‘VLO_ADD_MEMORY’

‘VLO_ADD_MEMORY(vlo, str, length)’

adds memory starting with address given as the second macro parameter and with length given as the third parameter to the end of VLO whose descriptor is given as the first parameter.

Macro ‘VLO_ADD_STRING’

‘VLO_ADD_STRING(vlo, str)’

adds C string (with end marker 0) given as the second macro parameter to the end of VLO whose descriptor is given as the first parameter. Before the addition the macro deletes last character of the VLO. The last character is suggested to be C string end marker 0.

4 Package for work with stacks of objects

The package ‘objstack’ is based on package ‘allocate’ and implements efficient work with stacks of objects (OS). Work with the object on the stack top is analogous to one with a variable length object. One motivation for the package is the problem of growing char strings in symbol tables. Memory for OS is allocated by segments. A segment may contain more one objects. The most recently allocated segment contains object on the top of OS. If there is not sufficient free memory for the top object than new segment is created and the top object is transferred into the new segment, i.e. there is not any memory reallocation. Therefore the top object may change its address. But other objects never change address.

Because arguments of all macros which return a result (‘OS_TOP_LENGTH’, ‘OS_TOP_BEGIN’, ‘OS_TOP_BOUND’, and ‘OS_TOP_END’) may be evaluated many times no side-effects should be in the arguments.

The package uses package ‘allocate’. The interface part of the package is file ‘objstack.h’. Defining macro ‘NDEBUG’ (e.g. by option ‘-D’ in C compiler command line) before the package macros usage disables fixing some internal errors and errors of usage of the package. The implementation part is file ‘objstack.c’. The interface contains the following definitions and macros:

Type ‘os_t’

describes a descriptor of stack of objects. All work with stack of objects is executed by the following macros through the descriptors. Structure (implementation) of this type is not needed for using stack of objects. But it should remember that work with the stack through several descriptors is not safe.

Macro ‘OS_DEFAULT_SEGMENT_LENGTH’

has value which is default size of memory segments which will be allocated for OS when the stack is created (with zero initial segment size). This is also minimal size of all segments. Original value of the

macros is equal to 512. This macro can be redefined in C compiler command line or with the aid of directive '#undef' before any using the package macros.

Macro 'OS_CREATE'

```
'OS_CREATE(os, initial_segment_length)'
```

creates OS which contains the single zero length object. OS descriptor is given as the first macro parameter. Minimum size of memory segments which will be allocated for OS is given as the second parameter. If the second parameter is equal to zero the allocated memory segments length is equal to 'OS_DEFAULT_SEGMENT_LENGTH'. But in any case the segment length is always not less than maximum alignment.

Macro 'OS_DELETE'

```
'OS_DELETE(os)'
```

is used for freeing all memory used by OS whose descriptor is given as the macro parameter.

Macro 'OS_EMPTY'

```
'OS_EMPTY(os)'
```

is used for removing all objects and freeing all memory allocated for OS except for the first segment.

Macro 'OS_TOP_FINISH'

```
'OS_TOP_FINISH(os)'
```

creates new variable length object with initial zero length on the top of OS whose descriptor is given as the macro parameter. The work (analogous to one with variable length object) with object which was on the top of OS is finished, i.e. the object will never more change address.

Macro 'OS_TOP_NULLIFY'

```
'OS_TOP_NULLIFY(os)'
```

makes that length of variable length object on the top of OS whose descriptor is given as the macro parameter will be equal to zero.

Macro 'OS_TOP_LENGTH'

```
'OS_TOP_LENGTH(os)'
```

returns current length of variable length object on the top of OS whose descriptor is given as the macro parameter.

Macros 'OS_TOP_BEGIN', 'OS_TOP_END', 'OS_TOP_BOUND'

```
'OS_TOP_BEGIN(os)', 'OS_TOP_END(os)', 'OS_TOP_BOUND(os)'
```

return pointer to correspondingly the first and the last byte of variable length object on the top of OS whose descriptor is given as the macros parameter, and pointer to the last byte plus one. Remember that the top object may change own place after any addition.

Macro 'OS_TOP_SHORTEN'

```
'OS_TOP_SHORTEN(os, n)'
```

removes n bytes from the end of variable length object on the top of OS whose descriptor is given as the first parameter. The top variable length object is nullified if its length is less than n.

Macro 'OS_TOP_EXPAND'

```
'OS_TOP_EXPAND(os, length)'
```

increases length of variable length object on the top of OS whose descriptor is given as the first parameter on number of bytes given as the second parameter. The values of bytes added to the end of variable length object on the top of OS will be not defined.

Macro 'OS_TOP_ADD_BYTE'

```
'OS_TOP_ADD_BYTE(os, b)'
```

adds byte given as the second parameter to the end of variable length object on the top of OS whose descriptor is given as the first parameter.

Macro 'OS_TOP_ADD_MEMORY'

```
'OS_TOP_ADD_MEMORY(os, str, length)'
```

adds memory starting with address given as the second macro parameter and with length given as the third parameter to the end of variable length object on the top of OS whose descriptor is given as the first parameter.

Macro 'OS_TOP_ADD_STRING'

```
'OS_TOP_ADD_STRING(os, str)'
```

adds C string (with end marker 0) given as the second macro parameter to the end of variable length string on the top of OS whose descriptor is given as the first parameter. Before the addition the macro deletes last character of the top variable length object. The last character is suggested to be C string end marker 0.

5 Package for work with hash tables

The most compilers use search structures. Here the package ‘hashtab’ which implements expandable hash tables is suggested. This abstract data implements features analogous to ones of public domain functions ‘hsearch’, ‘hcreate’ and ‘hdestroy’. The goal of the abstract data creation is to implement additional needed features. The abstract data permits to work simultaneously with several expandable hash tables. Besides insertion and search of elements the elements from the hash tables can be also removed. The table element can be only a pointer. The size of hash tables is not fixed. The hash table will be expanded when its occupancy will become big.

The abstract data implementation is based on generalized Algorithm D from Knuth’s book "The art of computer programming". Hash table is expanded by creation of new hash table and transferring elements from the old table to the new table.

The package uses package ‘allocate’. The interface part of the abstract data is file ‘hashtab.h’. The implementation part is file ‘hashtab.c’. The interface contains the following external definitions:

Type ‘hash_table_entry_t’

is described as ‘void *’ and represents hash table element type. Empty entries have value ‘NULL’.

Type ‘hash_table_t’

describes hash table itself. All work with hash table should be executed only through functions mentioned below.

Function ‘create_hash_table’

```
‘hash_table_t create_hash_table
   (size_t size,
    unsigned (*hash_function) (hash_table_entry_t el_ptr),
    int (*eq_function) (hash_table_entry_t el1_ptr,
                        hash_table_entry_t el2_ptr))’
```

creates and returns hash table with length slightly longer than value of function parameter ‘size’. Created hash table is initiated as empty (all the hash table entries are NULL). The hash table will use functions ‘hash_function’, ‘eq_function’ given as the function parameters to evaluate table element hash value and function to test on equality of two table elements.

Function ‘delete_hash_table’

```
‘void delete_hash_table (hash_table_t htab)’
```

frees memory allocated for hash table given as parameter ‘htab’. Naturally the hash table must already exist.

Function ‘empty_hash_table’

```
‘void empty_hash_table  (hash_table_t htab)’
```

makes hash table given as parameter 'htab' empty. Naturally the hash table must already exist. If you need to remove all table elements, it is better to use this function than several times function 'remove_element_from_hash_table_entry'. This function does not change size of the table or clear statistics about collisions.

Function 'find_hash_table_entry'

```
'hash_table_entry_t *find_hash_table_entry
                     (hash_table_t htab,
                      hash_table_entry_t element,
                      int reserve)'
```

searches for hash table entry which contains element equal to value of the function parameter 'element' or empty entry in which 'element' can be placed (if the element does not exist in the table). The function parameter 'reserve' is to be nonzero if the element is to be placed in the table. The element should be inserted into the table entry before another call of 'find_hash_table_entry'. The table is expanded if occupancy (taking into account also deleted elements) is more than 75%. The occupancy of the table after the expansion will be about 50%.

Function 'remove_element_from_hash_table_entry'

```
'void remove_element_from_hash_table_entry
      (hash_table_t htab, hash_table_entry_t element)'
```

removes element from hash table_entry whose value is given as the function parameter. Hash table entry for given value should be not empty (or deleted). The hash table entry value will be marked as deleted after the function call.

Function 'hash_table_size'

```
'size_t hash_table_size (hash_table_t htab)'
```

returns current size of given hash table.

Function 'hash_table_elements_number'

```
'size_t hash_table_elements_number (hash_table_t htab)'
```

returns current number of elements in given hash table.

Function 'get_searches'

```
'int get_searches (hash_table_t htab)'
```

returns number of searhces during all work with given hash table.

Function 'get_collisions'

```
'int get_collisions (hash_table_t htab)'
```

returns number of occurred collisions during all work with given hash table.

Function 'get_all_searches'

```
'int get_all_searches (void)'
```

returns number of searches during all work with all hash tables.

Function 'get_all_collisions'

```
'int get_all_collisions (void)'
```

returns number of occurred collisions during all work with all hash tables.

6 Package for work with source code positions

The compilers often use internal representation which stores source code positions. Here package 'position' which serves to support information about source positions of compiled files taking into account all included files is suggested.

The strategy of the package usage can be follows. Function 'initiate_positions' is called by the first. After that function 'start_file_position' is called when a file is opened for compilation as source or included file. Members 'line_number' and 'column_number' of variable 'current_position' are modified correspondingly during given file compilation. The value of 'current_position' can be stored into internal representation for usage for output messages on the following passes. Function 'finish_file_position' is called when a processed file is closed. Function 'finish_positions' may be called after all processing a source file.

The package uses packages 'vlobject' and 'objstack' which use package 'allocate'. The interface part of the package is file 'position.h'. The implementation part is file 'position.c'. The interface contains the following external definitions:

Type 'position_t'

is structure which describes a file position. The structure has the following members:

Member 'file_name'

is name of file to which given position belongs.

