SHILKA (keywords description translator)

Vladimir Makarov, vmakarov@gcc.gnu.org Apr 5, 2001

This document describes SHILKA (translator of a keyword description into code for fast recognition of the keywords).

Contents

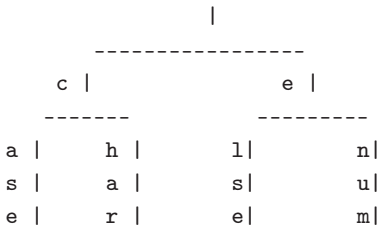
1 Introduction

SHILKA is translator of keywords description into code for fast recognition of keywords and standard identifiers in compilers. SHILKA is implemented with the aid of other components of COCOM toolset.

SHILKA is analogous to GNU package 'gperf' but not based on perfect hash tables. SHILKA rather uses minimal pruned O-trie for for keyword recognition. As consequence SHILKA can take the presumable frequency of keyword occurences in the program into accout. Gperf can not make it. Therefore as rule keyword recognition code generated by SHILKA is faster than one generated by Gperf up to 50%.

SHILKA is suitable for fast recognition from few keywords to huge dictionary of words (strings).

What is minimal pruned O-trie? Let us consider what is trie. If we have four keywords: case, char, else, enum. We can recognize the keywords with the following structure called trie.

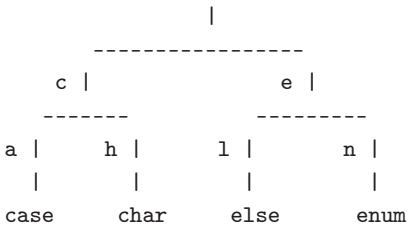


The corresponding code for the keywords recognition based on this structure could be

```
if (kw[0] == 'c')
  {
    if (kw[1] == 'a')
      {
        if (kw[2] == 's')
          {
            if (kw[3] == 'e')
              {
                /* we recognize keyword case */
              }
            else
              /* this is not a keyword */
          }
        else
         /* this is not a keyword */
      }
    else if (kw[1] == 'h')
      {
        if (kw[2] == 'a')
          {
            if (kw[3] == 'r')
              {
                /* we recognize keyword char */
              }
            else
              /* this is not a keyword */
          }
        else
         /* this is not a keyword */
      }
    else
      /* this is not a keyword */
  }
else if (kw[0] = 'e')
  {
    if (kw[1] == 'l')
      {
        if (kw[2] == 's')
```

```
            {
              if (kw[3] == 'e')
                {
                  /* we recognize keyword else */
                }
              else
                /* this is not a keyword */
            }
          else
           /* this is not a keyword */
        }
      else if (kw[1] == 'n')
        {
          if (kw[2] == 'u')
            {
              if (kw[3] == 'm')
                {
                  /* we recognize keyword enum */
                }
              else
                /* this is not a keyword */
            }
          else
           /* this is not a keyword */
        }
      else
        /* this is not a keyword */
    }
```

You can see in the example above that it is not necessary to test all characters of the keywords. Instead of this, we can test only several characters of the keywords and test all kewyord at the end of final decision that given string is a keyword. Such technique results in another structure called pruned trie:

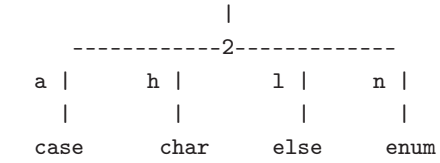


The corresponding code for the keywords recognition based on this structure could be

```
 if (kw[0] == 'c')
   {
     if (kw[1] == 'a')
       {
         if (strcmp (kw, "case") == 0)
           /* we recognize keyword case */
         else
           /* this is not a keyword */
```

```
      }
    else if (kw[1] == 'h')
      {
        if (strcmp (kw, "char") == 0)
          /* we recognize keyword char */
        else
          /* this is not a keyword */
      }
    else
      /* this is not a keyword */
  }
else if (kw[0] = 'e')
  {
    if (kw[1] == 'l')
      {
        if (strcmp (kw, "else") == 0)
          /* we recognize keyword else */
        else
          /* this is not a keyword */
      }
    else if (kw[1] == 'n')
      {
        if (strcmp (kw, "enum") == 0)
          /* we recognize keyword enum */
        else
          /* this is not a keyword */
      }
    else
      /* this is not a keyword */
  }
```

Probably you found that if we test keywords characters in another order (not in sequential order), we could recognize keywords faster. Using such approach results in another structure called pruned O-trie:



Here number on the intersection means what keyword character (1st, 2nd, ...) is tested. The corresponding code for the keywords recognition based on this structure could be

```
if (kw[1] == 'a')
  {
    if (strcmp (kw, "case") == 0)
      /* we recognize keyword case */
    else
      /* this is not a keyword */
  }
else if (kw[1] == 'h')
```

```
  {
    if (strcmp (kw, "char") == 0)
      /* we recognize keyword char */
    else
      /* this is not a keyword */
  }
else if (kw[1] == 'l')
  {
    if (strcmp (kw, "else") == 0)
      /* we recognize keyword else */
    else
      /* this is not a keyword */
  }
else if (kw[1] == 'n')
  {
    if (strcmp (kw, "enum") == 0)
      /* we recognize keyword enum */
    else
      /* this is not a keyword */
  }
else
  /* this is not a keyword */
```

And finally, minimal in the phrase "minimal pruned O-trie" means that we found pruned O-trie with minimal number of testing the keyword characters. Generally speaking we can introduce notion cost for pruned O-trie and search for prunned O-trie with minimal cost. Shilka takes probability of keyword occurences in program into account for evaluation of the cost of prunned O-trie.

2 Keywords description language

A keywords description describes mainly keywords (or standard identifiers) of a language and possibly the keywords frequencies in typical program. The frequencies are used for generation of pruned O-trie for faster keyword recognition in the program. The default action returning code of recognized keyword can be changed by user defined action, e.g. returning pointer to the structure describing given keyword.

2.1 Layout of keywords description

Keywords description structure has the following layout which is similar to one of YACC file.

```
DECLARATIONS
%%
KEYWORDS
%%
ADDITIONAL C/C++ CODE
```

The '%%' serves to separate the sections of description. All sections are optional. The first '%%' starts section of keywords and is obligatory even if the section is empty, the second '%%' may be absent if section of additional C/C++ code is absent too.

The section of declarations contains optional declaration of type of generated function for keyword recognition. If the type is given then actions for all keywords and keyword '%other' with the action should be given. Otherwise the function will return codes of recognized keywords (of type 'int' or 'enum' – see SHILKA Usage). The section of declarations may contain subsections of code on C/C++.

The next section contains keywords list which describes keyword itself and optional code of the keyword, frequency, and action.

The additional C/C++ code can contain any C/C++ code you want to use. Often functions which are not generated by the translator but are relative to the keyword recognition can go here. This code without changes is placed at the end of file generated by the translator.

Full YACC syntax of SHILKA description file is placed in Appendix 1.

2.2 Declarations

The section of declarations may contain the following construction:

```
%type IDENTIFIER
```

Only the first such construction is taken into account. All subsequent constructions '%type' are ignored.

Such constructions declare type of data recognized by function 'KR_find_keyword'. E.g. the function can return pointer to structure which contains additional information of recognized keyword. By default the function returns only code of recognized keyword (of integer type or enumeration – see SHILKA Usage). If construction '%type' is present, we should also put construction '%other' in the keyword section and actions returning data of the declared type for each keyword definition in the section of keywords. Otherwise SHILKA will generate warning messages.

There may be also the following constructions in the declaration section

```
%local {
   C/C++ DECLARATIONS
}

%import {
   C/C++ DECLARATION
}

and

%export {
   C/C++ DECLARATION
}
```

which contain any C/C++ declarations (types, variables, macros, and so on) used in sections.

The local C/C++ declarations are inserted at the begin of generated implementation file (see section 'generated code') but after include-directive of interface file (if present – see SHILKA Usage).

C/C++ declarations which start with '%import' are inserted at the begin of generated interface file. If the interface file is not generated, the code is inserted at the begin of the part of implementation file which would correspond the interface file.

C/C++ declarations which start with '%export' are inserted at the end of generated interface file. For example, such exported C/C++ code may contain definitions of external variables and functions which refer to definitions generated by SHILKA. If the interface file is not generated, the code is inserted at the end of the part of implementation file which would correspond the interface file.

All C/C++ declarations are placed in the same order as in the section of declarations.

2.3 Keywords

The section of declarations is followed by section of keyword definitions. Usually keyword definition is described by the following construction

```
IDENTIFIER = IDENTIFIER frequency action
```

What is identifier. What is action.

Here the first identifier is a keyword itself. The second identifier determines the keyword name which will be generated as macro definition or enumeration (see SHILKA usage) for given keyword. The name is formed from the prefix KR_ and the second identifier. The second identifier with = may be absent. In this case the keyword name is formed from prefix KR_ and the keyword itself (i.e. the first identifier).

