数理解析研究所講究録
1185 巻 2001 年 182-190

CKit: A Preprocessor for Algorithmic Experiments

Mario SZEGEDY

Rutgers University
110 Frelinghuysen Road,
Piscataway, NJ 08854-8019 USA
szegedy@cs.rutgers.edu

Abstract: CKit [8] is a general purpose preprocessor that I originally designed for and tested in code theoretical and combinatorial experiments. In this article I explore the most important features of this preprocessor, describe its design principles, and illustrate its novel use with examples. I introduce the notions of *macros with associative power* and *lexical inheritance*, and give examples to their use in CKit.

Keywords: programming languages, preprocessors, combinatorial experiments.

1 Introduction

In November, 1996 I finished implementing an experimental version of a new preprocessor for the programming language C. The version of this preprocessor, that I call CKit, runs on UNIX platform, and is mostly designed to help generating programs for a variety of combinatorial algorithmic experiments. Among existing software CKit is most similar to m4 [5] [6], which is a general purpose preprocessor implemented on UNIX platform, but the most influential languages to the design of CKit were Tcl and Perl, and not m4.

CKit extends the lexical and preprocessing abilities of known preprocessors (such as gcc and m4), so that if the programmer chooses to reach his/her goal by writing a lot of macros (instead of C procedures), CKit provides a powerful tool to do so. From the habit of writing an excessive amount of macros I arrived at a new approach to program organization in which the building blocks of the program are editorial modules. These do not simply correspond to C procedures, but they also allow for relevant syntactic shortcuts. Whether by making the CKit features even more general and powerful we can buy complete syntactic freedom for ourselves is an open question. I have written packages that help to imitate the look and feel of the Maple and Mathematica platforms. While we get the same speed in program writing, C programs, that are generated with CKit, can run more than 200 times faster than their Maple counterparts. This dramatic difference occurs chiefly for search type problems.

2 Objectives and Design Principles

CKit operates solely on text, and is oblivious about the target program's semantics. Its environment typically consists of several macro packages and master files. (There is no conceptual difference between macro packages and master files.) Macro packages can serve many purposes. One use of them is to create commands that give high level features and brevity to programs. Another typical use of them is identical to that of old fashioned C packages (i.e. to serve as a pool of useful methods), but the most typical use is the combination of both. Master files describe programs, input-output transformations, and possibly different modes to run user programs. CKIT's further development hinges on the following plan:

1. To automate a wide range of program editorial activities. In the input the programmer should be able to describe these ideas in a simple language.

2. To build on our observation that useful design concepts, such as that of inheritance for object oriented languages, have lexical counterparts that use surprisingly simple resources. Correspondingly CKit intro-

duces and implements *macros with associative power*, and *lexical inheritance* (see later).

3. To reduce code repetition. Code repetition is said to be eliminated completely by object oriented programming, but many contest this claim.

4. To use a small number of general purpose commands, and defer all specifics to the macro packages.

5. To convert the "garbage" produced during program development into "beauty." In particular, CKit should provide a friendly environment for alternative program constructs that coexist until the programmer finishes his/her experiments and chooses his/her favorite one.

6. Possibly to provide means for program testing.

3 The Structure of the CKit Text

The CKit text is made up of six components:

Commands are editorial instructions, which determine how and when text replacements and insertions take place. They also offer means of communication between the programmer, CKit, and the UNIX shell.

Macro Calls call for the text which is associated with the macro definition of their identifier. The number of arguments of the caller must agree with the number of arguments of the definition, otherwise the macro call is not recognized.

Labels mark places in the text for later reference. After the entire text has been processed, labels disappear. When `text` is sent to `@label` by the `label += text$` command, the label reappears at the end of the text sent. Thus consecutive insertions to the same label cause the corresponding texts to be arranged in the order of their insertion.

Empty Replacements are used to separate identifiers and to protect arguments with commas. As soon as the parser recognizes them, they disappear from the program text.

Comments of the form `/< text >/` delimit `text` which is to be eliminated by the parser.