Members 'line_number', 'column_number'

are source line and column corresponding to given position.

Member 'path'

is pointer to another position structure representing position of include-clause which caused immediately given file compilation.

Variable 'no_position'

has value of type 'position_t' has members with values equals to zero or 'NULL'. The value does not correspond to concrete file position.

Variable 'current_position'

has value which is current file position.

Function 'initiate_positions'

```
'void initiate_positions (void)'
```

initiates the package. Value of variable 'current_position' becomes equal to 'no_position'.

Function 'finish_position'

```
'void finish_positions (void)'
```

frees all memory allocated during package work.

Function 'position_file_inclusion_level'

```
'int position_file_inclusion_level (position_t position)'
```

returns level of inclusion of file of position given as the function parameter. The level numbers are started with zero for positions corresponding non-included files and for positions which does not correspond to concrete file.

Function 'start_file_position'

```
'void start_file_position (const char *file_name)'
```

copys position structure (by dynamic memory allocation) in variable 'current_position' and sets up new values of members of 'current_position'. Values of members 'file_name', 'line_number', 'column_number', and 'path' become equal to the function parameter value, 1, 0, and pointer to the copied structure. Values of 'current_position' during different calls of the function must be different (e.g. different columns or lines), i.e. positions of different include-clauses must be different.

Function 'finish_file_position'

```
'void finish_file_position (void)'
```

recovers previous value of variable 'current_position', more exactly sets up the variable by structure to which the variable member 'path' refers.

Function 'compare_positions'

```
'int compare_positions (position_t position_1,
                        position_t position_2)'
```

compares two positions given by parameters of type 'position_t' and returns -1 (if the first position is less than the second), 0 (if the first position is equal to the second) or 1 (if the first position is greater than the second). The order of positions is lexicographic.

7 Package for output of compiler messages

The most of compilers report error messages for incorrect program. Here the package 'errors' which serves to output one-pass or multi-pass compiler messages of various modes (errors, warnings, fatal, system errors and appended messages) in Unix style or for traditional listing is suggested. The package also permits adequate error reporting for included files.

The package uses packages 'vlobject', 'objstack', 'position' which use package 'allocate'. Therefore package 'position' have to be initiated before any work with this package. The interface part of the package is file 'errors.h'. The implementation part is file 'errors.c'. The maximum length of generated error message is suggested to be not greater then 'MAX_ERROR_MESSAGE_LENGTH'. The default value (150) of this macro can be redefined with corresponding C compiler option '-DMAX_ERROR_MESSAGE_LENGTH=...' during compilation of file 'errors.c'. The interface contains the following external definitions:

Integer variables 'number_of_errors', 'number_of_warnings'

have values which are number of correspondingly errors and warnings fixed after the most recent package initiation.

Integer variable 'maximum_number_of_errors'

has value which is maximum number of errors which will be fixed. If an error is fixed with number equals to 'maximum_number_of_errors' then special fatal error 'fatal error – too many errors' with position of given error is fixed instead of the error. And all following errors are ignored. Zero value of the variable means that the special fatal error will never fixed.

Integer constant 'default_maximum_number_of_errors'

defines originally value of variable 'maximum_number_of_errors'. The constant value is '50'.

Variable 'fatal_error_function'

contains pointer to function without parameters which will be called after fixing a fatal error. The fatal error function is suggested to do not return the control back.

Function 'default_fatal_error_function'

```
'void default_fatal_error_function (void)'
```

defines originally value of variable 'fatal_error_function'. The function only calls 'exit (1)'.

Function 'initiate_errors'

```
'void initiate_errors (int immediate_output_flag)'
```

initiates the package in regime depending on parameter value. If the parameter value is nonzero than all fixed messages are output immediately. Otherwise the compiler messages are stored until function 'output_errors' are called.

Function 'finish_errors'

```
'void finish_errors (void)'
```

frees all memory allocated during package work.

Function 'output_errors'

```
'void output_errors (void)'
```

sorts (stable sorting) all fixed messages by their positions, outputs ones, and deletes ones. Appended messages will be output after corresponding error or warning. This function should be used only in regime of storing messages.

Function 'error'

```
'void error (int fatal_error_flag, position_t
             position, const char *format, ...)'
```

fixes error (fatal error if the first parameter value is nonzero) at the position given as the second parameter. If the error is fatal than functions 'output_errors' and '*fatal_error_function' are called. The diagnostic messages are formed analogous to output of function 'printf'. For example,

```
error (1, current_position, "fatal error - no memory");
```

Function 'warning'

```
'void warning (position_t position, const char *format, ...)'
```

is analogous to the previous but is used to fix a warning.

Function 'append_message'

```
'void append_message (position_t position,
                      const char *format, ...)'
```

When regime of immediately output of fixed message is on this function is analogous to the previous (except for incrementing variable 'number_of_warnings'). In opposite case the appended message will be output with the most recent fixed error or warning independently from value of the first parameter. Of course the previously fixed error or warning must exist.

For example, this function may be used for generation of messages of type

```
'<file>:<line>:<position-1>: repeated declaration'
```

and then

```
'<file>:<line>:<position-1>: previous declaration'.
```

Description of function 'default_output_error_function' contains explanation why decremented position is output.

Function 'system_error'

```
'void system_error (int fatal_error_flag,
                    position_t position,
                    const char *format, ...)'
```

is analogous to function 'error' but serves to fix a system error. The current system message without head blanks (given by standard function 'strerror') is placed after the message formed by the function parameters. For example, the following call may be used when a file is not opened

```
system_error  (1, current_position,
               "fatal error - %s:", new_file_name);
```

Variable 'output_error_function'

contains pointer to function which is used to output error message. The function has three parameters – flag of appended message, message position and message itself.

Function 'default_output_error_function'

```
'void default_output_error_function
      (int appended_message_flag, position_t position,
       const char *message)'
```

Originally value of variable 'output_error_function' is equal to this function. The function is oriented to output in Unix style according to GNU standard. To output a listing the value of variable 'output_error_function' should be changed. The function output message in the following formats:

```
MESSAGE                                (NULL file name)
FILE_NAME:1: MESSAGE                   (zero line number)
FILE_NAME:LINE_NUMBER: MESSAGE         (zero column number)
FILE_NAME:LINE_NUMBER:COLUMN_NUMBER: MESSAGE  (all other cases)
```

After that the function outputs newline. The function also outputs additional messages 'in file processed from ...' if given message is not appended message and corresponds to file different from one of previous output error. This message reflects path of the message position (see package 'position'), i.e. reflects positions of corresponding include-clauses.

8 Package for work with command line

To make easy process of command line, here abstract data 'commline' is suggested. This abstract data implements features analogous to ones of public domain function 'getopt'. The goal of the abstract data creation is to use more readable language of command line description and to use command line description as help output of program.

POSIX terminology concerning command line is used here. Command line is divided into command name and arguments. The arguments are subdivided into options, option-arguments and operands. Option starts with '-'. All arguments after first '–' in command line are treated as operands.

The description of command line is made up from two parts. Any part (or both) may be absent in the description. First part contains option-arguments names of options which are in the second part. option-arguments names are separated by white space. The second part starts with percents '%%' and contains any text in which description of options are placed. Any description of option starts with character "' followed by character '-' and finishes by character "'. White spaces may precede option-argument name. It means that the corresponding option has obligatory separate option-argument. For example, the following may be a part of description of options of a pascal compiler command line.

```
dir xxx file
%%
command line:  pc [options] file ...
                                Options:
'-0'      Pascal standard level 0    '-1'      Pascal standard Level 1.
'-29000'  Am29000 code generation    '-29050'* Am29050 code generation
'-c'      only object files creation '-el'     output of listing
'-g'      information for debuggers  '-Idir'   data task units directory
'-lxxx'   library                    '-Ldir'   libraries directory
'-o file' output file                '-O'      all optimizations
'-S'      only ass. code creation    '-v'      loaded processes indication
'-w'      no warnings generation
               Star * marks defaults
```

In this example options with names '-I', '-l', '-L' and '-o' have option-arguments but only option with name '-o' has separate option-argument, i.e. option-argument which is represented by separate argument after given option in command line.

The interface part of the abstract data is file 'commline.h'. The package uses package 'vlobject' which use package 'allocate'. The implementation part is file 'commline.c'. The interface contains the following external definitions:

Function

```
'int start_command_line_processing
     (int argc, char **argv, const char *description)'
```

must be called before any work with abstract data. The function processes command line description given as string parameter and command line itself given as two parameter 'argc' and 'argv'. The function also initiates variables 'argument_vector' and 'argument_count' by parameters 'argc' and 'argv'. The function returns 0 if error in command line description is fixed, otherwise returns 1 (it means success).

Function 'output_command_line_description'

```
'void output_command_line_description (void)'
```

outputs the second part (without '%%') of description of options to stderr. This function should be called when it is necessary to show the program usage.

Function 'next_operand'

```
'int next_operand (int flag_of_first)'
```

returns command line argument number of next operand if the function parameter is nonzero. Otherwise the function returns number of the first operand in the command line. The function returns 0 if all operands are already processed. Returned number may be used as index of array 'argument_vector' to access corresponding operand.

Function 'number_of_operands'

```
'int number_of_operands (void)'
```

returns number of operands in the command line.

Function 'next_option'

```
'int next_option (int flag_of_first)'
```

returns command line argument number of next option if the function parameter is nonzero. Otherwise the function returns number of the first option in the command line. The function returns 0 if all options are already processed. Returned number may be used as index of array 'argument_vector' to access corresponding option.

Function 'option_characteristics'

```
'char *option_characteristics (int argument_number,
                               int *option_has_argument)'
```

returns pointer to option name which describes the command line argument with number 'argument_number' given as the first parameter of the function. The function returns NULL if the corresponding option in the command line description is not found or an option described as with option-argument has not option-argument in the command line. Remember that option name with option-argument differs from option in the command line (e.g. '-U' and '-Ufoo'). If the option in the command line description is found than the function sets up correspondingly the second function parameter 'option_has_argument'. The case of returned NULL and '*option_has_argument' equals to TRUE means that given option must have option-argument but the option has not option-argument in the command line.

