TinyTalk

Generated by Doxygen 1.9.4

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

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Module Documentation

6.1 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)

6.1.1 Detailed Description

6.1.2 Function Documentation

6.1.2.1 env_add()

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

adding a name defintion the an environment

```
59                                              {
60      tt_assert( env != NULL );
61      t_slot *n = env_get( env, name );
62      if( !n ) {
63          n = talloc_zero( env, t_slot );
64          n->next = env->slots;
65          n->name = name;
66          n->val = NULL;
67          env->slots = n;
68      }
69  }
```

References env_get(), s_slot::name, s_slot::next, s_env::slots, and s_slot::val.

Referenced by class_enter(), and method_enter().

6.1.2.2 env_clear()

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

clearing the values from the environment

6.1.2.3 env_dump()

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

output of the environment to stderr

```
15                                                  {
16     bool empty = true;
17     fprintf( stderr, "[[[ ENV %s\n", reason );
18     for( t_env * e = env; e; e = e->next ) {
19         empty = false;
20         fprintf( stderr, "---- Frame [%p] ----\n", e );
21         for( t_slot * s = e->slots; s; s = s->next ) {
22             fprintf( stderr, "ENV: %s:  %p\n", s->name, s->val );
23         }
24     }
25     if( empty )
26         fprintf( stderr, "---- Empty Frames ----\n" );
27     fprintf( stderr, "ENV %s ]]]\n", reason );
28 }
```

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

Referenced by env_set(), and env_set_local().

6.1.2.4 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

```
38                                                  {
39     for( t_slot * n = env->slots; n; n = n->next ) {
40         if( strcmp( name, n->name ) == 0 )
41             return n;
42     }
43     return NULL;
44 }
```

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

Referenced by env_add(), env_get_all(), and env_set_local().

6.1.2.5 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	refers to the top level environment
env_found	if not null receives the reference to the environment where the name has been found.

```
46                                                                          {
47      for( t_env * e = env; e; e = e->next ) {
48          t_slot *s = env_get( e, name );
49          if( s ) {
50              if( env_found )
51                  *env_found = e;
52              return s;
53          }
54      }
55      // env_dump( env, "GET_ALL" );
56      abort(  );
57  }
```

References env_get(), and s_env::next.

Referenced by env_set().

6.1.2.6 env_new()

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

new environment, inheriting definitions from another environment

```
31                                  {
32      t_env *env = talloc_zero( parent, t_env );
33      env->next = parent;
34      return env;
35  }
```

References s_env::next.

Referenced by class_enter(), and method_enter().

6.1.2.7 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

```
88                                                                      {
89      t_env *env_found = NULL;
90      t_slot *n = env_get_all( env, name, &env_found );
91      if( n ) {
92          n->val = val;
93          talloc_reference( env_found, val );
94      }
95      else {
96          env_dump( env, "ERROR" );
97          abort(  );
98      }
99  }
```

References env_dump(), env_get_all(), and s_slot::val.

6.1.2.8 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. ∗

```
71                                                                  {
72     t_slot *n = env_get( env, name );
73     if( n ) {
74         tt_assert( val );
75         if( n->val && n->val != val ) {
76             talloc_unlink( env, n->val );
77             n->val = val;
78         }
79         n->val = val;
80         talloc_reference( env, n->val );
81     }
82     else {
83         env_dump( env, "ERROR" );
84         abort(  );
85     }
86 }
```

References env_dump(), env_get(), and s_slot::val.

Referenced by block_handler(), and method_exec().

6.2 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)
- const char * string_cstr (t_object *self)
- t_object * integer_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 * transcript_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

6.2.1 Detailed Description

6.2.2 Macro Definition Documentation

6.2.2.1 MSG_DUMP

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

message selector for dumping an object

6.2.3 Function Documentation

6.2.3.1 block_handler()

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

handle block messages

```
194                                                                         {
195     t_object *result = self;
196     t_block *b = self->u.data;
197     t_env *env = b->env;
198     msg_add( "block handler" );
199     if( cstr_equals( "dump", sel ) ) {
200         msg_add( "dumping block.." );
201     }
202     else if( 0 == strcmp( "value", sel ) ) {
203 
204         result = simulate( env, b->statements );
205     }
206     else if( 0 == strcmp( "value:", sel ) ) {
207         assert( args );
208         assert( args[0] );
209         assert( args[0]->handler );
210         for( int i = 0; i < b->params.count; i++ ) {
211             tt_assert( env != NULL );
212             env_set_local( env, b->params.names[i], args[i] );
213         }
214         object_send_void( args[0], MSG_DUMP, NULL );
215         result = simulate( env, b->statements );
216     }
217     else if( 0 == strcmp( "whileFalse:", sel ) ) {
218         for( ;; ) {
219             t_object *r = simulate( env, b->statements );
220             object_send_void( r, "ifFalse:", args );
221             if( r->handler == true_handler )
222                 break;
223         }
224     }
225     else
226         result = method_exec( self, "Block", sel, args );
227     return result;
228 }
```

References s_namelist::count, cstr_equals(), s_object::data, 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(), s_block::statements, and s_object::u.

6.2.3.2 char_handler()

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

handle character messages

```
78                                                                          {
79     t_object *result = self;
80     if( 0 == strcmp( "==", sel ) ) {
81         if( args[0]->handler == char_handler ) {
82             if( self->u.intval == args[0]->u.intval )
83                 result = global.True;
84             else
85                 result = global.False;
86         }
87         else
88             result = global.False;
89     }
90     else if( cstr_equals( "dump", sel ) ) {
91         msg_add( "char:  %c", self->u.intval );
92     }
```

```
93     else
94         result = method_exec( self, "Char", sel, args );
95     return result;
96 }
```

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().

6.2.3.3 class_enter()

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

beginning a new class

```
4                                    {
5     require_classes( );
6     t_classdef *odef = itab_read( classes, name );
7
8
9     if( odef == NULL) {
10         printf( "new class %s\n", name );
11         odef = (t_classdef *)_talloc_zero(classes, sizeof(t_classdef), "t_classdef");
12         odef->id = gd.classnum++;
13         odef->name = talloc_strdup( odef, name );
14         odef->env = env_new(gd.env);
15         env_add(odef->env, "CLASSVAR");
16         itab_append( classes, name, odef );
17     }
18     current_class = odef;
19 }
```

6.2.3.4 cstr_equals()

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

compare to cstrings and return if they are equal

```
37                                                  {
38     return ( 0 == strcmp( a, b ) );
39 }
```

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

6.2.3.5 eval()

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

evaluate an expression

```
264                                                 {
265     t_object *result = NULL;
266     assert( expr );
267 
268     switch ( expr->tag ) {
269         case tag_string:
270             msg_add( "eval str %s", expr->u.strvalue );
271             result = object_new( string_handler );
272             result->u.data = talloc_strdup( result, expr->u.strvalue );
273             break;
274 
275         case tag_number:
276             result = object_new( int_handler );
277             result->u.intval = expr->u.intvalue;
278             msg_add( "eval number %d", result->u.intval );
279             break;
280 
281         case tag_char:
282             msg_add( "eval char" );
283             result = object_new( char_handler );
284             result->u.intval = expr->u.intvalue;
285             break;
286 
287         case tag_message:
288             result = eval_messages( env, expr );
289             break;
290 
291         case tag_block:
292             msg_add( "eval block" );
293             result = object_new( block_handler );
294             expr->u.block.env = env_new( env );
295             result->u.data = &expr->u.block;
296             for( int i = 0; i < expr->u.block.params.count; i++ ) {
297                 env_add( expr->u.block.env, expr->u.block.params.names[i] );
298             }
299             for( int i = 0; i < expr->u.block.locals.count; i++ ) {
300                 env_add( expr->u.block.env, expr->u.block.locals.names[i] );
301             }
302             break;
303 
304         case tag_ident:
305             result = NULL;
306             t_slot *slot = env_get_all( env, expr->u.ident, NULL );
307             result = slot->val;
308             if( result )
309                 msg_add( "eval ident %s -> %p(%p)", expr->u.ident, result,
310                         result->handler );
311             else {
312                 // env_dump( env, "IDENT no RESULT" );
313                 tt_assert( result );
314             }
315             break;
316 
317         case tag_assignment:
318             msg_add( "eval assignment" );
319             result = eval( env, expr->u.assignment.value );
320             msg_add( "assign result[%p] to %s", result, expr->u.assignment.target );
321             tt_assert( env );
322             env_set( env, ( t_name ) expr->u.assignment.target, result );
323             break;
324 
325         case tag_array:
326             {
327                 int n = expr->u.exprs.count;
328                 msg_add( "eval array %d elements", n );
329                 result = object_new( array_handler );
330                 result->u.vars.cnt = n;
331                 result->u.vars.vs = talloc_array( result, t_object *, n );
332                 for( int i = 0; i < expr->u.exprs.count; i++ ) {
333                     result->u.vars.vs[i] = eval( env, expr->u.exprs.list[i] );
334                 }
335             }
336             break;
337 
338         default:
339             msg_add( "error:  unknown eval tag:  %d", expr->tag );
340             // msg_print_last(  );
```

```
341             abort(  );
342             break;
343     }
344     return result;
345 }
```

References s_expression::tag.

6.2.3.6 int_handler()

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

handle integer messges

```
20                                                                          {
21     t_object *result = self;
22     assert( sel );
23     msg_add( "int handler:  (%d) %s", self->u.intval, sel );
24     if( 0 == strcmp( sel, "to:do:" ) ) {
25         assert( args[0]->handler == int_handler );
26         int start = self->u.intval;
27         int finish = args[0]->u.intval;
28 
29         for( int i = start; i <= finish; i++ ) {
30             t_object *par[1];
31             par[0] = object_new( int_handler );
32             par[0]->u.intval = i;
33             object_send_void( args[1], "value:", par );
34         }
35     }
36     else if( 0 == strcmp( sel, MSG_DUMP ) ) {
37         msg_add( "int:  %d", self->u.intval );
38     }
39     else if( cstr_equals( "asString", sel ) ) {
40         result = object_new( string_handler );
41         result->u.data = talloc_asprintf( result, "%d", self->u.intval );
42     }
43     else if( cstr_equals("*", sel)){
44         tt_assert(args[0]->handler == int_handler);
45         result = object_new( int_handler );
46         result->u.intval = self->u.intval * args[0]->u.intval;
47     }
48     else
49         result = method_exec( self, "Integer", sel, args );
50     return result;
51 }
```

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(), integer_meta_handler(), and string_handler().

