TinyTalk

1.0.0

Generated by Doxygen 1.9.1

Chapter 1

TT Language

1.1 Introduction

TT Technical Details

Sample definitions:

```
OrderedCollection class [
    main: args [ '{1}/{2}' format: {10. 'Peter ist doof!'} ]
    main2: args [
        1 to: 9 do: [ :i |
            1 to: i do: [ :j |
                Transcript
                    show: ('{1} * {2} = {3}' format: {j. i. j * i});
                    show: ' '
            ].
            Transcript show: ' '; cr.
        ]
    ]
]
String [
format: collection [
    "
    Format the receiver by interpolating elements from collection,
    as in the following examples:
    ('Five is {1}.' format: { 1 + 4})
                >>> 'Five is 5.'
    ('Five is {five}.' format: (Dictionary with: #five -> 5))
                >>>  'Five is 5.'
    ('In {1} you can escape \{ by prefixing it with \\' format: {'strings'})
                >>> 'In strings you can escape { by prefixing it with \'
    ('In \{1\} you can escape \{ by prefixing it with \\' format: {'strings'})
                >>> 'In {1} you can escape { by prefixing it with \'
    "

    ^ self species
        new: self size
        streamContents: [ :result |
            | stream |
            stream := self readStream.
            [ stream atEnd ]
                whileFalse: [ | currentChar |
                    (currentChar := stream next) == ${
                        ifTrue: [ | expression index |
                            expression := stream upTo: $}.
                            index := Integer readFrom: expression ifFail: [ expression ].
                            result nextPutAll: (collection at: index) asString ]
                        ifFalse: [
```

```
                                currentChar == $\
                                    ifTrue: [ stream atEnd
                                            ifFalse: [ result nextPut: stream next ] ]
                                    ifFalse: [ result nextPut: currentChar ] ] ] ] ]
]
Stream [
  nextPutAll: aCollection [^ aCollection do: [:each | self nextPut: each]]
]
ByteArray class [
      newWithSize: n []
]
```

Chapter 2

TT Technical Details

2.1 Main Features

2.2 Details

2.3 Chapter 1: Memory Management

2.4 Syntax

```
object_ident ::= IDENT.
object_ident ::= IDENT IDENT.
unary_pattern ::= IDENT.
binary_pattern ::= BINOP IDENT.
keyword_pattern ::= KEYWORD IDENT.
keyword_pattern ::= keyword_pattern KEYWORD IDENT.
all ::= object_defs.
object_defs ::=.
object_defs ::= object_defs object_ident LBRACK var_list method_defs RBRACK.
object_defs ::= object_defs object_ident LARROW IDENT LBRACK var_list method_defs RBRACK.
var_list ::=.
var_list ::= BAR idents BAR.
idents ::= IDENT.
idents ::= idents IDENT.
method_defs ::=.
method_defs ::= method_defs msg_pattern LBRACK var_list statements RBRACK.
method_defs ::= method_defs msg_pattern VERBATIM.
msg_pattern ::= unary_pattern.
msg_pattern ::= binary_pattern.
msg_pattern ::= keyword_pattern.
statements ::= return_statement.
statements ::= return_statement DOT.
statements ::= expression DOT statements.
statements ::= expression.
statements ::= expression DOT.
```

```
return_statement ::= UARROW expression.
expression ::= IDENT LARROW expr.
expression ::= basic_expression.
basic_expression ::= primary.
basic_expression ::= primary messages cascaded_messages.
basic_expression ::= primary cascaded_messages.
basic_expression ::= primary messages.
primary ::= IDENT.
primary ::= STRING.
primary ::= LBRACK block_body RBRACK.
primary ::= LBRACE expression RBRACE.
block_body ::= block_arguments BAR var_list statements.
block_body ::= var_list statements.
block_body ::= var_list.
block_arguments ::= COLON IDENT.
block_arguments ::= block_arguments COLON IDENT.
messages ::= unary_messages.
messages ::= unary_messages keyword_message.
messages ::= unary_messages binary_messages.
messages ::= unary_messages binary_messages keyword_message.
messages ::= binary_messages.
messages ::= binary_messages keyword_message.
messages ::= keyword_message.
unary_messages ::= IDENT.
binary_messages ::= binary_message.
binary_messages ::= binary_message binary_messages.
binary_message ::= BINOP binary_argument.
binary_argument ::= primary unary_messages.
binary_argument ::= primary.
keyword_message ::= KEYWORD keyword_argument.
keyword_message ::= keyword_message KEYWORD keyword_argument.
keyword_argument ::= primary.
keyword_argument ::= primary unary_messages.
keyword_argument ::= primary unary_messages binary_messages.
cascaded_messages ::= SEMICOLON messages.
cascaded_messages ::= cascaded_messages SEMICOLON messages.
atom ::= IDENT.
atom ::= STRING.
unary_call ::= unary_call IDENT.
binary_call ::= binary_call BINOP unary_call.
unary_call ::= atom.
binary_call ::= unary_call.
expr ::= binary_call.
```

2.5 Chapter 3: Implementation

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 5

Module Documentation

5.1 Environment

Functions

- void env_add (t_env *env, const t_name name)
- void env_clear (t_env *env)
- void env_set (t_env *env, const t_name name, t_object *value)
- t_slot * env_get (t_env *env, const t_name name)
- void env_dump (t_env *env, const char *)
- void env_set_local (t_env *env, const t_name name, t_object *value)
- t_env * env_new (t_env *parent)
- t_slot * env_get_all (t_env *env, const t_name name, t_env **env_found)

5.1.1 Detailed Description

5.1.2 Function Documentation

5.1.2.1 env_add()

```
void env_add (
            t_env * env,
            const t_name name )
```

adding a name defintion the an environment

Referenced by class_enter(), and method_enter().

5.1.2.2 env_clear()

```
void env_clear (
            t_env * env )
```

clearing the values from the environment

5.1.2.3 env_dump()

```
void env_dump (
            t_env * env,
            const char * reason )
```

output of the environment to stderr

References s_slot::next, and s_env::next.

Referenced by method_exec().

5.1.2.4 env_get()

```
t_slot* env_get (
            t_env * env,
            const t_name name )
```

getting the slot defining the current name value pair fitting the name

5.1.2.5 env_get_all()

```
t_slot* env_get_all (
            t_env * env,
            const t_name name,
            t_env ** env_found )
```

searching the name in alle environments starting at the top walking down.

Parameters

name	is the name of the defintion to be searched
env	referes to the top level environment
env_found	if not null receives the reference to the environment where the name has been found.

5.1.2.6 env_new()

```
t_env* env_new (
            t_env * parent )
```

new environment, inheriting definitions from another environment

Referenced by class_enter(), and method_enter().

5.1.2.7 env_set()

```
void env_set (
            t_env * env,
            const t_name name,
            t_object * value )
```

setting a value to a name defintion in the environment

5.1.2.8 env_set_local()

```
void env_set_local (
            t_env * env,
            const t_name name,
            t_object * value )
```

sets only the local definition if any. does not walk up the env chain. *

Referenced by block_handler(), and method_exec().

5.2 Internal Functions

Data Structures

- struct s_globals

Macros

- #define MSG_DUMP "dump"

Functions

- void class_enter (const char *name)
- bool is_ident_char (int c)

  check if character is part of an identifier.
- bool is_binary_char (int c)
- bool src_clear (void)
- bool src_add (const char *line)
- bool src_read (const char *name)
- bool src_dump (void)
- bool readLine (void)

  read one line from stdin stores the result into `gd.line`.
- bool readChar (char *t)

  read one character from input and store it somewhere.
- bool readStringToken (void)

  read string token.
- void parse_verbatim (char c)
- bool nextToken (void)

  read next token.
- char * method_name (const char *class, const char *sel)
- void require_classes (void)
- void require_current_class (void)
- void method_enter (t_message_pattern *mp)
- bool cstr_equals (const char *, const char *)
- t_object * object_new (t_message_handler hdl)
- t_object * object_send (t_object *self, const char *sel, t_object **args)
- void object_send_void (t_object *self, const char *sel, t_object **args)
- t_object * simulate (t_env *env, t_statements *stmts)
- t_object * eval (t_env *env, t_expression *expr)
- t_object * string_handler (t_object *self, const char *sel, t_object **args)
- t_object * string_meta_handler (t_object *self, const char *sel, t_object **args)
- t_object * int_handler (t_object *self, const char *sel, t_object **args)
- t_object * char_handler (t_object *self, const char *sel, t_object **args)
- t_object * block_handler (t_object *self, const char *sel, t_object **args)
- t_object * stream_handler (t_object *self, const char *sel, t_object **args)
- t_object * method_exec (t_object *self, const char *clsname, const char *sel, t_object **args)

Variables

- struct s_globals global

5.2.1 Detailed Description

5.2.2 Macro Definition Documentation

5.2.2.1 MSG_DUMP

```
#define MSG_DUMP "dump"
```

message selector for dumping an object

5.2.3 Function Documentation

5.2.3.1 block_handler()

```
t_object* block_handler (
            t_object * self,
            const char * sel,
            t_object ** args )
```

handle block messages

```
00251                                                                   {
00252     t_object *result = self;
00253     t_block *b = self->u.data;
00254     t_env *env = b->env;
00255     msg_add( "block handler" );
00256     if( cstr_equals( "dump", sel ) ) {
00257         msg_add( "dumping block.." );
00258     }
00259     else if( 0 == strcmp( "value", sel ) ) {
00260 
00261         result = simulate( env, b->statements );
00262     }
00263     else if( 0 == strcmp( "value:", sel ) ) {
00264         assert( args );
00265         assert( args[0] );
00266         assert( args[0]->handler );
00267         for( int i = 0; i < b->params.count; i++ ) {
00268             tt_assert( env != NULL );
00269             env_set_local( env, b->params.names[i], args[i] );
00270         }
00271         object_send_void( args[0], MSG_DUMP, NULL );
00272         result = simulate( env, b->statements );
00273     }
00274     else if( 0 == strcmp( "whileFalse:", sel ) ) {
00275         for( ;; ) {
00276             t_object *r = simulate( env, b->statements );
00277             object_send_void( r, "ifFalse:", args );
00278             if( r->handler == true_handler )
00279                 break;
00280         }
00281     }
00282     else
00283         result = method_exec( self, "Block", sel, args );
00284     return result;
00285 }
```

References s_namelist::count, cstr_equals(), s_block::env, env_set_local(), s_object::handler, method_exec(), msg_add(), MSG_DUMP, s_namelist::names, object_send_void(), s_block::params, simulate(), and s_block::statements.

5.2.3.2 char_handler()

```
t_object* char_handler (
            t_object * self,
            const char * sel,
            t_object ** args )
```

handle character messages

```
00095                                                                   {
00096     t_object *result = self;
00097     if( 0 == strcmp( "==", sel ) ) {
00098         if( args[0]->handler == char_handler ) {
00099             if( self->u.intval == args[0]->u.intval )
00100                 result = global.True;
00101             else
00102                 result = global.False;
```

```
00103         }
00104         else
00105             result = global.False;
00106     }
00107     else if( cstr_equals( "dump", sel ) ) {
00108         msg_add( "char: %c", self->u.intval );
00109     }
00110     else
00111         result = method_exec( self, "Char", sel, args );
00112     return result;
00113 }
```

References char_handler(), cstr_equals(), s_globals::False, global, s_object::intval, method_exec(), msg_add(), s_globals::True, and s_object::u.

Referenced by char_handler(), stream_handler(), and string_handler().