CKit Externals are those strings of characters that do not belong to any of the above categories, e.g., identifiers that are not macro calls, most non-alphabetical characters, etc. These do not prompt an action from the parser, and they are skipped during processing.

These components can follow each other in any order, and in any number. However, semantics will naturally determine their order, e.g., an identifier is recognized as a macro call only in case it appears in the text after the corresponding macro definition. Some CKit commands may embed other commands, macro calls, etc. These commands are called *inclusive commands* and will be discussed more details in the next section.

4 CKit Commands

4.1 Types of CKit Commands

CKit command fall into four categories:

1. *Cut and paste commands,* e.g. `marker+:text$`, which cuts `text` and pastes it to the place marked by `@marker`. Other commands in this category are `cond??text$`, which expands to `text` if `cond` holds (`cond` is of the form `macrocall=value`), otherwise it expands to the empty string. `name!?text$` expands to `text` if `name` is not defined, otherwise it expands to the empty string. In both cases `text` is parsed from its start.

2. *Macro creator commands.* The most frequently used of them is `macroname/arglist ::text$`. This command either creates a macro, if it does not exist already, or if it does, then it causes the old definition to be replaced with the new one. The command `macroname/arglist+:text$` adds `text` to the text of an existing macro, and if the macro does not exist, it creates a new one.

Op.	Syntax	Description
::	`block +: text$`	Defines a macro.
+:	`block +: text$`	Adds extra text to macro `block` or to a place marked with `block`
??	`block ?? text$`	Includes text under condition defined in `block`.
!?	`block !? text$`	Includes text if `block` is not defined.
><	`>< block`	Invalidates a previously defined macro.
<:	`block1 <: block2`	Causes `block1` to inherit all lexical properties of `block2`.
|>	`|> block`	Interpolates a file.

3. *Commands defining lexical associations.* The CKit command for creating an associative array is `name1//name2::text$`, where `text` is the value of `name1` for argument `name2`. In CKit macros and associative arrays, and their common generalization, which is called *macros with associative power* are handled by the same mechanism. *Lexical Inheritance*, when an identifier inherits all associative features of another one, is an artifact of this machinery. The corresponding command `name1<:name2` is discussed in Chapter 9 in more details.

4. *CKit control flow commands.* CKit flow control commands play a role in experimenting with alternative program constructs. These commands are new to CKit and they are out of the scope of the present discussion.

4.2 The Syntax of CKit Commands

Every CKit command has to contain a command operator, which is a two letter symbol. The most important operators are summarised in the table in the top of the page.

Block: A sequence of non-whitespace characters preceded and followed by a whitespace character.

Text: Arbitrary sequence of characters in which every `$` symbol is the closing mark of an embedded CKit command.

The block of non white space characters that is immediately before the command operator is called the *preclause* of a command. The *postclause* of a command is either `block` or a `text$` (see the definition of block and text above). White spaces around the command operator are optional. Some CKit commands do not have preclause or the postclause. Commands which have `text$` as postclause are called `inclusive`, for other CKit commands can be part of `text`. Command must be followed by a white space character or by the `$` symbol.

5 Creating and Expanding Macros

Macros are identifiers which define text, and come with or without arguments. If a macro has arguments, it defines text patterns with variable parts to be filled in. In CKit there are two different ways to denote the variable parts. `MAX/x/y::($1)>($2)?($1):($2)$` and `MAX/x/y::(x)>(y)?(x):(y)$` are equally legal definitions of the same macro. The number of arguments of a macro is fixed in CKit. If a macro is called with the wrong number of arguments, it is not an error: it simply will be recognized as a CKit external. When macros are expanded, first their arguments are interpolated, the macro call is then replaced with the text of the macro, which is then parsed from the start. The `@@` string separates identifiers, which disappears after parsing, e.g., `concat/x/y::x@@y$` defines a macro which concatenates its two arguments.

