Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

A Frame-Based Message-Passing Parser for C

Ctalk, an object-oriented preprocessor for ANSI C, contains a novel parser design that's well suited to interpreting object-oriented languages.

February 01, 2006
URL:http://www.drdobbs.com/cpp/a-frame-based-message-passing-parser-for/184402070

A message-passing parser makes it easy to evaluate and interpret object-oriented languages, because the parser design allows the language interpreter to evaluate language references and the objects they refer to within their context.

The program described in this article, ctalk, is an object-oriented preprocessor for ANSI C that allows programmers to include language features such as classes, objects, and methods in C programs.

Ctalk is an offshoot of a project to write an interpreter for the Smalltalk language, and ctalk's design owes much to the object-oriented concepts and terminology of Smalltalk's design.

In addition, unlike object-oriented extensions to the C language, such as C++ and Objective C, ctalk's design is intended to require as little change to the C source code as possible.

Lexical Analysis And Messages

As with most languages, evaluation requires multiple passes through the input source code. Ctalk's first parser pass performs lexical analysis, splits the input into messages that are stored on the parser's message stack, and adds stack frames for use by the language interpreter in later evaluation steps.

Figure 1 shows the steps the parser uses to create messages and frames, and analyze the messages. The creation and function of stack frames is subsequently described.

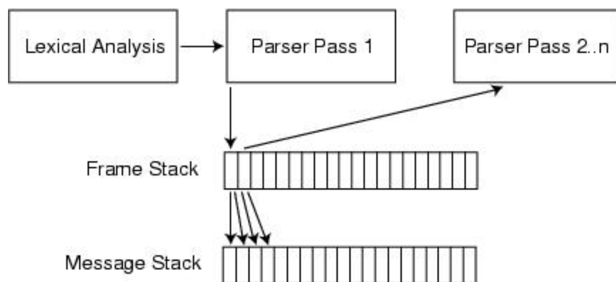

Figure 1: The steps the parser uses to create messages and frames, and analyze the messages.

Listing One shows part of the subroutine `lexical()`, which, like most lexical analyzers, splits the source program's text input contained in the array `buf[]` into tokens, and returns a value corresponding to the input's lexical type.

Listing One

```
int lexical (char *buf, long long *idx, MESSAGE *m) {

  int c, i, j;
  int is_float = 0;
  int tmpint;
  double tmpfloat;
  char tmpbuf[MAXMSG];
  char *q;

  c = buf[(*idx)++];

  /* WHITESPACE */
  if (c == ' ' || c == '\t' || c == '\f') {
    if (!isspace ((int)buf[*idx])) {
      sprintf (m -> name, "%c", c);
      m -> tokentype = WHITESPACE;
      return WHITESPACE;
    }
    for (i = 0, (*idx)--;
```

```
         ((buf[*idx + i] == ' ') || (buf[*idx + i] == '\t') ||
          (buf[*idx] == '\f'));
         i++, (*idx)++)
      m -> name[i] = buf[*idx];
    m -> name[i] = 0;
    m -> tokentype = WHITESPACE;
    return WHITESPACE;
  }

  /* LABEL */
  if (isalnum ((int)c) || c == '_') {
    (*idx)--;
    q = tmpbuf;
    while (isalnum ((int)buf[*idx]) || (buf[*idx] == '_'))
      *q++ = buf[(*idx)++];
    *q = 0;
    strcpy (m -> name, tmpbuf);
    m -> tokentype = LABEL;
    return LABEL;
  }

.
.
.

  sprintf (m -> name, "%c", c);
  m -> tokentype = CHAR;
  return c;
}
```

The third argument to `lexical()`, `m`, is a MESSAGE structure `typedef` that contains the token's text, its token type, and other attributes of the token. Listing Two shows the MESSAGE structure `typedef`. `Lexical()` fills in only the `name` and `tokentype` members, and the interpreter fills in the other attributes during later parser passes.

Listing Two

```
typedef struct _message {
  char sig[7];
  char name[MAXLABEL];
  char value[MAXLABEL];
  OBJECT *value_obj;
  OBJECT *obj;
  int tokentype;
  int evaled;
  int output;
  int error_line;
  int error_column;
  OBJECT *receiver_obj;
  struct _message *receiver_msg;
} MESSAGE;
```

After the token is analyzed, the parser pushes the message onto the message stack. Ctalk could easily pass MESSAGE structures among functions in the parser and interpreter, but instead it passes the message's stack index, and most of the parser and interpreter functions require only the stack index of a message as an argument.