5.2.3.3 class_enter()

```
void class_enter (
             const char * name )
```

beginning a new class

```
00004                                       {
00005     require_classes( );
00006     t_classdef *odef = itab_read( classes, name );
00007
00008
00009     if( odef == NULL) {
00010         printf( "new class %s\n", name );
00011         odef = (t_classdef *)_talloc_zero(classes, sizeof(t_classdef), "t_classdef");
00012         odef->id = gd.classnum++;
00013         odef->name = talloc_strdup( odef, name );
00014         odef->env = env_new(gd.env);
00015         env_add(odef->env, "CLASSVAR");
00016         itab_append( classes, name, odef );
00017     }
00018     current_class = odef;
00019 }
```

5.2.3.4 cstr_equals()

```
bool cstr_equals (
             const char * a,
             const char * b )
```

compare to cstrings and return if they are equal

```
00037                                                   {
00038     return ( 0 == strcmp( a, b ) );
00039 }
```

Referenced by block_handler(), char_handler(), int_handler(), stream_handler(), and string_handler().

5.2.3.5 eval()

```
t_object* eval (
            t_env * env,
            t_expression * expr )
```

evaluate an expression

```
00321                                                   {
00322     t_object *result = NULL;
00323     assert( expr );
00324
00325     switch ( expr->tag ) {
00326         case tag_string:
00327             msg_add( "eval str %s", expr->u.strvalue );
00328             result = object_new( string_handler );
00329             result->u.data = talloc_strdup( result, expr->u.strvalue );
00330             break;
00331
00332         case tag_number:
00333             result = object_new( int_handler );
00334             result->u.intval = expr->u.intvalue;
00335             msg_add( "eval number %d", result->u.intval );
00336             break;
00337
00338         case tag_char:
00339             msg_add( "eval char" );
00340             result = object_new( char_handler );
00341             result->u.intval = expr->u.intvalue;
00342             break;
00343
00344         case tag_message:
00345             result = eval_messages( env, expr );
00346             break;
00347
00348         case tag_block:
00349             msg_add( "eval block" );
00350             result = object_new( block_handler );
00351             expr->u.block.env = env_new( env );
00352             result->u.data = &expr->u.block;
00353             for( int i = 0; i < expr->u.block.params.count; i++ ) {
00354                 env_add( expr->u.block.env, expr->u.block.params.names[i] );
00355             }
00356             for( int i = 0; i < expr->u.block.locals.count; i++ ) {
00357                 env_add( expr->u.block.env, expr->u.block.locals.names[i] );
00358             }
00359             break;
00360
00361         case tag_ident:
00362             result = NULL;
00363             t_slot *slot = env_get_all( env, expr->u.ident, NULL );
00364             result = slot->val;
00365             if( result )
00366                 msg_add( "eval ident %s -> %p(%p)", expr->u.ident, result,
00367                         result->handler );
00368             else {
00369                 env_dump( env, "IDENT no RESULT" );
00370                 tt_assert( result );
00371             }
00372             break;
00373
00374         case tag_assignment:
00375             msg_add( "eval assignment" );
00376             result = eval( env, expr->u.assignment.value );
00377             msg_add( "assign result[%p] to %s", result, expr->u.assignment.target );
00378             tt_assert( env );
00379             env_set( env, ( t_name ) expr->u.assignment.target, result );
00380             break;
00381
00382         case tag_array:
00383             {
00384                 int n = expr->u.exprs.count;
00385                 msg_add( "eval array %d elements", n );
00386                 result = object_new( array_handler );
00387                 result->u.vars.cnt = n;
00388                 result->u.vars.vs = talloc_array( result, t_object *, n );
00389                 for( int i = 0; i < expr->u.exprs.count; i++ ) {
00390                     result->u.vars.vs[i] = eval( env, expr->u.exprs.list[i] );
00391                 }
00392             }
00393             break;
00394
00395         default:
00396             msg_add( "error: unknown eval tag: %d", expr->tag );
00397             msg_print_last(  );
```

```
00398             abort(  );
00399             break;
00400     }
00401     return result;
00402 }
```

References s_expression::tag.

5.2.3.6 int_handler()

```
t_object* int_handler (
              t_object * self,
              const char * sel,
              t_object ** args )
```

handle integer messges

```
00197                                                                    {
00198     t_object *result = self;
00199     assert( sel );
00200     msg_add( "int handler: (%d) %s", self->u.intval, sel );
00201     if( 0 == strcmp( sel, "to:do:" ) ) {
00202         assert( args[0]->handler == int_handler );
00203         int start = self->u.intval;
00204         int finish = args[0]->u.intval;
00205 
00206         for( int i = start; i <= finish; i++ ) {
00207             t_object *par[1];
00208             par[0] = object_new( int_handler );
00209             par[0]->u.intval = i;
00210             object_send_void( args[1], "value:", par );
00211         }
00212     }
00213     else if( 0 == strcmp( sel, MSG_DUMP ) ) {
00214         msg_add( "int: %d", self->u.intval );
00215     }
00216     else if( cstr_equals( "asString", sel ) ) {
00217         result = object_new( string_handler );
00218         result->u.data = talloc_asprintf( result, "%d", self->u.intval );
00219     }
00220     else
00221         result = method_exec( self, "Integer", sel, args );
00222     return result;
00223 }
```

References cstr_equals(), s_object::data, int_handler(), s_object::intval, method_exec(), msg_add(), MSG_DUMP, object_new(), object_send_void(), string_handler(), and s_object::u.

Referenced by int_handler(), and string_handler().

5.2.3.7 is_binary_char()

```
bool is_binary_char (
              int c )
```

binary chars are special ones for binary message names

```
00228                                {
00229     switch ( c ) {
00230         case '!':
00231         case '%':
00232         case '&':
00233         case '*':
00234         case '+':
00235         case ',':
00236         case '/':
00237         case '<':
```

```
00238         case '=':
00239         case '>':
00240         case '?':
00241         case '@':
00242         case '\\':
00243         case '~':
00244         case '|':                                   // sollte laut Vorschlag ein Binary Operator sein.
      Kollidiert aber mit der temporary declaration.
00245 // das muss dann wohl auf der Syntaxebene geklärt werden.
00246         case '-':
00247             return true;
00248         default:
00249             return false;
00250     }
00251 }
```

5.2.3.8 is_ident_char()

```
bool is_ident_char (
            int c )
```

check if character is part of an identifier.

Parameters

in	*c*	character to classify.

Returns

true if if c is an identifier character.

```
00223                               {
00224     return isalpha( c ) || isdigit( c ) || c == '_';
00225 }
```

5.2.3.9 method_enter()

```
void method_enter (
            t_message_pattern * mp )
```

enter a method

```
00006                                           {
00007     require_current_class(  );
00008     char *sel = talloc_strdup( NULL, mp->parts.names[0] );
00009     for( int i = 1; i < mp->parts.count; i++ ) {
00010         sel = talloc_strdup_append( sel, mp->parts.names[i] );
00011     }
00012     char *nm = method_name( current_class->name, sel );
00013     t_methoddef *odef = itab_read( methods, nm );
00014     if( odef == NULL ) {
00015         odef = talloc_zero( methods, t_methoddef );
00016         odef->sel = talloc_strdup( odef, sel );
00017         odef->env = env_new(current_class->env);
00018         env_add(odef->env,"self");
00019         namelist_copy( &odef->args, &mp->names );
00020         for(int i=0;i<odef->args.count;i++){
00021             env_add(odef->env, odef->args.names[i]);
00022         }
00023 
00024         talloc_steal( odef, odef->args.names );
00025         assert( talloc_get_type( odef->args.names, t_name ) );
00026         itab_append( methods, nm, odef );
```

```
00027     }
00028     current_method = odef;
00029     talloc_steal(odef, mp);
00030     talloc_free( nm );
00031     talloc_free( sel );
00032 }
```

References s_methoddef::args, s_namelist::count, s_classdef::env, s_methoddef::env, env_add(), env_new(), itab_append(), itab_read(), method_name(), s_classdef::name, namelist_copy(), s_namelist::names, s_message_pattern::names, s_message_pattern::parts, require_current_class(), and s_methoddef::sel.

5.2.3.10 method_exec()

```
t_object* method_exec (
            t_object * self,
            const char * clsname,
            const char * sel,
            t_object ** args )
```

execute a method

```
00227                                                {
00228     t_object *result = self;
00229     t_methoddef *m = method_read( clsname, sel );
00230     if( m ) {
00231         t_env *env = m->env;
00232         tt_assert( env );
00233         msg_add( "selector %s is defined on %s (env:%p, self:%p, handler:%p)",
00234                  sel, clsname, env, self, self->handler );
00235         for( int i = 0; i < m->args.count; i++ ) {
00236             env_set_local( env, m->args.names[i], args[i] );
00237         }
00238         env_set_local( env, "self", self );
00239         env_dump( env, "after self" );
00240         result = simulate( env, m->statements );
00241         msg_add( "done simulation of method %s", sel );
00242     }
00243     else {
00244         msg_add( "%s %s not found.", clsname, sel );
00245         msg_print_last(  );
00246         abort(  );
00247     }
00248     return result;
00249 }
```

References s_methoddef::args, s_namelist::count, s_methoddef::env, env_dump(), env_set_local(), msg_add(), msg_print_last(), s_namelist::names, simulate(), and s_methoddef::statements.

Referenced by block_handler(), char_handler(), int_handler(), stream_handler(), string_handler(), and string_meta_handler().

5.2.3.11 method_name()

```
char * method_name (
            const char * class,
            const char * sel )
```

construct the method name from combination of class and selector

```
00002                                                {
00003     char *result = talloc_strdup( NULL , class );
00004     result = talloc_strdup_append( result, "/" );
00005     result = talloc_strdup_append( result, sel );
00006     return result;
00007 }
```

Referenced by method_enter().

5.2.3.12 nextToken()

```
bool nextToken (
            void  )
```

read next token.

parse the next token

This is a more detailed description.

