R1 Compiler Project

Copy your `S1.java` to `R1.java`. Then replace every occurrence of "`S1`" in `R1.java` with "`R1`".

Be sure your name is in `R1.java` on line 10.

Modify `R1.java` so that it meets the specifications for R1 described below.

Compile your R1 compiler with

javac R1.java

Compile `S1.s` (which is in the J1 Software Package) with your R1 compiler with

java R1 S1

Assemble the output file `S1.a` created by your R1 compiler with

a S1.a

Finally run the executable program in `S1.e` created by the assembler with

e S1 /c

Submit R1.java, S1.a and S1.<*family name*>.log (the log file that the e program creates when it runs s1.e) in the form requested. If files requested, do not submit a ZIP FILE.

15.1 SPECIFICATIONS FOR R1

Your R1 compiler should compile the program in S1.s. Your Java code should correspond to the translation grammar in the file R1.tg (in the software package).

See specification of code generator methods on the following page.

The translation grammar for R1 differs from the translation gramar for S1 in two significant ways. First, it does not directly call the emitInstruction() method in the code generator. emitInstruction() is now a private method in the code generator. For example, to generate code for the assignment statement, the R1 parser calls the assign() method in the code generator (see line 32 in Fig. 19.6) which, in turn, calls emitInstruction(). Second, the factor() method does not generate any code. It simply returns a symbol table index. The code to load a factor into the ac register is emitted by either the add() method in termList() or the mult() method in factorList().

The methods in the R1 code generator that are not in the S1 code generator are

public void directive(String d)

Outputs the directive d to the assembly language file. The program() method in the parser calls the directive() method with

```
codeGen.directive("!register");
```

to output the !register directive (see line 5 in Fig. 19.6). This directive causes the a and e programs to reconfigure to the register instruction set.

public void assign(int left, int expVal)

Outputs the ld-st sequence for an assignment statement by making calls to emitInstruction() (see line 26 in Fig. 19.6).

public void println(int expVal)

Outputs the ld-dout-ldc-aout sequence for a println statement by making calls to emitInstruction() (see line 36 in Fig. 19.6).

public void add(int left, int right)

Outputs the ld-add-st sequence for an add operation by making calls to emitInstruction() (see Fig. 19.5a and line 57 in Fig. 19.6)

public void mult(int left, int right)

Outputs the ld-mult-st sequence for a multiplication operation by making calls to emitInstruction() (see line 57 in Fig. 19.6). mult() is similar to add().

private int getTemp()

Generates the next temporary variable from the sequence "@t0", "@t1", "@t2",..., enters it into the symbol table, and returns its symbol table index (see Fig. 19.5b). getTemp() is called by add() and mult() (see Fig 19.5a).

private void emitInstruction(String op, int opndIndex)

The code in this method consists of a call to another emitInstruction() method (the one with two String parameters):

```
emitInstruction(op, symTab.getSymbol(opndIndex));
```

R1 also has a third emitInstruction() method (the one with a single parameter whose type is String).