The Chinese University of Hong Kong, Shenzhen

CSC4180

Compiler Construction

CSC4180 Assignment 2

Student Name:

Bodong Yan

Student Number:

119010369

March 19, 2023

Contents

1 Execution Method and Notice

Execution and Output The environment needed for my code is simple, just including make and basic C++ environment. You can build the dokcer image with the attached Dockerfile, and run the code with the commands in Assignment2's description. As for the output, since the generated tokens list can be very long, so the generated tokens will be in both std output and the file ”output.txt” for convenience after the scanning finished.

Notice 1 If you find that the code cannot be compiled in the docker image because command ”g++” is not found, you can update the apt-get with ”apt-get update” and install build-essential with ”apt-get install build-essential”.

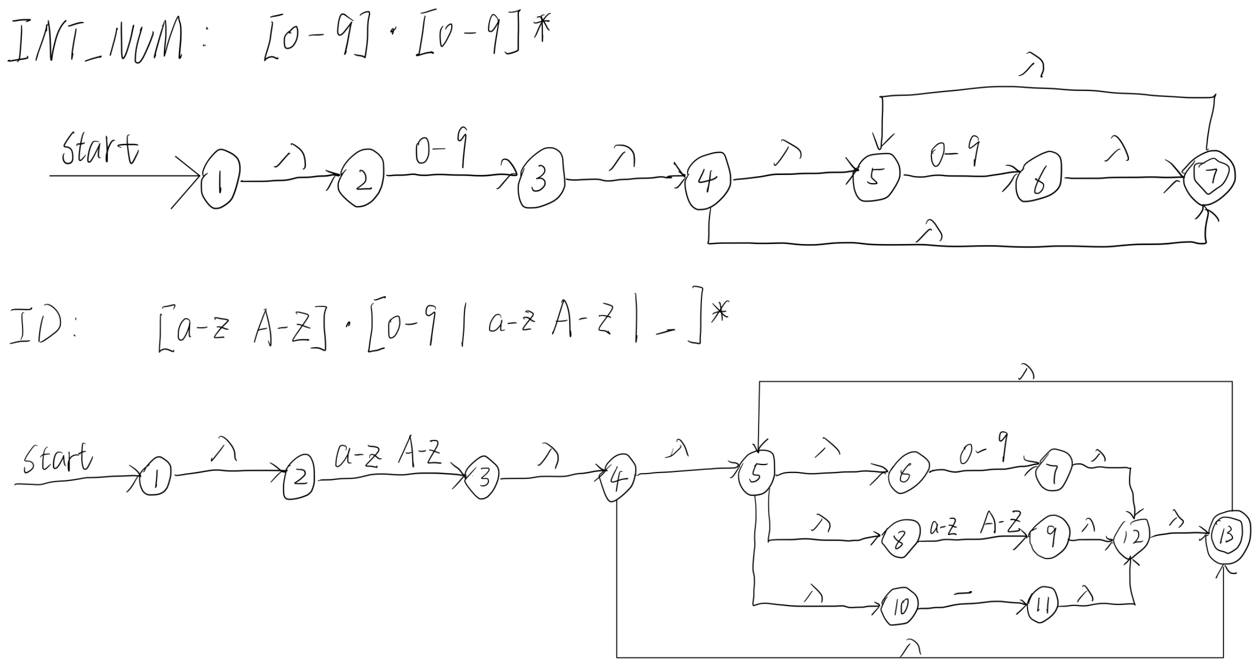
2 NFA and DFA

2.1 Regular Expression to NFA

Since the regular expression of keywords and special symbols can be easily translated to DFA, this subsection will only show the process of converting ID and INT_NUM from regular expression to NFA.

As is described in the document, the regular expressions of INT_NUM and ID are $[0-9]+$ and $[a-zA-Z]+[0-9|a-zA-Z|\_]*$ respectively. By using the formula in Lecture 3, the NFA can be obtained in Figure 1.

Figure 1: NFA of INT_NUM and ID



2.2 NFA to DFA

Now, the NFA of the two tokens are obtained. However, NFA cannot be used directly in the scanner. By applying the transformation steps shown in Lecture 3, the DFA of INT_NUM and ID are shown in Figure 2. The state number is modified from NFA for global design.