6.2.3.7 integer_meta_handler()

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

handle integer meta messages

```
4                                                   {
5     t_object *result = self;
```

```
6     if( cstr_equals( "readFrom:ifFail:", sel ) ) {
7         long num = strtol( string_cstr( args[0] ), NULL, 10 );
8         if( errno )
9             result = object_send( args[1], "value", NULL );
10         else {
11             result = object_new( int_handler );
12             result->u.intval = num;
13         }
14     }
15     else
16         result = method_exec( self, "IntegerMeta", sel, args );
17     return result;
18 }
```

References cstr_equals(), int_handler(), s_object::intval, method_exec(), object_new(), object_send(), string_cstr(), and s_object::u.

6.2.3.8 is_binary_char()

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

binary chars are special ones for binary message names

```
228                                  {
229     switch ( c ) {
230         case '!':
231         case '%':
232         case '&':
233         case '*':
234         case '+':
235         case ',':
236         case '/':
237         case '<':
238         case '=':
239         case '>':
240         case '?':
241         case '@':
242         case '\\':
243         case '~':
244         case '|':                                                    // sollte laut Vorschlag
      ein Binary Operator sein.  Kollidiert aber mit der temporary declaration.
245 // das muss dann wohl auf der Syntaxebene geklärt werden.
246         case '-':
247             return true;
248         default:
249             return false;
250     }
251 }
```

6.2.3.9 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.

```
223                             {
224     return isalpha( c ) || isdigit( c ) || c == '_';
225 }
```

6.2.3.10 method_enter()

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

enter a method

set this method to be the current one. all further fiddeling with this current method goes into this one then. this state is usefull to load complexity off from the parser

```
11                                          {
12     require_current_class(  );
13 //  make selector from parts by concatenating them
14     char *sel = talloc_strdup( NULL, mp->parts.names[0] );
15     for( int i = 1; i < mp->parts.count; i++ ) {
16         sel = talloc_strdup_append( sel, mp->parts.names[i] );
17     }
18 //  create a consistent method name from class name and selector
19     char *nm = method_name( current_class->name, sel );
20 //  lookup this method name in our table
21     t_methoddef *odef = itab_read( methods, nm );
22     if( odef == NULL ) {
23 //  create a new method entry
24         odef = talloc_zero( methods, t_methoddef );
25         odef->sel = talloc_strdup( odef, sel );
26 //      prepare an environment TODO: how is this environment handled for multiple entries?
27         odef->env = env_new( current_class->env );
28         env_add( odef->env, "self" );
29         namelist_copy( &odef->args, &mp->names );
30         for( int i = 0; i < odef->args.count; i++ ) {
31             env_add( odef->env, odef->args.names[i] );
32         }
33
34         talloc_steal( odef, odef->args.names );
35         assert( talloc_get_type( odef->args.names, t_name ) );
36         itab_append( methods, nm, odef );
37     }
38
39 /*  else we should think of some reaction
40 *  option a) replace the definition
41 *  option b) throw an error
42 */
43     current_method = odef;
44     talloc_steal( odef, mp );
45     talloc_free( nm );
46     talloc_free( sel );
47 }
```

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.

6.2.3.11 method_exec()

```
t_object * method_exec (
            t_object * self,
            const char * clsname,
```

```
            const char * sel,
            t_object ** args )
```

execute a method

```
170                                         {
171     t_object *result = self;
172     t_methoddef *m = method_read( clsname, sel );
173     if( m ) {
174         t_env *env = m->env;
175         tt_assert( env );
176         msg_add( "selector %s is defined on %s (env:%p, self:%p, handler:%p)",
177                 sel, clsname, env, self, self->handler );
178         for( int i = 0; i < m->args.count; i++ ) {
179             env_set_local( env, m->args.names[i], args[i] );
180         }
181         env_set_local( env, "self", self );
182         // env_dump( env, "after self" );
183         result = simulate( env, m->statements );
184         msg_add( "done simulation of method %s", sel );
185     }
186     else {
187         msg_add( "%s %s not found.", clsname, sel );
188         // msg_print_last(  );
189         abort(  );
190     }
191     return result;
192 }
```

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

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

6.2.3.12 method_name()

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

construct the method name from combination of class and selector

```
2                                                       {
3     char *result = talloc_strdup( NULL , class );
4     result = talloc_strdup_append( result, "/" );
5     result = talloc_strdup_append( result, sel );
6     return result;
7 }
```

Referenced by method_enter().

6.2.3.13 nextToken()

```
bool nextToken (
            void  )
```

read next token.

parse the next token

This is a more detailed description.

Returns

true if successful

```
378                 {
379     char c;
380     bool result = false;
381     while( true ) {
382         while( readChar( &c ) && isspace( c ) );
383         if( c == '"' ) {
384             while( readChar( &c ) && c != '"' );
385         }
386         else
387             break;
388     }
389     if( gd.state == 1 ) {
390         if( isalpha( c ) ) {
391             int idx = 0;
392             for( ;; ) {
393                 gd.buf[idx++] = c;
394                 readChar( &c );
395                 if( !is_ident_char( c ) )
396                     break;
397             }
398             if( c == ':'  )  {
399                 gd.buf[idx++] = c;
400                 gd.token = TK_KEYWORD;
401             }
402             else {
403                 gd.pos--;
404                 gd.token = TK_IDENT;
405             }
406             gd.buf[idx] = 0;
407             result = true;
408         }
409         else if( is_binary_char( c ) ) {
410             for( int idx = 0; is_binary_char( c ); idx++ ) {
411                 gd.buf[idx] = c;
412                 gd.buf[idx + 1] = 0;
413                 readChar( &c );
414             }
415             gd.pos--;
416             gd.token = 0;
417             gd.token = TK_BINOP;
418             result = true;
419             if( strcmp( ":=", gd.buf ) == 0 ) {
420                 gd.token = TK_ASSIGN;
421                 result = true;
422             }
423             else if( strcmp( "<-", gd.buf ) == 0 ) {
424                 gd.token = TK_LARROW;
425                 result = true;
426             }
427             else if( strcmp( "|", gd.buf ) == 0 ) {
428                 gd.token = TK_BAR;
429                 result = true;
430             }
431             else if( 0 == strcmp( "<", gd.buf ) ) {
432                 gd.token = TK_LT;
433                 result = true;
434             }
435             else if( 0 == strcmp( ">", gd.buf ) ) {
436                 gd.token = TK_GT;
437                 result = true;
438             }
439         }
440         else if( isdigit( c ) ) {
441             int idx = 0;
442             while( isdigit( c ) ) {
443                 printf( "### digit %c\n", c );
444                 gd.buf[idx++] = c;
445                 gd.buf[idx] = 0;
446                 readChar( &c );
447             }
448             gd.pos--;
449             gd.token = TK_NUMBER;
450             result = true;
451         }
452         else {
453             switch ( c ) {
454                 case '\":
455                     result = readStringToken(  );
456                     break;
457                 case '.':
458                     result = true;
459                     gd.token = TK_DOT;
460                     break;
461                 case ';':
```

```
462                     result = true;
463                     gd.token = TK_SEMICOLON;
464                     break;
465                 case '(':
466                     result = true;
467                     gd.token = TK_LPAREN;
468                     break;
469                 case ')':
470                     result = true;
471                     gd.token = TK_RPAREN;
472                     break;
473                 case '[':
474                     result = true;
475                     gd.token = TK_LBRACK;
476                     break;
477                 case ']':
478                     result = true;
479                     gd.token = TK_RBRACK;
480                     break;
481                 case '{':
482                     result = true;
483                     gd.token = TK_LBRACE;
484                     break;
485                 case '}':
486                     result = true;
487                     gd.token = TK_RBRACE;
488                     break;
489                 case '#':
490                     readChar( &c );
491                     for( int idx = 0; is_ident_char( c ) || c == ':'; idx++ ) {
492                         gd.buf[idx] = c;
493                         gd.buf[idx + 1] = 0;
494                         readChar( &c );
495                     }
496                     gd.pos--;
497                     gd.token = TK_SYMBOL;
498                     result = true;
499                     break;
500                 case '^':
501                     result = true;
502                     gd.token = TK_UARROW;
503                     break;
504                 case ':':
505                     result = true;
506                     gd.token = TK_COLON;
507                     readChar( &c );
508                     if( c == '=' ) {
509                         gd.token = TK_ASSIGN;
510                     }
511                     else
512                         gd.pos--;
513                     break;
514                 case '$':
515                     result = true;
516                     gd.token = TK_CHAR;
517                     readChar( &c );
518                     gd.buf[0] = c;
519                     gd.buf[1] = 0;
520                     break;
521                 default:
522                     gd.pos--;
523                     break;
524             }
525         }
526     }
527     return result;
528 }
```

6.2.3.14 object_new()

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

new object

```
14                                              {
15     t_object *result = talloc_zero( NULL, t_object );
16     result->handler = hdl;
17     return result;
```

```
18 }
```

References s_object::handler.

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

6.2.3.15 object_send()

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

send a message to an object

```
19                                                                          {
20     return o->handler( o, sel, args );
21 }
```

Referenced by integer_meta_handler(), object_send_void(), and string_meta_handler().

6.2.3.16 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

```
22                                                                          {
23     t_object *x = object_send( o, sel, args );
24     if( x != o ) {
25 // fprintf( stderr, "send void %s leaves these parents:\n", sel );
26 // talloc_show_parents( x, stderr );
27     }
28     talloc_unlink( NULL, x );
29 }
```

References object_send().