Returns

true if successful

```
00378                   {
00379     char c;
00380     bool result = false;
00381     while( true ) {
00382         while( readChar( &c ) && isspace( c ) );
00383         if( c == '"' ) {
00384             while( readChar( &c ) && c != '"' );
00385         }
00386         else
00387             break;
00388     }
00389     if( gd.state == 1 ) {
00390         if( isalpha( c ) ) {
00391             int idx = 0;
00392             for( ;; ) {
00393                 gd.buf[idx++] = c;
00394                 readChar( &c );
00395                 if( !is_ident_char( c ) )
00396                     break;
00397             }
00398             if( c == ':' ) {
00399                 gd.buf[idx++] = c;
00400                 gd.token = TK_KEYWORD;
00401             }
00402             else {
00403                 gd.pos--;
00404                 gd.token = TK_IDENT;
00405             }
00406             gd.buf[idx] = 0;
00407             result = true;
00408         }
00409         else if( is_binary_char( c ) ) {
00410             for( int idx = 0; is_binary_char( c ); idx++ ) {
00411                 gd.buf[idx] = c;
00412                 gd.buf[idx + 1] = 0;
00413                 readChar( &c );
00414             }
00415             gd.pos--;
00416             gd.token = 0;
00417             gd.token = TK_BINOP;
00418             result = true;
00419             if( strcmp( ":=", gd.buf ) == 0 ) {
00420                 gd.token = TK_ASSIGN;
00421                 result = true;
00422             }
00423             else if( strcmp( "<-", gd.buf ) == 0 ) {
00424                 gd.token = TK_LARROW;
00425                 result = true;
00426             }
00427             else if( strcmp( "|", gd.buf ) == 0 ) {
00428                 gd.token = TK_BAR;
00429                 result = true;
00430             }
00431             else if( 0 == strcmp( "<", gd.buf ) ) {
00432                 gd.token = TK_LT;
00433                 result = true;
00434             }
00435             else if( 0 == strcmp( ">", gd.buf ) ) {
00436                 gd.token = TK_GT;
00437                 result = true;
00438             }
00439         }
00440         else if( isdigit( c ) ) {
00441             int idx = 0;
00442             while( isdigit( c ) ) {
```

```
00443                 printf( "### digit %c\n", c );
00444                 gd.buf[idx++] = c;
00445                 gd.buf[idx] = 0;
00446                 readChar( &c );
00447             }
00448             gd.pos--;
00449             gd.token = TK_NUMBER;
00450             result = true;
00451         }
00452         else {
00453             switch ( c ) {
00454                 case '\"':
00455                     result = readStringToken(  );
00456                     break;
00457                 case '.':
00458                     result = true;
00459                     gd.token = TK_DOT;
00460                     break;
00461                 case ';':
00462                     result = true;
00463                     gd.token = TK_SEMICOLON;
00464                     break;
00465                 case '(':
00466                     result = true;
00467                     gd.token = TK_LPAREN;
00468                     break;
00469                 case ')':
00470                     result = true;
00471                     gd.token = TK_RPAREN;
00472                     break;
00473                 case '[':
00474                     result = true;
00475                     gd.token = TK_LBRACK;
00476                     break;
00477                 case ']':
00478                     result = true;
00479                     gd.token = TK_RBRACK;
00480                     break;
00481                 case '{':
00482                     result = true;
00483                     gd.token = TK_LBRACE;
00484                     break;
00485                 case '}':
00486                     result = true;
00487                     gd.token = TK_RBRACE;
00488                     break;
00489                 case '#':
00490                     readChar( &c );
00491                     for( int idx = 0; is_ident_char( c ) || c == ':'; idx++ ) {
00492                         gd.buf[idx] = c;
00493                         gd.buf[idx + 1] = 0;
00494                         readChar( &c );
00495                     }
00496                     gd.pos--;
00497                     gd.token = TK_SYMBOL;
00498                     result = true;
00499                     break;
00500                 case '^':
00501                     result = true;
00502                     gd.token = TK_UARROW;
00503                     break;
00504                 case ':':
00505                     result = true;
00506                     gd.token = TK_COLON;
00507                     readChar( &c );
00508                     if( c == '=' ) {
00509                         gd.token = TK_ASSIGN;
00510                     }
00511                     else
00512                         gd.pos--;
00513                     break;
00514                 case '$':
00515                     result = true;
00516                     gd.token = TK_CHAR;
00517                     readChar( &c );
00518                     gd.buf[0] = c;
00519                     gd.buf[1] = 0;
00520                     break;
00521                 default:
00522                     gd.pos--;
00523                     break;
00524             }
00525         }
00526     }
00527     return result;
00528 }
```

5.2.3.13 object_new()

```
t_object* object_new (
            t_message_handler hdl )
```

new object

```
00014                                                   {
00015     t_object *result = talloc_zero( NULL, t_object );
00016     result->handler = hdl;
00017     return result;
00018 }
```

References s_object::handler.

Referenced by int_handler(), stream_handler(), string_handler(), and string_meta_handler().

5.2.3.14 object_send()

```
t_object* object_send (
            t_object * self,
            const char * sel,
            t_object ** args )
```

send a message to an object

```
00019                                                                                   {
00020     return o->handler( o, sel, args );
00021 }
```

Referenced by object_send_void(), and string_meta_handler().

5.2.3.15 object_send_void()

```
void object_send_void (
            t_object * self,
            const char * sel,
            t_object ** args )
```

send a message to an object ignoring the result

```
00022                                                                                   {
00023     t_object *x = object_send( o, sel, args );
00024     if( x != o ) {
00025 // fprintf( stderr, "send void %s leaves these parents:\n", sel );
00026 // talloc_show_parents( x, stderr );
00027     }
00028     talloc_unlink( NULL, x );
00029 }
```

References object_send().

Referenced by block_handler(), int_handler(), and string_handler().

5.2.3.16 parse_verbatim()

```
void parse_verbatim (
             char c )
```

parse the verbatim chars (not used)

```
00363                                {
00364     int i = 0;
00365     gd.buf[i] = 0;
00366     readChar( &c );
00367     while( c != '}' ) {
00368         gd.buf[i++] = c;
00369         gd.buf[i] = 0;
00370         readChar( &c );
00371     }
00372     gd.token = TK_VERBATIM;
00373 }
```

References gd::buf, readChar(), and gd::token.

5.2.3.17 readChar()

```
bool readChar (
             char * t )
```

read one character from input and store it somewhere.

read the next char from the source

Parameters

in	*t*	c-string of some sort.

Returns

true if successful

```
00331                            {
00332     bool result = true;
00333     if( gd.state == 0 ) {
00334         result = readLine(  );
00335     }
00336     if( result ) {
00337         *t = gd.line[gd.pos++];
00338         while( *t == 0 ) {
00339             if( readLine(  ) ) {
00340                 *t = gd.line[gd.pos++];
00341             }
00342             else {
00343                 result = false;
00344                 break;
00345             }
00346         }
00347     }
00348     return result;
00349 }
```

References gd::line, gd::pos, readLine(), and gd::state.

5.2.3.18 readLine()

```
bool readLine (
            void  )
```

read one line from stdin stores the result into gd.line.

read the next line from the source

trailing blanks are removed.

```
00308                   {
00309     if( gd.src_iter == NULL ) {
00310         gd.src_iter = itab_foreach( gd.src );
00311     }
00312     else {
00313         gd.src_iter = itab_next( gd.src_iter );
00314     }
00315     if( gd.src_iter ) {
00316         gd.line = itab_value( gd.src_iter );
00317         gd.line_count++;
00318         printf( "%2d:%s\n", gd.line_count, gd.line );
00319         gd.pos = 0;
00320         gd.state = 1;
00321         return true;
00322     }
00323     else {
00324         gd.line = "";
00325         gd.state = 2;
00326         return false;
00327     }
00328 }
```

References itab_foreach(), itab_next(), itab_value(), gd::line, gd::line_count, gd::pos, gd::src, gd::src_iter, and gd::state.

5.2.3.19 readStringToken()

```
bool readStringToken (
            void  )
```

read string token.

read a string token from the source

Returns

true if successful

```
00351                           {
00352     int idx = 0;
00353     char c;
00354     while( readChar( &c ) && '\"' != c ) {
00355         if( c == '\\' )
00356             readChar( &c );
00357         gd.buf[idx++] = c;
00358     }
00359     gd.buf[idx] = 0;
00360     gd.token = TK_STRING;
00361     return true;
00362 }
```

References gd::buf, readChar(), and gd::token.

5.2.3.20 require_classes()

```
void require_classes (
            void  )
```

require definition of classes

```
00009                           {
00010     if( !classes ) {
00011         classes = itab_new( );
00012         methods = itab_new( );
00013         method_names = itab_new( );
00014         variables = itab_new( );
00015         strings = itab_new( );
00016 
00017         gd.classnum = 1;
00018 
00019         class_enter( "Behavior" );
00020         class_enter( "Object" );
00021 
00022         string_class_num = gd.classnum - 1;
00023 
00024 
00025         gd.classnum = 100;
00026     }
00027 }
```

References class_enter(), gd::classnum, and itab_new().

Referenced by class_enter().

5.2.3.21 require_current_class()

```
void require_current_class (
            void  )
```

require that there is a current class

```
00003                                 {
00004     assert( current_class != NULL );
00005 }
```

Referenced by method_enter().

5.2.3.22 simulate()

```
t_object* simulate (
            t_env * env,
            t_statements * stmts )
```

simulate the execution of a list of statemtents

```
00404                                                                   {
00405     t_object *result = NULL;
00406     assert( stmts );
00407 
00408     msg_add( "simulate" );
00409 
00410     while( stmts ) {
00411         switch ( stmts->type ) {
00412             case stmt_message:
00413                 msg_add( "message stmt" );
00414                 result = eval( env, stmts->expr );
00415                 break;
00416             case stmt_return:
00417                 msg_add( "return stmt" );
```

```
00418                 result = eval( env, stmts->expr );
00419                 msg_add( "returning value and leaving method...\n" );
00420                 msg_print_last(  );
00421                 stmts = NULL;
00422                 break;
00423
00424             default:
00425                 msg_add( "error: unkonwn stmt type: %d\n", stmts->type );
00426                 msg_print_last(  );
00427                 abort(  );
00428                 break;
00429         }
00430         if( stmts )
00431             stmts = stmts->next;
00432     }
00433     return result;
00434 }
```

References msg_add(), and s_statements::type.

Referenced by block_handler(), and method_exec().

5.2.3.23 src_add()

```
bool src_add (
              const char * line )
```

add a line to the source table

adding one line to the source that will be parsed.

```
00264                                   {
00265     int n = itab_lines( gd.src );
00266     char buf[10];
00267     sprintf( buf, "%09d", n + 1 );
00268     itab_append( gd.src, buf, talloc_strdup( gd.src, line ) );
00269 }
```

References itab_append(), itab_lines(), and gd::src.

5.2.3.24 src_clear()

```
bool src_clear (
              void  )
```

clear the source table

clear and initialize the source that will alter be parsed.

needs to be called before using *src_add*. *src_read* will do it automatically.

```
00253                       {
00254     if( gd.src ) {
00255         talloc_free( gd.src );
00256     }
00257     gd.src = itab_new(  );
00258     if( gd.src_iter ) {
00259         talloc_free( gd.src_iter );
00260     }
00261     gd.src_iter = NULL;
00262 }
```

References itab_new(), gd::src, and gd::src_iter.

5.2.3.25 src_dump()

```
bool src_dump (
            void  )
```

dump the source table

dumps all the lines of the current source.

```
00295                   {
00296     for( struct itab_iter * x = itab_foreach( gd.src );
00297          x; x = itab_next( x ) ) {
00298         printf( "%s:%s\n", itab_key( x ), itab_value( x ) );
00299     }
00300 }
```

References itab_foreach(), itab_key(), itab_next(), itab_value(), and gd::src.

5.2.3.26 src_read()

```
bool src_read (
            const char * name )
```

read a line of the source table

read a file into src itab.

read file into itab.

read a file into src itab.

```
00274                                   {
00275     FILE *f = fopen( name, "r" );
00276     char buf[1000];
00277     char *line;
00278     int line_no = 1;
00279     src_clear(  );
00280     for( ;; ) {
00281         line = fgets( buf, sizeof( buf ), f );
00282         if( line == NULL )
00283             break;
00284         int n = strlen( line );
00285         while( n > 0 && isspace( line[--n] ) )
00286             line[n] = 0;
00287         char line_number[10];
00288         sprintf( line_number, "%09d", line_no );
00289         itab_append( gd.src, line_number, talloc_strdup( gd.src, line ) );
00290         line_no++;
00291     }
00292     fclose( f );
00293 }
```

References itab_append(), gd::src, and src_clear().

