ASSESSMENT A1

Question:
a) Find the runtime error in the given code and explain the reason.

Answer:
There is a runtime error because of the array (long arr[N]). In this code, user enters a long variable to initialise the length of the array. However, in C++, the length of the array must be initialised and determined at compile time, as the array used here is called a Variable-Length Array (VLA). This is to allow for the compiler to allocate appropriate memory needed for the array in the stack. Thus, in this program, as user has compiled the array at compile time without initialisation, it has a runtime error as the program does not know how much memory should be allocated to this array, making it unreliable and give some unexpected behaviour, such as displaying the maximum value of 'long' instead of the user's input after initialising the length of the array.

Thus, to fix this, the array should be made to be flexible, so that the length can be determined during runtime, in other words, user can initialise it after compiling it. Below is an example of how the array can be made flexible:

```
#include <iostream>
using namespace std;

int main() {
    long long N;
    cin >> N; //initialise array length

    // Dynamically allocate memory for the array
    long *arr = new long[N];

    for(int i = 0; i < N; i++){
        cin >> arr[i]; //get user input
    }

    for(int i = 0; i < N; i++){
        cout << arr[i] << " "; //print user input
    }

    // Free the dynamically allocated memory
    delete[] arr; //delete array

    return 0;
}
```

The array can be made it so that it is dynamically allocated memory, through the use of pointer *arr. An additional code is that the array can be deleted afterwards using the delete keyword.

QUESTION:
b) Define a simple grammar rule for a while loop in a programming language.

ANSWER (with reference to Swift Programming Language and C):

```
while_statement ::= 'while' '(' condition-list ')' embedded_statement
condition-list ::= condition | condition ‘,’ condition-list
condition ::= expression
embedded_statement ::= assign_statement | while_statement | if_statement
assign_statement ::= variable '=' expression
if_statement ::= ‘if’ ‘(‘ condition-list ‘)’ ‘{‘ embedded_statement ‘}’
```

It is to be noted that embedded_statement can have other types of statements other than assigning, while loop and if else statements.

An example while loop using the grammar rule:

```
x = 1
while (x < 5) {
   printf(“x is %d\n”, x);
   x ++;
}
```

QUESTION:
c) Define a grammar rule for a function declaration in a programming language, including parameters and return type.

ANSWER:
This is based on C functions.

```
function declaration ::= return_type identifier ‘(‘ parameter_list ‘)’ ‘;’
return_type ::= ‘void’ | data_type
data_type ::= ‘int’ | ‘float’ | ‘bool’ | ‘char’ | ‘string’
identifier ::= [a-zA-Z_][a-zA-Z0-9_]*
parameter_list ::= parameter | parameter ‘,’  parameter_list
parameter ::= data_type identifier
```

It is to be noted that data type can be any valid data type, including defining new data types. Identifier is any valid name, e.g. variable name like ‘name’ or ‘age’, or function name such as ‘addFunc’ or ‘main’.

A function in C also has a function body, but is not stated here.

An example C function using this grammar rule:

```
int subtract (int a, int b)
```

QUESTION:
d) Define a BNF grammar rule for a function call statement that supports passing parameters.

ANSWER:
<function_call> ::= <identifier> **(** <argument_list> **)**
<identifier> ::= **[a-zA-Z_][a-zA-Z0-9_]***
<argument_list> ::= <expression> | <expression> **,** <argument_list>
<expression> ::= <variable> | <constant> | <arithmetic> | <boolean_expression>

ASSESSMENT A2

QUESTION:

a) Tokens generated by lexer for following code:

```
#include <iostream>
int main() {
int num1 = 42;
double pi = 3.14159;
if (num1 > 0) {
std::cout << "Positive number" << std::endl;
} else {
std::cout << "Non-positive number" << std::endl;
}
return 0;
}
```

ANSWER:

<token, lexeme> pairs

1. <#include, #include>
2. <<iostream>, <iostream>>
3. <id, int>
4. <id, main>
5. <lparen, (>
6. <rparen,)>
7. <lbrace, {>
8. <id, int>
9. <id, num1>
10. <assign, =>
11. <integer, 42>
12. <semi, ;>
13. <id, double>
14. <id, pi>
15. <assign, =>
16. <double, 3.14159>
17. <semi, ;>
18. <if, if>
19. <lparen, (>
20. <id, num1>
21. <relation_op, >>
22. <integer, 0>
23. <rparen,)>
24. <lbrace, {>
25. <namespace, std>
26. <scope_op , ::>
27. <id, cout>
28. <left_shift_op , <<>
29. <string, “Positive number”>
30. <left_shift_op , << >
31. <namespace, std>
32. <scope_op , ::>
33. <id, endl>