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

6.2.3.17 parse_verbatim()

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

parse the verbatim chars (not used)

```
363                                {
364     int i = 0;
365     gd.buf[i] = 0;
366     readChar( &c );
367     while( c != '}' ) {
368         gd.buf[i++] = c;
369         gd.buf[i] = 0;
370         readChar( &c );
371     }
372     gd.token = TK_VERBATIM;
373 }
```

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

6.2.3.18 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

```
331                         {
332     bool result = true;
333     if( gd.state == 0 ) {
334         result = readLine(  );
335     }
336     if( result ) {
337         *t = gd.line[gd.pos++];
338         while( *t == 0 ) {
339             if( readLine(  ) ) {
340                 *t = gd.line[gd.pos++];
341             }
342             else {
343                 result = false;
344                 break;
345             }
346         }
347     }
348     return result;
349 }
```

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

6.2.3.19 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.

```
308                 {
309     if( gd.src_iter == NULL ) {
310         gd.src_iter = itab_foreach( gd.src );
311     }
312     else {
313         gd.src_iter = itab_next( gd.src_iter );
314     }
315     if( gd.src_iter ) {
316         gd.line = itab_value( gd.src_iter );
317         gd.line_count++;
318         printf( "%2d:%s\n", gd.line_count, gd.line );
319         gd.pos = 0;
320         gd.state = 1;
321         return true;
322     }
323     else {
```

```
324         gd.line = "";
325         gd.state = 2;
326         return false;
327     }
328 }
```

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

6.2.3.20 readStringToken()

```
bool readStringToken (
             void  )
```

read string token.

read a string token from the source

Returns

true if successful

```
351                          {
352     int idx = 0;
353     char c;
354     while( readChar( &c ) && '\"' != c ) {
355         if( c == '\\' )
356             readChar( &c );
357         gd.buf[idx++] = c;
358     }
359     gd.buf[idx] = 0;
360     gd.token = TK_STRING;
361     return true;
362 }
```

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

6.2.3.21 require_classes()

```
void require_classes (
             void  )
```

require definition of classes

```
9                           {
10     if( !classes ) {
11         classes = itab_new( );
12         methods = itab_new( );
13         method_names = itab_new( );
14         variables = itab_new( );
15         strings = itab_new( );
16
17         gd.classnum = 1;
18
19         class_enter( "Behavior" );
20         class_enter( "Object" );
21
22         string_class_num = gd.classnum - 1;
23
24
25         gd.classnum = 100;
26     }
27 }
```

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

Referenced by class_enter().

6.2.3.22 require_current_class()

```
void require_current_class (
             void  )
```

require that there is a current class

```
3                                {
4     assert( current_class != NULL );
5 }
```

Referenced by method_enter().

6.2.3.23 simulate()

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

simulate the execution of a list of statemtents

```
347                                                          {
348     t_object *result = NULL;
349     assert( stmts );
350
351     msg_add( "simulate" );
352
353     while( stmts ) {
354         switch ( stmts->type ) {
355             case stmt_message:
356                 msg_add( "message stmt" );
357                 result = eval( env, stmts->expr );
358                 break;
359             case stmt_return:
360                 msg_add( "return stmt" );
361                 result = eval( env, stmts->expr );
362                 msg_add( "returning value and leaving method...\n" );
363                 //msg_print_last(  );
364                 stmts = NULL;
365                 break;
366
367             default:
368                 msg_add( "error:  unkonwn stmt type:  %d\n", stmts->type );
369                 // msg_print_last(  );
370                 abort(  );
371                 break;
372         }
373         if( stmts )
374             stmts = stmts->next;
375     }
376     return result;
377 }
```

References msg_add(), and s_statements::type.

Referenced by block_handler(), and method_exec().

6.2.3.24 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.

```
264                                  {
265     int n = itab_lines( gd.src );
266     char buf[10];
267     sprintf( buf, "%09d", n + 1 );
268     itab_append( gd.src, buf, talloc_strdup( gd.src, line ) );
269 }
```

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

6.2.3.25 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.

```
253                     {
254     if( gd.src ) {
255         talloc_free( gd.src );
256     }
257     gd.src = itab_new(  );
258     if( gd.src_iter ) {
259         talloc_free( gd.src_iter );
260     }
261     gd.src_iter = NULL;
262 }
```

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

6.2.3.26 src_dump()

```
bool src_dump (
            void  )
```

dump the source table

dumps all the lines of the current source.

```
295                     {
296     for( struct itab_iter * x = itab_foreach( gd.src );
297          x; x = itab_next( x ) ) {
298         printf( "%s:%s\n", itab_key( x ), itab_value( x ) );
299     }
300 }
```

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

6.2.3.27 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.

```
274                                 {
275     FILE *f = fopen( name, "r" );
276     char buf[1000];
277     char *line;
278     int line_no = 1;
279     src_clear(  );
```

```
280     for( ;; ) {
281         line = fgets( buf, sizeof( buf ), f );
282         if( line == NULL )
283             break;
284         int n = strlen( line );
285         while( n > 0 && isspace( line[--n] ) )
286             line[n] = 0;
287         char line_number[10];
288         sprintf( line_number, "%09d", line_no );
289         itab_append( gd.src, line_number, talloc_strdup( gd.src, line ) );
290         line_no++;
291     }
292     fclose( f );
293 }
```

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

6.2.3.28 stream_handler()

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

handle stream messages

```
98                                                                          {
99     t_object *result = self;
100     if( cstr_equals( "upTo:", sel ) ) {
101         tt_assert( args[0]->handler = char_handler );
102         char sep = args[0]->u.intval;
103         int idx = self->u.vals.i[0];
104         int start = idx;
105         int max = self->u.vals.i[1];
106         char *chars = ( char * )self->u.vals.p[0];
107         while( idx < max ) {
108             if( chars[idx] == sep )
109                 break;
110             idx++;
111         }
112         result = object_new( string_handler );
113         int len = idx - start;
114         result->u.data = talloc_zero_array( result, char, len + 1 );
115         memcpy( result->u.data, chars + start, len );
116         self->u.vals.i[0] = idx + 1;
117     }
118     else if( 0 == strcmp( "atEnd", sel ) ) {
119         if( self->u.vals.i[0] < self->u.vals.i[1] )
120             result = global.False;
121         else
122             result = global.True;
123     }
124     else if( 0 == strcmp( "next", sel ) ) {
125         result = object_new( char_handler );
126         result->u.intval = ( ( char * )self->u.vals.p[0] )[self->u.vals.i[0]];
127         self->u.vals.i[0]++;
128     }
129     else if( cstr_equals( "nextPut:", sel ) ) {
130         tt_assert( args[0]->handler == char_handler );
131         int idx = self->u.vals.i[0];
132         int max = self->u.vals.i[1];
133         char *chars = ( char * )self->u.vals.p[0];
134         char c = args[0]->u.intval;
135         tt_assert(chars);
136         if( max <= idx ) {
137             max *= 2;
138             chars = talloc_realloc( self, chars, char, max + 1 );
139             tt_assert(chars);
140             self->u.vals.i[1] = max;
141             self->u.vals.p[0] = chars;
142         }
143         chars[idx] = args[0]->u.intval;
144         self->u.vals.i[0] = idx + 1;
145     }
146     else if( cstr_equals( "dump", sel ) ) {
147         msg_add( "Stream len:%d pos:%d", self->u.vals.i[1],
148                  self->u.vals.i[0] );
```

```
149     }
150     else
151         result = method_exec( self, "Stream", sel, args );
152     return result;
153 }
```

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

Referenced by string_handler(), and string_meta_handler().

6.2.3.29 string_cstr()

```
const char * string_cstr (
             t_object * self )
```

cstring

returns the CString reference of a String object Errors: Object is no String object

Parameters

self	the string object

```
8                                           {
9     assert( self->handler == string_handler );
10     return self->u.data;
11 }
```

References s_object::data, s_object::handler, string_handler(), and s_object::u.

Referenced by integer_meta_handler(), string_handler(), and transcript_handler().

6.2.3.30 string_handler()

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

handle string messages

```
41                                                                          {
42     t_object *result = self;
43     const char *self_data = string_cstr( self );
44     if( 0 == strcmp( sel, MSG_DUMP ) ) {
45         msg_add( "str:  '%s'", self_data );
46     }
47     else if( cstr_equals( "asString", sel ) ) {
48 // all set...
49     }
50     else if( cstr_equals( "do:", sel ) ) {
51         int n = strlen( self_data );
52         for( int i = 0; i < n; i++ ) {
53             t_object *r = object_new( char_handler );
54             r->u.intval = self_data[i];
55             object_send_void( args[0], "value:", &r );
56         }
```

```
57      }
58      else if( 0 == strcmp( sel, "readStream" ) ) {
59          t_object *result = object_new( stream_handler );
60          result->u.vals.i[0] = 0;
61          result->u.vals.i[1] = strlen( self->u.data );
62          result->u.vals.p[0] = self->u.data;
63          msg_add( "stream for readStream:  %p", result );
64          return result;
65      }
66      else if( 0 == strcmp( sel, "species" ) ) {
67          return global.String;
68      }
69      else if( 0 == strcmp( sel, "size" ) ) {
70          result = object_new( int_handler );
71          result->u.intval = strlen( self->u.data );
72      }
73      else {
74          result = method_exec( self, "String", sel, args );
75      }
76      return result;
77  }
```

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

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

6.2.3.31 string_meta_handler()

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

handle string meta messages

```
15                                                  {
16      t_object *result = self;
17      if( 0 == strcmp( "new:streamContents:", sel ) ) {
18          int size = args[0]->u.intval;
19          msg_add( "new string with size:  %d", size );
20          t_object *par[1];
21          t_object *stream = object_new( stream_handler );
22
23          stream = object_new( stream_handler );
24          stream->u.vals.i[0] = 0;
25          stream->u.vals.i[1] = size;
26          stream->u.vals.p[0] = talloc_array( stream, char, size + 1 );
27          talloc_reference( self, stream );
28          par[0] = stream;
29          msg_add( "stream for streamContent:  %p", stream );
30          t_object *o = object_send( args[1], "value:", par );
31  // talloc_unlink(NULL, par[0] );
32          result = object_new( string_handler );
33          result->u.data = talloc_strdup( result, stream->u.vals.p[0] );
34      }
35      else
36          result = method_exec( self, "StringMeta", sel, args );
37
38      return result;
39  }
```

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

6.2.3.32 transcript_handler()

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

handle transcript messages

```
4                                                   {
5     t_object *result = self;
6     if(cstr_equals("show:", sel)){
7         printf("--->|%s|\n", string_cstr(args[0]));
8     }
9     return result;
10 }
```

References cstr_equals(), and string_cstr().