The two DFAs also cannot directly be used in programming as the DFAs of all tokens should be merged into one single DFA for the convenience of scanner's implementation. The final DFA is very large, thus only an example will be shown here. Figure 3 shows the example of merging the DFAs of "if", "int" and ID, state 46, 48, 49 are the same states in the DFA of ID. As is shown in Figure 3, token "int" occupies state 1, 2 and 3 while state 3 is final state of "int", however, there will be

IDs like ”int1”, ”in_t2”, or ”i_3nt”. Thus, state 1, 2, 3 should have links to state 46, 48, 49 to cover those situations. Finally, the resulted DFA for this scanner should include all the tokens’ states and cover all the situations.

Figure 2: DFA of INT_NUM and ID

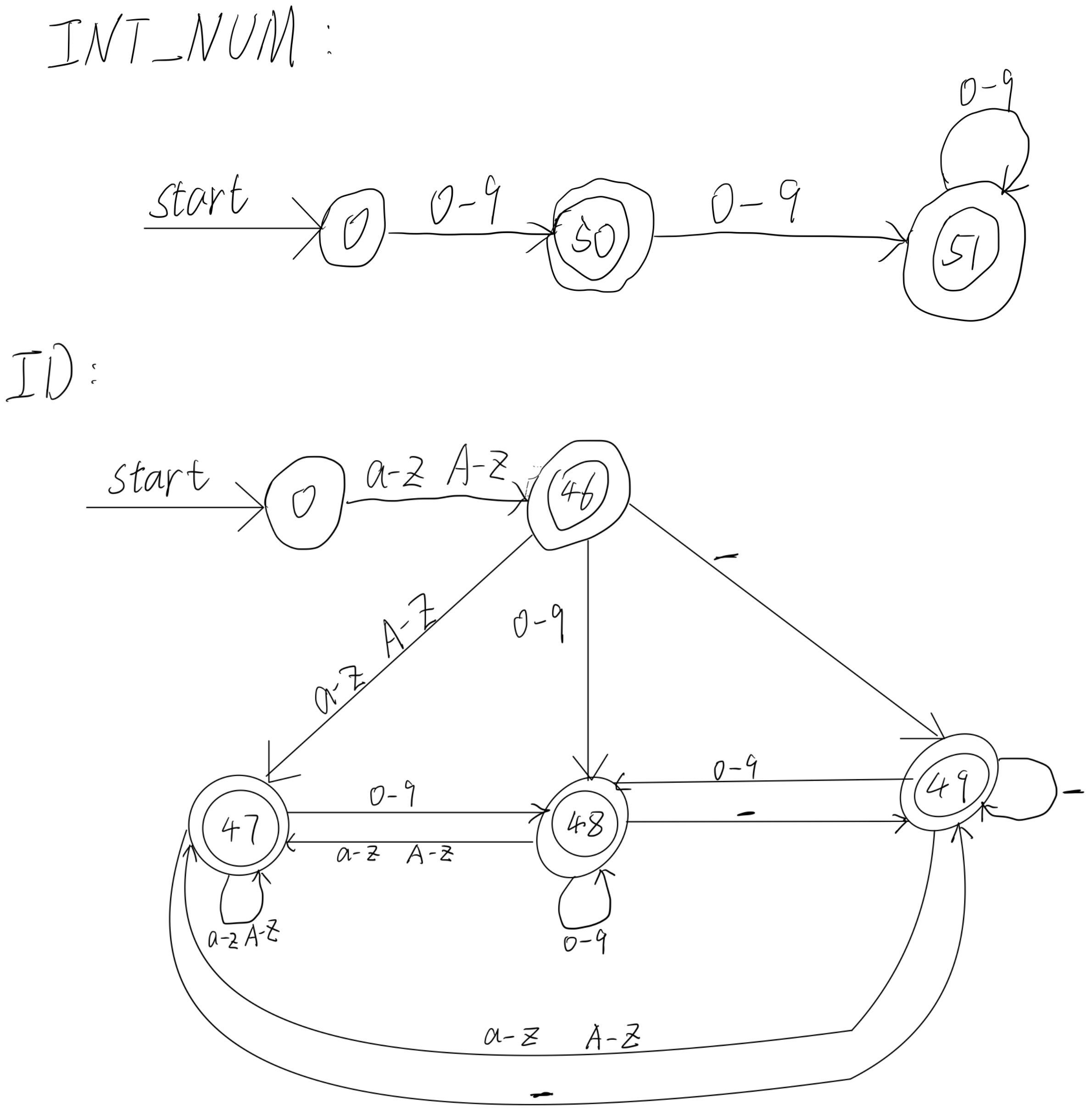
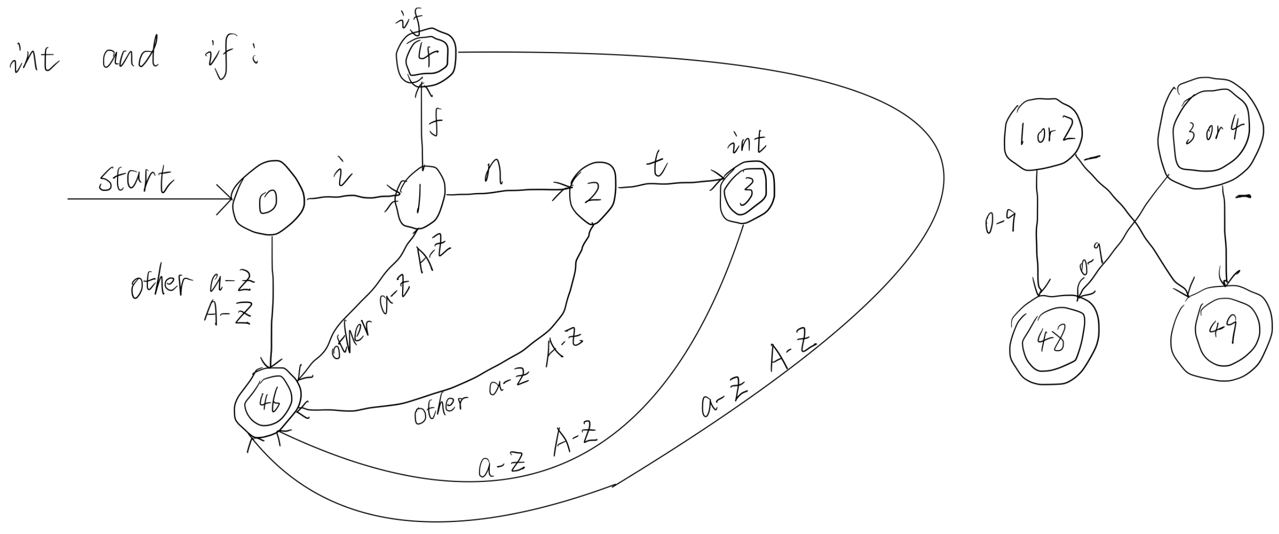