34. <semi, ;>
35. <rbrace, }>
36. <else, else>
37. <lbrace, {>
38. <namespace, std>
39. <scope_op , ::>
40. <id, cout>
41. <left_shift_op , <<>
42. <string, “Non-positive number”>
43. <left_shift_op , <<>
44. <namespace, std>
45. <scope_op , ::>
46. <id, endl>
47. <semi, ;>
48. <rbrace, }>
49. <return, return>
50. <integer, 0>
51. <semi, ;>
52. <rbrace, }>

Total: 52 tokens

QUESTION

b) Consider the following code snippet and provide an overview of the parsing process.

```
int main() {
int x = 5;
if (x > 0) {
std::cout << "Positive number" << std::endl;
}
return 0;
}
```

ANSWER

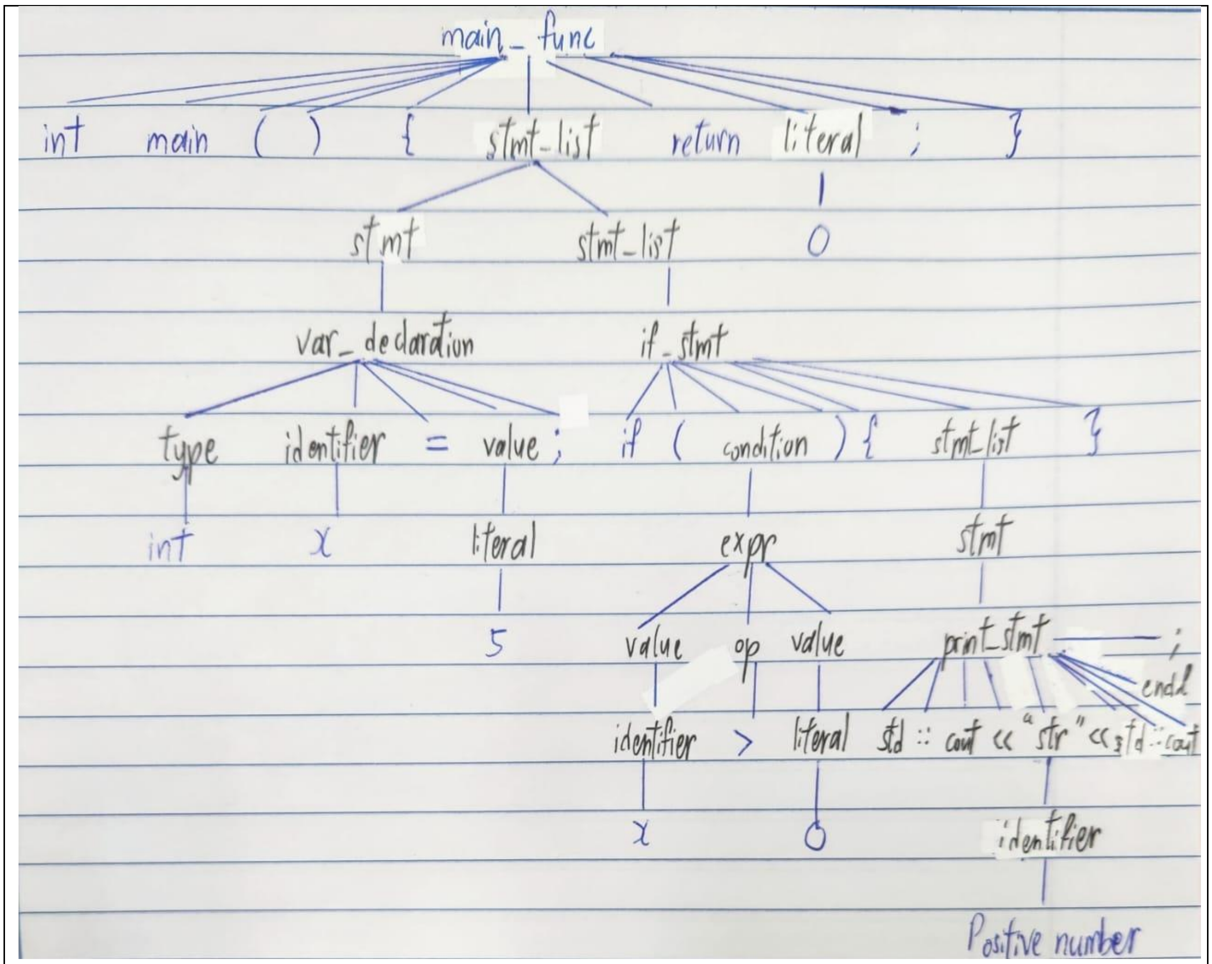
Step 1: identifying all the tokens

1. int
2. main
3. (
4.)
5. {
6. int
7. x
8. =
9. 5
10. ;
11. if
12. (
13. x
14. >

15. 0
16.)
17. {
18. std
19. ::
20. cout
21. <<
22. “Positive number”
23. <<
24. std
25. ::
26. endl
27. ;
28. }
29. return
30. 0
31. ;
32. }

Step 2: Grammar rules
<main_func> ::= **int main () {** <stmt_list> **return** <literal>**}**
<stmt_list> ::= <stmt> | <stmt> <stmt_list>
<stmt> ::= <var_declaration> | <if_stmt> | <print_stmt>
<var_declaration> ::= <type> <identifier> **=** <value> **;**
<type> ::= **int** | **bool** | **double** | **float**
<identifier> ::= **[a-zA-Z_][a-zA-Z0-9_]***
<if_stmt> ::= **if (**<condition>**) {**<stmt_list>**}**
<condition> ::= <expr>
<expr> ::= <value> <op> <value>
<op> ::= **==** | **!=** | **<** | **>** | **<=** | **>=**
<value> ::= <identifier> | <literal>
<literal> ::= <int_lit>
<int_lit> ::= <digit>+
<digit> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
<print_stmt> ::= **std::cout <<** ”<str>**” <<std::endl;**
<str> ::= <identifier>

Step 3: Draw a Parser tree (diagram)



QUESTION

c) Consider the following C++ code snippet with nested structures and function calls:

```
#include <iostream>
struct Point {
int x;
int y;
};
