MM Research Preprints, 99–112
No. 15, April 1997. Beijing

On the Interactive and Programming Environment for STAR[1)]

Yu Ju, Jun Liu, Zhuojun Liu
Institute of System Science
Academia Sinica, Beijing 100080, P. R. China
Email: {yju,liujun,zliu}@mmrc.iss.ac.cn

Abstract. The progress of computer algebra has greatly stimulated the development of mathematical software such as **maple** and **reduce**. These software systems prove to be powerful and helpful especially for symbolic computations. However, people still need much perfect software products to handle all kinds of applications. This is even true for considering the applications only related to algebraic computations. Since the appearance of Wu's method from the late of 1970s, it has taken more attention from the field of computer algebra and automated reasoning. Although there exist some packages implementing Wu's method, there is no complete software system that not only implements Wu's method but also provides some other friendly and powerful features. Under the circumstances, we begin to develop the software system STAR (Small Tool for Algebraic Research) from 1995. STAR will implement Wu's method and provides facilities to manipulate polynomials. Besides this, other features will be added to it. In this paper, we describe STAR-PSE, an interactive and programming environment for STAR. It was implemented by C++. Unix utilities **lex** and **yacc** are also used to establish this environment.

1. Introduction

Computer algebra is that part of computer science which designs, analyzes, implements and applies algebraic algorithm [**?**]. Computer algebra was also called symbolic computation, since it manipulates mathematical expression in a symbolic rather than numerical way, such as one does algebra with pencil and paper. Furthermore, algebraic objects can be represented exactly in the memory of a computer, so that algebraic computations can be performed without loss of precision. Since 1960, many software systems devoted to various class of symbolic computations have been developed. Major representatives of them are **macsyma**, **reduce** and **maple**. Although these sorts of systems are continuously increasing in efficiency and capacity, much more powerful software tools are still needed. It is well known polynomial

[1)]This work was supported by the Climbing Project Fundation of P.R. China.

computation plays an important role in many symbolic manipulation systems. For example, the projects of POSSO and FRISCO in Europe, based on Grobner Basis method, are trying to create highly efficient, versatile polynomial solvers. On the other hand, Wu's method(also known as the characteristic set method) [**?**, **?**], since it mainly considers the zero structure of polynomial equations, provides a powerful tool to deal with polynomials, especially can be used to establish our own polynomial processor STAR [**?**]. STAR means Small Tool for Algebraic Research, will be a Maple-like system. The capacity of the characteristic set computation will be put into its kernel.

In MMRC(Mathematics-Mechanization Research Center), STAR is a continuous effort for mathematical software developing related to the previous work of CSET, SACCS, WSOLVE and GPROVER [**?**].

The developing of STAR is divided into several stages and the whole system consists of some subsystems such as the number system of STAR, the engine part of STAR and so on.

In this paper, we describe the implementation of the interactive and programming environment for STAR. These features are part of ones in a problem solving environment(PSE). Instead of having a real PSE, we implement a so-called STAR-PSE, since not all features of PSE are included in our work at this point. However, STAR-PSE will be elaborated into a true PSE in the future. It is also true that STAR will continuously increase its efficiency and capacity, and become a comperehensive tool for polynomial processing.

STAR-PSE was made by C++, Unix utilities **lex** and **yacc** are also used for its establishing.

The organization of this paper is as follows: Next section, we describe the features of PSE and illustrade STAR-PSE. In section 3, we discuss techniques and methods used in the implementation of STAR-PSE. Section 4 gives the outline of how the STAR-PSE is implemented. In the last section, we briefly discuss the future work.

2. PSE's features and SATR-PSE

PSE is the abbreviation of Problem Solving Environments. It refers to the software environment that assists in the development, solution and analysis of some problems [**?**].

A key feature of PSE is its ability to converse with the scientist in one's own terms. If one views computer algebra system as a mathematical PSE, then it exhibits good abilities of conversing in mathematical terms.

Also, PSE must provide users a programming capacity. This can save a lot of work for people. For example, if you want to add the integers from 1 to 100, would you really like to type $1 + 2 + \cdots + 100$? Of course you can get the result by doing so. But why not chose a for-statement to do the work if the PSE provides this facility? It is more efficient and can avoid some mistakes you may make when typing $1 + 2 + \cdots + 100$.

Another feature of PSE is that it can assist the user with the selection of method for solving certain problems. A modern computational scientist attempting to use a software library for a simple operation such as matrix factorization would be overwhelmed by the choices that are available. Depending on the properties of the matrix, one needs to use a particular routine that best applies to that matrix. It is always possible to use a general method that works for all matrices. However, this amounts to a waste of computing resources that could easily have been avoided if the software could automatically select the right method to use.

