ADL

The easy App Development Language

1 Contents

2 Features

2.1 Aims

ADL aims to be a simple, moderately easy-to-learn interpreted programming language that makes developing games and graphical apps much simpler. It implements a `LL(0)` parser and as such uses NO look ahead when parsing a desired input.

2.2 Passes

It makes a total of four passes over the file when running the code:

1. Reading the file into a `String` while via a LibGDX `FileHandle`.
2. Tokenizing it into a `Queue` of `Token`s.
3. Grouping it into a `Queue` of `Phrase`s.
4. Parsing each `Phrase` one-by-one.

I intend to (possibly) add in one more pass or set of passes to check for errors in the code so avoid the user having to get to that specific point when running the code before it crashes. For example, the C# parser uses about a dozen syntax checking passes.

2.3 Variables and Scopes

Variables can be currently of two types – `text` and `int`. String literals are delimited by double quote marks: “”. Integers are suffixed with ‘i’ to denote integers (similar to ‘f’ and ‘L’ in Java). I believe that having numbers implied that

they are just integers with no suffix in Java can be illogical, and so the 'i' suffix in ADL was created. It can also give a hand to the number processor so that it knows exactly when a number ends.

Variables in ADL, no matter where they are instantiated, are global variables. Variables can be manually deleted via the `delete` keyword. There is no garbage collector in ADL.

3 Syntax

3.1 Phrases

The ADL interpreter uses phrases to parse files with speed. The concept of a phrase is that tokens will be concatenated into groups (the phrases) that perform given tasks. For example, the `PRINT_VAR` phrase combines tokens into a phrase with structure as follows (pseudocode):

```
Phrase(
     PRINT_VAR,
     Token("print"), Token(<varName>), Token(semicolon))
```

The upside to phrases as opposed to parsing individual tokens and parsing them into runnable code at runtime is that phrases can be pre-compiled; tokens are concatenated into phrases at 'compile time' and so some errors can be checked for before the program is even run. It is also the trade-off between a slight start-up time overhead instead of slower-running code (the former I much prefer).