# HW3 Supplementary

Our test cases will totally be based on this document.

## *Variable Declaration and Initialization*

- **int**

```
int a;       /* Make sure to initialize it to zero */
int a = 2;   /* Initialize with INT_CONST */
int a = 2.0; /* Initialize with FLOAT_CONST, but do casting */

/* Initialize with expressions
 * Be sure also do casting
 */
int a = 4 + b;        /* Arithmetic operations */
int a = sqrt(d);      /* Function call return value */
int a = sqrt(c) - 5;
```

- **float**

```
float a;       /* make sure to initialize it to zero */
float a = 2;   /* Initialize with FLOAT_CONST */
float a = 2.0; /* Initialize with INT_CONST, but do casting */

/* Initialize with expressions
 * Be sure also do casting
 */
float a = 4.5 + b;      /* Arithmetic operations */
float a = sqrt(d);      /* Function call return value */
float a = sqrt(c) - 5;
```

- **string**

```
string a;                /* This will not be tested */
string a = "Compiler";   /* Initialize with STR_CONST */
```

- **bool**

  Since Jasmin do not have bool type, take it as int with value 0 or 1

```
bool a = TRUE;   /* Initialize with TRUE, FALSE */
```

- **Check Undeclared/ Redeclared variable**

This check should be applied to all types of variable

```
/* Raise semantic error! Undeclared variable */
int a = b + 5; /* b is a undeclared varaible */

/* Raise semantic error! Redeclared variable */
int a = 6;
int a = b + 5; /* a is a redeclared varaible */
```

## *Arithmetic operations*

- Only **int** and **float** should be considered to do arithmetic operations, **other types will not be tested**
- The arithmetic operations include:

```
++ -- / * % + - += -= *= /= %=
```

- You should handle the **precedence** of operands, for example:

```
int b = 5;
int a = 1 + b * 5;   /* a should be 26, not 30*/
```

- You should do **type casting** if needed

```
float b = 5.2;
int a = 1 + b * 5;
/* 1 + 5.2 * 5 (cast 5 to 5.0)
 * => 1 + 26.0 (cast 1 to 1.0)
 * => 27.0     (cast 27.0 to 27)
 * => 27
 */
```

- **For % and %=,** two operands can only be **int**, if not then raise a semantic error

```
/* The operands are int, valid */
int a = 25 % 5;
float a = 25 % 5;

/* The followings are invalid,
 * dump a semantic error message
 */
int a = 25.5 % 5;
int a = 25 % 5.2;
int a = 25.5 % 5.2;
```

- Detect divide by zero

```
/* Raise divide by zero semantic error */
int a = 10 / 0;
int b = 0;
a / b;          /* b is 0 */
```

## *Relational operations*

- Only **int** and **float** should be considered to do relational operations
- The operands will be the same type

```
float a = 5.2;
float b = 6.9;
int c = 5;
int d = 6;

/* int to int */
a == b

/* float to float */
c != d

/* No need to consider
 * float to int or
 * int to float/
a <= c
d >= 5
```

- The result of a relational operation should be **bool** (TRUE, FALSE)

## *If….else statement*

- You should consider the following if...else statement:

```
/* if */
if (<relational_operation>) {
  ...
}

/* if...else.. */
if (<relational_operation>) {
 ...
} else {
 ...
}

/* if...else if...else */
if (<relational_operation>) {
 ...
} else if (<relational_operation>) {
 ...
} else if (<relational_operation>) {
 ...
} else {
 ...
}

/* if...else if... */
if (<relational_operation>) {
 ...
} else if (<relational_operation>) {
 ...
} else if (<relational_operation>) {
 ...
}

/* nested, includes the above */
if (<relational_operation>) {
 ...
} else if {
  if (<relational_operation>) {
    ...
  } else {
    ...
  }
} else {
 ...
}
```

- You should only consider the simplest relational operations, **&&, ||, ! will not be tested**

```
/* int to int */
int a = 10;
int b = 10;
if (a <= b) {
 ...
}

/* float to float */
float a = 5.2;
float b = 6.9;
if (a == b) {
 ...
}


/* We will not test this
 * &&, ||, !/
if (a == 5 && b ==6) {
 ...
}
```

### *while statement*

- You should consider the following while statement:

```
/* while */
while (<relational_operation>) {
  ...
}
```