CKit introduces a new array of macro expansions, which has not been applied so far. This, together with the ability of inserting text to an arbitrary point of the program text, gives a better control over program editing. The main new macro features of CKit and examples to their use are listed in the table below:

New Feature:	Reference:
Macro calls with argument modifier	Section 8.1
Macros with associative power	Section 7.2, 7.3, 9
Macros with hidden arguments	Section 9
Lexical inheritance	Section 9

6 Building an Experiment

The author has used CKit for a variety of combinatorial applications. Among them are:

1. A problem of Erdos about sequences of natural numbers with all-distinct subset-sums.
2. Finding a false coin in a collection of coins [1] [2].
3. Finding maximal clique and optimal coloring of graphs.
4. Finding mixed (binary-ternary) codes [3],
5. Finding upper bounds on the size of ternary codes of length 7, minimum distance 4 [3] [7].

We shall demonstrate the operation of CKit through the highlights of a software, which was designed to find mixed binary-ternary codes with given parameters [3]. Some macros have been changed in this article for a more concise exposition.

A codeword of a mixed binary-ternary code is a sequence of digits $(b_1 \dots b_k t_1 \dots t_l)$ such that k and l are fixed, and $b_1, \dots, b_k \in \{0,1\}$; $t_1, \dots, t_l \in \{0,1,3\}$. Our goal is to find a set C of codewords as large as possible such that the Hamming distance between each two codewords is at least d. The Hamming distance between the two codewords $(b_1 \dots b_k t_1 \dots t_l)$ and $(b'_1 \dots b'_k t'_1 \dots t'_l)$ is defined as $|\{i \mid b_i \neq b'_i\}| + |\{j \mid t_j \neq t'_j\}|$. The numbers k, l, and d are the parameters of the mixed code. The maximal size code with these parameters is denoted by $N(k,l,d)$.

The software highlighted in subsequent sections uses a clique finder algorithm of Johnson and Applegate, and its goal to give lower bounds on $N(k,l,d)$. The problem about mixed codes was brought to the author's attention by Neil Sloane, who with coauthors in [3] extensively studied the problem. The author's contribution to their work is a slight improvement for parameters $k = 8$, $l = 1$, and $d = 3$ ($N(k,l,d)$ was improved from 50 to 48).

In the next two sections we build up a set of macros, taking the reader through new CKit features simultaneously. Once we define a macro, we use it in later paragraphs without warning.

7 Creating Basic Macros

Basic macros are the ones that are likely to be useful in all our programs. Among them are: program template macros, declaration macros, print macros, timers, data store/retrieve macros, data visualization macros. Representative examples of the first three groups are presented in this section.

7.1 Program Templates

CKit is a text processor, which has no knowledge of the syntax of the language of our object (in our case C). If we want CKit to insert declarations, function prototypes, typedefs into the appropriate places, we have to let the CKit compiler know where these places are in the program text. In general, we can mark places in text with `@identifier`, The following C-program template defines a macro that describes the structure of a simple C program:

```
PROG ::
# include <stdio.h>
@includes
@typedefs
@declarations
@prototypes
int main (int arge, char past *argv){ $

END_PROG ::  }
@procedures $
```

All our programs will use the `PROG` and `END_PROG` macros.

7.2 Declarations

Declarations of variables through CKit can be useful for two reasons. First, many programmers like to declare a variable where it is used first. The syntax of C (unlike e.g., the syntax of Java) does not allow for this. Secondly, it is often advantageous to keep track of the type of a variable in order to be able to write in-line procedures that can depend on the argument's type. The following macro declares a global integer, and then replaces the text of the macro call with the variable's name:

```
Int/name ::
name !? declarations +:
int name;$ name//type :: int$
name$
```

The above macro illustrates a number of features in CKit. The macro `Int` has one formal parameter: `name`. The macro text includes further CKit commands: the macro definition `name//type::int$`, the conditional inclusion command `name!?declarations+:int name;$ name//type::int$`, which contains the former as well as the insertion command `declarations+:int name;$`. We show the operation of this macro through a sample program, which relies on the

```
FOR/var/bnd :: for(Int(var)=0;
var < (bnd); var ++)$
```

macro, and prints integers from 0 to 10, and their cubes below them:

```
|>general.x
PROG
FOR(i,10) printf("%5d", i);
printf("\n")
FOR(i,10) printf("%5d", i*i*i)
END_PROG
```