6.2.4 Variable Documentation

6.2.4.1 global

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

global

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

6.3 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)

6.3.1 Detailed Description

sorted list of structures -> tables with primary index

6.3.2 Function Documentation

6.3.2.1 itab_append()

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

append new line

```
140                                                                     {
141     assert( itab != NULL );
142     if( itab->total == itab->used ) {
143         itab->total *= 2;
144         itab->rows =
145                 talloc_realloc( itab, itab->rows, struct itab_entry,
146                                 itab->total );
147     }
148     struct itab_entry *row = &itab->rows[itab->used];
149     row->key = talloc_strdup( itab, key );
150     row->value = value;
151     itab->used++;
152
153     qsort( itab->rows,                          // base
154            itab->used,                          // nmemb
155            sizeof( struct itab_entry ),         // size
156            itab_entry_cmp );
157 }
```

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().

6.3.2.2 itab_dump()

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

dump content to output

```
174                                 {
175     assert( itab );
176     for( int i = 0; i < itab->used; i++ ) {
177         fprintf( stderr, "%s:  %p\n", itab->rows[i].key, itab->rows[i].value );
178     }
179 }
```

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

6.3.2.3 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

```
134                                                                 {
135     const struct itab_entry *a = aptr;
136     const struct itab_entry *b = bptr;
137     return strcmp( a->key, b->key );
138 }
```

References itab_entry::key.

Referenced by itab_append(), and itab_read().

6.3.2.4 itab_foreach()

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

start iteration

```
181                                                         {
182     if( tab->used > 0 ) {
183         struct itab_iter *r = talloc_zero( NULL, struct itab_iter );
184         r->tab = tab;
185         r->pos = 0;
186         return r;
187     }
188     else
189         return NULL;
190 }
```

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

Referenced by readLine(), and src_dump().

6.3.2.5 itab_key()

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

key of current iterator

```
207                                             {
208     return iter->tab->rows[iter->pos].key;
209 }
```

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

Referenced by src_dump().

6.3.2.6 itab_lines()

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

how many lines?

returns the number of lines in the table

```
100                                         {
101     assert( itab != NULL );
102     return itab->used;
103 }
```

References used.

Referenced by src_add().

6.3.2.7 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.

```
120                             {
121     struct itab *r = talloc_zero( NULL, struct itab );
122     r->total = 10;
123     r->used = 0;
124     r->rows = talloc_array( r, struct itab_entry, r->total );
125     return r;
126 }
```

References rows, total, and used.

Referenced by require_classes(), and src_clear().

6.3.2.8 itab_next()

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

cycle through iterator

```
192                                                     {
193     iter->pos++;
194     if( iter->tab->used > iter->pos ) {
195         return iter;
196     }
197     else {
198         talloc_free( iter );
199         return NULL;
200     }
201 }
```

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

Referenced by readLine(), and src_dump().

6.3.2.9 itab_read()

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

find a line by key

```
159                                                 {
160     assert( itab );
161     assert( key );
162     struct itab_entry dummy = { key, NULL };
163     struct itab_entry *r = bsearch( &dummy,
164                                     itab->rows,
165                                     itab->used,
166                                     sizeof( struct itab_entry ),
167                                     itab_entry_cmp );
168     if( r )
169         return r->value;
170     else
171         return NULL;
172 }
```

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

Referenced by class_enter(), and method_enter().

6.3.2.10 itab_value()

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

value of current iterator

```
203                                         {
204     return iter->tab->rows[iter->pos].value;
205 }
```

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

Referenced by readLine(), and src_dump().

6.4 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.

6.4.1 Detailed Description

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

See also

nextToken.

6.4.2 Function Documentation

6.4.2.1 is_binary_char()

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

binary chars are special ones for binary message names

```
228                                 {
229     switch ( c ) {
230         case '!':
231         case '%':
232         case '&':
233         case '*':
234         case '+':
235         case ',':
236         case '/':
237         case '<':
238         case '=':
239         case '>':
240         case '?':
241         case '@':
242         case '\\':
243         case '~':
244         case '|':                                                          // sollte laut Vorschlag
      ein Binary Operator sein.  Kollidiert aber mit der temporary declaration.
245 // das muss dann wohl auf der Syntaxebene geklärt werden.
246         case '-':
247             return true;
248         default:
249             return false;
250     }
251 }
```

Referenced by nextToken().

6.4.2.2 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.

```
223                             {
224     return isalpha( c ) || isdigit( c ) || c == '_';
225 }
```

Referenced by nextToken().

6.4.2.3 nextToken()

```
bool nextToken ( )
```

read next token.

This is a more detailed description.

Returns

true if successful

```
378                     {
379     char c;
380     bool result = false;
381     while( true ) {
382         while( readChar( &c ) && isspace( c ) );
383         if( c == '"' ) {
384             while( readChar( &c ) && c != '"' );
385         }
386         else
387             break;
388     }
389     if( gd.state == 1 ) {
390         if( isalpha( c ) ) {
391             int idx = 0;
392             for( ;; ) {
393                 gd.buf[idx++] = c;
394                 readChar( &c );
395                 if( !is_ident_char( c ) )
396                     break;
397             }
398             if( c == ':'  )  {
399                 gd.buf[idx++] = c;
400                 gd.token = TK_KEYWORD;
401             }
402             else {
403                 gd.pos--;
404                 gd.token = TK_IDENT;
405             }
406             gd.buf[idx] = 0;
407             result = true;
408         }
409         else if( is_binary_char( c ) ) {
410             for( int idx = 0; is_binary_char( c ); idx++ ) {
411                 gd.buf[idx] = c;
412                 gd.buf[idx + 1] = 0;
413                 readChar( &c );
414             }
415             gd.pos--;
416             gd.token = 0;
417             gd.token = TK_BINOP;
418             result = true;
419             if( strcmp( ":=", gd.buf ) == 0 ) {
420                 gd.token = TK_ASSIGN;
421                 result = true;
422             }
423             else if( strcmp( "<-", gd.buf ) == 0 ) {
424                 gd.token = TK_LARROW;
425                 result = true;
426             }
427             else if( strcmp( "|", gd.buf ) == 0 ) {
428                 gd.token = TK_BAR;
429                 result = true;
430             }
```

```
431             else if( 0 == strcmp( "<", gd.buf ) ) {
432                 gd.token = TK_LT;
433                 result = true;
434             }
435             else if( 0 == strcmp( ">", gd.buf ) ) {
436                 gd.token = TK_GT;
437                 result = true;
438             }
439         }
440         else if( isdigit( c ) ) {
441             int idx = 0;
442             while( isdigit( c ) ) {
443                 printf( "### digit %c\n", c );
444                 gd.buf[idx++] = c;
445                 gd.buf[idx] = 0;
446                 readChar( &c );
447             }
448             gd.pos--;
449             gd.token = TK_NUMBER;
450             result = true;
451         }
452         else {
453             switch ( c ) {
454                 case '\":
455                     result = readStringToken(  );
456                     break;
457                 case '.':
458                     result = true;
459                     gd.token = TK_DOT;
460                     break;
461                 case ';':
462                     result = true;
463                     gd.token = TK_SEMICOLON;
464                     break;
465                 case '(':
466                     result = true;
467                     gd.token = TK_LPAREN;
468                     break;
469                 case ')':
470                     result = true;
471                     gd.token = TK_RPAREN;
472                     break;
473                 case '[':
474                     result = true;
475                     gd.token = TK_LBRACK;
476                     break;
477                 case ']':
478                     result = true;
479                     gd.token = TK_RBRACK;
480                     break;
481                 case '{':
482                     result = true;
483                     gd.token = TK_LBRACE;
484                     break;
485                 case '}':
486                     result = true;
487                     gd.token = TK_RBRACE;
488                     break;
489                 case '#':
490                     readChar( &c );
491                     for( int idx = 0; is_ident_char( c ) || c == ':'; idx++ ) {
492                         gd.buf[idx] = c;
493                         gd.buf[idx + 1] = 0;
494                         readChar( &c );
495                     }
496                     gd.pos--;
497                     gd.token = TK_SYMBOL;
498                     result = true;
499                     break;
500                 case '^':
501                     result = true;
502                     gd.token = TK_UARROW;
503                     break;
504                 case ':':
505                     result = true;
506                     gd.token = TK_COLON;
507                     readChar( &c );
508                     if( c == '=' ) {
509                         gd.token = TK_ASSIGN;
510                     }
511                     else
512                         gd.pos--;
513                     break;
514                 case '$':
515                     result = true;
516                     gd.token = TK_CHAR;
517                     readChar( &c );
```

```
518                     gd.buf[0] = c;
519                     gd.buf[1] = 0;
520                     break;
521                 default:
522                     gd.pos--;
523                     break;
524             }
525         }
526     }
527     return result;
528 }
```

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

6.4.2.4 parse_verbatim()

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

parse the verbatim chars (not used)

```
363                                 {
364     int i = 0;
365     gd.buf[i] = 0;
366     readChar( &c );
367     while( c != '}' ) {
368         gd.buf[i++] = c;
369         gd.buf[i] = 0;
370         readChar( &c );
371     }
372     gd.token = TK_VERBATIM;
373 }
```

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

6.4.2.5 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

```
331                         {
332     bool result = true;
333     if( gd.state == 0 ) {
334         result = readLine(  );
335     }
336     if( result ) {
337         *t = gd.line[gd.pos++];
338         while( *t == 0 ) {
```

```
339             if( readLine(  ) ) {
340                 *t = gd.line[gd.pos++];
341             }
342             else {
343                 result = false;
344                 break;
345             }
346         }
347     }
348     return result;
349 }
```

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

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

6.4.2.6 readLine()

```
bool readLine ( )
```

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

trailing blanks are removed.

```
308                 {
309     if( gd.src_iter == NULL ) {
310         gd.src_iter = itab_foreach( gd.src );
311     }
312     else {
313         gd.src_iter = itab_next( gd.src_iter );
314     }
315     if( gd.src_iter ) {
316         gd.line = itab_value( gd.src_iter );
317         gd.line_count++;
318         printf( "%2d:%s\n", gd.line_count, gd.line );
319         gd.pos = 0;
320         gd.state = 1;
321         return true;
322     }
323     else {
324         gd.line = "";
325         gd.state = 2;
326         return false;
327     }
328 }
```

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().