5.2.3.27 stream_handler()

```
t_object* stream_handler (
            t_object * self,
            const char * sel,
            t_object ** args )
```

handle stream messages

```
00115                                                                           {
00116     t_object *result = self;
00117     if( cstr_equals( "upTo:", sel ) ) {
00118         tt_assert( args[0]->handler = char_handler );
00119         char sep = args[0]->u.intval;
00120         int idx = self->u.vals.i[0];
00121         int start = idx;
00122         int max = self->u.vals.i[1];
00123         char *chars = ( char * )self->u.vals.p[0];
00124         while( idx < max ) {
00125             if( chars[idx] == sep )
00126                 break;
00127             idx++;
00128         }
00129         result = object_new( string_handler );
00130         int len = idx - start;
00131         result->u.data = talloc_zero_array( result, char, len + 1 );
00132         memcpy( result->u.data, chars + start, len );
00133         self->u.vals.i[0] = idx + 1;
00134     }
00135     else if( 0 == strcmp( "atEnd", sel ) ) {
00136         if( self->u.vals.i[0] < self->u.vals.i[1] )
00137             result = global.False;
00138         else
00139             result = global.True;
00140     }
00141     else if( 0 == strcmp( "next", sel ) ) {
00142         result = object_new( char_handler );
00143         result->u.intval = ( ( char * )self->u.vals.p[0] )[self->u.vals.i[0]];
00144         fprintf(stderr, "stream next <%c> at %d/%d [%p]\n", result->u.intval,
00145         self->u.vals.i[0],
00146         self->u.vals.i[1], self );
00147         self->u.vals.i[0]++;
00148     }
00149     else if( cstr_equals( "nextPut:", sel ) ) {
00150         tt_assert( args[0]->handler == char_handler );
00151         int idx = self->u.vals.i[0];
00152         int max = self->u.vals.i[1];
00153         char *chars = ( char * )self->u.vals.p[0];
00154         char c = args[0]->u.intval;
00155         fprintf(stderr, "stream put <%c> at %d/%d [%p]\n", c, idx, max, self);
00156         tt_assert(chars);
00157         if( max <= idx ) {
00158             max *= 2;
00159             chars = talloc_realloc( self, chars, char, max + 1 );
00160             tt_assert(chars);
00161             self->u.vals.i[1] = max;
00162             self->u.vals.p[0] = chars;
00163         }
00164         chars[idx] = args[0]->u.intval;
00165         self->u.vals.i[0] = idx + 1;
00166     }
00167     else if( cstr_equals( "dump", sel ) ) {
00168         msg_add( "Stream len:%d pos:%d", self->u.vals.i[1],
00169                  self->u.vals.i[0] );
00170     }
00171     else
00172         result = method_exec( self, "Stream", sel, args );
00173     return result;
00174 }
```

References char_handler(), cstr_equals(), s_object::data, s_globals::False, global, s_object::intval, method_exec(), msg_add(), object_new(), string_handler(), s_globals::True, and s_object::u.

Referenced by string_handler(), and string_meta_handler().

5.2.3.28 string_handler()

```
t_object* string_handler (
            t_object * self,
            const char * sel,
            t_object ** args )
```

handle string messages

```
00030                                                                                {
00031     t_object *result = self;
00032     const char *self_data = ( const char * )self->u.data;
00033     if( 0 == strcmp( sel, MSG_DUMP ) ) {
00034         msg_add( "str: '%s'", self_data );
00035     }
00036     else if( cstr_equals( "asString", sel ) ) {
00037 // all set...
00038     }
00039     else if( cstr_equals( "do:", sel ) ) {
00040         int n = strlen( self_data );
00041         for( int i = 0; i < n; i++ ) {
00042             t_object *r = object_new( char_handler );
00043             r->u.intval = self_data[i];
00044             object_send_void( args[0], "value:", &r );
00045         }
00046     }
00047     else if( 0 == strcmp( sel, "readStream" ) ) {
00048         t_object *result = object_new( stream_handler );
00049         result->u.vals.i[0] = 0;
00050         result->u.vals.i[1] = strlen( self->u.data );
00051         result->u.vals.p[0] = self->u.data;
00052         msg_add("stream for readStream: %p", result);
00053         return result;
00054     }
00055     else if( 0 == strcmp( sel, "species" ) ) {
00056         return global.String;
00057     }
00058     else if( 0 == strcmp( sel, "size" ) ) {
00059         result = object_new( int_handler );
00060         result->u.intval = strlen( self->u.data );
00061     }
00062     else {
00063         result = method_exec( self, "String", sel, args );
00064     }
00065     return result;
00066 }
```

References char_handler(), cstr_equals(), s_object::data, global, int_handler(), s_object::intval, method_exec(), msg_add(), MSG_DUMP, object_new(), object_send_void(), stream_handler(), s_globals::String, s_object::u, and s_object::vals.

Referenced by int_handler(), stream_handler(), and string_meta_handler().

5.2.3.29 string_meta_handler()

```
t_object* string_meta_handler (
            t_object * self,
            const char * sel,
            t_object ** args )
```

handle string meta messages

```
00004                                                {
00005     t_object *result = self;
00006     if( 0 == strcmp( "new:streamContents:", sel ) ) {
00007         int size = args[0]->u.intval;
00008         msg_add( "new string with size: %d", size );
00009         t_object *par[1];
00010         t_object *stream = object_new(stream_handler);
00011
00012         stream = object_new( stream_handler );
00013         stream->u.vals.i[0] = 0;
```

```
00014         stream->u.vals.i[1] = size;
00015         stream->u.vals.p[0] = talloc_array( stream, char, size + 1 );
00016         talloc_reference(self, stream);
00017         par[0] = stream;
00018         msg_add("stream for streamContent: %p",stream);
00019         t_object *o = object_send( args[1], "value:", par );
00020 // talloc_unlink(NULL, par[0] );
00021         result = object_new( string_handler );
00022         result->u.data = talloc_strdup( result, stream->u.vals.p[0] );
00023     }
00024     else
00025         result = method_exec( self, "StringMeta", sel, args );
00026 
00027     return result;
00028 }
```

References s_object::data, s_object::intval, method_exec(), msg_add(), object_new(), object_send(), stream_handler(), string_handler(), s_object::u, and s_object::vals.

5.2.4 Variable Documentation

5.2.4.1 global

```
struct s_globals global  [extern]
```

global

Referenced by char_handler(), stream_handler(), and string_handler().

5.3 ITab

Data Structures

- struct itab_entry

  structure of an entry in the itab.
- struct itab

  structure of itab
- struct itab_iter

  iterator over elements of an itab.

Functions

- int itab_lines (struct itab *itab)
- struct itab * itab_new ()

  create a new itab with default parameters.
- int itab_entry_cmp (const void *aptr, const void *bptr)

  compares the keys of two entries
- void itab_append (struct itab *itab, const char *key, void *value)
- void * itab_read (struct itab *itab, const char *key)
- void itab_dump (struct itab *itab)
- struct itab_iter * itab_foreach (struct itab *tab)
- struct itab_iter * itab_next (struct itab_iter *iter)
- void * itab_value (struct itab_iter *iter)
- const char * itab_key (struct itab_iter *iter)

5.3.1 Detailed Description

sorted list of structures -> tables with primary index

5.3.2 Function Documentation

5.3.2.1 itab_append()

```
void itab_append (
              struct itab * itab,
              const char * key,
              void * value )
```

append new line

```
00140                                                                   {
00141     assert( itab != NULL );
00142     if( itab->total == itab->used ) {
00143         itab->total *= 2;
00144         itab->rows =
00145                 talloc_realloc( itab, itab->rows, struct itab_entry,
00146                                 itab->total );
00147     }
00148     struct itab_entry *row = &itab->rows[itab->used];
00149     row->key = talloc_strdup( itab, key );
00150     row->value = value;
00151     itab->used++;
00152
00153     qsort( itab->rows,                          // base
00154            itab->used,                          // nmemb
00155            sizeof( struct itab_entry ),         // size
00156            itab_entry_cmp );
00157 }
```

References itab_entry_cmp(), itab_entry::key, rows, total, used, and itab_entry::value.

Referenced by class_enter(), method_enter(), src_add(), and src_read().

5.3.2.2 itab_dump()

```
void itab_dump (
              struct itab * itab )
```

dump content to output

```
00174                                   {
00175     assert( itab );
00176     for( int i = 0; i < itab->used; i++ ) {
00177         fprintf( stderr, "%s: %p\n", itab->rows[i].key, itab->rows[i].value );
00178     }
00179 }
```

References itab_entry::key, rows, used, and itab_entry::value.

5.3.2.3 itab_entry_cmp()

```
int itab_entry_cmp (
            const void * aptr,
            const void * bptr )
```

compares the keys of two entries

compare entries

Returns

- $<$ 0, when first key is lower
- == 0, when both keys are equal
- $>$ 0, when second key is lower

```
00134                                                                   {
00135     const struct itab_entry *a = aptr;
00136     const struct itab_entry *b = bptr;
00137     return strcmp( a->key, b->key );
00138 }
```

References itab_entry::key.

Referenced by itab_append(), and itab_read().

5.3.2.4 itab_foreach()

```
struct itab_iter * itab_foreach (
            struct itab * tab )
```

start iteration

```
00181                                                       {
00182     if( tab->used > 0 ) {
00183         struct itab_iter *r = talloc_zero( NULL, struct itab_iter );
00184         r->tab = tab;
00185         r->pos = 0;
00186         return r;
00187     }
00188     else
00189         return NULL;
00190 }
```

References itab_iter::pos, itab_iter::tab, and used.

Referenced by readLine(), and src_dump().

5.3.2.5 itab_key()

```
const char * itab_key (
            struct itab_iter * iter )
```

key of current iterator

```
00207                                               {
00208     return iter->tab->rows[iter->pos].key;
00209 }
```

References itab_entry::key, itab_iter::pos, rows, and itab_iter::tab.

Referenced by src_dump().

5.3.2.6 itab_lines()

```
unsigned itab_lines (
            struct itab * itab )
```

how many lines?

returns the number of lines in the table

```
00100                                           {
00101     assert( itab != NULL );
00102     return itab->used;
00103 }
```

References used.

Referenced by src_add().

5.3.2.7 itab_new()

```
struct itab * itab_new (
            void  )
```

create a new itab with default parameters.

new

Returns

reference to an itab structure.

Detailed description follows here.

```
00120                           {
00121     struct itab *r = talloc_zero( NULL, struct itab );
00122     r->total = 10;
00123     r->used = 0;
00124     r->rows = talloc_array( r, struct itab_entry, r->total );
00125     return r;
00126 }
```

References rows, total, and used.

Referenced by require_classes(), and src_clear().

5.3.2.8 itab_next()

```
struct itab_iter * itab_next (
            struct itab_iter * iter )
```

cycle through iterator

```
00192                                                   {
00193     iter->pos++;
00194     if( iter->tab->used > iter->pos ) {
00195         return iter;
00196     }
00197     else {
00198         talloc_free( iter );
00199         return NULL;
00200     }
00201 }
```

References itab_iter::pos, itab_iter::tab, and used.

Referenced by readLine(), and src_dump().

5.3.2.9 itab_read()

```
void * itab_read (
            struct itab * itab,
            const char * key )
```

find a line by key

```
00159                                                   {
00160     assert( itab );
00161     assert( key );
00162     struct itab_entry dummy = { key, NULL };
00163     struct itab_entry *r = bsearch( &dummy,
00164                                     itab->rows,
00165                                     itab->used,
00166                                     sizeof( struct itab_entry ),
00167                                     itab_entry_cmp );
00168     if( r )
00169         return r->value;
00170     else
00171         return NULL;
00172 }
```

References itab_entry_cmp(), itab_entry::key, rows, used, and itab_entry::value.

Referenced by class_enter(), and method_enter().

5.3.2.10 itab_value()

```
void * itab_value (
            struct itab_iter * iter )
```

value of current iterator

```
00203                                           {
00204     return iter->tab->rows[iter->pos].value;
00205 }
```

References itab_iter::pos, rows, itab_iter::tab, and itab_entry::value.

Referenced by readLine(), and src_dump().