Function 'last_option_place'

```
'int last_option_place (const char *option_name)'
```

returns number of last option with given option name in the command line. The function returns 0 if the option is absent in the command line.

Function 'option_argument'

```
'char *option_argument (const char *option_name)'
```

returns pointer to argument of last option in the command line with given option name. The function returns NULL if the option is absent in the command line. The function must be called only for options which have argument separated by white spaces.

Variables 'argument_count', 'argument_vector'

have analogous values as parameters 'argc' and 'argv' of function 'main'. See also description of 'start_command_line_processing'.

9 Package for work with bit strings

The package for work with bit strings is used to implement package 'IEEE'. But of course the package can be used for solving other tasks.

Here a bit is given by address (start address) of byte from which counting bits starts and its displacement which is any non negative number of bit from the start address. The most significant bit of the start address byte has number 0. The bit string is given by its first bit and its length in bits.

The interface part of the package is file 'bits.h'. The implementation part is file 'bits.c'. The interface contains the following external definitions of macros and functions:

Macro 'BIT'

```
'BIT(start_byte, bit_displacement)'
```

returns given bit value as integer value '0' or '1'. There is macro 'SET_BIT (start_byte, bit_displacement, bit) for changing value of a bit. Parameter 'bit' must have value '0' or '1'.

Function 'is_zero_bit_string'

```
'int is_zero_bit_string (const void *start_byte,
                         int bit_displacement,
                         int bit_length)'
```

returns '1' if given bit string contains only zero value bits, 0 otherwise.

Function 'bit_string_set'

```
'void bit_string_set (void *start_byte, int
                      bit_displacement, int bit,
                      int bit_length)'
```

sets up new value of all bits of given bit string. This function is bit string analog of standard C function 'memset'.

Function 'bit_string_copy'

```
'void bit_string_copy (void *to, int to_bit_displacement,
                       const void *from,
                       int from_bit_displacement,
                       int bit_length)'
```

copys a bit string to another bit string. The bit strings must be nonoverlapped. This function is bit string analog of standard C function 'memcpy'.

Function 'bit_string_move'

```
void bit_string_move (void *to, int to_bit_displacement,
                      const void *from,
                      int from_bit_displacement,
                      int bit_length)'
```

copys a bit string to another bit string. The bit strings can be overlapped. This function is bit string analog of standard C function 'memmove'.

Function 'bit_string_comparison'

```
'int bit_string_comparison
     (const void *str1, int bit_displacement1,
      const void *str2, int bit_displacement2,
      int bit_length)'
```

returns 0 if the bit strings are equal, 1 if the first bit string is greater than the second, -1 if the first bit string is less than the second. This function is bit string analog of standard C function 'memcmp'.

10 Package for machine-independent arbitrary precision integer arithmetic

Abstract data 'arithm' may be used for implementation of a cross-compiler. This abstract data implements arbitrary precision integer and unsigned integer number arithmetic by machine independent way. The implementation of the package functions are not sufficiently efficient in order to use for run-time. The package functions are oriented to implement constant-folding in compilers. This package is necessary because host machine may not support such arithmetic for target machine. For example, VAX does not support does not support more 32-bits integer numbers arithmetic.

The numbers in packages are represented by bytes in big endian mode, negative integer numbers are represented in complementary code. All sizes are given in bytes and must be positive. Results of executions of all functions can coincide with a operand(s). All functions of addition, subtraction, multiplication, division, evaluation of remainder, shift, changing size and transformation of string into number fix overflow. The overflow is fixed when result can not be represented by number of given size.

The interface part of the abstract data is file 'arithm.h'. The implementation part is file 'arithm.c'. The maximum length of integer number is suggested to be not greater then 'MAX_INTEGER_OPERAND_SIZE'.

The default value (128) of this macro can be redefined with corresponding C compiler option '-DMAX_INTEGER_OPERAND_SIZE=...' during compilation of file 'arithm.c'. But in any case the minimal value of the macros will be 16. The interface contains the following external definitions:

Variable 'overflow_bit'

has only two values 0 or 1. The value '1' corresponds to overflow. The variable value are modified by all functions of addition, subtract, multiplication, division, evaluation of remainder, shift, changing size and transformation of string into number.

Variable 'const unsigned char *zero_constant'

represents zero (unsigned) integer of any size.

Function 'void default_arithmetic_overflow_reaction (void)'

Originally reaction on all integer and unsigned integer overflow is equal to this function. The function does nothing. Reaction on overflow for integers or unsigned integers is called after setting up variable 'overflow_bit'.

Function 'set_integer_overflow_reaction'

```
'void (*set_integer_overflow_reaction
        (void (*function) (void))) (void)'
```

changes reaction on integer overflow and returns previous overflow reaction function. There is analogous function

```
'set_unsigned_integer_overflow_reaction'
```

for unsigned integer overflow.

Function 'integer_maximum'

```
'void integer_maximum   (int  size,  void  *result)'
```

creates given size (in bytes) maximal integer constant which is placed in memory whose address is given by the second parameter. There is analogous function

```
'integer_minimum' and
'unsigned_integer_maximum'.
```

Function 'add_integer'

```
'void add_integer (int size,
                   const void *op1, const void *op2,
                   void *result)'
```

makes integer addition of integers of given size. The function fixes overflow when result can not be represented by number of given size. There are analogous functions which implement other integer operations:

‘subtract_integer’,
‘multiply_integer’,
‘divide_integer’,
‘integer_remainder’.

Also there are analogous functions

‘subtract_unsigned_integer’,
‘multiply_unsigned_integer’,
‘divide_unsigned_integer’,
‘unsigned_integer_remainder’

for unsigned integers.

Function ‘integer_shift_left’

```
‘void  integer_shift_left (int size,  const void *operand,
                           int bits, void *result)’
```

makes left shift of integer of given size on given number of bits. If number of bits is negative the function makes shift to right actually. The function fixes overflow when result can not be represented by number of given size, i.e. in other words the opposite shift (to right) results in number not equal to source operand. There are analogous functions which implement another integer operation

‘integer_shift_right’.

Also there are analogous functions

‘unsigned_integer_shift_left’ and
‘unsigned_integer_shift_right’ for unsigned integers.

Function ‘integer_or’

```
‘void integer_or (int size, const void *op1, const void *op2,
                  void *result)’
```

makes bitwise ‘or’ of integers of given size. There is analogous functions which implement bitwise ‘and’ or ‘xor’:

‘integer_and’,
‘integer_xor’.

Also there are equivalent functions

‘unsigned_integer_or’,
‘unsigned_integer_and’,
‘unsigned_integer_xor’,

for unsigned integers.

Function

```
'void integer_not (int size, const void *operand,
                   void *result)'
```

makes bitwise 'not' of integer of given size. There is equivalent function for unsigned integer:

'unsigned_integer_not'.

Function 'eq_integer'

'int eq_integer (int size, const void *op1, const void *op2)'

compares two integers of given size on equality and returns 1 or 0 depending on result of the comparison. There are analogous functions which implement other integer operations:

'ne_integer',
'gt_integer',
'lt_integer',
'ge_integer',
and 'le_integer'.

Also there are analogous functions

'eq_unsigned_integer',
'ne_unsigned_integer',
'gt_unsigned_integer',
'lt_unsigned_integer',
'ge_unsigned_integer',
and 'le_unsigned_integer'

for unsigned integers.

Function 'change_integer_size'

```
'void  change_integer_size (int operand_size,
                            const  void *operand,
                            int result_size, void *result)'
```

changes size of integer. The function fixes overflow when result can not be represented by number of given size. There is analogous function

'change_unsigned_integer_size'

for unsigned integers.

Function 'integer_to_string'

```
'char *integer_to_string (int  size, const void *operand,
                          char *result)'
```

transforms integer of given size to decimal ascii representation. Sign is present in result string only for negative numbers. The function returns value 'result'. There is analogous function

```
'unsigned_integer_to_string'
```

for unsigned integers.

Function 'integer_to_based_string'

```
'char *integer_to_based_string (int  size, const void *operand,
                                int base, char *result)'
```

transforms integer of given size to ascii representation with given base. The base should be between 2 and 36 including them. Digits more 9 are represented by 'a', 'b' etc. Sign is present in result string only for negative numbers. The function returns value 'result'. There is analogous function

```
'unsigned_integer_to_based_string'
```

for unsigned integers.

Function 'integer_from_string'

```
'char *integer_from_string (int  size, const char *operand,
                            void *result)'
```

skips all white spaces at the begin of source string and transforms the tail of the source string (decimal ascii representation with possible sign '+' or '-') to given size integer and returns pointer to first non digit in the source string. If the string started with invalid integer representation the result will be zero. The function fixes overflow when result can not be represented by number of given size. There is analogous function

```
'unsigned_integer_from_string'
```

for unsigned integers. But sign '+' is believed to be not part of unsigned integer.

Function 'integer_from_based_string'

```
'char *integer_from_based_string (int  size, const char *operand,
                                  int base, void *result)'
```

skips all white spaces at the begin of source string and transforms the tail of the source string (ascii representation with given base and with possible sign '+' or '-') to given size integer and returns pointer to first non digit in the source string. The base should be between 2 and 36 including them. Digits more 9 are represented by 'a' (or 'A'), 'b' (or 'B') etc. If the string started with invalid integer representation the result will be zero. The function fixes overflow when result can not be represented by number of given size. There is analogous function

```
'unsigned_integer_from_based_string'
```

for unsigned integers. But sign '+' is believed to be not part of unsigned integer.

11 Package for machine-independent IEEE floating point arithmetic

Abstract data 'IEEE' may be used for implementation of a cross-compiler. This abstract data implements IEEE floating point arithmetic by machine independent way with the aid of package 'arithm'. This abstract data is necessary because host machine may not support such arithmetic for target machine. For example, VAX does not support IEEE floating point arithmetic. The floating point numbers are represented by bytes in big endian mode. The implementation of the package functions are not sufficiently efficient in order to use for run-time. The package functions are oriented to implement constant-folding in compilers. All integer sizes (see transformation functions) are given in bytes and must be positive.