6.4.2.7 readStringToken()

```
bool readStringToken (
            void  )
```

read string token.

Returns

true if successful

```
351                         {
352     int idx = 0;
353     char c;
354     while( readChar( &c ) && '\"' != c ) {
355         if( c == '\\' )
356             readChar( &c );
357         gd.buf[idx++] = c;
358     }
359     gd.buf[idx] = 0;
360     gd.token = TK_STRING;
361     return true;
362 }
```

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

Referenced by nextToken().

6.4.2.8 src_add()

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

adding one line to the source that will be parsed.

```
264                                 {
265     int n = itab_lines( gd.src );
266     char buf[10];
267     sprintf( buf, "%09d", n + 1 );
268     itab_append( gd.src, buf, talloc_strdup( gd.src, line ) );
269 }
```

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

6.4.2.9 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.

```
253                 {
254     if( gd.src ) {
255         talloc_free( gd.src );
256     }
257     gd.src = itab_new(  );
258     if( gd.src_iter ) {
259         talloc_free( gd.src_iter );
260     }
261     gd.src_iter = NULL;
262 }
```

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

Referenced by src_read().

6.4.2.10 src_dump()

```
bool src_dump ( )
```

dumps all the lines of the current source.

```
295                     {
296     for( struct itab_iter * x = itab_foreach( gd.src );
297          x; x = itab_next( x ) ) {
298         printf( "%s:%s\n", itab_key( x ), itab_value( x ) );
299     }
300 }
```

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

6.4.2.11 src_read()

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

read file into itab.

read a file into src itab.

```
274                                     {
275     FILE *f = fopen( name, "r" );
276     char buf[1000];
277     char *line;
278     int line_no = 1;
279     src_clear(  );
280     for( ;; ) {
281         line = fgets( buf, sizeof( buf ), f );
282         if( line == NULL )
283             break;
284         int n = strlen( line );
285         while( n > 0 && isspace( line[--n] ) )
286             line[n] = 0;
287         char line_number[10];
288         sprintf( line_number, "%09d", line_no );
289         itab_append( gd.src, line_number, talloc_strdup( gd.src, line ) );
290         line_no++;
291     }
292     fclose( f );
293 }
```

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

6.5 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 ()

6.5.1 Detailed Description

6.5.2 Macro Definition Documentation

6.5.2.1 MSG_LOG_LEN

```
#define MSG_LOG_LEN 200
```

length of log

6.5.3 Typedef Documentation

6.5.3.1 t_msg

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

contains a log line

6.5.4 Function Documentation

6.5.4.1 msg_add()

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

adding a message

```
560                                     {
561     va_list ap;
562     msg_init(  );
563     va_start( ap, msg );
564
565     vsnprintf( msgs.msgs[msgs.pos], 199, msg, ap );
566     msgs.pos = ( msgs.pos + 1 ) % msgs.size;
567     va_end( ap );
568 }
```

References msg_init().

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

6.5.4.2 msg_init()

```
void msg_init ( )
```

initialize application messages

```
552                     {
553     if( msgs.size != MSG_LOG_LEN ) {
554         msgs.size = MSG_LOG_LEN;
555         msgs.pos = 0;
556     }
557 }
```

Referenced by msg_add().

6.5.4.3 msg_print_last()

```
void msg_print_last ( )
```

print the last messages from the log

```
571                         {
572     printf( "--------------------------------------\n" );
573     const char *fmt = "%03d --- %s\n";
574     int n = 1;
575     for( int i = msgs.pos; i < msgs.size; i++ ) {
576         if( msgs.msgs[i][0] )
577             printf( fmt, n++, msgs.msgs[i] );
578         msgs.msgs[i][0] = 0;
579     }
580     for( int i = 0; i < msgs.pos; i++ ) {
581         if( msgs.msgs[i][0] )
582             printf( fmt, n++, msgs.msgs[i] );
583         msgs.msgs[i][0] = 0;
584     }
585 }
```

6.6 Syntax Messages

Functions

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

6.6.1 Detailed Description

6.6.2 Function Documentation

6.6.2.1 message_add_msg()

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

add a message

```
596                                                     {
597     while( ms->next )
598         ms = ms->next;
599     ms->next = m;
600 }
```

References s_messages::next.

6.7 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)

6.7.1 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.

6.7.2 Typedef Documentation

6.7.2.1 t_name

```
typedef const char* t_name
```

name

6.7.2.2 t_namelist

```
typedef struct s_namelist t_namelist
```

structure containing the counter and array of names.

6.7.2.3 t_names

```
typedef struct s_names* t_names
```

array of names

6.7.3 Function Documentation

6.7.3.1 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

```
27                                                      {
28     nl->count++;
29     nl->names = talloc_realloc( NULL, nl->names, t_name, nl->count );
30     nl->names[nl->count - 1] = talloc_strdup( nl->names, name );
31 }
```

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

6.7.3.2 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.

```
36                                                      {
37     to->count = from->count;
38     to->names = talloc_array( NULL, t_name, to->count );
39     assert(talloc_get_type(from->names, t_name));
40     for( int i = 0; i < to->count; i++ ) {
41         to->names[i] = talloc_strdup( to->names, from->names[i] );
42     }
43     assert(talloc_get_type(to->names, t_name));
44 }
```

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

Referenced by method_enter().

6.7.3.3 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.

```
17                                      {
18     nl->count = 0;
19     nl->names = NULL;
20 }
```

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

6.8 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** }

6.8.1 Detailed Description

6.8.2 Typedef Documentation

6.8.2.1 t_assignment

```
typedef struct s_assignment t_assignment
```

an assignment consists of a target name and an expression

6.8.2.2 t_block

```
typedef struct s_block t_block
```

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

6.8.2.3 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.

6.8.2.4 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.

6.8.2.5 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

6.8.2.6 t_expression_list

```
typedef struct s_expression_list t_expression_list
```

a list of expressions

6.8.2.7 t_expression_tag

```
typedef enum e_expression_tag t_expression_tag
```

expression type

6.8.2.8 t_message_cascade

```
typedef struct s_message_cascade t_message_cascade
```

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

6.8.2.9 t_message_handler

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

message handler

6.8.2.10 t_message_pattern

```
typedef struct s_message_pattern t_message_pattern
```

message pattern as it is parsed.

6.8.2.11 t_messages

```
typedef struct s_messages t_messages
```

message calling structure

6.8.2.12 t_methoddef

```
typedef struct s_methoddef t_methoddef
```

method definition

6.8.2.13 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)

6.8.2.14 t_pattern

```
typedef struct s_pattern* t_pattern
```

patterns

6.8.2.15 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.

6.8.2.16 t_statement_type

```
typedef enum e_statement_type t_statement_type
```

statement type

6.8.2.17 t_statements

```
typedef struct s_statements t_statements
```

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

6.8.3 Enumeration Type Documentation

6.8.3.1 e_expression_tag

```
enum e_expression_tag
```

expression type

```
109                             {
110     tag_string,
111     tag_char,
112     tag_message,
113     tag_number,
114     tag_ident,
115     tag_block,
116     tag_array,
117     tag_assignment
118 } t_expression_tag;
```

6.8.3.2 e_statement_type

```
enum e_statement_type
```

statement type

```
73                              {
74     stmt_return = 100,
75     stmt_assign,
76     stmt_message
77 } t_statement_type;
```

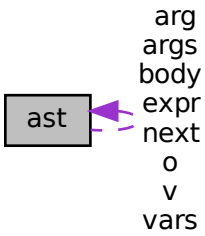
Chapter 7

Data Structure Documentation

7.1 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

7.1.1 Detailed Description

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

7.1.2 Field Documentation

7.1.2.1 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.

7.1.2.2 next

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

next argument

next keyword in the list

7.1.2.3

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

union

7.1.2.4 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

7.2 classinfo Struct Reference

Data Fields

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

7.2.1 Detailed Description

details of a class

7.2.2 Field Documentation

7.2.2.1 meta

```
bool classinfo::meta
```

is it a meta class

7.2.2.2 name

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

name

7.2.2.3 num

```
int classinfo::num
```

number for identification

7.2.2.4 super

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

name of super class

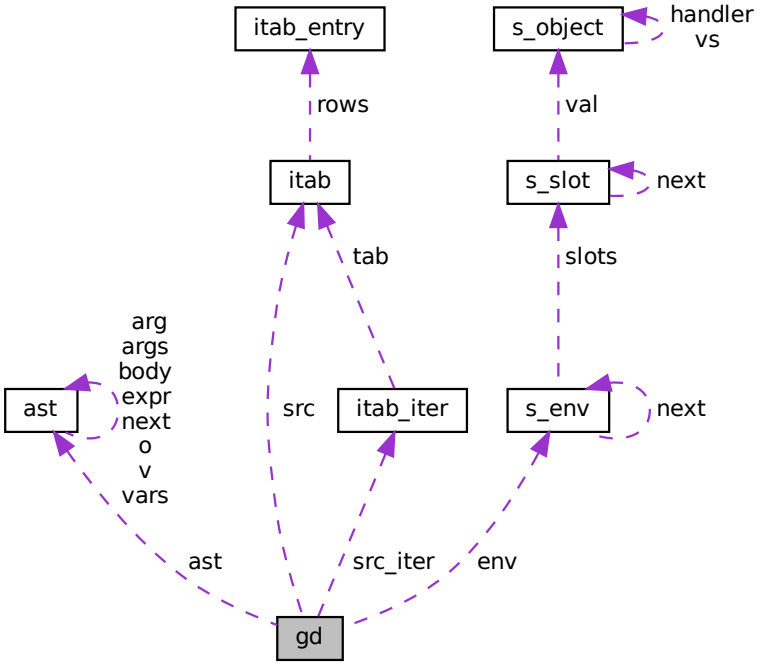
The documentation for this struct was generated from the following file:

- lib.c

7.3 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**

  ...