Optional frequency of the keyword occurence in program (in some abstract unities) is integer value and is used for generation of minimal cost pruned O-trie. If the frequency is absent, its value will be 1.

Optional action (any C/C++ block) is executed when the keyword is recognized. Usually the action will be only returning pointer to the structure describing the recognized keyword. Default action is simply returning the keyword name whose value is keyword code defined as macro or enumeration constant. If you are going to return data distinct from the keyword code, we should use declaration %type (see above) and should define actions for all keywords and for special keyword definition %other with action (see below). Otherwise SHILKA will generate warning messages.

Identifier in SHILKA is the same as one in C. If you want to recognize also strings distinct from SHILKA identifier, you should use the following construction

```
STRING = IDENTIFIER frequency action
```

The string here is C string. The identifier after = is obligatory in such construction. All other is the same as in the previous construction.

The following construction determine optional action when no one keyword is recognized.

```
OTHER action
```

Default action is only returning special code KR_ _not_found. If you are going to return data distinct from the keyword code, we probably should return value NULL here.

Only the first such construction is taken into account. All subsequent such constructions are ignored.

3 Generated code

A specification as described in the previous section is translated by SHILKA (keywords description translator) into interface and implementation files having the same names as one of specification file and correspondingly

suffixes '.h' and '.c' (C code) or '.cpp' (C++ code). By default the interface file is not generated.

3.1 C++ code

The interface and implementation files consist of the following definitions of generated macros, types, and functions:

Class 'KR_recognizer'.

Object of the class describes a keyword recognizer. If the interface file is generated, the interface file contains the class. The implementation file contains the class functions themself. The class has the following public members:

1. Function

   ```
   '<type> KR_find_keyword (const char *KR_keyword,
                            int KR_length)'
   ```

   The function has two parameters: keyword being recognized and length in characters of the keyword. By default the function returns code of recognized keyword (integer type or enumeration – see flag '-enum'). The function will return value of type given in construction '%type' of the description file. Usually the value contains some additional information about recognized keyword. By default the keyword is suggested to be stored according to C representation (with zero byte as end marker). If this is not true, you can use option '-length' during generation of the keyword recognizer by SHILKA. In this case the function 'strncmp' instead of 'strcmp' is used for keywords comparison.

   If option '-inline' has been used, the function is defined as inline. This can significantly speed up keywords recognition.

2. Function

   ```
   'void KR_output_statistics (void)'
   ```

   The function outputs number of occurrences for each keywords since the keyword recognizer start. The function also outputs relative frequencies of occurrences (this value is always nonzero). The frequencies output after work recognizer on a large typical program can be used for setting up keyword frequencies in the description file for generation of faster keyword recognizer. The function do nothing if SHILKA flag '-strip' has been used during the keyword recognizer or flag '-statistics' has been not used. Flag '-statistics' simply generates macro definition '__KR__DEBUG__' which includes code for picking up and outputting statistics about keyword occurrences.

3. Function

   ```
   'void KR_reset (void)'
   ```

   The function initiates keyword recognizer.

4. Constructor

   ```
   'void KR_recognizer (void)'
   ```

   The constructor simply calls function 'KR_reset'.

5. Destructor

   ```
   '~KR_recognizer (void)'
   ```

   This destructor is for accurate finish work with the keyword recognizer.

Macros or enumeration (see option '-enum')

which declare keyword codes. Macros or enumeration constants have the same name as one in the description file and prefix 'KR_' (see also option '-p'). SHILKA generates additional code 'KR__not_found'. By default such code means that any keyword is not found. Macros or enumeration are generated in interface file only if option '-export' is present on SHILKA command line. By default, the macros or the enumeration are generated in the implementation file. Usually, the last case means that only the keyword recognizer implementation file is used, and the file is included into another C/C++ file.

3.2 C code

The interface and implementation files consist of the following definitions of generated macros, types, and functions:

1. Macros or enumeration (see option '-enum') which declare keyword codes. Macros or enumeration constants have the same name as one in the description file and prefix 'KR_' (see also option '-p'). SHILKA generates additional code 'KR__not_found'. By default such code means that any keyword is not found. Macros or enumeration are generated in interface file only if option '-export' is present on SHILKA command line. By default, the macros or the enumeration are generated in the implementation file. Usually, the last case means that only the keyword recognizer implementation file is generated, and the file is included into another C/C++ file.

