|  |
| --- |
| CS 440 |
| Programming Languages and Translators |
|  |
| MCL: Parse report |
| 03/19/2017 |

|  |
| --- |
| Yuyang Luo  yluo41@hawk.iit.edu  Cai Chen  zdthm007@gmail.com |

**Symbol Table**

We created an unordered map named STable to be the symbol table of this parser (typedef unordered\_map<string,ID\_V\*> STable). The key value(string) is a string to store the identifier’s name, and the mapped value(ID\_V\*) is a struct pointer to store the identifier’s properties. We store the type, value and its location in the table of the identifier.

struct ID\_V {

ID\_V(int type,char\* id=NULL,int v=0, matrix \*m=NULL,int temp=0);

~ID\_V();

int type;

char \*id;

int val;

int t;

matrix \*var;

} ;

**Variables’ Define and Initialization:**

In this parser, we define two types: integer and matrix. If you define another type to a variable, it will cause an error. You just can initial integer with an integer type and a matrix with a matrix type or an array type, otherwise you will get an error. If you define an identifier more than once in a scope, it will cause an error. If you define a type but try to assign the identifier with a value of another type, it will cause an error. We use a type\_check method to do the type error check. Since we stored everything of an identifier in the ID\_V struct, we can check the type of this identifier easily.

You can give a value to an identifier or not when you define it. We use one 1024-length temporary array(a pointer array of struct ID\_V) to store the values of integer or matrix. Meanwhile, we use a method called newSymbolVal(char \* id, int type, ID\_V \*src) to save the initialized information with identifiers. We will check if the Symbol table(the unordered map named STable) has this identifier first, if it has already defined, the parser will report an error "symbol %s already define". If not, we can store the corresponding information of the identifier to the symbol table. Now we build a connection between identifiers and their values so that we can take out the value of an identifier easily based on its name. If you didn’t give a value to an identifier when you define it, we set a default value 0 for it.

**Variables’ Operation and Updating Value:**

In this parser, you can do the simple four operations of integers and plus, minus and multiply of matrixes. Also you can do a multiply and divide operation between an integer and a matrix. We still provide the reverse method to integer and matrix(e.g. rev 1 = -1), but we do not support these operations: matrix / matrix and int / matrix.

We just use one operation method to fit two types: integer and matrix. We use struct pointer to pass the parameters to operating method. Then we can get the types and values of two identifiers, then we can do type check to operation corresponding method for each data type. For example:

ID\_V \* mut (ID\_V\* a, ID\_V\* b)

{

if(TYPE(a)==INT&&TYPE(b)==INT)

return temp\_val(INT,INTV(a)\*INTV(b),NULL);

matrix\* temp=NULL;

if(TYPE(a)==INT&&TYPE(b)==MAT)

temp=matrix\_muI(MATV(b),INTV(a));

else if(TYPE(a)==MAT&&TYPE(b)==MAT)

temp=matrix\_mu(MATV(a),MATV(b));

else

temp=matrix\_muI(MATV(a),INTV(b));

if(temp!=NULL)

return temp\_val(MAT,0,temp);

yyerror("dimension match error");

exit(-1);

}

For the matrix, after we process the calculation of variables’ value, we use a temporary array to save the struct pointer. Then we can update the symbol table easily using the new data. First, we find the identifier’s information struct ID\_V through its name, then we use the new struct pointer stored in that temporary array to replace the information of that identifier in the symbol table. The update method code is below:

void updateSymbolVal (char\* id,ID\_V\* src)

{

ID\_V \* idv=symbolVal(id);

updateSymbolVal (idv,src);

}

void updateSymbolVal (ID\_V\* idv,ID\_V\* src)

{

type\_check(idv,src);

if(TYPE(idv)==MAT&&src->var)

idv->var=matrix\_copy(src->var);

else

idv->val=src->val;

if(TEMP(src))

{

temp\_[src->t]=NULL;

delete src;

}

}

**AST Generation and Output:**

We defined a struct named AST to store each node of the AST we will output.

typedef struct ast

{

void\* cot;

int type;

int var;

struct ast \*l,\*r;

int par;

int lv;

} AST;

As you can see above, we store its type, value, left child node, right child node, and the level of the root, void\* cot used to print the value or type name in ast. Whenever we process an identifier, we will make a node for the operators and the interralated value sequentially. At last, we will print out the binary tree as our AST.

