Compiler Lab
Specification

Simplified C

Tobias Kahse, Frank Steiler

May 12, 2014

Contents

1 Introduction

This document specifies the programming language *Simplified C*. As its name indicates the syntax is based on C, but the range of functions is reduced.

The source code is parsed by ANTLRv4, converted into JVM assembly language code, and finally compiled to Java byte code using Jasmin. However, the compiler is designed to easily exchange the assembly code instructions and thus, compiling source code for various systems is possible.

2 General Description

As mentioned in the introduction the range of functions of the programming language *Simplified C* is reduced in comparison with C. *Simplified C* supports the following functions:

- Usage of (unsigned) integer and boolean data types
- Conversion of integer to boolean and boolean to integer values
- Variable declaration and assignment
- Distinction between global and local variables (scopes)
- Basic integer and boolean calculations
- While and doWhile loops
- Case distinction with if/then/else
- Printing to the system's standard output
- Functions

Furthermore the compiler will be capable of calculating .limit stack correctly and parsing source code to an abstract class representation.

3 Detailed Description

Within this section all functions of *Simplified C* are described in detail. Sample code snippets are going to specify how the syntax will look like.

3.1 General Source Code Structure

Every *Simplified C* program must provide a main function. This function will be used as an entry point for the program. The main function will always return a boolean value, which indicates whether the execution was successful or not. Global variables have to be defined at the beginning of the source code.

3.2 Output to stdout

Boolean and integer values can be printed to the system's standard output using the function `println`. Boolean values will be printed as either true or false. The usage of `println` is specified in all following code snippets.

3.3 Variables

Simplified C supports two different data types, unsigned integer and boolean. The values can be stored within variables. Each variable has to be declared using the appropriate data type. The syntax for variable declaration, initialization, and assignment can be found in all following code snippets, e.g. Listing 1 on the following page.

3.4 Type Conversion

It is possible to convert integer to boolean and the other way around. To convert a value use the operator `(int)` respectively `(boolean)` in front of the variable which is going to be converted. Keep in mind that the conversion from integer to boolean is lossy.

The integer `0` is converted to the boolean value `false`, every other integer is converted to `true`. The boolean value `true` is converted to `1`, while `false`

is converted to `0`. An example of for the type conversion can be found in listing 1.

Listing 1: Code Sample: Type Conversion

```
boolean main() {
  int a = 2;
    //Assign 2 to a
  boolean b = (boolean)a;
    //Convert a to boolean; b = true

  a = (int)b;
    //Convert b to int; a = 1

  println(a);
    //Print 1 to stdout
  println(b);
    //Print true to stdout

  return true;
    //Returns true (successful execution)
}
```

3.5 Basic Calculations

Simplified C supports the following basic integer and boolean operations:

- Addition
- Subtraction
- Multiplication
- Division
- Logical AND

- Logical OR

The result of each basic calculation can be assigned to a variable of the appropriate type. Furthermore function parameters can be the result of basic calculations.

For all calculations the known precedence rules for integer and boolean arithmetic (e.g. multiplication and division first, then addition and subtraction) apply. An example for basic calculations can be found in listing 2.

Listing 2: Code Sample: Basic Calculations

```
boolean main() {
  int a = 5;
    //Assign 5 to the variable a
  int b = 2;
    //Assign 2 to the variable b
  int c = a + b;
    //Assign the sum of a and b to c; c = 7
  int d = a * c / b - b;
    //Assign the calculation result to d; d = ((5 * 7)
   /2)-2 = 15
  boolean e = true;
    //Assign boolean value true to e
  boolean f = e && false;
    //Assign e AND false; f = false

  println(d);
    //Print 15 to stdout
  println(e);
    //Print true to stdout
  println(f);
    //Print false to stdout

  return true;
    //Returns true (successful execution)
```

```
}
```

3.6 Case Distinction

For case distinctions the commands `if` and `else` can be used. The command `else if` is not supported. The bodies of the cases have to be surrounded by curly braces. This is done to avoid the Dangling-Else problem which could occur during the parsing process of the source code. Furthermore case distinctions can be cascaded infinitely. The condition can be composed of every supported boolean resulting calculation. Integer values can be compared by using comparison operators. Only two integers can be compared at the same time. The following comparison operators can be used:

- Equals (`==`)
- Unequal (`!=`)
- Greater than (`>`)
- Less than (`<`)
- Greater equal (`>=`)
- Less equal (`<=`)

Furthermore multiple boolean values or comparison results can be combined by using the earlier defined boolean operators AND and OR. An example for case distinction can be found in listing 3.

Listing 3: Code Sample: Case Distinction

```
boolean main() {
  int a = 1;
    //Assign 1 to variable a
  boolean b = true;
    //Assign true to variable b
```

```
if (b) {
  //Check if b is true
  if (a == 1) {
    //Check if a is equal to 1
    if (a > 2 || b) {
      //Check if a is greater than two OR b is true
      println(b);
        //Print b to stdout
    }
      //End of if body

    if (a < 1) {
      //Check if a is smaller than 1
      println(a);
        //Unreachable statement
    } else {
      //Body is executed if a is not smaller than 1
      println(a);
        //Print a to stdout
      println(b);
        //Print b to stdout
    }
      //End of else body
  } else {
    //Body is executed if a is not equal to 1
    println(a);
      //Unreachable statement
  }
    //End of else body
}
  //End of if body

//Output:
```

```
  //true
  //1
  //true

  return true;
    //Return true (successful execution)
}
```

3.7 Loops

Simplified C supports two types of loops, `While` and `Do While`. The while loop is, as in C, a head-controlled loop. On the other hand do while is a tail-controlled loop. The control conditions are as defined in section 3.6 on page 6. Furthermore loops can be cascaded infinitely just like case distinctions defined in section 3.6 on page 6. An example for while loops can be found in listing 4 and an example for do while loops can be found in listing 5 on the next page.