void printPoint(const Point& p) {
std::cout << "(" << p.x << ", " << p.y << ")";
}
int main() {
Point origin = {0, 0};
printPoint(origin);
if (origin.x == 0 && origin.y == 0) {
std::cout << "\nOrigin detected!" << std::endl;
}
return 0;
}
```

Define a BNF grammar rule that encompasses the syntactic elements present in this code, including struct declaration, function definition, and if statements.

ANSWER

```
<program> ::= <include_stmt> <struct_declaration> <func_def> <main_func>
<include_stmt> ::= #include <<library_name>>
<library_name> ::= <identifier>
<identifier> ::= [a-zA-Z_][a-zA-Z0-9_]*
<struct_declaration> ::= struct <identifer> {<var_declaration>}
<var_declaration> ::= <type> <identifier> ;
<type> ::= int | bool | double | float | const | <user_defined>
<user_defined> ::= <identifier>

<function_def> ::= <return_type> <identifier> (<parameter_list>) {<stmt_list>}
<return_type> ::= void | <type>
<parameter_list> ::= <parameter> | <parameter> , <parameter_list>
<parameter> ::= <type> <identifier>
<stmt_list> ::= <stmt> | <stmt> <stmt>
<stmt> ::= <declaration> | <assign> | <if_stmt> | <func_call> | <print_stmt>
<declaration> ::= <parameter> = <value> ;
<assign> ::= <identifier> = <value>;
<if_stmt> ::= if (<condition>) {<stmt_list>}
<condition> ::= <expr>
<expr> ::= <value> <op> <value>
<op> ::= == | != | < | > | <= | >=
<value> ::= <identifier> | <literal>
<literal> ::= <int_lit>
<int_lit> ::= <digit>+
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<func_call> ::= <identifier>(<identifier>);
<print_stmt> ::= std::cout << ”<str>” <<std::endl;
<str> ::= <identifier>

<main_func> ::= int main () { <stmt_list> return <literal>}
```

ASSESSMENT A3

QUESTION

Create a parser for a simplified markup language that supports tags, attributes, and nested elements. Define a BNF grammar rule for this markup language using C++.