5.4 Tokenizer

Functions

- bool is_ident_char (int c)

  check if character is part of an identifier.
- bool is_binary_char (int c)
- bool src_clear ()
- bool src_add (const char *line)
- bool src_read (const char *name)
- bool src_dump ()
- bool readLine ()

  read one line from stdin stores the result into `gd.line`.
- bool readChar (char *t)

  read one character from input and store it somewhere.
- bool readStringToken (void)

  read string token.
- void parse_verbatim (char c)
- bool nextToken (void)

  read next token.

5.4.1 Detailed Description

convert stdin into tokens. each token is returned by the call to

See also

nextToken.

5.4.2 Function Documentation

5.4.2.1 is_binary_char()

```
bool is_binary_char (
            int c )
```

binary chars are special ones for binary message names

```
00228                                   {
00229     switch ( c ) {
00230         case '!':
00231         case '%':
00232         case '&':
00233         case '*':
00234         case '+':
00235         case ',':
00236         case '/':
00237         case '<':
00238         case '=':
00239         case '>':
00240         case '?':
00241         case '@':
00242         case '\\':
00243         case '~':
00244         case '|':                               // sollte laut Vorschlag ein Binary Operator sein.
       Kollidiert aber mit der temporary declaration.
00245 // das muss dann wohl auf der Syntaxebene geklärt werden.
00246         case '-':
00247             return true;
00248         default:
00249             return false;
00250     }
00251 }
```

Referenced by nextToken().

5.4.2.2 is_ident_char()

```
bool is_ident_char (
            int c )
```

check if character is part of an identifier.

Parameters

in	*c*	character to classify.

Returns

true if if c is an identifier character.

```
00223                               {
00224     return isalpha( c ) || isdigit( c ) || c == '_';
00225 }
```

Referenced by nextToken().

5.4.2.3 nextToken()

```
bool nextToken ( )
```

read next token.

This is a more detailed description.

Returns

true if successful

```
00378                     {
00379     char c;
00380     bool result = false;
00381     while( true ) {
00382         while( readChar( &c ) && isspace( c ) );
00383         if( c == '"' ) {
00384             while( readChar( &c ) && c != '"' );
00385         }
00386         else
00387             break;
00388     }
00389     if( gd.state == 1 ) {
00390         if( isalpha( c ) ) {
00391             int idx = 0;
00392             for( ;; ) {
00393                 gd.buf[idx++] = c;
00394                 readChar( &c );
00395                 if( !is_ident_char( c ) )
00396                     break;
00397             }
00398             if( c == ':' ) {
00399                 gd.buf[idx++] = c;
00400                 gd.token = TK_KEYWORD;
00401             }
00402             else {
00403                 gd.pos--;
00404                 gd.token = TK_IDENT;
00405             }
00406             gd.buf[idx] = 0;
00407             result = true;
00408         }
00409         else if( is_binary_char( c ) ) {
00410             for( int idx = 0; is_binary_char( c ); idx++ ) {
00411                 gd.buf[idx] = c;
00412                 gd.buf[idx + 1] = 0;
00413                 readChar( &c );
00414             }
00415             gd.pos--;
00416             gd.token = 0;
00417             gd.token = TK_BINOP;
00418             result = true;
00419             if( strcmp( ":=", gd.buf ) == 0 ) {
00420                 gd.token = TK_ASSIGN;
00421                 result = true;
00422             }
00423             else if( strcmp( "<-", gd.buf ) == 0 ) {
00424                 gd.token = TK_LARROW;
00425                 result = true;
00426             }
00427             else if( strcmp( "|", gd.buf ) == 0 ) {
00428                 gd.token = TK_BAR;
00429                 result = true;
00430             }
```

```
00431             else if( 0 == strcmp( "<", gd.buf ) ) {
00432                 gd.token = TK_LT;
00433                 result = true;
00434             }
00435             else if( 0 == strcmp( ">", gd.buf ) ) {
00436                 gd.token = TK_GT;
00437                 result = true;
00438             }
00439         }
00440         else if( isdigit( c ) ) {
00441             int idx = 0;
00442             while( isdigit( c ) ) {
00443                 printf( "### digit %c\n", c );
00444                 gd.buf[idx++] = c;
00445                 gd.buf[idx] = 0;
00446                 readChar( &c );
00447             }
00448             gd.pos--;
00449             gd.token = TK_NUMBER;
00450             result = true;
00451         }
00452         else {
00453             switch ( c ) {
00454                 case '\'':
00455                     result = readStringToken(  );
00456                     break;
00457                 case '.':
00458                     result = true;
00459                     gd.token = TK_DOT;
00460                     break;
00461                 case ';':
00462                     result = true;
00463                     gd.token = TK_SEMICOLON;
00464                     break;
00465                 case '(':
00466                     result = true;
00467                     gd.token = TK_LPAREN;
00468                     break;
00469                 case ')':
00470                     result = true;
00471                     gd.token = TK_RPAREN;
00472                     break;
00473                 case '[':
00474                     result = true;
00475                     gd.token = TK_LBRACK;
00476                     break;
00477                 case ']':
00478                     result = true;
00479                     gd.token = TK_RBRACK;
00480                     break;
00481                 case '{':
00482                     result = true;
00483                     gd.token = TK_LBRACE;
00484                     break;
00485                 case '}':
00486                     result = true;
00487                     gd.token = TK_RBRACE;
00488                     break;
00489                 case '#':
00490                     readChar( &c );
00491                     for( int idx = 0; is_ident_char( c ) || c == ':'; idx++ ) {
00492                         gd.buf[idx] = c;
00493                         gd.buf[idx + 1] = 0;
00494                         readChar( &c );
00495                     }
00496                     gd.pos--;
00497                     gd.token = TK_SYMBOL;
00498                     result = true;
00499                     break;
00500                 case '^':
00501                     result = true;
00502                     gd.token = TK_UARROW;
00503                     break;
00504                 case ':':
00505                     result = true;
00506                     gd.token = TK_COLON;
00507                     readChar( &c );
00508                     if( c == '=' ) {
00509                         gd.token = TK_ASSIGN;
00510                     }
00511                     else
00512                         gd.pos--;
00513                     break;
00514                 case '$':
00515                     result = true;
00516                     gd.token = TK_CHAR;
00517                     readChar( &c );
```

```
00518                     gd.buf[0] = c;
00519                     gd.buf[1] = 0;
00520                     break;
00521                 default:
00522                     gd.pos--;
00523                     break;
00524             }
00525         }
00526     }
00527     return result;
00528 }
```

References gd::buf, is_binary_char(), is_ident_char(), gd::pos, readChar(), readStringToken(), gd::state, and gd::token.

5.4.2.4 parse_verbatim()

```
void parse_verbatim (
              char c )
```

parse the verbatim chars (not used)

```
00363                                {
00364     int i = 0;
00365     gd.buf[i] = 0;
00366     readChar( &c );
00367     while( c != '}' ) {
00368         gd.buf[i++] = c;
00369         gd.buf[i] = 0;
00370         readChar( &c );
00371     }
00372     gd.token = TK_VERBATIM;
00373 }
```

References gd::buf, readChar(), and gd::token.

5.4.2.5 readChar()

```
bool readChar (
              char * t )
```

read one character from input and store it somewhere.

Parameters

in	*t*	c-string of some sort.

Returns

true if successful

```
00331                           {
00332     bool result = true;
00333     if( gd.state == 0 ) {
00334         result = readLine(  );
00335     }
00336     if( result ) {
00337         *t = gd.line[gd.pos++];
00338         while( *t == 0 ) {
```

```
00339             if( readLine(  ) ) {
00340                 *t = gd.line[gd.pos++];
00341             }
00342             else {
00343                 result = false;
00344                 break;
00345             }
00346         }
00347     }
00348     return result;
00349 }
```

References gd::line, gd::pos, readLine(), and gd::state.

Referenced by nextToken(), parse_verbatim(), and readStringToken().

5.4.2.6 readLine()

```
bool readLine ( )
```

read one line from stdin stores the result into `gd.line`.

trailing blanks are removed.

```
00308                   {
00309     if( gd.src_iter == NULL ) {
00310         gd.src_iter = itab_foreach( gd.src );
00311     }
00312     else {
00313         gd.src_iter = itab_next( gd.src_iter );
00314     }
00315     if( gd.src_iter ) {
00316         gd.line = itab_value( gd.src_iter );
00317         gd.line_count++;
00318         printf( "%2d:%s\n", gd.line_count, gd.line );
00319         gd.pos = 0;
00320         gd.state = 1;
00321         return true;
00322     }
00323     else {
00324         gd.line = "";
00325         gd.state = 2;
00326         return false;
00327     }
00328 }
```

References itab_foreach(), itab_next(), itab_value(), gd::line, gd::line_count, gd::pos, gd::src, gd::src_iter, and gd::state.

Referenced by readChar().

5.4.2.7 readStringToken()

```
bool readStringToken (
             void  )
```

read string token.

Returns

true if successful

```
00351                           {
00352     int idx = 0;
00353     char c;
00354     while( readChar( &c ) && '\"' != c ) {
00355         if( c == '\\' )
00356             readChar( &c );
00357         gd.buf[idx++] = c;
00358     }
00359     gd.buf[idx] = 0;
00360     gd.token = TK_STRING;
00361     return true;
00362 }
```

References gd::buf, readChar(), and gd::token.

Referenced by nextToken().

5.4.2.8 src_add()

```
bool src_add (
             const char *  )
```

adding one line to the source that will be parsed.

```
00264                                   {
00265     int n = itab_lines( gd.src );
00266     char buf[10];
00267     sprintf( buf, "%09d", n + 1 );
00268     itab_append( gd.src, buf, talloc_strdup( gd.src, line ) );
00269 }
```

References itab_append(), itab_lines(), and gd::src.

5.4.2.9 src_clear()

```
bool src_clear ( )
```

clear and initialize the source that will alter be parsed.

needs to be called before using *src_add*. *src_read* will do it automatically.

```
00253                       {
00254     if( gd.src ) {
00255         talloc_free( gd.src );
00256     }
00257     gd.src = itab_new(  );
00258     if( gd.src_iter ) {
00259         talloc_free( gd.src_iter );
00260     }
00261     gd.src_iter = NULL;
00262 }
```

References itab_new(), gd::src, and gd::src_iter.

Referenced by src_read().

5.4.2.10 src_dump()

```
bool src_dump ( )
```

dumps all the lines of the current source.

```
00295                    {
00296     for( struct itab_iter * x = itab_foreach( gd.src );
00297          x; x = itab_next( x ) ) {
00298         printf( "%s:%s\n", itab_key( x ), itab_value( x ) );
00299     }
00300 }
```

References itab_foreach(), itab_key(), itab_next(), itab_value(), and gd::src.

5.4.2.11 src_read()

```
bool src_read (
             const char * name )
```

read file into itab.

read a file into src itab.

```
00274                                    {
00275     FILE *f = fopen( name, "r" );
00276     char buf[1000];
00277     char *line;
00278     int line_no = 1;
00279     src_clear(  );
00280     for( ;; ) {
00281         line = fgets( buf, sizeof( buf ), f );
00282         if( line == NULL )
00283             break;
00284         int n = strlen( line );
00285         while( n > 0 && isspace( line[--n] ) )
00286             line[n] = 0;
00287         char line_number[10];
00288         sprintf( line_number, "%09d", line_no );
00289         itab_append( gd.src, line_number, talloc_strdup( gd.src, line ) );
00290         line_no++;
00291     }
00292     fclose( f );
00293 }
```

References itab_append(), gd::src, and src_clear().

