The Semantics of LPL

For the most part, the semantics of LPL will be familiar to programmers with basic knowledge of languages such as Java or Python. But there are some peculiarities and some key technical details that you will need to consider carefully when developing your compilers.

Dynamic Semantics (Run-Time Behaviours)

The Basic LPL Language (Part 1 of the coursework)

This section considers just the basic language, without methods or arrays. This is the version of LPL which is relevant for Part 1 of the coursework.

- All global variables are initialized to zero by default. (Note that in the basic language, all variables are global.)
- The only data type in the basic version of LPL is `int`. All expressions of type `int` evaluate to 32-bit twos-complement integers. Wherever a Boolean value would normally be expected, 0 is interpreted as false and all other values are interpreted as true. (The full version of the language includes array types.)
- The `print` and `println` statements have the same behaviour as Java's corresponding PrintStream methods for arguments of type `int`. Unlike Java, there is no overloading, these *only* take integer arguments. The `newline` statement has the same behaviour as the Java PrintStream method `println()` (no arguments).
- The `printch` statement interprets its argument as an 8-bit ASCII code (the three high-order bytes are discarded) and prints the corresponding character.
- The comparison operators (**<**, **<=**, **==**) and Boolean operators (**!**, **&&** and **||**) always return 0 (for false) or 1 (for true).
- Like C and Java, the binary Boolean operators have "short-cut" behaviours:
  - When the first argument of **&&** evaluates to false (0) the second argument is not evaluated at all, and 0 is returned immediately.
  - When the first argument of **||** evaluates to true (any value other than 0) the second argument is not evaluated at all, and 1 is returned immediately.
- Division by zero causes the program to halt immediately (this, conveniently, is exactly the behaviour of the SSM div instruction, so your compiler doesn't need to do anything clever).
- In all language constructs, sub-expressions are evaluated from left-to-right.
- In a switch-statement, each case is guarded by an integer constant. The switch expression is first evaluated and then cases are checked from top to bottom and only the statement for the first matching case (or the default statement, if no cases match) is executed. After this, the switch-statement exits (unlike "classic" switch-statements, there is no fall-through behaviour, so break statements are not required). For simplicity, there is no requirement that the case guards are disjoint but note that later cases which repeat earlier guards are redundant, since only the first matching case can ever be chosen.

The Full LPL Language (Part 2 of the coursework)

The full language includes methods and arrays.

- Global variables of array type are initialized to null. (In practice, you will implement null as 0. Any other choice would simply make the implementation unnecessarily complicated.)
- There are two kinds of method: *functions*, declared with the keyword fun, and *procedures*, declared with the keyword proc. Functions return values, procedures (like void methods in Java) do not. Function calls can be used as expressions, procedure calls cannot. Both procedure and function calls can be used as statements (when a function call is used as a statement its return value is discarded).
- Return statements in methods are optional. As in Java, there are two kinds of return statements: a *value*-return specifies a return value (these are used in functions); a *plain*-return does not (these are used in procedures). If a function exits without explicitly executing a return statement, it returns either 0 or null, according to its return type. Plain-return statements are also permitted in the main body of a program, where their effect is to immediately halt the program.
- As in Java, array types in LPL are *reference* types. Concretely, this means that the data stored in a variable of array type, or passed as a value to a method parameter of array type, is actually a reference to (ie the memory address of) the array data. All arrays are dynamically allocated in the heap (using **`new`**).
- The elements of a newly created array are all initialized to 0 or null, depending on their type (But, as mentioned above, in practice this just means that all array elements are initialized to 0, since you will implement null as 0.)
- Like Java, LPL does not have true multi-dimensional arrays. Instead it has the ability to create arrays of arrays. For example, `new int[100][]` creates an array of length 100 where each array element is of type `int[]` and is initialised to null. Note that in Java it is possible to write `new int[7][50]`, which would create an array of length 7 and initialize its elements by simultaneously creating 7 arrays of length 50; this is not possible in LPL. (But, of course, a loop can be written in LPL to achieve this effect).
- Any attempt to access (either read or write) an element of a null array reference simply causes the program to halt immediately. (This is to simplify the implementation. For a realistic language some kind of error reporting/exception handling mechanism would be needed.)
- If an LPL program attempts to access an array element out of bounds, the behaviour is *undefined*. (This means that your implementation is free to simply ignore this possibility and assume that it doesn't happen. Again, this is to simplify the implementation.)

