

COMPILER ALGORITHM LANGUAGE (CAL): AN INTERPRETER AND COMPILER

Abhinav Bhatele, Shubham Satyarth, Sanjeev K Aggarwal
Department of Computer Science and Engineering
IIT Kanpur - 208016
India

ABSTRACT: We have designed a Compiler Algorithm Language (CAL) to provide compiler writers with a language which is close to actual algorithmic notation. In this work, we have developed an interpreter and debugger for CAL which can be used by researchers for algorithm testing. We also provide a compiler that can translate CAL programs to C code which can be plugged elsewhere. We have provided a Graphical User Interface to make it convenient for the user to use the interpreter and compiler. Another feature of our system is web-enabling of the entire project so that a remote user would not have to download the entire code. Altogether, we have attempted to save the compiler writers from the trouble of writing lengthy programs for their algorithms.

KEY WORDS: Compiler Algorithms, Interpreter, Compiler, Syntax directed evaluation, Virtual Machine

1 Introduction

An important part of compiler design is to develop faster and more efficient algorithms for parsing, optimization, translation and code generation. Researchers continue to invent new and better techniques for solving these compiler design problems. Compiler writers express these algorithms in a pseudo code which is easy to understand.

Optimization and code generation phases are the heart of advanced compiler design. This requires the compiler to do an analysis of the programs which are to be compiled. The major analysis techniques include - (1) control flow analysis, (2) data flow analysis, (3) data dependence analysis, and (4) alias analysis. Such analysis again requires writing algorithms for a given problem [1].

Algorithms for such analysis are very complex and involve data structures like sets, tuples and records. Converting these algorithms to programs becomes a difficult task as most high level languages do not support data structures and control flow constructs required in these languages. One would like to quickly test the algorithms without getting into the trouble of writing the equivalent high level language program.

We have designed a Compiler Algorithm Language (CAL) which is very close to the notation used for writing algorithms. This relieves the researcher of the task of expressing their algorithms in high level programming

languages, We provide an interpreter for this language so that algorithms can be quickly checked and debugged [2]. We also provide a debugger that helps in analysis of the algorithms. Further, compiler writers want to plug-in their new algorithms into already existing systems. For this purpose we provide a compiler which converts an algorithm into equivalent C code. We have also web-enabled our tools. This saves users from installing the interpreter/debugger/compiler on their own desktops. A user can develop, debug and convert an algorithm expressed in CAL to C code by using the web enabled set of tools.

1.1 Related Work

Stark algorithm language was developed as a language for showing provable algorithms by Richard Stark in 1968. This was called an instructional algorithmic language [3]. INTERCAL was developed by Woods and Lyon in 1972 [4]. It has various implementations like INTERCAL-72, C-INTERCAL etc [3]. Another instructional language was developed in 1974 at the Bowling Green State University. This was a hybrid of FORTRAN, PL/I and Algol 68 and was called LINUS (Language for INstructional USE) [5].

SETL (for Set Language) is the language which comes closest to CAL. It was developed by Schwartz and Dewar [6]. SETL allows a large variety of programming problem to be solved in an efficient manner. SETL is a very high-level language whose syntax and semantics are based on the standard set theoretical dictions of mathematics.

2 Our System

THE LANGUAGE: Our language is called *Compiler Algorithm Language* or CAL in short. It is similar to ICAN (Informal Compiler Algorithm Notation) proposed in [1]. It derives features from many languages such as C and Pascal and extends them with natural notations for objects like sets, tuples, sequences, functions and arrays.

A CAL program consists of a series of type definitions, followed by a series of variable declarations, procedure declarations and an optional main program. The syntax is so designed that every compound statement includes an ending delimiter. As a result, separators are not needed between statements. Tabs, comments and blanks

are considered as white spaces. The generic simple types are *boolean*, *integer*, *real*, and *character*. It supports most of the control flow statements like *if*, *case*, *while*, and *for*.

INTERPRETER: The interpreter saves the user the trouble of converting the algorithm to a high level language for checking its correctness. The interpreter does a line by line interpretation of the program and gives the desired output. The front end of the interpreter consists of a lexical and syntax analyzer. The back end of the interpreter also called the executor[2] does a syntax directed evaluation of the program.

COMPILER: It converts an algorithm written in CAL to a piece of C code. It is possible to plug-in this C code into a larger program. The compiler uses the front end of the interpreter. The back end does the translation to C code.