7.3.1 Detailed Description

structure containing global data

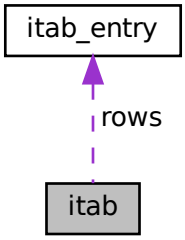
The documentation for this struct was generated from the following file:

- global.h

7.4 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

7.4.1 Detailed Description

structure of itab

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

- lib.c

7.5 itab_entry Struct Reference

structure of an entry in the itab.

Data Fields

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

7.5.1 Detailed Description

structure of an entry in the itab.

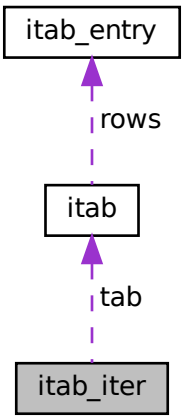
The documentation for this struct was generated from the following file:

- lib.c

7.6 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

7.6.1 Detailed Description

iterator over elements of an itab.

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

- lib.c

7.7 methodinfo Struct Reference

Data Fields

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

7.7.1 Detailed Description

details of a method

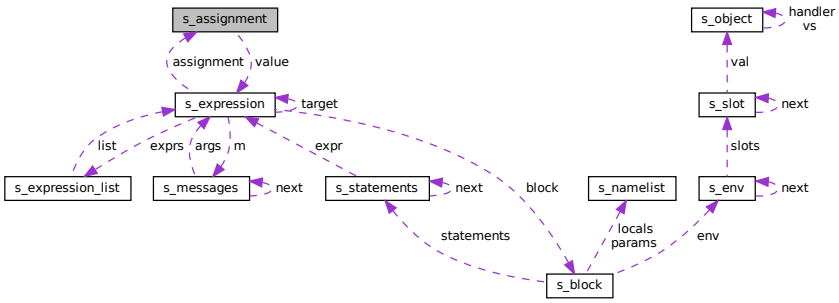
The documentation for this struct was generated from the following file:

- lib.c

7.8 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

7.8.1 Detailed Description

an assignment consists of a target name and an expression

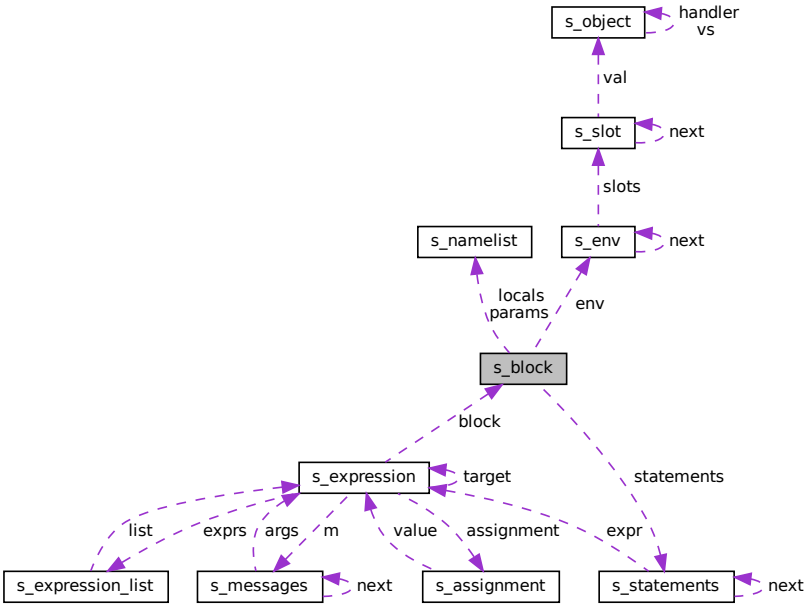
The documentation for this struct was generated from the following file:

- lib.h

7.9 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

7.9.1 Detailed Description

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

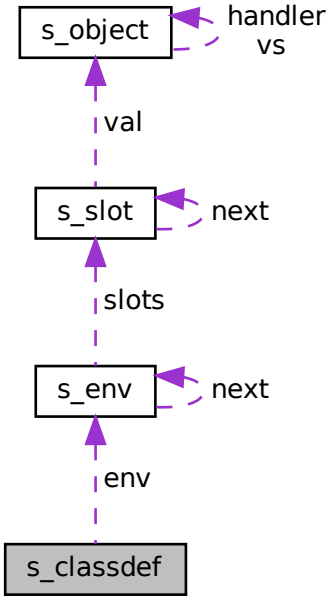
The documentation for this struct was generated from the following file:

- lib.h

7.10 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

7.10.1 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.

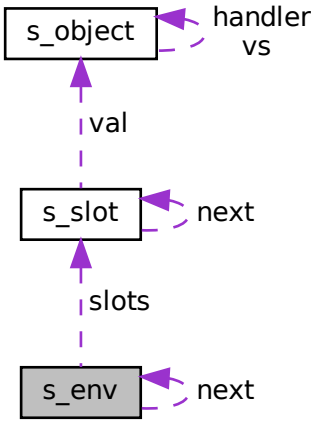
The documentation for this struct was generated from the following file:

- lib.h

7.11 s_env Struct Reference

`#include <lib.h>`

Collaboration diagram for s_env:



Data Fields

- t_slot * **slots**

  slots
- struct s_env * **next**

  next environment

7.11.1 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.

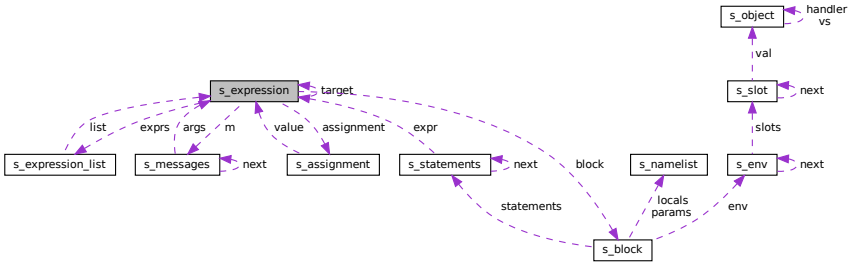
The documentation for this struct was generated from the following file:

- lib.h

7.12 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

7.12.1 Detailed Description

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

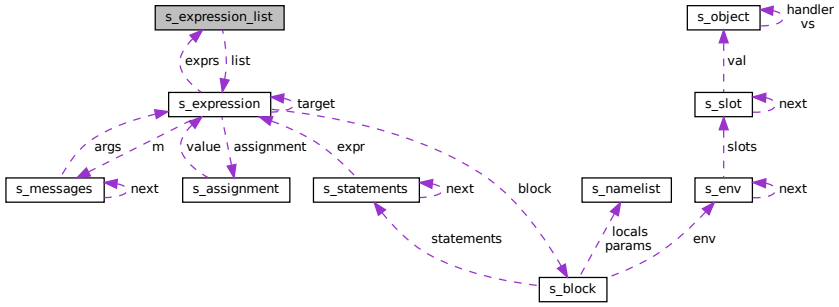
The documentation for this struct was generated from the following file:

- lib.h

7.13 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

7.13.1 Detailed Description

a list of expressions

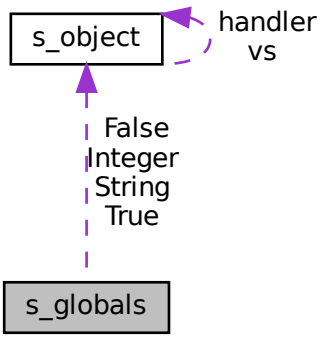
The documentation for this struct was generated from the following file:

- lib.h

7.14 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

7.14.1 Detailed Description

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

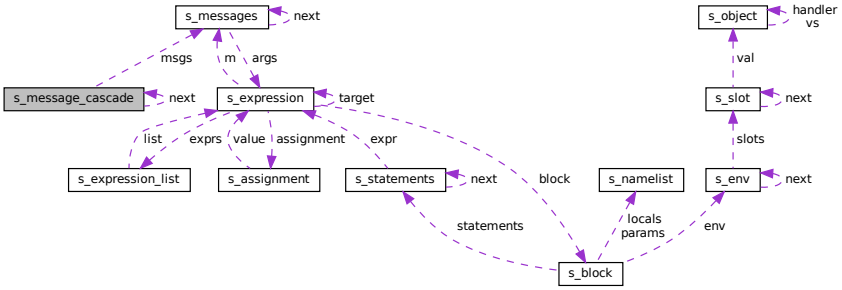
The documentation for this struct was generated from the following file:

- internal.h

7.15 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

7.15.1 Detailed Description

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

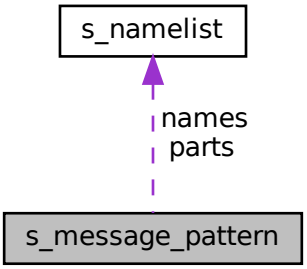
The documentation for this struct was generated from the following file:

- lib.h

7.16 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

7.16.1 Detailed Description

message pattern as it is parsed.