5.5 Messages

Data Structures

- struct **s_msgs**

Macros

- #define MSG_LOG_LEN 200

Typedefs

- typedef char t_msg[200]

Functions

- void msg_init ()
- void msg_add (const char *msg,...)
- void msg_print_last ()

5.5.1 Detailed Description

5.5.2 Macro Definition Documentation

5.5.2.1 MSG_LOG_LEN

```
#define MSG_LOG_LEN 200
```

length of log

5.5.3 Typedef Documentation

5.5.3.1 t_msg

```
typedef char t_msg[200]
```

contains a log line

5.5.4 Function Documentation

5.5.4.1 msg_add()

```
void msg_add (
             const char * msg,
              ...  )
```

adding a message

```
00560                                    {
00561     va_list ap;
00562     msg_init(  );
00563     va_start( ap, msg );
00564
00565     vsnprintf( msgs.msgs[msgs.pos], 199, msg, ap );
00566     msgs.pos = ( msgs.pos + 1 ) % msgs.size;
00567     va_end( ap );
00568 }
```

References msg_init().

Referenced by block_handler(), char_handler(), int_handler(), method_exec(), simulate(), stream_handler(), string_handler(), and string_meta_handler().

5.5.4.2 msg_init()

```
void msg_init ( )
```

initialize application messages

```
00552                   {
00553     if( msgs.size != MSG_LOG_LEN ) {
00554         msgs.size = MSG_LOG_LEN;
00555         msgs.pos = 0;
00556     }
00557 }
```

Referenced by msg_add().

5.5.4.3 msg_print_last()

```
void msg_print_last ( )
```

print the last messages from the log

```
00571                       {
00572     printf( "------------------------------------\n" );
00573     const char *fmt = "%03d --- %s\n";
00574     int n = 1;
00575     for( int i = msgs.pos; i < msgs.size; i++ ) {
00576         if( msgs.msgs[i][0] )
00577             printf( fmt, n++, msgs.msgs[i] );
00578         msgs.msgs[i][0] = 0;
00579     }
00580     for( int i = 0; i < msgs.pos; i++ ) {
00581         if( msgs.msgs[i][0] )
00582             printf( fmt, n++, msgs.msgs[i] );
00583         msgs.msgs[i][0] = 0;
00584     }
00585 }
```

Referenced by method_exec().

5.6 Syntax Messages

Functions

- void message_add_msg (t_messages *ms, t_messages *m)

5.6.1 Detailed Description

5.6.2 Function Documentation

5.6.2.1 message_add_msg()

```
void message_add_msg (
            t_messages * ms,
            t_messages * m )
```

add a message

```
00596                                                   {
00597     while( ms->next )
00598         ms = ms->next;
00599     ms->next = m;
00600 }
```

References s_messages::next.

5.7 Name List

Data Structures

- struct s_namelist
- struct s_names

Typedefs

- typedef const char * t_name
- typedef struct s_namelist t_namelist
- typedef struct s_names * t_names

Functions

- void namelist_init (t_namelist *nl)

  clear the structure for further usage.
- void namelist_add (t_namelist *nl, const t_name name)
- void namelist_copy (t_namelist *to, t_namelist *from)

5.7.1 Detailed Description

list of names are very common. Basically they are a counter for the list length and an array of const char pointers. These functions help to build up these structures the pointer array and the counter are held in a structure that is not allocated here but is a reglar part of another structure.

5.7.2 Typedef Documentation

5.7.2.1 t_name

```
typedef const char* t_name
```

name

5.7.2.2 t_namelist

```
typedef struct s_namelist t_namelist
```

structure containing the counter and array of names.

5.7.2.3 t_names

```
typedef struct s_names* t_names
```

array of names

5.7.3 Function Documentation

5.7.3.1 namelist_add()

```
void namelist_add (
            t_namelist * nl,
            const t_name name )
```

adding a name to the name list. Memory will be allocated by the name list and also the name will be copied. The paramter can safely being freed after this call.

Parameters

nl	the modified list
name	the string to be added

```
00027                                                   {
00028     nl->count++;
00029     nl->names = talloc_realloc( NULL, nl->names, t_name, nl->count );
00030     nl->names[nl->count - 1] = talloc_strdup( nl->names, name );
00031 }
```

References s_namelist::count, and s_namelist::names.

5.7.3.2 namelist_copy()

```
void namelist_copy (
            t_namelist * to,
            t_namelist * from )
```

make a deep copy of a name list

Parameters

to	the target name list, which doesn't need to be initialized
from	the source to be copied.

```
00036                                                       {
00037     to->count = from->count;
00038     to->names = talloc_array( NULL, t_name, to->count );
00039     assert(talloc_get_type(from->names, t_name));
00040     for( int i = 0; i < to->count; i++ ) {
00041         to->names[i] = talloc_strdup( to->names, from->names[i] );
00042     }
00043     assert(talloc_get_type(to->names, t_name));
00044 }
```

References s_namelist::count, and s_namelist::names.

Referenced by method_enter().

5.7.3.3 namelist_init()

```
void namelist_init (
             t_namelist * nl )
```

clear the structure for further usage.

The *namelist* itself is not allocated but could be part of an already allocated structure.

Parameters

nl	reference to an existing structure to be initialized.

```
00017                                       {
00018     nl->count = 0;
00019     nl->names = NULL;
00020 }
```

References s_namelist::count, and s_namelist::names.

5.8 Internal_structures

Data Structures

- struct s_expression_list
- struct s_pattern
- struct s_classdef
- struct s_statements
- struct s_methoddef
- struct s_message_pattern
- struct s_assignment
- struct s_block
- struct s_expression
- struct s_messages
- struct s_message_cascade
- struct s_object
- struct s_slot
- struct s_env

Typedefs

- typedef struct s_expression_list t_expression_list
- typedef struct s_pattern * t_pattern
- typedef struct s_classdef t_classdef
- typedef enum e_statement_type t_statement_type
- typedef struct s_statements t_statements
- typedef struct s_methoddef t_methoddef
- typedef struct s_message_pattern t_message_pattern
- typedef enum e_expression_tag t_expression_tag
- typedef struct s_assignment t_assignment
- typedef struct s_block t_block
- typedef struct s_expression t_expression
- typedef struct s_messages t_messages
- typedef struct s_message_cascade t_message_cascade
- typedef struct s_object *(* t_message_handler) (struct s_object *, const char *sel, struct s_object **args)
- typedef struct s_object t_object
- typedef struct s_slot t_slot
- typedef struct s_env t_env

Enumerations

- enum e_statement_type { **stmt_return** = 100 , **stmt_assign** , **stmt_message** }
- enum e_expression_tag {
  tag_string , **tag_char** , **tag_message** , **tag_number** ,
  tag_ident , **tag_block** , **tag_array** , **tag_assignment** }

5.8.1 Detailed Description

5.8.2 Typedef Documentation

5.8.2.1 t_assignment

```
typedef struct s_assignment t_assignment
```

an assignment consists of a target name and an expression

5.8.2.2 t_block

```
typedef struct s_block t_block
```

a block has parameters, locals and statments the environment should not be used anymore.

5.8.2.3 t_classdef

```
typedef struct s_classdef t_classdef
```

a class has an id (not pouplated right now) also consists of a name, the name of the metaclass and the name of the super class. the environment should not be used anymore.

5.8.2.4 t_env

```
typedef struct s_env t_env
```

an environment is a list of slots. and also points to a lower environment. The names in this environment supersede the ones in the lower environments.

5.8.2.5 t_expression

```
typedef struct s_expression t_expression
```

an expression is either an integer, a string, a symbol, an assignment, a block, or a message call

5.8.2.6 t_expression_list

```
typedef struct s_expression_list t_expression_list
```

a list of expressions

5.8.2.7 t_expression_tag

```
typedef enum e_expression_tag t_expression_tag
```

expression type

5.8.2.8 t_message_cascade

```
typedef struct s_message_cascade t_message_cascade
```

a cascade of messages to the same target. is it still used somewhere?

5.8.2.9 t_message_handler

```
typedef struct s_object*( * t_message_handler) (struct s_object *, const char *sel, struct
s_object **args)
```

message handler

5.8.2.10 t_message_pattern

```
typedef struct s_message_pattern t_message_pattern
```

message pattern as it is parsed.

5.8.2.11 t_messages

```
typedef struct s_messages t_messages
```

message calling structure

5.8.2.12 t_methoddef

```
typedef struct s_methoddef t_methoddef
```

method definition

5.8.2.13 t_object

```
typedef struct s_object t_object
```

an object contains mainly of the handler function and some data area. broken down into some specific alternatives for convenience. (but are not so convenient)

5.8.2.14 t_pattern

```
typedef struct s_pattern* t_pattern
```

patterns

5.8.2.15 t_slot

```
typedef struct s_slot t_slot
```

one slot of the environment. A linked list of name and value pairs. Values are expressed as objects. Names have to be cstrings.

5.8.2.16 t_statement_type

```
typedef enum e_statement_type t_statement_type
```

statement type

5.8.2.17 t_statements

```
typedef struct s_statements t_statements
```

linked list of statements. The type, expression and the next statments.

5.8.3 Enumeration Type Documentation

5.8.3.1 e_expression_tag

```
enum e_expression_tag
```

expression type

```
00109                               {
00110     tag_string,
00111     tag_char,
00112     tag_message,
00113     tag_number,
00114     tag_ident,
00115     tag_block,
00116     tag_array,
00117     tag_assignment
00118 } t_expression_tag;
```

5.8.3.2 e_statement_type

```
enum e_statement_type
```

statement type

```
00073                               {
00074     stmt_return = 100,
00075     stmt_assign,
00076     stmt_message
00077 } t_statement_type;
```

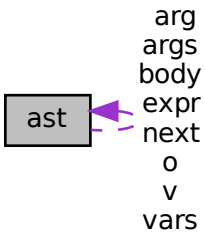
Chapter 6

Data Structure Documentation

6.1 ast Struct Reference

`#include <global.h>`

Collaboration diagram for ast:



Data Fields

- int tag

  *descriminator for the union, tags start with AST_*
- union {
  struct {
  char * v

  string value owned by the syntax tree
  } str

  string node
  struct {
  char * v

  id value owned by the syntax tree
  } id

  id node
  struct {

struct ast * o
method target
char * sel
selector
struct ast * arg
list of arguments
} unary
unary method call node
struct {
struct ast * v
argument value node
struct ast * next
next argument
} **arg**
struct **argdef** {
const char * key
const char * name
parameter name
struct ast * next
next keyword in the list
} **argdef**
struct {
struct ast * **v**
struct ast * **next**
} **stmt**
struct {
char * **var**
struct ast * **expr**
} **asgn**
struct {
char * **name**
char * **super**
int **num**
struct ast * **vars**
struct ast * **next**
} **cls**
struct {
char * **v**
struct ast * **next**
} **names**
struct {
const char * **name**
struct ast * **args**
char * **classname**
char * **src**
struct ast * **body**
struct ast * **next**
} **methods**
} u

6.1.1 Detailed Description

old structure for the abstract syntax tree. shouldn't be used anymore.

6.1.2 Field Documentation

6.1.2.1 key

```
const char* ast::key
```

Keyword including the colon at the end if it is no keyword then the plain unary or binary name is here.