Figure 3: Merge int, if, and ID



2.3 DFA Representation in Program

The final DFA is represented by a 2-dimensional integer array in my program. The column corresponds to the state number, the row corresponds to totally 128 characters' ASCII code, and the element inside is the next state to go when one character is scanned in the current state. Here let's still take token "int" as an example, since its DFA is shown in Figure 3. The initial state is 0, and if character "i" is scanned in state 0, the state will go to state 1, so DFA[0][105] = 1 as ASCII code 105 corresponds to character "i". Moreover, DFA[1][110] = 2, DFA[2][116] = 3. During the execution of the scanner, if DFA[x][y] turns out to be 0, this means that the input file has syntax error and will be rejected by this scanner. The DFA is initialized by function initialize_DFA() before the scanner starts.

Another part of DFA representation in my program will be explained in next section, which is about the tokens' terminal state and is highly related to the design of my scanner's driver.

3 Scanner's Driver

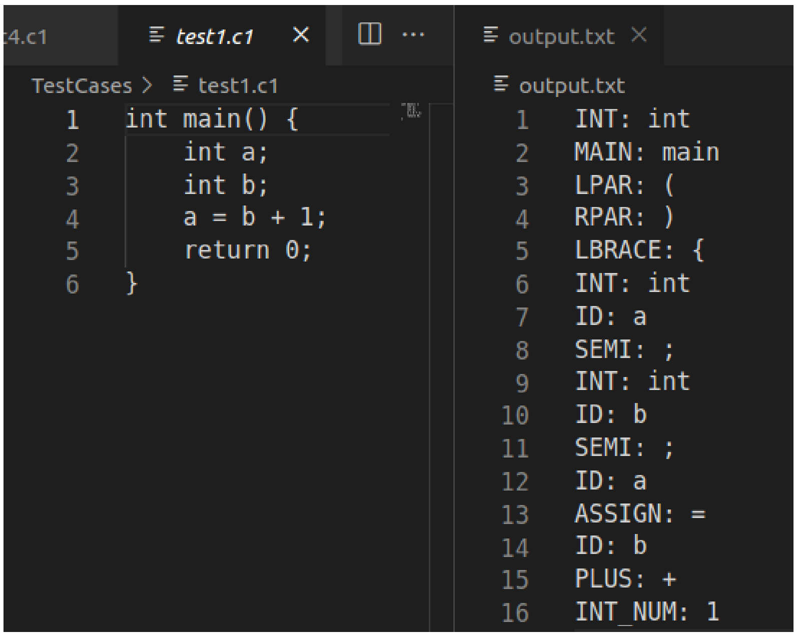
The driver of the scanner have mainly two functions: get characters from the input file one by one and output corresponding tokens by using the DFA. Initially, I just get one character and go to the corresponding state in each iteration. However, I found that the program cannot decide when to output a token with current DFA. For example, when the input is a stream of characters ”i”, ”n”, ”t”, the current state now reaches 3, which is the final state of token ”int”. However, if the following character is a-z or A-Z, the token will be like ”intA” which is an ID token. Thus, the DFA should have terminal state/symbols for each tokens and the scanner should look ahead one character to determine whether to output a token or continue the process. In each iteration, there will be one current character and one next character. In the next iteration, the next character's value will be assigned to the current character, and the next character will be a new character from the input file.

The terminal state/symbols should be designed carefully since each token's terminal symbols are different from each other. Let's take previous case as an example, after a stream of character ”i”, ”n”, ”t”, current state will be 3, and if the next character is an empty space or end of line, the next state number will be ”-1”. The minus sign indicate that the scanner now should out put a token and ”1” means the first token should be the target, which is token ”int”. But if the next character is a letter or digit, the scanner will do nothing but continue to the next iteration. The case is different for token ”return”. After a stream of character ”r”, ”e”, ”t”, ”u”, ”r”, ”n”, current state will be 12, if the next character is an empty space, end of line or ”;”, the next state number will be ”-4”, which means that the scanner should output the fourth token: ”return” should be output now. After the token is output, current state will go back to zero and the scanner will continue to the next iteration.

4 Testcases Output

The tokens that are output in std command line only shows the the name of the token. This section will show the result in ”output.txt” which contains not only the name of the token but also the value. All 5 testcases output are the same as the given expected output.