2. Function

   ```
   '<type> KR_find_keyword (const char *KR_keyword,
                            int KR_length)'
   ```

   The function has two parameters: keyword being recognized and length in characters of the keyword. By default the function returns code of recognized keyword (integer type or enumeration – see flag '-enum'). The function can also return value of type given in construction '%type' of the description file. Usually the value contains some additional information about recognized keyword. By default the keyword is suggested to be stored according to C representation (with zero byte as end marker). If this is not true, you can use option '-length' during generation of the keyword recognizer by SHILKA. In this case the function 'strncmp' instead of 'strcmp' is used for keywords comparison.

   If option '-inline' has been used, the function is defined as inline. This can significantly speed up keywords recognition.

   If the interface file is generated, the interface file contains external definition of the function. The implementation file contains the function itself.

3. Function

   ```
   'void KR_reset (void)'
   ```

   The function initiates keyword recognizer. Therefore the function must be called by the first. If the interface file is generated, the interface file contains external definition of the function. The implementation file contains the function itself.

4. Function

   ```
   'void KR_output_statistics (void)'
   ```

The function outputs number of occurrences for each keywords since the keyword recognizer start. The function also outputs relative frequencies of occurrences (this value is always nonzero). The frequencies output after work recognizer on a large typical program can be used for setting up keyword frequencies in the description file for generation of faster keyword recognizer. The function do nothing if SHILKA flag '-strip' has been used during the keyword recognizer or flag '-statistics' has been not used. Flag '-statistics' simply generates macro definition '__KR__DEBUG__' which includes code for picking up and outputting statistics about keyword occurrences.

If the interface file is generated, the interface file contains external definition of the function. The implementation file contains the function itself.

4 SHILKA Usage

```
SHILKA(1)                        User Manuals                        SHILKA(1)



NAME
       shilka - keywords description translator

SYNOPSIS
       shilka [ -c++ --statistics -inline -strip -length -case -no-definitions
       -interface -export -enum -pprefix -time -fast n -w -h -help -v]  speciâĂŘ
       fication-file

DESCRIPTION
       SHILKA  generates code for fast recognition of pipeline hazards of proâĂŘ
       cessor which is described in  specification  file.   The  specification
       file must have suffix '.shilka'

       The  generated  code  consists of optional interface and implementation
       files having the same names as one of  specification  file  and  correâĂŘ
       spondingly  suffixes  '.h'  (if option -interface is given) and '.c' (C
       code) or '.cpp' (C++ code).

       The fast recognition of keywords is based on structure called as  miniâĂŘ
       mal  prunned  0-trie.  Full documentation of SHILKA is in SHILKA User's
       manual.

OPTIONS
       The options which are known for SHILKA are:

       -c++   Output of C++ code instead of C code (which is default).

       -statistics
              Generation of macro definition which switches on  gathering  and
              printing  keyword  occurrence  statistics.  Then such statistics
              can be used for setting up frequencies in the  description  file
              for faster recognition of keywords.
```

-inline
Generation of function of keyword recognition as inline (take care with used C compiler). This can be speed up keyword recogâĂŘ nition.

-strip Striping off code for gathering and printing statistics. By default the code is present in the implementation file. The code can be switched on by definition of a macro (see option -statistics).

-length
Usage of strncmp instead of strcmp to compare keywords. This can be useful if keyword is represented by a string without C string end marker (' ').

-case Keywords case is ignored during their recognition. This is useâĂŘ ful for recognition keywords (or standard identifiers) in such language as Fortran.

-no-definitions
No generation of macros (or enumeration) defining identifiers of keywords.

-interface
Generation of additional interface file (with suffix '.h').

-export
SHILKA generates macros defining identifiers of keywords in the interface file (instead of in the implementation file). This option has sense only when -interface is present.

-enum Output of enumeration instead of macro definitions for identiâĂŘ fiers of keywords.

-pprefix
Usage of prefix instead of 'KR_' (default) for names of generâĂŘ ated objects.

-time Output of time statistics of the SHILKA run into stderr.

-fast n
If number of rested unchecked characters is less or equal n, then functions 'strcmp' or 'strncmp' is not used at all. Instead of this, simply comparing rested unchecked characters is used. This can speed up keyword recognition. The default value of the parameter is 3.

-w Disable generation of all warnings.