GUI and WEB ENABLING: The most effective interface for a user is a graphical one which can encompass the interpreter and the compiler as well. The GUI has an editor and a prompt for the interpreter. There are options for debugging and compilation.

A user may find it cumbersome to download the compiler code and install it. For such users we have a web enabled interface. A user will be able to compile his code through a script running on a http server.

3 Features of Compiler Algorithm Language

A CAL program consists of a series of type definitions, followed by a series of variable declarations, procedure declarations and an optional main program. A typical program looks like:

```
<type_definitions>
<variable_declarations>
<procedure_declarations>
<optional_main_program>
```

A type definition consists of a type name followed by an equals sign and a type expression, such as

```
inset = set of integer
```

3.1 Data Types

The language has four generic simple data types: 1. *integer*, 2. *real*, 3. *character*, and 4. *boolean*. The constructed data types are 1. *array*, 2. *set*, and 3. *function*.

A variable declaration consists of the name of the variable, followed by an optional initialization, followed by a colon and the variable's type, e.g.,

```
is := 1,3,3: intset
```

3.2 Operators

An expression is either a constant, a variable, *nil*, a unary operator followed by an expression or two expressions

separated by an operator. The operands and operators should be of compatible types. Here, we provide a list of operators in the language.

3.2.1 Unary Operators for basic data types

!	Negation of booleans
-	Negation of integers and reals

3.2.2 Basic Operators for basic data types

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder
~	Exponentiation
&	And
	Or
!	Not

3.2.3 Unary Operators for composite data types

~	Gives an arbitrary element of a set
---	-------------------------------------

3.2.4 Binary Operators for composite data types

UNION	Union
INTERSECTION	Intersection
DIFF	Difference
INSET	Belongs to the set
NOTINSET	Does not belong to the set
->	Specifies the mapping of the function

3.2.5 Conditional Operators

=	Equal
<>	Not Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The language also provides two quantifiers: 1. *FORALL* and 2. *EXISTS*.

3.3 Simple Statements

The language provides basic statements like assignment, procedure call, return, goto and I/O statements.

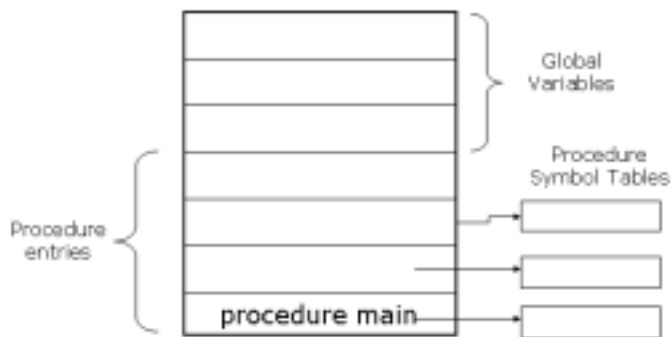


Figure 1. Symbol Table

3.4 Control flow statements

The language supports all basic control flow statements like:

- if-then-else
- switch-case
- for loop
- while loop
- repeat-until loop

3.5 Procedures

A procedure declaration consists of the procedure's name, its parameter list, an optional return type, followed by its parameter declarations and the body. The procedure body starts with the keyword `begin` and finishes with the keyword `end`. A procedure call follows the same pattern as in other high level languages.

3.6 Keywords

The various keywords in CAL are:

- | | | | | |
|-----------|---------|---------|--------|---------|
| array | begin | boolean | by | case |
| character | default | do | each | elif |
| else | end | esac | false | fi |
| for | goto | if | in | inout |
| integer | nil | od | of | out |
| procedure | real | repeat | return | returns |
| set | to | true | until | where |
| while | | | | |

Some additional keywords are:

- | | | | |
|-------|--------------|--------|----------|
| DIFF | EXISTS | FORALL | NOTINSET |
| NULL | INSET | OUTPUT | UNION |
| INPUT | INTERSECTION | | |