Each call of `FOR` results in the expansion of the `Int` macro with parameter `i`. During the first call parameter `i` is unknown to CKit. Hence, the command `!?` (meaning include `text`, if `block` before the command operator is not defined) expands to `declarations +: int name;$ name//type :: int$` . When this text is interpolated and parsed, it has two effects: First, text `int name;` will be inserted to the place marked `@declarations`, secondly a macro with identifier `name`, and an associative argument `type` is defined. From this point on `i` and `i//type` will be defined (`name` has the empty string as its default value). When the `Int` macro is called again with argument `i`, then the conditional inclusion expands to the empty string, since `i` is already known. The only effect of this call is that a copy of `i` replaces the macro call.

7.3 Macros with Associative Power

Next we elaborate more on the concept of macros with associative power, taking another declaration macro as an example. The macro

```
Intvec/name/vlength ::
name !? declarations +:
int name[vlength],
name@@dim;$
name//type :: int$
name//length ::
vlength$$
```

contains two commands that declare macros with associative arguments, both with identifier `name`. These macros are not differentiated through their identifiers, but by their arguments. `name(type)` expands to `intvec`, and `name(length)` expands to `vlength` (where `name` and `vlength` depend on the parameters `Intvec` is called. For all practical purposes `name` behaves like an associative array, but it is more general. `name` with any other argument than `type` and `length` is defined as the empty string by default. If the macro `Intvec` included `name/any::I am name$`, then calling `name` with a default argument would expand to `I am name`.

Macro `Vectorcopy` in the example below creates an array (`vect2`), which is identical to another one (`vect1`). The second macro in the same example uses `name(type)` to put the appropriate print instruction into the program text. For the later we assume that macros `Prinint` and `Printintvec` to print integers and arrays of integers are already defined.

```
Vectorcopy/vect1/vect2 ::
Intvec(vect2,vect1(length))
FOR(copyvar, vect1@@dim)
vect2[copyvar] =
vect1[copyvar];$

Print/name ::
name//type=int ??
Printint(name)$
name//type=intvec ??
Printintvec(name)$$
```

7.4 Higher Level Templates:

The possibility of building a hierarchy of templates is particular to those preprocessors with inclusive macros.

```
Simpledecl/Declmacro/type ::
Declmacro/name ::
name !? declarations +:
Type name;$ name//type ::
Type$ name$$
```

expands to declaration macros similar to the one, which declares an integer in the beginning of this section. E.g `Simpledecl(Real,real)` expands to the definition of the macro `Real`, which declares reals, or `Simpledecl(Register,register)` expands to the definition of `Register` which declares registers. The above example shows how to build macros to reach the level of abstraction in program organization that saves the most code.

8 Creating a Macro Platform for an Experiment

In this section we give a representative cross section of those *specific* macros that the author relied on, when searching for mixed codes of given parameters. These macros are contained in file `codes23.x`. Since we need the general macros as well, the file `codes23.x` starts with

```
|> general.x
```

Assume that macros `DIM2` `DIM3` and `DIST` define the parameters k, l, and d of the code. Each codeword will be represented by a 4 byte integer such that the first k bit represents the binary digits of the code and the following $2l$ bits represent the ternary digits (two bits for each digit). First we build formulas to express the distance between two codewords:

```
EX1/i ::  (((i) >> Int(kk)) % 2)$
EX2/i ::  (((i) >> Int(kk)) % 4)$

gexpr/i/j/gsum ::  Int(gsum)=0;
FOR(kk,DIM2) {
   if(EX1(i) != EX1(j) gsum++; }
for(kk=DIM2;kk<DIM2+2*DIM3;kk+=2) {
   if(EX2(i) != EX2(j) gsum++; }$
```