**Print and Clean:**

We set a print method to print the result of identifier’s value and a clean method to release the used buffer of the parser.

**Compile Order:**

$ yacc MCL\_yacc.y -d

$ lex MCL\_lex.l

$ gcc MCL.c MCL.h lex.yy.c y.tab.c

**Code Example:**

int a=1;

int b=2,c=3;

a=b+c;

print a;

print 3+2;

print 1+2\*3;

matrix \_x[2,3]={1,2,3,4,5,6};

matrix \_y[3,2]={1,2,3,4,5,6},\_z[3,3]={1,1,1,2,2,2,3,3,3};

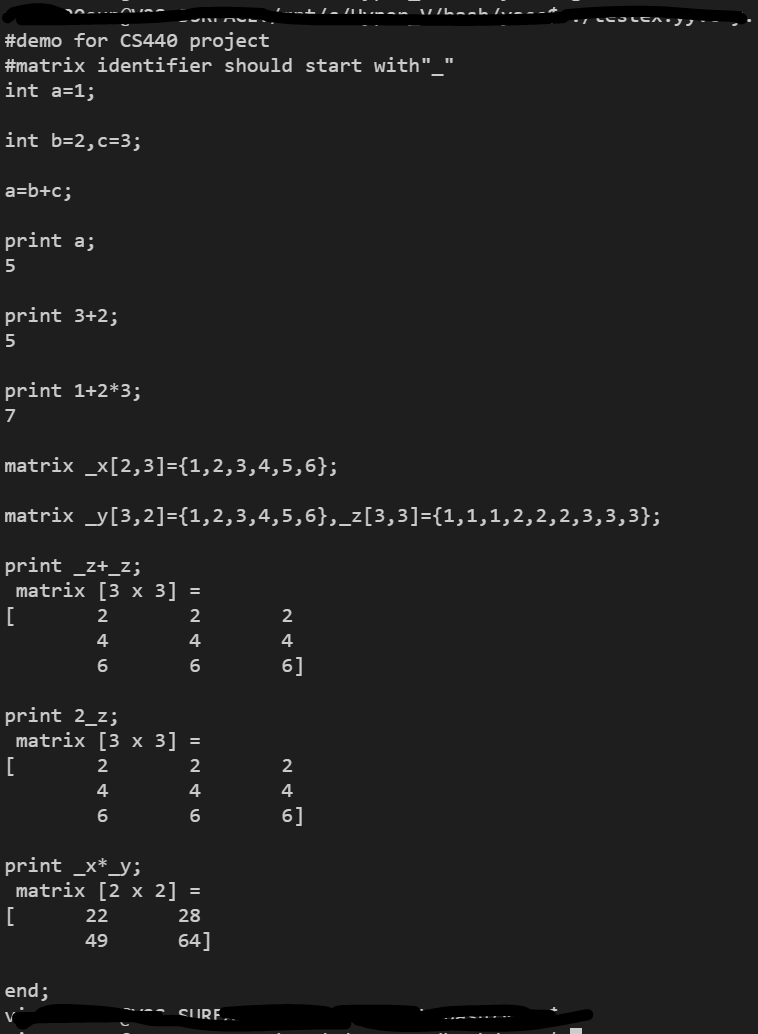
print \_z+\_z;

print 2\_z;

print \_x\*\_y;

end;

//also include in file MCL\_test\_example.txt

Outputs: 

**Output AST:** replace MCL\_yacc.y with MCL\_yacc\_ast.y, the program will output ast tree with each nodes with its id and its parent’s id and the nodes in same level will be in same line;

Example:

