IN2009 Coursework 2021-22

> *It is assumed that you will develop your coursework solution as an IntelliJ project, starting from the LP2022 project provided, and submit it as a Zip file,* ***created using IntelliJ*** *according to the instructions below. If you plan to develop your solution in some other way, you* ***must*** *make special arrangements in advance: email s.hunt@city.ac.uk no later than* ***21 March 2022****. Submissions in any other format (this includes Zip files created by any other process)* ***will not be marked*** *without prior approval.*

Submission

From the IntelliJ File menu:

> Export ⇒ Project to Zip File…

Use `LP2022_` followed by your student number as the file name. For example, if your student number is 201234567 then you must export your project as:

```
LP2022_201234567.zip
```

Submit this Zip file in Moodle.

Assessment

Your work will primarily be assessed by automated tests for correctness. Grammar files not in the correct format (see below) are untestable and therefore will obtain zero marks. The same applies to uncompilable Java or SSM assembly code.

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 Tutorial 3 solutions. 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/lpse/lpse.sbnf
```

(Java version 17 or above is required). If choice-conflicts are detected you may need to calculate some FIRST, FOLLOW and selection sets to determine the cause (you are *not* required to submit these calculations for assessment). Simple grammar transformations should suffice to resolve any such problems.

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 Tutorial 3 solutions.

Task 1 [25%]

Define an LL(1) grammar for the language LPse, described below. You will find many examples of LPse programs in the test folders provided. Write your grammar in the file `data/lpse/lpse.sbnf` and implement the token definitions for your grammar in `lang.lpse.LPseTokens.java`.

LPse Syntax

An LPse program starts with the keyword **`begin`** and ends with the keyword **`end`**. Between these keywords the body of a program consists of a (possibly empty) sequence of statements. A statement is either an assignment-statement, an if-statement, a while-statement, a **`printint`**-statement or a **`printchar`**-statement. Apart from if-statements and while-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 variable names, integer literals and the following operators: **`+`**, **`-`**, **`*`**, **`/`**, **`<`**, **`<=`**, **`==`**. A variable name 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 LPse 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.lpse.LPseParser.java`. You are required to implement your parser by hand (not using a parser generation tool).

Task 3 [25%]

Implement a compiler in `lang.lpse.LPseCompiler.java`. Start by copying your parser code, then add code-generation to your parser methods. Your compiler must output valid Simple Stack Machine assembly 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 assembly 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.lpse.LPseCompiler add.lpse temp.ssma
```

LPse Semantics

LPse is untyped. All expressions evaluate to an integer. There are no Booleans. If and while-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.

Task 4 [25%]

EITHER

Define a modified LL(1) grammar, parser and compiler for the language variant LPop. This language is the same as LPse except that expressions no longer have to be fully bracketed. Instead, operators associate according to their precedence (as in the BODMAS conventions). Operator precedence is as follows: the lowest precedence operators are the comparison operators (**<**, **<=**, **==**), next in precedence are (**+**, **-**) and the highest precedence operators are (*****, **/**). Operators of equal precedence associate to the left. For example, the following two expressions are equivalent:

```
1 + 2 == x * 7 - 9 < -6 * 12 / y
((1 + 2) == ((x * 7) - 9)) < ((-6 * 12) / y))
```

(Note: this is indeed a strange expression, and one that would not be well-typed in a language like Java. But the LP languages are untyped, and all expressions evaluate to integers, so it is well-defined.) You may use the "one-way street" technique described in [Dos Reis, 2012] to define an expression grammar which incorporates the required precedence rules.

OR

Define a modified LL(1) grammar and parser (but **not** a compiler) for the language variant LPfun. This language is LPse 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:

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

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

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).

References

[Dos Reis, 2012] Dos Reis, Anthony J., *Compiler Construction Using Java, JavaCC, and Yacc*, 2012. (Full text available online via City Library.)