ANSWER

Part 1: Parser for HTML

```
#include <iostream>
#include <vector> //dynamic array
#include <regex> //use smatch
#include <stack>
#include <sstream>

using namespace std;

// to extract HTML tags from given string - using str ref &
vector<string> extractHTMLTags(const string& input) {
   vector<string> tags; //to store tags dynamically - cannot predict size
   regex tagPattern("</*([a-zA-Z]+)[^>]*([^<]*?)>"); // Regular expression to match HTML tags
   smatch tagMatch; //store regex - can perform search

   string filteredHtmlContent;
   //init iterator - use sregex_iterator (begin of input, end of input, iterate over with tagPattern)
   //until init reaches end of matches (from tagPattern)
   for (auto init = sregex_iterator(input.begin(), input.end(), tagPattern); init != sregex_iterator();
++init) {
      tags.push_back(init->str()); // store matched tags in vector
   }

   return tags;
}

//check if HTML tags are nested - dyanmic arr to track tags
bool isTagsNested(const vector<string>& tags) {
   stack<string> tagStack; //track opening tags (i.e. no /)
   for (const string& tag : tags) { //goes through each tag in tags (from extract HTML tags func)
      if (tag[1] != '/') { // check if it's an opening tag
         string tagName = tag.substr(1, tag.find(' ') - 1); // get tag name (e.g. <h1 header="..."> ==
h1)
         tagStack.push(tagName); // push tag onto stack
      } else { // It's a closing tag
         string closingTagName = tag.substr(2, tag.find('>') - 2); // get tag name from closing tag (e.g.
</h1> == h1)
         if (!tagStack.empty() && tagStack.top() == closingTagName) { // if stack is not empty and top
of stack matches closing tag
            tagStack.pop(); // pop the opening tag from stack
         } else {
            return false; // tags are not nested properly
         }
```

```
        }
    }
    return tagStack.empty(); // if stack is empty, tags are properly nested
}

//find and return attributes from HTML tags - dynamic arr with tags
vector<string> extractAttributes(const string& tag) {
    vector<string> attributeTokens; //dynamic arr to store attributes
    stringstream ss(tag.substr(1, tag.find('>') - 1));

    string token;
    while (ss >> token) { //loop through each token from stringstream
        size_t pos = token.find('='); //find occurrence for = (string::npos == no occuring =)
        if (pos != string::npos) { //if there's =
            attributeTokens.push_back(token.substr(0, pos)); // get attribute name
        }
    }
    return attributeTokens;
}

int main() {
    //html input
    string rawHtmlContent = "<head style='display:none';><div1 width='100px'><div
height='200px'></div></div1></head>";

    vector<string> tags = extractHTMLTags(rawHtmlContent); //get tags

    for (const auto& tag : tags) { //check each tag in html str
        if (tag[1] != '/') { // check if it's an opening tag
            vector<string> attributes = extractAttributes(tag); // get attributes from tag
            cout << "Tag: " << tag.substr(1, tag.find('>') - 1) << endl; // print tag name
            for (const auto& attribute : attributes) {
                cout << "Attribute: " << attribute << endl; // print attribute
            }
        }
    }

    if (isTagsNested(tags)) { //check if tags are nested properly
        cout << "HTML tags are nested." << endl;
    } else {
        cout << "HTML tags are not nested." << endl;
    }

    return 0;
}
```

Part 2: BNF Grammar Rule for HTML
<html_content> ::= **<**<tag>**>** <html_body> **</**<tag>**>**
<tag> ::= <identifer> <attribute>
<identifier> ::= **[a-zA-Z_][a-zA-Z0-9_]***

```
<attribute> ::= <identifier> = <value>
<value> ::= <identifier> | <literal>
<literal> ::= <int_lit> | <special_char>
<int_lit> ::= <digit>+
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<special_char> ::= . | # | ! | @ | \ | “ | “ …
<html_body> ::= <identifier>
```

ASSESSMENT A4

QUESTION

What is Binding in C++? Explain the concepts of early and late binding in C++ and how they differ from each other. Explain with reasonable examples.