Functions of addition, subtraction, multiplication, division, conversion floating point numbers of different formats can fix input exceptions. If an operand of such operation is trapping (signal) not a number then invalid operation and reserved operand exceptions are fixed and the result is (quiet) NaN, otherwise if an operand is (quiet) NaN then only reserved operand exception is fixed and the result is (quiet) NaN. Operation specific processing the rest of special case values of operands is placed with description of the operation. In general case the function can fix output exceptions and produces results for exception according to the following table. The result and status for a given exceptional operation are determined by the highest priority exception. If, for example, an operation produces both overflow and imprecise result exceptions, the overflow exception, having higher priority, determines the behavior of the operation. The behavior of this operation is therefore described by the Overflow entry of the table.

Exception	Condition		Result	Status
Overflow	masked overflow exception	sign + IEEE_RN(_RP)	+Inf	IEEE_OFL and IEEE_IMP
		sign + IEEE_RZ(_RM)	+Max	
		sign - IEEE_RN(_RM)	-Inf	
		sign - IEEE_RZ(_RP)	-Max	
	unmasked overflow exception	Precise result	See above	IEEE_OFL
		Imprecise result		IEEE_OFL and IEEE_IMP
Underflow	masked underflow exception	Imprecise result	Rounded result	IEEE_UFL and IEEE_IMP
	unmasked underflow exception	Precise result	result	IEEE_UFL
		Imprecise result	Rounded result	IEEE_UFL and IEEE_IMP

```
-----------|------------------------------|-------|-------------
           |masked imprecise exception    |Rounded|IEEE_IMP
 Imprecise |                              |result |
           |------------------------------|-------|-------------
           |unmasked imprecise exception  |Rounded|IEEE_IMP
           |                              |result |
```

The package uses package 'bits'. The interface part of the abstract data is file 'IEEE.h'. The implementation part is file 'IEEE.c'. The interface contains the following external definitions:

Macros 'IEEE_FLOAT_SIZE', 'IEEE_DOUBLE_SIZE', 'IEEE_QUAD_SIZE'

have values which are sizes of IEEE single, double, and quad precision floating point numbers ('4', '8', and '16' correspondingly).

Macros 'MAX_SINGLE_10_STRING_LENGTH', 'MAX_DOUBLE_10_STRING_LENGTH', 'MAX_

have values which are maximal length of string generated by functions creating decimal ascii representation of IEEE floats (see functions IEEE_single_to_string, IEEE_doublele_to_string, and IEEE_quad_to_string).

Macros 'MAX_SINGLE_16_STRING_LENGTH', 'MAX_DOUBLE_16_STRING_LENGTH', 'MAX_

have values which are maximal length of string generated by functions creating binary ascii representation of IEEE floats with given base (see functions IEEE_single_to_binary_string, IEEE_doublele_to_binary_string, and IEEE_quad_to_binary_string).

Types 'IEEE_float_t', 'IEEE_double_t', and 'IEEE_quad_t'

represent correspondingly IEEE single precision, double, and quad precision floating point numbers. The size of these type are equal to 'IEEE_FLOAT_SIZE', 'IEEE_DOUBLE_SIZE', and 'IEEE_QUAD_SIZE'.

Function 'IEEE_reset'

```
'void IEEE_reset (void)'
```

and to separate bits in mask returned by functions

```
'IEEE_get_sticky_status_bits',
'IEEE_get_status_bits', and
'IEEE_get_trap_mask'.
```

Function 'IEEE_get_trap_mask'

```
'int IEEE_get_trap_mask (void)'
```

returns exceptions trap mask. Function

```
'int IEEE_set_trap_mask (int mask)'
```

sets up new exception trap mask and returns the previous.

If the mask bit corresponding given exception is set, a floating point exception trap does not occur for given exception. Such exception is said to be masked exception. Initial exception trap mask is zero. Remember that more one exception may be occurred simultaneously.

Function 'IEEE_set_sticky_status_bits'

```
'int IEEE_set_sticky_status_bits (int mask)'
```

changes sticky status bits and returns the previous bits.

Function

```
'int IEEE_get_sticky_status_bits (void)'
```

returns mask of current sticky status bits. Only sticky status bits corresponding to masked exceptions are updated regardless whether a floating point exception trap is taken or not. Initial values of sticky status bits are zero.

Function 'IEEE_get_status_bits'

```
'int IEEE_get_status_bits (void)'
```

returns mask of status bits. It is supposed that the function will be used in trap on an floating point exception. Status bits are updated regardless of the current exception trap mask only when a floating point exception trap is taken. Initial values of status bits are zero.

Constants 'IEEE_RN', 'IEEE_RM', 'IEEE_RP', 'IEEE_RZ'

defines rounding control (round to nearest representable number, round toward minus infinity, round toward plus infinity, round toward zero).

Round to nearest means the result produced is the representable value nearest to the infinitely-precise result. There are special cases when infinitely precise result falls exactly halfway between two representable values. In this cases the result will be whichever of those two representable values has a fractional part whose least significant bit is zero.

Round toward minus infinity means the result produced is the representable value closest to but no greater than the infinitely precise result.

Round toward plus infinity means the result produced is the representable value closest to but no less than the infinitely precise result.

Round toward zero, i.e. the result produced is the representable value closest to but no greater in magnitude than the infinitely precise result. There are two functions

```
'int IEEE_set_round (int round_mode)'
```

which sets up current rounding mode and returns previous mode and

```
'int IEEE_get_round (void)'
```

which returns current mode. Initial rounding mode is round to nearest.

Function 'default_floating_point_exception_trap'

```
'void default_floating_point_exception_trap (void)'
```

Originally reaction on occurred trap on an unmasked floating point exception is equal to this function. The function does nothing. All occurred exceptions can be found in the trap with the aid of status bits.

Function 'IEEE_set_floating_point_exception_trap'

```
'void (*IEEE_set_floating_point_exception_trap
        (void (*function) (void))) (void)'
```

sets up trap on an unmasked exception. Function given as parameter simulates floating point exception trap.

Function 'IEEE_positive_zero'

```
'IEEE_float_t IEEE_positive_zero (void)'
```

returns positive single precision zero constant. There are analogous functions which return other special case values:

```
'IEEE_negative_zero',
'IEEE_NaN',
'IEEE_trapping_NaN',
'IEEE_positive_infinity',
'IEEE_negative_infinity',
'IEEE_double_positive_zero',
'IEEE_double_negative_zero',
'IEEE_double_NaN',
'IEEE_double_trapping_NaN',
'IEEE_double_positive_infinity',
'IEEE_double_negative_infinity'.
'IEEE_quad_positive_zero',
'IEEE_quad_negative_zero',
'IEEE_quad_NaN',
'IEEE_quad_trapping_NaN',
'IEEE_quad_positive_infinity',
'IEEE_quad_negative_infinity'.
```

According to the IEEE standard NaN (and trapping NaN) can be represented by more one bit string. But all functions of the package generate and use only one its representation created by function 'IEEE_NaN' (and 'IEEE_trapping_NaN', 'IEEE_double_NaN', 'IEEE_double_trapping_NaN', 'IEEE_quad_NaN', 'IEEE_quad_trapping_NaN'). A (quiet) NaN does not cause an Invalid Operation exception and can be reported as an operation result. A trapping NaN causes an Invalid Operation exception if used as in input operand to floating point operation. Trapping NaN can not be reported as an operation result.

Function 'IEEE_is_positive_zero'

```
'int IEEE_is_positive_zero (IEEE_float single_float)'
```

returns 1 if given number is positive single precision zero constant. There are analogous functions for other special case values:

```
'IEEE_is_negative_zero',
'IEEE_is_NaN',
'IEEE_is_trapping_NaN',
'IEEE_is_positive_infinity',
'IEEE_is_negative_infinity',
'IEEE_is_positive_maximum' (positive max value),
'IEEE_is_negative_maximum',
'IEEE_is_positive_minimum' (positive min value),
'IEEE_is_negative_minimum',
'IEEE_is_double_positive_zero',
'IEEE_is_double_negative_zero',
'IEEE_is_double_NaN',
'IEEE_is_double_trapping_NaN',
'IEEE_is_double_positive_infinity',
'IEEE_is_double_negative_infinity',
'IEEE_is_double_positive_maximum',
'IEEE_is_double_negative_maximum',
'IEEE_is_double_positive_minimum',
'IEEE_is_double_negative_minimum'.
'IEEE_is_quad_positive_zero',
'IEEE_is_quad_negative_zero',
'IEEE_is_quad_NaN',
'IEEE_is_quad_trapping_NaN',
'IEEE_is_quad_positive_infinity',
'IEEE_is_quad_negative_infinity',
'IEEE_is_quad_positive_maximum',
'IEEE_is_quad_negative_maximum',
'IEEE_is_quad_positive_minimum',
'IEEE_is_quad_negative_minimum'.
```

In spite of that all functions of the package generate and use only one its representation created by function 'IEEE_NaN' (or 'IEEE_trapping_NaN', or 'IEEE_double_NaN', or 'IEEE_double_trapping_NaN', or 'IEEE_quad_NaN', or 'IEEE_quad_trapping_NaN'). The function 'IEEE_is_NaN' (and 'IEEE_trapping_NaN', and 'IEEE_double_NaN', and 'IEEE_double_trapping_NaN', and 'IEEE_quad_NaN', and 'IEEE_quad_trapping_NaN') determines any representation of NaN.

Function 'IEEE_is_normalized'

```
'int IEEE_is_normalized (IEEE_float_t single_float)'
```

returns TRUE if single precision number is normalized (special case values are not normalized). There is analogous function

‘IEEE_is_denormalized’

for determination of denormalized number. There are analogous functions

‘IEEE_is_double_normalized’ and
‘IEEE_is_double_denormalized’ and
‘IEEE_is_quad_normalized’ and
‘IEEE_is_quad_denormalized’

for doubles and quads.