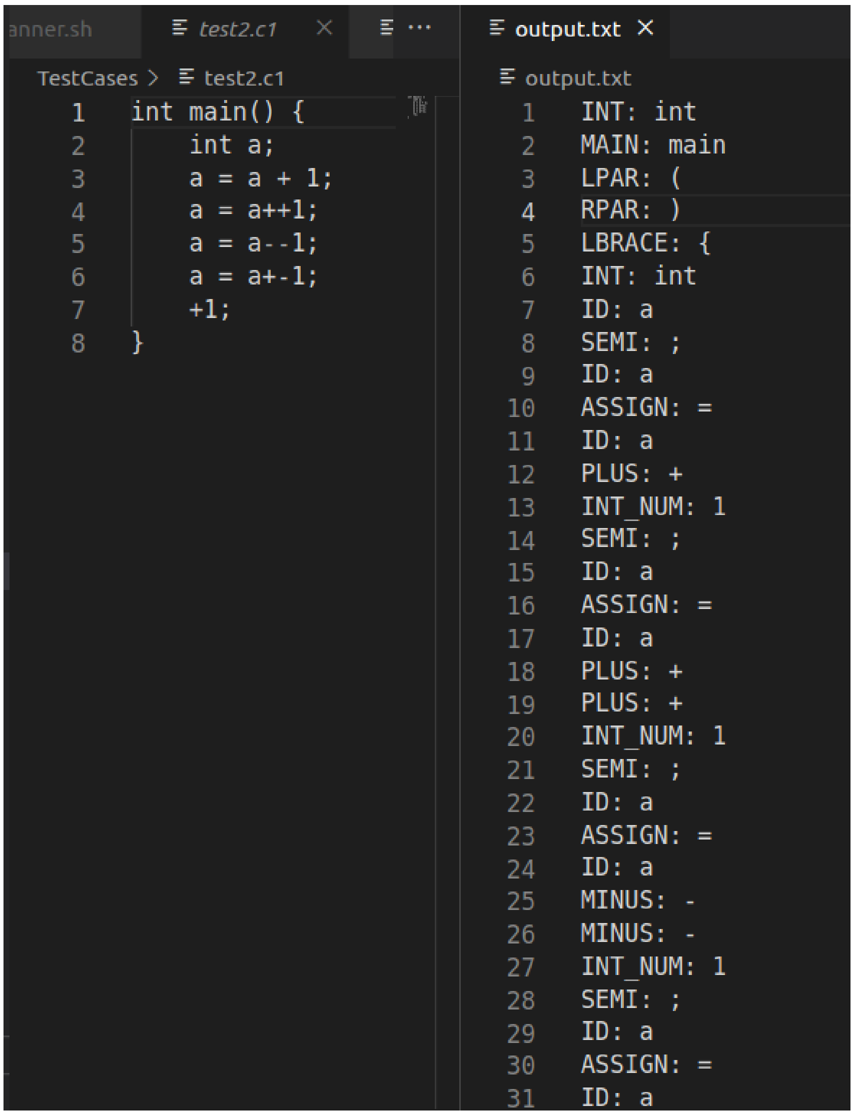
Figure 4: Output of test1



```
int main() {
    int a;
    int b;
    a = b + 1;
    return 0;
}
```

```
INT: int
MAIN: main
LPAR: (
RPAR: )
LBRACE: {
INT: int
ID: a
SEMI: ;
INT: int
ID: b
SEMI: ;
ID: a
ASSIGN: =
ID: b
PLUS: +
INT_NUM: 1
```