- You should only consider the simplest relational operations, **&&, ||, ! will not be tested** (same with if..else statement)

```
/* int to int */
int a = 10;
int b = 10;
if (a <= b) {
 ...
}

/* float to float */
float a = 5.2;
float b = 6.9;
if (a == b) {
 ...
}


/* No need to consider
 * &&, ||, !/
if (a == 5 && b ==6) {
 ...
}
```

## *Functions and Function Call*

- Only **variables** and **constants of all types** will be tested as the **formal parameter**
- **Every function can only be declared or defined once** if the function is declared or defined for the second time, raise a semantic error
- Jasmin can't do function declaration, thus you don't need to generate any assembly code in the function declaration, but you should insert the function into the symbol table for further function definition check

```
/* declare */
int a (<formal_parameter>);

/* define */
int b (<formal_parameter>) {
 ...
}

/* forward declaration, declare first then define */
void c (<formal_parameter>);
 ...

void c (<formal_parameter>) {
 ...
}

/* define first then declare */
void d (<formal_parameter>) {
 ...
}
 ...

void d (<formal_parameter>);


/* Re-declared error! declare for second time */
float e (<formal_parameter>);
 ...

float e (<formal_parameter>);

/* This test case will not be tested
 * since this is not listed in our grading list
 * Re-defined error!
```

```
 * define for second time
 */
void d (<formal_parameter>) {
 ...
}
 ...

void d (<formal_parameter>) {
 ...
}
```

- Both declaration and definition should have the same **return type**, **formal parameter type** and the **number of formal parameter**

```
/* The function
 * return type,
 * formal parameter type and
 * the number of formal parameter should be the same
 */
void d (int a, int b);
 ...

void d (int a, int b) {
 ...
}

/* Raise semantic error
 * the type of formal parameter is not the same
 * error message: function formal parameter is not the same
 */
void d (int a);
 ...

void d (float a) {
 ...
}

/* Raise semantic error
 * the number of formal parameter is not the same
 * error message: function formal parameter is not the same
 */
void d (int a);
 ...

void d (int a, int c) {
 ...
}
```

```
/* Raise semantic error
 * the return type is not the same
 * error message: function return type is not the same
 */
void d (int a);
 ...

int d (int a) {
 ...
}

/* Raise semantic error
 * the return type and the formal parameter is not the same
 * error message:
 *   1. function return type is not the same
 *   2. function formal parameter is not the same
 */
void d (int a);
 ...

int d (int a, int c) {
 ...
}
```

- Only **int, float, bool and void** should be considered as **function return type**
- Every function definition would be given a return statement at the end, even if it is a void function

```
/* int function, must return a int */
int foo1 () {
  ...
  return 1;
}
/* float function, must return a float */
float foo2 () {
  ...
  return 1.5;
}

/* void function, must return */
void foo2 () {
  ...
  return;
}
```

```
/* Need not to consider
 * the given function without a return statement */
```

- Only **int, float variable** and **constant** will be tested as return value
- **Functions should be declared or defined before main function, due to the limitation of Jasmin.**
- Check if the function has been **defined** before invocation.

```
/* defined before invocation */
int foo1 () {
  ...
}

void main() {
  foo1();
}

/* Raise semantic error!
 * error message: function formal parameter is not the same */
int foo1 () {
  ...
}

void main() {
  foo1(20);
}

/* Raise semantic error! Undeclared functions */
void main() {
  foo1();
}

/* This test case will not be tested */
void foo3 (){
 ...
}

void main() {
 int a = foo3(); /* Assign a variable with void function*/
 a = foo3() + 2; /* Do Arithmetic operations */
}

/* This test case will not be tested
 * Declared but not defined!
 * Since Jasmin can't do function declaration,
```

```
 * thus the function must be defined before invocation */
float foo2 ();

void main() {
 foo2();
}

float foo2() {
 ...
}
```

- You should check the **return type**, **formal parameter type**, the **number of formal parameter type** when invoking the function call

## *Print functions*

- The print function will only take **int, float and string** variables and constants

```
/* defined before invocation */
int a;
print(1);
print(2.5);
print("HELLO");
print(a);       /* Print out 0 */

/* print with arithmetic operations will not be tested */
print(a + b);
```

## *Reminders*

- Semantic error should keep on parsing input file
- Syntactic error should stop parsing input file
- **Do not generate .j file if any of the error occur**