CMPT 432 - Fall 2019 - Dr. Labouseur

Lab One

John DeFalco

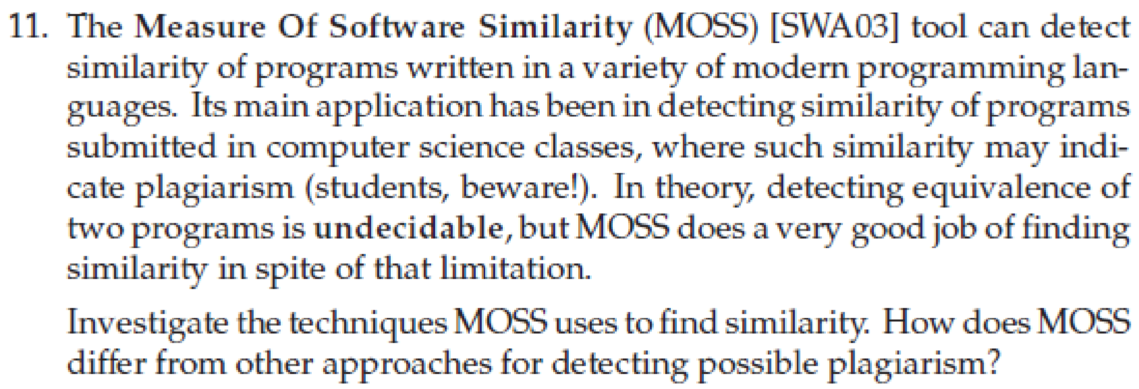
john.defalco1@marist.edu

September 4th, 2019

1 Crafting a Compiler

Below are the examples listed on the requirements document for the lab from the *Crafting a Compiler* textbook.

1.1 Problem 1.11



11. The **Measure Of Software Similarity** (MOSS) [SWA03] tool can detect similarity of programs written in a variety of modern programming languages. Its main application has been in detecting similarity of programs submitted in computer science classes, where such similarity may indicate plagiarism (students, beware!). In theory, detecting equivalence of two programs is **undecidable**, but MOSS does a very good job of finding similarity in spite of that limitation.

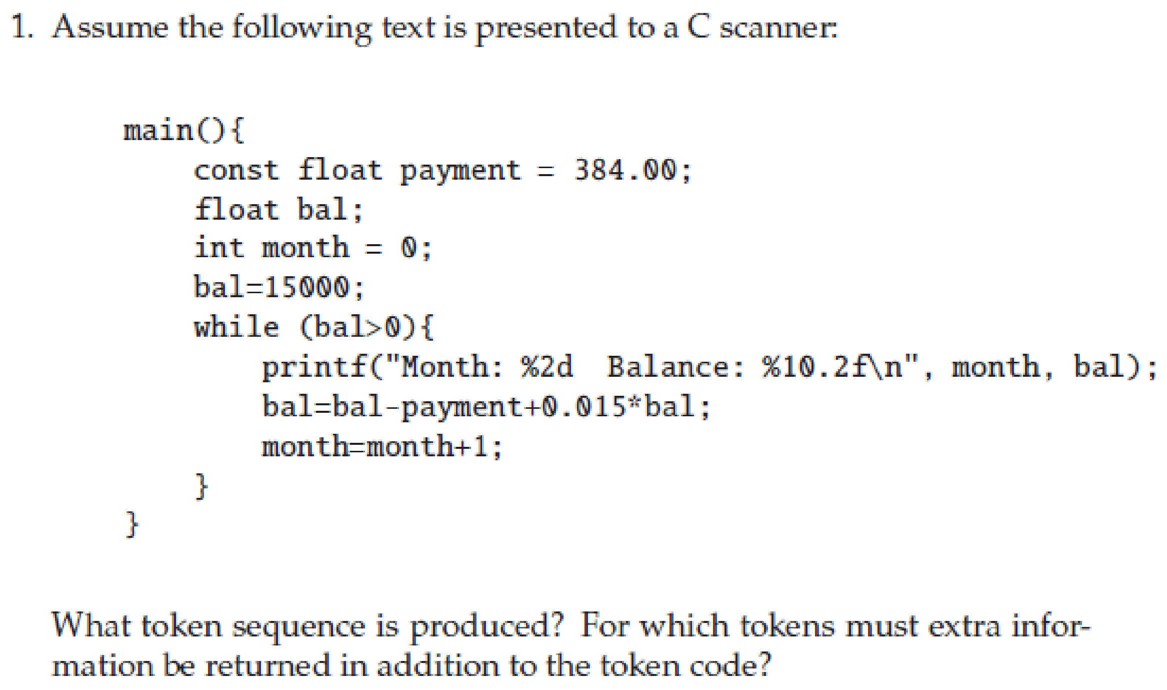
    Investigate the techniques MOSS uses to find similarity. How does MOSS differ from other approaches for detecting possible plagiarism?

