IN2009 Coursework 2022-23 Part 1

Submission

*Submit your solutions as **three separate files** in Moodle:*

1. `lp23s.sbnf`
2. `lp23ef.sbnf`
3. `lp23.sbnf`.

*The file names are case-sensitive and must be **exactly** as specified here. Do **NOT** submit your solutions packaged as a zip, rar, etc.*

You are asked to write formal grammars for three small programming languages. Each language is a backwards-compatible extension of the previous one. The simplest version is *LP23s*. The second language, *LP23ef*, adds an extended syntax for if-statements, plus function definitions. The third and final language, *LP23*, adds support for arrays and an increment operator. The syntax of each language is described informally in the following pages. Your task is to translate these informal descriptions into formal context-free grammars in SBNF format (as used in Tutorial 3, for example). You are provided with a prototype for `lp23.sbnf` to get you started (this already includes a significant part of the grammar for expressions, see also the solution to Tutorial 2). No Java coding is required for this coursework but you will probably find it convenient to work at the terminal in IntelliJ (any IntelliJ project configured to use the SBNF.jar library will suffice).

All three grammars **must** be LL(1). Assessment will be carried out by automated testing. Any grammar which is not LL(1) will immediately fail all the tests and so will be given a mark of zero. The SBNF library includes a tool to check if your grammar is LL(1), for example:

```
java LL1Check lp23s.sbnf
```

Note that the LL1Check program also reports FIRST, FOLLOW and choice-sets (which will be useful if your grammar is not LL(1) and so needs to be modified).

To help you test your grammar, test inputs are provided for each language. The test file names give a rough indication of their relative sizes (for example, the tests with names like `test_100_x.lp23` are roughly twice the size of those with names like `test_50_y.lp23`). Use the smaller tests first. Testing is done using the Parse program (see Tutorial 3). Note that each test is accompanied by a "mutated" version. This is a *negative* test input which has been deliberately altered to contain a syntax error. Parsing the mutant *should* result in a parse exception or lexical error. It is just as important for your grammars to reject invalid inputs as it is for them to accept valid inputs.

Comments: comments in the *LP23* languages start with `//` and extend to the end of the line. The SKIP token definition in the prototype grammar file effectively causes the lexer to treat comments as whitespace. *Do not modify or remove the* SKIP *token definition*.

Identifiers: all three languages use the same token type for identifiers (for variable names in *LP23s* and additionally for function names in *L23ef* and *LP23*). An identifier is a non-empty sequence of letters, digits, underscores and $-signs; the first character in an identifier cannot be a digit or a $-sign; if the first character is an underscore, there must be at least one additional character.

In the following language descriptions, wherever a *sequence* is specified, it is allowed to be an empty sequence (unless otherwise stated).

1. **[50 marks]** Language *LP23s*. An *LP23s* program starts with the keyword **`main`** followed by a sequence of statements enclosed between a pair of curly brackets. For example:

   ```
   main { x <- 10; printchar x; }
   ```

   There are five kinds of statement: printint-, princtchar-, assignment-, conditional- and loop-statements (in the above example the first statement is an assignment-statement and the second is a printchar-statement).

   A printint-statement starts with the keyword **`printint`**, followed by an expression, and is terminated by a semicolon. A printchar-statement is the same except it uses keyword **`printchar`**.

   An assignment statement uses the assignment operator **`<-`** with a variable name (an identifier) on the left and an expression on the right, and is terminated by a semicolon. For example:

   ```
   doobry <- 7 * 21;
   ```

   A conditional-statement starts with the keyword **`if`** followed by an expression (the condition), then the keyword **`do`**, then a sequence of statements (the true-branch), then the keyword **`eldo`**, then another sequence of statements (the false-branch), and is terminated by the keyword **`endif`**. For example:

   ```
   if x == y do printint 5; x <- x * 2; eldo endif
   ```

   (Note that the false-branch is empty in this example, but of course it doesn't have to be empty in general.)

   A loop-statement starts with the keyword **`repeat`**, followed by a sequence of statements, then the keyword **`until`**, followed by an expression, and is terminated by a semicolon. For example:

   ```
   repeat y <- y * 5; x <- !x; until x == 0;
   ```

   An expression is a simple expression, optionally followed by an operator clause. A simple expression is an integer literal, a variable name, a bracketed expression or a prefix operator followed by a simple expression. The prefix operators are **`-`** (unary minus) and **`!`** (logical negation). An operator clause is a binary operator followed by a simple expression. The binary operators are: **`*`**, **`/`**, **`-`**, **`+`**, **`<`**, **`<=`**, **`==`**, **`&&`**, **`||`**.

2. **[30 marks]** Language *LP23ef* extends *LP23s* by enhancing the syntax for conditional-statements and by adding function definitions and function calls.

   Conditional-statements can now have zero or more **elif** -branches after the initial true-branch, and the final **eldo**-branch becomes optional. Three examples:

```
if x < 0 do
   y <- x;
elif x < 1 do
   // nothing
elif x < 2 do
   y <- 4*x;
   printchar 32;
eldo
   y <- -1;
endif
```

```
if y do
   printint 77;
endif
```

```
if p && q do
   x <- y + 1;
elif q || r do
   y <- x - 1;
endif
```

   After the closing curly bracket of the main program body, a sequence of function definitions is now allowed. A function definition starts with the keyword **def**, followed by a function name (an identifier) followed by a formal-parameter list and then a function body. A formal-parameter list is a sequence of identifiers, separated by commas and enclosed in brackets. A function body is a sequence of statements enclosed between a pair of curly brackets. Two example function definitions:

```
def foo(x, y) { bar(); return bar()*z; }
def bar() { printint x; return foo(6, 23) + 1; }
```

   Function calls are now allowed, both as a new form of simple expression and as a new form of statement. A function call is a function name followed by an actual-parameter list. An actual-parameter list is a sequence of expressions, separated by commas and enclosed in brackets. When a function call is used as a statement, it must be terminated by a semicolon.

   One other new kind of statement is allowed: the return-statement. A return-statement starts with the keyword **return**, followed by an expression, and is terminated by a semicolon. Return-statements are allowed anywhere that other statements are allowed (not just in function bodies). It is also legal for a function body to not contain any return statements.

3. **[20 marks]** Language *LP23* extends *L23ef* by adding an increment operator and support for arrays. The increment operator is a prefix operator (**++**) and it can only be applied to an *assignable* expression (see below). Example uses of the increment operator:

```
repeat ++doobry; until 99 < doobry;
x <- ++arr[9] * 3;
```

Note that an increment expression can also be used as a statement (as it is in the first example above) in which case it must be terminated by a semicolon.

Arrays are created by malloc-expressions. A malloc-expression is the keyword **malloc** followed by an expression (specifying the array-size) enclosed in square brackets. A malloc-expression is a new kind of simple expression. Arrays are accessed and updated using a conventional square-brackets notation. To keep the grammar simple, only variable names (not arbitrary expressions) can appear before the square brackets in array expressions. Examples:

```
arr <- malloc[64 * z];
printint arr[2 + y];
arr[2 * y] <- 99;
```

Things that can appear on the left of an assignment-statement are said to be *assignable expressions* (or sometimes they are known as *L-expressions*). In *LP23s* and *LP23ef*, the only assignable expressions are variable names. In *LP23*, array expressions (like `arr[2 * y]` in the above example) are also assignable.