Dilemmas: Design decisions, challenges faced and chosen solutions

Submitted by:

Daniel Cohen, Shani Haker, Michal Ben Haim

Design decisions:

1. Ease of use as a primary concept of syntax building.
2. Identifying all variable types through the same keyword.
3. Differentiating interactive mode and script mode by file reading.
4. Basic syntax to assign variables will have an opening 'VAR' keyword, identifier and '=' before assigning a value.
5. Picking reserved keywords [shown in user guide]

Challenges:

1. **How should we handle consecutive symbols that each have meanings – for example, '=' (variable assignment) and '==' (equal to)?**
   Answer: We decided on expanding our tokens to support both symbol meanings to enable a complete language. Thus, when approaching a token with two similar consecutive symbols, we invoke a function checking for the both them which also alternates the type of the token when necessary (as the example above, say '==' is read -> at first, the token will be recognized as variable assignment symbol '=', but after invoking the secondary check, and ensuring the second character is another '=' -> the function will alternate the token type into 'equal to'.)
2. **Implementing 'bool' variables: using 'BOOL' or 'VAR' to identify true/false values?**
   Answer: Eventually, we decided on using the same keyword for both numerical and binary (Boolean) values for ease of language use (same keyword for all variable types), plus keeping the language as broad as possible by keeping the number of keywords to the bare minimum.
3. **implementation of reading code blocks- should files be used?**
   Answer: Eventually, we indeed chose file reading as the appropriate solution for reading larger code blocks. When trying to consider other solutions, they didn't seem as effective or didn't offer the same ease of use which we chose as a leading concept.
4. **ambiguous ! meaning - logical not for Boolean expression and mathematical not (representing 'not equals') - how should we differ its implementation it to support both ways?**

Answer: To face this challenge, we decided to add a new token option – a logical not. Plus, we decided on changing the syntax – any attempt to use a logical not must be followed by parentheses, thus they play as separating symbols between a logical not [!(a > b)] and mathematical not [a != b]

5. **Is logical not a Unary or Binary Operator?**
   Answer: We tested both ways through experimenting with the code. At first, we implemented a logical not as a binary operator – causing issues with finding appropriate values for both expected arguments for a complete binary operation. Thus, after logical errors were raised (incorrected computed results for logical expressions), we decided to implement logical not as a unary operator that will operate on a given expression result.
6. **How can we avoid variables names with saved keywords like 'True' or 'False'?**
   Answer: Since we weren't required to assign words as variable values, the task of assigning a string-type value to a variable was challenging, since the only token we allocated for reading letters consequently was 'Identifier' - it immediately raised the issues of having an illegal identifier. We then decided on expanding our available tokens, and checking the list of tokens read thus far, if an illegal keyword came before the list of tokens had at least two tokens, it would through an exception. Markling it as an attempt to use 'TRUE' or 'FALSE' as variable identifiers.
7. **Implementing '.lambda' files – how can we handle reading a file without causing errors in the one-by-line mode?**
   Answer: We primarily decided the function of reading files, without consideration to the required suffix at first, must be done through a different command to avoid any collision with the already written interactive mode. To do that, we decided on using a regular expression which looks for a specific pattern we chose to invoke file reading [-F filename.lambda]. From there on, we used other functions like 'spilt' to retrieve the extensions and eliminate mismatching results.