Higher level macros use this notion of distance. Some of them are:

```
MAKESET/set/expressions/formula ::...
MAKESUBSET/set1/set2/expressions/formula
::...
EACHINSET/set/property/indicator ::...
INSET/set/indicator ::...
MAXCODE/inputset/outputset ::...
FAR ::...
ATLEAST/point/distance/indicator ::...
```

What is worthwhile to notice about them is how reserved they are about the use of parameters. `MAXCODE` takes a subset of all strings with parameters k and l, $S_{k,l}$, returns a maximal code in them (second parameter). This macro is the interface with the clique finder algorithm of Johnson and Applegate. Macros do not have a return value. We often take the last parameter of the macro as our return value. If this value is Boolean, we call it an "indicator." Macro `MAKESET` creates a subset of $S_{k,l}$ by inputing a list of several expressions in its second argument, and comprising the indicators of these expressions into one Boolean formula in the second. To implement this we do not have to go beyond the capabilities of the commands that have been discussed so far. In the next subsection we discuss a feature of CKit which comes handy, when dealing with macros that expect names of expressions in their argument.

8.1 Argument Modifiers in Macro Calls

Let us have the following macro that assigns the first k values of an expression to an array:

```
ASSIGN/vect/k/expr ::  FOR(i,k) vect[i]
= expr(i);$
```

If Expr is a macro of an expression with only one argument, then the above macro can be called with Expr as its last argument. But if we have an expression Expr(x,y), which has two parameters, and we want to assign Expr(i,i) to vect[i], first we have to define a new expression, which has only one parameter, by

```
Expr1/x ::  Expr(x,x)$
```

and then use Expr1 as the last argument of ASSIGN. This solution is too cumbersome if Expr1 is used only once. In CKit it is possible to interpret the argument list of a macro by putting an argument modifier enclosed in brackets before the the actual arguments of a macro call. The macro call

```
Expr[\1,\1](i)
```

is equivalent to Expr(i,i). We may write ASSIGN(vect,100,Expr[\1,\1]) to give the value of Expr(i,i) to vect[i] for i< 100.

In general, if the number of arguments of a macro mymacro is k in the macro definition, then

mymacro[text_1,text_2,...,text_k](arg_1,..., arg_l)

is a valid call, where k is arbitrary, and text_1, text_2, etc. may contain references to arg_1, arg_2, etc. This call is equivalent to

mymacro(text_1',text_2',...,text_k'),

where text_i' is obtained from text_i by replacing every occurrence of \j with arg_j for $1 \leq j \leq l$. If l is 0, the parentheses are empty, but they should not be left out.

9 Building Schemes

Schemes are program constructs, that once are built, can be used in more than one contexts. Procedures are such constructs in procedural languages, and classes in object oriented languages. CKit schemes are *macro constructs*. Let us take a local optimization scheme for finding mixed codes as an example.

The input for the scheme is a mixed code (set), which is not optimal. In order to improve on the code we delete some of the code words (SHRINK), and then we find a maximum set of codewords, which can be added to the smaller set such that the union of the old (smaller) and new (newcode) set again forms a code. We repeat this process bnd times. The scheme also has an input parameter param, which controls the size of the ball from which we leave out the codewords in the shrinking process. The CKit description of this scheme is as follows:

```
LOCOPT/set/bnd/param ::
FOR(locopt, bnd) {
    RANDOM(locopt,rnd)
    SHRINK(rnd)
    EXPAND
}

//RANDOM/locopt/rnd ::
    while(!ind) {
            Int(rnd) = rand()
            INSET(rnd,all,ind)
            if(ind) break;
    }$

//SHRINK/rnd ::
MAKESUBSET(smaller,set, ATLEAST(rnd,
param,indic), indic)$

//EXPAND ::
MAKESUBSET(farset,basic,EACHINSET(
smaller,FAR,ind),ind)
MAXCODE(farset,newcode) MAKESET(set,
INSET(smaller,ind1)
INSET(newcode,ind2), ind1|ind2)$$
```

What we need to notice about macros RANDOM, SHRINK and EXPAND is that their definitions in the definition of LOCOPT start with the // symbol. This symbol in CKit means lexical association. Macros like this constitute the *hidden parameters* of the encapsulating macro, and cannot be directly called from outside of the macro that encapsulates them. CKit uses the associative power of the encapsulating macro to implement hidden parameters. It is possible to call the macro EXPAND from outside LOCOPT with LOCOPT(EXPAND).