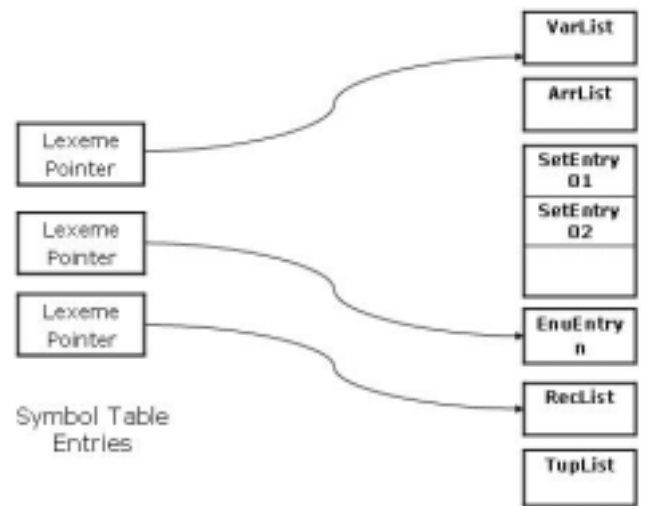


Figure 2. Symbol Table

4 Interpreter

We have provided an interpreter for this language. It is a one pass interpreter that does syntax directed evaluation. The concept of the interpreter is similar to a virtual machine that simulates a processor and memory. For procedure calls and loops, the virtual machine uses different objects of the executor.

4.1 Symbol Table

We use a symbol table which contains the references to the location of the actual values. Each global variable encountered is directly stored in the symbol table while each procedure has an entry in the symbol table with a pointer to a file which stores the actual code. This is done to avoid a second pass on the procedures. Symbol table for a procedure is initialized as soon as the procedure is encountered. When there is a procedure call in `main()`, the code for that procedure is again parsed and this time executed.

There are two levels of the symbol table. Outer level entries just have the lexeme and a pointer to the location where the actual value is stored. There are different linked lists for different data types. Each entry in the symbol table points to an entry in one of these linked lists.

4.2 Implementation

We have used Java as our implementation language. This has facilitated the use of rich set of Java collection framework for data storage. The use of Java has also simplified type handling by using Object as the universal type and then typecasting it to the required type. Use of Java also aids GUI development and web-enabling. We have used `jflex`[7] and `cup`[8] for lexical and syntax analysis respectively.

The front end of the interpreter consists of the lexical analyzer (lexer.java) and syntax analyzer (parser.java). The back end or the executor consists of the Symbol Table cum Scratch Memory (SymbolTable.java), Virtual Machine (VirtualMachine.java) and The Interpreter (MainProg.java).

5 Compiler

The compiler converts a CAL program into C code. It may not be a complete program but a set of functions which can be used in another program. The compiler has also been developed using the grammar built for the interpreter.

The compiler converts the data types like char, int, real into basic data types in C and the rest into structures like linked lists, arrays etc. We have created a header files setheader.h for the data structures required to initialize a complex data types and various functions to do operations on these data structures.

Each function is converted into a C function with the same name while the main() function is converted into a function called the_main(). It is this function together with the previously declared functions which can be plugged into some other C program.

5.1 Implementation

The front end of the compiler consists of the lexical analyzer (lexer.java) and syntax analyzer (translator.java). The back end uses files like editor.java and FileCreator.java to output the file with the C program. The file generated is called 'prog.c'.

6 Easy-to-use Interfaces

The idea behind interfaces like Web and GUI is to save the user of our interpreter and translator from the trouble of downloading and installing the system.

6.1 Web-enabling

We have used the apache tomcat server to run our interpreter and translator on the server side. This receives requests (ServletUtilities.java) from servlets when a remote user posts data on the form.

The data is passed on to the interpreter and/or the translator which then produces the output (ShowParameters.java) which is passed back to the client side.

6.2 Graphical User Interface

The GUI has been made in java with the help of netbeans and standard swing and awt api's. The various functionalities available with the GUI (MainPanel.java) are:

1. File Handling - New, Open, Save, Save As
2. File Editing - Copy, Cut, Paste, Delete, Find, Find Next

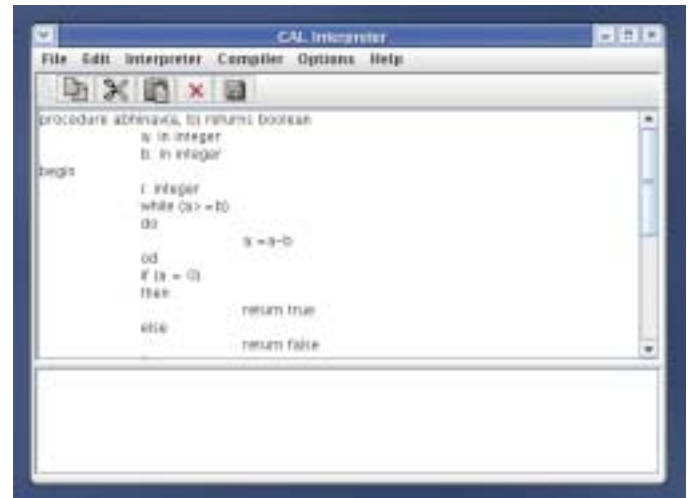


Figure 3. GUI for CAL Interpreter and Compiler

and Replace

3. Interpreter - Execute
4. Compiler - Translate
5. Other Options - Go to line, Select All, Set Background Color
6. Help - Help Contents and About CAL Interpreter

The GUI (Figure 3) has an editor pane for the file and an output box where one can see the output of the interpreter or the translated C program.