ANSWER

Binding in C++ means to convert variables and names into memory addresses. It is done for functions as well, where the corresponding function is called correctly according to the binding done by compiler. In relation to that, binding is done either during compilation or at runtime.

Early binding in C++ is done during compile time. It is also called as static binding. If the compiler knows which function to call at compile time, compiler uses early binding. Below is an example code of how early binding works. It defines a class 'Piano' and a function called 'musician()'. If called, it will print 'Beethoven'. This function call is resolved at compile time as the compiler knows which function to implement based on the object declared (myPiano and musician()). Thus, it uses early binding and calls function musician(), and 'Beethoven' is printed.

```
#include <iostream>
using namespace std;

class Piano {
   public:
      void musician(){
         cout << "Beethoven" << endl;
      }
};

int main(){
   Piano myMusician;
   myMusician.musician(); //early binding
   return 0;
}
```

Late binding is done during run time. It is also called dynamic binding. This is when the compiler does not know which function to call at compile time, or when there are indirect function calls, causing functions to be called during run time instead. This can be done via function pointers, where the memory address of a function is stored, or virtual functions. Below is an example of late binding using function pointers and virtual functions. Here, there is a parent class called Instrument, and can be polymorphed by class Piano. Function overriding takes place here, from function musician(). Note that class Instrument is declared as virtual, allowing for dynamic polymorphism. Then, there is a function pointer, Instrument*. As it stores an address and does not directly indicate which function to call, the compiler will refer to the address of the object it is referring to, in this case a 'Piano' object, and then calls the virtual function through the pointer. Thus, the result is "Beethoven".

```
#include <iostream>
using namespace std;

class Instrument {
   public:
      virtual void musician(){
            cout << "Some musician" << endl;
        }
};

class Piano : public Instrument {
    public:
        void musician() override {
            cout << "Beethoven" << endl;
        }
};

int main(){
    Instrument* myMusician = new Piano();
    myMusician->musician(); //late binding
    delete myMusician;
    return 0;
}
```

Some notable differences between early binding and late binding include performance, flexibility, and error checking.

For early binding, it can be faster than late binding as function calls are resolved at compile time instead of runtime, compared to late binding. Early binding can also catch compile time errors, compared to late binding which can cause runtime errors instead, making it harder to identify and fix. However, early binding is not as flexible as late binding as no dynamic behaviour is allowed, due to compiling everything at compile time. Late binding is more flexible in this case.

ASSESSMENT A5

QUESTION

Define the concept of type inference in Haskell. Provide an example illustrating type inference.

ANSWER

Type inference is where a concrete type within a type system is deduced. It allows for implicit variables within an expression. This also leads to less verbose code, or less explicit code.

An example of type inference is subtracting two numbers, not knowing their exact type. They could be integer, float, double, etc. That is where type inference comes in, where their type is inferred, and will be deduced using variables. We will also add it to a list, which we also need to infer the type for that list's elements.

getAge a b c = (a – b) : c

The function subtract takes in a b and c. Thus, we can first deduce that a, b, and c have their own types.

a :: v, b :: w, c :: x

Next, we have to infer the operator '-' and its type. In this case, it will take in 2 arguments and give 1 result, which should be the same type as its arguments. It should also only take in a value where it belongs in the class type of 'Num', as '-' is an arithmetic operator.

(+) :: (Num d) => y => y -> y

Before we simplify anything, we must deduce the list's type where subtracted a and b will be added to. Logically, to add anything to a list, we will take in an element, the list, and return the updated list by adding the element to the list using the syntactic sugar colon (:). The type must stay the same, and it must be noted that a list's type is static and does not change.

(:) :: z -> [z] -> [z]

Now, we can start inferring what a and b should be. Referring to the '-' operator and its deduced type logic, it should take in 2 elements of type 'y'. Thus, we can infer that a = y and b = y.
Next, as (a-b) is the element to be added to list c, we can deduce that y = c, as y is to be added to the list of z. Then, we can deduce that the element z is c. By extension, the list of z, [z] is c.

To recap:
a :: y, b :: y, c :: [z]

Also, as we deduced that y = c, so **c :: [y].**

Thus, our final type inference for this function is:

getAge :: (Num y) => y -> y -> [y] -> [y]

It is to be noted that the last [y] refers to the final result of the updated list.

ASSESSMENT A6

QUESTION

Why is type checking important in a programming language? Explain with programming examples in any language.