Lexical Inheritance

The use of hidden parameters becomes apparent with the introduction of *lexical inheritance*.

```
NEWLOCOPT <:  LOCOPT
```

declares that CKit accepts NEWLOCOPT in any textual context, where LOCOPT is accepted. The hid-

den parameters of NEWLOCOPT are the same as those of LOCOPT, but new ones can be added, and the old ones can be redefined. For instance

```
NEWLOCOPT <:  LOCOPT ::
//SHRINK/rnd ::
if(locopt<100) {
SHRINK1(rnd) } else {
SHRINK2(rnd) }
//SHRINK1/rnd ::
MAKESUBSET(smaller,set,ATLEAST(rnd,5,
indic), indic)$
//SHRINK2/rnd ::
MAKESUBSET(smaller,set,ATLEAST(rnd,
param, indic), indic)$
```

redefines NEWLOCOPT(SHRINK), and creates two new hidden parameters SHRINK1 and SHRINK2 for NEWLOCOPT. With the above CKit text segment NEWLOCOPT expands in the same way as LOCOPT, but SHRINK expands in the newly defined manner.

10 Alternative Program Constructs

CKit is prepared to help the programmer through his/her experiments before he/she writes large chunks of well-organized codes. The experimental aspect of CKit is even more pronounced in providing a framework for alternative program constructs. Taking the use of macros we built, we can attempt to find a maximal sized mixed code with parameters 8, 1, and 3 with:

```
|> codes23.x
DIM1 ::  8$
DIM2 ::  1$
DIST ::  3$

ALTERNATIVE1 ::
MAKESET(set,,)
NEWLOCOPT(set,400,4)
SAVE(set,result.8.1.3)$

ALTERNATIVE2 ::
RETRIEVE(set,result.8.1.3)
LOCOPT(set,400,6)
SAVE(set,result.8.1.3)$

PROG
ALTERNATIVE1
END_PROG
```

The above code contains two programs worth of macros, but CKit compiles only the one that is in the definition of ALTERNATIVE1. If later we need to run the second alternative, we can change the last three lines of the above program to:

```
PROG
ALTERNATIVE2
END_PROG
```

If we encapsulate all our experiments in a macro, we can easily put several experiments into a single file. CKit provides means to automate the comparison of the outcome of these experiments. This feature is new to CKit, and we do not go into the details of it.

Acknowledgments: I am grateful to Noga Alon and Neil Sloane for providing me with test instances from their research. The engine of my coding theory programs was a clique finder program developed by David Johnson and David Applegate, and I am grateful to them for letting me use it. Finally, I would like to thank to Kiem-Phong Vo for his professional advice on my software.

References

[1] N. Alon, D. N. Kozlov and V. H. Vu, The geometry of coin-weighing problems, Proc. 37^{th} IEEE FOCS, IEEE (1996), 524-532

[2] N. Alon and V. H. Vu, Anti-Hadamard matrices, coin weighing, threshold gates and indecomposable hypergraphs, J. Combinatorial Theory, Ser. A, in press.

[3] A.E. Brouwer, Heikki O. Hamalainen, Patric R. J. Ostergard, N. J. A. Sloane, Bounds on Mixed Binary/Ternary Codes, accepted to IEEE transactions on Information Theory

[4] Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language, Prentice-Hall Software Series

[5] m4.info, http://www.inf.tu-dresden.de/info/m4.info

[6] Pre-Processors, http://www.qpsf.edu.au/software/pre/pre.html

[7] Mario Szegedy, The Bipartite Graph Method: A New Way to Obtain Upper Bounds On Non-Linear Codes ATT Labs Research, unpublished

[8] Mario Szegedy, Gizella Stefan Szegedy. Manual for the CKit Program Generator Kit, ATT Labs Research, Tech Report