Most of the data types in ctalk contain a `sig` member, which allows the program to check the validity of the data when necessary. The following macro checks for a valid message:

```
#define IS_MESSAGE(x) (!memcmp
             ((void *)x, "MESSAGE", 7))
```

Ctalk's other data types have corresponding macros. If an error occurs within the interpreter, the program can break off processing with a statement like the following:

```
if (!m || !IS_MESSAGE (m))

 error ("%s is not a message.",
                        m -> name);
```

Stack Frames

The first pass of the parser also creates stack frames for each language statement and function in the input source code. Listing Three shows the parser's FRAME structure `typedef`.

Listing Three

```
typedef struct _frame {
  int scope;
  int ret_frame_top;
  int var_frame_top;
```

```
  int message_frame_top;
  int arg_frame_top;
} FRAME;
```

Each frame records the statement's index in the message stack as well in the argument, variable, and return value stacks, which the interpreter uses in later steps.

The first parser pass also determines if the scope of each frame is global or local. The frame's scope in turn determines the scope of newly created objects, and tells the interpreter to delete local objects when they go out of scope.

The use of frames simplifies the parser and interpreter design considerably. The frames delimit the boundaries of each statement in the parser's message stack and the other stacks used by the interpreter, so it is easy to determine the beginning and end of each statement, and the statement's corresponding indexes in the other interpreter stacks.

The second and later parser passes need only the frame's stack index to interpret each statement of the language, and the second pass can evaluate the messages in a stack frame as many times as necessary to resolve the statement to a single value.

Interpreting C

In addition to the main parser, ctalk also has other simpler and more specialized parsers that interpret C preprocessor statements and ANSI C statements.

These parsers maintain their own message stacks, and although they are similar in design to the main parser, they can be much simpler because the values they derive from the input are also much simpler.

This strategy also simplifies the design of the interpreter. By handing off complex but specialized chores to subroutines, the ctalk interpreter can concentrate on evaluating its own language constructs and generating output code.

Listing Four shows the parser that determines if a statement is a C function declaration. The function needs only to return TRUE or FALSE depending on the syntax of its input.

Listing Four

```
int is_c_function_declaration (char *start_ptr) {

  int i;
  int returnval = TRUE;
  int openparen = FALSE;
  int lasttoken = ERROR,
    lasttoken_2 = ERROR,
    lasttoken_3 = ERROR;
  int lasttoken_ptr;
  char *buf;
  char *open_block_ptr;

  if ((open_block_ptr = index (start_ptr, '{')) == NULL)
    return FALSE;

  if ((buf = (char *) calloc (open_block_ptr - start_ptr + 2,
                              sizeof (char))) == NULL)
    error ("Is_c_function_declaration: %s", strerror (errno));


  strncpy (buf, start_ptr, open_block_ptr - start_ptr + 1);
  buf[open_block_ptr - start_ptr + 1] = 0;

  tokenize (buf);

  if (c_messages[c_message_ptr + 1] -> tokentype != OPENBLOCK)
    parser_error
      ("Is_c_function_declaration: Function start not found.");

  for (i = N_MESSAGES; i > c_message_ptr; i--) {

    if ((c_messages[i] -> tokentype == WHITESPACE) ||
        (c_messages[i] -> tokentype == NEWLINE))
      continue;

    switch ( c_messages[i] -> tokentype )
      {
      case LABEL:
        if (!strcmp (c_messages[i] -> name, "main"))
          main_declaration = TRUE;
        break;
      case OPENPAREN:
        if ((c_messages[lasttoken_ptr] -> tokentype != LABEL) ||
            is_c_keyword (c_messages[lasttoken_ptr]->name))
          returnval = FALSE;
        if ((lasttoken_2 != LABEL) && (lasttoken_2 != CHAR))
          returnval = FALSE;
        /* Check for a method declaration. */
        if ((lasttoken_3 == LABEL) && (lasttoken_2 == LABEL) &&
            (lasttoken == LABEL))
```

```
          returnval = FALSE;
        openparen = TRUE;
        break;
      case CLOSEPAREN:
        if (!openparen)
          returnval = FALSE;
        break;
      case SEMICOLON:
        returnval = FALSE;
        goto done;
        break;
      case OPENBLOCK:
        if (c_messages[lasttoken_ptr] -> tokentype != CLOSEPAREN)
          returnval = FALSE;
        break;
      case CLOSEBLOCK:
        returnval = FALSE;
      default:
        break;
      }

    lasttoken_ptr = i;
    lasttoken_3 = lasttoken_2;
    lasttoken_2 = lasttoken;
    lasttoken = c_messages[i] -> tokentype;
  }

 done:
  delete_c_messages ();
  free (buf);

  return returnval;

}
```

`Is_c_function_declaration()` can syntactically match declarations of the following types, as well as others. In addition, the function can distinguish between C function declarations and ctalk method declarations without performing semantic analysis; for example, by checking for ctalk's, "method," method or looking up object references:

```
int main (int argc, char **argv) {
}

char *newstring (void) {
}

FILE *openfile (char *path) {
}
```

Most of the functions that distinguish C language statements, such as function and variable declarations, are able to use the messages of the already tokenized input. `Is_c_function_declaration()`, however, must perform its own tokenization because it helps determine the placement of stack frames during the first pass of the parser.