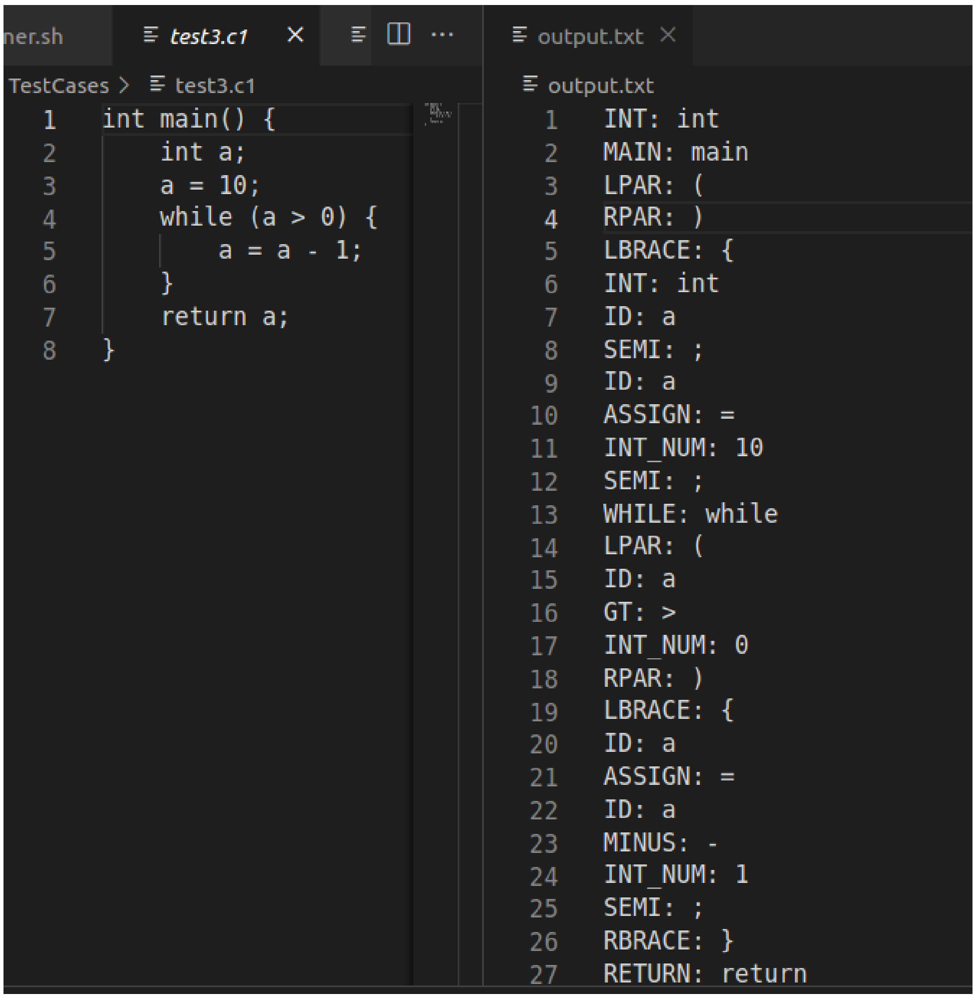
Figure 5: Output of test2



```
int main() {
    int a;
    a = a + 1;
    a = a++1;
    a = a--1;
    a = a+-1;
    +1;
}
```

```
INT: int
MAIN: main
LPAR: (
RPAR: )
LBRACE: {
INT: int
ID: a
SEMI: ;
ID: a
ASSIGN: =
ID: a
PLUS: +
INT_NUM: 1
SEMI: ;
ID: a
ASSIGN: =
ID: a
PLUS: +
PLUS: +
INT_NUM: 1
SEMI: ;
ID: a
ASSIGN: =
ID: a
MINUS: -
MINUS: -
INT_NUM: 1
SEMI: ;
ID: a
ASSIGN: =
ID: a
```