6.1.2.2 next

```
struct ast* ast::next
```

next argument

next keyword in the list

6.1.2.3

```
union { ... } ast::u
```

union

6.1.2.4 v

```
char* ast::v
```

string value owned by the syntax tree

id value owned by the syntax tree

The documentation for this struct was generated from the following file:

- global.h

6.2 classinfo Struct Reference

Data Fields

- bool meta
- char * name
- char * super
- int num

6.2.1 Detailed Description

details of a class

6.2.2 Field Documentation

6.2.2.1 meta

```
bool classinfo::meta
```

is it a meta class

6.2.2.2 name

```
char* classinfo::name
```

name

6.2.2.3 num

```
int classinfo::num
```

number for identification

6.2.2.4 super

```
char* classinfo::super
```

name of super class

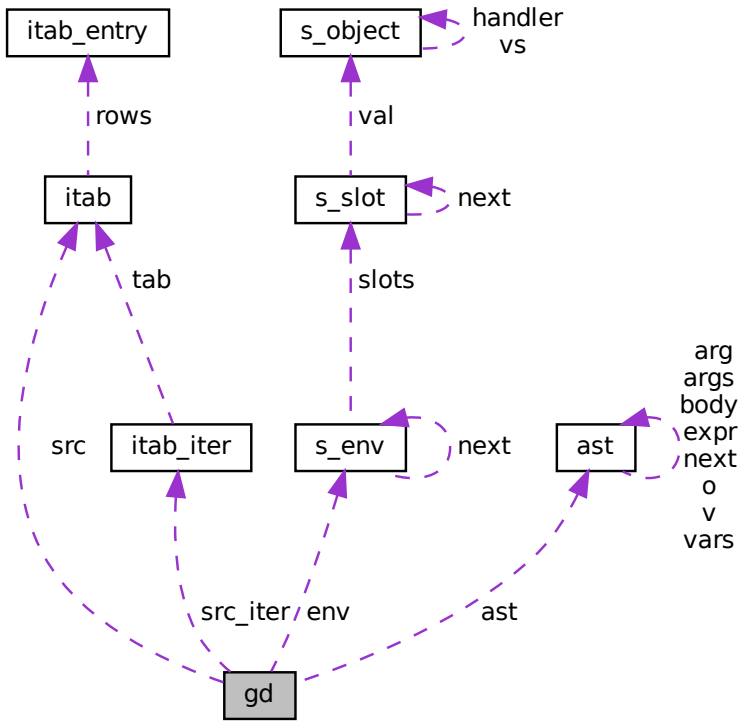
The documentation for this struct was generated from the following file:

- lib.c

6.3 gd Struct Reference

`#include <global.h>`

Collaboration diagram for gd:



Data Fields

- int state
  0 - init, 1 - running, 2 - end
- int paridx
  ...
- int token
  ...
- int pos
  ...
- char buf [50]
  ...
- char * line
  ...
- int line_count
  ...
- struct ast * ast

...

- int classnum

  ...

- struct itab * src

  ...

- struct itab_iter * src_iter

  ...

- struct s_env * env

  ...

6.3.1 Detailed Description

structure containing global data

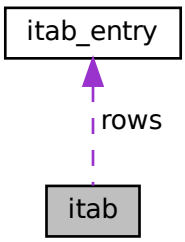
The documentation for this struct was generated from the following file:

- global.h

6.4 itab Struct Reference

structure of itab

Collaboration diagram for itab:



Data Fields

- unsigned total

  total number of available entries

- unsigned used

  actual used number of entries

- struct itab_entry * rows

  array of all entries

6.4.1 Detailed Description

structure of itab

The documentation for this struct was generated from the following file:

- lib.c

6.5 itab_entry Struct Reference

structure of an entry in the itab.

Data Fields

- const char * key
  key
- void * value
  binary value

6.5.1 Detailed Description

structure of an entry in the itab.

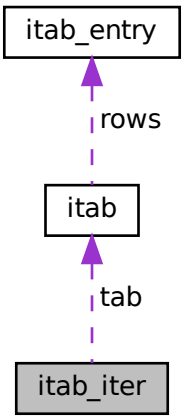
The documentation for this struct was generated from the following file:

- lib.c

6.6 itab_iter Struct Reference

iterator over elements of an itab.

Collaboration diagram for itab_iter:



Data Fields

- struct itab ∗ tab
  table to be used
- unsigned pos
  current position in the table

6.6.1 Detailed Description

iterator over elements of an itab.

The documentation for this struct was generated from the following file:

- lib.c

6.7 methodinfo Struct Reference

Data Fields

- char ∗ classname
  name of the class
- char ∗ name
  name of the method

6.7.1 Detailed Description

details of a method

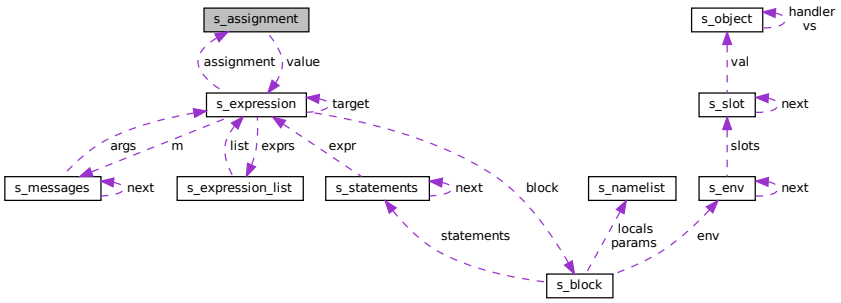
The documentation for this struct was generated from the following file:

- lib.c

6.8 s_assignment Struct Reference

`#include <lib.h>`

Collaboration diagram for s_assignment:



Data Fields

- const char * target
  name that should be assigned
- struct s_expression * value
  value that is evaluated and assigned to the name

6.8.1 Detailed Description

an assignment consists of a target name and an expression

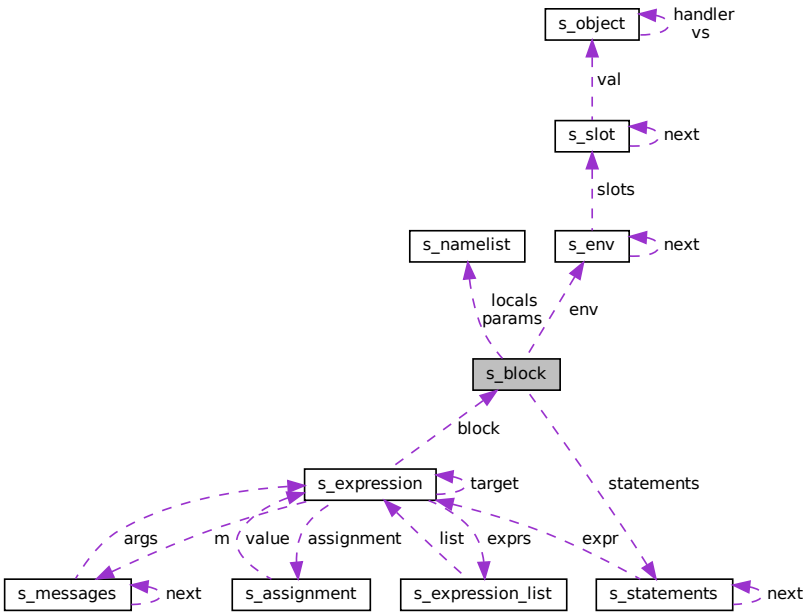
The documentation for this struct was generated from the following file:

- lib.h

6.9 s_block Struct Reference

`#include <lib.h>`

Collaboration diagram for s_block:



Data Fields

- t_namelist params
  list of defined parameters
- t_namelist locals
  list of local variables
- t_statements * statements
  statements
- struct s_env * env
  unused

6.9.1 Detailed Description

a block has parameters, locals and statments the environment should not be used anymore.

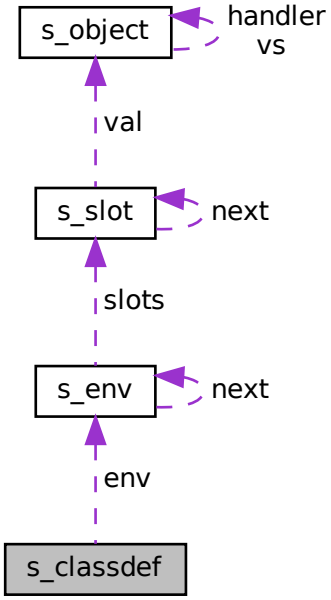
The documentation for this struct was generated from the following file:

- lib.h

6.10 s_classdef Struct Reference

`#include <lib.h>`

Collaboration diagram for s_classdef:



Data Fields

- int id
  number for identification (not yet used)
- char * name
  name
- char * meta
  name of meta class
- char * super
  name of super class
- struct s_env * env
  unused

6.10.1 Detailed Description

a class has an id (not pouplated right now) also consists of a name, the name of the metaclass and the name of the super class. the environment should not be used anymore.

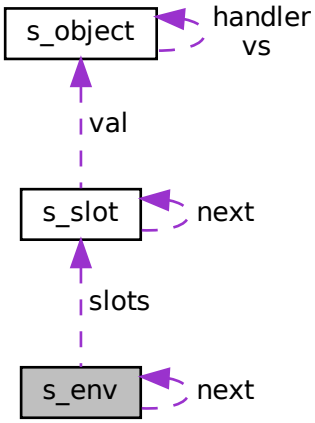
The documentation for this struct was generated from the following file:

- lib.h

6.11 s_env Struct Reference

`#include <lib.h>`

Collaboration diagram for s_env:



Data Fields

- t_slot * slots
  slots
- struct s_env * next
  next environment

6.11.1 Detailed Description

an environment is a list of slots. and also points to a lower environment. The names in this environment supersede the ones in the lower environments.

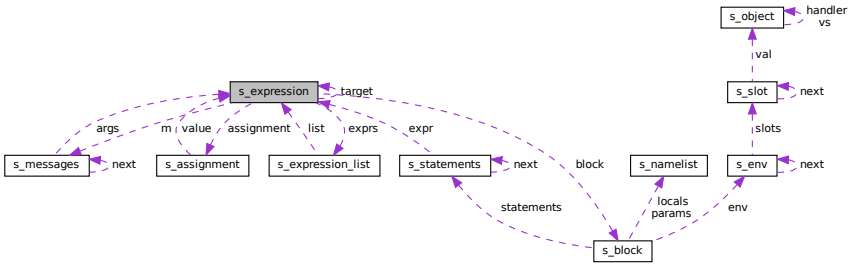
The documentation for this struct was generated from the following file:

- lib.h

6.12 s_expression Struct Reference

`#include <lib.h>`

Collaboration diagram for s_expression:



Data Fields

- t_expression_tag tag
  tag
- union {
  int intvalue
  integer value
  const char * strvalue
  string value
  const char * ident
  symbol
  t_expression_list exprs
  expression list
  struct **msg** {
  struct s_expression * target
  target
  struct s_messages * m
  messages
  } **msg**
  t_assignment assignment
  assignment
  t_block block
  block
  } u

  union

6.12.1 Detailed Description

an expression is either an integer, a string, a symbol, an assignment, a block, or a message call

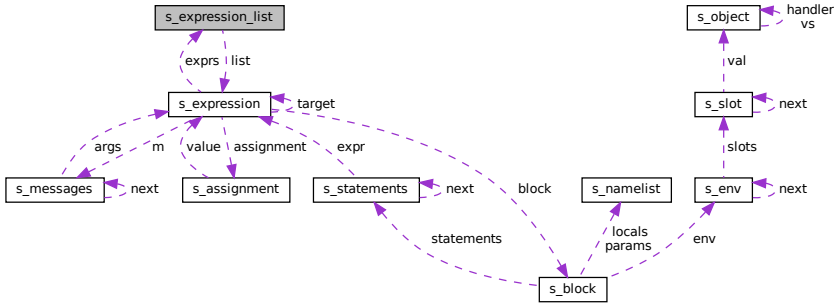
The documentation for this struct was generated from the following file:

- lib.h

6.13 s_expression_list Struct Reference

`#include <lib.h>`

Collaboration diagram for s_expression_list:



Data Fields

- int count

  number of expression defined
- struct s_expression ** list

  list

6.13.1 Detailed Description

a list of expressions

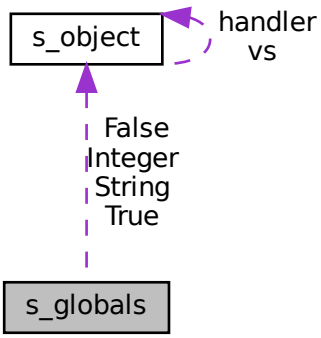
The documentation for this struct was generated from the following file:

- lib.h

6.14 s_globals Struct Reference

`#include <internal.h>`

Collaboration diagram for s_globals:



Data Fields

- t_object * String
  reference to the string meta object
- t_object * Integer
  reference to the integer meta object
- t_object * True
  reference to the True object
- t_object * False
  reference to the False object

6.14.1 Detailed Description

global definitions structure, references objects to be used somehwere in the code. these objects or classes are predefined ones

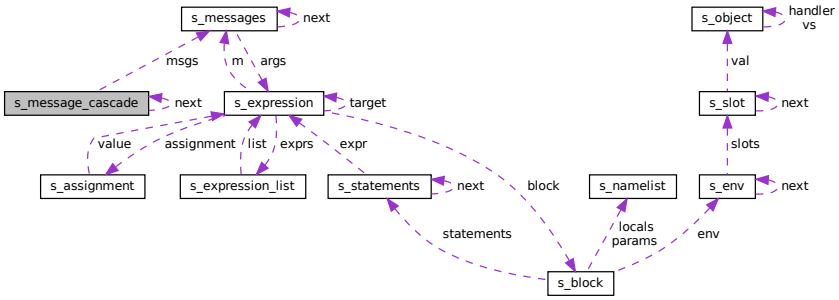
The documentation for this struct was generated from the following file:

- internal.h

6.15 s_message_cascade Struct Reference

```
#include <lib.h>
```

Collaboration diagram for s_message_cascade:



Data Fields

- t_messages * msgs
  messages
- struct s_message_cascade * next
  next messages

6.15.1 Detailed Description

a cascade of messages to the same target. is it still used somewhere?

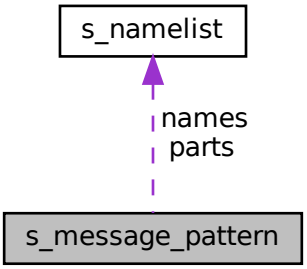
The documentation for this struct was generated from the following file:

- lib.h

6.16 s_message_pattern Struct Reference

`#include <lib.h>`

Collaboration diagram for s_message_pattern:



Data Fields

- t_namelist parts

  parts of the pattern
- t_namelist names

  names of the pattern

6.16.1 Detailed Description

message pattern as it is parsed.

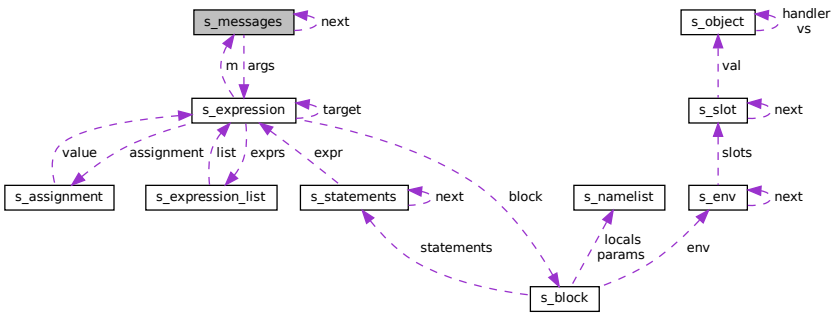
The documentation for this struct was generated from the following file:

- lib.h

6.17 s_messages Struct Reference

`#include <lib.h>`

Collaboration diagram for s_messages:



Data Fields

- bool cascaded

  either it's cascaded or nested nested means the result of the previous method invokation is used as the target for the next invokation cascaded means that all methods are invoked on the same first target.
- char * sel

  selector
- int argc

  argument count
- t_expression ** args

  expressions for the arguments
- struct s_messages * next

  next message, if any

6.17.1 Detailed Description

message calling structure

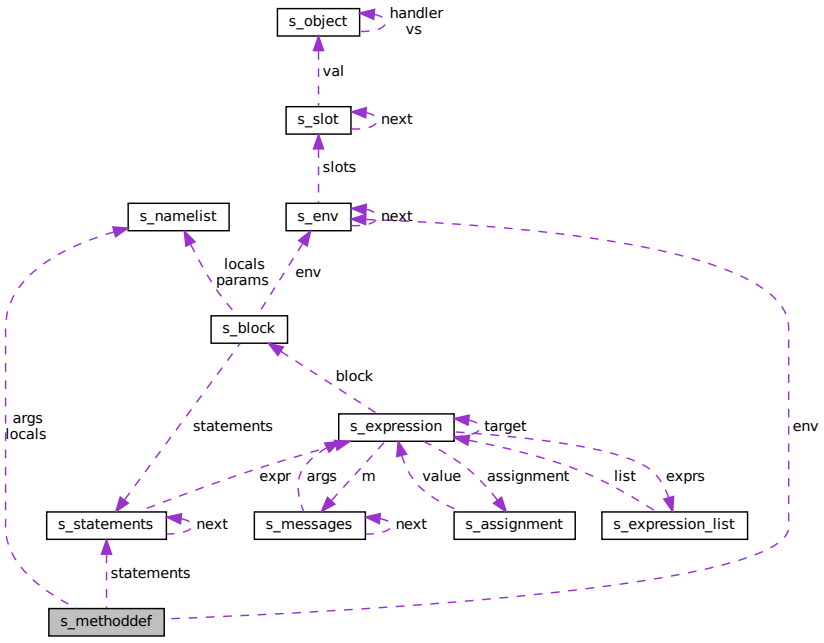
The documentation for this struct was generated from the following file:

- lib.h

6.18 s_methoddef Struct Reference

`#include <lib.h>`

Collaboration diagram for s_methoddef:



Data Fields

- char * sel
  selector
- t_namelist args
  arguments
- t_namelist locals
  locals
- t_statements * statements
  statements
- struct s_env * env
  unused

6.18.1 Detailed Description

method definition

The documentation for this struct was generated from the following file:

- lib.h

6.19 s_namelist Struct Reference

`#include <lib.h>`

Data Fields

- int count
  number of names in the list
- t_name * names
  array of names

6.19.1 Detailed Description

structure containing the counter and array of names.

The documentation for this struct was generated from the following file:

- lib.h

6.20 s_names Struct Reference

`#include <lib.h>`

Collaboration diagram for s_names:

Data Fields

- char * name
  name
- t_names next
  next

6.20.1 Detailed Description

a linked list of names

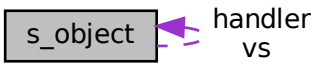
The documentation for this struct was generated from the following file:

- lib.h

6.21 s_object Struct Reference

```
#include <lib.h>
```

Collaboration diagram for s_object:



Data Fields

- t_message_handler handler
  handler for the messages
- union {
  void * data
  opacue data pointer
  int intval
  integer value
  struct {
  int i [10]
  list of integer values
  void * p [10]
  list of pointer values
  } vals
  values
  struct {
  struct s_object ** vs

list of object values
int cnt
count of objects in the list
} vars
general vars
} u

union

6.21.1 Detailed Description

an object contains mainly of the handler function and some data area. broken down into some specific alternatives for convenience. (but are not so convenient)

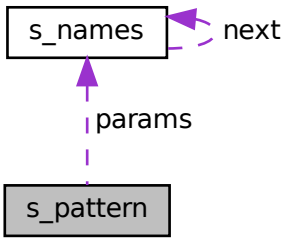
The documentation for this struct was generated from the following file:

- lib.h

6.22 s_pattern Struct Reference

```
#include <lib.h>
```

Collaboration diagram for s_pattern:



Data Fields

- char * selector
  selector
- t_names params
  parameter names

6.22.1 Detailed Description

method pattern with the selector parts concatenated and the list of names in order.

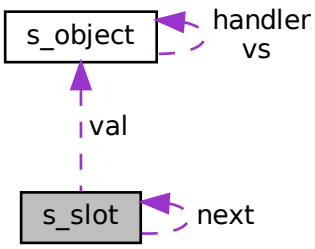
The documentation for this struct was generated from the following file:

- lib.h

6.23 s_slot Struct Reference

`#include <lib.h>`

Collaboration diagram for s_slot:



Data Fields

- const char * name
  name
- t_object * val
  value
- struct s_slot * next
  next slot

6.23.1 Detailed Description

one slot of the environment. A linked list of name and value pairs. Values are expressed as objects. Names have to be cstrings.

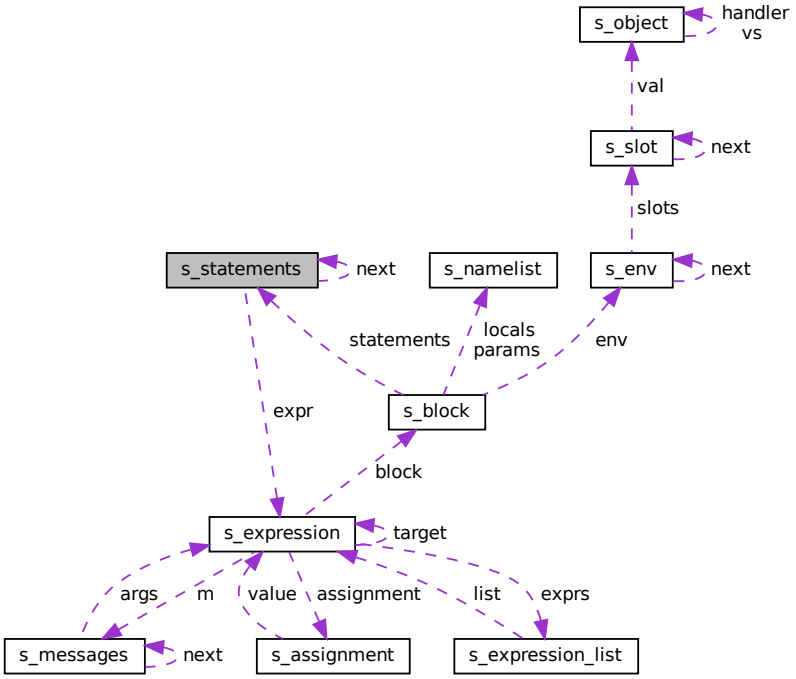
The documentation for this struct was generated from the following file:

- lib.h

6.24 s_statements Struct Reference

`#include <lib.h>`

Collaboration diagram for s_statements:



Data Fields

- t_statement_type type
  type
- struct s_expression * expr
  expression
- struct s_statements * next
  next statement

6.24.1 Detailed Description

linked list of statements. The type, expression and the next statments.

The documentation for this struct was generated from the following file:

- lib.h

6.25 stringinfo Struct Reference

Data Fields

- int num
  number of the string

6.25.1 Detailed Description

details of a string

The documentation for this struct was generated from the following file:

- lib.c

6.26 varinfo Struct Reference

Data Fields

- char * classname
  name of the class
- char * name
  name of the variable

6.26.1 Detailed Description

details of a global variable

The documentation for this struct was generated from the following file:

- lib.c