-h, -help
Output of brief help message about SHILKA usage.

```
       -v     Creation  of  description  file  containing details how the code
              generated by SHILKA will recognize the keywords.

FILES
       file.shilka
              SHILKA specification file
       file.c
              generated C implementation file
       file.cpp
              generated C++ implementation file
       file.h
              generated interface file

       There are no any temporary files used by SHILKA.

ENVIRONMENT
       There are no environment variables which affect SHILKA behavior.

DIAGNOSTICS
       SHILKA diagnostics is self-explanatory.

AUTHOR
       Vladimir N. Makarov, vmakarov@gcc.gnu.org

SEE ALSO
       msta(1), oka(1), sprut(1), nona(1).  SHILKA manual.

BUGS
       Please, report bugs to https://github.com/dino-lang/dino/issues.



COCOM                              5 APR 2001                          SHILKA(1)
```

5 Future of SHILKA development

Some C/C++ compilers generate bad code for switch-statement. In this case, generation of table-driven keyword recognition based on minimal pruned O-trie would be useful.

6 Appendix 1 - Syntax of SHILKA description file (YACC description)

```
%token PERCENTS LOCAL IMPORT EXPORT TYPE EQUAL OTHER
%token IDENTIFIER STRING CODE_INSERTION ADDITIONAL_C_CODE
%token NUMBER
%start description
```

```
%%

description : declaration_part  PERCENTS
              keyword_definition_list  ADDITIONAL_C_CODE
            ;

declaration_part :
                 | declaration_part  declaration
                 ;

declaration : TYPE IDENTIFIER
            | LOCAL  CODE_INSERTION
            | IMPORT CODE_INSERTION
            | EXPORT CODE_INSERTION
            ;

keyword_definition_list :
                        | keyword_definition_list  keyword_definition
                        ;

keyword_definition : IDENTIFIER optional_name frequency optional_action
                   | STRING EQUAL IDENTIFIER frequency optional_action
                   | OTHER optional_action
                   ;

optional_name :
              | EQUAL IDENTIFIER
              ;

frequency :
          | NUMBER
          ;

optional_action :
                | CODE_INSERTION
                ;
```

7 Appendix 2 - Example of SHILKA description file

```
/* Keywords of ANSI C. */
%import {
   struct keyword {int kw; int primitive_type_flag;};
   typedef struct keyword *keyword_t;
}
%type keyword_t
%%
if            {static struct keyword res = {KR_if, 0}; return }
do            {static struct keyword res = {KR_do, 0}; return }
int           {static struct keyword res = {KR_int, 1}; return }
```

```
for           {static struct keyword res = {KR_for, 0}; return }
case          {static struct keyword res = {KR_case, 0}; return }
char     1600 {static struct keyword res = {KR_char, 1}; return }
auto          {static struct keyword res = {KR_auto, 0}; return }
goto          {static struct keyword res = {KR_goto, 0}; return }
else     5700 {static struct keyword res = {KR_else, 0}; return }
long          {static struct keyword res = {KR_long, 0}; return }
void     8200 {static struct keyword res = {KR_void, 1}; return }
enum          {static struct keyword res = {KR_void, 0}; return }
float         {static struct keyword res = {KR_float, 1}; return }
short         {static struct keyword res = {KR_short, 0}; return }
union         {static struct keyword res = {KR_union, 0}; return }
break         {static struct keyword res = {KR_break, 0}; return }
while         {static struct keyword res = {KR_while, 0}; return }
const         {static struct keyword res = {KR_const, 0}; return }
double        {static struct keyword res = {KR_double, 0}; return }
static   7300 {static struct keyword res = {KR_static, 0}; return }
extern        {static struct keyword res = {KR_extern, 0}; return }
struct        {static struct keyword res = {KR_struct, 0}; return }
return   1000 {static struct keyword res = {KR_return, 0}; return }
sizeof   1100 {static struct keyword res = {KR_sizeof, 0}; return }
switch        {static struct keyword res = {KR_switch, 0}; return }
signed        {static struct keyword res = {KR_signed, 0}; return }
typedef       {static struct keyword res = {KR_typedef, 0}; return }
default       {static struct keyword res = {KR_default, 0}; return }
unsigned      {static struct keyword res = {KR_unsigned, 0}; return }
continue      {static struct keyword res = {KR_continue, 0}; return }
register      {static struct keyword res = {KR_register, 0}; return }
volatile      {static struct keyword res = {KR_volatile, 0}; return }
%other        {return NULL;}
```

8 Appendix 3 - Run of SHILKA on the previous description file

This output is produced on Compaq Aero 486 SX/25 with 12Mb under Linux 2.0.29.

```
bash$ shilka -time d.shilka

32 keywords, 32 different keyword names
5.18 average keywords length, 4.80 weighted average keywords length
34600 achieved trie cost, 27500 minimal cost, 125 % ratio

  building pruned 0-trie: 0.10, output: 0.29
Summary:
  parse time 0.11, analyze time 0.00, generation time 0.39, all time:0.51
```