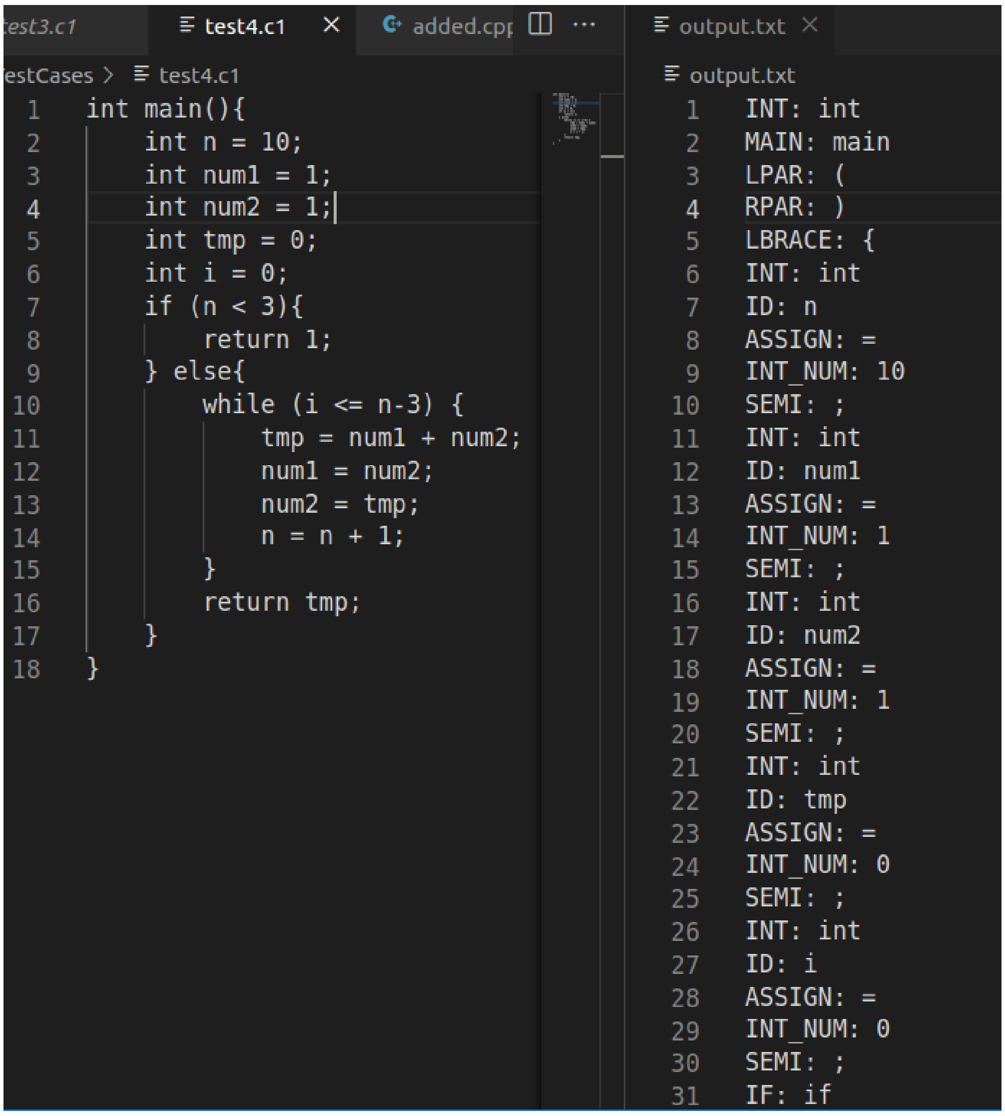
Figure 6: Output of test3



```
int main() {
    int a;
    a = 10;
    while (a > 0) {
        a = a - 1;
    }
    return a;
}
```

```
INT: int
MAIN: main
LPAR: (
RPAR: )
LBRACE: {
INT: int
ID: a
SEMI: ;
ID: a
ASSIGN: =
INT_NUM: 10
SEMI: ;
WHILE: while
LPAR: (
ID: a
GT: >
INT_NUM: 0
RPAR: )
LBRACE: {
ID: a
ASSIGN: =
ID: a
MINUS: -
INT_NUM: 1
SEMI: ;
RBRACE: }
RETURN: return
```