It is worth noting that `is_c_function_declaration()` checks for the `main()` function declaration. This allows the interpreter to place the ctalk initialization code in the output file at the end of the preamble, immediately before `main()`, and to call the initialization code as the first statement in `main()` after the variable declaration statements. GCC, at least, requires that variable declarations occur at the beginning of a program block, before any statements.

The preprocessor statement parser is slightly more complex than `is_c_function_declaration()` because it must analyze `#ifdef... #else... #endif` clauses, as well as parenthetical subexpressions and `#define` and `#undef` statements.

Subexpression Analysis

When faced with subexpressions, such as the following preprocessor directive, ctalk analyzes the statement from the innermost expression outwards:

```
#if (defined __USE_ISOC99 || \
    (defined __GNUC__ && defined __USE_MISC))
```

Listing Five shows the function `macro_subexpr()`, which recursively analyzes subexpressions in preprocessor statements. The variables `inner_start` and `inner_end` are message-stack indexes that point to the beginning and end of the subexpression, respectively, excluding the subexpression's parentheses.

Listing Five

```
int macro_subexpr (MESSAGE **messages, int start_ptr,
                   int end_ptr)
{

  int i, inner_start, inner_end, inner_result, result;

  /* Look for an inner set of parentheses and evaluate
     that expression first. */
  if ((inner_start =
```

```
        scanforward (messages, start_ptr - 1, end_ptr, OPENPAREN))
      != -1) {
    if ((inner_end =
         match_paren (messages, inner_start, end_ptr + 1))
        != -1) {

      inner_result = macro_subexpr (messages, inner_start,
                                    inner_end);

    }
  }

  result = macro_parse (NULL, start_ptr - 1, end_ptr + 1);

  for (i = start_ptr; i >= end_ptr; i--) {
    ++(messages[i] -> evaled);
    messages[i] -> tokentype = PREPROCESS_EVALED;
    sprintf (messages[i] -> value, "%d", result);
  }

  return result;
}
```

The function `macro_parse()` is the main preprocessor parser. It analyzes each Boolean subexpression or macro define to TRUE or FALSE. The `for()` loop at the end of the function sets the statement's message tokens to PREPROCESS_EVALED to signal that the expression does not need to be evaluated again.

The `macro_subexpr()` function also sets the value of each token in the subexpression to either True or False by setting its message's value member. In the next upper level parser, the following statements can retrieve the value of a subexpression element from its message:

```
case PREPROCESS_EVALED:
  result = atoi (m -> value);
  break;
```

Ctalk messages, when evaluated, typically contain values that are more complex than true or false, so to generalize the MESSAGE structure, it contains members for text strings as well as value objects.

Interpreting the Language

While the ctalk language itself is far from complete, the interpreter design provides the basic functions of an object-oriented language interpreter. Listing Six shows the "Hello, world" program written in ctalk.

Listing Six

```
Object method set_value (char *s) {
  self -> value = strdup (arg(0));
}

Object method value (void) {
  return self -> value;
}

Object class HelloClass;

int main (int argc, char **argv) {

  HelloClass new helloObject;

  helloObject set_value "Hello, world!";

  printf ("%s\n", helloObject value);

  return 0;
}
```

The interpreter uses only three primitive methods written in C: "class," "new," and "method." The interpreter includes their initialization functions in the output program's initialization code, as well as the definition of the "Object" class.

The function `resolve()`, in class.c, which is too lengthy to list here, does the work of resolving objects and methods. Although it might be desirable to write the function using a structure similar to that of `is_c_function_declaration()` in Listing Four, `resolve()` relies on scanning the message stack frame and hands off interpretation chores, like argument processing, to other subroutines.

The design of the "method" primitive also allows operator overloading. If the input source code contains a method declaration such as the following, the interpreter treats it as an overloaded operator, using "+" as an alias to the function "plus" instead of a C language statement:

```
Method + plus (int a) {
}
```

In most cases, the interpreter evaluates primitive objects and methods the same as objects declared in the input source code. When the interpreter encounters a label, it determines whether the label is a reference to an object. If the interpreter finds a label followed by another label, it looks for a method

in the preceding object's method dictionary, or in the method dictionary of the object's superclass.