Static Semantics (Compile-Time Constraints)

Well-formed Programs and Lexical Scoping

- Programs which include multiple declarations for the same global variable, or which include uses of undeclared variables, are considered badly formed and should be rejected by the compiler.

- Programs which include multiple declarations for the same method name, or which include uses of undeclared methods, are considered badly formed and should be rejected by the compiler.
- Within each method declaration, the formal parameter names and the local variable names must all be distinct (the two sets of names must not overlap and the same name cannot appear twice within either the formal parameter list or the local variable list). Any program which contains a method declaration violating these constraints is considered badly formed and should be rejected by the compiler.
- It *is* permitted for a global variable name to be re-declared as either a formal parameter or local variable within a method. In this case, the inner declaration is the one which is applicable within the method body. The inner declaration is said to create a "hole" in the scope of the outer declaration. Note that the applicable declaration for any use of a variable name is determined solely by where the use occurs in the text of the program, and so is statically determined (fixed at compile-time). This is called *lexical scoping*.

The LPL Type System

LPL is a statically typed language, which means that a complete compiler should reject any programs which are not well-typed. However, when implementing the compile methods, you can assume that the program being compiled has already been type-checked. (If you are asked to implement type-checking, this will be as a separate phase of the compiler, not within the AST compile methods.) Making this assumption significantly simplifies the implementation task (you don't have to worry about how to generate code for a nonsensical expression like `7[new int[null]]`, for example).

(Note that the type system is not relevant for the basic language: since it only has a single type, no type mismatches can occur.)

- Integer literals have type `int`.
- Applications of the binary arithmetic operators (this includes the comparison and Boolean operators) are well-typed if their operands have type `int`, in which case the resulting expression also has type `int`. The same applies to the unary Boolean-negation operator.
- Assignment statements are well-typed if the type of the left-hand-side matches the type of the right-hand-side.
- Statements of the form `print` *e*, `println` *e*, and `printch` *e* are well-typed if *e* has type `int`.
- If-statements, switch-statements and while-statements are well-typed if the expression part has type `int` and all sub-statements are well-typed.
- A method call *m*(e_1,…,e_n) is well-typed if *m* is declared with exactly *n* parameters, each e_i is well-typed, and its type matches the declared type of the corresponding formal parameter of *m*.
- The type of a well-typed function call *f*(e_1,…,e_n) is the declared return type of *f*.
- A value-return statement `return` *`e;`* is well-typed if the type of *e* matches the return type of the enclosing function declaration.
- A well-typed array-creation expression of the form `new int[`*e*`]` has type `int[]`. A well-typed array-creation expression of the form `new int[`*e*`][]` has type `int[][]`. A well-typed array-

creation expression of the form `new int[`*e*`][][]` has type `int[][][]`. Etc, etc. In all cases, the array-creation expression is well-typed if *e* has type `int`.

- If e_0 has an array type *T*`[]…[]` (*n* sets of square brackets) and e_1 … e_n each have type `int`, then e_0`[`e_1`]…[`e_n`]` has type *T*.
- An expression of the form *e*`.length` is well-typed if *e* has an array type, in which case the whole expression has type `int`.
- The type of `null` is the special array type `<any>[]` (this is not a type which can be explicitly written in an LPL program).
- The special array type `<any>[]` matches all array types. Otherwise, types only match if they are identical.
- A program is well-typed if all its component parts are well-typed.