Function ‘IEEE_add_single’

‘IEEE_float_t IEEE_add_single (IEEE_float_t single1,
IEEE_float_t single2)’

makes single precision addition of floating point numbers. There are analogous functions which implement other floating point operations:

‘IEEE_subtract_single’,
‘IEEE_multiply_single’,
‘IEEE_divide_single’,
‘IEEE_add_double’,
‘IEEE_subtract_double’,
‘IEEE_multiply_double’,
‘IEEE_divide_double’.
‘IEEE_add_quad’,
‘IEEE_subtract_quad’,
‘IEEE_multiply_quad’,
‘IEEE_divide_quad’.

Results and input exceptions for operands of special cases values (except for NaNs) are described for addition by the following table

first operand	second operand		
	+Inf	-Inf	Others
+Inf	+Inf none	NaN IEEE_INV(_RO)	+Inf none
-Inf	NaN IEEE_INV(_RO)	-Inf none	-Inf none
Others	+Inf none	-Inf none	

Results and input exceptions for operands of special cases values (except for NaNs) are described for subtraction by the following table

first operand	second operand: +Inf	-Inf	Others
+Inf	NaN IEEE_INV(_RO)	+Inf none	+Inf none
-Inf	-Inf none	NaN IEEE_INV(_RO)	-Inf none
Others	-Inf none	+Inf none	

Results and input exceptions for operands of special cases values (except for NaNs) are described for multiplication by the following table

first operand	second operand: +Inf	-Inf	0	Others
+Inf	+Inf none	-Inf none	NaN IEEE_INV(_RO)	(+-)Inf none
-Inf	-Inf none	+Inf none	NaN IEEE_INV(_RO)	(+-)Inf none
0	NaN IEEE_INV(_RO)	NaN IEEE_INV(_RO)	(+-)0 none	(+-)0 none
Others	(+-)Inf none	(+-)Inf none	(+-)0 none	

Results and input exceptions for operands of special cases values (except for NaNs) are described for division by the following table

first operand	second operand: +Inf	-Inf	0	Others
+Inf	NaN IEEE_INV(_RO)	NaN IEEE_INV(_RO)	(+-)Inf none	(+-)Inf none
-Inf	NaN IEEE_INV(_RO)	NaN IEEE_INV(_RO)	(+-)Inf none	(+-)Inf none
0	(+-)0 none	(+-)0 none	NaN IEEE_INV(_RO)	(+-)0 none
Others	(+-)0 none	(+-)0 none	(+-)Inf IEEE_DZ	

Function 'IEEE_eq_single'

```
‘int IEEE_eq_single (IEEE_float_t single1,
                     IEEE_float_t single2)’
```

compares two single precision floating point numbers on equality and returns 1 or 0 depending on result of the comparison. There are analogous functions which implement other integer operations:

```
‘IEEE_ne_single’,
‘IEEE_gt_single’,
‘IEEE_lt_single’,
‘IEEE_ge_single’,
‘IEEE_le_single’,
‘IEEE_eq_double’,
‘IEEE_ne_double’,
‘IEEE_gt_double’,
‘IEEE_lt_double’,
‘IEEE_ge_double’,
‘IEEE_le_double’.
‘IEEE_eq_quad’,
‘IEEE_ne_quad’,
‘IEEE_gt_quad’,
‘IEEE_lt_quad’,
‘IEEE_ge_quad’,
‘IEEE_le_quad’.
```

Results and input exceptions for operands of special cases values are described for equality and inequality by the following table

first operand	second operand: SNaN	second operand: QNaN	second operand: Others
SNaN	FALSE IEEE_INV	FALSE IEEE_INV	FALSE IEEE_INV
QNaN	FALSE IEEE_INV	FALSE none	FALSE none
Others	FALSE IEEE_INV	FALSE none	

Results and input exceptions for operands of special cases values are described for other comparison operation by the following table

first operand	second operand: SNaN	second operand: QNaN	second operand: Others
SNaN	FALSE IEEE_INV	FALSE IEEE_INV	FALSE IEEE_INV

```
-------|-------------|--------------|----------
QNaN   |   FALSE     |   FALSE      |  FALSE
       |  IEEE_INV   |  IEEE_INV    | IEEE_INV
-------|-------------|--------------|----------
Others |   FALSE     |   FALSE      |
       |  IEEE_INV   |  IEEE_INV    |
```

Transformation functions

```
'IEEE_double_t IEEE_single_to_double
               (IEEE_float_t single_float)',

'IEEE_float_t IEEE_double_to_single
              (IEEE_double_t double_float)',

'IEEE_quad_t IEEE_single_to_quad
             (IEEE_float_t single_float)',

'IEEE_float_t IEEE_quad_to_single
              (IEEE_quad_t quad_float)',

'IEEE_quad_t IEEE_double_to_quad
             (IEEE_double_t double_float)',

'IEEE_double_t IEEE_quad_to_double
              (IEEE_quad_t quad_float)',

'IEEE_float_t IEEE_single_from_integer
              (int size, const void *integer)',

'IEEE_float_t IEEE_single_from_unsigned_integer
              (int size, const void *unsigned_integer)',

'IEEE_double_t IEEE_double_from_integer
               (int size, const void *integer)',

'IEEE_double_t IEEE_double_from_unsigned_integer
               (int size, const void *unsigned_integer)',

'IEEE_quad_t IEEE_quad_from_integer
             (int size, const void *integer)',

'IEEE_quad_t IEEE_quad_from_unsigned_integer
             (int size, const void *unsigned_integer)',

'void IEEE_single_to_integer
      (int size, IEEE_float_t single_float, void *integer)',

'void IEEE_single_to_unsigned_integer
      (int size, IEEE_float_t single_float,
       void *unsigned_integer)',
```

```
‘void IEEE_double_to_integer
      (int size, IEEE_double_t double_float, void *integer)’,

‘void IEEE_double_to_unsigned_integer
      (int size, IEEE_double_t double_float,
       void *unsigned_integer)’.

‘void IEEE_quad_to_integer
      (int size, IEEE_quad_t quad_float, void *integer)’,

‘void IEEE_quad_to_unsigned_integer
      (int size, IEEE_quad_t quad_float,
       void *unsigned_integer)’.
```

Actually no one output exceptions occur during transformation of single precision floating point number to double and quad precision number or of double precision floating point number to quad precision number. No input exceptions occur during transformation of integer numbers to floating point numbers. Results and input exceptions for operand of special cases values (and for NaNs) are described for conversion floating point number to integer by the following table

Operand	Result & Exception
SNaN	0 IEEE_INV(_RO)
QNaN	0 IEEE_INV(_RO)
+Inf	IMax IEEE_INV
-Inf	IMin IEEE_INV
Others	

Results and input exceptions for operand of special cases values (and for NaNs) are described for conversion floating point number to unsigned integer by the following table

Operand	Result & Exception
SNaN	0 IEEE_INV(_RO)
QNaN	0 IEEE_INV(_RO)
+Inf	IMax IEEE_INV

```
     -Inf or   |    0
negative number|  IEEE_INV
 --------------|-------------------
     Others    |
               |
```

Results and exceptions for NaNs during transformation of floating point numbers to (unsigned) integers are differed from the ones for operations of addition, multiplication and so on.

Function 'IEEE_single_to_string'

```
'char *IEEE_single_to_string (IEEE_float_t single_float,
                              char *result)'
```

transforms single precision to decimal ascii representation with obligatory integer part (1 digit), fractional part (of constant length), and optional exponent. Signs minus are present if it is needed. The special cases IEEE floating point values are represented by strings 'SNaN', 'QNaN', '+Inf', '-Inf', '+0', and '-0'. The function returns value 'result'. There are analogous functions

```
'IEEE_string_to_double'
'IEEE_string_to_quad'
```

for doubles and quads. Current round mode does not affect the resultant ascii representation. The function outputs 9 decimal fraction digits for single precision number, 17 decimal fraction digits for double precision number, and 36 for quad precision number.

Function 'IEEE_single_to_binary_string'

```
'char *IEEE_single_to_binary_string (IEEE_float_t single_float,
                                     int base, char *result)'
```

The function is analogous to IEEE_single_to_string but transforms float number into to binary ascii representation with obligatory integer part (1 digit) of given base, optional fractional part of given base, and optional binary exponent (decimal number giving power of 2). The binary exponent starts with character 'p' instead of 'e'. Signs minus are present if it is needed. The special cases IEEE floating point values are represented by strings 'SNaN', 'QNaN', '+Inf', '-Inf', '+0', and '-0'. The function returns value 'result'. Value of parameter base should be 2, 4, 8, or 16. There are analogous functions

```
'IEEE_string_to_binary_double'
'IEEE_string_to_binary_quad'
```

for doubles and quads. Current round mode does not affect the resultant ascii representation.

Function 'IEEE_single_from_string'

```
'char *IEEE_single_from_string (const char *operand,
                                IEEE_float_t *result)'
```

skips all white spaces at the begin of source string and transforms tail of the source string to single precision floating point number. The number must correspond the following syntax

```
['+' | '-'] [<decimal digits>] [ '.' [<decimal digits>] ]
     [ ('e' | 'E') ['+' | '-'] <decimal digits>]
```

or must be the following strings 'SNaN', 'QNaN', '+Inf', '-Inf', '+0', or '-0'. The function returns pointer to first character in the source string after read floating point number. If the string does not correspond floating point number syntax the result will be zero and function returns the source string.

The function can fix output exceptions as described above. There are analogous functions

```
'IEEE_double_from_string'
'IEEE_quad_from_string'
```

for doubles and quads. Current round mode may affect resultant floating point number. It is guaranteed that transformation 'IEEE floating point number -> string -> IEEE floating point number' results in the same IEEE floating point number if round to nearest mode is used. But the reverse transformation 'string with 9 (or 17 or 36) digits -> IEEE floating point number -> string' may results in different digits of the fractions in ascii representation because a floating point number may represent several such strings with differences in the least significant digit. But the ascii representations are identical when functions 'IEEE_single_from_string', 'IEEE_double_from_string', 'IEEE_quad_from_string' do not fix imprecise result exception or less than 9 (or 17 or 36) digits of the fractions in the ascii representations are compared.