MOSS uses an algorithm called *winnowing* for document fingerprinting, so it can then compare between standardized versions of the documents to check for similarities[1]. Through this method, MOSS looks for similar syntax between documents to determine if there is plagiarism or not. This is different than standard

[1] [1]

plagiarizing prevention software, as the usual method involves looking at word frequency in the document to determine if there is plagiarism. MOSS's method is much more efficient, as it is easy to change variable names and types around enough to not raise suspicion. However, when copying someone else's work it is more difficult to change the syntax of the program, because the individual copying likely doesn't understand the program well enough to deconstruct it and rebuild it to look more unique.

1.2 Problem 3.1



1. Assume the following text is presented to a C scanner:

```
main(){
    const float payment = 384.00;
    float bal;
    int month = 0;
    bal=15000;
    while (bal>0){
        printf("Month: %2d  Balance: %10.2f\n", month, bal);
        bal=bal-payment+0.015*bal;
        month=month+1;
    }
}
```

   What token sequence is produced? For which tokens must extra information be returned in addition to the token code?

Looking at the text, there are roughly 56 tokens presented in the source code. The tokens, with their respective types, are listed in Table 2.1.

Along with these tokens, the ones that would need to pass on more information would be those that are special characters, operators, keywords, and constants. These tokens need to specify exactly what kind of constant or character they are, so the parser can understand the context of what is going on within the source code. For example, special characters include deliminator and end of line characters, operators are all binary operators including the assignment operator, keywords are those special words specified by the language, and constants can be number constants, decimal constants, or even character constants.

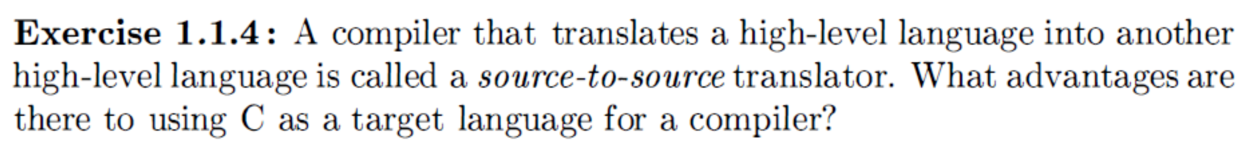
Token	**Type**	**Token**	**Type**
main	Identifier	{	Delimiter
(	Delimiter	printf	Identifier
)	Delimiter	(	Delimiter
{	Delimiter	"Month...n"	String
const	Keyword	,	Separator Character
float	Keyword	month	Identifier
payment	Identifier	,	Separator Character
384.00	Constant	bal	Identifier
;	Eol Character	)	Delimiter
float	Keyword	;	Eol Character
bal	Identifier	bal	Identifier
;	Eol Character	=	Operator
int	Keyword	bal	Identifier
month	Identifier	-	Operator
=	Operator	payment	Identifier
0	Constant	0.015	Constant
;	Eol Character	*	Operator
bal	Identifier	bal	Identifier
=	Operator	;	Eol Character
15000	Constant	month	Identifier
;	Eol Character	=	Operator
while	Keyword	month	Identifier
(	Delimiter	+	Operator
bal	Identifier	1	Constant
>	Operator	;	Eol Character
0	Constant	}	Delimiter
)	Delimiter	}	Delimiter

Table 1.1: Token sequence in Problem 3.1

2 Dragon Book

Below are the examples listed on the requirements document for the lab from the *Compilers (Dragon)* textbook .