7 Some Illustrative Examples

The CAL code for finding closure of a set of items (used by a parser) looks like:

```
G: set of Productions
procedure closure(I) returns set of Items
  I: in set of Items;
begin
  J: set of Items;
  X: Items;
  Y: Production;
  J := I;
  for each X INSET J
  do
    for each Y INSET G
    do
      if .Y NOTINSET J
      then
        J := J + .Y
      fi
    od
  od
  return J
end
```

The corresponding C code for the above algorithm is rather involved and is given below:

```

struct Items
{
    //Code for Items
    Items *next;
}
struct Production
{
    //Code for Production
    Productions *next;
}
struct setofItems
{
    Items head;
}
struct setofProductions
{
    Production head;
}*G;
// Initialize G

void setofItems closure(setofItems I)
{
    setofItems *J = I;
    setofProductions *K;
    K=G;
    Items *X;
    Items *X1;
    Production *Y;
    X = J->head;
    Y = G->head
    while(X!=NULL)
    {
        while(Y!=NULL)
        {
            Productions Z = K->head;
            while(K!=NULL)
            {
                if(.Y==Z)
                {
                    X1 = Z;
                    X1->next = J->head;
                    J->head = X1;
                }
                Z=Z->next;
            }
            Y=Y->next;
        }
        X=X->next;
    }
    return J;
}

```

Another example which illustrates input and output:

```

procedure exam(x,y) returns boolean
    x, y: out integer;
begin
    is: in intset;
    INPUT is;
    tv := true: boolean;
    z: integer;
    for each z INSET is (z > 0) do
        if x = z then
            return tv
        fi
    od;
    return y INSET is
end

```

The corresponding C code is:

```

boolean exam(int x, int y)
{
    int n;
    printf("Enter the size of the set");
    scanf("%d",n);
    int is[n];
    printf("Enter the set");
    for(int i=0;i<n;i++)
        scanf("%d",is[i]);
    boolean tv = true;
    for(int i=0;i<n;i++)
    {
        if(is[i]==x)
            return tv;
    }
    for(int i=0;i<n;i++)
    {
        if(is[i]==y)
            return true;
    }
    return false;
}

```

8 Conclusion

We faced several challenges while doing this work. Due to one pass nature of the interpreter, handling of loops and procedures in syntax directed evaluation was difficult. Special handling was required for loops and procedures.

The size of the code for various phases is quite large. The parser for the interpreter runs into 7000 lines whereas for the translator runs into 5000 lines. The code for the GUI is also about 700 lines. The lexical analyzer takes more than 30 tokens and the grammar for our language has above 200 productions. We have tested our interpreter and compiler for a large number of diverse variety of programs.

We have tested all the implemented data types and loops and other structures.

The tool set was given to a large number of programmers in the compiler community. They reported that CAL had a rich feature set and the system was very useful for expressing algorithms which used sets and set operations.

References

- [1] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, Inc., 1997
- [2] Ronald Mak, *Writing Compilers and Interpreters*, John Wiley and Sons, Inc., 1996
- [3] Richard Stark, *A Language for Algorithms*, The Computer Journal, pp40-44, 14(1), 1971
- [4] The INTERCAL Programming Language REVISED REFERENCE MANUAL, Woods, Donald R. and Lyon, James M. Revised by Howell, Louis and Raymond, Eric S. Available at <http://catb.org/esr/intercal/intercal.ps.gz>
- [5] John D. Woolley and Leland R. Miller, *LINUS: A structured language for instructional use* Proceedings of the 4th SIGCSE symposium on Computer science education, pp125-128, 1974.
- [6] Schwartz, Jacob T., Dewar, R.B.K., Dubinsky, E., and Schonberg, E., "Programming With Sets: An Introduction to SETL", 1986. ISBN 0-387-96399-5.
- [7] <http://jflex.de/index.html>
- [8] <http://www.cs.princeton.edu/appel/modern/java/CUP/>