Listing 4: Code Sample: While

```
boolean main() {
  int i = 3;
    //Assign 3 to variable i
  int j;
    //Declare variable j

  while (i != 0) {
    //Start while loop; repeat if i is not equal to 0
    j = 2;
      //Assign 2 to j
    while (j > 0) {
      //Start while loop; repeat if j is greater than 0
      j = j - 1;
        //Assign difference j - 1 to j
```

```
      println(i);
        //Print i to stdout
      println(j);
        //Print j to stdout
    }
      //Body of inner loop ends
    i = i - 1;
      //Assign difference i - 1 to i
  }
    //Body of outer loop ends

  //Output:
  //3
  //1
  //3
  //0
  //2
  //1
  //2
  //0
  //1
  //1
  //1
  //0

  return true;
    //Returns true (successful execution)
}
```

Listing 5: Code Sample: Do While

```
boolean main() {
  boolean i = false;
    //Assign false to i
```

```
  int j;
    //Declare variable j

  do {
    //Start (outer) doWhile loop
    j = 0;
      //Assign 0 to j
    do {
      //Start (inner) doWhile loop
      println(j);
        //Print j to stdout
      j = j + 1;
        //Assign the sum j + 1 to j
    } while(j < 5);
      //(Inner) loop condition; Repeat if j is smaller
   than 5
  } while(i);
    //(Outer) Loop condition; Repeat if i is true

  //Output:
  //0
  //1
  //2
  //3
  //4


  return true;
    //Returns true (successful execution)
}
```

3.8 Functions

Each function consists of a function head and a function body. The function head starts with the data type of the return value. If the function is not supposed to have any return value the the key word `void` is used. The data type is followed by the function name, and a list of parameters in parenthesis. It is also possible that this list is empty.

The function body is surrounded by curly braces, consists of a list of variable declarations, followed by a list of statements, and has to end with a return statement. Nevertheless, return statements can also occur within the body. However, the last statement of a function always has to be a return statement. This does not apply to functions without return value. Although these can contain return statements, they do not have to. Every function can be called within a function body. This includes the function itself (recursion). An example for functions can be found in listing 6.

Listing 6: Code Sample: Functions

```
boolean main() {
  int r;
    //Declare variable r

  r = faculty(5);
    //Call faculty with the parameter 5
    //and assign return value to r

  println(r);
    //Print r to stdout; output: 120
  emptyLine();
    //Print empty line to stdout

  return true;
    //Return true (successful execution)
}
```

```
int faculty(int n) {
  //Declare function faculty
  //faculty returns an int and takes
  //one int parameter

  if (n == 0 || n == 1) {
    //Check if n is equal to 0 OR n is equal to 1
    return 1;
      //Return 1
  } else {
    //Execute body if n is neither 0 nor 1
    return n * faculty(n-1);
      //Return the product of n and the faculty of n-1
  }
    //End of else body
  return 0;
}
  //End of function body

void emptyLine() {
  //Declare function emptyLine
  //emptyLine has no return value and
  //does not take any parameters
  println();
    //Print an empty line
  return;
    //Return from function
}
  //End of function body
```

3.9 Scopes

In *Simplified C* a distinction between global and local variables is made. As discussed in section 3.1 on page 3 global variables have to be defined at the beginning of the source code and can be used in every function. On top of that functions can overwrite global variables locally and have their own local variables, which are only accessible by the function. After a global variable was overwritten locally, the variable cannot be accessed within the function any more. An example for scopes can be found in listing 7.

Listing 7: Code Sample: Scopes

```
int i = 5;
  //Assign 5 to (global) variable i

boolean main() {
  int i = 1;
    //Assign 1 to (local) variable i

  helper();

  println(i);
    //Print (local) variable i to stdout; output = 1

  return true;
    //Returns true (successful execution)
}

void helper() {
  println(i);
    //Print (global) variable i to stdout; output = 5
  return;
}
```

3.10 Calculation of .limit stack

In JVM assembly code it is required to calculate the maximum stack size for each function. This is done to use the memory most efficiently by ensuring that no memory is allocated unnecessarily. The compiler for *Simplified C* will set the maximum stack size to the smallest possible amount.

3.11 Visitor

The compiler will use a visitor function to parse the source code. This function converts the parsed tokens into an abstract class representation. The representation consists of classes, which can be used to generate JVM assembly code, which can than be compiled to Java byte code using Jasmin.

Converting the tokens into an abstract class representation rather than a direct conversion into assembly code provides greater flexibility. This enables the user to apply post-processing, or to switch between different assembly language libraries. Concluding the compiler can be extended fairly easy.

Listings