It is not sufficient for a PSE to address only one dimension of the problem at hand. For example, in the parallel solution of PDEs, the PSE provides tools for automatically selecting an appropriate machine from the set of available parallel processors, maps the problem on to the selected machine, gathers the solution and, finally, assists in performance evaluation.

Other property of PSE including its behavior as the manager of computing resources related to the problem, it is important that the PSE manages issues such as machine allocation, mapping, and deallocation. Distribution of the tasks within a network of computing agent is another task that the PSE must manage. If the environment has multimedia capacity, then the PSE must manage these aspects as well in order to provide a comprehensive and cohesive user interface that exploits all available media in visualizing the problem and its solution.

2.1. STAR-PSE

As an important part of STAR project, STAR-PSE can provide a mathematical problem solving environment mainly based on characteristic set method. At present, it emphasizes on the first two features of PSE mentioned above.

STAR-PSE is interactive. This means that it can converse with users. It recognizes what is the input from users and asks STAR to complete the task if it can. Otherwise, it gives a message to tell the user that STAR can't do it or report some errors found in the stream of input. For example, under the control of STAR-PSE, give the following input,

```
>sum:= x^2+3*x+y+4;
```

here ">" is the prompt printed out by the system. It asks for your input. This example is an assignment statement. ";" means the end of a statement. After receiving the

input as above, the system will assign the value of a polynomial $x^2 + 3 * x + y + 4$ to the variable **sum**. Then another prompt ">" asks you to input further statement and so on. We can do more as following:

```
>sum:=0;
>for i from 1 to 10
 do
    sum:=sum+i;
 od;
```

This is a for-statement. After it is executed, the value of **sum** can be got by

```
>sum;
```

For our case, the output is 55. This explains the interactive feature of STAR-PSE. Besides that, people are much interested in the capacity to support programming. Here is an example:

```
>Addsum:=proc(num) local i,summary;
         summary:=0;
         for i from 1 to num
         do
            summary:=summary+i;
         od;
         return(summary);
 endp;
```

After a procedure Addsum() was defined as above, it can be used to summerize from 1 to **num**, where **num** is a parameter to be given when the procedure is called. We call the procedure like

```
>sum:=Addsum(100);

>sum;
```

then get output 5050. A further example shows the ability of STAR to do pseudo-remaider through STAR-PSE, which is an important manipulation in characteristic set computation:

```
>a:=x^4+1;
>b:=c*x^2+1;
>r:=prem(a,b,x);
>r;
              c(c^2+1)
```

2.2. The key working steps of STAR-PSE

In order to see how STAR-PSE provides the interactive and programming environment, we should understand what are the key working steps of STAR-PSE. They are:

lexical analysis

In this step, STAR-PSE must extract words from the input and classify them into the following different types:

key words
such as **for, from, to, do, od, proc, local, endp,** etc.

constant & variables
such as 150, 97, sum, i, num, etc.

separator
such as ":=" , ";" , etc.

syntax analysis

In this step, SATR-PSE decides the grammar structure of input to see whether it is a correct or not based on lexical analysis. For example, a for-statement is like:

```
for variable_name from expression to expression
do
  statement-list;
od;
```

It is the work of lexical analysis to recognize from the input stream the words such as **for, from, to**, etc. Meanwhile the work of syntax analysis is to check whether the words is organized according to the given format.

interpret and execute

If a input has passed the examination of the two steps above, then in this step the sentence will be interpreted and executed which needs to call other part of STAR.

3. Techniques and Methods Used for Implementing STAR-PSE

STAR-PSE is programmed by C++, an Object Oriented Programming(OOP) language. The central idea of OOP is to build programs using software objects. An object can be considered as a self-contained computing entity with its own data and procedures. On modern workstations, for example, windows, menus and file folders

are usually represented by software objects, which can be applied to many kinds of programs. Key OOP concepts include data abstraction, encapsulation, information hiding, inheritance and polymorphism. OOP offers such main advantages as simplicity, modularity, modifiability, extensibility, flexibility, maintainability and reusability [**?**]. OOP is a really powerful technique for developing STAR/STAR-PSE.

In section 2, we knew lexical analysis and syntax analysis are necessary for STAR-PSE to realize its programming feature. Usually, the analysis of an input stream can be split into two phases, lexical analysis and syntax analysis, which can simplify the overall design. In general, it is much complicated to implement them. Fortunately, we have found useful utilities **lex** and **yacc** from Unix to reduce our programming burden. Instead of making complicated programs, we only need to prepare files lex.l and yacc.y to define the function of STAR-PSE in using high level problem-oriented specification. Then run the utilites **lex** and **yacc** taken lex.l and yacc.y as the input files to produce the C++ programs yylex and yyparse respectively. These two programs would be integrated into STAR-PSE. Figure 1 gives an explanation to usage of **lex** and **yacc**.