Function 'IEEE_single_from_binary_string'

```
'char *IEEE_single_from_binary_string (const char *operand,
                                       int base,
                                       IEEE_float_t *result)'
```

The function is analogous to IEEE_single_to_string but transforms binary representation of the single precision floating point number. The number must correspond the following syntax

```
['+' | '-'] [<digits less base>] [ '.' [<digits less base>] ]
     [ ('p' | 'P') ['+' | '-'] <decimal digits>]
```

or must be the following strings 'SNaN', 'QNaN', '+Inf', '-Inf', '+0', or '-0'. The function returns pointer to first character in the source string after read floating point number. If the string does not correspond floating point number syntax the result will be zero and function returns the source string. The exponent (after character 'p' or 'P') defines power of two.

The function can fix output exceptions as described above. There are analogous functions

```
'IEEE_double_from_binary_string'
'IEEE_quad_from_binary_string'
```

for doubles and quads. Current round mode can affect resultant floating point number if there are too many given digits.

Important note: All items (they contains word quad or QUAD in their names) relative to IEEE 128 bits floating point numbers are defined only when macro 'IEEE_QUAD' is defined. By default 'IEEE_QUAD' is not defined. It is made because supporting IEEE 18-bits numbers requires more 100Kb memory.

12 Ticker package

The package 'ticker' implements a timer. Timer can be activated or can be stopped. The timer accumulates execution time only when it is in active state. The interface part of the package is file 'ticker.h'. The implementation part is file 'ticker.c'. The interface contains the following external definitions and macros:

Type 'ticker_t'

describes a timer. Knowledge of structure (implementation) of this type is not needed for using timer.

Function 'create_ticker'

```
'ticker_t create_ticker (void)'
```

returns new timer. The state of the timer is active. This function call must to be the first for given timer.

Function 'ticker_off'

```
'void ticker_off (ticker_t *ticker)'
```

stops the timer given as the first parameter.

Function 'ticker_on'

```
'void ticker_on (ticker_t *ticker)'
```

activates the timer given as the first parameter.

Public function 'active_time'

```
'double active_time (ticker_t ticker)'
```

returns time in seconds as double float value in which given timer was active.

Public function 'active_time_string'

```
'const char *active_time_string (void)'
```

returns string representation of time in seconds in which the timer given as the function parameter was active. Remember that this function must be the single in a C++ expression because the string is stored in a static variable of the function.

13 Expandable sparse set

The package ‘bits’ can be used to represent sets. In many applications (e.g. in compiler optimizations) sets are sparse for all possible elements values. The package ‘spset’ implements sparse sets described in "An Efficient Representation for Sparse Sets" by Preston Briggs and Linda Torczon. The sparse set implementation has the following features:

- It is fast. It has complexity O(1) for testing, insertion, remove of the element and O(n) for union, intersection, difference, comparison, and traverse of the sets, where ‘n’ is number of elements in the set.
- The set has a size. Elements with values less than size can be stored in the set. The memory is needed for the set representation is O(size).
- When the element is inserted and does not fit into the set, the set is automatically extended to store the element and may be elements with even bigger values.

The interface part of the sparse set is file ‘spset.h’. The implementation part is file ‘spset.c’. The interface contains the following external definitions and macros:

Type ‘spset_elem_t’

is an unsigned integer type representing the set element.

Macro ‘SPSET_MAX_ELEM’

is a maximal possible value of the previous type.

Type ‘spset_t’

represents a sparse set. Knowledge of structure (implementation) of this type is not visible and not needed for using the parser.

Function ‘spset_init’

```
‘void spset_init (spset_t *s, spset_elem_t size)’
```

should be called the first. It initializes set ‘s’ with initial size close to ‘size’ but not less than it. You can work only with initialized and non-finished sets.

Function ‘spset_finish’

```
‘void spset_finish (spset_t *s)’
```

should be called the last. It finishes set work with set ‘s’.

Function ‘spset_size’

```
‘spset_elem_t spset_size (spset_t *s)’
```

returns the current size of the set ‘s’. Elements with values less than the size can be inserted without an expansion.

Function 'spset_cardinality'

```
'spset_elem_t spset_cardinality (spset_t *s)'
```

returns number of elements containing currently in set 's'.

Function 'spset_copy'

```
'void spset_copy (spset_t *to, spset_t *from)'
```

copies value of set 'from' to set 'to'.

Function 'spset_swap'

```
'void spset_swap (spset_t *s1, spset_t *s2)'
```

swaps values of sets 's1' and 's2'.

Function 'spset_in_p'

```
'int spset_in_p (spset_t *s, spset_elem_t el)'
```

returns TRUE if set 's' constains element 'el'.

Function 'spset_insert'

```
'int spset_insert (spset_t *s, spset_elem_t el)'
```

inserts element 'el' into set 's'. Return TRUE if the set has been changed, in order words, it did not contain the element. If the set size is not enough to contain the element, the set is expanded.

Function 'spset_remove'

```
'int spset_remove (spset_t *s, spset_elem_t el)'
```

removes element 'el' from set 's'. Return TRUE if the set has been changed, in order words, it really contained the element.

Function 'spset_clear'

```
'int spset_clear (spset_t *s)'
```

removes all elements from set 's'. Return TRUE if the set has been changed, in order words, it was not empty.

Function 'spset_equal_p'

```
'int spset_equal_p (spset_t *s1, spset_t *s2)'
```

returns TRUE if sets ‘s1’ and ‘s2’ contain exactly the same elements.

Function ‘spset_intersect’

```
‘int spset_intersect (spset_t *s1, spset_t *s2)’
```

removes elements in set ‘s1’ which are not is set ‘s2’. Returns TRUE if the set ‘s1’ has been changed, in order words, if ‘s1’ or ‘s2’ had any non-common elements.

Function ‘spset_unity’

```
‘int spset_unity (spset_t *s1, spset_t *s2)’
```

adds all elements of set ‘s2’ to set ‘s1’. Returns TRUE if the set ‘s1’ has been changed, in order words, if any new element was added to ‘s1’. The set can be automatically expanded if it is necessary.

Function ‘spset_diff’

```
‘int spset_intersect (spset_t *s1, spset_t *s2)’
```

removes all elements of set ‘s2’ from set ‘s1’. Returns TRUE if the set ‘s1’ has been changed, in order words, if ‘s1’ and ‘s2’ had any common elements.

Function ‘spset_shrink’

```
‘void spset_shrink (spset_t *s)’
```

decreases size of set ‘s’ if it is possible. During set life, the set can be automatically expanded but it is never automatically shrinked. After some element removes, it might be possible to shrink set size to save memory. The function serves this purpose.

Function ‘spset_release_unused_memory’

```
‘void spset_release_unused_memory (void)’
```

releases internal memory pool. The pool is used to speed up size changing of sets. It is wise to call the function when you finish to work with all sets.

Function ‘spset_print’

```
‘void spset_print (FILE *f, spset_t *s)’
```

prints all elements of set ‘s’ into file ‘f’.

Function ‘spset_debug’

```
‘void spset_debug (spset_t *s)’
```

prints all elements of set ‘s’ into stderr.

Type ‘spset_iterator_t’

is used to define iterators for traverses of sets (see the macro below).

Macro ‘EXECUTE_FOR_EACH_SPSET_ELEM’

```
‘EXECUTE_FOR_EACH_SPSET_ELEM (SET, EL, SI) c-stmt’
```

executes ‘c-stmt’ for each element of ‘SET’. During execution, the current element value is assigned to ‘EL’. You need to declare somewhere and use iterator ‘SI’ in the macro call. The order of traverse of the set elements is undefined. The traverse behaviour when the traversed set is changed during the traverse is not defined.

14 Expandable compact sparse set

The package ‘spsets’ can be used to represent few sets as the set memory is proportional to maximal possible element value. And as this value can be big, the sets can be big too. In some applications (e.g. basic block pseudo live info at the beginning and end of each basic block in compiler) you need a lot of sets and spset package is not acceptable for such uses as it will require huge memory. We need a set representation whose memory is proportional to number of set elements which for sparse sets is much smaller than maximal possible element value. The package ’cspset’ (expandable compact sparse sets) implements such representation. We use special case hash tables of appropriate sizes to keep the tables compact but still fast for access. The algorithm complexities of set operations in practice is the same as for package ‘spset’ but have bigger constants.

The interface part of the sparse set is file ‘cspset.h’. The implementation part is file ‘cspset.c’. The interface is very similar to ‘spset‘ one. It contains the following external definitions and macros:

Type ‘cspset_elem_t’

is an unsigned integer type representing the set element. It is the same type as spset_elem_t.

Constant ‘cspset_max_elem’

is a maximal possible value of the previous type for the element. That is pretty big value very close to maximal value of the above type.

Type ‘cspset_t’

represents a compact sparse set. Knowledge of structure (implementation) of this type is not visible and not needed for using the parser.

Function ‘cspset_init’

```
‘void cspset_init (cspset_t *s, cspset_elem_t size)’
```

should be called the first. It initializes set ‘s’ with initial size close to ‘size’ but not less than it. You can work only with initialized and non-finished sets.

Function ‘cspset_finish’

```
'void cspset_finish (cspset_t *s)'
```

should be called the last. It finishes set work with set 's'.

Function 'cspset_size'

```
'cspset_elem_t cspset_size (cspset_t *s)'
```

returns the current size of the set 's'. The set without an expansion can contain elements whose quantity is close to the set size.

Function 'cspset_cardinality'

```
'cspset_elem_t cspset_cardinality (cspset_t *s)'
```

returns number of elements containing currently in set 's'.

