ECOTE – Final project

Semester: 2018L

Author: Ángel García Malagón

Subject: Compiling Techniques

I. General overview and assumptions

My task is to write a program finding common subexpressions and substituting
them with temporary location. Identifiers in expression are one character, operators +,*,/,-,()
This problem is know like Common Subexpression Elimination(CSE).The benefits
of performing CSE are great enough that it is a commonly used optimization.
My program is written in C++ and it will be apply to a code of my language, and we will obtain the same code but without common subexpressions.

II. Functional requirements

As I said previously, my program is written in C++ and it is applied to a code of my language, which is based on C++ . My language is defined by its lexer and its syntax. The lexer is the set of different types of tokens available in the language. For my language, its lexer is :

{**ENDFILE,**
ERROR,
EMPTY,
INT,
RETURN,
VOID,
ASSIGN,
SEMICOLON,
COMMA,
LEFT_PARENTHESE,
RIGHT_PARENTHESE,
LEFT_BRACE,
RIGHT_BRACE,
COMMENT,
PLUS,
MINUS,
MULTI,
DIV,
NUM,
ID}

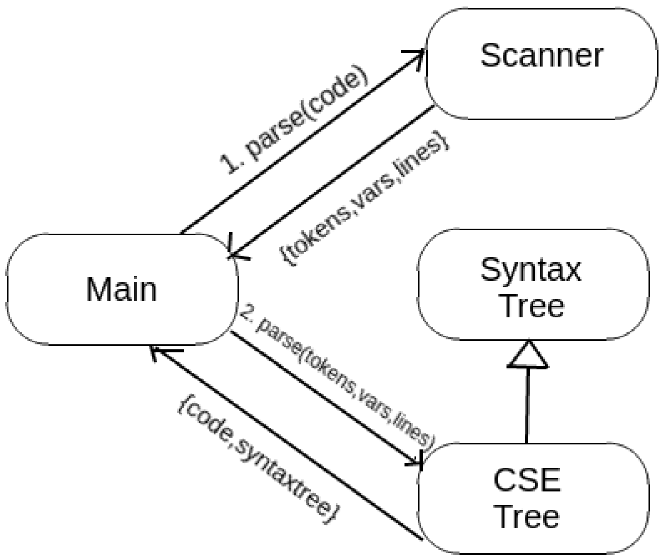
With this lexer, when my program read a sample of code, the Scanner will check if all elements of that code correspond to a valid token of my language. If any element don't correspond to a valid token, that means that there is a lexical error in that code. On the other hand, the syntax defines the structure that a code of my language must follow. For my language, its syntax is the following:

<PROGRAM>	**->**	**<DECLARATION>**
<DECLARATION>	**->**	**<TYPE> ID LEFT_PARENTHESE RIGHT_PARENTHESE <COMPOUNT_STMT>**
<TYPE>	**->**	**VOID | INT**
<COMPOUNT_STMT>	**->**	**LEFT_BRACE <LOCAL_DECLARATIONS> <STATEMENT_LIST> RIGHT_BRACE**
<LOCAL_DECLARATIONS>	**->**	**<VAR_DECLARATION>***
<VAR_DECLARATION>	**->**	**<TYPE> ID SEMICOLON**
<STATEMENT_LIST>	**->**	**<STATEMENT>***
<STATEMENT>	**->**	**<RETURN_STMT> | <EXPRESSION_STMT>**
<EXPRESSION_STMT>	**->**	**<ASSIGN_EXPRESSION> SEMICOLON**
<RETURN_STMT>	**->**	**RETURN <EXPRESSION> SEMICOLON**
<ASSIGN_EXPRESSION>	**->**	**<FACTOR> ASSIGN <SIMPLE_EXPRESSION>**
<SIMPLE_EXPRESSION>	**->**	**<TERM> (<ADDOP> <TERM>)***
<ADDOP>	**->**	**PLUS | MINUS**
<TERM>	**->**	**<FACTOR> (<MULOP> <FACTOR>)***
<MULOP>	**->**	**MULTI | DIV**
<FACTOR>	**->**	**LEFT_PARENTHESE <EXPRESSION> RIGHT_PARENTHESE | ID | NUM**

With this syntax, my Syntax Analyser will check if the code follows that syntax, That means that the code follows the syntax productions. If the code doesn't follow the syntax, the Syntax Analyser throws a Syntax Error . If there is no error in the code, the SyntaxTree of the code will be contruct and I will apply the CSE algorithm is that Tree.

III. Implementation

General architecture



First, I read the input file in the main and I extract the code from it. After that, I use Scanner to analyze the code, extracting the set of tokens, variables and lines of each token. With these set I will construct the CseTree, that is a SyntaxTree, but in this Tree I will apply the algorithm . When I apply the algorithm I will be able to obtain the code with the common subexpressions eliminated.

