IN2009 Coursework

Grammar Files (`.sbnf`)

The tasks require you to write a context-free grammar for two variants of a simple programming language. You must write these CFG's in the Simple BNF format, as described at the end of this document. You will find examples in the data folder of the TeenyWeenyIR project. Your CFG's are required to be LL(1). To verify that your grammar is LL(1), and that it is in the correct format, you may use the LL1Check tool provided. For example:

```
java -jar LL1Check.jar data/rse/rse.sbnf
```

(Java version 17 or above is required). Simple grammar transformations should suffice to resolve any problems (eg choice-conflicts or left-recursion).

Token Definitions

You must provide token definitions corresponding to the terminal symbols in your `.sbnf` files. In each case, you must implement your token definitions as the DEFS constant in the relevant Java class. The format is the same as the one used in TeenyWeenyTokens.java, in the TeenyWeenyIR project.

Task 1 [25%]

Define an LL(1) grammar for the language Rse, described below. You will find many examples of Rse programs in the test folders provided. Write your grammar in the file `data/rse/rse.sbnf` and implement the token definitions for your grammar in `src/lang/rse/RseTokens.java`. Note that the prototype RseTokens file includes a definition for an OPERATOR token. You can (and should) keep this token definition and use it when defining your grammar.

Rse Syntax

An LPse program starts with the keyword **`main`** followed by an opening curly bracket, followed in turn by a (possibly empty) sequence of statements, and terminated by a closing curly bracket. A statement is either an assignment-statement, an **if**-statement, a **do-until**-statement, a **`printint`**-statement or a **`printchar`**-statement. Apart from if-statements and do-until-statements, statements must be terminated by a semi-colon. You can determine details of the syntax of each kind of statement from the examples provided in the test folders.

Statements contain expressions built up from identifiers (variable names), integer literals and the following operators: **+**, **-**, *****, **/**, **<**, **<=**, **==**. An identifier is a non-empty sequence of letters, digits, underscores and $-signs; the first character in a variable name cannot be a digit or a $-sign; if the first character is an underscore, there must be at least one additional character. The operators all have the same meaning as in Java when applied to integers (which is the only data type in the Rse language). Expressions must be fully-bracketed. For example, these six expressions are fully-bracketed:

```
123             x * 4           2 - (y * 7)
(y * 7) / 3     ((5))           (1 + (-2 * 3))
```

These three expressions are *not* fully-bracketed:

```
1 + 2 * 3       3 * z + 9       1 + (2 + 3) + 4
```

Task 2 [25%]

Implement a recursive-descent parser for your grammar in `lang.rse.RseParser.java`. You are required to implement your parser by hand (not using a parser generation tool).

Task 3 [20%]

Implement a compiler in `lang.lpse.RseCompiler.java`. Start by copying your parser code, then add code-generation to your parser methods. Your compiler must output valid IR code. Note that the prototype compiler includes an `emit` method, which you should use instead of `System.out.println`. The `main` method has been written so that, if you provide a second file name on the command line, your IR code will be written into that file instead of to the terminal (this allows you to output assembly code directly to a file, without using a shell redirect operator). For example:

```
java lang.rse.RseCompiler add.rse temp.ir
```

Rse Semantics

Rse is untyped. All expressions evaluate to an integer. There are no Booleans. If and do-until-statements treat 0 as false and any other value as true. The comparison operators (**<**, **<=**, **==**) always evaluate to either 0 (false) or 1 (true). Variables are not declared, they are just used. All variables are automatically initialised to zero. In a do-until statement, the body statement is

executed first, then the loop test is evaluated; if the test evaluates to 0 the loop is repeated, otherwise the loop terminates. For example, the following do-until loop outputs three 9's:

```
n = 3;
do {
  printint 9;
  n = n - 1;
} until (n == 0)
```

Task 4 [40%]

The Rfun language is Rse plus function definitions and function calls. A function can be called within an expression, or as a statement (the return value is ignored in the latter case). Here is an example:

```
main { printint fac(5); newline(); }
fundef fac(x)
begin
   if (x < 1) return 1; else return x * fac(x - 1);
end
fundef newline() begin printchar 13; printchar 10; end
```

See the test folders for more examples. Note that it is legal for an Rfun function to terminate without executing a return statement (it returns 0 by default). It is also legal for the main body of an Rfun program to contain return statements (the effect is to halt execution of the program, the return value is ignored).

a) [20%] Define an LL(1) grammar for Rfun in `data/rfun/rfun.sbnf`, token definitions in `lang.rfun.RfunTokens` and implement a parser in `lang.rfun.RfunParser`.

b) [20%] Implement a compiler for Rfun in `lang.rfun.RfunCompiler`.

The Simple BNF Format

Terminal symbols are sequences of upper-case letters and underscores, starting with an upper-case letter.

Non-terminal symbols are sequences of upper and lower-case letters, digits and underscores, starting with an upper-case letter and containing at least one lower-case letter or digit.

Rules have the form *nt* `->` *symbol-sequence* where *nt* is a non-terminal and *symbol-sequence* is a sequence of zero or more terminals and non-terminals, each optionally followed by a Kleene-`*`. Each rule must be written on a separate line. Empty lines and comments are also allowed (comments start with two hyphens `--` and extend to the end of the line).