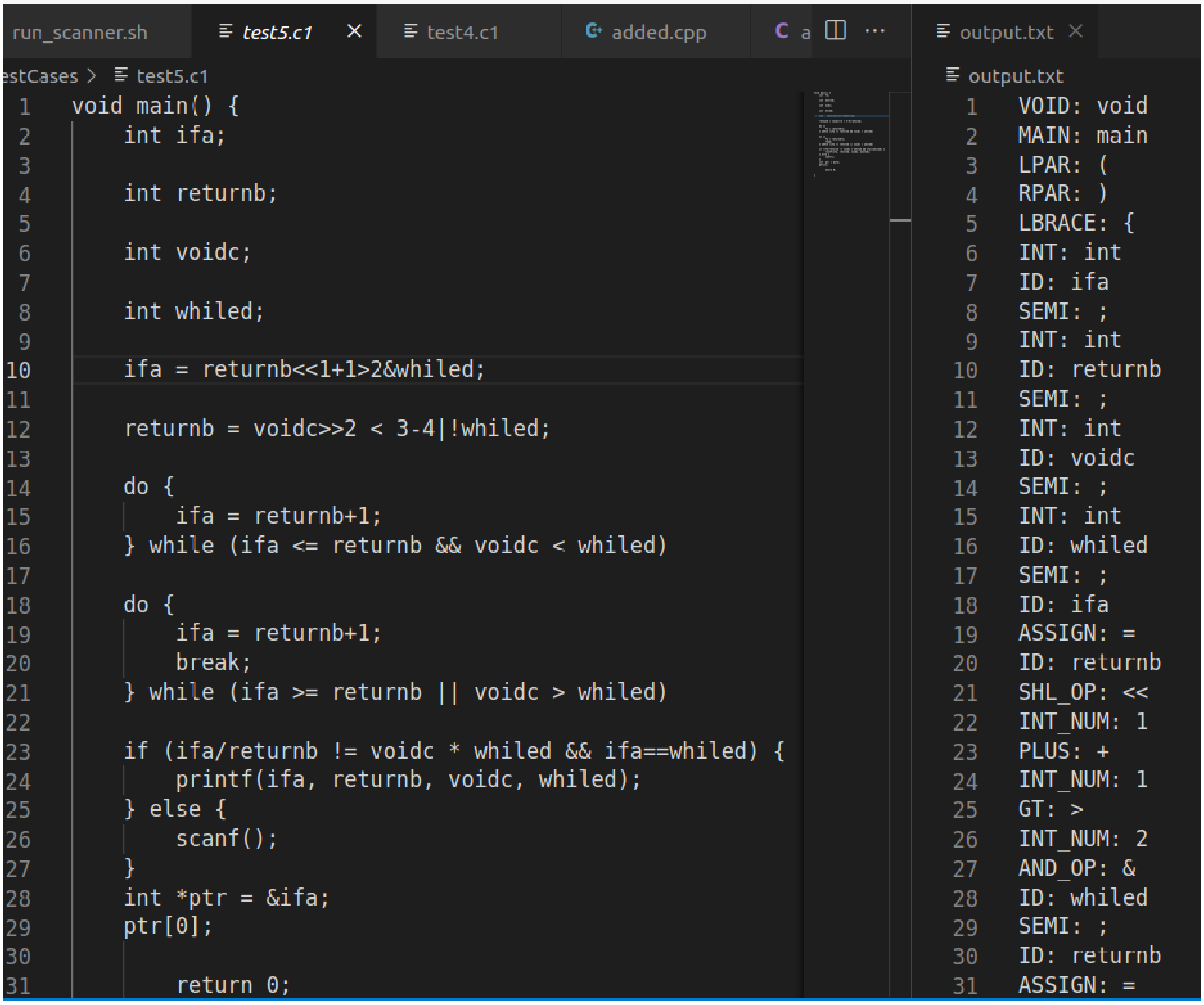
Figure 7: Output of test4



```
int main(){
    int n = 10;
    int num1 = 1;
    int num2 = 1;
    int tmp = 0;
    int i = 0;
    if (n < 3){
        return 1;
    } else{
        while (i <= n-3) {
            tmp = num1 + num2;
            num1 = num2;
            num2 = tmp;
            n = n + 1;
        }
        return tmp;
    }
}
```

```
INT: int
MAIN: main
LPAR: (
RPAR: )
LBRACE: {
INT: int
ID: n
ASSIGN: =
INT_NUM: 10
SEMI: ;
INT: int
ID: num1
ASSIGN: =
INT_NUM: 1
SEMI: ;
INT: int
ID: num2
ASSIGN: =
INT_NUM: 1
SEMI: ;
INT: int
ID: tmp
ASSIGN: =
INT_NUM: 0
SEMI: ;
INT: int
ID: i
ASSIGN: =
INT_NUM: 0
SEMI: ;
IF: if
```

Figure 8: Output of test5



```
void main() {
    int ifa;

    int returnb;

    int voidc;

    int whiled;

    ifa = returnb<<1+1>2&whiled;

    returnb = voidc>>2 < 3-4|!whiled;

    do {
        ifa = returnb+1;
    } while (ifa <= returnb && voidc < whiled)

    do {
        ifa = returnb+1;
        break;
    } while (ifa >= returnb || voidc > whiled)

    if (ifa/returnb != voidc * whiled && ifa==whiled) {
        printf(ifa, returnb, voidc, whiled);
    } else {
        scanf();
    }
    int *ptr = &ifa;
    ptr[0];

        return 0;
```

```
VOID: void
MAIN: main
LPAR: (
RPAR: )
LBRACE: {
INT: int
ID: ifa
SEMI: ;
INT: int
ID: returnb
SEMI: ;
INT: int
ID: voidc
SEMI: ;
INT: int
ID: whiled
SEMI: ;
ID: ifa
ASSIGN: =
ID: returnb
SHL_OP: <<
INT_NUM: 1
PLUS: +
INT_NUM: 1
GT: >
INT_NUM: 2
AND_OP: &
ID: whiled
SEMI: ;
ID: returnb
ASSIGN: =
```