ANSWER

Type checking is important to ensure that the correct data types are used for specific functions or variables. The compiler checks the program for well-formedness constraints, or it checks abstract syntax trees if it is well-typed or not. This means that the compiler can check if the wrong data type is being used, and prevent subsequent errors that arise from incompatible type usage. These are called type rules for a language. A language can be either statically typed or dynamically typed. A statically typed language is where type checking is performed at compile time, while a dynamically typed language is performed at runtime. A statically typed language such as Haskell has its variables determined at compile time and cannot be changed unless compiled again. An example of how Haskell's type checking is important is shown below:

```
addNumbersList :: [Int] -> Int
addNumbersList xs = sum xs

total = addNumbersList [10,20,"30"]
```

In the example above, there is a function where it takes in a list of integers and returns an integer. In the variable 'total', a list passing in 2 integers and 1 string will result in a compile error, as 'Int' cannot be matched with '[Char]'. This results in type checking being successful as integers cannot be added with strings, ensuring that no unexpected or unwanted behaviour occurs. This can also allow for faster detection of errors, as Haskell's compiler will explicitly mention the incompatible types being used. If Haskell were a dynamically typed language, there wouldn't be an explicit error as this error would become a runtime error instead, and error detection becomes less efficient. In addition, IDEs can detect such errors and suggest alternatives or auto completion before the code is compiled.

ASSESSMENT A7

QUESTION

Explain the concept of operational semantics in programming languages. How does it differ from denotational semantics? Explain how operational semantics rules could be defined for a simple loop statement.

ANSWER

Semantics for a certain programming language mean how each language's structure can reflect the behaviour of programs written in that language. One such approach to giving a program a semantics are operational semantics, which uses relational approaches to represent the program's behaviour directly. It involves inductively defined relations to show the values produced, or program states between transitions. This method is good for gaining a better understanding of a program's behaviour and is more intuitive compared to denotational semantics. Operational semantics have two types, which are big step and small step operational semantics. Big step operational semantics describes how an expression can produce a value. Essentially it is written as E ⇓ V, where E evaluates to value V.

In comparison, denotational semantics involve mapping programs to mathematical structures like functions or sets with representative values after calculation. It mainly focuses on being abstract and the mathematical relationship between mathematical objects, making it more suitable for formal analysis.

A simple loop statement such as a while loop can be represented using both small step and big step operational semantics. Below is a simple while loop statement.

```
while (condition){
   loop_body
}
```

A loop statement consists of a Boolean condition that will evaluate some expression or variable to either true or false, and the loop body. If the condition evaluates to true, then the loop body is to be executed, until the condition becomes false. Once the condition evaluates to false, the loop body is skipped. Thus, there should be two rules, one if the condition is true and one if the condition is false.

$$(1)\ \frac{P\ \Downarrow \text{true} \quad S\ \Downarrow S' \quad while\ P\ do\ S' \Downarrow S''}{while\ P\ do\ S' \Downarrow S''} whiletrue$$

$$(2)\ \frac{P\ \Downarrow \text{false}}{while\ P\ do\ S' \Downarrow\ S} whilefalse$$

For the semantic rules above, P refers to the condition which can be either true or false, S refers to the loop's body statement, S' is the loop's body statement after executing it once, and S'' refers to the end state of the entire loop that can happen more than once. In (1), if P evaluates to true, then the body is executed once and evaluates to a new state, S'. Then, as P is true, it will execute S' and evaluate to a new state of the program, S''. In (2), if P evaluates to false, then it will stop the loop and program transitions to the final state of S, which is the initial state before the loop began. The loop terminates here.

A loop can also be represented in small step semantic rules, which is shown below:

$$(3)\ \frac{P \rightarrow \text{true}}{while\ P\ do\ S \rightarrow\ S'; while\ P\ do\ S} whiletrue$$

$$(4)\ \frac{P \rightarrow \text{false}}{while\ P\ do\ S \rightarrow\ skip} while false$$

For the semantic rules above, P is the condition which can be true or false, S is the loop body's initial state, and S' is the body's statement after execution. In (3), if P is true, then it will execute the loop's body statement, and transition to a new state of S'. The loop will continue executing the body's statement. Then, in (4), if P is false, then it will terminate with 'skip'.