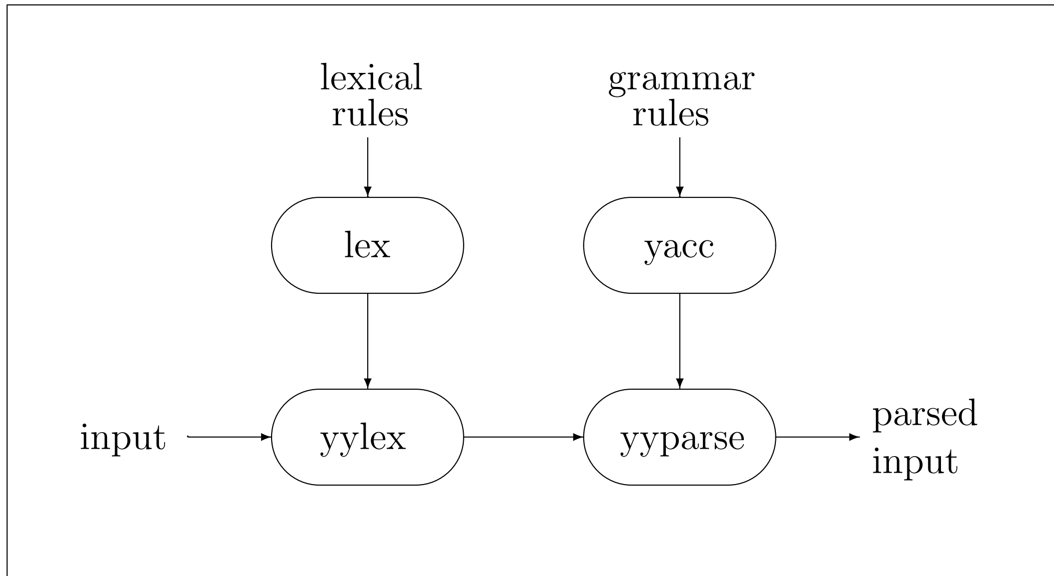

Fig. 1. lex with yacc

According to Figure 1, in order to establish STAR-PSE, the most important work for us to do is to write lexical rules into lex.l and grammar rules into yacc.y, which need the understanding of regular expression, BNF and grammar [**?**].

3.1. lex.l and yylex

As a part of STAR-PSE, yylex is generated from lex.l by **lex**. As a lexical

analyzer, yylex often works with syntax analyzer together. Below is a section of source specification in lex.l

```
for     {do_nothing++;return(FOR);}
from    {return(FROM);}
to      {return(TO);}


[a-zA-Z_][a-zA-Z0-9_]*
            {
                slen=strlen(yytext)+1;
                yylval.name=new char[slen];
                if(yylval.name!=NULL)
                {  strcpy(yylval.name, yytext);
                   return(IDENT);
                }
                else
                   error(NEW);
            }
```

The left hand side of the rules is described by regular expressions. The right hand side of the rules is the program section that will be executed on recognizing the pattern specified on the left.

3.2. yacc.y and yyparse

Similarly, yyparse is generated from yacc.y by **yacc**. Its duty is to do syntax analysis according to the description in yacc.y. A fragment of a grammar specification is explained as following

```
% start list                            --------(1)
% union{ int pos;                       --------(2)
         char* name;
         class Poly* poly_type_ptr;

         /* other class member */
       }
% token <pos> FOR FROM TO               ---------(3)
% token <name> IDENT                    ---------(4)
% type  <poly_type_ptr> poly;           ---------(5)

/* other specification */
```

```
poly:term                                  ---------(6)
     {  /* actions to do */ };
    |'-' term
     {  /* actions to do */ };
    | poly '+' term
     {  /* actions to do */ };
    | poly '-' term
     {  /* actions to do */ };
```

The first line specifies the single distinguished non-terminal symbols S of the grammar for the programming language. In this case, S=*list* ; Line 3 defines FOR, FROM, and TO to be tokens. Line 4 defines IDENT to be token with a value of type *name* , a pointer to character. These tokens are the terminals of the grammar. Line 5 defines a non-terminal *poly* with a value of type `poly_type_ptr` , a pointer to a class of *Poly* . Line 6 is a BNF expression for a grammar rule, which describes what the structure of a polynomial should be like.