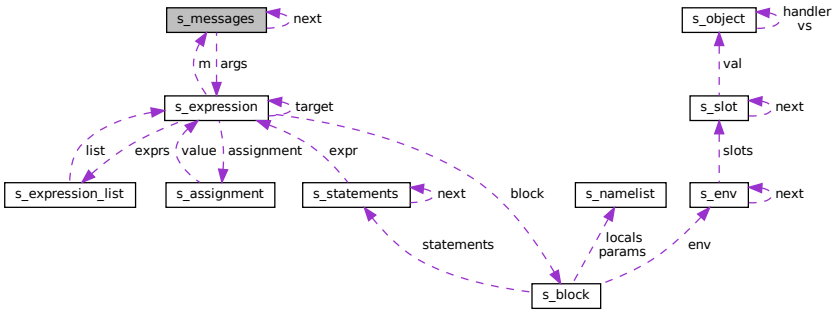
The documentation for this struct was generated from the following file:

- lib.h

7.17 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

7.17.1 Detailed Description

message calling structure

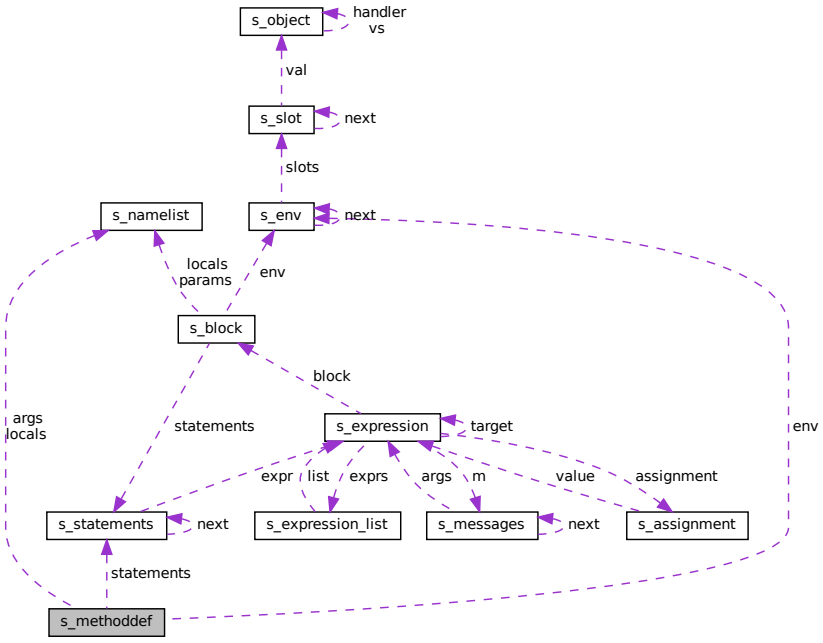
The documentation for this struct was generated from the following file:

- lib.h

7.18 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

7.18.1 Detailed Description

method definition

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

- lib.h

7.19 s_namelist Struct Reference

`#include <lib.h>`

Data Fields

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

7.19.1 Detailed Description

structure containing the counter and array of names.

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

- lib.h

7.20 s_names Struct Reference

`#include <lib.h>`

Collaboration diagram for s_names:

Data Fields

- char * **name**
  name
- t_names **next**
  next

7.20.1 Detailed Description

a linked list of names

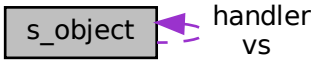
The documentation for this struct was generated from the following file:

- lib.h

7.21 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

7.21.1 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)

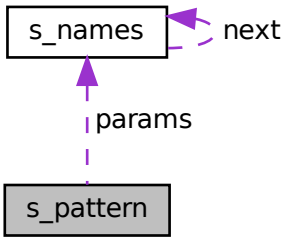
The documentation for this struct was generated from the following file:

- lib.h

7.22 s_pattern Struct Reference

`#include <lib.h>`

Collaboration diagram for s_pattern:



Data Fields

- char * **selector**
  selector
- t_names **params**
  parameter names

7.22.1 Detailed Description

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

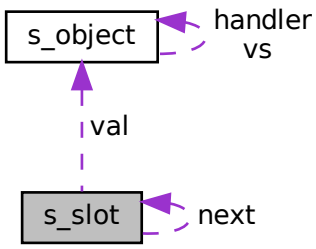
The documentation for this struct was generated from the following file:

- lib.h

7.23 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

7.23.1 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.

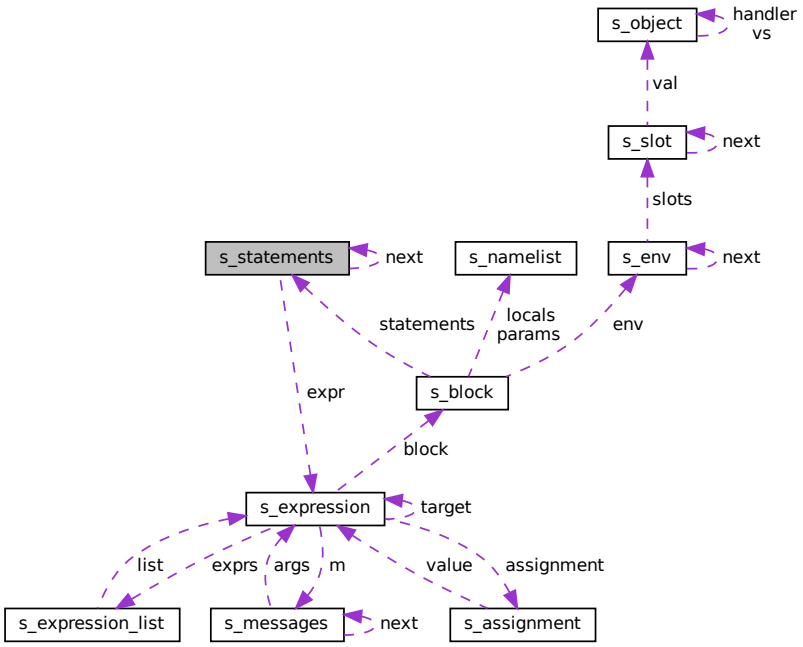
The documentation for this struct was generated from the following file:

- lib.h

7.24 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

7.24.1 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

7.25 stringinfo Struct Reference

Data Fields

- int **num**
  number of the string

7.25.1 Detailed Description

details of a string

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

- lib.c

7.26 varinfo Struct Reference

Data Fields

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

7.26.1 Detailed Description

details of a global variable

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

- lib.c

Chapter 8

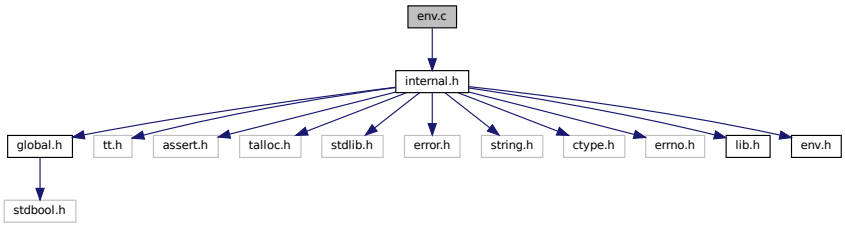
File Documentation

8.1 env.c File Reference

Environment.

```
#include "internal.h"
```

Include dependency graph for env.c:



Functions

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

8.1.1 Detailed Description

Environment.

Author

your name (you@domain.com)

Version

0.1

Date

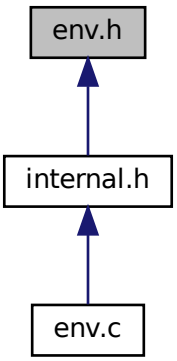
2022-08-09

Copyright

8.2 env.h File Reference

Environment Header.

This graph shows which files directly or indirectly include this file:



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)

8.2.1 Detailed Description

Environment Header.

Author

Peter Jaeckel (jaeckel@acm.org)

Version

0.1

Date

2022-08-09

Copyright

8.3 env.h

Go to the documentation of this file.

```
1
17 void env_add( t_env * env, const t_name name );
19 void env_clear( t_env * env );
21 void env_set( t_env * env, const t_name name, t_object * value );
23 t_slot *env_get( t_env * env, const t_name name );
25 void env_dump(t_env*env, const char*);
29 void env_set_local( t_env * env, const t_name name, t_object * value );
31 t_env *env_new( t_env * parent );
38 t_slot *env_get_all( t_env * env, const t_name name, t_env ** env_found );
39
```

8.4 global.h

```
1 #include <stdbool.h>
2 bool nextToken(  );
3
4 #define AST_STRING          1
5 #define AST_IDENT           2
6 #define AST_UNARY_CALL      3
7 #define AST_STMT            4
8 #define AST_ASSIGN          5
9 #define AST_UNARY           6
10 #define AST_NAMES           7
11 #define AST_CLASS           8
12 #define AST_METHOD          9
13 #define AST_META            10
14 #define AST_ARG             11
15 #define AST_ARGDEF          12
16
20 struct ast {
21     int tag;
22     union {
23         struct {
24             char *v;
25         } str;
26         struct {
27             char *v;
28         } id;
29         struct {
30             struct ast *o;
31             char *sel;
```

```
32              struct ast *arg;
33          } unary;
34          struct {
35              struct ast *v;
36              struct ast *next;
37          } arg;
38          struct argdef {
39              const char *key;
41              const char *name;
42              struct ast *next;
43          } argdef;
44          struct {
45              struct ast *v;
46              struct ast *next;
47          } stmt;
48          struct {
49              char *var;
50              struct ast *expr;
51          } asgn;
52          struct {
53              char *name;
54              char *super;
55              int num;
56              struct ast *vars;
57              struct ast *next;
58          } cls;
59          struct {
60              char *v;
61              struct ast *next;
62          } names;
63          struct {
64              const char *name;
65              struct ast *args;
66              char *classname;
67              char *src;
68              struct ast *body;
69              struct ast *next;
70          } methods;
71      } u;
72 };
73
74 #define meth_name   u.methods.name
75 #define meth_class  u.methods.classname
76 #define meth_body   u.methods.body
77 #define meth_next   u.methods.next
78
82 struct gd {
83     int state;
84     int paridx;
85     int token;
86     int pos;
87     char buf[50];
88     char *line;
89     int line_count;
90     struct ast *ast;
91     int classnum;
92     struct itab *src;
93     struct itab_iter *src_iter;
94     struct s_env * env;
95 };
96
97 extern struct gd gd;
98
99 extern void parse();
```

