Lab session 3b: compiler construction
Semantic analysis for Mini Pascal

The second part of the 3rd lab session of the course *Compiler Construction* consists of extending the parser for MiniPas with semantic analysis. Since the language MiniPas is that small, there is no need to build an AST (Abstract Syntax Tree). It is sufficient to augment the parser with actions that perform the semantic analysis. On detection of a semantic error, an informative error should be produced, and the 'compiler' should abort.

Note that the specification of this lab exercise is deliberately kept vague. Given the grammar of MiniPas, you should be able to infer which checks the semantic analysis can/should perform. That is part of designing/constructing a compiler for some programming language.

Note 1: In MiniPas (like in real Pascal), there is no `return` statement for functions. Values are returned by assigning a value to the 'function name'. For example, the following trivial function always returns the integer 42:

```
FUNCTION return42(n : integer) : integer;
BEGIN
  return42 := 0;
  WHILE return42 < 42 DO
    return42 := return42 + 1
END;
```

Note 2: In MiniPas (like in Pascal, C), global variables are declared outside functions/procedures (at the top of the program). Local variables are declared within a function/procedure, and can 'shadow' global variables. Note that in MiniPas it is not possible to shadow a local variable for a second time (i.e. there exist only two block levels).

Note 3: In MiniPas there are only two standard types: **integer** and **real**. All binary arithmetic operators applied to an **integer** and a **real** return a **real** (i.e. the **integer** is promoted).

Note 4: Implicitly, there is also a 'stripped' boolean type because there are tests/guards for the **if-then-else** and **while**-statements. However, you cannot assign boolean values to a variable, and you can also not make compound boolean expressions (i.e. there is no **and**, **or**, and **not**). Keep remembering that miniPas is a stripped language designed as an exercise in compiler construction. It is not (and never will be) a full-fledged programming language.

Note 5: It is allowed to assign an **integer** value to a variable of type **real**. Moreover, it is allowed to assign a **real** to an **integer**, in which case the same happens as in C (truncation).

Note 6: Unfortunately there is an error in the grammar. Please change the grammar rule for `Factor` as follows:

```
Factor   -->    id
          |     id ( expression_list )
          |     id [ expression ]
          |     num
          |     ( expression )
```

The difference with the original rule is two-fold: the rule `Factor -> not Factor` has been removed, and the rule `Factor -> id [ expression ]` is introduced. The latter rule allows the usage of indexing in arrays, where `expression` must be an integer valued expression in the right index range.

Handing in your work: The deadline for this lab is **Wednesday Jan. 10th, 23:59**! For this lab session, you only need to hand in a tar file containing all files (including a Makfile) necessary to compile your project. A report is not needed. However, we do require a `README` file explaining a list of checks that your implementation of semantic analysis performs. Themis will not compile your code, and will always accept it. The TAs will test your program by hand. It is simply not possible to test semantic analysis by a judge, simply because the reported error messages vary too much. We will post several examples of useful test cases (useful for debugging your analyzer) on Nestor.