When `resolve()` locates an object reference (for example, the label "Object" in the statement "Object class HelloClass"), it sets the corresponding message's value to the object, as shown in the following program statements:

```
if ((result_object = get_class_object (m -> name)) != NULL) {
    m -> obj = result_object;
    return result_object;
}
```

Then, when `parser_pass()` calls `resolve()` with the stack index of the next label, the message for the method "class" in this example, `resolve()` scans back to find out if the reference to "Object" has already been resolved.

`resolve()` then checks whether "class" is a method in "Object"s method dictionary, and if so, it performs the following statements—the message `m` in this case contains the method:

```
if (!strcmp (m_prev_label -> obj -> classname, "Class")) {
  if ((method = get_method (m_prev_label -> obj, m ->
                         name))
    != NULL)
  {
    m -> receiver_msg = m_prev_label;
    m -> receiver_obj = m_prev_label -> obj;
    m -> tokentype = METHODLABEL;
  }
```

`resolve()` does the same for instantiated class members such as `helloObject`. In these cases, `resolve()` sets the method message's token type to METHODLABEL and sets the message's `receiver_obj` member to the "Object" object referenced by the previous message's `obj` member. This simplifies the statement's processing further on.

If possible, `resolve()` immediately processes the method message. In conceptual, object-oriented terms, this step is known as, "sending a message to an object." In ctalk's case, a simple function call with the method and its receiver object is sufficient.

The `method_message()` function performs the work of executing the method, if it is a primitive method. If the source code contains the method definition, as in the case of "set_value" and "value," the interpreter inserts the runtime function call into the output, along with the function calls that save the method's arguments both in the method's internal dictionary and on the runtime argument stack.

Because `resolve()` also sets the method message's `receiver_msg` member to the stack index of the receiver message, the methods and method-call protocol can set the `value_obj` of the expression to the statement's value, as in the case of the "new" primitive method, which also sets the class and scope of the new object, and store it in the interpreter's dictionary:

```
m_receiver -> value_obj = arg_object;
```

The method-call protocol also sets the `evaled` message member of each of the statement's messages. If the parser encounters the statement again, it can use `evaled` to determine if further processing is necessary, or simply retrieve the value of the statement, generally from the left-most message, with the following macro:

```
#define M_VALUE_OBJ(m) ((m -> value_obj && \
  IS_OBJECT(m -> value_obj)) ? \
  m -> value_obj : ((m -> obj && IS_OBJECT(m -> obj))\
  ? m -> obj : NULL))
```

The macro M_VALUE_OBJ returns either the message's value object, its object reference, or NULL, depending on how the interpreter processed the message.

The interpreter then, for example, can set the left-most message `value_obj` of the expression "1 + 1.5" to "2.5" without coercing the argument objects to another class, changing the message's token type, or reprocessing the statement.

Again, by handing off specialized chores to subroutines that evaluate particular statements, like method arguments, the design of the interpreter becomes much simpler.

Code Generation

Generating the ctalk runtime code is fairly simple. In most cases, the interpreter is able to simply replace the ctalk statements with the C language function calls for ctalk's runtime functions, in ctalklib.h.

As previously mentioned, C source-code modules do not need drastic changes to use ctalk. GCC compiles C source-code modules after processing with ctalk without additional changes.

As the ctalk language is in an early stage of development, the language imposes a few restrictions of its own. Ctalk does not, at this point, process objects and methods declared extern, and the program must declare methods and global objects in the preamble of the source module that contains `main()`.

In addition, because GCC is the compiler used in ctalk's development, declarations of local objects must occur after the C variable declarations or GCC will produce a syntax error.

The runtime library uses a slightly different interpreter scheme than the ctalk program. It contains stacks only for receivers and arguments, at least at this stage. This design simplifies method calls considerably. The ctalk library implements the keywords `self` (equivalent to `this` in C++) and `arg` as macros.

Conclusion

The ctalk language demonstrates how a frame-based, message-passing parser can provide a simple method to analyze program statements semantically as well as by using language syntax. The designs of the parser and the corresponding interpreter are flexible enough that the program can analyze ANSI C and ctalk statements and generate code that provides the object-oriented language's extensions within the C-language source code.

Robert Kiesling is the former maintainer of The Linux Frequently Asked Questions with Answers Usenet FAQ. He can be contacted at rkiesling@users.sourceforge.net.