Function 'cspset_copy'

```
'void cspset_copy (cspset_t *to, cspset_t *from)'
```

copies value of set 'from' to set 'to'.

Function 'cspset_swap'

```
'void cspset_swap (cspset_t *s1, cspset_t *s2)'
```

swaps values of sets 's1' and 's2'.

Function 'cspset_in_p'

```
'int cspset_in_p (cspset_t *s, cspset_elem_t el)'
```

returns TRUE if set 's' constains element 'el'.

Function 'cspset_insert'

```
'int cspset_insert (cspset_t *s, cspset_elem_t el)'
```

inserts element 'el' into set 's'. Return TRUE if the set has been changed, in order words, it did not contain the element. If the set size is not enough to contain the current number of elements, the set is expanded.

Function 'cspset_remove'

```
'int cspset_remove (cspset_t *s, cspset_elem_t el)'
```

removes element ‘el’ from set ‘s’. Return TRUE if the set has been changed, in order words, it really contained the element.

Function ‘cspset_clear’

```
‘int cspset_clear (cspset_t *s)’
```

removes all elements from set ‘s’. Return TRUE if the set has been changed, in order words, it was not empty.

Function ‘cspset_equal_p’

```
‘int cspset_equal_p (cspset_t *s1, cspset_t *s2)’
```

returns TRUE if sets ‘s1’ and ‘s2’ contain exactly the same elements.

Function ‘cspset_intersect’

```
‘int cspset_intersect (cspset_t *s1, cspset_t *s2)’
```

removes elements in set ‘s1’ which are not is set ‘s2’. Returns TRUE if the set ‘s1’ has been changed, in order words, if ‘s1’ or ‘s2’ had any non-common elements.

Function ‘cspset_unity’

```
‘int cspset_unity (cspset_t *s1, cspset_t *s2)’
```

adds all elements of set ‘s2’ to set ‘s1’. Returns TRUE if the set ‘s1’ has been changed, in order words, if any new element was added to ‘s1’. The set can be automatically expanded if it is necessary.

Function ‘cspset_diff’

```
‘int cspset_intersect (cspset_t *s1, cspset_t *s2)’
```

removes all elements of set ‘s2’ from set ‘s1’. Returns TRUE if the set ‘s1’ has been changed, in order words, if ‘s1’ and ‘s2’ had any common elements.

Function ‘cspset_shrink’

```
‘void cspset_shrink (cspset_t *s)’
```

decreases size of set ‘s’ if it is possible. During set life, the set can be automatically expanded but it is never automatically shrinked. After some element removes, it might be possible to shrink set size to save memory. The function serves this purpose.

Function ‘cspset_release_unused_memory’

```
‘void cspset_release_unused_memory (void)’
```

releases internal memory pool. The pool is used to speed up size changing of sets. It is wise to call the function when you finish to work with all sets.

Function 'cspset_print'

```
'void cspset_print (FILE *f, cspset_t *s)'
```

prints all elements of set 's' into file 'f'.

Function 'cspset_debug'

```
'void cspset_debug (cspset_t *s)'
```

prints all elements of set 's' into stderr.

Type 'cspset_iterator_t'

is used to define iterators for traverses of sets (see the macro below).

Macro 'EXECUTE_FOR_EACH_CSPSET_ELEM'

```
'EXECUTE_FOR_EACH_CSPSET_ELEM (SET, EL, SI) c-stmt'
```

executes 'c-stmt' for each element of 'SET'. During execution, the current element value is assigned to 'EL'. You need to declare somewhere and use iterator 'SI' in the macro call. The order of traverse of the set elements is undefined. The traverse behaviour when the traversed set is changed during the traverse is not defined.

Function 'cspset_to_spset'

```
'void cspset_to_spset (cspset_t *from, spset_t *to)'
```

is used to transform compact sparse set 'from' to sparse set 'to'.

Function 'cspset_from_spset'

```
'void cspset_from_spset (cspset_t *to, spset_t *from)'
```

is used to transform compact sparse set 'to' from sparse set 'from'.

15 YAEP

YAEP is an abbreviation of Yet Another Earley Parser. The package 'YAEP' implements earley parser. The earley parser implementation has the following features:

- It is sufficiently fast and does not require much memory. This is the fastest implementation of Earley parser which I know. The main design goal is to achieve speed and memory requirements which are necessary to use it in prototype compilers and language processors. It parses 30K lines of C program per second on 500 MHz Pentium III and allocates about 5Mb memory for 10K line C program.

- It makes simple syntax directed translation. So an abstract tree is already the output of YAEP.
- It can parse input described by an ambiguous grammar. In this case the parse result can be an abstract tree or all possible abstract trees. Moreover it produces the compact representation of all possible parse trees by using DAG instead of real trees. This feature can be used to parse natural language sentences.
- It can parse input described by an ambiguous grammar according to the abstract node costs. In this case the parse result can be an minimal cost abstract tree or all possible minimal cost abstract trees. This feature can be used to code selection task in compilers.
- It can make syntax error recovery. Moreover its error recovery algorithms finds error recovery with minimal number of ignored tokens. It permits to implement parsers with very good error recovery and reporting.
- It has fast startup. There is no practically delay between processing grammar and start of parsing.
- It has flexible interface. The input grammar can be given by YACC-like description or providing functions returning terminals and rules.
- It has good debugging features. It can print huge amount of information about grammar, parsing, error recovery, translation. You can even output the result translation in form for a graphic visualization program.

The interface part of the parser is file 'yaep.h'. The implementation part is file 'yaep.c'. The interface contains the following external definitions and macros:

Struct 'grammar'

describes a grammar. Knowledge of structure (implementation) of this type is not visible and not needed for using the parser.

Macro 'YAEP_NIL_TRANSLATION_NUMBER'

is reserved to be designation of empty node for translation.

Macro 'YAEP_NO_MEMORY'

is error code of the parser. The parser functions return the code when parser can not allocate enough memory for its work.

Macro 'YAEP_UNDEFINED_OR_BAD_GRAMMAR'

is error code of the parser. The parser functions return the code when we call parsing without defining grammar or call parsing for bad defined grammar.

Macro 'YAEP_DESCRIPTION_SYNTAX_ERROR_CODE'

is error code of the parser. The code is returned when the grammar is defined by description and there is syntax error in the description.

Macro 'YAEP_FIXED_NAME_USAGE'

is error code of the parser. The code is returned when the grammar uses reserved names for terminals and nonterminals. There are two reserved names '$S' (for axiom) and '$eof' for end of file (input end marker). The parser adds these symbols and rules with these symbols to the grammar given by user. So user should not use these names in his grammar.

Macro 'YAEP_REPEATED_TERM_DECL'

is error code of the parser. The code is returned when the grammar contains several declarations of terminals with the same name.

Macro 'YAEP_NEGATIVE_TERM_CODE'

is error code of the parser. The code is returned when the grammar terminal is described with negative code.

Macro 'YAEP_REPEATED_TERM_CODE'

is error code of the parser. The code is returned when the two or more grammar terminals are described with the same code.

Macro 'YAEP_NO_RULES'

is error code of the parser. The code is returned when the grammar given by user has no rules.

Macro 'YAEP_TERM_IN_RULE_LHS'

is error code of the parser. The code is returned when grammar rule given by user contains terminal in left hand side of the rule.

Macro 'YAEP_INCORRECT_TRANSLATION'

is error code of the parser. The code is returned when grammar rule translation is not correct. The single reason for this is translation of the rule consists of translations of more one symbols in the right hand side of the rule without forming an abstract tree node.

Macro 'YAEP_NEGATIVE COST'

is error code of the parser. The code is returned when abstract node has a negative cost.

Macro 'YAEP_INCORRECT_SYMBOL_NUMBER'

is error code of the parser. The code is returned when grammar rule translation contains incorrect symbol number which should be nonnegative number less than rule right hand side length.

Macro 'YAEP_UNACCESSIBLE_NONTERM'

is error code of the parser. The code is returned when there is grammar nonterminal which can not be derived from axiom.

Macro 'YAEP_NONTERM_DERIVATION'

is error code of the parser. The code is returned when there is grammar nonterminal which can not derive a terminal string.

Macro 'YAEP_LOOP_NONTERM'

is error code of the parser. The code is returned when there is grammar nonterminal which can derive only itself. The parser does not work with such grammars.

Macro 'YAEP_INVALID_TOKEN_CODE'

is error code of the parser. The code is returned when the parser got input token whose code is different from all grammar terminal codes.

Enumeration 'yaep_tree_node_type'

describes all possible nodes of abstract tree representing the translation. There are the following enumeration constants:

'YAEP_NIL'

the corresponding node represents empty translations.

'YAEP_ERROR'

the corresponding node represents translation of special terminal 'error' (see error recovery).

'YAEP_TERM'

the corresponding node represents translation of a terminal.

'YAEP_ANODE'

the corresponding node represents an abstract node.

'YAEP_ALT'

the corresponding node represents an alternative of the translation. Such nodes creates only when there are two or more possible translations. It means that the grammar is ambiguous.

Structure 'yaep_tree_node'

represents node of the translation. The nodes refer for each other forming DAG (direct acyclic graph) in general case. The main reason of generating DAG is that some input fragments may have the same translation, when there are several parsings of input (which is possible only for ambiguous grammars). But DAG may be created even for unambigous grammar because some nodes (empty and error nodes) exist only in one exemplar. When such nodes are not created, the translation nodes forms a tree. This structure has the following members:

Member 'type' of type 'enum yaep_tree_node_type'

representing type of the translation node.

Union 'val'

Depending on the translation node type, one of the union members 'nil', 'error', 'term', 'anode', and 'alt' of the structure types described below is used to represent the translation node.

Structure 'yaep_nil'

represents empty node. It has no members. Actually the translation is DAG (not tree) in general case. The empty and error nodes are present only in one exemplar.

Structure 'yaep_error'

represents translation of special terminal 'error'. It has no members. The error node exists only in one exemplar.

Structure 'yaep_term'

represents translation of terminals. It has the following two members:

Integer member 'code'

representing code of the corresponding terminal.