8.5 internal.h

```
1 #include "global.h"
2 #include "tt.h"
3 #include <assert.h>
4 #include <talloc.h>
5 #include <stdlib.h>
6 #include <error.h>
7 #include <string.h>
8 #include <ctype.h>
9 #include <assert.h>
10 #include <errno.h>
11
12 #include "lib.h"
13 #include "env.h"
14
19 void class_enter( const char *name );
20
```

```
21 bool is_ident_char( int c );
23 bool is_binary_char( int c );
25 bool src_clear( void );
27 bool src_add( const char *line );
29 bool src_read( const char *name );
31 bool src_dump( void );
33 bool readLine( void );
35 bool readChar( char *t );
37 bool readStringToken( void );
39 void parse_verbatim( char c );
41 bool nextToken( void );
43 char *method_name( const char *class, const char *sel );
45 void require_classes( void );
47 void require_current_class( void );
49 void method_enter( t_message_pattern * mp );
50
52 #define MSG_DUMP "dump"
53
57 struct s_globals {
58     t_object *String;
59     t_object *Integer;
60     t_object *True;
61     t_object *False;
62 };
63
65 extern struct s_globals global;
66
68 bool cstr_equals( const char *, const char * );
70 t_object *object_new( t_message_handler hdl );
72 t_object *object_send( t_object * self, const char *sel, t_object ** args );
74 void object_send_void( t_object * self, const char *sel, t_object ** args );
75
76 char *method_name( const char *, const char * );
77
79 t_object *simulate( t_env * env, t_statements * stmts );
81 t_object *eval( t_env * env, t_expression * expr );
82
84 t_object *string_handler( t_object * self, const char *sel,
85                           t_object ** args );
87 t_object *string_meta_handler( t_object * self, const char *sel,
88                                t_object ** args );
90 const char* string_cstr(t_object* self);
91
93 t_object *integer_meta_handler( t_object * self, const char *sel, t_object ** args );
95 t_object *int_handler( t_object * self, const char *sel, t_object ** args );
97 t_object *char_handler( t_object * self, const char *sel, t_object ** args );
99 t_object *block_handler( t_object * self, const char *sel, t_object ** args );
101 t_object *stream_handler( t_object * self, const char *sel,
102                           t_object ** args );
104 t_object *transcript_handler( t_object * self, const char *sel,
105                               t_object ** args );
106
108 t_object *method_exec( t_object * self, const char *clsname, const char *sel,
109                        t_object ** args );
110
118 int itab_lines( struct itab *itab );
120 struct itab *itab_new( void );
122 int itab_entry_cmp( const void *aptr, const void *bptr );
124 void itab_append( struct itab *itab, const char *key, void *value );
126 void *itab_read( struct itab *itab, const char *key );
128 void itab_dump( struct itab *itab );
130 struct itab_iter *itab_foreach( struct itab *tab );
132 struct itab_iter *itab_next( struct itab_iter *iter );
134 void *itab_value( struct itab_iter *iter );
136 const char *itab_key( struct itab_iter *iter );
```

8.6 lib.h

```
1 #ifndef _LIB_H
2 #define _LIB_H
3
4 #define tt_assert(x)                                                    \
5 if (!(x))                                                              \
6 {                                                                       \
7 printf("assert failed:  %s %s %d\n", #x, __FILE__, __LINE__); \
8         msg_print_last();                                               \
9         abort();                                                        \
10     }
11
16 typedef const char *t_name;
20 typedef struct s_namelist {
21     int count;
22     t_name *names;
```

```
23 } t_namelist;
25 typedef struct s_names *t_names;
29 struct s_names {
30     char *name;
31     t_names next;
32 };
40 typedef struct s_expression_list {
41     int count;
42     struct s_expression **list;
43 } t_expression_list;
44
45
50 struct s_pattern {
51     char *selector;
52     t_names params;
53 };
55 typedef struct s_pattern *t_pattern;
56
57
64 typedef struct s_classdef {
65     int id;
66     char *name;
67     char *meta;
68     char *super;
69     struct s_env *env;
70 } t_classdef;
71
73 typedef enum e_statement_type {
74     stmt_return = 100,
75     stmt_assign,
76     stmt_message
77 } t_statement_type;
78
83 typedef struct s_statements {
84     t_statement_type type;
85     struct s_expression *expr;
86     struct s_statements *next;
87 } t_statements;
88
92 typedef struct s_methoddef {
93     char *sel;
94     t_namelist args;
95     t_namelist locals;
96     t_statements *statements;
97     struct s_env *env;
98 } t_methoddef;
99
103 typedef struct s_message_pattern {
104     t_namelist parts;
105     t_namelist names;
106 } t_message_pattern;
107
109 typedef enum e_expression_tag {
110     tag_string,
111     tag_char,
112     tag_message,
113     tag_number,
114     tag_ident,
115     tag_block,
116     tag_array,
117     tag_assignment
118 } t_expression_tag;
119
123 typedef struct s_assignment {
124     const char *target;
125     struct s_expression *value;
126 } t_assignment;
127
131 typedef struct s_block {
132     t_namelist params;
133     t_namelist locals;
134     t_statements *statements;
135     struct s_env *env;
136 } t_block;
137
141 typedef struct s_expression {
142     t_expression_tag tag;
143     union {
144         int intvalue;
145         const char *strvalue;
146         const char *ident;
147         t_expression_list exprs;
148         struct msg {
149             struct s_expression *target;
150             struct s_messages *m;
151         } msg;
152         t_assignment assignment;
```

```
153         t_block block;
154     } u;
155 } t_expression;
156
159 typedef struct s_messages {
160     bool cascaded;
161     char *sel;
162     int argc;
163     t_expression **args;
164     struct s_messages *next;
165 } t_messages;
166
170 typedef struct s_message_cascade {
171     t_messages *msgs;
172     struct s_message_cascade *next;
173 } t_message_cascade;
174
175
177 typedef struct s_object *( *t_message_handler ) ( struct s_object *,
178                                                   const char *sel,
179                                                   struct s_object ** args );
185 typedef struct s_object {
186     t_message_handler handler;
187     union {
188         void *data;
189         int intval;
190         struct {
191             int i[10];
192             void *p[10];
193         } vals;
194         struct {
195             struct s_object **vs;
196             int cnt;
197         } vars;
198     } u;
199 } t_object;
200
206 typedef struct s_slot {
207     const char *name;
208     t_object *val;
209     struct s_slot *next;
210 } t_slot;
211
218 typedef struct s_env {
219     t_slot *slots;
220     struct s_env *next;
221 } t_env;
222
234 bool is_ident_char( int c );
235
242 bool src_clear(  );
243
247 bool src_dump(  );
248
251 bool src_add( const char * );
254 bool src_read( const char * );
255
256 /* @brief read a line from input and store it somewhere.
257 @return true if successful
258 */
259 bool readLine( void );
260
266 bool readChar( char *t );
267
271 bool readStringToken( void );
272
276 bool nextToken( void );
277
281 void ast_dump( int level, struct ast *ast );
282 void ast_fill_classes( struct ast *ast );
283 void ast_fill_methods(  );
284 void ast_generate_methods(  );
285
286 struct ast *ast_new( int tag );
287
288 //void class_add_def(const char* name, const char* super, struct ast * vars, struct ast* methods);
289 //void class_add_meta(const char* name, const char* super, struct ast * vars, struct ast* methods);
290 void class_enter( const char *name );
291 void class_dump_all(  );
292
293 t_methoddef *method_read( const char *class, const char *selector );
294 void method_enter( t_message_pattern * );
295 void method_stmts( t_statements * );
296
297 void string_register( const char *str );
298
299 void c_generate( FILE * );
```

```
300
301 void method_def( t_pattern pattern, void *locals, void *directive,
302                   void *statements );
303 void method_def_verb( t_pattern pattern, void *coding );
304
305 int parser_main( int, char ** );
306
307 void namelist_init( t_namelist * );
308 void namelist_add( t_namelist *, const t_name );
309 void namelist_copy( t_namelist * to, t_namelist * from );
310
311
312 void *itab_read( struct itab *, const char * );
313
314 void msg_add( const char *msg, ...  );
315 void msg_print_last(  );
316
317 void message_add_msg( t_messages * ms, t_messages * m );
318 void message_add_arg( t_messages * m, t_expression * x );
319
320 #endif
```

8.7 proto.h

```
1 /* class_enter.c */
2 void class_enter(const char *name);
3 /* lib.c */
4 void namelist_init(t_namelist *nl);
5 void namelist_add(t_namelist *nl, const char *name);
6 void namelist_copy(t_namelist *to, t_namelist *from);
7 int itab_lines(struct itab *itab);
8 struct itab *itab_new(void);
9 int itab_entry_cmp(const void *aptr, const void *bptr);
10 void itab_append(struct itab *itab, const char *key, void *value);
11 void *itab_read(struct itab *itab, const char *key);
12 void itab_dump(struct itab *itab);
13 struct itab_iter *itab_foreach(struct itab *tab);
14 struct itab_iter *itab_next(struct itab_iter *iter);
15 void *itab_value(struct itab_iter *iter);
16 const char *itab_key(struct itab_iter *iter);
17 _Bool is_ident_char(int c);
18 _Bool is_binary_char(int c);
19 _Bool src_clear(void);
20 _Bool src_add(const char *line);
21 _Bool src_read(const char *name);
22 _Bool src_dump(void);
23 _Bool readLine(void);
24 _Bool readChar(char *t);
25 _Bool readStringToken(void);
26 void parse_verbatim(char c);
27 _Bool nextToken(void);
28 /* method_name.c */
29 char *method_name(const char *class, const char *sel);
30 /* require_classes.c */
31 void require_classes(void);
32 /* require_current_class.c */
33 void require_current_class(void);
34 /* method_enter.c */
35 void method_enter(t_message_pattern *mp);
36 /* method_stmts.c */
37 void method_stmts(t_statements *stmts);
```

8.8 replace.h

```
1 #ifndef REPLACE_H
2 #define REPLACE_H
3
4 #define TALLOC_BUILD_VERSION_MAJOR 2
5 #define TALLOC_BUILD_VERSION_MINOR 3
6 #define TALLOC_BUILD_VERSION_RELEASE 3
7
8 #include <stdbool.h>
9 #include <stdint.h>
10 #include <string.h>
11 #include <errno.h>
12 #include <limits.h>
13 #include <stddef.h>
14 #ifndef MIN
15 #define MIN(a,b) ((a)<(b)?(a):(b))
```

```
16 #endif
17
18 #ifndef MAX
19 #define MAX(a,b) ((a)>(b)?(a):(b))
20 #endif
21
22 #define HAVE_CONSTRUCTOR_ATTRIBUTE
23 #define HAVE_VA_COPY
24
25 #endif
```