Listing 2

```
typedef struct _message {
  char sig[7];
  char name[MAXLABEL];
  char value[MAXLABEL];
  OBJECT *value_obj;
  OBJECT *obj;
  int tokentype;
  int evaled;
  int output;
  int error_line;
  int error_column;
  OBJECT *receiver_obj;
  struct _message *receiver_msg;
} MESSAGE;
```

Listing 3

```
typedef struct _frame {
  int scope;
  int ret_frame_top;
  int var_frame_top;
  int message_frame_top;
  int arg_frame_top;
} FRAME;
```

Listing 4

```
int is_c_function_declaration (char *start_ptr) {

  int i;
  int returnval = TRUE;
  int openparen = FALSE;
  int lasttoken = ERROR,
    lasttoken_2 = ERROR,
    lasttoken_3 = ERROR;
  int lasttoken_ptr;
  char *buf;
  char *open_block_ptr;

  if ((open_block_ptr = index (start_ptr, '{')) == NULL)
    return FALSE;

  if ((buf = (char *) calloc (open_block_ptr - start_ptr + 2,
                              sizeof (char))) == NULL)
    error ("Is_c_function_declaration: %s", strerror (errno));


  strncpy (buf, start_ptr, open_block_ptr - start_ptr + 1);
  buf[open_block_ptr - start_ptr + 1] = 0;

  tokenize (buf);

  if (c_messages[c_message_ptr + 1] -> tokentype != OPENBLOCK)
    parser_error
      ("Is_c_function_declaration: Function start not found.");

  for (i = N_MESSAGES; i > c_message_ptr; i--) {

    if ((c_messages[i] -> tokentype == WHITESPACE) ||
        (c_messages[i] -> tokentype == NEWLINE))
      continue;

    switch ( c_messages[i] -> tokentype )
      {
      case LABEL:
        if (!strcmp (c_messages[i] -> name, "main"))
          main_declaration = TRUE;
```

```
        break;
      case OPENPAREN:
        if ((c_messages[lasttoken_ptr] -> tokentype != LABEL) ||
            is_c_keyword (c_messages[lasttoken_ptr]->name))
          returnval = FALSE;
        if ((lasttoken_2 != LABEL) && (lasttoken_2 != CHAR))
          returnval = FALSE;
        /* Check for a method declaration. */
        if ((lasttoken_3 == LABEL) && (lasttoken_2 == LABEL) &&
            (lasttoken == LABEL))
          returnval = FALSE;
        openparen = TRUE;
        break;
      case CLOSEPAREN:
        if (!openparen)
          returnval = FALSE;
        break;
      case SEMICOLON:
        returnval = FALSE;
        goto done;
        break;
      case OPENBLOCK:
        if (c_messages[lasttoken_ptr] -> tokentype != CLOSEPAREN)
          returnval = FALSE;
        break;
      case CLOSEBLOCK:
        returnval = FALSE;
      default:
        break;
      }

    lasttoken_ptr = i;
    lasttoken_3 = lasttoken_2;
    lasttoken_2 = lasttoken;
    lasttoken = c_messages[i] -> tokentype;
  }

 done:
  delete_c_messages ();
  free (buf);

  return returnval;

}
```

Listing 5

```
int macro_subexpr (MESSAGE **messages, int start_ptr,
                   int end_ptr)
{

  int i, inner_start, inner_end, inner_result, result;

  /* Look for an inner set of parentheses and evaluate
     that expression first. */
  if ((inner_start =
        scanforward (messages, start_ptr - 1, end_ptr, OPENPAREN))
      != -1) {
    if ((inner_end =
         match_paren (messages, inner_start, end_ptr + 1))
        != -1) {

      inner_result = macro_subexpr (messages, inner_start,
                                    inner_end);

    }
  }

  result = macro_parse (NULL, start_ptr - 1, end_ptr + 1);

  for (i = start_ptr; i >= end_ptr; i--) {
    ++(messages[i] -> evaled);
    messages[i] -> tokentype = PREPROCESS_EVALED;
    sprintf (messages[i] -> value, "%d", result);
  }

  return result;
}
```

Listing 6

```
Object method set_value (char *s) {
  self -> value = strdup (arg(0));
}
```

```
Object method value (void) {
  return self -> value;
}

Object class HelloClass;

int main (int argc, char **argv) {

  HelloClass new helloObject;

  helloObject set_value "Hello, world!";

  printf ("%s\n", helloObject value);

  return 0;
}
```

Terms of Service | Privacy Statement | Copyright © 2020 UBM Tech, All rights reserved.