4. Implementation of STAR-PSE

In fact, section 3 has partially discussed how to implement STAR-PSE. Recall section 2, after finishing lexical and syntax analysis, STAR-PSE needs to interpret the meaning and make an execution accordingly. In this section, we briefly discuss how to do it. For more detail discussion, the readers should refer to [**?**].

4.1. Variable Table

We need a variable table to record the value of variables, which is defined as

```
Var  VarTab[VarTabLen]
```

where the type of Var is defined as:

```
struct{
        char* name;
        int ifdefine;
         /* ifdefine==0 <==> val_ptr==NULL
            ifdefine==1 <==> val_ptr points
            to the value */
        class Poly* val_ptr;
      }
```

Variables that appear in if-statement, for-statement, while-statement or procedure are called inside-variable. Otherwise, they are called outside-variable. These two kinds of variable are treated in a quite different way. On receiving an outside-variable that doesn't appear in the left hand side of an assign-statement, STAR-PSE will look up the variable table to see whether there's an entry for it. If there's not, then STAR-PSE make an entry for it and mark the variable as undefined. If there's an entry for the variable already and it is marked as defined, then the variable is substituted by the value stored in the variable table. This is called evaluation of variable. This process is continued until the variable is substituted by an expression in which all the variables are marked undefined. For example:

```
>c:=b;
>b:=a;
>x:=y+c;
```

On receiving the variable c in $y+c$, c will be substituted by b , and then b is substituted by a . So, the result of x is $y+a$. However, on receiving an inside-variable, such as variable b in the if-statement:

```
>if ( a>3 ) then
   x:=b;
 fi;
```

Only after we get "fi", can we determinate whether it has a legal syntax or not. If it has a correct syntax and a true condition, it can be executed. At that time, b is evaluated. So, when we first see the line $x := b$; in the if-statement, we do nothing except to write down the line as what it is in input. The work of evaluation and assigning the value of b to x is done only when the syntax is correct and the condition is TRUE.

Variables can also be classified into global variables, local variables and parameters in a procedure. Variables defined in a procedure definition are local variables. They are stored in the local variable list of the procedure. So a local variable `local_var` may apear in two different procedure making no confusion. Parameters of procedure are stored in the parameter list of the procedure. Other variables are global variables and stored in the variable table. In the above example, x and b are inside variables, also they are global variables.

4.2. Statement templates

In order to make an interpret and execution, we have to construct statement templates. Here, we just consider the implementation of if-statement, which reveals the general ideas to implement other statements. The if-statement is like:

```
if(condition) then
    then_list
else
    else list
fi;
```

The class `If_clause` is defined as:

```
class If_clause
{ public:
    /* operations on if-statement */
  private:
    Bool_cell bool;
    Run_list  then_list;
    Run_list  else_list;
}
```

If the condition is ture, that is, *bool* is equal to TRUE, then the `then_list` is executed. Otherwise, if the condition is false, the `else_list` is executed.

```
Algorithm:
if_run(If_clause ifs)
input : a if statement ifs
output: no output. Just execute the statement.
```

The condition of if-statement is a little different from the boolean expression in that it may have another value besides true and false. We called this value "unknown". For example:

```
>if(a>3) then b:=1;
     else   b:=2;
 fi;
```

in which, a is not defined before. In this case, you can't determinate whether $a > 3$ is true or false. And thus, neither the then-list nor the else-list is executed. It does nothing.

4.3. Procedure

Before the end of section 4, we discuss how to implement *procedure* in STAR-PSE.

The definition of procedure is like:

```
proc_name:=proc(parameters)[local local_var_list;]
                do_list;
           endp;
```

Here, [ ] is optional.

After receiving a procedure definition, the `procedure name`, `parameters`, `local variables` and `do_list` should be put into a table of procedure. The procedure table is declared as:

```
PROC_DEF proc_tab[ProcTabLen];
```

in which `PROC_DEF` is defined as:

```
struct{
      char* name;
      class Var_list* para_list;
      class Var_list* local_list;
      class Run_list* body_list;
      }
```

A procedure may has its own local variables. The definition domain of a local variable is within the definition of the procedure. So, two different procedures may select the same string as the name of a local variable, but these two local variables refer to different ones. Also, a local variable may have the same name as a global variable. In this case, within the definition of the procedure, the name refers to the local variable; outside the procedure definition, it refers to the global variable.