Member 'attr' of type '* void'

is reference for the attribute of the corresponding terminal.

Structure 'yaep_anode'

represents abstract node. It has the following two members:

Member 'name' of type 'const char *'

representing name of anode as it given in the corresponding rule translation.

Member 'cost' of type 'int'

representing cost of the node plus costs of all children if the cost flag is set up. Otherwise, the value is cost of the abstract node itself.

Member 'children' of type 'struct yaep_tree_node **'

is array of nodes representing the translations of the symbols given in the rule with the abstract node.

Structure 'yaep_alt'

represents an alternative of the translation. It has the following two members:

Member 'node' of type 'struct yaep_tree_node *'

representing alternative translation.

Member 'next' of type 'struct yaep_tree_node *'

is reference for the next alternative of translation.

Function 'yaep_create_grammar'

```
'struct grammar *yaep_create_grammar (void)'
```

should be called the first. It actually creates an yaep parser with undefined grammar. You can use two or more parsers simultaneously. The function returns 'NULL' if there is no memory.

Function 'yaep_error_code'

```
'int yaep_error_code (struct grammar *g)'
```

returns the last occurred error code (see the possible error codes above) for given parser. If the function returns zero, no error was found so far.

Function 'yaep_error_message'

```
'const char *yaep_error_message (struct grammar *g)'
```

returns detail message about last occurred error. The message always corresponds to the last error code returned the previous function.

Function 'yaep_read_grammar'

```
'int yaep_read_grammar (struct grammar *g, int strict_p,
                        const char *(*read_terminal) (int *code),
                        const char *(*read_rule)
                                     (const char ***rhs,
                                      const char **abs_node,
```

```
                                                    int *anode_cost,
                                                    int **transl))'
```

is one of two functions which tunes the parser to given grammar. The grammar is read with the aid functions given as parameters.

'read_terminal' is function for reading terminals. This function is called before function 'read_rule'. The function should return the name and the code of the next terminal. If all terminals have been read the function returns NULL. The terminal code should be nonnegative.

'read_rule' is function called to read the next rule. This function is called after function 'read_terminal'. The function should return the name of nonterminal in the left hand side of the rule and array of names of symbols in the right hand side of the rule (the array end marker should be 'NULL'). If all rules have been read, the function returns 'NULL'. All symbol with name which was not provided function 'read_terminal' are considered to be nonterminals. The function also returns translation given by abstract node name and its fields which will be translation of symbols (with indexes given in array given by parameter 'transl') in the right hand side of the rule. All indexes in 'transl' should be different (so the translation of a symbol can not be represented twice). The end marker of the array should be a negative value. There is a reserved value of the translation symbol number denoting empty node. It is value defined by macro 'YAEP_NIL_TRANSLATION_NUMBER'. If parameter 'transl' is 'NULL' or contains only the end marker, translations of the rule will be empty node. If 'abs_node' is 'NULL', abstract node is not created. In this case 'transl' should be null or contain at most one element. This means that the translation of the rule will be correspondingly empty node or the translation of the symbol in the right hand side given by the single array element. The cost of the abstract node if given is passed through parameter 'anode_cost'. If 'abs_node' is not 'NULL', the cost should be greater or equal to zero. Otherwise the cost is ignored.

There is reserved terminal 'error' which is used to mark start point of error recovery.

Nonzero parameter 'strict_p' value means more strict checking the grammar. In this case, all nonterminals will be checked on ability to derive a terminal string instead of only checking axiom for this.

The function returns zero if it is all ok. Otherwise, the function returns the error code occured.

Function 'yaep_parse_grammar'

```
'int yaep_parse_grammar (struct grammar *g, int strict_p,
                         const char *description)'
```

is another function which tunes the parser to given grammar. The grammar is given by string 'description'. The description is similiar YACC one. It has the following syntax:

```
file : file terms [';']
     | file rule
     | terms [';']
     | rule

terms : terms IDENTIFIER ['=' NUMBER]
      | TERM
```

```
rule : IDENTIFIER ':' rhs [';']

rhs : rhs '|' sequence [translation]
    | sequence [translation]

sequence :
         | sequence IDENTIFIER
         | sequence C_CHARACTER_CONSTANT

translation : '#'
            | '#' NUMBER
            | '#' '-'
            | '#' IDENTIFIER [NUMBER] '(' numbers ')'

numbers :
        | numbers NUMBER
        | numbers '-'
```

So the description consists of terminal declaration and rules sections.

Terminal declaration section describes name of terminals and their codes. Terminal code is optional. If it is omitted, the terminal code will the next free code starting with 256. You can declare terminal several times (the single condition its code should be the same).

Character constant present in the rules is a terminal described by default. Its code is always code of the character constant.

Rules syntax is the same as YACC rule syntax. The single difference is an optional translation construction starting with '#' right after each alternative. The translation part could be a single number which means that the translation of the alternative will be the translation of the symbol with given number (symbol numbers in alternative starts with 0). Or the translation can be empty or '-' which mean empty node. Or the translation can be abstract node with given name, optional cost, and with fields whose values are the translations of the alternative symbols with numbers given in parentheses after the abstract node name. You can use '-' in abstract node to show that empty node should be used in this place. If the cost is absent it is believed to be one. The cost of terminal, error node, and empty node is always zero.

There is reserved terminal 'error' which is used to mark start point of error recovery.

Function 'yaep_set_lookahead_level'

```
'int yaep_set_lookahead_level (struct grammar *grammar,
                               int level)'
```

sets up level of usage of look ahead in parser work. Value zero means no usage of lookaheads at all. Lookahead with static (independent on input tokens) context sets in parser situation (value 1) gives the best results with the point of view of space and speed, lookahead with dynamic (dependent on input tokens) context sets in parser situations (all the rest parameter values) does slightly worse, and no usage of lookaheads does the worst. The default value is 1 (lookahead with static situation context sets). The function returns the previously set up level. If the level value is negative, zero is used instead of it. If the value is greater than two, two is used in this case.

Function 'yaep_set_debug_level'

```
'int yaep_set_debug_level (struct grammar *grammar,
                           int level)'
```

sets up level of debugging information output to 'stderr'. The more level, the more information is output. The default value is 0 (no output). The debugging information includes statistics, result translation tree, grammar, parser sets, parser sets with all situations, situations with contexts. The function returns the previously set up debug level. Setting up negative debug level results in output of translation for program 'dot' of graphic visualization package 'graphviz'.

Function 'yaep_set_one_parse_flag'

```
'int yaep_set_one_parse_flag (struct grammar *grammar,
                              int flag)'
```

sets up building only one translation tree (parameter value 0) or all parse trees for ambiguous grammar for which several parsings are possible. For unambiguous grammar the flag does not affect the result. The default value is 1. The function returns the previously used flag value.

Function 'yaep_set_cost_flag'

```
'int yaep_set_cost_flag (struct grammar *grammar, int flag)'
```

sets up building only translation tree (trees if we set up one_parse_flag to 0) with minimal cost. For unambiguous grammar the flag does not affect the result. The default value is 0. The function returns the previously used flag value.

Function 'yaep_set_error_recovery_flag'

```
'int yaep_set_error_recovery_flag (struct grammar *grammar,
                                   int flag)'
```

sets up internal flag whose nonzero value means making error recovery if syntax error occurred. Otherwise, syntax error results in finishing parsing (although function 'syntax_error' in function 'yaep_parse' will be called once). The default value is 1. The function returns the previously used flag value.

Function 'yaep_set_recovery_match'

```
'int yaep_set_recovery_match (struct grammar *grammar,
                              int n_toks)'
```

sets up recovery parameter which means how much subsequent tokens should be successfully shifted to finish error recovery. The default value is 3. The function returns the previously used flag value.

Function 'yaep_parse'

```
'int yaep_parse (struct grammar *grammar,
                 int (*read_token) (void **attr),
                 void (*syntax_error)
                      (int err_tok_num, void *err_tok_attr,
                       int start_ignored_tok_num,
                       void *start_ignored_tok_attr,
                       int start_recovered_tok_num,
                       void *start_recovered_tok_attr),
                 void *(*parse_alloc) (int nmemb),
                 void (*parse_free) (void *mem),
                 struct yaep_tree_node **root,
                 int *ambiguous_p)'
```

is major parser function. It parses input according the grammar. The function returns the error code (which can be also returned by 'yaep_error_code'). If the code is zero, the function will also return root of the parse tree through parameter 'root'. The tree representing the translation. Value passed through 'root' will be 'NULL' only if syntax error was occurred and error recovery was switched off. The function sets up flag passed by parameter 'ambiguous_p' if we found that the grammar is ambiguous (it works even we asked only one parse tree without alternatives).

Function 'read_token' provides input tokens. It returns code the next input token and its attribute. If the function returns negative value we've read all tokens.

Function 'syntax_error' called when synatctic error has been found. It may print an error message about syntax error which occurred on token with number 'err_tok_num' and attribute 'err_tok_attr'. The following four parameters describes made error recovery which ignored tokens starting with token given by 3rd and 4th parameters. The first token which was not ignored is described by the last parameters. If the number of ignored tokens is zero, the all parameters describes the same token. If the error recovery is switched off (see comments for 'yaep_set_error_recovery_flag'), the third and the fifth parameters will be negative and the forth and the sixth parameters will be 'NULL'.

Function 'parse_alloc' is used by YAEP to allocate memory for parse tree representation (translation). After calling 'yaep_free_grammar' we free all memory allocated for the parser. At this point it is convenient to free all memory but parse tree. Therefore we require the following function. So the caller will be responsible to allocate and free memory for parse tree representation (translation). But the caller should not free the memory until 'yaep_free_grammar' is called for the parser. The function may be called even during reading the grammar not only during the parsing. Function 'parse_free' is used by the parser to free memory allocated by 'parse_alloc'. If it is 'NULL', the memory is not freed.

Function 'yaep_free_grammar'

```
'void yaep_free_grammar (struct grammar *grammar)'
```

frees all memory allocated for the parser. This function should be called the last for given parser.