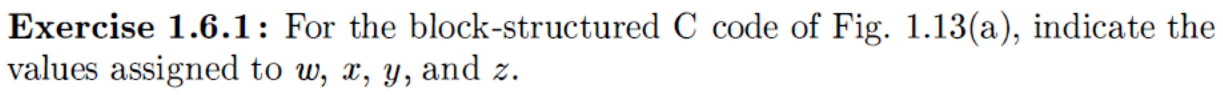
2.1 Problem 1.1.4



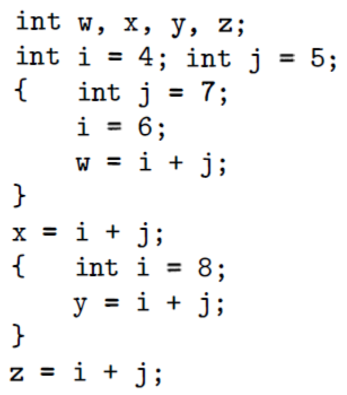
Exercise 1.1.4: A compiler that translates a high-level language into another high-level language is called a *source-to-source* translator. What advantages are there to using C as a target language for a compiler?

The *C* language has compilers on almost any platform available, making it somewhat universal as one could run it anywhere. Having *C* as the target language for a translator would mean that no matter what the source language is, the program being compiled would be able to be executed on any platform that runs *C*. Due to this being a significant percentage of platforms available, the initial source code would then in part become way more widely available.

2.2 Problem 1.6.1



Exercise 1.6.1: For the block-structured C code of Fig. 1.13(a), indicate the values assigned to w, x, y, and z.



```
int w, x, y, z;
int i = 4; int j = 5;
{   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{   int i = 8;
    y = i + j;
}
z = i + j;
```

(a) Code for Exercise 1.6.1

Starting with w, we can see that it is assigned a value within one block, but not the global block encompassing the entire code snippet. Within the embedded block, the variable $j = 7$ and $i = 6$. We can see that $w = i + j$, so we can see that $w = 13$ within the scope of the embedded block. Outside of that block however, w has not been assigned any value.

Moving on to x, we see that globally it has the definition of $x = i + j$. While this is the same definition as w, it is not within the same embedded block, and the values of i and j are now different because they're within different scopes. While for w, $j = 7$ and $i = 6$, in this new scope j is now equal to 5 and i is equal to 4, as per the second line of the code provided. Therefore, if $x = i + j$, then $x = 9$.

Similarly to w, y is given a definition in another embedded block within the code. The definition of y is once again the same as both w and x, where $y = i + j$. However, in this scope, only the value for i has been declared, $i = 8$. Because there is no definition for j within the scope of the embedded block, we can take the value assigned outside in the global block, where $j = 5$. So if $y = i + j$, we can see that $y = 13$. Like w, this value is just within the embedded block, and the variable y does not have a value assigned to it outside of that scope.

Finally the definition of z can be found at the last line of the code snippet, with the same definition as the other variables, $z = i + j$. The values of i and j will be the same as the ones used for the definition of x, as the definition of z is within the scope of the global block. In this scope, i is defined as $i = 4$ and j is defined as $j = 5$, making $z = 9$, the same value as x.

References

[1] J. Hage, P. Rademaker, and N. van Vugt. A comparison of plagiarism detection tools. Technical report, 2010.