A procedure may recursively call itself. In this case, each one has different value of parameters and local variables. So, we must keep down the values for each one, which is implemented by using a stack. It is declared as:

```
Call_Recorder Call_Stack[Call_Stack_len];
```

When a procedure is called, its parameters are first evaluated and then pushed into the stack. Also, the local variables are pushed into the stack. In the procedure body, if a statement refers to a parameter or a local variable, then it refers to the value in the stack. The stack pointer `Call_SP` gives the right position of the value.

```
Algorithm:
proc_run(Proc_call pcall)
input : a procedure call, that is,the procedure name
        and its parameters
output: the return value of the procedure
steps :
```

```
if(actual_parameter!=NULL)
   actual_parameter <-- eval(actual_parameter);
   // evaluate the actual parameters
in_run_list++;
Call_SP++;
if(Call_SP >= Call_Stack_Len)
{  print{"out of call stack"};
   return_val <-- 0;
   in_run_list--;
   Call_SP--;
   return;
}
Call_Stack[Call_SP]. local_list <-- local variable list
  // push the local variables
Call_Stack[Call_SP]. actual_list <-- actual parameters
  //push the parameters
run(body_list);
  // run the procedure body
Call_SP--;
in_run_list--;
return;
}
```

We use the variable `return_val` to record the result of the procedure call. If there's an error during the call, then it is set to zero. Otherwise, it is set to the result of the call. If the result of the procedure call is used by other statement, then the value can be gotten from `return_val`.

5. Remarks

STAR-PSE has been established as an interactive and programming environment for STAR. It works with other part of STAR, for example the number system of STAR[**?**] quite well. However there are still lots of work to do about STAR-PSE before it become a real PSE. On the other hand, as a powerful software tool, a symbolic computation software should be not only an interactive system, but also a friendly perfect work platform so that it is easy to access and use especially for beginner. To obtain these goals, as the first step, we should build STAR-PSE as a GUI system. The example below shows we have begun to do this[**?**].

People believe a visualizable calculating software tool will make the result come alive. In the future, we will add more graphic features to our system STAR.

Acknowledgments

We would like to thank all colleagues of ours for STAR project, especially are indebted to Wen-tsun WU and Wen-da WU for their generous support. Naixiao ZHANG from Peking University also provided many useful suggestions.

References

[1] ALFRED V.AHO, JEFFERY D.ULLMAN. Principles of Compiler Design. ADDISON-WESLEY PUBLISHING COMPANY, 1977.

[2] Alkiviadis G.Akritas. Elements of Computer Algebra with Applications. John Wiley & Sons, Inc. 1989

[3] Bruce W.Char, Keith O.Geddes, Gaston H.Gonnet, Benton L.Leong, Michael B.Monagan, Stephen M.Watt. Maple V Language Reference Manual. Spring-Verlag, 1991.

[4] B. Buchberger, G.E. Collins, R. Loos, R. Albrecht, *Computer Algebra — Symbolic and Algebraic Computation*, Second Edition, Springer-Verlag , 1988.

[5] J.H. Davenport, Y.Siret and E.Tournier Computer Algebra. Academic Press, 1988.

[6] Johnson, S.C. "YACC-yet another compiler compiler." CSTR32, Bell Laboratories, Murray Hill, N.J., 1974.

[7] Y. Ju, *A Problem Solving Environment for STAR*, Thesis for Master degree, Institute of Systems Science, Academia Sinica, Beijing 1996.

[8] Lesk, M.E. "LEX-a lexical analyzer generator." CSTR 39, Bell Laboratories, Murray Hill, N.J., 1975.

[9] Jun LIU & Zhuojun LIU, Upgrade and application of lex and yacc in message-driven GUI syntax analyzing software, preprint (in Chinese).

[10] Zhuojun LIU & Wenda WU. STAR: Small Tool for Algebraic Research. Proceedings of 2nd ASCM, Kobe in Japan, 1996.

[11] Paul S.Wang C++ with Object Oriented Programming. PWS PUBLISHING COMPANY, Boston, 1994.

[12] Sanjiva Weerawarana. Problem Solving Environments for Partial Differential Equation Based Applications. Augest 1994. Ph.D thesis.

[13] SUN microsystems. Programming Utilities & Libraries. 1990.

[14] Dingkang Wang & Lihong Zhi. Software Development in MMRC. Proceedings of the First Asian Technology Conference in Mathematics. Singapore, 1995.

[15] Wu Wen-tsun "On the Decision Problem and the Mechanical Theorem Proving and the Mechanization of Theorem in Elementary Geometry." Scientia Sinica 21, 1978.

[16] Wu Wen-tsun "Basic Principles of Mechanical Theorem Proving in Geometrics". J. of Sys. Sci. and Math. 1984.

[17] Hong YANG, Kai HUANG, Zhuojun LIU, The number system of STAR, preprint, (in Chinese)