Data structures

Scanner → Used to extract tokens and variables from code
SyntaxTree → Create a SyntaxTree of code using the set of tokens and variables
Node → Node of SyntaxTree
TokenType → Type of token allowed by the scanner
CseTree → SyntaxTree when I will apply CSE algorithm

Module descriptions

<u>Scanner</u>

This module performs a lexical analysis of the input code. The Scanner goal is to check that all elements of input code correspond with a valid token of my language
If any element doesn't correspond with a valid token, Scanner throws a Lexical Error. Otherwise, I will obtain three sets from the scanner: set of tokens, variables and line of each token.
This analysis is performed in parse method. In this method, the scanner read the input code char by char and check if these char correspond with a valid token or a valid keyword or variable.

<u>SyntaxTree</u>

The Syntax Analyzer is to check if the input code follows the syntax of my language
This means that the code follows the syntax productions shown above
If Syntax Analyzer throws no error, the SyntaxTree will be created.
In this module, we also have a Hash Table storing the declared variables. This is to check if in the code it's used a variable which it isn't declared.
When the Syntax Analyser is analyzing the code, to see if the following token in the list follows the syntax, it uses the method matchToken(token), which check if the currentToken correspond with the token which is expected. This module has methods to iterate in the token list, that it will be useful to check if the list of tokens follows the syntax. The rest of methods correspond with the productions of my syntax.

<u>CseTree</u>

This module is also a SyntaxTree, this means that this module inherits all attributes and methods from SyntaxTree. This module has also other attributes and methods to perform CSE algorithm. I store the number of common subexpressions found and a HashTable with the new temporal variables created with the common subexpression.
The algorithm is performed in the method applyCSE(). The pseudocode of the algorithm is:

ApplyCSE

```
    assigns ← getAssigns()

    **while** assigns != NULL **do**
        tocomp ← assigns.sibling
        **foreach** cs **in** cs_found **do**
            find(assigns, cs)
        // compare each expression with the rest
        **while** tocomp != NULL **do**
            find(assigns,tocomp)
        assigns ← assigns.sibling

    **foreach** cs **in** cs_found **do**
        createNewVariable(cs.first)
        createNewExpression(cs.second)
```

The method ‘*find’ search a common subexpression between tocomp and assigns and if it would find any cs it will substitute it for a new temporal variable (or a already created temporal variable if that common subexpression is already in cs_found*

Input/output description

Input will be a file with code of my language with the following structure:

```
int main(){
    int a0;
    int a1;
    int a2;
    .
    . n vars
    .
    int an;

    Assign1;
    Assign2;
    Assign3;
    Assign4;
    …
}
```

Assign(n) is any assign expression (i.e: a=b*c+d)

Obviously, the input can have a different structure, but if that structure <u>doesn’t</u> follow the syntax there will be a syntax/lexical error

By default, in my program the output will be the code with common subexpressions eliminated. The code will be obtain only if no error was detected by the lexical/syntax analyser. If there was any error in the code, in the output file you can find the error and the line where the error occurred.

As I said previously, that ouput is the default output. I can also print the CseTree after apply the algorithm and the SyntaxTree before apply the algorithm.

Example:

Input	Output

int main(){
  int a;
  int b;
  int c;
  int d;
  int e;
  int z;
 	
  a=b*c+d;
  z=c*b-e;
 	
}
	int main(void){
	        int a;
	        int b;
	        int c;
	        int d;
	        int e;
	        int z;
	        int tmp0;
	
	        tmp0=c*b;
	        a=tmp0+d;
	        z=tmp0-e;
	}

I

Others

IV. Functional test cases

I have created 10 test files to check if my program works well. To execute the program you have to use the command:

```
make run
```

Then, 10 output files will be created in the output folder.
To see all cases, you only have to execute my program and check the output files in the output folder.
Anyway, I write some of those tests below:

test3	test3.out
int main(){	int main(void){
int a;	int a;
int b;	int b;
int c;	int c;
int d;	int d;
int e;	int e;
int z;	int z;
int v;	int v;
	int tmp2;
	int tmp0;
z=b*e-(e/d);	int tmp1;
z=e*b+(d/e);	
d=2*z*(b+c);	tmp2=b+c;
v=a*(b+c);	tmp0=e*b;
e=a*(e/d);	tmp1=e/d;
	z=tmp0-tmp1;
}	z=tmp0+d/e;
	d=2*z*tmp2;
	v=a*tmp2;
	e=a*tmp1;
	}

test5	test5.out
int main(){	Line 13-> Syntax Error: Unexpected token ID.
int a;	SEMICOLON expected.
int b;	
int c;	
a=a*b+c	
b=b/c+3;	
}	

test6	test6.out

int main(){
  int t;
  int x;
  int h;
  int p;
  int y;
 	
  t=h/y+h;
  x=x+y-2*t;
  p=h*(x+y)+y+x;
 	
}
	int main(void){
	        int t;
	        int x;
	        int h;
	        int p;
	        int y;
	        int tmp0;
	
	        tmp0=x+y;
	        t=h/y+h;
	        x=tmp0-2*t;
	        p=h*tmp0+tmp0;
	
	}

test4	test4.out

int main(){
  int a[];
  int b;
  int c;
  int d;
  int e;
 	
  e=b-d;
  d=a+c;
 	
}
	Line 8-> Lexical error: [isn't a valid element of my lenguage.